SMART and FLEXible mobile DATA COLLECTOR for GIS

(Acronym, MOBILO)

ENTERPRISES 0916/0055

[D11] Report in WP5 activities

(Automatic Traffic Sign Recognition)

Deliverable n.	D11	**Deliverable title**	Report in WP5 activities
Workpackage	WP5	**WP title**	Automatic Traffic Sign Recognition
Editors	Elias Frentzos (GEO), Dimitrios Skarlatos (CUT)		
Contributors	Elias Frentzos (GEO), Dimitrios Skarlatos (CUT), Lefteris Tournas (CUT), Petros Katsikadakos (GEO)		
Status	Final		
Distribution	Confidential		
Issue date	2020-07-31	**Creation date**	2020-07-01

Contents

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
SDK	Software Development Kit
GPS/GNSS	Global Positioning System / Global Navigation Satellite System
INS/IMU	inertial navigation system / inertial measurement unit
RTK	Real Time Kinematic
CPU	Central Processing Unit
GPU	Graphics Processing Unit
COCO	Common Objects in Context
SSD	Single Shot Multi-Box Detector

REVISION CHART AND HISTORY LOG

REV	DATE	REASON
0.1	01/07/2020	Initial
0.2	15/07/2020	Draft
0.3	31/07/2020	Final

Executive Summary

This deliverable describes the results of MOBILO's project WP5. The specific goal of WP5 was to develop a traffic sign detection module, to automatically identify traffic signs along vehicle tracks. To solve the traffic sign recognition task using the deep learning approach, a convolutional neural network (CNN) was adopted. Due to the lack of huge amount of data required to train a CNN from scratch, the transfer-learning technique was employed. Transfer-learning is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision tasks and fine-tuned to recognize additional classes. For this purpose, a publicly available object-detection model that was pre-trained on the Microsoft COCO (Common Objects in Context) dataset is used. The selected model is the "SSD ResNet50 V1 FPN 640x640" model, freely available by Google. The model was fine-tuned on a new training dataset generated in the framework of MOBILO, to recognize the 25 most popular traffic sign types in Cyprus. The deep learning library TensorFlow 2.4 was used for training and testing. The detection effectiveness varies from 30% to 99%, depending on the scale of the traffic sign, the orientation, the background environment, and the damages, vandalisms, etc. that often occur. However, the implementation of a reliable traffic sign detection model requires continuous training with new data acquired over time, under different environmental conditions. The developed transfer-learning approach for traffic sign recognition in MOBILO allows the re-training of the object detection model in less than 24 hours, whatever new training data are available.

1 Introduction

Traffic signs are an integral part of the road infrastructure. They provide critical information, sometimes compelling recommendations, for road users, which in turn requires them to adjust their driving behavior to make sure they adhere with whatever road regulation currently enforced. Traffic signs need to be consistently inventoried; their condition needs to be in a reliable state to allow a trusting use by the drivers.

Traditionally, standard computer vision methods were employed to detect and classify traffic signs on video images, but these required considerable and time-consuming manual work to handcraft important features in images. Recent advantages in artificial intelligence allow the use of deep learning models that reliably classify traffic signs in a fully automatic way. Such a deep learning architecture has been implemented in the framework of MOBILO, to automatically identify traffic signs along vehicle tracks.

Artificial Intelligence has been witnessing a monumental growth in bridging the gap between the capabilities of humans and machines. Researchers and enthusiasts alike, work on numerous aspects of the field to make amazing things happen. One of many such areas is the domain of Computer Vision. The agenda for this field is to enable machines to view the world as humans do, perceive it in a similar manner and even use the knowledge for a multitude of tasks such as Image & Video recognition, Image Analysis & Classification, Media Recreation, Recommendation Systems, Natural Language Processing, etc. The advancements in Computer Vision with Deep Learning have been constructed and perfected with time, primarily over one particular algorithm — a Convolutional Neural Network.

2 Traffic sign recognition

2.1 Introduction

This section describes our approach for efficiently detecting and recognizing traffic signs in near real-time, taking into account the various condition, illumination and visibility challenges through the means of transfer learning. We tackle the traffic sign detection problem using the state-of-the-art of multi-object detection systems such as the “Single Shot Multi-Box Detector” (SSD) [1] combined with the ResNet50 V1 feature extractor. For this purpose, a publicly available object-detection model that was pre-trained on the Microsoft COCO dataset [2] is used. The selected model is the “SSD ResNet50 V1 FPN 640x640” model, freely available by the TensorFlow Model Zoo. The model was fine-tuned on a new dataset generated in the framework of MOBILO, to recognize the 25 most popular traffic sign types in Cyprus.

In the following, after a short introduction to Convolutional Neural Networks (CNNs), their implementation to traffic sign recognition is discussed. The generation of a new traffic sign dataset for Cyprus, as well as the training and evaluation process are also presented.

2.2 Convolutional Neural Networks

A Convolutional Neural Network (ConvNet/CNN) [3] is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-

processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNetshave the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the entire visual area.

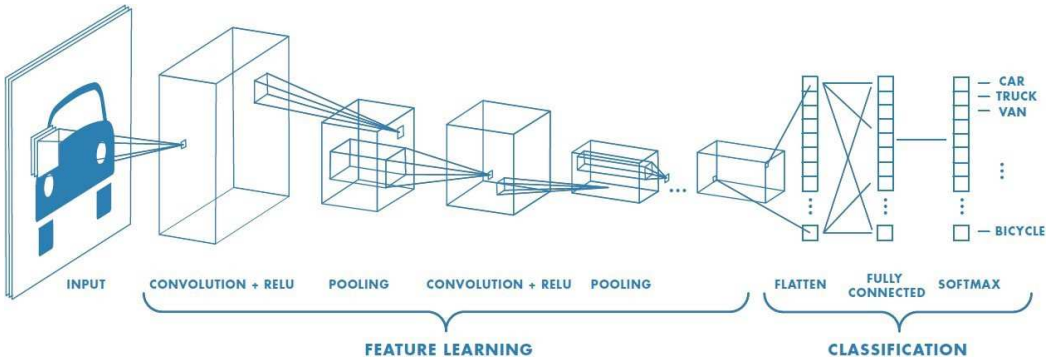


Figure 1: Convolutional Neural Network

A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters (Figure 1). The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

2.2.1 Convolution layers

The input data on a ConvNet are RGB images which have been separated by their three-color planes — Red, Green, and Blue. However, computationally intensive things would get once the images reach higher dimensions, say 8K (7680×4320 pixels). The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets.

The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the Kernel/Filter, K, represented in yellow in figure 2.

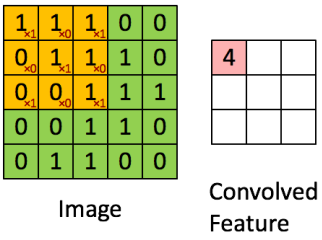


Figure 2: 3x3 Convolution Kernel

The filter moves to the right with a certain Stride Value till it parses the complete image width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed. In case of images with multiple channels (e.g. RGB), the Kernel has the same depth as that of the input image. Matrix Multiplication is performed between Kernel and image pixel values and all the results are summed with the bias to give us a squashed one-depth channel Convoluted Feature Output.

The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image. ConvNets need not be limited to only one Convolutional Layer. Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network, which has the wholesome understanding of images in the dataset, similar to how we would (Figure 3).

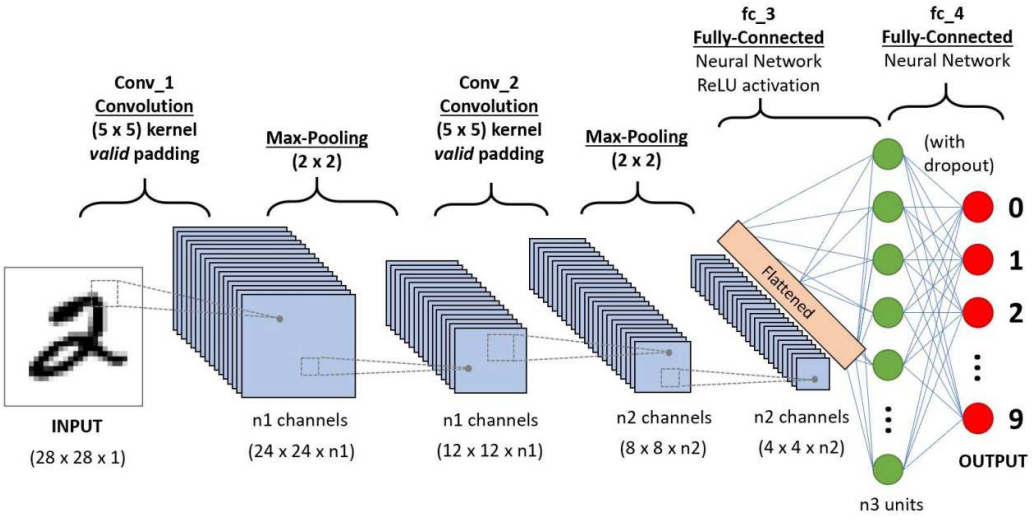


Figure 3: CNN with 2 convolution layers

2.2.2 Pooling Layer

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel (Figure 4). Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism.

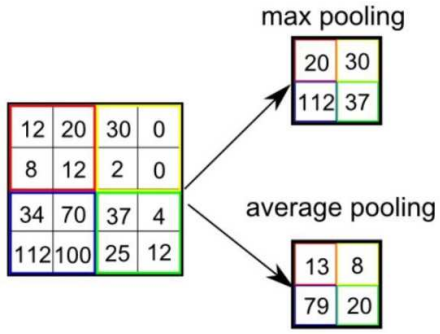


Figure 4: 3x3 Max and average pooling

The Convolutional Layer and the Pooling Layer, together form the *i*-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-level details even further, but at the cost of more computational power.

After going through the above process, we have successfully enabled the model to understand the features. Moving on, we are going to flatten the final output and feed it to a regular Neural Network for classification purposes.

2.2.3 Classification

The classification of the input images is accomplished by adding a Fully-Connected layer at the end of the ConvNet architecture (Figure 5). Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space.

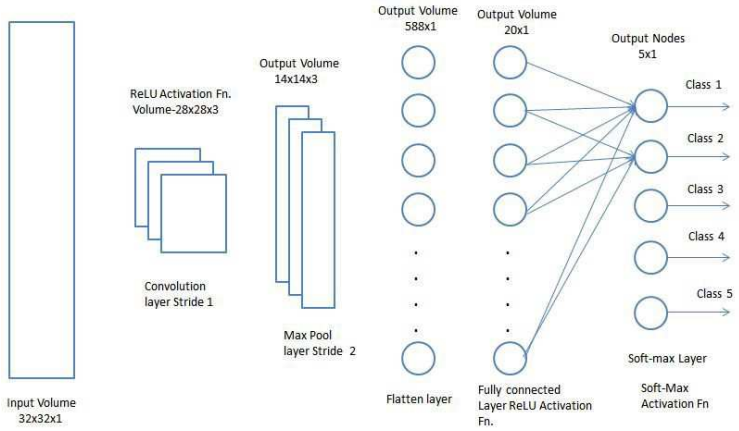


Figure 5: Classification

Now that we have converted our input image into a suitable form for our Multi-Level Perceptron, we shall flatten the image into a column vector. The flattened output is fed to a feed-forward neural network and backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the Softmax Classification technique.

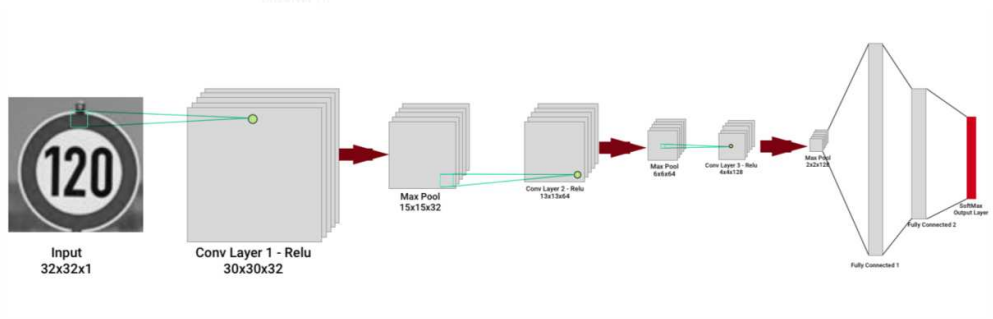


Figure 6: CNN for traffic sign recognition

2.3 Implementation to traffic sign recognition

The deep leering model that we implemented in MOBILO is based on the customization of an existing object detection model to recognize traffic signs on video images. This technique is known as “transfer learning” and is widely used to re-train a model to recognize classes not in the original set. The selected model is the “SSD ResNet50 V1 FPN 640x640” model, freely available by the TensorFlow Model Zoo. The model was fine-tuned on a new dataset generated in the framework of MOBILO, to recognize the 25 most popular traffic sign types in Cyprus.

A large amount of training data is necessary in order to train a deep learning model. The training dataset consists of thousands of images for each traffic sign class, under different scales, orientations and lighting conditions [4] (Figure 7). Such kind of information is not

easily collected, but hopefully is available by several Institutes and organizations worldwide (German Institute for Neuroinformatic, http://benchmark.ini.rub.de).

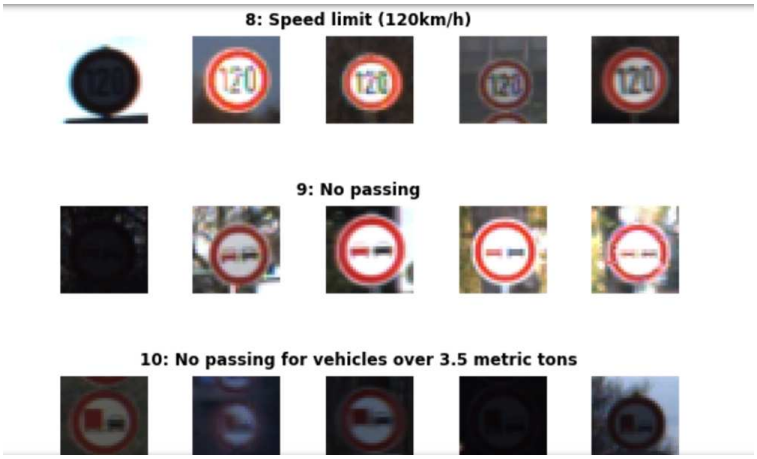


Figure 7: Image samples from a training dataset

In order to extent the available dataset and improve the accuracy of the classification results, image enhancement and data augmentation techniques will are employed. As many of the training images suffer from low contrast (blurry, dark), visibility will be improved applying OpenCV's Contrast Limiting Adaptive Histogram Equalization function. Data augmentation will be used in an attempt to:

- extend dataset and provide additional pictures in different lighting settings and orientations
- improve model's ability to become more generic
- improve test and validation accuracy, especially on distorted images

Affine transformations is applied to augment the images (scale, rotation and translation). A sample of augmented images by affine transformations is shown in Figure 8.

Figure 8: Augmented image samples

The expected classification accuracy is in the order of 95% or even better. It mainly depends on the conformity of the available training datasets to local actual conditions. The implementation of the model is done using Python 3.5 with TensorFlow 2.4.

3 Collecting Data

In order to enhance our deep learning model for traffic sign recognition we proceeded to the collection of traffic signs throughout Cyprus we the aim of our MMS and other available sources. Traditionally label tagging in images is performed with LabelImg [5]. LabelImg is a graphical image annotation tool. It is written in Python and uses Qt for its graphical interface. Annotations are saved as XML files in PASCAL VOC format, the format used by ImageNet. Besides, LabelImg it also supports YOLO format. On the other hand, given that LebelImg supports only image files, without any automation and ability of sample management, we decided to develop new tools for data gathering. These tools are currently available in MOBILO data collection software, as well as a standalone application, which will be presented in the following paragraphs.

We therefore used our MOBILO system, in its current form, and performed data gathering throughout Cyprus, especially in the area of Limassol. The system was mounted on top of a vehicle which was navigated in urban and rural areas collecting video and position data. An example image of the area covered in one data collection date can be found in Figure 9.

Figure 9: *Area covered in one day data collection*

3.1 Collecting data with MOBILO Data processing software

MOBILO data collection tool was used to identify and tag images on the video frames with appropriate tags of traffic sign information. All tags where stored as feature data in the software's project files; we also developed a tool for exporting such feature data into PASCAL VOX XML format.

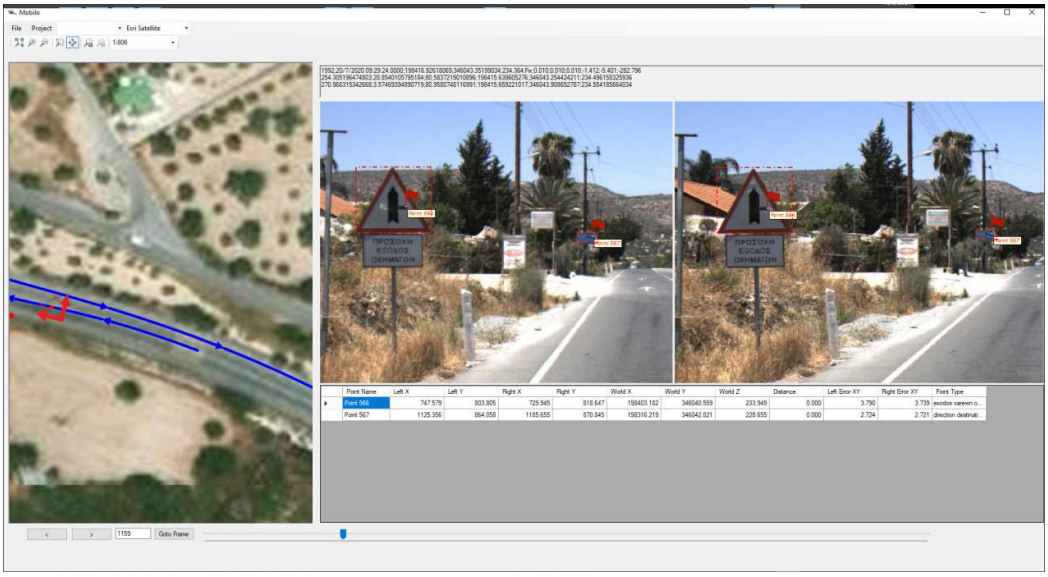


Figure 10: Example screenshot of traffic sign data collection with MOBILO software

PASCAL VOC XML format is an annotation format originally created for the Visual Object Challenge (VOC) and throughout years has become a common interchange format for object detection labels. It's well-specified and can be exported from many labeling tools including CVAT, VoTT, and RectLabel. The attributes used in the XML format are as follows:

- **Folder**: the folder containing the image
- **Filename**: the relative name of the physical image contained in the above folder
- **Path**: the absolute path of the image
- **Size**: height and width in pixels, the depth indicates the number of channels; 3 for RGB image and 1 for B/W
- **Object**:
    - **Name**: contains the name of the object being annotated,
    - **Pose**: specifies the orientation
    - **Truncated**: if object extends beyond the bounding box, 1 if this is true else 0
    - **Difficult**: evaluation difficulty. If object is difficult then 1 else 0.
- **BndBox**: bounding box is consisted of [xmin-top left, ymin-top left,xmax-bottom right, ymax-bottom right]

An example PASCAL VOC attributes file can be found in Figure 11.

```
<annotation>
<folder>image_folder</folder>
<filename>imageName.jpg</filename>
<path>C:\Users\user\image_folder\imageName.jpg</path>
<source>
<database>unknown</database>
</source>
<size>
<width>500</width>
<height>500</height>
<depth>3</depth>
</size>
<segmented>0</segmented>
<object>
<name>ΑΝΩΤΑΤΟΟΡΙΟΤΑΧΥΤΗΤΑΣ 70</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>233.6245</xmin>
<ymin>134.2795</ymin>
<xmax>304.5851</xmax>
<ymax>207.4236</ymax>
</bndbox>
</object>
</annotation>
```

Figure 11: *Example PASCAL VOC attributes*

User may use MOBILO data processing software [D14] to draw a tag around an object and assign a tag value. Clicking on a corner of the box containing an object, the cursor will change into a crosshair which by left clicking and dragging will form a bounding box. By releasing the left mouse button, the rectangle will be placed on screen and the Labeling dialog will open for inserting the name of the annotation. To resize a rectangle on screen, place the cursor at any one of the bounding box borders and after the cursor changes, left click and draw to resize. All data are stored into the MOBILO's project folder and can be exported in PASCAL VOC xml format, together with the respective images.

Finally, we have collected over 3000 samples displayed on 1700 images.

3.2 SharpLabelImage application development and usage

Apart from the images collected by our system we decided to enhance our deep learning model with images acquired by other sources, such as, Google Maps Street View etc. On the other hand, label tagging in images performed with LabelImg [5] requires having the actual image already stored in a location on the computer's disk. As such, the need for a new tool that can be used in combination with such images displayed in computer's screen arises.

Specifically, SharpLabelImage is a graphical image data labeling and annotation tool using object bounding boxes to tag and identify specific portions of an image. For each image in the dataset processed by the SharpLabelImage tool, a single XML file is produced and exported, containing the annotations in the Pascal Visual Object Classes (VOC) format.

The SharpLabelImage [D17] application is an extension of the LabelImg program. While both programs offer similar functions and annotation tools, SharpLabelImage imports images, not only as files from the local system as LabelImg does, but is also able to capture sizable screenshots from computer monitors and transfer them in the application for annotation.

Apart from importing single images, the user can import datasets of PASCAL VOC xml files and their accompanying images for further manipulation of their objects. These datasets could have either being produced from SharpLabelImage or from another similar data labeling program.

The main difference of SharpLabelImage with other alternatives is that it allows capturing screenshots from computer's desktop, enabling thus the ability to use images from any source that is available on computer screen such as, online maps and photos etc. Moreover, it provides tools for PASCAL VOC xml files mass manipulation, such as renaming annotations, on all files in selected folders, displaying cropped images by annotation etc. A detailed software description will be included on deliverable [D17].

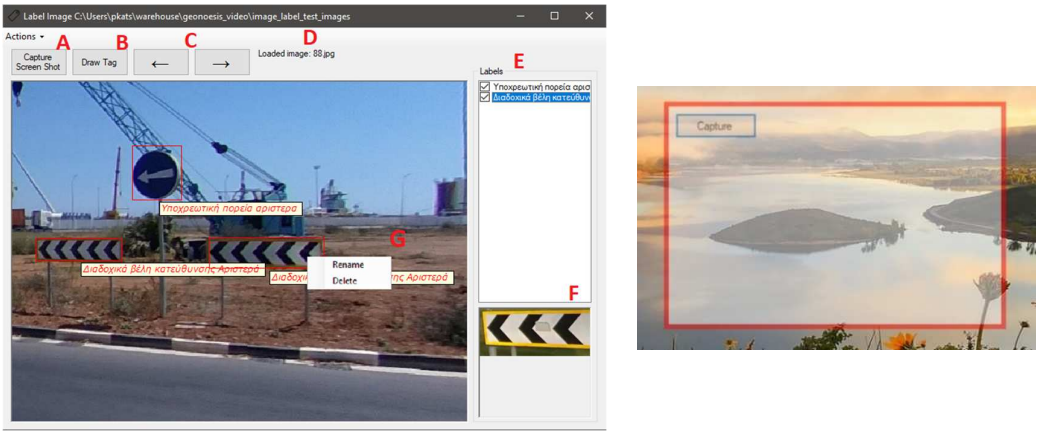
3.3 SharpLabelImage short description

SharpLabelImage requires user to define two folders from their local system.

- The captured image folder for storing and loading images and their PASCAL VOC xml, and,
- the sample image folder for saving sample thumbnails for each annotation taken while using the program.

These folders can be specified by left clicking the corresponding options from the Design Area Action.

In order to capture a screen shot, the capture Screen Shot button must be pressed. This will activate the snipping rectangle area which can be positioned by clicking with the left mouse button and dragging it to the desired portion of screen. To resize the captured rectangle is done by clicking and dragging its borders. User can press the Capture button in order to transfer the snippet on the Design Area or press the Esc button to cancel the procedure and close the snipping tool.



(a) (b)

Figure 12: SharpLabelImage Design area (a) and screenshot area (b)

The Edit Label form offers a detailed report of all the labels used by the program, the snapshots taken and provides the user with the ability to delete or rename a label from all images. To open the form left click the Edit option from the Action menu.

The table provides information about the labels found in the PASCAL VOC xml.

- Label: the name of the label in alphabetical order.
- Count: the number of times the label has been used in the annotated images.
- Sample: a sample thumbnail image corresponding to the label.

By left clicking a row of the table a label is selected and the image information where this label has been used are loaded in the right-hand side Annotation Information table.

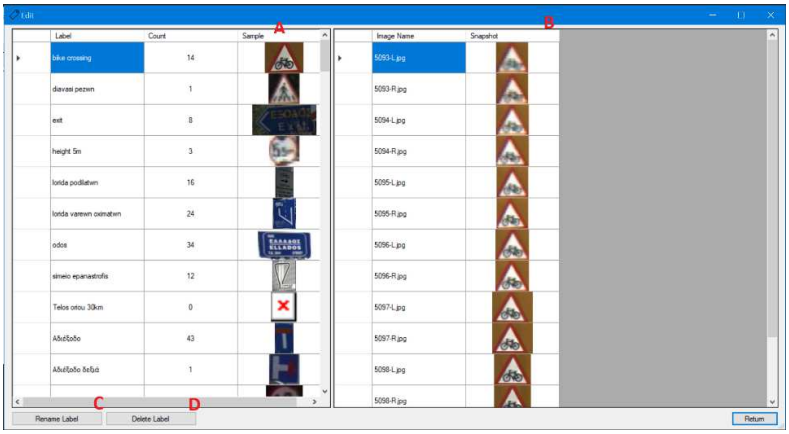


Figure 13: Managing and editing labels

To rename a label user may left click the row of the Report table that he wishes to rename and then left click the Rename Label button. An input text box will open for inserting the new label name. Because this action will replace the label from all PASCAL VOC xml in which it is found extra confirmation is required by the user.

The project's code base is written in C# version 5 and its graphical interface utilizes Windows Forms compiled in .Net Framework 4.5 and was developed in Visual Studio 2019 IDE. The project's code will be ported into GitHub as an Open-Source Project available to download by everyone.

3.4 Collecting data with SharpLabelImage

Using SharpLabelImage we performed data collection over Google Street View images displayed on a browser. The data were collected throughout Cyprus, especially in the areas of Nicosia, Limassol and Pafos. We have collected over 700 images containing over 1500 samples.

4 Training with TensorFlow 2.4

4.1 Introduction

Tensorflow [6] is an open-source library for numerical computation and large-scale machine learning that ease Google Brain TensorFlow, the process of acquiring data, training models, serving predictions, and refining future results. Tensorflow bundles together Machine Learning and Deep Learning models and algorithms, using Python as a convenient front-end and runs it efficiently in optimized C++.

Tensorflow allows developers to create a graph of computations to perform. Each node in the graph represents a mathematical operation and each connection represents data. Hence, instead of dealing with low-details like figuring out proper ways to hitch the output of one function to the input of another, the developer can focus on the overall logic of the application.

TensorFlow is at present the most popular software library. There are several real-world applications of deep learning that makes TensorFlow popular. Being an Open-Source library for deep learning and machine learning, TensorFlow finds a role to play in text-based applications, image recognition, voice search, and many more. The image recognition is covered by a dedicated API, the TensorFlow Object Detection API.

The TensorFlow Object Detection API is an open-source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. There are already pre-trained models in their framework which are referred to as Model Zoo. A collection of pre-trained models trained on various datasets are available to the users for further customization, such as

- the COCO (Common Objects in Context) dataset,
- the KITTI dataset,
- and the Open Images Dataset.

These various models have different architecture and thus provide different accuracies but there is a trade-off between speed of execution and the accuracy in placing bounding boxes around detected objects. Transfer learning techniques can be applied on these pre-trained models. Transfer Learning is one of the major developments in the case of Deep Learning for Object Detection [7]. In transfer learning, we take a pre-trained model performing classification on a dataset and then apply this same model to another set of classification task by just optimizing some hyperparameters. Transfer Learning has two benefits:

- It requires less time to train the model as it is already trained on a different task
- It can be used for tasks which have smaller dataset as the model is already trained on a larger dataset and the weights are transferred to the new task

Transfer learning techniques are employed to train an existing object detection model to recognize traffic sign plates in the framework of MOBILO. From the available pre-trained models by TensorFlow 2 Detection Model Zoo, the “SSD ResNet50 V1 FPN 640x640” model was selected, since it provides a relatively good trade-off between performance and speed.

4.2 Prerequisites

Several software tools must be installed to train an object detection model in TensorFlow 2.4 on Windows 10 environment. The required packages are:

	Software package	Description	
1	Anaconda Python 3.7	Anaconda is not a requirement in order to install and use TensorFlow but is strongly suggested due to its intuitive way of managing packages and setting up new virtual environments for working with Python.	Optional
2	TensorFlow 2.4	TensorFlow is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.	
3	CUDA Toolkit v10.1	Although using a GPU to run TensorFlow is not necessary, the computational gains are substantial. Therefore, if a compatible CUDA-enabled GPU is available, it is recommended to install the appropriate CUDA Toolkit.	Optional
4	CuDNN 7.6.5	The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.	Optional
5	TensorFlow Object Detection API	The TensorFlow Object Detection API is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models.	
6	Google Protobuf	The Tensorflow Object Detection API uses Protobufs to configure model and training parameters. Protobuf are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.	
7	COCO API	COCO is a large image dataset designed for object detection, segmentation, person keypoints detection, stuff segmentation, and caption generation. COCO API provides Matlab, Python, and Lua APIs that assists in loading, parsing, and visualizing the annotations in COCO.	

4.3 Partition the Dataset

From the available annotated images, it is a general convention to use only part of them for training, and the rest is used for evaluation purposes. Typically, the ratio is 9:1, i.e., 90% of

the images are used for training and the rest 10% is maintained for testing. All training images, together with their corresponding *.xml files, should be placed inside a folder, for example "tensorflow/images/train". Similarly, all testing images, with their *.xml files, should be placed inside "tensorflow/images/test". This process can be automatically done using a script provided by the TensorFlow Object Detection API:

```
partition_dataset.py [-h] [-i IMAGEDIR] [-o OUTPUTDIR] [-r RATIO] [-x]

optional arguments:
  -h, --help       Show help message
  -i IMAGEDIR      Path to the folder where the image dataset is stored.
  -o OUTPUTDIR     Path to the output folder where the train and test dirs should be
                   created. Defaults to the same directory as IMAGEDIR.
  -r RATIO         The ratio of the number of test images over the total number of
                   images. The default is 0.1.
  -x               Set this flag if you want the xml annotation files to be
                   processed and copied over.
```

From the available annotated images, 1027 were finally used. From these annotations, 932 (90%) were used for training and 104 (10%) for testing.

4.4 Create Label Map

TensorFlow requires a label map, which namely maps each of the used labels to an integer value. This label map is used both by the training and detection processes. The labels (i.e. the training classes) used in Mobilo are presented in the following table:

Class id	Class name	Number of labels
1	Bus stop	38
2	Danger animals	39
3	Danger due to frequent movement of children	38
4	Danger of pedestrian crossing	39
5	Dangerous right turn	44
6	Dangerous two opposite or consecutive (continuous) turns, the first right	46
7	Deadlock	32
8	It is forbidden to overtake motor vehicles, except for two-wheeled motorcycles without a basket	50
9	mandatory course bottom right	38
10	Mandatory course bottom right or left	62
11	mandatory course lower left	49
12	Mandatory course on the left	32
13	Parking	37
14	Prioritization	56
15	Reversal is prohibited	60
16	Right turn is prohibited	62
17	Road curves	48
18	Road narrows on both sides	42
19	Speed control	38

Class id	Class name	Number of labels
20	speed limit 30	78
21	speed limit 50	60
22	speed limit 65	59
23	speed limit 80	43
24	Traffic lights	58
25	Traffic lights in front	44

Below is an example of the first classes on the label map used in Mobilo:

```
item {
    id: 1
    name: 'Bus stop'
}

item {
    id: 2
    name: 'Danger animals'
}
…
```

4.5 Create TensorFlow Records

When annotations have been generated and split into the desired training and testing subsets, annotations must be converted into the so called TFRecord format. This can be accomplished by running a script that iterates through all *.xml files in the tensorflow/images/train and tensorflow /images/test folders, and generates a *.record file for each of the two:

```
generate_tfrecord.py [-h] [-x XML_DIR] [-l LABELS_PATH] [-o OUTPUT_PATH] [-i
IMAGE_DIR] [-c CSV_PATH]

optional arguments:
  -h, --help       show help message and exit
  -x XML_DIR       Path to the folder where the input .xml files are stored.
  -l LABELS_PATH,  Path to the labels (.pbtxt) file.
  -o OUTPUT_PATH,  Path of output TFRecord (.record) file.
  -i IMAGE_DIR,    Path to the folder where the input image files are stored.
  -c CSV_PATH,     Path of output .csv file. If none provided, then no file will be
                   written.
```

4.6 Configuring a Training Job

In order to configure a training job, the latest pre-trained network for the model we wish to use must be downloaded. This can be done by simply clicking on the name of the desired model in the table found in TensorFlow Detection Model Zoo:

https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

In case of Mobilo, the "SSD ResNet50 V1 FPN 640x640" model is used. The model is available in zip format and must be extracted in a separate folder, for example "tensorflow/pre-trained-models". The configuration of the training pipeline is accomplished by making some critical changes to the pipeline.config file. The changes that should be applied are as follows:

```
Line 1: num_classes: 25 # Set this to the number of different label classes

Line 161: fine_tune_checkpoint: "…" # Path to checkpoint of pre-trained mode

Line 172: label_map_path: "…" # Path to label map file

Line 174: input_path: "…" # Path to training TFRecord file

Line 182: label_map_path: "…" # Path to label map file (the same as Line 172)

Line 186: input_path: "…" # Path to testing TFRecord
```

Once the above changes have been applied to the pipeline.config file, the network training can be start.

4.7 Training the Model

The training of the model is completed by running the following python script, provided by the TensorFlow 2 Object Detection API:

```
python model_main_tf2.py [--model_dir=…] [--pipeline_config_path=…]
```

where --model_dir points to the path where the new trained model will be saved and --pipeline_config_path is set to the path of the pipeline.config file.

The training outputs logs only every 100 steps by default, therefore after a while, a log for the loss at step 100 should be displayed:

```
INFO:tensorflow:Step 100 per-step time 1.153s loss=0.761
I0716 05:26:55.879558  1364 model_lib_v2.py:632] Step 100 per-step time 1.153s
loss=0.761
...
```

Training times can be affected by a number of factors such as:

- The computational power of hardware used (either CPU or GPU): Obviously, the more powerful your PC is, the faster the training process.
- Whether you the TensorFlow CPU or GPU variant is used: In general, even when compared to the best CPUs, almost any GPU graphics card will yield much faster training and detection speeds.
- The complexity of the objects we are trying to detect: In case of traffic sign detection, the training will be a much more challenging and time-consuming process, due to the high variability of the shape and size of signs, combined with a highly dynamic background.

A very nice feature of TensorFlow, is that it allows to continuously monitor and visualize a number of different training/evaluation metrics, while a model is being trained. The specific tool that allows us to do all that is Tensorboard. TensorBoard is a web server, which (by default) listens to port 6006 of the machine. Assuming that everything went well, a typical TensorBoard screen will be like this:

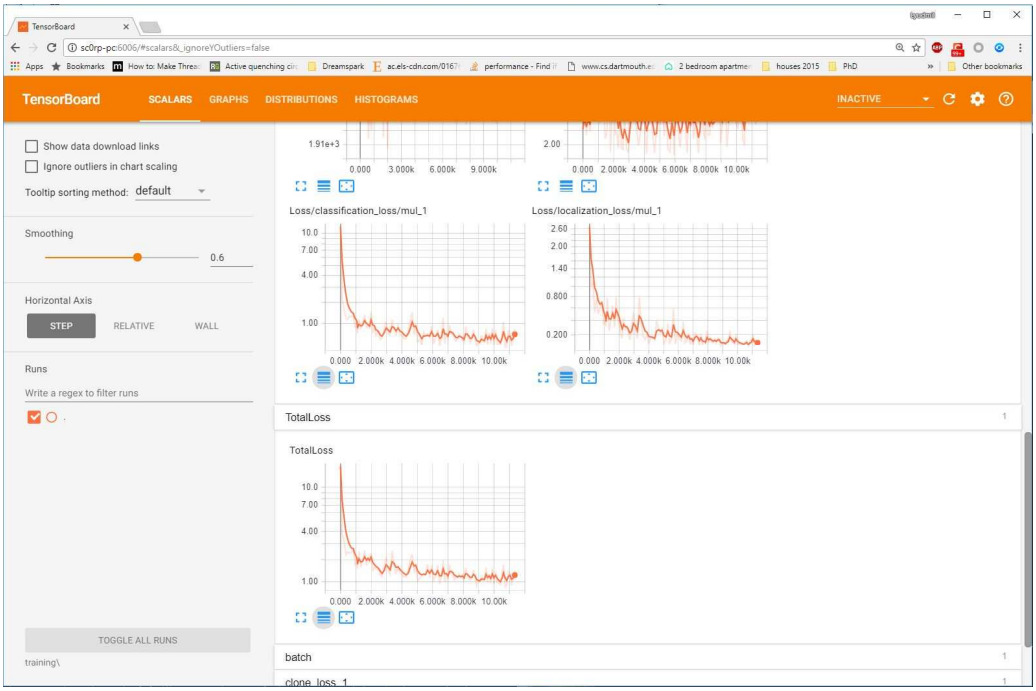


Figure 14: Training monitoring with TensorBoard

In case of Mobilo, the hardware equipment used include: Intel CoreTM i7-5820k CPU, NVIDIA GeForece GTX 1080 Ti with 11GB RAM DDR5, 64 GB RAM DDR4. The training time using this equipment was in the order of 20 hours.

4.8 Exporting a Trained Model

Once a training job is complete, the model should be extracted to a newly trained inference graph, which will be later used to perform the object detection. This can be done employing the exporter_main_v2.py python script, available by the TensorFlow Object Detection API.

```
exporter_main_v2.py [--input_type image_tensor] [--pipeline_config_path "…"]
                        [--trained_checkpoint_dir "…"] [--output_directory "…']
```

Where --pipeline_config_path the path to the pipeline.config file, --trained_checkpoint_dir the folder where the trained model exists and --output_directory the folder where the exported model will be saved.

5 Results

The evaluation of the results of the traffic sign detection model developed in the framework of MOBILO is caried out using the AI_Detect function implemented in the MOBILO API [D15]. The function takes as input an image and returns a number of detected classes with corresponding detection scores, expressed in %. The location of a traffic sign on the image is expressed by four integer values (upper, left, width, height) forming a rectangle around the object:

Figure 15: Traffic sign detection using MOBILO API

The detection score varies from 30% to 99%, depending on the size of the object, the orientation, the background environment, and the artifacts (damages, vandalisms, etc.) that are frequently observed on traffic signs in Cyprus. Some classes may be lost, although there are included in the 25 classes on the training dataset. This is because the number of training images on this classes is limited and should be improved in feature trainings.

As a conclusion, the development of a reliable traffic sign detection module requires continuous training with new data acquired over time, under different environmental conditions. The greater number of training data, the better results will be expected. The developed training configuration allows the update of the training model in 24 hours, whatever new data are coming.

6 References

[1] W. Liu, D. Anguelov, D. Erhan, C. Szegedy and S. Reed, "SSD: Single shot multibox detector," *Lecture Notes in Computer Science,* vol. 9905, pp. 31-37, 2015.

[2] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollr and L. Zitnick, "Microsoft coco: Common objects in context.," in *Computer Vision – ECCV 2014 13th European Conference*, Zurich, 2014.

[3] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way," 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed 2021].

[4] A. Shustanova and P. Yakimov, "CNN Design for Real-Time Traffic Sign Recognition," *Procedia Engineering,* vol. 201, pp. 718-725, 2017.

[5] "LabelImg," [Online]. Available: https://github.com/tzutalin/labelImg. [Accessed 07 2020].

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis and J. Dean, "Tensorflow: A system for large-scale machine learning," *Symposium on Operating Systems Design and Implementation,* pp. 265-283, 2016.

[7] TensorFlow, "Training Custom Object Detector," 2021. [Online]. Available: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#preparing-the-workspace. [Accessed 2021].