Compliance Design of Automotive Systems A.Y. 2019/20
Authors: Filippo Cenonfolo, Matteo Lenti, Damiano Vandi

CNN for Traffic Sign Recognition on Raspberry Pi

Summary

1. Introduction

The evolution of connected cars towards higher levels of automation needs advanced systems for the recognition of traffic signs. Autonomous cars from level 2 implement solutions that allow both the detection and the classification of road signs. With the state-of-the-art technology, the best approach for this application is represented by Machine Learning algorithms, in particular Convolutional Neural Networks (CNNs) for image classification. These approaches are often related to high computational power while cars are equipped with embedded systems that present performance constraints. For these reasons, the aim of this project is to develop a network for traffic signs classification with a high level of accuracy, deploy it on an embedded system and test it using a validation technique. We used the German Traffic Sign Recognition Benchmark (GTSRB) to train and test the networks, which is composed of thousands of RGB images divided into 43 different classes. We implemented a modified version of ResNet (one of the classical models for image classification) specifically tailored for Traffic Sign Recognition. The CNN extract features from 2D images in order to assign the correct class to each element of the dataset. We deployed the trained model through automatic code generation on a Raspberry Pi which is the embedded system that we used to simulate a car ECU.

The remainder of this paper is structured as follows:

- Model-Based Design: this section describes the approach we followed for the development of the project starting from high-level requirements for the network design up to the deployment and test on the target hardware.
- Software Tools: this section presents the software tools we used to build the network, manage the dataset, generate the code for the Raspberry Pi, verify the behavior of the system and finally generate the documentation for the code.
- Neural Network Configuration: this section presents the CNN model used (mResNet) and the steps taken to train and test it in order to verify its performance before the deployment.
- Raspberry Deployment: this section describes the procedure followed to deploy the trained network on the target hardware and the validation technique applied to test the system.

2. Model-based design

For the realization of the project, we followed the V-Model development cycle which is the standard approach in model-based design.

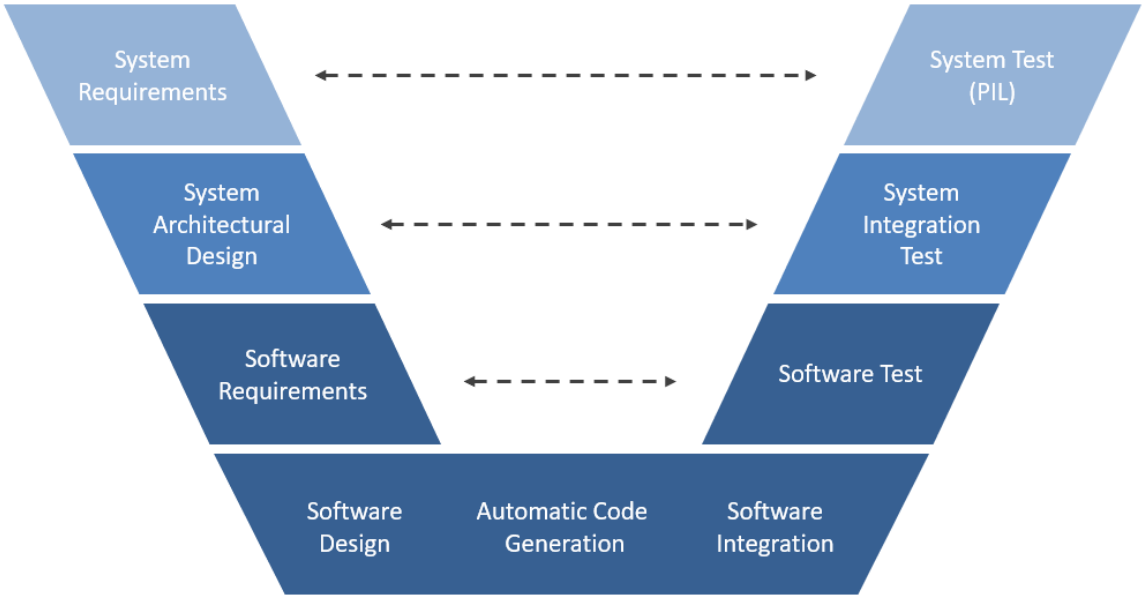


Figure 1 V-Model Development Cycle

The figure presents the phases of the V-Model considered for the development of the project which are briefly described in the following list:

- System Requirements: high-level definition of the system behavior including all the requirements needed to obtain a high accuracy road sign classification system that runs on the embedded device.
- System Architectural Design: definition of individual software modules for the network model, its training and validation, the deployment on the Raspberry Pi and the final validation phase on the target hardware.
- Software Requirements: characterization of the structure of the Convolutional Neural Network, the pre-processing of the images contained in the dataset and the settings used during the training phase including optimizer, batch size and number of epochs. This phase includes also the definition of input and expected output values as well as standard library dependencies.
- Software Design: actual implementation on MATLAB of the software modules defined in the previous step using both hand-written code and live scripts generated by graphical block diagramming tools.
- Automatic Code Generation: C++ code generation considering all the details of the architecture of the target hardware using MATLAB Coder.
- Software Integration: integration of the auto-generated code on the target device with the pre-installed ARM Compute and OpenCV libraries and with the MATLAB scripts running on desktop PC.
- Software Test: preliminary software tests run in MATLAB on desktop PC in order to verify the correct behavior of the scripts developed and the classification accuracy of the trained Neural Network.
- System Integration Test: verify the correct communication on the desktop PC among the MATLAB scripts that represent different modules. This phase includes also a check on the correctness of the code generated for the Raspberry.
- System Test (PIL): final test including Processor in the Loop strategy where the Raspberry is considered as a black box that receives a set of traffic sign images and returns the classified images.

3. Software Tools

We used specific software tools for all the phases of the development of the project. The following subsections present the tools we have chosen and how we decided to exploit them for our final purpose.

3.1. Dataset

The starting point for our project was the German Traffic Sign Recognition Benchmark (GTSRB) dataset presented in the paper "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition". This research aimed to create a complex dataset in order to compare the classification performance of machine learning algorithms with the human ability of recognizing traffic signs.

The dataset is composed of more than 50,000 RGB PPM images that were converted to PNG format which is the one better supported by MATLAB. The images have sizes varying between 15x15 and 250x250 belonging to 43 different classes. The images are divided into two sets, one used for training and validation and the other for testing. The training set is organized in a hierarchical structure with one directory per class, while the test set uses a csv document with the ground truth annotations.

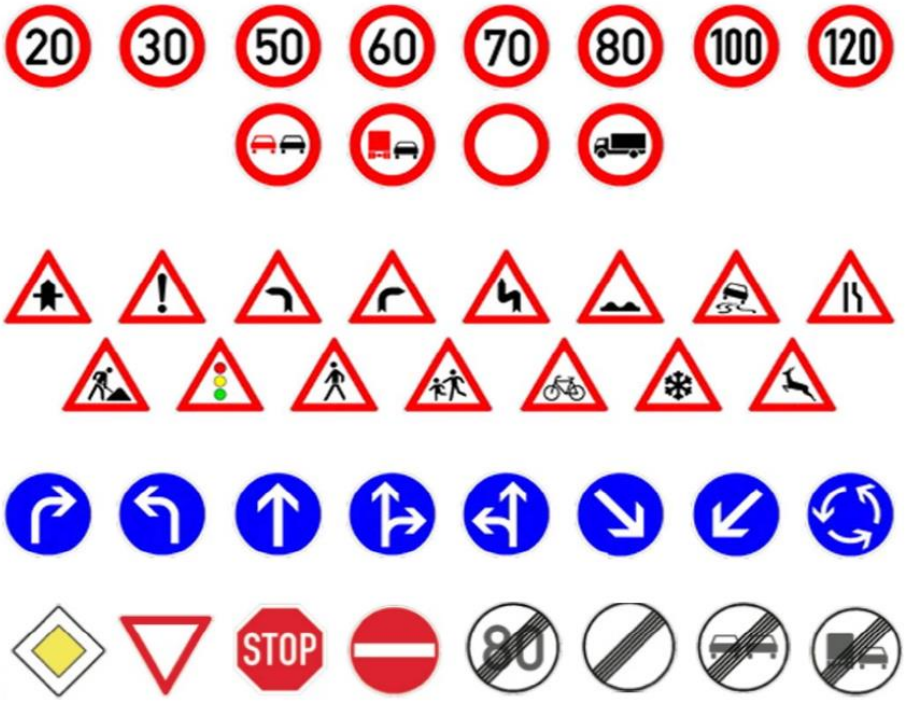


Figure 2 GTSRB Dataset

3.2. MATLAB Add-Ons

The entire project is developed using MATLAB because it offers great support for both Machine Learning algorithms and embedded devices. This support is given by several Add-Ons that are available for the MATLAB platform.

For the part of the project that was mainly related to Convolutional Neural Network we used the following Add-Ons:

- **MATLAB Deep Learning Toolbox**: this Add-On provides a framework for designing and implementing deep neural networks with algorithms, pretrained models and apps. In particular, it includes the Deep Network Designer app, which allows the implementation and analysis of advanced deep Neural Networks using an interface composed of interactive blocks and translates the result into an auto-generated MATLAB live script. This Add-On offers custom training loops, shared weights, automatic differentiation and support for hardware acceleration on NVIDIA GPU. Apps and plots help visualize activations and monitor training progress.
- **MATLAB Coder Interface for Deep Learning Libraries**: this Add-On provides the ability to customize the generated code by leveraging target-specific libraries on the embedded target. With this support package, it is possible to integrate with libraries optimized for specific CPU targets for deep learning such as the ARM Compute Library for ARM architectures, which is the one needed for the deployment on the Raspberry Pi.

For the second part of the project, which was mainly related to the code generation for the embedded device (Raspberry Pi), we installed the following Add-Ons:

- **MATLAB Support Package for Raspberry Pi Hardware**: this Add-On enables the communication between a PC running MATLAB and a Raspberry Pi. The support package functionality is extended if MATLAB Coder is installed.
- **MATLAB Coder**: this Add-On generates C or C++ code from MATLAB code for a variety of hardware platforms, including embedded hardware. The integration of Coder and Raspberry Support Package allows taking the same MATLAB code used to interactively control the Raspberry Pi from the computer and deploy it directly to the Raspberry Pi to run as a standalone executable.
- **MATLAB Embedded Coder**: this Add-On enhances MATLAB Coder for production use with support for code customization, target-specific optimizations, code traceability, and software-in-the-loop (SIL) and processor-in-the-loop (PIL) verification.

3.3. m2html

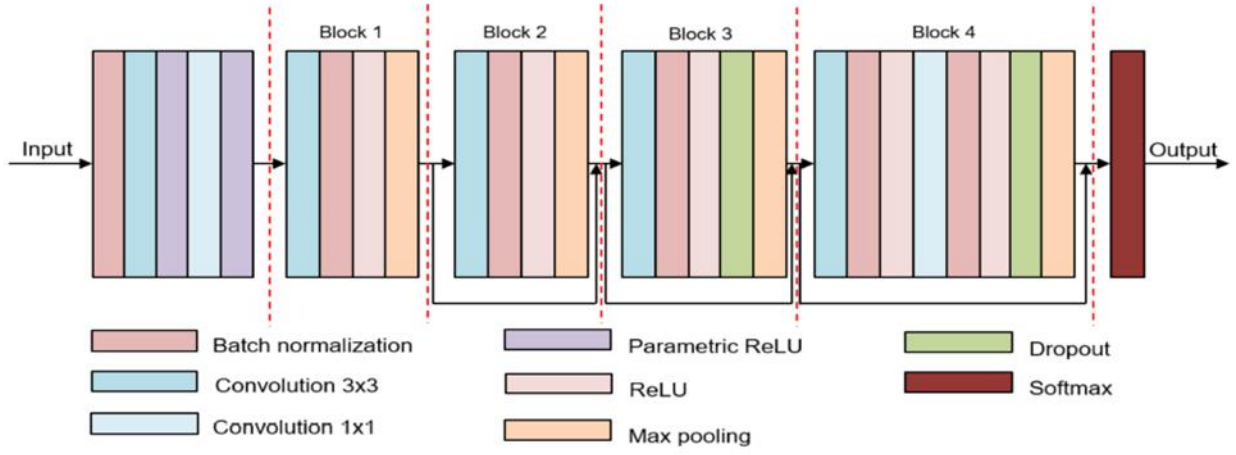
m2html is the tool used to automatically generate HTML documentation from MATLAB m-files. Moreover, it walks recursively in directories, finds dependencies between functions and generates a dependency graph using the dot tool of GraphViz. The m2html tool offers two different templates: default (*blue*) and *frame*. The one used is *frame* which also supports the graph visualization.

4. Neural Network Configuration

In this section, we describe the model of the Convolutional Neural Network that we chose and the steps taken to implement, train and test it in order to verify its correct behavior before deploying it on the target hardware.

4.1. mResNet

The idea for the model of CNN that we implemented came from the paper "*Traffic sign recognition and classification with modified residual networks"* which deals with the problem of road sign classification using a modified version of the famous ResNet architecture. The network is composed of different blocks. Input data is fed to a pre-processing block which includes a batch normalization, two convolutional layers and parametric ReLu. The PReLu layer is a generalization of the traditional rectified unit used to improve model fitting. After pre-processing there are four main blocks. The first two blocks have the same layers, the third adds a Dropout layer and the fourth includes two convolutions, one dropout and max pooling. All four blocks contain standard ReLu layers to add non-linearities to the model. Blocks two, three and four implement a residual connection where the previous output feeds only the first subsequent block. Blocks three and four include a Dropout layer which randomly zeroes some of the elements of the input tensor with 10% probability set as a parameter. This is done in order to prevent overfitting. At the end of the network, there is a fully connected layer that feeds a Softmax function which produces the probability value for each class.



THE DETAILED ARGUMENTS FOR HIDDEN LAYERS. THE OUTPUT SIZE IS AN OUTPUT SIZE OF EACH LAYER. THE NUMBER IN THE OUTPUT SIZE REPRESENTS A*A. IN THE 'OTHER' ROW, THE NUMBERS ARE THE PARAMETERS OF MAX POOLING AND DROPOUT RESPECTIVELY. "-" MEANS THAT HERE DOES NOT NEED THIS KIND OF ARGUMENT.

Layer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Output size	32	32	32	32	32	32	32	32	16	16	16	16	8	8	8	8	8	4	4	4	4	4	4	4	4	1	-
Feature map	-	8	-	8	-	32	-	-	-	64	-	-	-	96	-	-	-	-	128	-	-	43	-	-	-	-	-
Other	-	-	-	-	-	-	-	-	2	-	-	-	2	-	-	-	0.9	2	-	-	-	-	-	-	0.9	4	-

Figure 3 mResNet Architecture

The network was implemented using the layers and residual connections available in the interactive blocks of the app Deep Network Designer. The result of this operation is a DAG (Direct Acyclic Graph) which can be imported in MATLAB with the live script autogenerated by the Designer app.

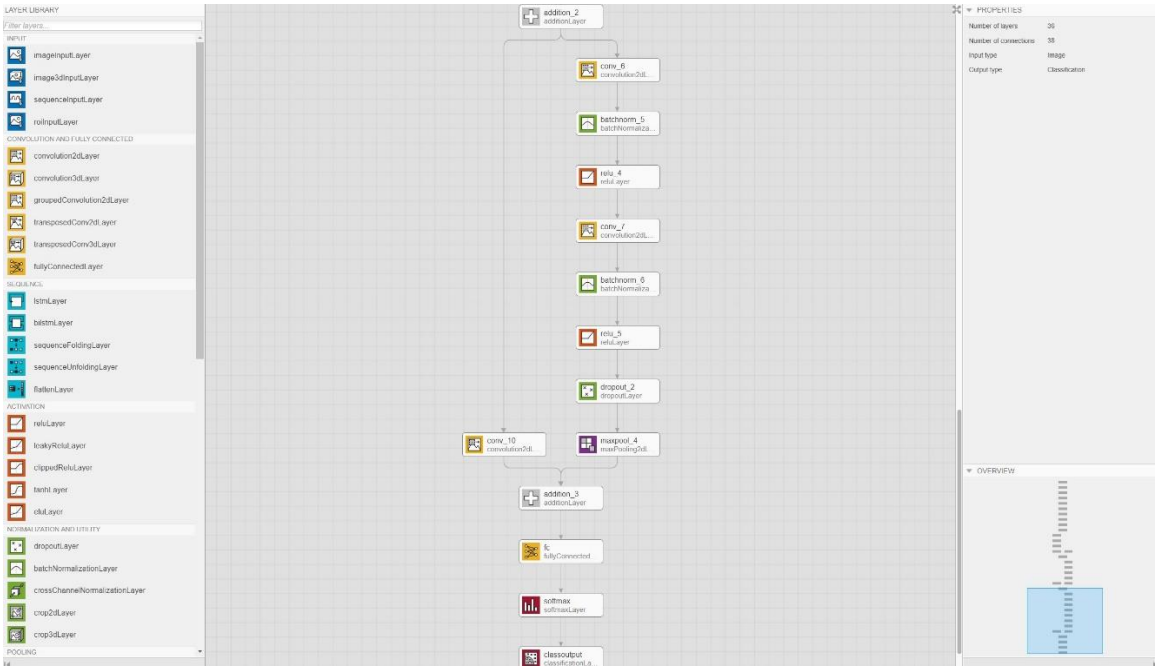


Figure 4 Last Layers of mResNet on Deep Network Designer

4.2. Neural Network Training

In order to parallelize the work we split the training phase into three modules: a function for the dataset management (*DatasetReading.m*), a function for the training settings (*NetworkTraining.m*) and a main script which calls the two functions (*TrainingMain.m*) and returns the trained network.

4.2.1. DatasetReading

All the images of the dataset are loaded using the imageDatastore class from MATLAB which associates each image to the corresponding ground truth class extracted from the folder name. The training set is randomly divided in two parts, one for training containing 80% of the images and one for validation with the remaining ones. After this, the two image subsets are processed using the augmentedImageDatastore class which applies a resize transformation. This is done in order to have the same dimensions for all the images which in our case is 32x32 to reduce computation time.

4.2.2. NetworkTraining

This function defines the training options for the Convolutional Neural Network and actually launches the training phase.

The training options used are described in the following list:

- Optimizer: Adam

  It is the recommended optimization algorithm to use because it computes individual adaptive learning rates for different parameters based on estimates of first and second moments of the gradient. The initial learning rate we used is 0.001 which is the suggested one.
- Epochs: 100

  The network was trained for 100 epochs performing a shuffling of the images for each epoch. We kept 100 epochs even though the validation accuracy stabilized earlier because we found that stopping the training at an intermediate point leads to a worse test accuracy.
- Batch size: 128

  We chose a batch size equal to 128 (the batch size is the number of images concurrently processed by the network) in order to speed up the training process. 128 was selected to have a batch size greater than the number of classes.
- Validation frequency: 245

  We chose the validation frequency equal to 245 iterations because it allows to perform the validation phase after each epoch. The validation process is done using the associated image subset created using imageDatastore. In this way we monitored the accuracy of the network during training.
- Plot: training process

  This option is selected in order to visualize the plot of the training accuracy and the corresponding loss over epochs. The following figure presents the training process of the mResNet network.

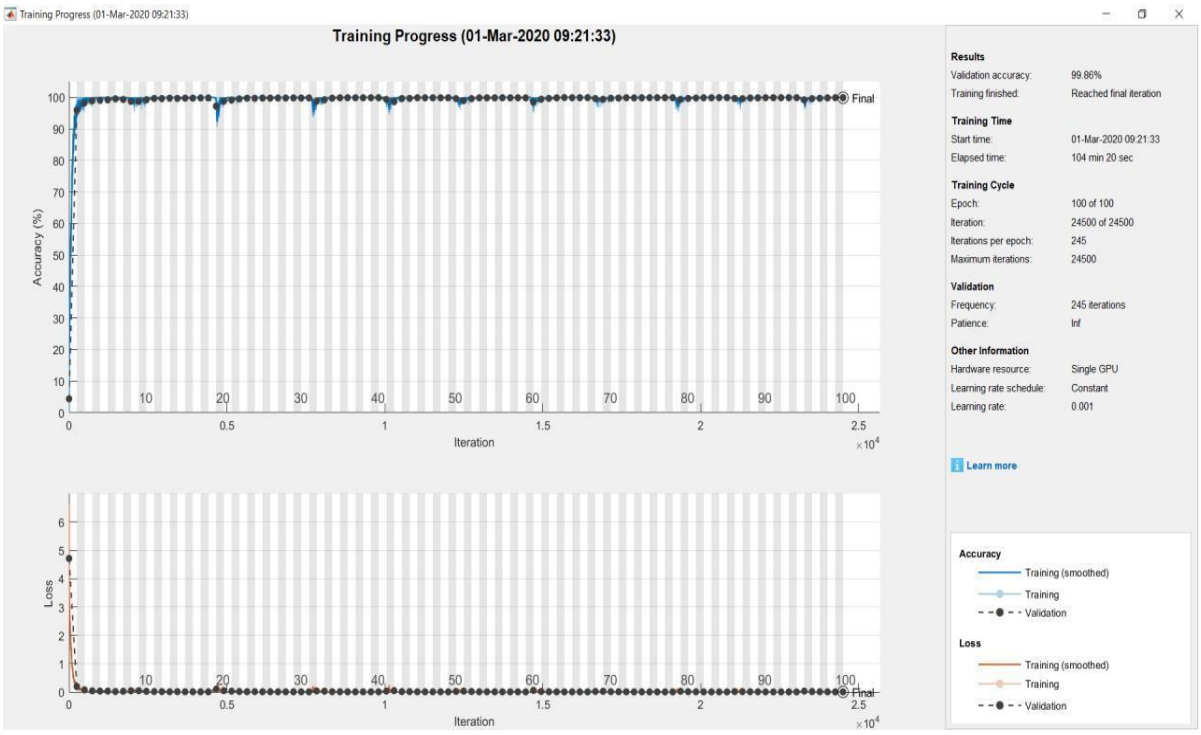


Figure 5 Validation Accuracy Plot

4.2.3. TrainingMain

This script recalls the mResNet model previously created, defines the path for the dataset saved in the local PC folder and executes the two functions previously described. Finally, it returns the trained network which will be tested and later deployed on the Raspberry.

The training and validation of the network were performed exploiting the CUDA cores of an NVIDIA GeForce 940MX with 4GB of GDDR5 VRAM.

4.3. Neural Network Test

In order to verify the correct behavior of the trained network and evaluate its performance before the deployment on Raspberry Pi, we developed and executed a test script (*NetworkTest.m*).

This script loads the pre-trained mResNet model and the test set included in the GTSRB dataset which includes more than 12,000 images belonging to the 43 classes. The ground truth labels for these test images are extracted from the GT-final_test.csv annotation file. After a resize transformation the ground truth labels are compared with the predicted ones obtained using the classify function. The script returns the classification accuracy obtained by the network over the test images. The accuracy obtained is 98.48%.

5. Raspberry Deployment

In this section, we describe the steps taken to deploy the trained mResNet on the embedded system considered. This part of the project aims to obtain a Raspberry Pi that works as a black-box classifier that receives traffic sign images as input and returns the prediction for the same images.

5.1. Raspberry Pi

The low power embedded system used was a Raspberry Pi 3 Model B featuring a Quad Core 1.2 GHz CPU and 1GB RAM. We installed on the SD Card the Raspbian OS version provided by Mathworks. It is a Debian-based computer operating system for Raspberry Pi which includes the basic packages necessary for the interface with MATLAB. The test phase done on the Raspberry also required the installation of ARM Compute library v19.02 necessary for the deployment of deep learning application and OpenCV library v3.1.0 for computer vision operations like image resize.

5.2. Deployed Functions

We present the two functions contained in the scripts *mResNet_classify.m* and *Postprocess.m* that we deployed on the Raspberry using automatic code generation facilities offered by the Add-On MATLAB Coder.

5.2.1. mResNet_classify

This function starts loading the pre-trained mResNet model available in *trained_net.mat* file. Then it does a resize of the image passed as input argument in order to match the dimensions (32x32) expected by the first layer of the network. It assigns a label to the image using the *predict* function and calls the *Postprocess.m*

function passing the predicted class and the resized image as arguments. Finally, it returns the output image obtained by the *Postprocess.m* function.

5.2.2. Postprocess

This function generates the output image adding to the input image a text section that displays the name of the class corresponding to the label predicted by the network. This obtained image is returned to the calling function.

5.3. Automatic Code Generation

In the model-based design paradigm, one of the key aspects is the automatic code generation used at the bottom of the V-Model development cycle which guarantees a faster and more reliable way of implementing low-level software. In our case, we used this technique to generate a C++ executable script from the two functions previously described in order to perform the classification of the traffic sign input images provided on the embedded device. For this purpose, we developed the script *TestCodeGenerator.m* in which we defined the settings needed for the automatic C++ code generation.

5.3.1. TestCodeGenerator

This script establishes a connection with the Raspberry Pi board on which we want to deploy the functions, it defines all the configuration settings needed for the correct auto-generation of the code and it generates and deploys the code on the target hardware. The main settings that we used to configure the coder are:

- Target language: C++
  We chose the C++ programming language because it is the standard for this kind of embedded applications.
- ARM Architecture: armv7
  This is the architecture of the Raspberry Pi board on which the code is executed. We also defined the ARM Compute Library Version (19.02) previously installed on the Raspberry necessary for deep learning applications.
- Target hardware: Raspberry Pi
  This setting enables the coder to exploit the Raspberry Pi hardware support package.
- Verification mode: PIL
  We chose the PIL verification mode defined in the embedded coder to perform Processor in the Loop tests with the auto-generated code.
- Input vector type: images from 15x15 to 2000x2000
  This setting allows different dimensions for the input images passed to the C++ executable script generated.

Finally, we added the instruction *codegen* which is the one responsible for the actual generation, deployment and compiling of the C++ code on the target hardware. This instruction requires the name of the main function to be generated which in our case is *mResNet_classify* and the definition of the input vector type specified in the previous configuration.

5.4. PIL

The final tests of the system were performed using a Processor in the Loop technique in which we used the Raspberry running the previously generated code as a black-box fed with test input images. This is achieved by executing the MATLAB script *PILRaspberryTest.m* on a PC connected through ethernet to the Raspberry Pi.

5.4.1. PILRaspberryTest

This script loads the images from a folder where we placed some examples of traffic signs downloaded from the Internet and passes them as input to the Raspberry running the road sign recognition application. It receives the output images generated by the Raspberry and displays them in different figures in order to verify the correctness of the classification.

The following figure shows an example of input image and the corresponding result obtained from the classification on the Raspberry Pi using the Processor in the Loop technique described.

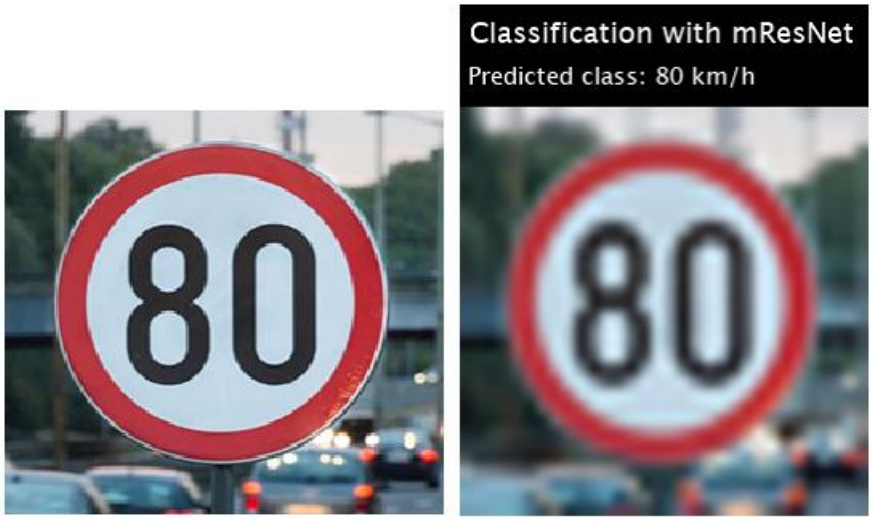


Figure 6 Input and Output Example Images

6. Conclusions

This paper aimed to apply the model-based design in order to develop a Convolutional Neural Network for traffic sign classification and deploy it on an embedded system. For this reason, we followed a V-Model development cycle, starting from the requirements for the network up to the successful Processor in the Loop test done with Raspberry Pi. A fundamental part of the development of the project was played by automatic code generation provided by MATLAB Coder Add-On that allowed the translation of high-level MATLAB scripts to C++ executable code deployed on embedded device. Moreover, the final phase of Processor in the Loop enabled us to test the system as a black-box simulating the acquisition of images from cameras installed on the car. The overall result of the project is positive because we were able to successfully deploy a Convolutional Neural Network capable of classifying traffic signs with high accuracy (higher than 98%) on a Raspberry Pi embedded device.

Figure index