ESCUELA TECNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Grado en Ingeniería en

Sistemas Audiovisuales y Multimedia

Trabajo Fin de Grado

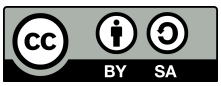
Study of Deep Learning Neural Networks with Keras

Autor: David Pascual Hernández

Tutor: José María Cañas Plaza

Co-tutor: Inmaculada Mora Jiménez

Curso académico 2016/2017



Agradecimientos

Resumen

Summary

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and motivation

1.2 Objectives

1.3 Methodology

1.4 Project structure

1.5 Gantt chart

Chapter 2

Framework

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

2.1 Keras

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

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

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**. An example of a sequential model built with Keras can be seen in the following chapter

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

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

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**. It's main arguments are:

- ***loss***: name of the **cost function** employed to check the difference between the predicted labels and the real ones. 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 [2] 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), mean absolute error and hinge are also provided by Keras.

- ***optimizer***: name of the optimizer that will update the weights values during training in order to minimize the loss function. The chosen algorithm for this task is **ADADELTA**. This optimizer is an extension of the **gradient descent** optimization method that has the particularity of adapting the learning rate during training with no need of manual tuning. According to the paper in which it is defined [3], it follows the next algorithm:

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ

Require: Initial parameter x_1

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

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

3: Compute gradient: g_t

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

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

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

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

8: **end for**

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

- ***metrics***: name of the functions that must be evaluated during training and testing. The only one that is going to be computed with Keras through this project, besides the loss function, which is automatically computed, is **accuracy**. It is defined as the proportion of examples for which the model produces the correct output [4]. Other measurements about the performance of the model are obtained with the **Scikit-learn** library (see Section 2.5).

.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. It's important to note that this new validation data won't be used for training anymore. *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.

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

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

.save() It stores the model into a **HDF5 file** (see Section 2.4), 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

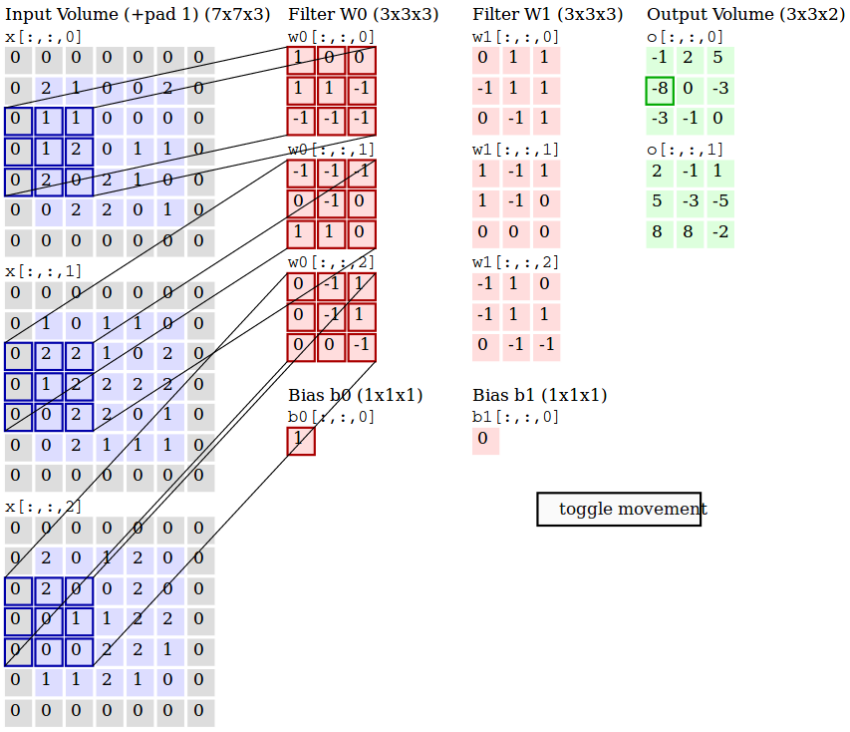
As it has been said before, the models are usually built as a **stack of layers**. These layers are added to the model using the ***.add()* method**, inside of which the kind of layer is declared and its particular parameters are set. Several kinds of layers are available, but only the ones that have 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 **convolutional neural network (CNN)**. It is formed by a fixed number of **filters/kernels** with a fixed size. These filters are convolved along the input image, generating each one a **feature or activation map** which will tell us to what extent the feature learned by that particular filter is present in the input image [5]. It's important to note that the depth of the filter will be equal to the number of channels of the input, which implies that each filter will generate just one activation map, instead of generating one for each channel.

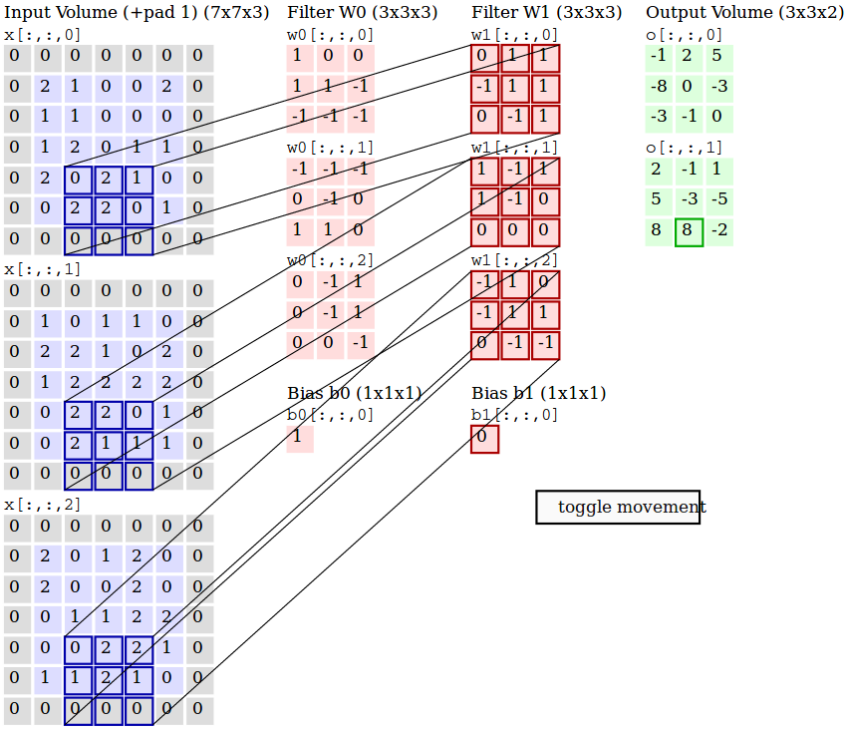
Keras provides different kinds of convolutional layers depending on the input dimensions: *Conv1D*, *Conv2D* and *Conv3D*. These are the main arguments required by Keras to define a convolutional layer:

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

Figure 2.1 shows how the convolutional layers work. In Figure 2.1a, the filter w_0 (3x3x3) is convolved with the input image (5x5x3). As padding is set to 1 pixel around the input and the stride is equal to 2, the operation will return a 3x3 activation map. The same procedure is followed in Figure 2.1b with the filter w_1. It generates another 3x3 activation map, ending up with a 3x3x2 output. These images have been extracted from [5]



(a)



(b)

Figure 2.1: Convolutional layer.

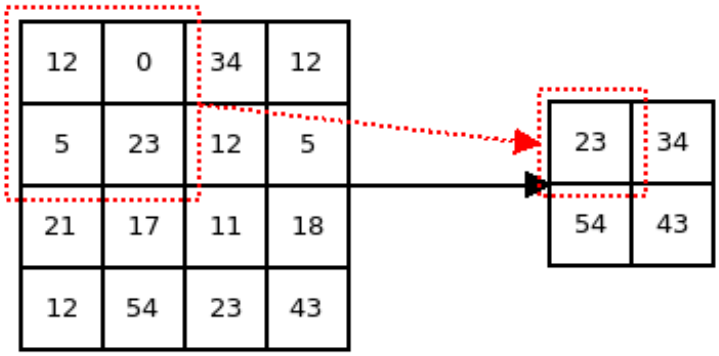


Figure 2.2: Example of a max. pooling operation.

Pooling layer It shifts a window of a certain size along the input image applying an operation (mean or maximum) that will return a ***downsampled* version** of it, reducing the computational cost and avoiding overfitting [6]. Figure 2.2 shows how the pooling operation is applied.

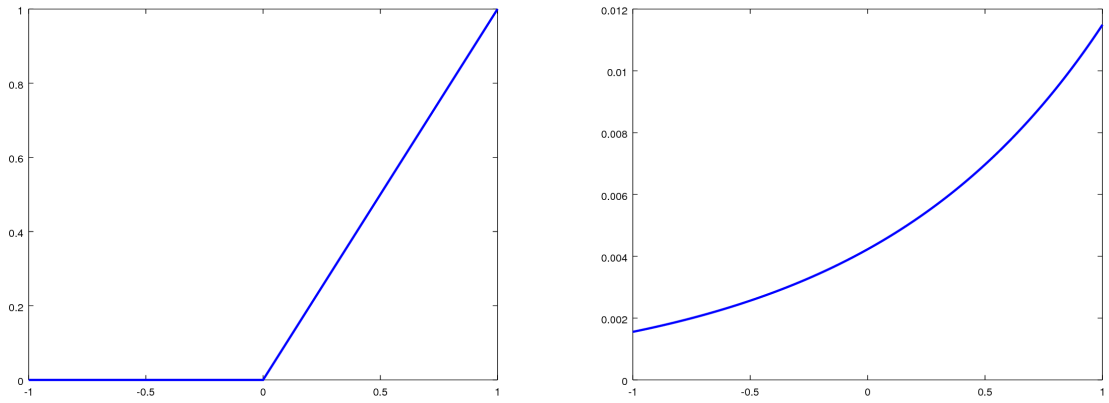
Depending on the dimensions of the input and the operation applied, Keras provides several pooling layers: *MaxPooling1D*, *MaxPooling2D*, *MaxPooling3D*, *AveragePooling1D*... The main arguments required by Keras to define these layers are:

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

Dense layer Fully-connected layers in Keras are defined as *Dense layers*. In a **fully-connected layer**, every neuron is connected to every activation (i.e. output) of the previous one [5]. The main argument of this layer is:

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

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, rectified linear unit (ReLU) and softmax. These are the ones that have been used during the development of this project:



(a) ReLU activation function.

(b) Softmax activation function.

Figure 2.3

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

$$g(z) = \max(0, z) \tag{2.2}$$

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

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\Sigma_j \exp(z_j)} \quad \text{for } j = 1, ..., N \tag{2.3}$$

These equations and definitions have been extracted from [4]. Figure 2.3 shows these activation functions plotted in the interval $[-1, 1]$.

Flatten layer It *flattens* the input. For instance, it converts the activation maps returned by the convolutional layers into a **vector of 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 [7]. It can also be understood as a technique that "trains

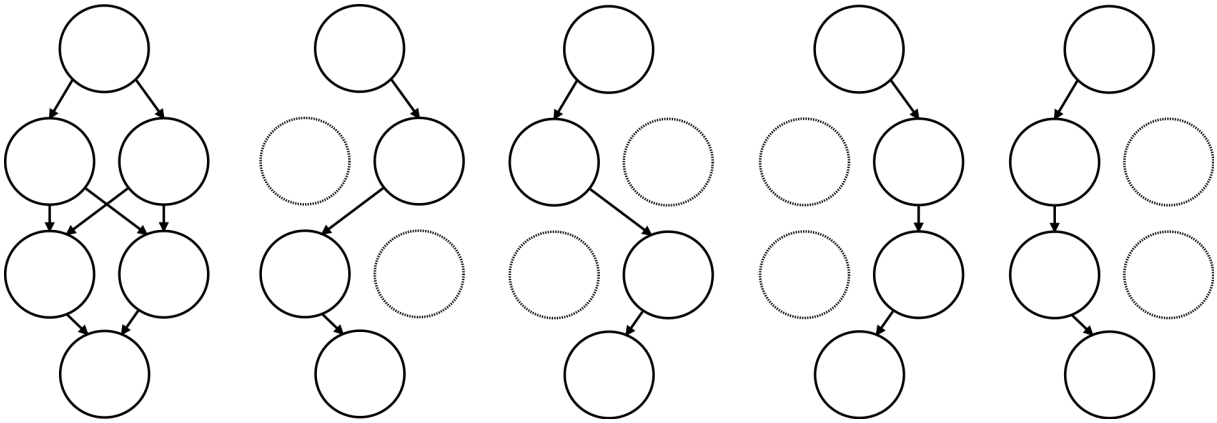


Figure 2.4: Subnetworks generated when using dropout.

an ensemble consisting of all subnetworks that can be structured by removing non-output units from an underlying base network" [4], as it can be seen in Figure 2.4.

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

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

2.1.3 Callbacks

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

- ***.History()***: it is automatically applied to every Keras model and is returned by the *.fit()* method. 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 quits 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 machine learning, besides adapting the images and extracting features before training that can improve the model performance (i.e. edge detection), it can be used to avoid **overfitting** through data augmentation. **Data augmentation** [8] consists in taking the samples that the dataset already contains and applying transformations to them, generating new samples that may be closer to real world and, in any case, enlarging the dataset with new data.

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

2.1.5 Utils

Keras include a module for multiple supplementary tasks called ***Utils***. The most important functionality for the project provided by this module is the ***.HDF5Matrix()* method**. It reads the **HDF5 datasets** (see Section 2.4), which are going to be used as inputs to the neural networks.

2.2 JdeRobot

JdeRobot is an open source middleware for robotics and computer vision [9]. It has been designed to simplify the software development within these fields. It's mostly written in C++ language and it's structured like a collection of components (tools and drivers)

that communicate to each other through **ICE interfaces** [4]. It is also compatible with **ROS** [5], which allows the interoperation of ROS nodes and JdeRobot components. This flexibility makes it very useful for our application. Its ***cameraserver* driver** is going to be employed to capture images from different video sources.

cameraserver

According to JdeRobot documentation [9], this driver can serve both real cameras and video files. It communicates with other components thanks to the ***Camera* interface**.

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

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

2.3 DroidCam

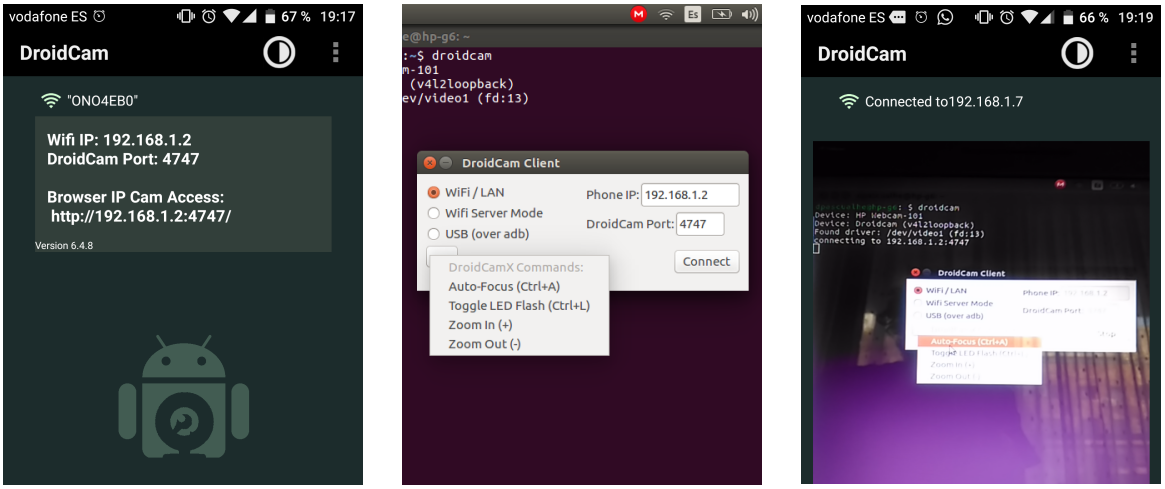
On one hand, **DroidCam** is an application for Android which serves the images captured with a **smartphone camera** [10]. On the other hand, it is a client for Linux which receives the video stream served by Android and makes it accessible for the computer as a **v4l2** [6] **device driver**. The Linux client can be connected to the phone camera over a USB cable or a WiFi network and allows the user to control camera flash, auto-focus and zoom. DroidCam provides the address 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 any browser.

An example of usage can be seen in Figure 2.5. First, the Android app is opened. It shows the address where the video will be served (see Figure 2.5a). Then, the address is set in the Linux client (see Figure 2.5b). Finally, when the *Connect* button is pressed, the connection is established (see Figure 2.5c)

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

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

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



(a) Android server. (b) Linux Client. (c) Connection established.

Figure 2.5

2.4 HDF5

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

According to HDF5 documentation [11], 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** [7] library for Python has been employed.

2.5 Scikit-learn

Scikit-learn is a machine learning library that includes a wide variety of algorithms for clustering, regression and classification [12]. It can be used at every stage of the machine learning 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

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

obtained from [13]):

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

$$\text{precision} = \frac{true_{positives}}{true_{positives} + false_{positives}} \tag{2.4}$$

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

$$\text{recall} = \frac{true_{positives}}{true_{positives} + false_{negatives}} \tag{2.5}$$

- **Confusion matrix**: a matrix where the element i, j represents the number of samples that belongs to the group i but has been classified as belonging to group j. True predictions can be found in the diagonal of the matrix, where $i = j$. An example of a confusion matrix constructed with Scikit-learn and displayed with Octave (see Section 2.6) can be found in Figure 2.6.

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.

2.6 Octave

GNU Octave [14] is a scientific programming language compatible with **Matlab**. It provides powerful tools for plotting, which have been used to visualize the data collected with Scikit-learn about the performance of the models. An example of Octave usage can be seen in Figure 2.6.

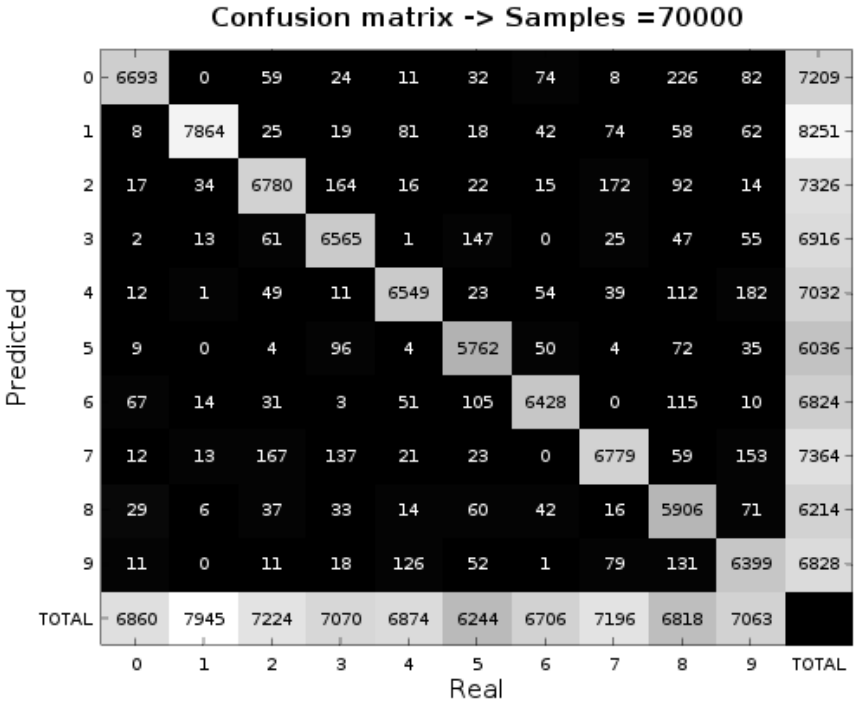


Figure 2.6: Example of a confusion matrix visualization using Octave.

Chapter 3

Digit classifier

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

- A **Keras model** (see Section 2.1.1), which classifies the images.
- A **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 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 [1] of an example provided by Keras [2] will be analyzed in the following sections. In this example, a CNN is trained and tested with the **MNIST database** of handwritten digits (see Section 4.1).

3.1.1 Adapting data

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

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

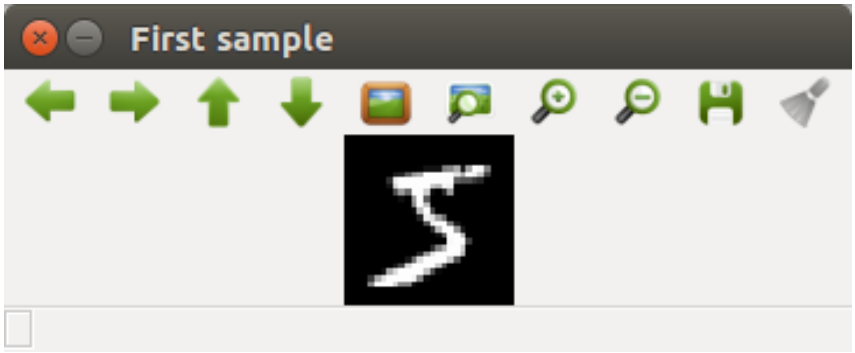


Figure 3.1: First sample of the MNIST database.

be seen 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 is 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 can be seen 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 is 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 2.2) 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 2.3).

The resulting architecture and the shape of the data as it goes through every layer can be seen 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:

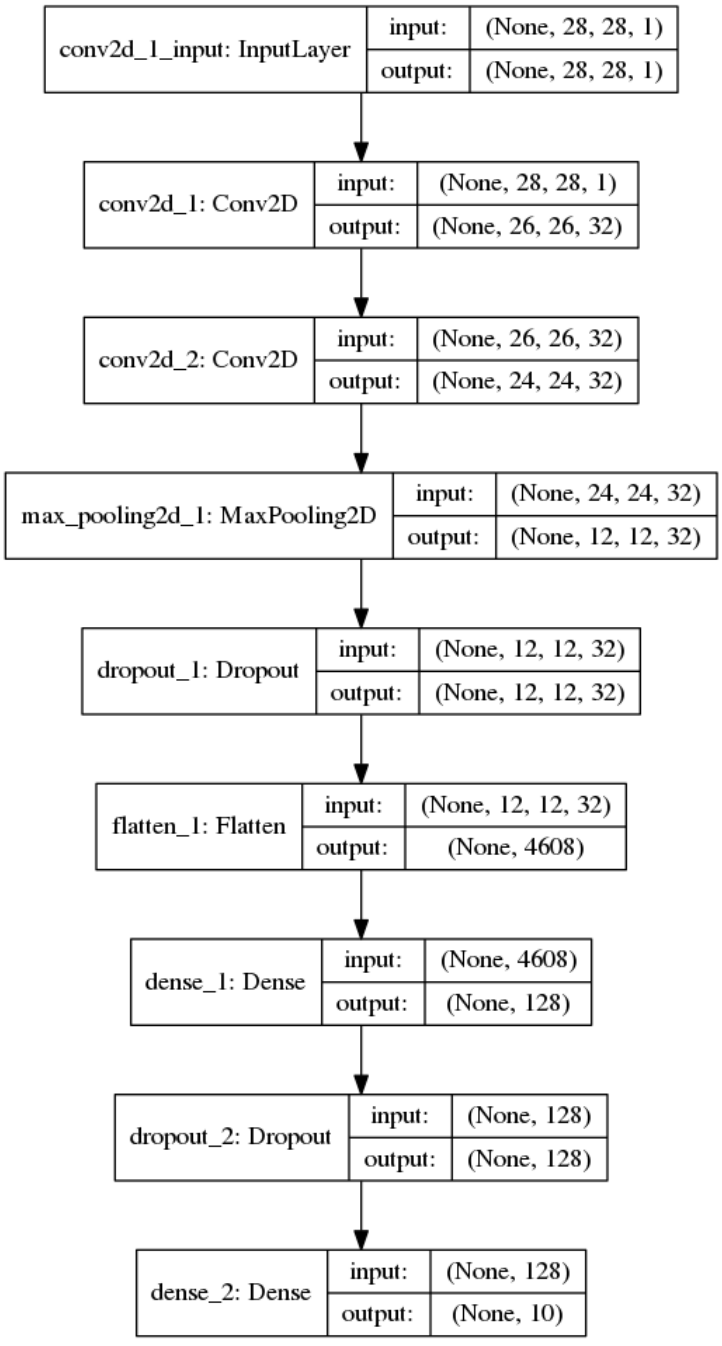


Figure 3.2: Diagram of a Keras sequential model.

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

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** (see Algorithm 1). 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 2.4). 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 can be seen 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 Component

Once the CNN has been trained and the resultant model is saved, the next milestone is to integrate it into a 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*** [3] and it is based on Nuria Oyaga code [4]. It relies on *Camera* and *GUI* classes. Figure 3.3 shows the program running.

3.2.1 *Camera* class

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

- **Acquisition**. The images are served by the JdeRobot component ***cameraserver*** (see Section 2.2). Depending on how its configuration file (*cameraserver.cfg*) has been set, the images can come from different kinds of video streams. Connection

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

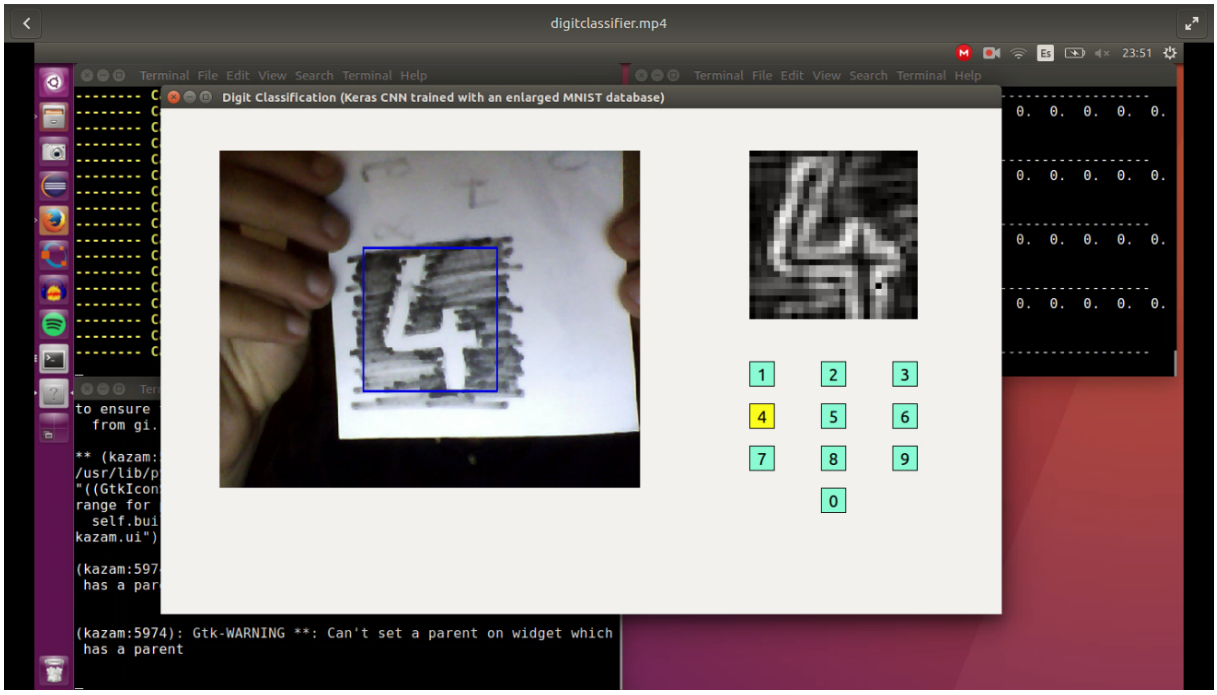


Figure 3.3: Example of *digitclassifier.py* execution.

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 can be seen in the code below.

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

In order to establish a connection with smartphone cameras, the **DroidCam application** for Android has been used (see Section 2.3). As this application turns the video stream provided by the smartphone into a **v4l2** [6] **device driver**, the video stream will be listed as another webcam and the number of device must be set in the *cameraserver* configuration file.

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

- **Preprocessing**. As the images can be captured with different devices, the

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

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 [15]. 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 [16], 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.
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]
```

3.2.2 *GUI* class

***GUI* class** [8] displays the original image, the processed image and the result of the classification, as it can be seen in Figure 3.3. It has been built employing the **pyQt package** [9] [10]. It is based in Nuria Oyaga code [11], but it has been updated from Qt4 to Qt5 thanks to the information provided by PyQT documentation [17].

3.2.3 Threads

In order to capture images and update the GUI concurrently, the ***threading* module** [18], 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
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. '''
```

[8] https://git.io/vH0YK
[9] https://pypi.python.org/pypi/PyQt4
[10] https://pypi.python.org/pypi/PyQt5
[11] https://git.io/vH0mx

```
  while(True):
     start_time = datetime.now()
     self.cam.update()
     end_time = datetime.now()

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

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

This code, as well as the one corresponding to the ***ThreadGUI* class**, can be accessed in GitHub [12] [13].

3.2.4 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 can be seen 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
```

[12] https://git.io/vH0lY
[13] https://git.io/vH0lW

```
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

Benchmark

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 **benchmark** has been developed. In this chapter, the datasets employed to train the models, as well as the tools developed 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 (Modified National Institute of Standards and Technology database) is a database of **handwritten digits** formed by a training set, which contains 60000 samples, and a test set, containing 10000 samples [19]. It's a *remixed* and reduced version of the original **NIST datasets** [1]. MNIST is a well-known benchmark for all kinds of machine learning 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

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

Figure 4.1: Samples extracted from the MNIST database.

useful for testing machine learning algorithms, it's not enough to train a model that aims to solve a **real-world task**, because the images are almost noiseless and share similar orientation, position, size and intensity levels.

4.1.2 Gradient images

The first issue with MNIST database that must be addressed is that the grayscale images that it contains share similar intensity levels: a white digit over a black background. In real world, the digits can be found written in several colors over different backgrounds and the datasets must resemble every possible combination. In order to achieve that generalization, 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 [16], convolving the image with the following kernels to highlight horizontal and

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

vertical edges, respectively:

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

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 [3] 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

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

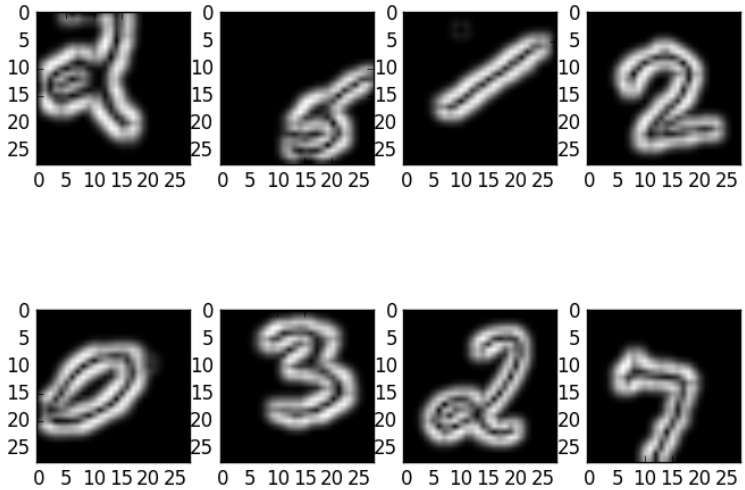


Figure 4.2: Samples generated with Keras from MNIST database.

this purpose, Keras includes noise layers such as the **GaussianNoise layer**, which adds Gaussian noise with a standard deviation distribution defined by the user. It's important to note that Keras treat noise layers as regularization methods that are only active during training time to avoid overfitting. In order to add noise to the generated samples, a GaussianNoise layer was established as the **input layer** of the model.

Handmade augmented datasets

The alternative to real-time data augmentation with Keras is building **our own datasets** applying the previously mentioned transformations to the images. My mate Nuria Oyaga has build 5 new databases with two sets each one: training and validation [4]. These are the new databases:

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

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

- **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 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).

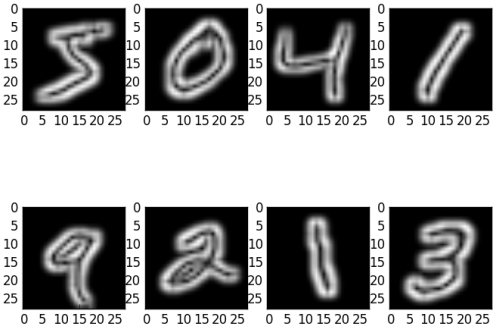
In Figure 4.3, the first samples of every handmade dataset can be seen.

From LMDB to HDF5

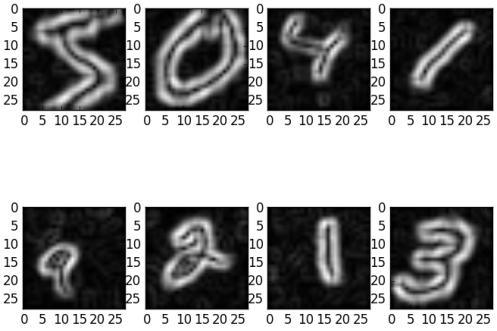
These databases were initially built to feed a **Caffe** [20] neural network. That's why they were saved as **LMDB files** [5]. In order to make it easier to feed the Keras model, the LMDB databases have been converted into **HDF5 files** (see Section 2.4). For this conversion, the script ***datasetconversion.py*** [6] has been written.

- **Reading the LMDB database**. The LMDB library for Python [7] was employed to open the database, initialize a cursor and iterate over each key-value pair in the database. In addition, Google's Protocol Buffers [8], a.k.a. Protobuf, was used to parse the data that was 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" [21]. Here can be seen the *.proto* file that defines the data structure used by Caffe to store the MNIST database, as obtained from [22]:

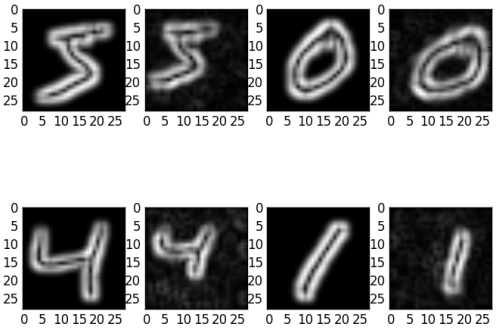
[5] `http://www.lmdb.tech/doc/`
[6] `https://git.io/vHWTe`
[7] `https://lmdb.readthedocs.io/en/release/#`
[8] `https://developers.google.com/protocol-buffers/`



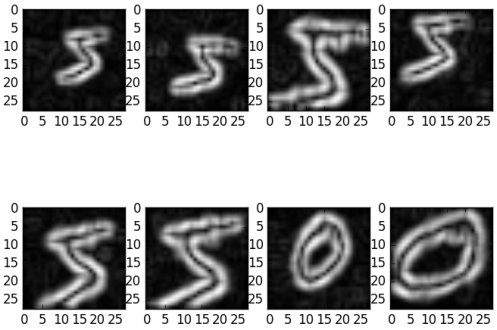
(a) Sobel dataset.



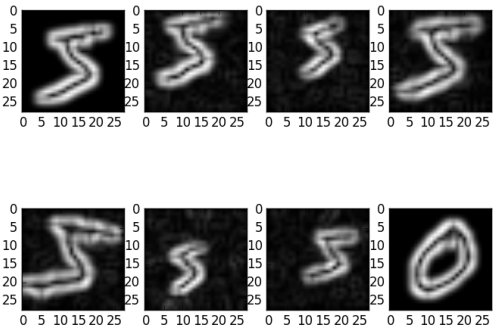
(b) 0-1 dataset.



(c) 1-1 dataset.



(d) 0-6 dataset.



(e) 1-6 dataset.

Figure 4.3

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

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

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

x = []
y = []
nb_samples = 0

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

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

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

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

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

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

- **Writing the HDF5 files**. After extracting the data, it was stored in a HDF5 file. Thanks to the **h5py library** [9] for Python, a HDF5 file with two *datasets* (label and data) was created. 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 go for the **handmade datasets**. While real-time data augmentation is really

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

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

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** [10] is totally **independent** from Keras. It calls functions that measure the performance of the model during **test and/or learning 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 2.5 and Section 2.1.1. Only log loss requires a probability distribution to be calculated. When calling the other functions, the predicted labels must be passed as an argument. The **predicted labels** are obtained as the indices of the maximum values in the probability distributions provided by the user.

- **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** [11] has been used. It provides the ***.savemat()*** method that saves Python dictionaries into Matlab ***.mat* files**, which are also compatible with Octave (see Section 2.6).

[10] `https://git.io/vHP47`

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

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)

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** (loss and accuracy) obtained after every epoch. It's interesting to face these validation results with the ones obtained after every batch during training.

LearningCurves [12] 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')))
```

[12] https://git.io/vHP4N

```
  self.accuracy.append(float(logs.get('acc')))
```

4.2.2 Octave function

Now that all the data has been collected, it has to be properly displayed. The function ***benchmark.m*** [13] 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 they can be seen in Figure 4.4. Additionally, it prints them to the **standard output**.

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 [5]. 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.

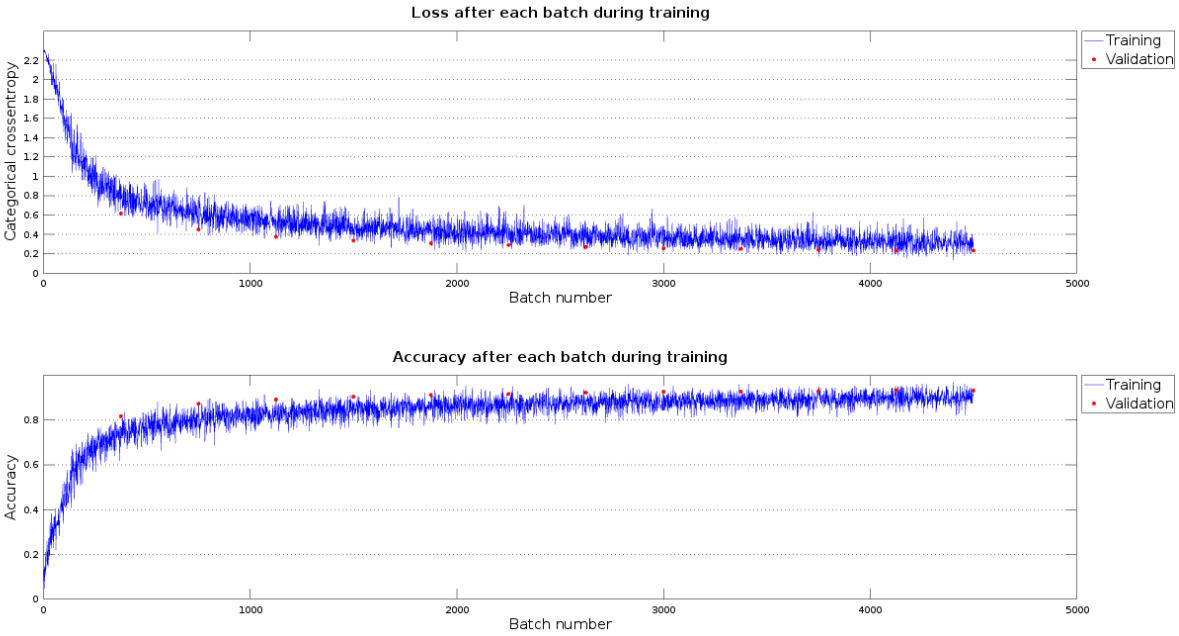
In order to understand how the Keras model is learning to classify the digits, the class ***layer_visualization.py*** [14] has been written. This class allows the user to display the filters that are learned in every convolutional layer of the model and their resulting activation maps.

4.3.1 Filters

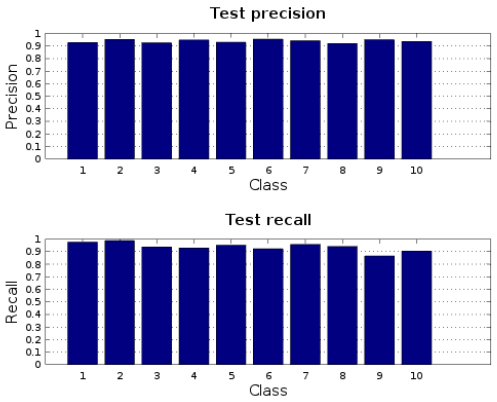
Keras provides a list containing every layer object in the model within its attribute *model.layers*. Layer objects properties can be accessed thanks to the *layer.get_config()* method. Since convolutional layers in Keras are named with the prefix *conv2d*, we iterate the names of the layers looking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

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

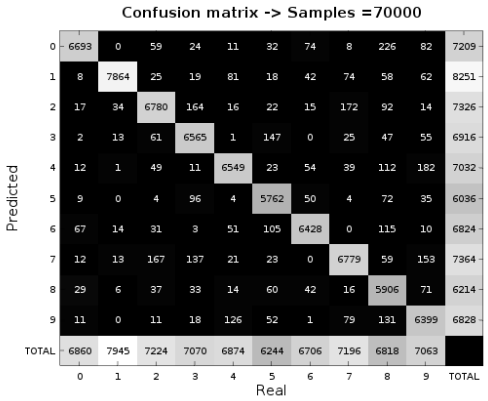
[13] https://git.io/vHPBt
[14] https://git.io/vH94o



(a) Learning curves.



(b) Precision and recall.



(c) Confusion matrix.

Figure 4.4

```
...
```

Once the convolutional layer has been found, accessing the filters is as easy as calling the *layer.get_weigths()* method. Besides that, the weights are reshaped to improve readability. The code that performs this process 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 with the *plot_data()* method, which has been written employing the **Matplotlib library** [23] for Python. Additionally, the shape and the maximum and minimum values of the weights are printed to the standard output. An example of how the filters are plotted can be seen in Figure 5.1

4.3.2 Activation maps

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

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

The activation maps are plotted using the same method than before, *plot_data()*, and some information about their shape and values is also printed to the standard output. An example of how the activation maps are plotted can be seen in Figure 5.3.

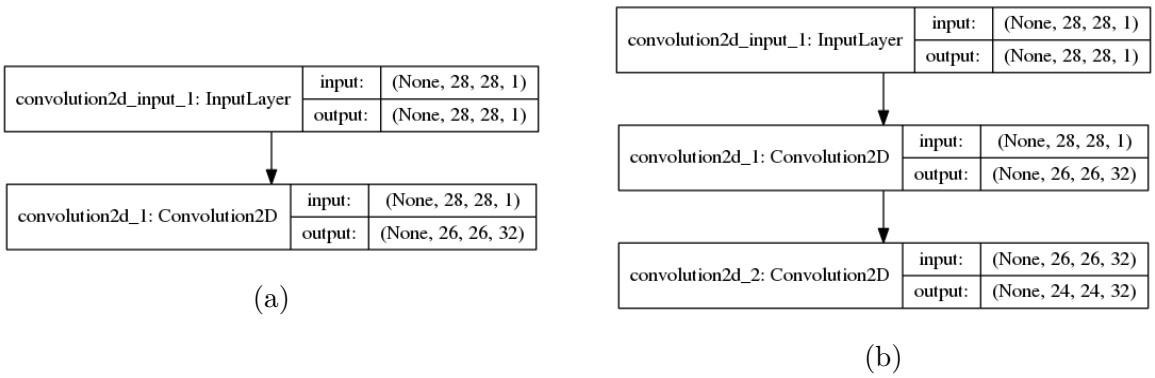


Figure 4.5: Truncated versions of the model.

Chapter 5

Evaluation

The CNN analyzed in Section **??** 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 Section **??** will be employed to evaluate the results achieved by each CNN. 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 the following sections belong to the convolutional layers of the *0-1; Patience=2* model that can be found in Section **??**. 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.

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.

In Figure 5.1, 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 as folllows:

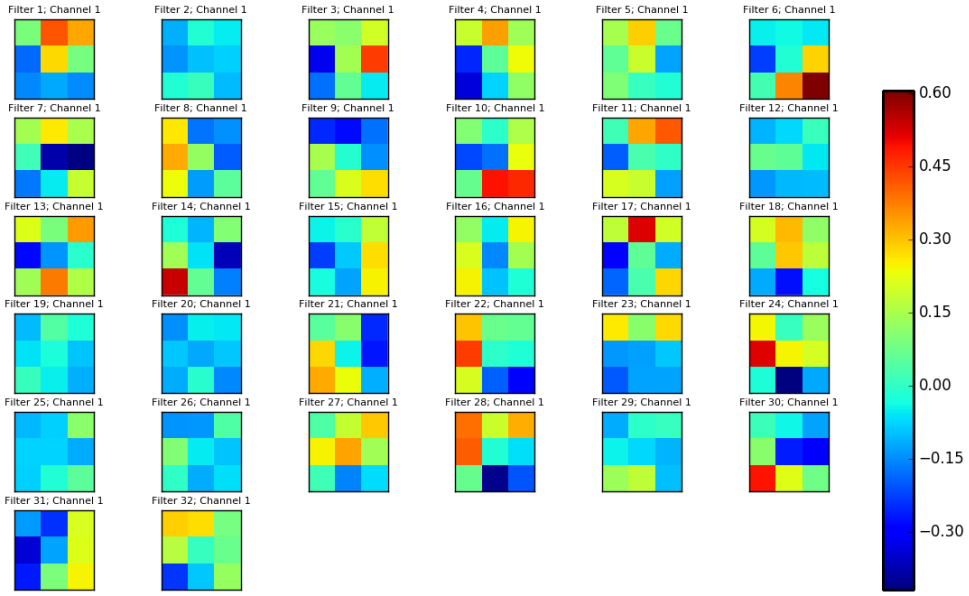


Figure 5.1: Filters of the first convolutional layer.

- **Horizontal edges**: filters 7, 9 and 23.
- **Vertical edges**: filters 8, 15 and 31.

The filters in the **second convolutional layer** have been displayed as well. The shape of the weights in this layer is (32, 3, 3, 32), which means that there are 32 filters with size 3x3. This time, its depth is 32, since there is one channel per activation map generated by the previous layer. As we get **deeper in the CNN** and the dimensionality grows, the filters look noisier and become harder to interpret, as it can be seen in Figure 5.2.

5.1.2 Activation maps

Figure 5.3 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 5.1, these activation maps correspond to filters with **almost flat coefficients**. This may be a signal of a high learning rate [5]. In this case, the learning rate is not explicitly declared: the ADADELTA optimizer 1 use an adaptive one.

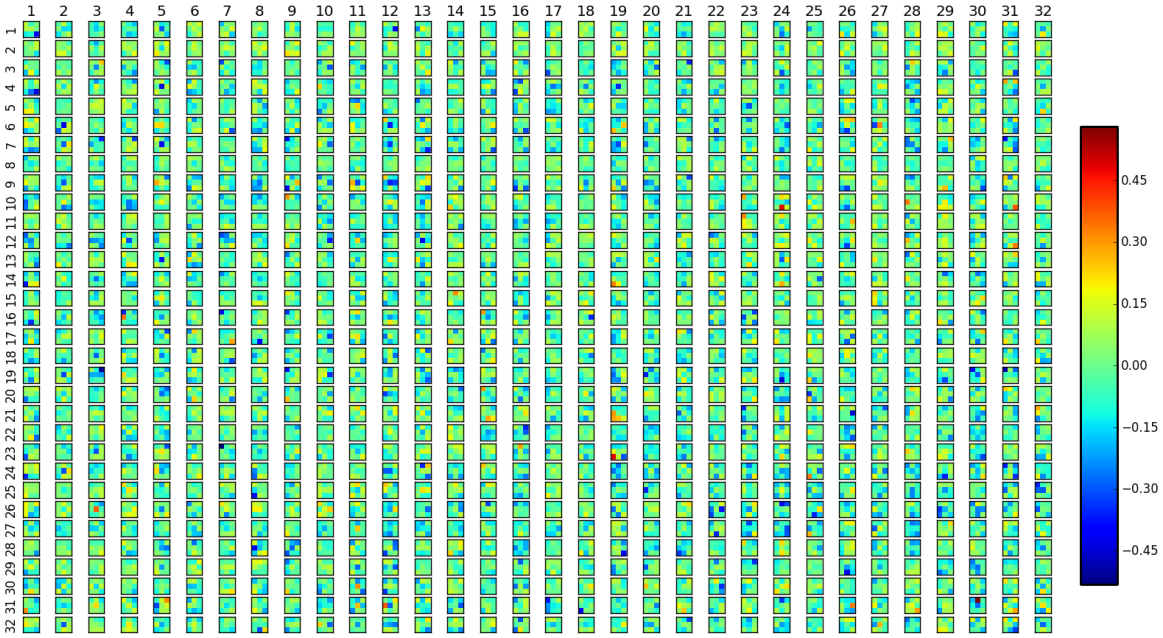


Figure 5.2: Filters of the second convolutional layer. Each row contains the channels (1-32) that correspond to each filter (1-32).

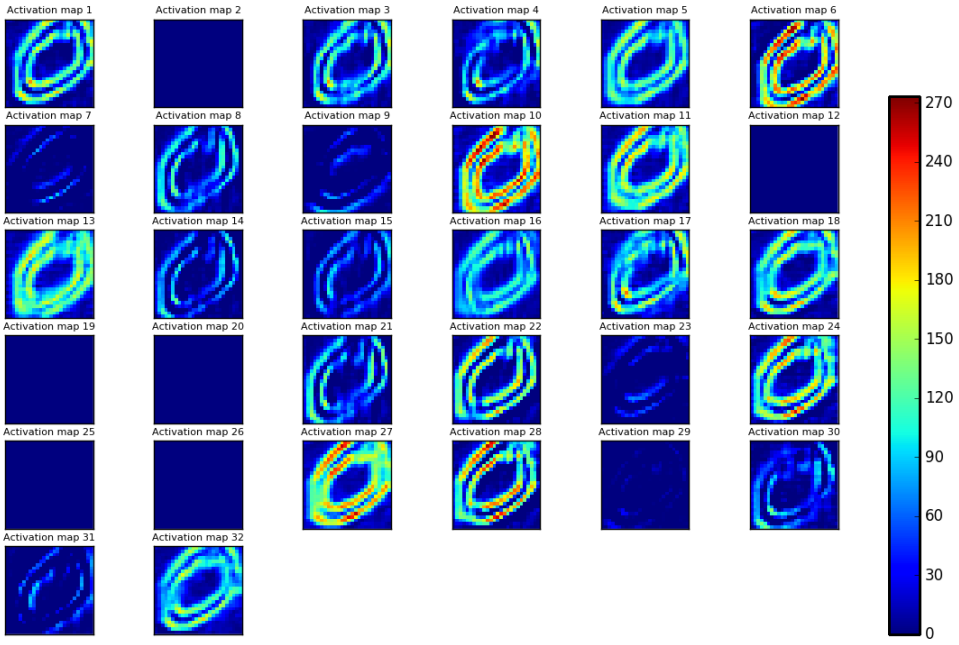


Figure 5.3: Activation maps of the first convolutional layer.

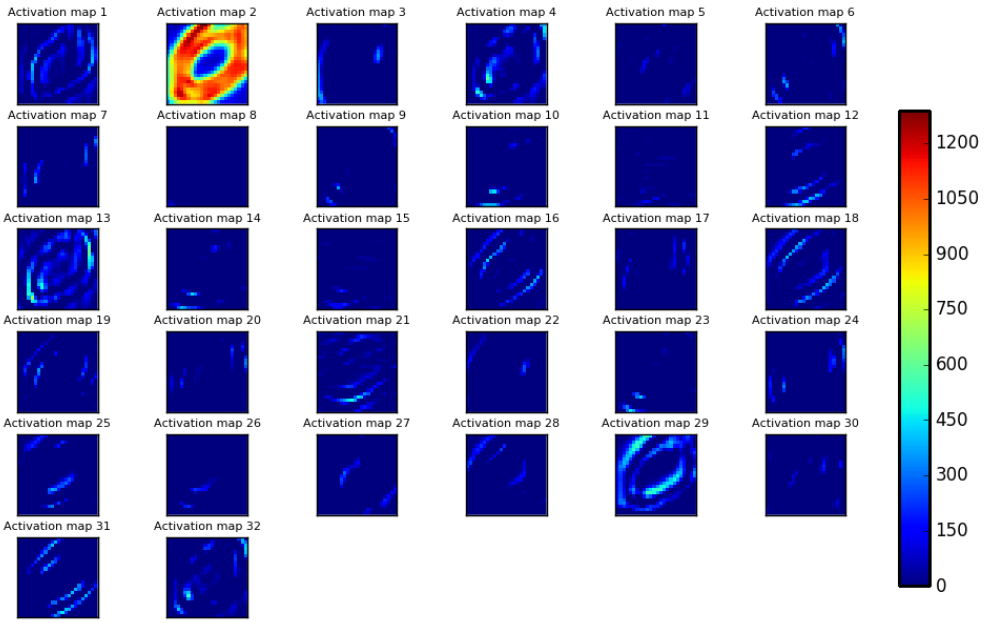


Figure 5.4: Activation maps of the second convolutional layer.

The activation maps of the **second layer** are shown in Figure 5.4. 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.

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 2.2) turn all the negative values to zero.

5.2 New 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 **six times more samples** than *0-1* and *1-1*.

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.

- 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.

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.

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**" [4]. 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 train dataset**. In this section, the effect of applying to the *0-1* model two regularization techniques (**early stopping** and **dropout**) is 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 5.5, 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

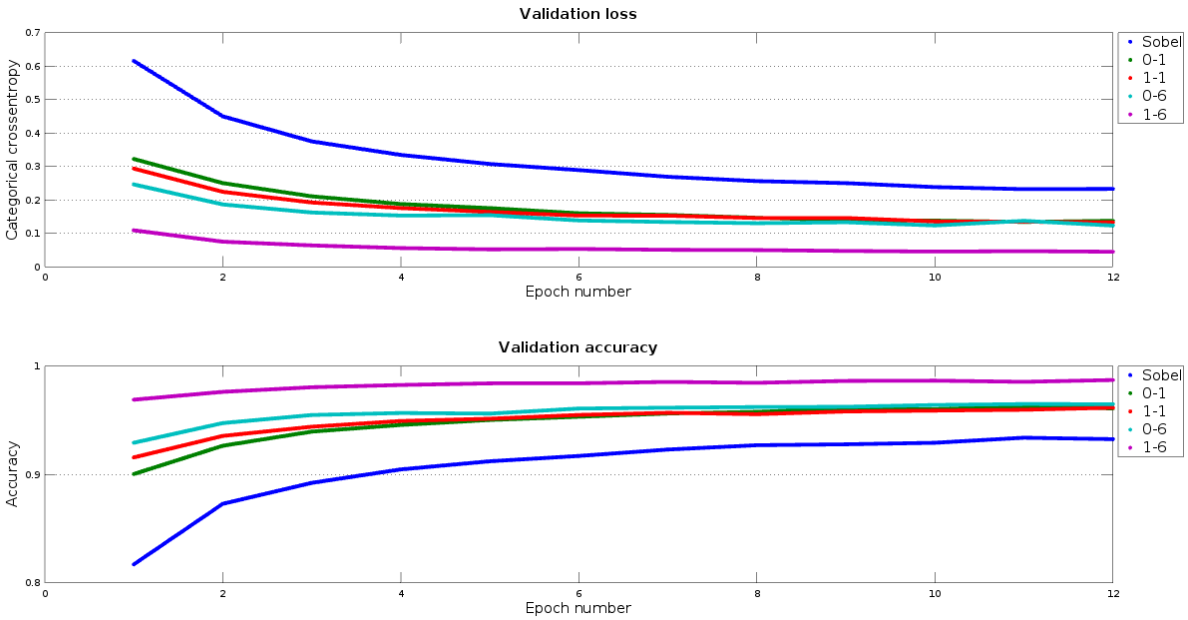


Figure 5.5: Validation results when training the model with different datasets.

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.

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, i.e. a 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 it's the 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 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 after Section 5.4 to reduce the computational cost.

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.

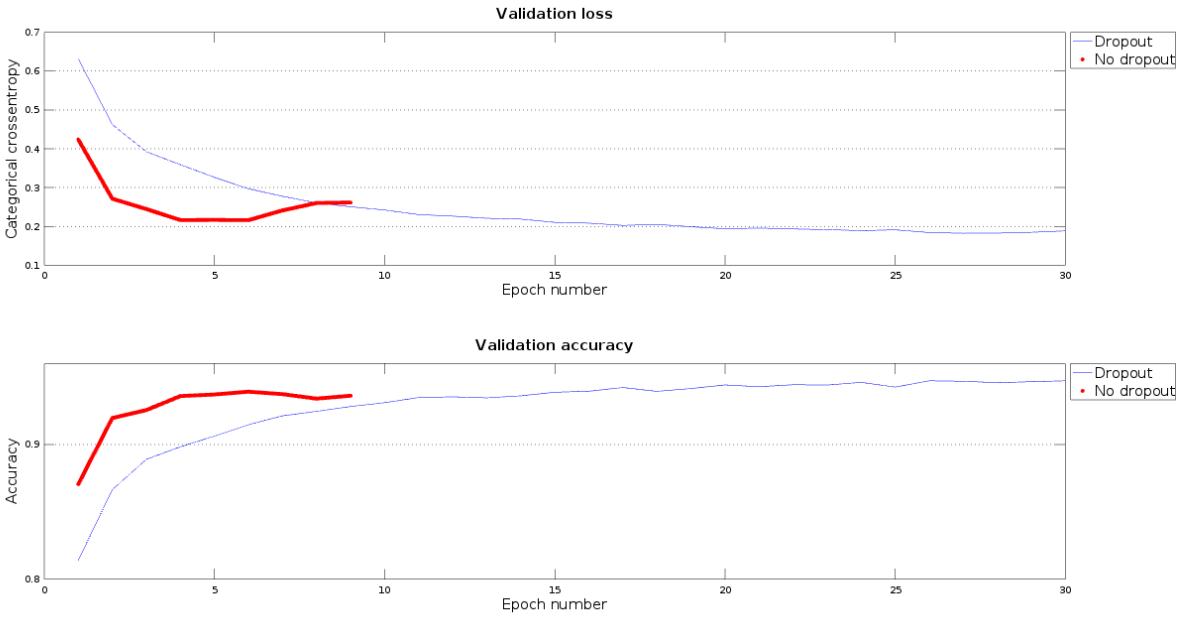


Figure 5.6: Validation results with and without dropout.

5.3.2 Dropout

The models that we're working with 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 model, the *0-1; Patience 2* dataset 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 5.6

Additionally, in Figure 5.7, the learning curves of the models can be seen. It's worth looking into these plots to realize that validation results are better than training results when the model is trained with dropout. This may look illogical, as the CNN should

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.

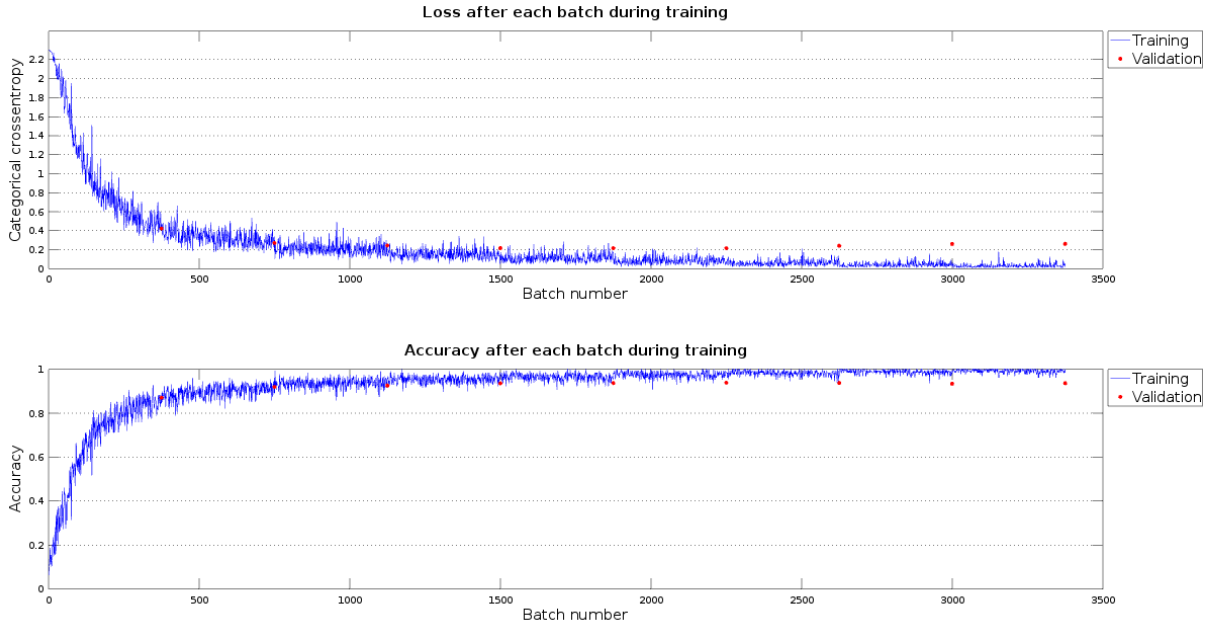
always perform better with samples that it has already seen. However, it's important to remember that dropout only applies during training and, as it will *switch off* a lot of weights in the CNN, much of its prediction power will be lost. During validation, there are no *switched off* weights, which allows the CNN to make better predictions. Besides that, in the figure can be seen that the training results are much better (i.e. lower loss and higher accuracy) 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

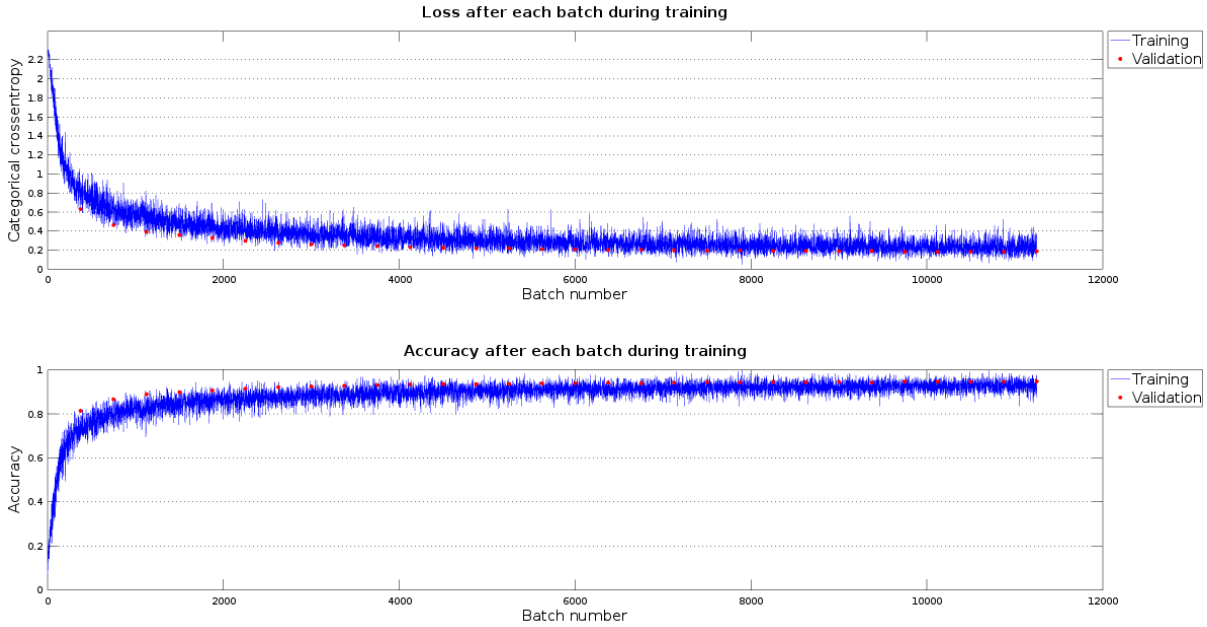
In order to check the influence of different architectures in the performance of the CNN, new models with a different number of convolutional layers have been trained and tested. The stopping rule used in these trainings is the one defined in the previous section and dropout is also applied. The decision 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, I triplicated the model with 2 convolutional layers and one MaxPooling layer. However, a MaxPooling layer was removed because the model ended up working with an empty image: 0x0 size.

As it can be seen 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 5.8), it can be assumed that when we increase the number of layers, 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



(a) Learning curves without dropout.



(b) Learning curves with dropout.

Figure 5.7

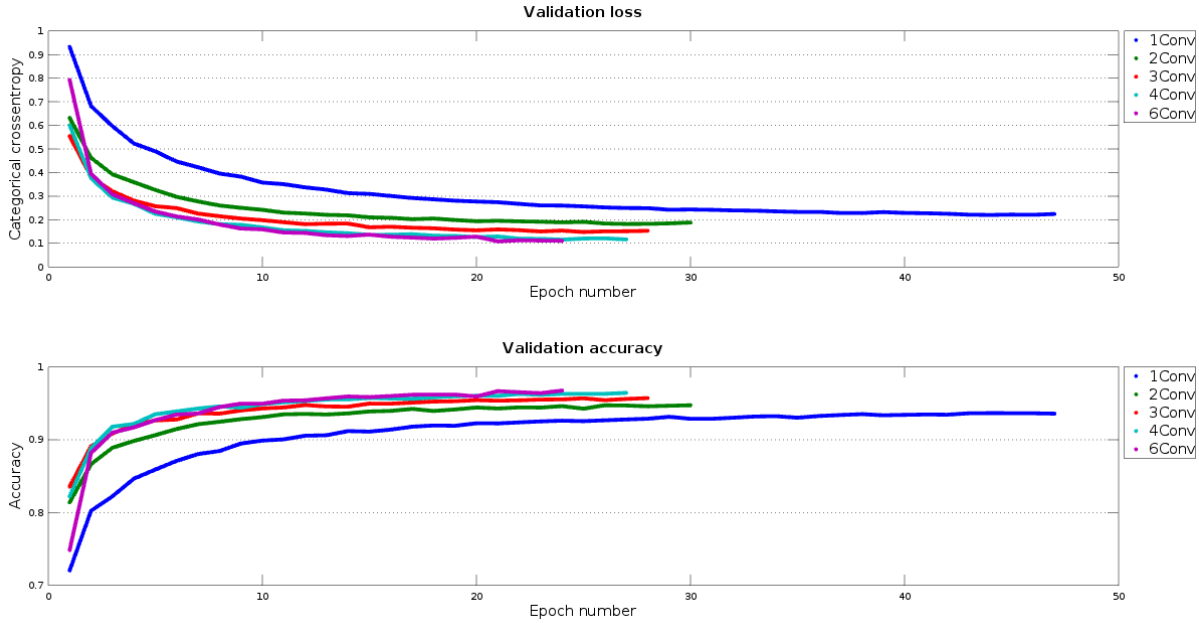


Figure 5.8: Validation results with different architectures.

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.

6Conv model, *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 from 2 to 5. The results obtained with the 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 in this case the improvement is not very significant.

Chapter 6

Conclusions

Bibliography

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

[2] 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).

[3] 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`.

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

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

[6] 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`.

[7] 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`.

[8] 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`.

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

[10] DEV47APPS. (2010-2016). Dev47apps, [Online]. Available: http://www.dev47apps.com/.

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

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

[13] 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).

[14] 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).

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

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

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

[18] 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).

[19] 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).

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

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

[22] 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).

[23] J. D. Hunter, "Matplotlib: A 2d graphics environment", *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.