An Autonomous Indoor Navigation System Based on Visual Scene Recognition Using Deep Neural Networks

Francesco Bidoia

Supervisors: **Lambert Schomaker**
Amirhossein Shantia

Artificial Intelligence
University of Groningen

This thesis is submitted for the degree of
Master of Science

Groningen University September 2017

Abstract

In this thesis we propose a novel approach for Indoor Navigation Systems that is based on visual information. This new method discards the need for precise Global Localization while adopting a more human-like approach. For this aim, we deeply analyze the current state of the art of computer vision, comparing the classical methods such as Scale Invariant Feature Transform (SIFT) and Visual Bag of Words, with the most recent successes of Convolutional Neural Networks in this field. A further analysis of the state of the art of Deep Neural Networks for image classification is proposed, which focuses on the similarities and differences when compared to navigation tasks. Based on this analysis, we developed a novel Deep Neural Network architecture that takes inspiration from the most recent Inception V3 architecture. The results obtained from specifically designed tests show how Visual Navigation tasks rely on geometrical properties of the scene. Although previous deep learning architectures have often made use of techniques such as pooling, our architecture does not use this. In fact, we show how our neural network significantly outperforms the state of the art of image classification in the particular task of visual navigation.

Table of contents

List of figures

Chapter 1

Introduction

This thesis reports a project with the goal to develop a complete indoor navigation system based only on visual information.
When we talk about indoor navigation systems, we refer to the systems that enable robots or vehicles to autonomously move in closed environments [1–5], like houses or factories, where the *Global Positioning System* [6](GPS) is not available. Even if this method is not as precise as others, it is important to divide navigation in two categories because of the nature of navigation itself: to be able to move from point to point it is absolutely essential for the system to know where it is. For this reason, indoor navigation mainly differs from the outdoor one for the techniques, technologies and algorithms used for localization. In fact these techniques are optimal for indoor situations because they do not have the restrictions of outside environments such as sunlight, range finder systems range and noise. [5, 3]
The state of the art in indoor navigation relies on lasers, sonars, depth cameras, and sensor-fusion system architectures. One of the key elements is being able to create a map of the environment, in order for the system to be able to perceive the environment and localize itself on the map. The most famous type of maps are grid-like occupancy maps, where the system's position is represented by the map's coordinates. To produce such maps mostly a range finder, like laser or ultrasound system, is used in combination with odometry and SLAM methods. However this system costs a lot. There are approaches that use visual information, and some can create such maps as well. [7, 8]
In our method, however, we want to remove the need of a laser system for the global localization of the robot. Moreover in the system proposed we do not use a grid-map approach. These two differences suggest the need to develop a novel approach, that we developed by taking inspiration from human navigation. This means a navigation system which will not rely on precise localization, while still being able to move safely in an indoor environment.

In order to show this in more detail, we will describe the general structure of navigation systems.

1.1 General Navigation System

In this section we show a generalized Navigation System, separating the different sub-systems and their relative tasks.

- *Localization:* this subsystem has the task to localize the robot in the environment, both globally and locally. It associates the inputs from the various sensors to the known map of the environment to localize the robot. On the other hand, the most common source for outdoor applications is the GPS.
- *Mapping:* this is an essential task for the localization system to work. It consists in creating a model, map, of the environment, where the localization system will project its position based on the surroundings.
- *Navigation:* this subsystem's goal is to decide the action the robot needs to perform in order to reach a destination. We can divide this sub-system into two components:
  - *Global Navigation:* this subsystem has the task to generate the path from the current position (given by the *Localization system*) to the destination, by having access to the Map.
  - *Local Navigation:* this subsystem has the task to ensure that the robot avoids obstacles in the path that are not present in the map. A typical example of these obstacles are moving objects, like chairs or people.
- *Sensor Fusion System:* this subsystem needs to read and translate the inputs from the various sensors into ready-to-use data for the other subsystems. It also needs to cope with sensor errors, mostly by combining different sensor readings (for example odometry and laser readings).

These subsystems are mainly separated to cope with the different tasks that are needed for the mobile robot to autonomously navigate safely.

1.2 Indoor Navigation Systems

The main difference between an outdoor Navigation System and an indoor Navigation System is the possibility to use GPS information as well. In fact for indoor situations, the GPS is not accessible; this creates the need for another source/algorithm for the robot to be able to localize itself in the environment. We can separate two situations: a hardwired navigation system, and a non-hardwired one:

- *Hardwired Navigation:* generally it means that there is a pre-establish path to follow. The robot will always follow the specific path, and the localization only needs to be on this path. This system has the advantage of being more precise when compared to non-hardwired systems, given its smaller position domain. But if the robot is outside the specified path, or this is blocked, the system can not work anymore till the situation is reconfigured. [9]

- *Non-Hardwired Navigation:* is a more general system. A map of the environment is needed, but the system will generate the optimal path every time from the current position to the destination. Generally this system is less precise, given the higher complexity of localizing itself; but it is more robust since the robot should be able to localize itself virtually everywhere, and, if possible, find the path to the destination.

In this thesis we will work with an indoor, Non-Hardwired Navigation System.

1.3 Proposed Navigation System

We propose a Non-Hardwired Navigation System only based on depth-camera information. This means that this system will work without the use of laser systems. In particular the depth-camera information will only be used in the Local Navigation System (LNS) for obstacle avoidance. The Localization will be performed only on pure-visual information.
The system proposed will be in general less precise and robust than a system that uses laser information for localization; but should drastically reduce the cost of the hardware, and is more similar to what humans do when they move in the environment.[10]
This means that this system is ideal to be used by cheap mobile robots; or with a future implementation of a locally more precise system, it could even be a valid solution for every type of Autonomous Navigation System.

1.4 Human Navigation

To be able to reach our destinations, humans strongly rely on visual information . Especially, we rely on specific landmarks like objects, doors and walls to be able to navigate in known indoor environments. In this case we can categorize our navigation system as a Local and a Global one. Reading directions and knowing through which rooms or hallways to pass to reach our goal can be considered as Global Navigation. Passing through doors and avoiding collisions with other people or objects can be considered as Local Navigation. Doing this blindfolded makes it difficult for us as well; this shows how necessary it is to have the ability to see the environment.
From these considerations we built our Navigation System.

1.5 Our Navigation System vs State Of The Art

Given the previous analysis on Human Navigation, we need landmarks in order to navigate. Those can be artificial or natural (already present in the environment). Using natural ones is normally very complicated given the generalization that these need; while artificial ones can be designed specifically to be unique, easy to recognize, and meaningful to the System. The disadvantage is that for using these, we need to modify the environment, and they need to be always visible by the robot. Changing or modifying these landmarks even a little, can stop the system from working correctly. For these reasons we decided to use artificial, non-expensive landmarks in the first phase, with the possibility to be removed later on. This will give all the benefits of Artificial Markers (later called Visual Markers), while removing the downside during navigation. More specifically these Visual Markers will be used to generate labels and meaning, from which the system can generalize and learn to use the Natural Visual Markers. The algorithms for recognizing and learning are proposed in chapter 3.2.
Taking inspiration from Human Navigation, we do not use a precise grid to navigate: by looking at our surrounding we humans do not know precisely (centimeters wise) where we are in a room; while we are still able to navigate and pass through tight corners or doors [10].

1.6 Research Questions

The first question is: are we able, with the current technology and algorithms, to create a working Localization System only relying on camera information?
This question immediately generates others: what algorithm should we use to extrapolate knowledge from the camera information? Are the standard computer vision algorithms suitable for this task?
In the recent time **CNNs** (Convolutional Neural Networks) have achieved impressive results in image classification problems and in general computer vision problems; will they be as good for Navigation Tasks? and why? In specific, how important is the *Geometrical Invariance* property for navigation/classification tasks?
In this thesis we will give an answer to these research questions, based on the results we will report.

Chapter 2

Robotic Operating System

For this project we use the Robotic Operating System [11] (*ROS*) as the main framework. In this section we will describe in general *ROS*, focusing on the main functions we are using. *ROS* is an Opensource operating system designed as a communication platform between multiple processes, sensors, and actuators. The system allows the control from very low level hardware (drivers for the engines, actuators, cameras, lasers, etc) to very high level decision making processes. This is achieved thanks to a system based on a server-client approach: every function, task and controller, is able to publish or read data on specific topics, allowing development through a module based approach.
It fully supports Python, while the main core is written in C++.

2.1 Alice

Alice is the name of the robot we use to perform all the experiments. It is a four wheels rectangular based robot, with two driving wheels, and two caster wheels, as can be seen in Figure 2.1. On the top of it, it has two infrared time of flight cameras, facing front and back. The front one can rotate over the vertical and horizontal axes thanks to two electric engines. Here we list the hardware specification of Alice:

- *Base and engines*: The base of Alice is the VolksBot RT3. This is a 13 Kg metal base, with two Maxon DC Motors, 150 W. It supports a payload of 40 Kg and has a top speed of 1.4 m/s.
- *Infrared 3D cameras*: Two identical Xtions are used for obstacle detection in the proximity of the robot. One facing backward with an angle of 15°, facing ground, where 0°is vertical; one facing forward, that can rotate over the Z and Y axis. Max camera RGB resolution is 1280x720, while depth point-cloud max resolution is 320x240.

Fig. 2.1 Robot Alice

2.2 Move Base

Move Base is the main component of the Navigation Stack. It is the combination of all the subsystems needed for Navigation. We will use only part of the system. This is the high level scheme; where the red line denotes the part of the system that we will not use (above the red line is NOT used; Figure 2.2) :

- *Global Planner* inputs: goal (destination), and Global Costmap. Output: optimal path.
- *Local Planner* inputs: optimal path. Output: command velocities. This is the output of the whole Move Base.
- *Odometry*: information on the rotation of the wheels, keeping track of the path
- *Laser*: using the adaptive Monte Carlo localization (AMCL) approach, and a saved map of the environment, localization is obtained thanks to the laser
- *Infra-red 3D cameras*: those are cameras with depth sensors based on infra-red. There are two of them: one on the back, fixed, and one on the front able to move.

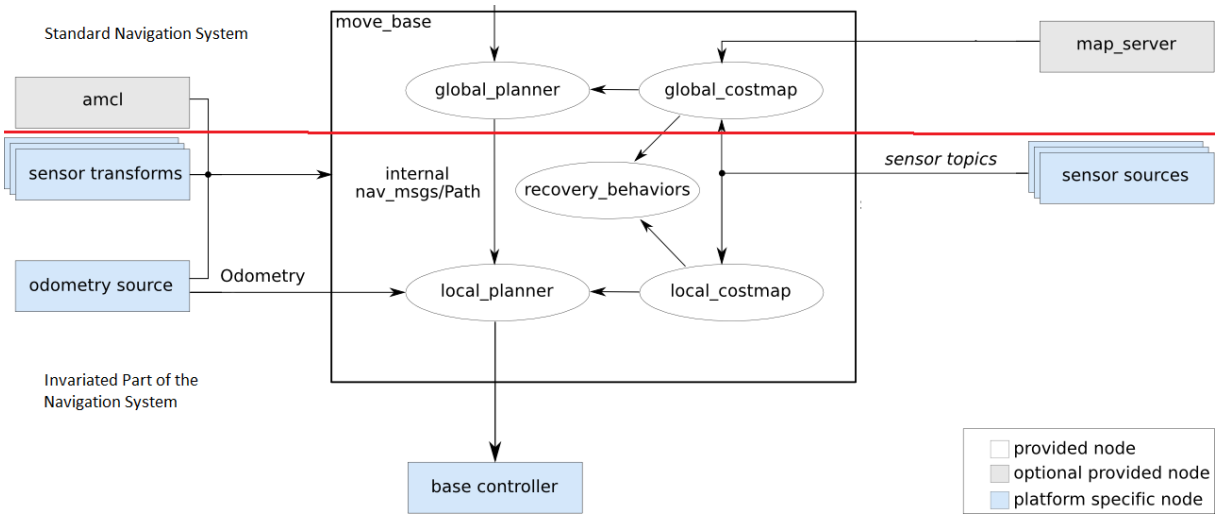


Fig. 2.2 Move Base Hight level scheme

Since in our project we decided to remove the laser sensor, and propose a new approach for *localization*, we need to substitute the *Global Planner, Global Costmap, AMCL* and *map server*. In fact, our goal is to have a *localization system* based only on visual camera information, that will function together with the existing *local planner system*.

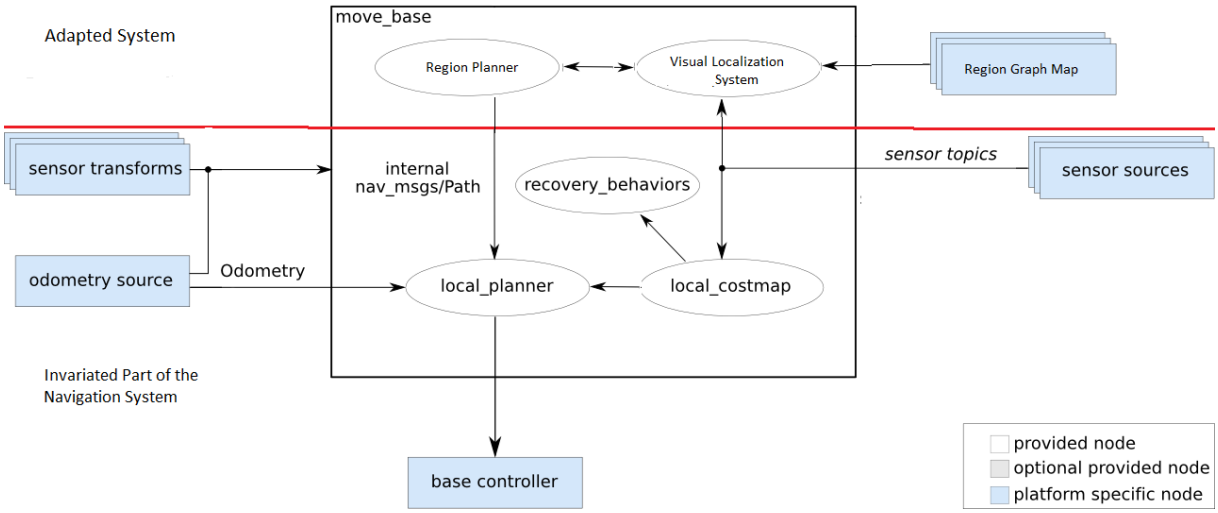


Fig. 2.3 Adapted Move Base Hight level scheme

This is the adapted version of Move Base that we propose:

- *Region Graph Map:* this is the map of the environment. This is a *Graph Map*, in contradiction to the previous *Grid Map*. Every node of it, that we will call *Regions*, represents a location in the real word, and it is defined using visual RGB information. The connection between regions is a geometrical distance that connects them together.

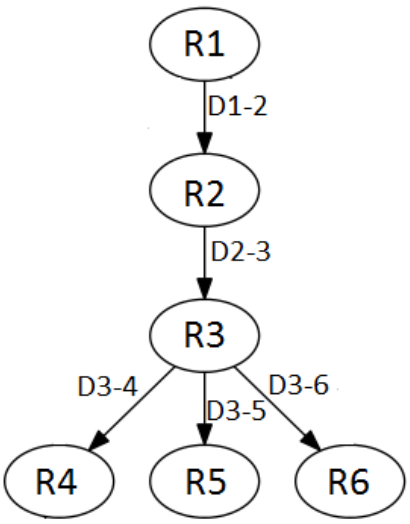
- *Visual Localization System:* this system receives as input the camera RGB information, and localizes the robot in the environment given the *Region Map*.
- *Region Planner:* this system computes the path from the current position to the destination, such as which nodes/regions to pass through, and translates in geometrical coordinates the current goal, in order to communicate with the *local planner*. This is necessary due to the different domains of the *global planner* and the *local planner*: a graph-based domain for the first, and a geometrical domain for the second.

2.3 Region Graph Map

When humans need to navigate, they know the route from the starting position to the destination. For example if they need to go from home to work, they know they need to exit the house, take the first street, turn onto a second street, etc, until they arrive to the work place. The same can be said if they need to go to the bathroom from the bedroom: exit the bedroom through the door, cross the hallway, pass through the bathroom door, and reach the destination.

Map Structure

We would also like to implement our visual-based navigation using human behavior, and the first step is to use an adequate map.



The map is based on the intermediate steps, or *Regions*. These *Regions* define a specific place in the environment, and the map consists of the geometrical connection between those regions. For example Figure 2.4 represent a possible *Graph Map*. During the mapping procedure, the *Regions* R1, R2, etc. have been defined using the visual RGB information; and the connection between these have been mapped with the geometrical distance from each other thanks to *Odometry* information. In fact to go from *Region 3* to *Region 6*, we need to move 2 meters in a certain direction.

Fig. 2.4 Graph Map

To cope with the orientation problem, every *Region* is defined by several *Nodes*. These *Nodes* represent a certain rotation, relative to the region itself.

Figure 2.5 is the representation of a region and its nodes. Each of these nodes are separated by a rotation of 45 degrees. Assuming that the robot is in the center, it will define one node for every angle. During the mapping process the robot will record a node: first it examines *Node 0*; next it will rotate 45° and scan *Node 45*, and so on until it reaches its starting orientation. These nodes, and their relative orientations, define a region.

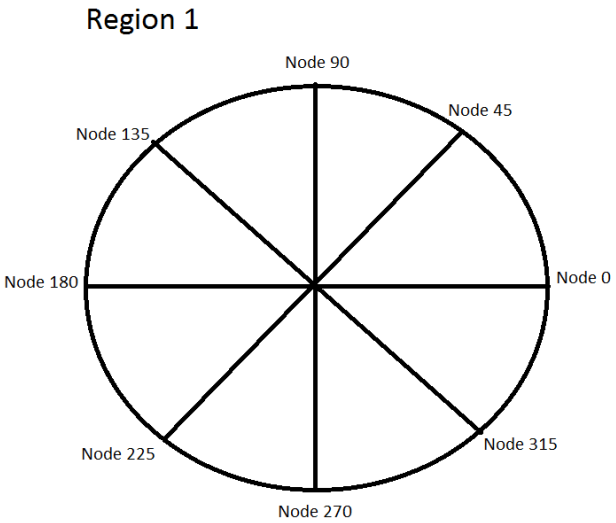


Fig. 2.5 Nodes orientations in a Region

The **Nodes** are the main components of the **Regions**, and they need to be unique based on the visual information received from the RGB camera. The input to generate a node is the output of our *Scene Recognition System*: a list of the top five most likely *scenes*.

Given a frame, the *Scene Recognition System* outputs a histogram of the top 5 scenes that can match the picture received. The **Node** is then defined by the combination of 3 histograms, each representing what the robot detects in front of it, 90° on the left, and 90° on the right. This will give the node the property to be unique both given its orientation, as its absolute position.

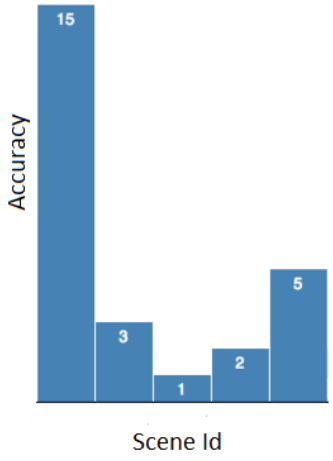
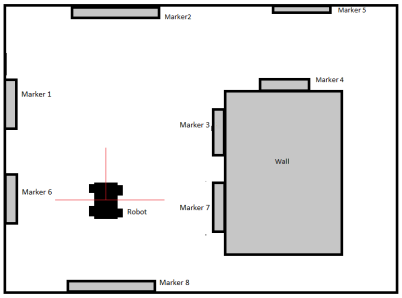
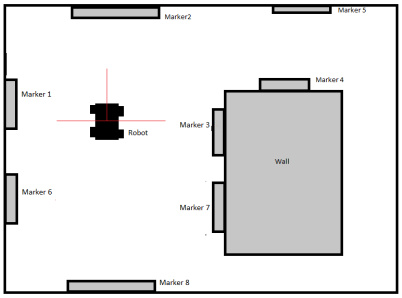


Fig. 2.6 Top 5 likely scenes

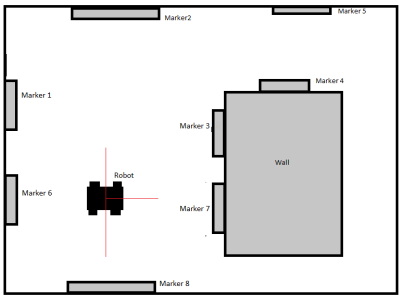
The **Node** is unique, and at the same time invariant to small positional or rotational changes of the robot. This can be seen in the next figure:



(a) **Node**: 0; **Region**: 0



(b) **Node**: 0; **Region**: 1



(c) **Node**: 270; **Region**: 0

Fig. 2.7 Robot positions in **Node** and **Region**

As we can see from Figure 2.7, small movements of the robot will lead to the same **Node** as long as the visual surroundings do not change enough. At the same time, by definition of **Region** and **Node**, moving forward enough will lead the robot into another **Node** in a different **Region**; while rotating on the spot, will lead to a new **Node** in the same **Region**.

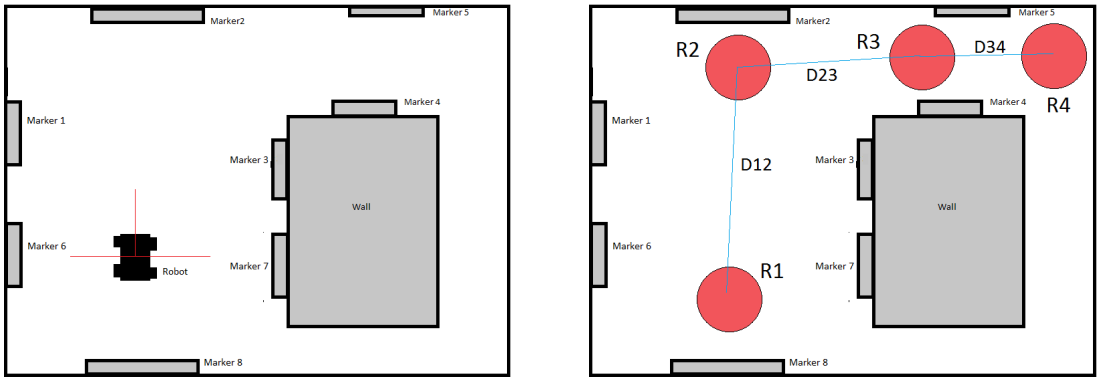
2.4 Mapping

Now that we have explained the structure and the various elements of the Map, we will show how this structure can be created during the mapping procedure. We assume that the *Scene Recognition System* is already operational.

Let's consider the robot is in the starting position, in Figure 2.7. The first operation is to define the current **Region**, with the *Region_Registration* function:

input :
Region=() `// generating an empty region`
for *(current-angle) in (0,45,90, ..., 360)* **do**
  `RotateRobot(`*current-angle*`) // rotate the robot to the desired relative angle`
  CurrentNode=() `// generating an empty node`
  for *(orientation) in (front, left, right)* **do**
    `RotateCamera(`*orientation*`) // this function rotate the camera to the desired angle`
    histogram ← `DetectScene() // this function call the Scene Detector System, and return the histogram of the top 5 scene detected`
    CurrentNode ← `SaveHistogram(`*histogram, orientation*`) // this function saves the histogram in the node at the proper position: front, left, right`
  end
  Region ← `SaveNode(`*CurrentNode, current-angle*`) // this function saves the current node in the region with the current robot's angle`
end
Return(Region)

Algorithm 1: Region_Registration



(a) Example Map (b) Example Regions

Fig. 2.8 Possible regions in a map

This will generate the first region (**R0**) in our map. The next step is to move the robot to another location. During the movement the robot will save the relative position from the starting region using the *Odometry* information. Once the new position is selected, the *Region Registration* function is called to generate **R2**. The geometrical connection between **R0** and **R1** is saved in the *Region Graph Map*. Each connection is saved and registered based on the following parameters:

- *Starting Region*: the region we start from (**R_x**)
- *Starting Node*: the node in **R_x** we start moving from (**N_x**)
- *Distance*: the distance in meters saved by the *Odometry* from **R0** to **R1**
- *Arriving Region*: the region of the destination.

Following this procedure we will map the whole area of interest. It is important to notice that the regions do not have to be close together, but just in meaningful places: i.e. beginning and end of an hallway, before a turn, in a cross-section etc. Given this property we can easily map vast areas with only few relevant regions: this is very similar to what we humans do to navigate.

Chapter 3

Visual Markers

In this section we will discuss the implementation, recognition, and creation of custom *Visual Markers* (VMs) that have been used in the project.
After having researched already existing types of VMs [12, 13], as can be seen in Figure 3.1, we decided to implement a custom type. All the VMs already existing have the common objective to hold information: some a lot, some a few; some were designed to be rotation invariant, some orientation invariant; and each of them were designed to accomplish their task in the best way. For this project we needed very specific VM properties :

- *1. Information*: our type of VM only needs to save a number, an Identification Digit (Id). No other explicit information is needed.
- *2. Prospective Invariant*: since the VMs are going to be applied on walls or objects, and the camera reading can be made with any angle, we need the VM to intrinsically hold information of the relative prospective between camera/VM.
- *3. Robust recognition*: given the complexity of the whole system, we need a design that eliminates all possible false positives, on the other end false negatives do not influence much the performance of the full system.
- *4. Recognition from distance*: while most of the VMs consider an optimal distance from detector to VM, for this project we need to be able to detect the VM as far as possible. This combined with point 3, gives a strong restriction for the design of the VM.

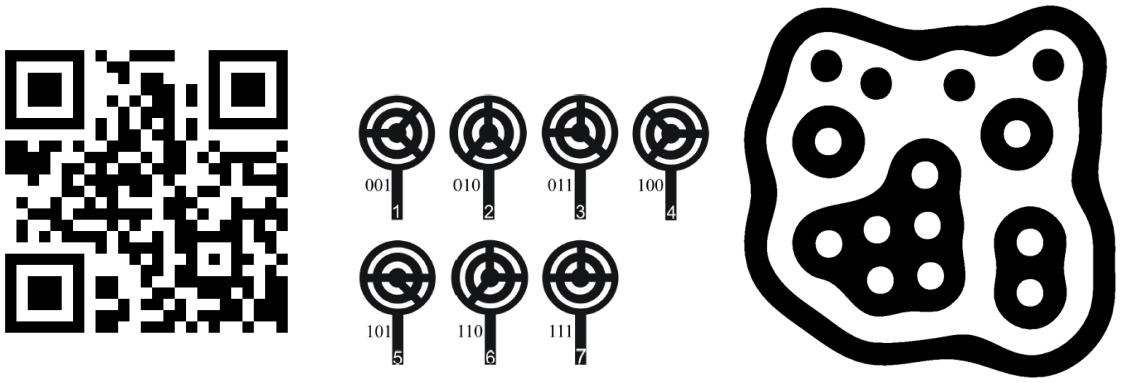


Fig. 3.1 Examples of various VMs. Starting from right: QR code, LARICS, reacTIVIsion

3.1 Custom Visual Marker

After analyzing the necessity of the system, and looking at existing types of VMs, we designed a custom VM that can be seen in Figure 3.2. In the process of designing it we focused on the points we previously described. Here we present the solution point by point:

- *1. Information*: while the 3 concentric squares are used only for identification, the central area holds the Id. This area is divided in 9 squares, each can be empty (white) or full (small square) representing the binary values 0 or 1. This allows a total of 2^9 possible Ids.
- *2. Prospective Invariant*: thanks to the square properties (same length borders, and 90 degree angles) it's always possible to determine the prospective between camera and VM.
- *3. Robust recognition*: to obtain the most robust recognition, the VM has 3 concentric squares, with equal difference in border length. This creates a very unique configuration, almost impossible to be repeated by accident by random lines in a picture.
- *4. Recognition from distance*: since the whole VM is fairly simple, there are very few features needed to distinguish to correctly identify this marker. This greatly influences the distance from which it is possible to recognize the VM.

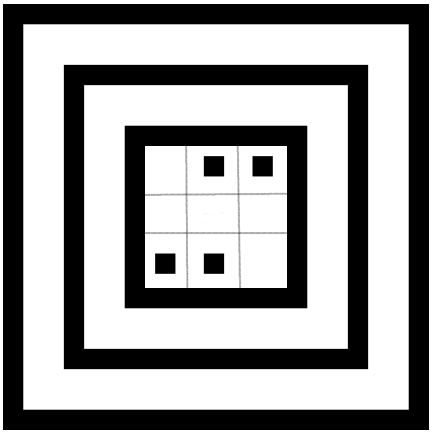


Fig. 3.2 Custom VM

3.2 Visual Marker Identification Process

In this section we will describe the identification process used by the system to identify a VM in a picture. The next graph will show the structure of the Identification process.

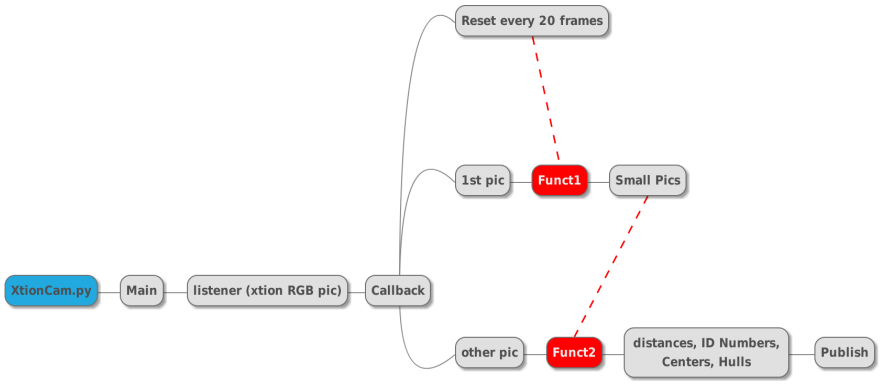


Fig. 3.3 Identification process

The graph in Figure 3.3 represents the whole Identification process: from the input (RGB picture) to the output (Publish). This system runs as services connected to ROS, and subscribes to the Xtion RGB picture's topic. Every time a new frame is available from the camera, this process will elaborate and publish the information of the detected VMs.
The first function, *Callback*, checks if this is the first frame received, and accordingly, calls

Function1 or *Function2*. Every 20 frames the counter is reset.

Fig. 3.4 Function1

Function1, described in Figure 3.4, generates multiple binarized pictures, using $\alpha * T$ thresholds; where T is the average ink density of the frame, and α is a value from $[0.1 - 2.0|step = 0.1]$. This is done to be sure to always find the optimal binarization threshold for every VM. In fact using only one threshold is never optimal in case there are multiple VMs in the picture, in different light condition. This process is computationally expensive, but it's only applied once every 20 frames.
Once all the binarized pictures are computed, the *Find Qrs* function is called, and this finds all the VMs in all the pictures. Once the VMs are found, we define a portion of picture around the VM with $A_{pic} = 1.5 * A_{vm}$, where $*A_{vm}$ is the area of the external square of the VM.
For every VM found in all the binarized pictures, a specific area, or sub picture, is generated: this is the output of *Function 1*.

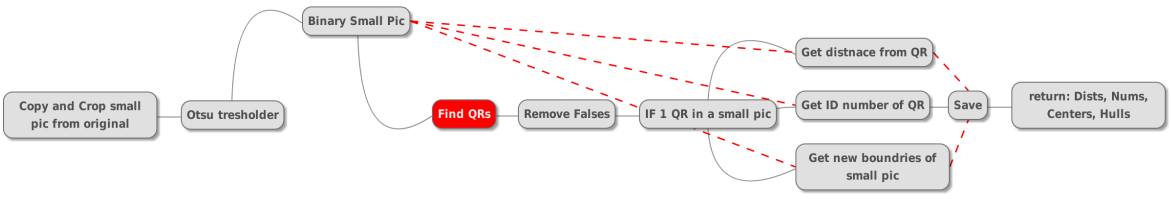


Fig. 3.5 Function2

Function2, described in Figure 3.5 takes as input the *small pictures* of *Function 1*. For each of those, it uses the *find QRs* function to find the VM inside the *small picture* and outputs the coordinates of the outer square of the VM found. Then for every unique VM found, it calculates the distance to it, the ID number, and updates the area to look in the next

frame. In order to do this, it finds the center of the VM and computes the coordinates of the new *small picture*. The output is a list of: distances, ID numbers, and the updated *small pictures*. This last list will be taken as input, togther with the next frame, at the next loop. Next is the *Find_QRs* code:

input : Picture; small pictures coordinates

frame ← `RGBtoGray(`*frame*`) // convert frame from color to gray`
for *small-pic-coordinates* ∈ *small pictures* **do**
  `// for every small picture found: extract small picture`
  small-pic ← `TakeFromFrame(`*small-pic-coordinates*`);`
  small-pic ← `OtsuBbinarization(`*small-pic*`) // convert gray to binary`
  squares ← `FindSquares(`*small-pic*`) // find VMs outer square`
  if `length` *(squares)==0* **then**
    small-pic ← `RemoveBlur(`*small-pic*`) // compensate horizontal Blur`
    squares ← `FindSquares(`*small-pic*`);`
  end
  `RemoveFalse(`*squares*`) // check geometrical feature of real VM`
  if `length` *(squares)==1* **then**
    `// we proceed only if there is one VM per small picture`
    dist ← `GetDistance(`*squares*`) // calculate the distance from VM`
    if *dist* $<$ 5000 **then**
      `// if the distance to the VM is less then 5 meters`
      ID-num ← `GetID(`*squares*`) // read the ID number of VM`
      small-pic-coordinates ← `CheckAndResize(`*squares*`) // given the identified VM, centers it in the new small picture`
    end
  end
end
Return (VMs IDs, Distance to VMs, Coordinates (Hull) of VMs)

Algorithm 2: Find_QRs

Here you can see the pseudo-code for the *Find_Squares* function:

input : small-pic
(contours, hierarchy) ← `FindContours(`*small-pic*`) // this function return all the closed contours in a picture, and their hierarchy`
for *(cont, hier) in (contours, hierarchy)* **do**
    valid-cont ← `IsAVm(`*cont, hier*`) // this function check if the contour is the outer square of a VM`
    if *valid-cont*==*True* **then**
        valid-contours ← *cont*
    end
end
Return(valid-contours)

Algorithm 3: Find_Squares

The function *Is_AVm* checks the correct nesting of contours, for representing our custom VM:

input : cont, hier
if `IsASquare(`*cont*`)` ==*True* **then**
    (child1, parent1) ← `GetHier(`*cont, hier*`) // given a contour check in the hierarchy for children and parents (contour inside and to which is inside`
    if *child1 and parent1* **then**
        (child2, parent2) ← `GetHier(`*child1, hier*`)` ;
        if *child2 and parent2* **then**
            (child3, parent3) ← `GetHier(`*child2, hier*`)` ;
            if *child3 and parent3* **then**
                (child4, parent4) ← `GetHier(`*child3, hier*`)` ;
                if *child4* **then**
                    Return True;
                end
            end
        end
    end
end
Return False

Algorithm 4: Is_AVm

Thanks to the combination of these functions we are able to identify a custom VM inside a picture or a frame of a video.

3.3 Visual Marker ID

Every VM has a unique ID, that identifies itself. The ID is red following this simple scheme:

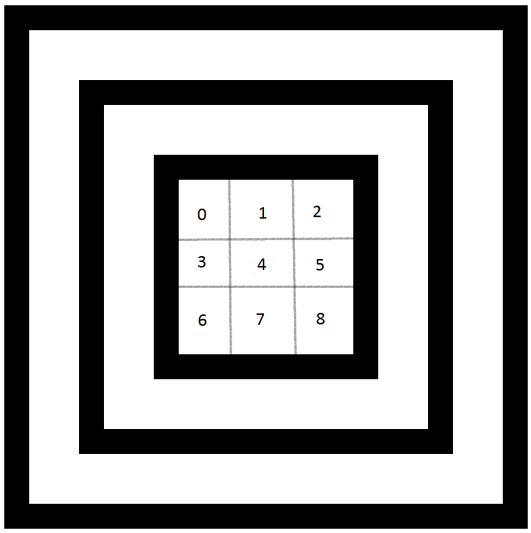


Fig. 3.6 VM Identifier

As can be seen in Figure 3.6, every portion of the center square is interpreted as a binary value. The presence of ink or not determines the value 1 or 0. The ID reader will calculate a number given the function:

$$ID = \sum_{i=0}^{8} 2^i * j_i \tag{3.1}$$

Where j_i can be 0 or 1, and refers to the portion inside the inner square.

Chapter 4

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are one of the latest, and most successful, Neural Network architectures. This type of architectures started to become extremely popular in 2012 [14–18], and started to outperform all the other architectures, giving them a strong focus from the scientific community. In particular CNNs perform really well with large and geometrical input: for these reasons they find a vast utilization in the Computer Vision field. While the convolution layers can be applied on 1D, 2D and 3D inputs, we will focus on 2D: the width and height of a picture.

4.1 Convolution Layers

The key component of CNNs is the implementation of specific layers, that apply convolutional operations on the input: these are called *Convolution Layers*.
A convolution operation is defined, in mathematics, as an operation over two functions. In the computer vision field, this means applying a filter over an image: where one function is the picture, and the second one is the filter itself. In the ANN field the principle is very similar; instead of a filter, a kernel is defined. Normally the kernel size is much smaller than the picture, and this is moved over the whole input. This can be seen in figure 4.1. While the kernel values are defined a priori, usually randomly, the randomly initialized, kernels are modified during the training procedure. It is important to notice that every kernel will have the same amount of trainable weights as the number of parameters: this means that a $3x3$ kernel will have a total of 9 weights. Thanks to this the number of weights is several times smaller compared to a *fully connected* layer.
When we apply the convolution we need to decide how to behave at the borders of the input: we can decide to always have the whole kernel "inside" the input matrix, resulting in a reduction of the input equal to half of the neighbor pixels (a $3x3$ filter will reduce one

pixel at the borders, losing a total of two pixels per dimension), or to make the center of the kernel central pixel to on the border itself. This last method does not change the output resolution. An example of the first case can be seen in figure 4.1, while an example of the second approach is visible in figure 4.3.

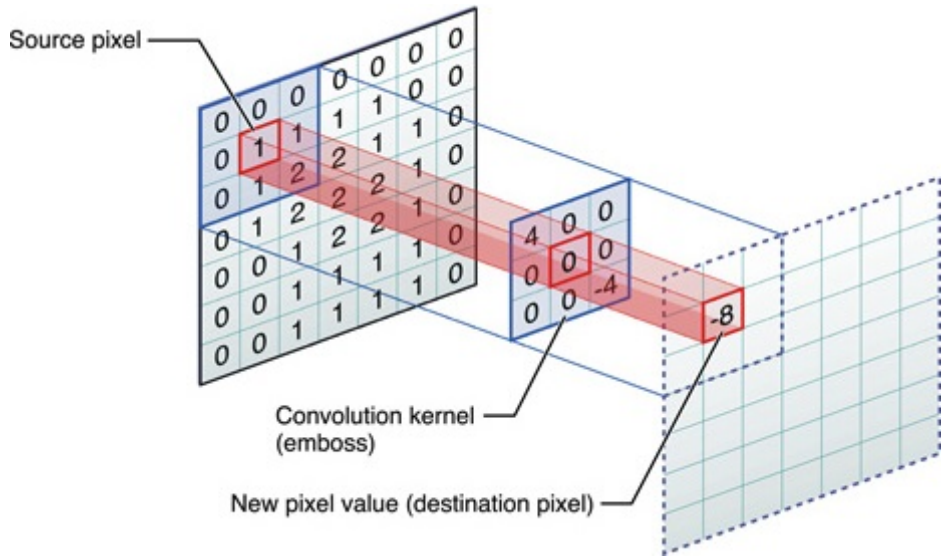


Fig. 4.1 Convolution operation

Also by deciding how much, or how many pixels/cells to move the kernel over the picture, we can reduce the original size of the input: considering the central cell, and its neighbors cells, we can move it one pixel or more. This will determine the output's size.

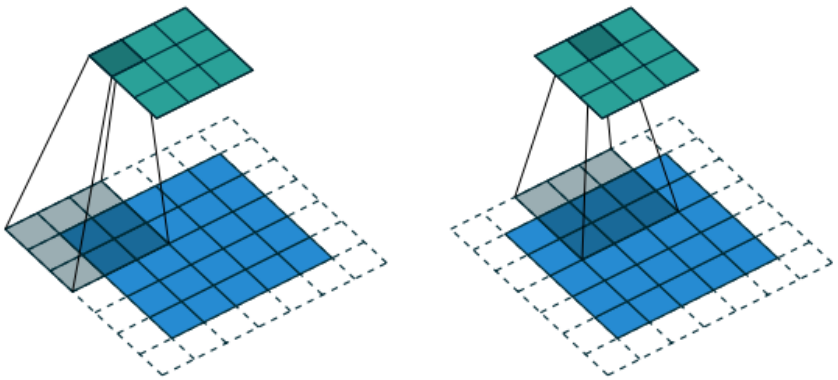


Fig. 4.2 Convolution with $stride = 2$

By looking at figure 4.3, we can clearly see the different results obtained by moving the kernel over the picture with more than one pixel: the distance in pixels between the current

cell and the next is called *stride*. This technique is used to greatly reduce the size of the input: considering an input of size WxH and $stride = s$, the output will have dimension $W_1xH_1 = \frac{W}{s}x\frac{H}{s}$. As an example applying a filter of $3x3$ with $stride = 2$ on a $128x128$ input, we obtain an output of $64x64$.
Reducing the input size also means reducing the intrinsic information of it. To cope with this, it is possible to apply many filters on the same input: this will produce as many outputs as the amount of different kernels used. These outputs are collected in a new dimension called *Depth* or *channels*.

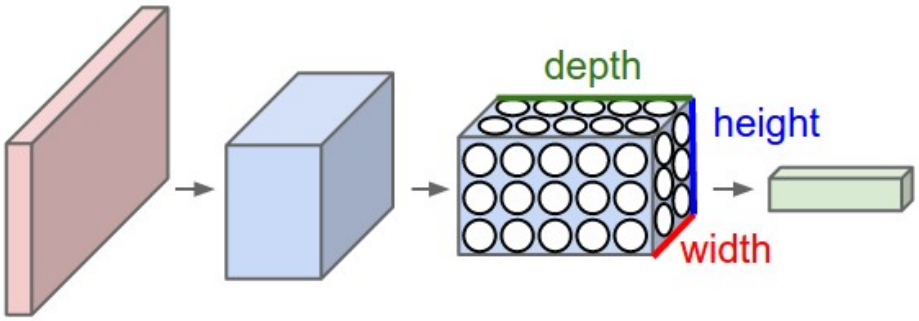
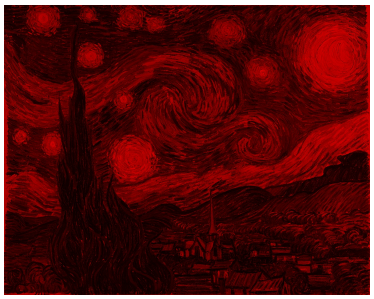
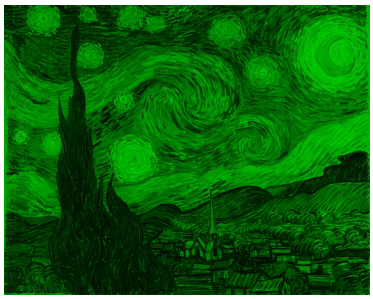


Fig. 4.3 Convolution with $stride = 2$

To better understand this we can think of pictures. A color picture can be represented with RGB or HSV values. This means that every pixel of the picture is defined by three features: Red, Green Blue for RGB, or Hue, Saturation and Value for HSV. We can also see this as three pictures on top of each other, where every picture, or *channel*, refers to the same picture but has different intrinsic information. This can be seen in figure 4.4.
In the same way, when we apply different kernels on the same pictures, they will specialize on recognize specific information, like the **R** channel contains the Red color information of the image. We can see from figures 4.4d, 4.4e, 4.4f that in a single convolution layer, the different kernels will focus on learning specific information, very close to the original image. Going deeper, and applying more convolution layers in series, we will analyze the data in an always higher and more abstract level. This can be seen in figures 4.4g, 4.4h, 4.4i. This represents very high level information: a very low size in terms of width and height ($15x15$) but with many different channels (24). At this level we transformed a $300x300x3$ (3 channels = RGB) size input in a 3D shaped tensor of size $15x15x24$ that contains really high level information, even reducing the actual tensor size. Thanks to their relative low (compared to *Fully Connected* layers) computational cost, it is possible to have many in series, and to go very deep in the information hyperspace.



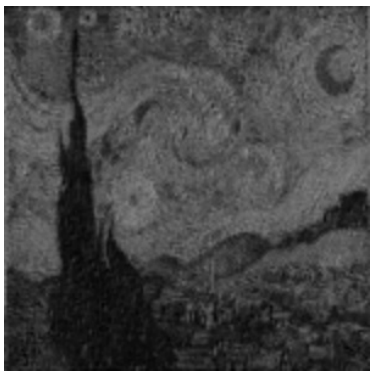
(a) **R**ed channel



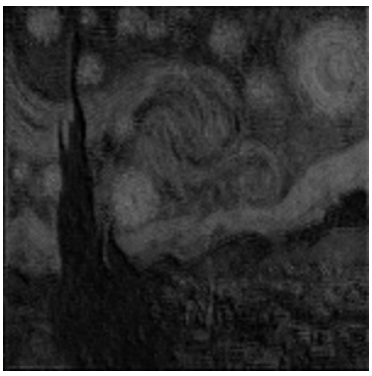
(b) **G**reen channel

(c) **B**lue channel

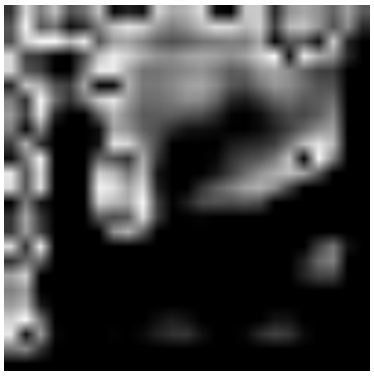
(d) First channel



(e) First channel



(f) First channel



(g) Last channel

(h) Last channel

(i) Last channel

Fig. 4.4 Different Channels

4.2 Activation Functions

Given the particular operations applied in Convolution layers, it is important to use proper activation functions. For example, a classical *Sigmoid* activation function is not ideal for this type of network. There are two main reasons for this:

- **Vanishing Gradient**: during back-propagation, the derivative of all the activation functions is needed to calculate the weight adjustments. Sigmoid's derivative, as can be seen in figure 4.5, has a very small value. If many convolution layers are stuck together, or if it is a very deep network, the back-propagated error at the end will be very low, almost zero.
- **Computational Complexity**: considering a fully connected layer, the sigmoid activation is computed once per every unit. If we consider convolution layers, the filter passes through the whole picture, and depending on the stride and padding, it can go over every single pixel. In this case the sigmoid operation would be computed once per every pixel, and every channel: as an example, cosider a $300x300$ color picture as input the first convolution layer will compute $300*300*3 = 270000$ times the sigmoid function. For this reason a more easy to compute and optimized activation function is needed.

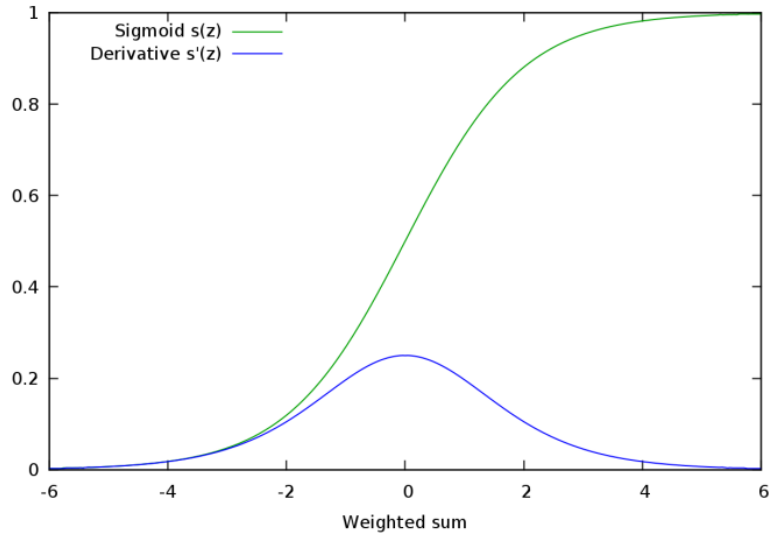


Fig. 4.5 Sigmoid function and first order derivative

Given the considerations mentioned a new activation function needs to be considered. But at the same time using a simple linear multiplication is not effective, since the network will not have the ability to approximate non linear functions: it will just be a complex, *linear*

transformation. To be able to solve non-linear problems, or to approximate these, we need non-linear activation function. The one that is easiest to compute, and yet non linear, is the **ReLU** activation function. This can be seen in figure 4.6.

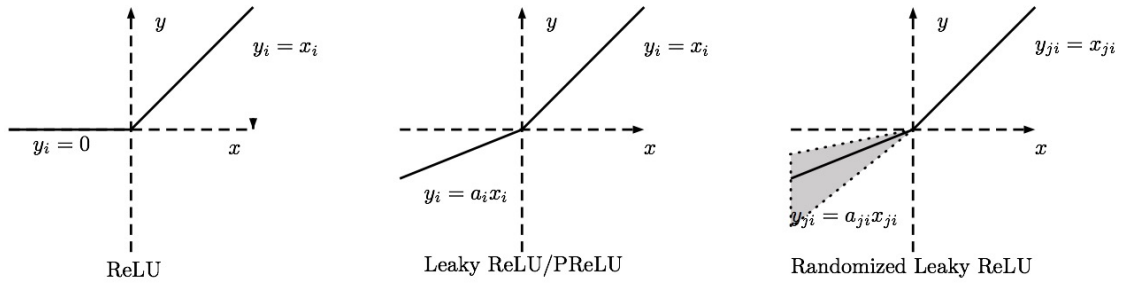


Fig. 4.6 ReLU and Similar activation functions

This is ideal given its computational simplicity (it is just a *max* operation), and has the non-linearity property needed. More complex, yet cheap to compute, activation functions can be used: **LRelu**, or Leaky Relu, performs a max operation over the input and a smaller linear function to cope with the negative part of the input. $LRelu = max(\alpha * x, x)$ with $0 < \alpha < 1$. This operation consists of a dot multiplication and a max function: still very computational cheap. It is also possible to add some randomness in the negative part of the input to force a more stable network.

Chapter 5

Scene Recognition

Once the system is able to detect and recognize Visual Markers, we are able to create in a fast and efficient way a *Data Set* of the environment we want to navigate in. Thanks to *VM Recognition System*, we are now able to immediately associate to every scene in our environment (by positioning different VMs in specific and well visible places) a label. Thanks to this we are able to simply record a video (possible by the robot Point Of View), and later analyze, frame by frame, all the VMs present in the scene; and save every frame based on its label (VM's id number). This will result in a perfect *Data Set*, easy and fast to create, perfect for any classification algorithm.
Once the data set is created, we can train the system to recognize the scenes; once the system is trained we can remove the VMs from the real scene. Our system will be able to detect the scenes of our environment, giving it enough visual landmarks to localize itself.
In this section we will present and compare two different methods used to achieve our goal of scene recognition. The results of real world experiments will be show.

5.1 Data Set Enhancement

The data set used to train and compare all the experiments contains 1408 unique pictures, generated from various recordings of the test environment, divided in 20 classes. This means the average number of pictures per class is 352; and the data set has a *Standard Deviation* of 174.
On top of those original data, we augmented the data set using **HSV** color space modifications. We augmented the data set to a total of 7040 pictures.

5.1.1 Hue Saturation Value (HSV) color space

HSV is one of the most used color spaces. The color information is divided in three channels:

- **Hue (H)**: this channel refers to the closed pure color of the current color. This is invariant to tints, tones and shades. The value, normalized from 0 to 1, refers to the color wheel where: 0 is red, $1/6$ is yellow, $1/3$ is green, and so forth around the color wheel.
- **Saturation (S)**: this channel refers to how white the color is. Pure color has *saturation* of 1, while white has a saturation of 0.
- **Value (V)**: this channel refers to the lightness of the color. A value of 0 refers to black, and 1 to the pure color.

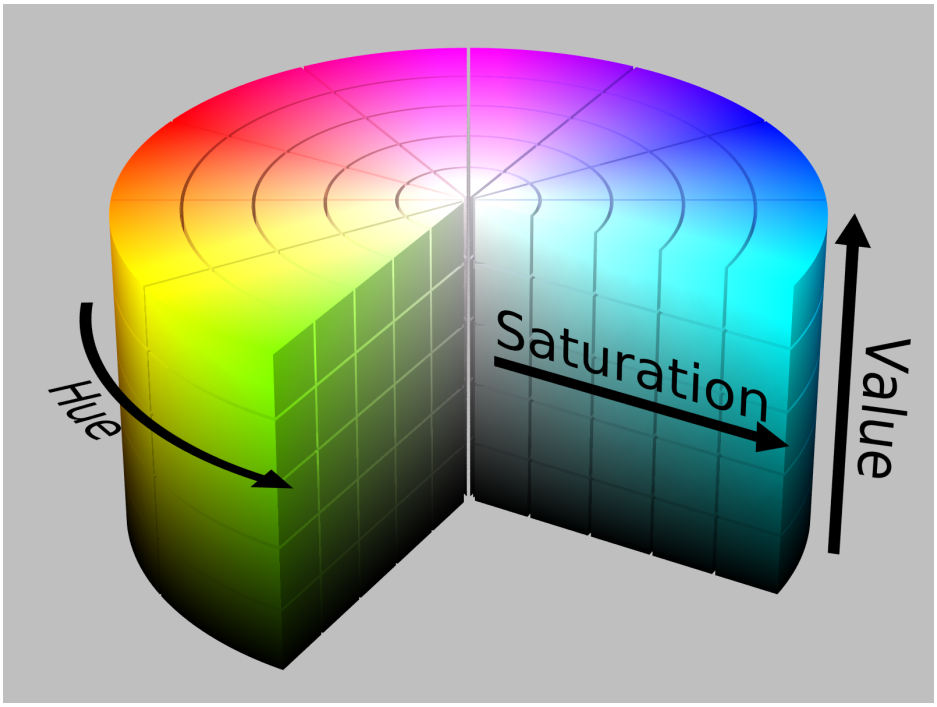
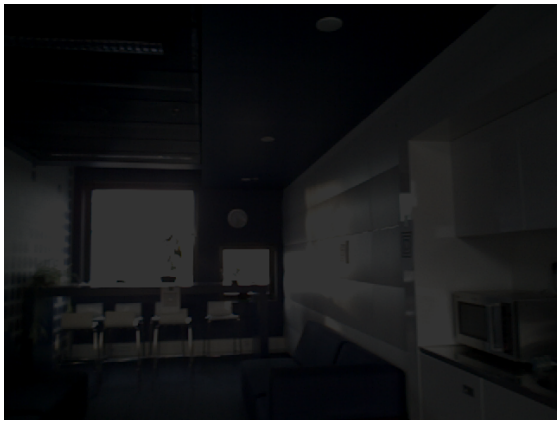


Fig. 5.1 HSV color space

Thanks to this colorspace it is easy to change the lightness of a picture, with well resemblance to actual real word light conditions. We enhanced our data set creating 4 copies of every picture, multiplying their *Value*, or lightness, by $0.2, 0.4, 1.4, 1.6$. In case the resulting value would exceed the max value of the channel, we set it at the max value.

Fig. 5.2 Original Picture

(a) 40% lightness



(b) 20% lightness

(a) 140% lightness

(b) 160% lightness

Fig. 5.4 Data Set enhancement in HSV colorspace example

5.2 Bag of Visual Words

The first approach to scene recognition uses Bag of Visual Words [19](BOW) with the Scale Invariant Feature Transform (SIFT) [20].

5.2.1 Scale Invariant Feature Transform

The *Scale Invariant Feature Transform*, or **SIFT**, has been use extensively in computer vision, especially in scene recognition, object detection, etc [20].
The first step in the **SIFT** algorithm is the key-points detection: this is done by using a cascade filtering approach to find candidate location that are then examined in details. Detecting locations that are invariant to scale change of the image can be accomplished by searching for stable features across all possible scales. *Gaussian function* is the scale-space kernel. Therefore, the scale space of an image is defined as a function, $L(x,y,\sigma)$, that is produced from the convolution of a variable-scale Gaussian, $G(x,y,\sigma)$, with the input image, $I(x,y)$:

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y) \tag{5.1}$$

where $*$ is the convolution operation in x and y, and

$$G(x,y,\sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{5.2}$$

To efficiently detect strong and scale invariant key-points, we will look at scale-space extrema in the difference-of-Gaussian, $D(x,y,\sigma)$, defined as follow:

$$D(x,y,\sigma) = (G(x,y,k\sigma) - G(x,y,\sigma)) * I(x,y) = L(x,y,k\sigma) - L(x,y,\sigma) \tag{5.3}$$

This function is pretty efficient to compute, since, after the smoothed picture L is compute for all the scales, D can be computed simply by image subtraction.

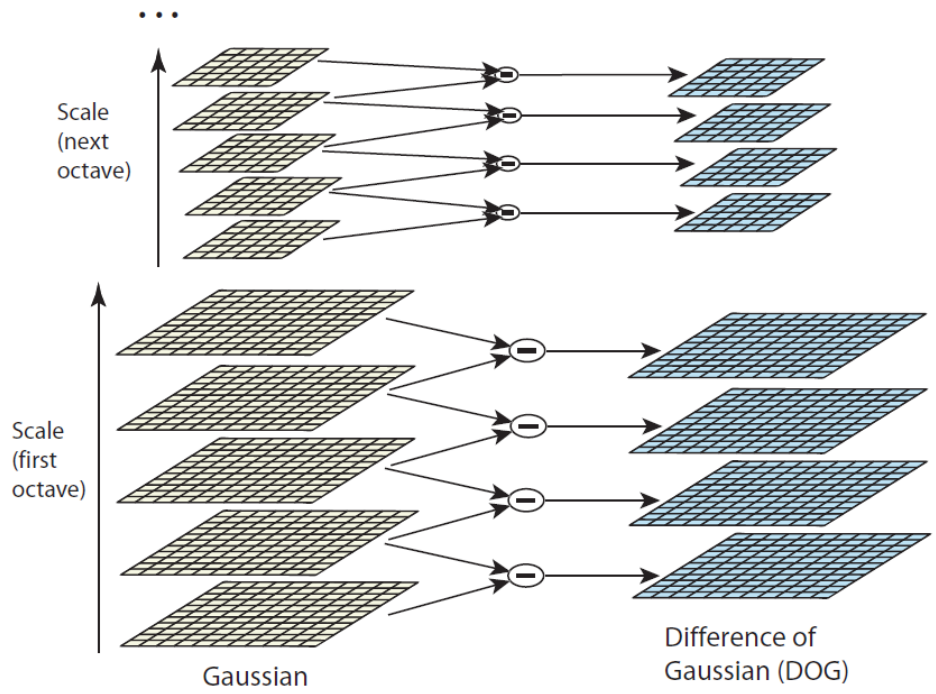


Fig. 5.5 Difference of Gaussian

The difference-of-Gaussian function provides a close approximation to the scale-normalized Laplacian of Gaussian, $\sigma^2\nabla^2 G$. Moreover a factor of σ^2 is needed for true scale invariance [20]. To better understand the relation between D and $\sigma^2\nabla^2 G$, we look at the heat distribution function:

$$\frac{\partial G}{\partial \sigma} = \sigma\nabla^2 G \tag{5.4}$$

From this we see that $\nabla^2 G$ can be computed using the finite difference of nearby scales $k\sigma$ and σ:

$$\sigma\nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma} \tag{5.5}$$

and finally:

$$G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2\nabla^2 G \tag{5.6}$$

This shows that when the difference-of-Gaussian function has scales differing by a constant factor, $(k-1)$ in this case, it already incorporates the σ^2 scale normalization required for the scale-invariant Laplacian.

Now to find the local maxima and minima of $D(x,y,\sigma)$, each sample point is compared to its eight neighbors in the current image and nine neighbors in the scale above and below:

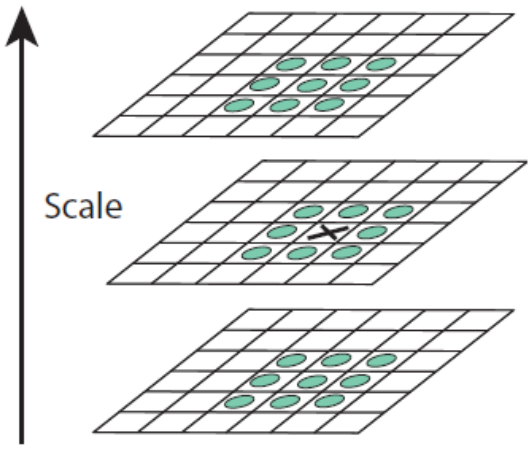


Fig. 5.6 Difference of Gaussian

It is selected only if it is larger than all of these neighbors or smaller than all of them. The cost of this check is reasonably low due to the fact that most sample points will be eliminated following the first few checks. Once the key-point is accepted as a candidate, the step is to perform a detailed fit to the nearby data for location, scale, and ratio of principal curvatures. This is done by using the Taylor expansion (up to the quadratic terms) of the scale-space function, $D(x,y,\sigma)$, shifted so that the origin is at the sample point:

$$D(\chi) = D + \frac{\partial D^T}{\partial \chi}\chi + \frac{1}{2}\chi^T \frac{\partial^2 D}{\partial \chi^2}\chi \tag{5.7}$$

where $\chi = (x,y,\sigma)^T$. To find the extremum, $\hat{\chi}$, we take the first derivative of this function with respect to χ and set it to zero, and we obtain:

$$\hat{\chi} = -\frac{\partial^2 D}{\partial \chi^2}^{-1} \frac{\partial D}{\partial \chi} \tag{5.8}$$

so we can now calculate:

$$D(\hat{\chi}) = D + \frac{1}{2}\frac{\partial D}{\partial \chi}^T \hat{\chi} \tag{5.9}$$

and all the extrema with a value of $|D(\hat{\chi})| < 0.03$ were discarded (assuming the pixel values are in range [0,1]). This is done to remove all the extrema with a low contrast.

Now that we have found strong key-points, we will compute the descriptors for each of them: for each key-point, we take a 16x16 cell around it, we divide it in four 4x4 cells. Then we apply the *Histogram of Oriented Gradients* transform to each cell, using 8 bins: meaning that every SIFT point will have as descriptor a 128 ($= 4*4*8$) feature long vector:

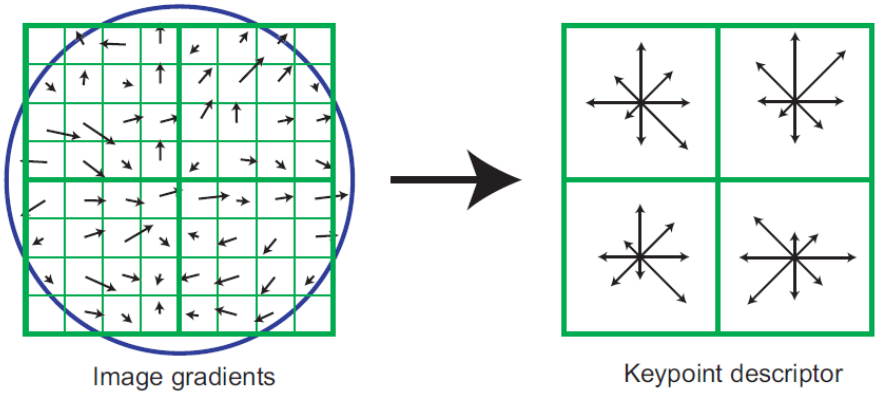
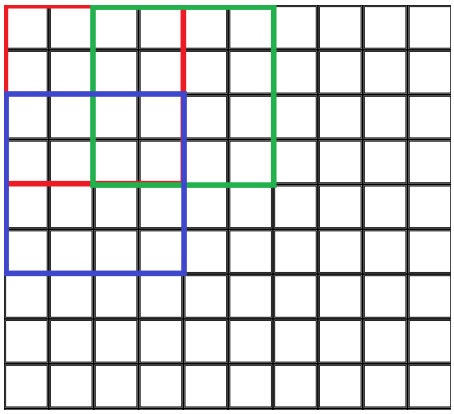


Fig. 5.7 Keypoint descriptor

In this picture we see the feature transform using the *Histogram of Oriented Gradients* (HOG)[21]. The resulting vector contains the features we will use for the *BOW* clusters.

5.2.2 BOW with Adapted SIFT

Now that we have defined the features we will use for BOW, we decided to use an adapted, or partial, SIFT algorithm that is more appropriate for our case. The input image will always have a fixed size of $1280*720$ pixels, since all of our pictures or frames come from the same robot camera.
Instead of using the *SIFT* detection algorithm previously described, we will use a fixed amount and position of key-points; this will generate a higher number of total features per picture, and will remove the computation time of finding the SIFT points using the classic method[sift].



Since we will use the standard *SIFT* key-points description method, where each point is analyzed by a 16x16 cell grid, we decided to take the centers with a distance of 8 pixels. This means that two consecutive key-points will include a total space of 32 pixels, with 50% overlap. This can be seen in the picture on the left: the red, green, and blue squares represent 3 descriptors of 3 key-points.

In this way we will extract $160x34$ ($1280/8 = 160$; $720/8 \approx 34$) cells from every pictures. Since every cell is described by 128 bins, we will extract $5440x128$ feature per picture.

After this process the BOW will generate N cluster points (decided by us at priory), and cluster all the features from all the pictures. Those *cluster* points will be the reference for the next step. On top of BOW we will use a *Neural Network* (NN) [cit], that will us as inputs an histogram of linear distances to those N cluster points; and as label the scene name.The NN will have as many inputs as the number of BOW clusters.

5.2.3 The Neural Network for the BOW

We use the Neural Network to associate the BOW feature (distances to cluster points) to their respective labels. To do so, we see from other researches [cit] that a very simple Neural Network can be used. Those are the specification of the NN used:

- **Number of inputs**: as many as the *BOW* cluster points
- **Number of layers**: input layer, 1 or 2 hidden layers, output layer
- **Activation functions**: the hidden layers nodes use *Leaked Rectified Linear Function* (LRELU).
- **Layer connections**: all the layers are fully connected
- **Number of outputs**: number of total unique VMs.

It's important to notice that if we change the number of cluster points, we need to retrain both the BOW and the NN. In fact while the BOW is an *unsupervised* algorithm, the NN

on top is a *supervised* algorithm that needs the output of the trained BOW and the labels generated from the previously seen *VMs Detector System*.

LRELU: this activation function is an adaptation of the *RELU* activation function, defined as $f(x) = max(0,x)$; while the *LRELU* is defined as: $f(x) = max(\alpha x, x)$, with $\alpha \in [0,1]$

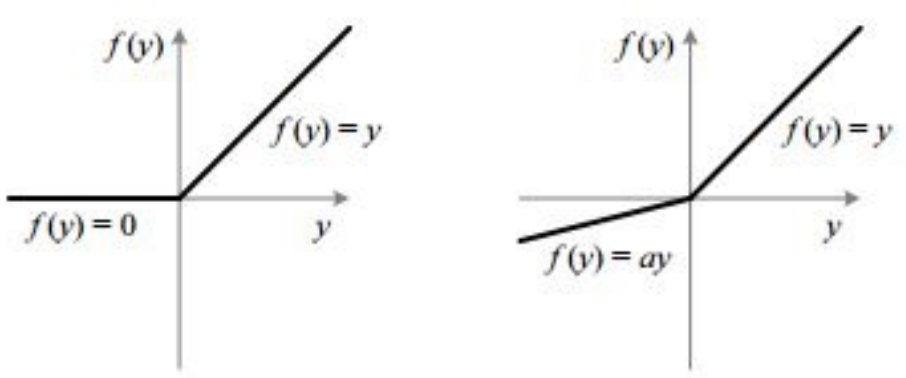


Fig. 5.8 RELU and LRELU functions

5.2.4 Experiments and Results

In this table we show the best results obtained with the *Bag of Visual Words* and the Neural Network.

Table 5.1

BOW_cluster	N_hidden1	N_hidden2	Min Val Error	Test Set Accuracy
1000	1000	1000	0,29	88,5
1000	**1000**	**500**	**0,27**	**88,5**
1200	1500	1000	0,35	86,27

Those reported are the best results obtained with various configurations and parameters. The best result is chose by lowest Validation error, and then highest Test Set Accuracy.

5.3 Scene Recognition with Inception V3 CNN

After testing and implementing the **BOW** system for *Scene Recognition*, we decided to try a different approach: **Inception Neural Network** (**INN**). Recently this new type of NN have achieved incredible results in the computer vision field, especially in images classification , setting the new state of the art in the field. Given the incredible success of those new architecture, we decided to try use those for our task. In particular we decide to use a specific model called **Inception V3**. [22] This NN is the current state of the art for the **ILSVRC 2014** *Data Set*.

5.3.1 Inception Neural Network

The **INN** is an evolution of the classic **Convolution Neural Network** (**CNN**). The main idea of the Inception architecture is to consider how an optimal local sparse structure of a convolutional neural network can be approximated and covered by readily available dense components.[23]
The main idea is to perform different operations on a single layer, and stack the results of those together.

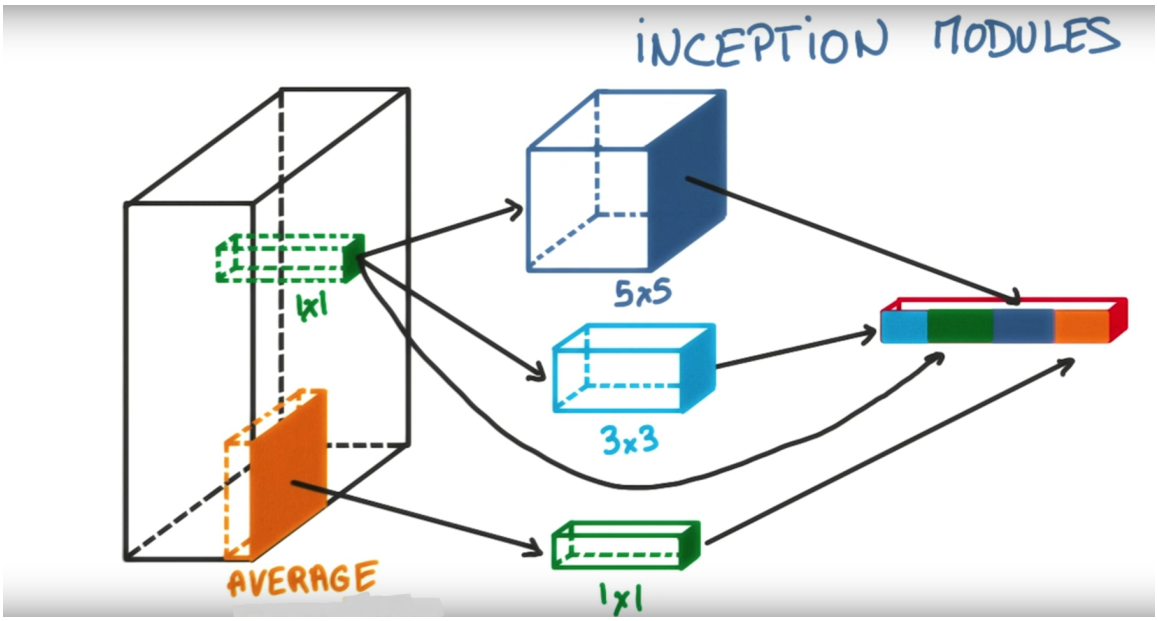


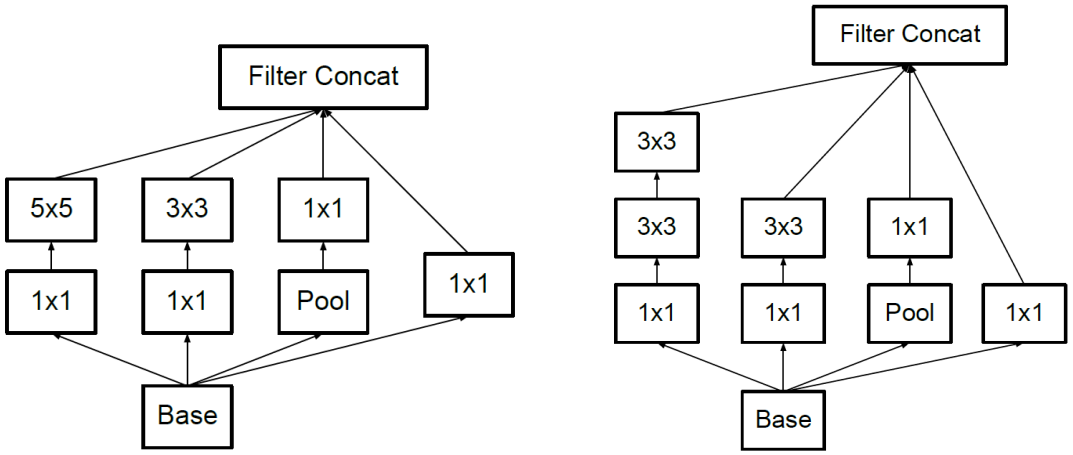
Fig. 5.9 Inception Module

In the **Inception Module**, three different operations are performed: a convolution with 5x5 and with 3x3 kernel; and an average pooling operation. Also both Convolution operations are preceded by a 1x1 convolution operation, in order to increase the depth of the Inception

Module.
All the results are then stuck together in the next layer, as can be seen in Figure **??**

5.3.2 Inception V3

The **Inception V3** architecture is a modified version of the classical **INN**. The main focus of this version is to drastically reduce computation time while improving overall results. This architecture is currently the state of art for the **ILSVRC 2014** *Data Set*. This is achieved with a relative low increase in computational cost ($2.5x$).[22]



(a) Inception Module

(b) Adapted Inception Module

The above pictures represent the original *Inception Module* (on the left), and the adapted *Inception V3 Module* (on the right). As we can see, instead of the 5x5 convolution, two consecutive 3x3 convolution are used. This immediately reduces the computational cost of the operation, given the smaller amount of weights to calculate: in fact a 5x5 convolution operation is more computational expensive than a 3x3 by a factor of $25/9 = 2.78$. And while the 5x5 convolution can detect dependencies of signal in further away units, if we use the same number of input for the first 3x3 convolution, the second 3x3 convolution will reduce to 1 output maintaining intact the 5x5 ratio input/output as in the *figure* 5.11
This replacement reduce the computational cost, while losing the geometrical information; but since we are using this for visual networks we can rely on the translation invariance propriety.

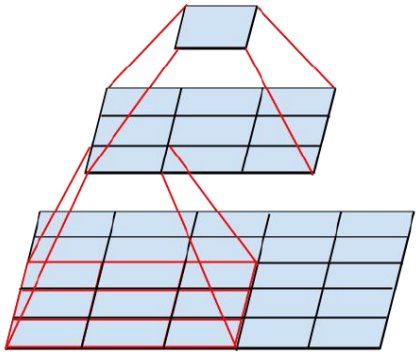
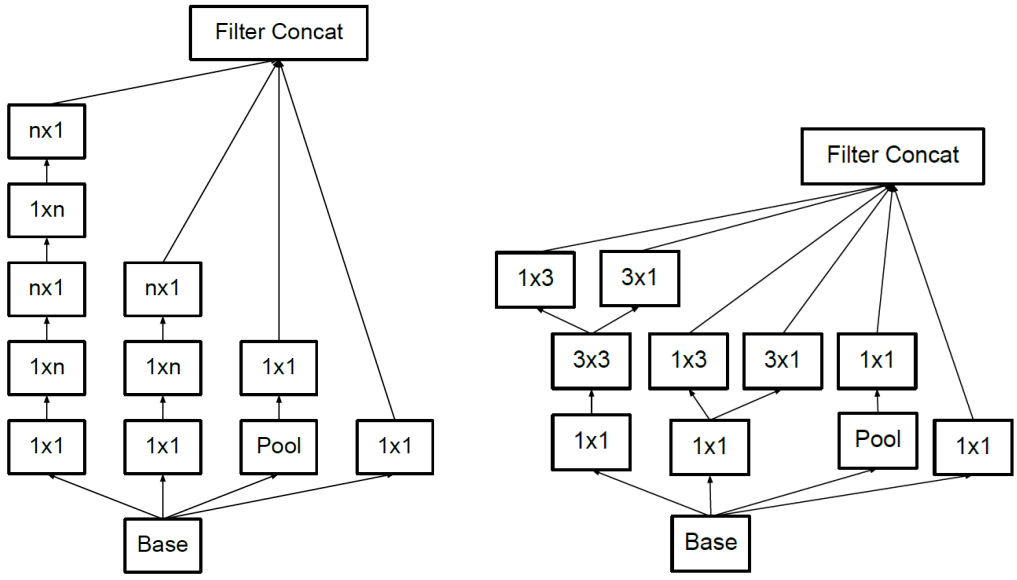


Fig. 5.11 Use of two 3x3 convolution instead of 5x5

Following the same idea, it's possible to exploit it further, by considering to replace a 3x3 convolution with a 3x1 and a 1x3 convolution in succession. Again this will reduce the computational time, while not losing meaningful information thank to the translation invariance propriety.

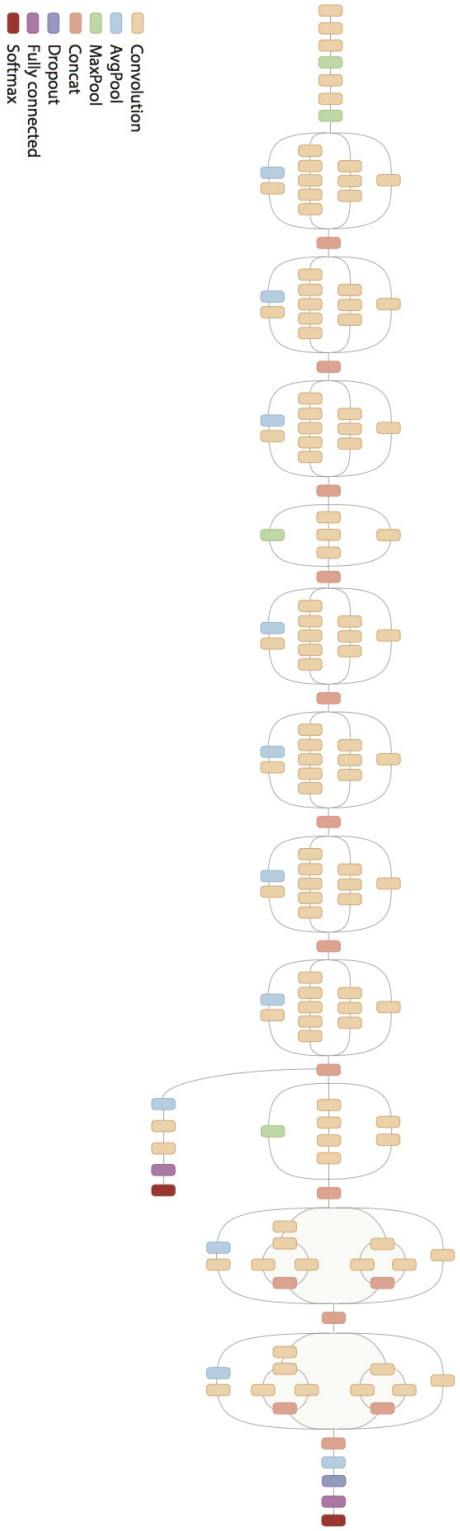


(a) Generalized Inception Module (b) Used Inception Module

Fig. 5.12 Inception Module adaptations

While the scheme represented in figure is least computational expensive possible, it will generate a representation bottlenecks. To avoid this, while maintaining computational reduction, and removing bottlenecks, can be seen in *figure 5.8*:

The **Inception Module** represented in *figure 5.8* is the final **Inception V3 Module**.



This is the full architecture of the Inception V3 network. The input connection is at the top, while the output at the bottom. The full architecture has 33 layers, considering the *Inception Module* as a single layer.

5.3.3 Inception V3 Use and Results

We used the fully trained *Inception architecture* till the last convolution layer. On top we added a hidden fully connected hidden layer, with 250 hidden units, and our output layer. The hidden layer uses the *RELU* activation function.
This has been done to be able to use the pre-trained parameters of the *Inception V3*, and only train our final hidden and output layer's weight for our *Scene Recognition System*.
In Figure 5.13 is possible to see the accuracy graph. We used 10% as validation set, 10% as test set, and the remaining 80% as training set. We obtained a 100% accuracy on the test set.

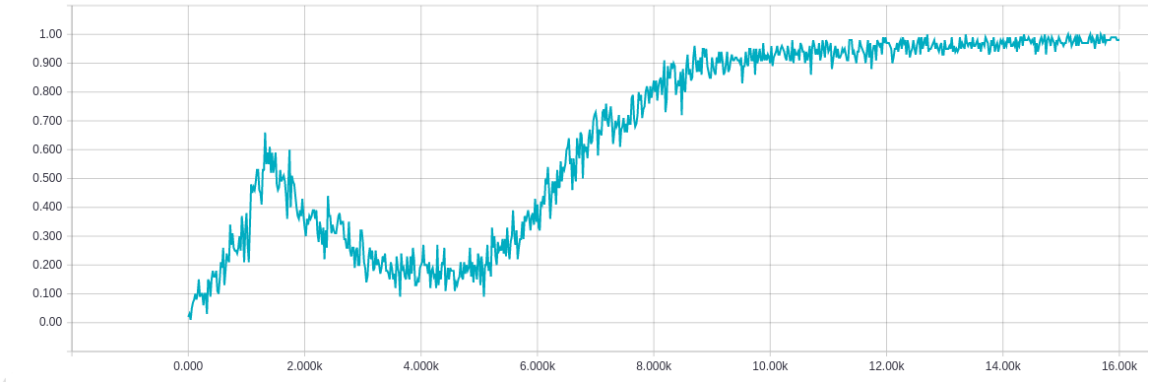


Fig. 5.13 Validation Accuracy

5.4 BOW vs INN

Given the better results obtained by the **INN**, on the same exact *Data Set*, we decide to leave the **BOW** algorithm for the much more performant **INN**. Other then better results, the **INN** is way faster to train over new data compared to the **BOW**, mainly tanks to the use of pre-trained network. In fact to train the final layer with optimal results it takes around 30 minutes with a data set of 7000 pictures; while takes around 3 days, on the same machine, to only compute the cluster point needed for the **Bow**. This huge difference is due the difference in algorithms: the **BOW** clustering algorithms runs on the *CPU* with a single thread given its sequential intrinsic propriety; while the training of the **INN** is computed on the *GPU* taking advantage of its highly scalable nature.

5.5 Real Word Performance

Given the high performance of the **INN** we where expecting similar good performance in the *Real World*: unfortunately this wasn't the case. During the test performed in the *Real World* we notice that when the image input was very similar to the one present in our *Data Set* the **INN** was performing according to our *Test Set*, but every time the input image was different from the ones present in the *Training Test* the results where **INN**'s predictions where completely off.
After investigating the problem we concluded that the **INN** didn't generalize well. Given the method used to capture the pictures for the *Data Set*, all the data per class are consecutive frames of the recorded video: this means those are quite similar to each other in term of feature computed from the pre-trained **INN**. To better understand this imagine taking a picture of a table in a room, then move two meters on your left, and take a second picture of the table. The difference in those two pictures, in terms of **INN** features, are almost none. All the difference in those two pictures lies in the geometrical proprieties of the objects itself: the table moved to the right in the picture, its prospective changed, etc.
To confirm our theory we performed a simple experiment: *Paragraph 5.5.1*

5.5.1 Geometrical Invariance Experiment

To prove that, regardless the optimal results of the **INN** on the *Test Set*, the poor performance in the *Real World* are due to a "wrong" feature selection of the pre-trained **INN** we designed the following text: we took a random picture (5.14a) from our *Data Set* and created a flipped copy (5.14b).

(a) Original Picture

(b) Flipped Picture

Fig. 5.14 Experiment pictures

We then use those two pictures as inputs for the pre-trained **INN**, and take as output the values of the last pre-trained layer. This output is indeed the input of our fully connected layer, and is the actual data we use to train our final layers.
Here we call $T1$ the tensor output from the original picture, and $T2$ the tensor output of the flipped picture. Then we compute the tensor D as $D = T1 - T2$.

$$len(T1) = 2048$$
$$Mean(T1) = 0.292797$$
$$STD(T1) = 0.296673$$

(a) T1 proprieties

$$len(T2) = 2048$$
$$Mean(T2) = 0.284759$$
$$STD(T2) = 0.304073$$

(b) T2 proprieties

$$len(D) = 2048$$
$$Mean(D) = 0.008037$$
$$STD(D) = 0.111435$$

(c) D proprieties

Analyzing the propriety of the resulting tensor D, we can see that the pre-trained **INN** generalized in a way that doesn't find much differences in the experiment pictures. In fact since tensor D represent the differences of the experiment's pictures in the feature space of the **INN**, the low *average* (Mean), and the low *Standard Deviation* (STD) are clear indicator of the similarity of those. This experiments, and the reported extensive use of *geometrical invariant* propriety used in the **INN** [22], prove the inability of this network to find differences in the geometric feature space.

5.5.2 Geometrical Variant Data Set

After the experiment discussed in the section 5.5.1, we decided to create a new *Data Set* with an explicit *geometrical variance* propriety; in order to further test the impact on the final results on the **INN**.
We created a new class with all the images of our *Data Set* flipped, and use half of those for the training, and the other half for testing. The order of the picture is randomize, in order to not have only some classes, flipped, in the training and the other in the test.
More precisely, our train set has all the original images, divided in their respective classes, and a new class with a copy of half the images flipped. The *test set* only has the other half of the flipped images. This is done to exactly see how the **INN** will perform with this extreme case.
We used 90% of the new data set as training and 10% as validation set; with the test set containing the rest of the flipped images. The label for those images is a new class called *flipped*. We trained and the network on the training set, and we stopped when we had a validation accuracy of 96%. After the network was trained, we test in on the test set, with only the flipped images:

- **Correct:** 0%
- **Tricked:** 97.43%
- **Missed:** 2.57%

With **Correct** we mean every time the NN classify the image as part of the class *flipped*. With **Tricked** we count the all the time the NN recognizes the image as the original one picture from which the flipped is created. Finally with **Missed** we counted all the time the NN classify the image neither as flipped, neither as the original one.

As this test results, the **INN** will classify all the flipped images as their original, non flipped, class. As we expected this pre-trained NN can not generalize well those types of *Data Set*.
We will use this new *Data Set* to test future implementation, in order the respond of the new systems regarding the *Geometrical Variance* property.

Chapter 6

New Deep Neural Network

After the results discussed in the previous section, we realized that to be able to well generalize our *Data Set*, while keeping the geometrical variance information, we needed a novel approach to **Deep Neural Network (DNN)**.
After analyzing the construction of the **INN**, we decided to rebuild a new model that will not rely on the *geometrical invariance* property.

6.1 ANN without Geometric Invariance

Once we identified, in the previous chapter, the influence of this property in our situation, we needed to analyze every element of our possible new ANN to identify every component that was relying on the *Geometric Invariance* property.
Starting from the **INN** structure cit[inception], we identify the main component that rely on this property as: *Pooling Layers* and the layer discussed in chapter 5.3.2, picture 5.11.

6.1.1 Pooling Layer

Pooling layers where designed to reduce the size of the input, while maintaining the key features in it. By applying a convolution operation on the input, it can reduce this by a factor of 2,3 or more while maintaining the most prominent information. This is done by applying computational inexpensive operations like *mean* or *max*. While these functions can well represent the input information, drastically reducing its dimensionality, these lose the geometrical information of it. As an example, when we apply a *mean* or *max* operation on a $3x3$ matrix, as output we will have only one number that represents the nine input numbers. While the output value can represent the "intensity" of the input, it can not represent the distribution or position of the input values. When these operations are applied in a NN,

we say that these operations maintain the features of the previous layers, but they lose the position or distribution of them.
It is true that the geometrical information lost only concerns a limited neighbor (in the example of $3x3$ cell, we lose a precision of 3), but when we consider DNN architectures, this can cause a way larger loss. If we take the **Inception V3** as an example, with an input of $300x300$, and 33 total layers, we have a Pooling layer on the 30th layer. At this level of deepness, the input has been "squeezed" to $\approx 10x10$ size, with multiple channels. The average operation used now, even if applied on a $3x3$ cell, will actually lose the geometry of $90x90$ neighbor relative to the input. This is given by the ratio $300x300/10x10$ of the original input, and the tensor input prior to the Pooling layer. The key element is the core idea of *back propagation* in ANNs. A *mean* or *max* operation is irreversible: from one number representing the mean or max of a $3x3$ input it is not possible to reconstruct the original $3x3$ input.

6.1.2 Convolution Layers with Strides

As an alternative to Pooling layers, we can apply convolution operations, with a fixed kernel and strides, to reduce the dimensionality of the input, while maintaining geometrical information.

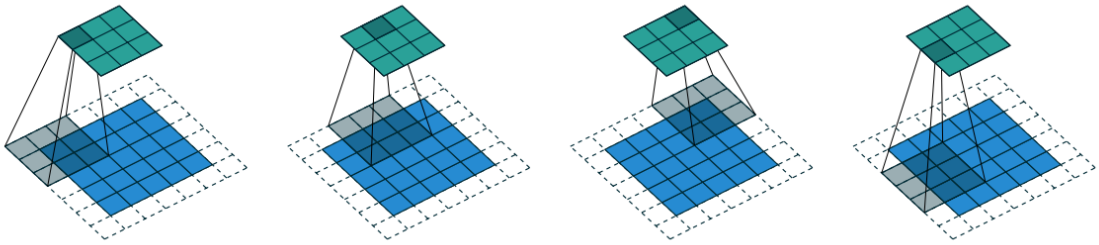
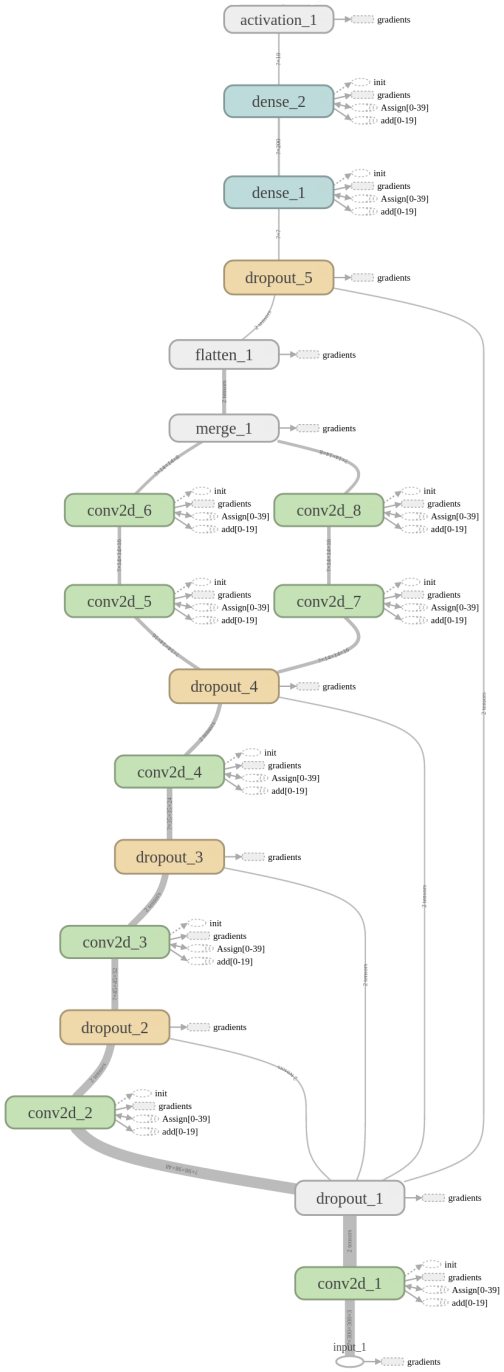


Fig. 6.1 Convolution operation with strides

In Figure 6.1 we can see how we can reduce, with a convolution operation, a $5x5$ input to a $3x3$ output with a $3x3$ kernel size. This operation maintains the *geometrical information* thanks to the kernel. When we do the back propagation operation, the geometrical propriety are intrinsic in the kernel values: each kernel's value represents the weights of one of the nine input pixels.

6.2 New Deep Neural Network Architecture

After the considerations discussed in the chapters 6.1.1, 6.1.2, we present our new DNN architecture:



DNN structure:

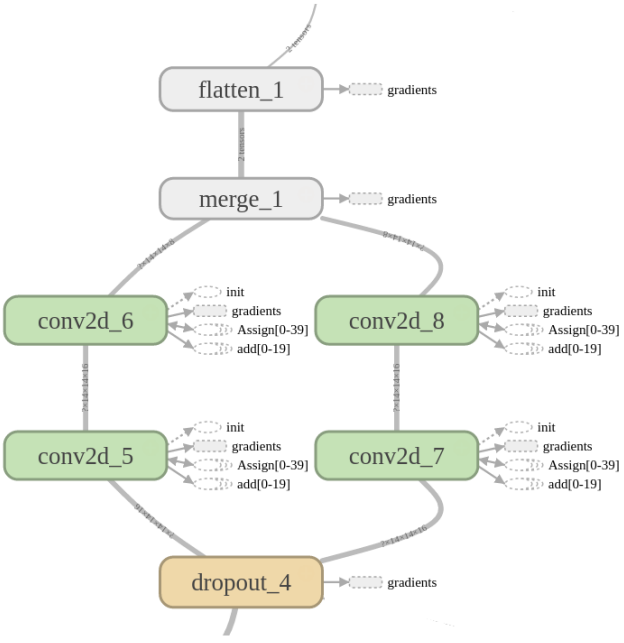
- **Input Layer** 6.2.1
- **Convolution Layer 1** 6.2.1
- **Convolution Layer 2** 6.2.1
- **Convolution Layer 3** 6.2.1
- **Convolution Layer 4** 6.2.1
- **Inception Layer** 6.2.1
- **Flatten Layer**
- **Fully Connected Layer** 6.2.1
- **Softmax Layer**
- **Output Layer** 6.2.1

6.2.1 DNN Description

Here we describe in details the full architecture of the DNN:

Layer Name	Layer Description	Tensor size	Weights
Input Layer	fully connected	300x300x3	0
Convolution Layer 1	7x7x48; strides=3; dropout=0.1	98x98x48	7104
Convolution Layer 2	9x9x32; strides=2; dropout=0.1	45x45x32	124448
Convolution Layer 3	11x11x24; strides=1; dropout=0.1	35x35x24	92952
Convolution Layer 4	22x22x16; strides=1; dropout=0.1	14x14x16	185872
Inception Layer	3x3x8 left; 5x5x8 right	28x14x8	4912
Flatten Layer		3136	0
Fully Connected Layer	200 hidden units	200	627400
Softmax Layer			0
Output Layer	12	10	2010

The *RELU* activation function is used in all the layers.
In particular we use a modified version of the *Inception Module*. Taking inspiration from the [cit inception] we created a new module that is able to maintain the geometrical information of the input.



(a) Modified Inception Module

We use in parallel two deep convolutions: on the left a $1x1$ kernel followed by a $3x3$, and on the right a $1x1$ kernel followed by a $5x5$. We remove the pooling layer and all the weight optimization discussed in Section 5.3.2. But we use the $1x1$ convolution: in fact this generate a deeper network and increases the channels, while performing a a simple dot operation that is extremely cheap in terms of computational resources.

Now the new architecture is described, we show the results obtained.

6.2.2 New DNN Results

Now that we designed a new architecture, that maintains the geometrical proprieties of the image in the whole architecture, we test it on the special data set discussed in Chapter 5.5.2. Here are the results obtained:

- **Correct:** 99.7%
- **Tricked:** 0%
- **Missed:** 0.3%

From these results we can be sure that our DNN is able to distinguish between flipped images, and to an extent, it maintains and elaborates the geometrical features of the input.

After this test we test the DNN on the classic data set used for navigation described in chapter 5.1.

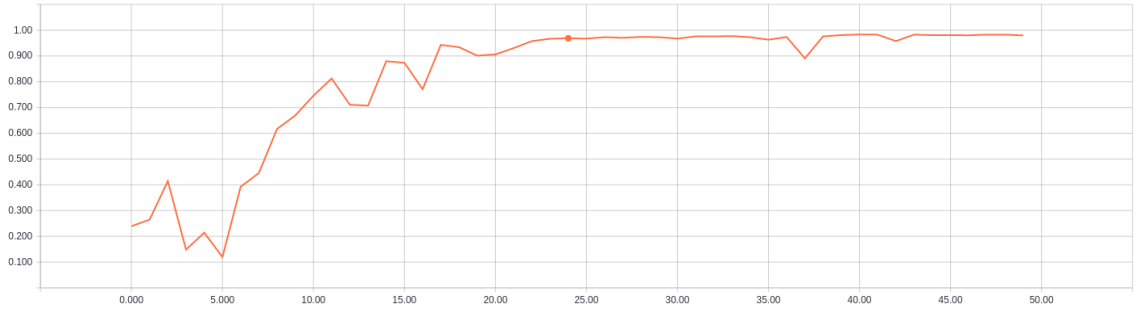
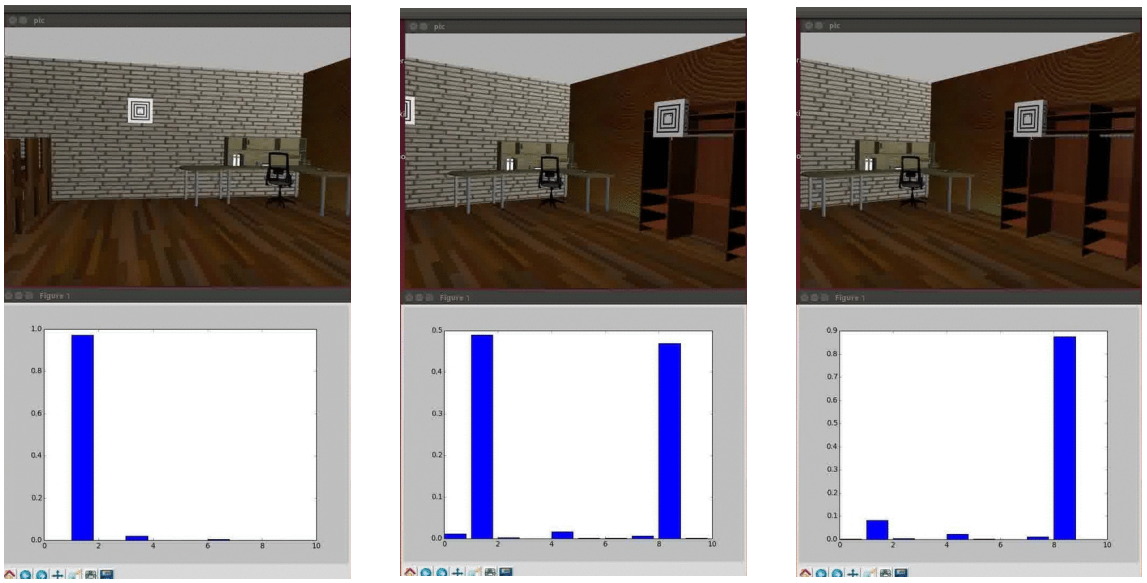


Fig. 6.4 New DNN accuracy

As we can see we obtain a total accuracy of 98.7% on the test set. Moreover, thanks to the previous experiments, we also rely on geometrical property. During live experiments, we can see a nice transition of class prediction when the camera moves from a VM to another. The direct correlation between the transition in the probability of the output of the DNN, and the geometrical transition of the camera shows even more the intrinsic geometrical propriety analyzed by the NN. This effect can be seen in figure 6.5. A deeper explanation is proposed in the section 6.2.3.

6.2.3 DNN Geometrical Results

Looking in detail we can see the image at the starting position (figure 6.5a), with the DNN output at the bottom. Here the output is presented as an histogram of probability over the total classes of the DNN. Each class represents a specific VM, and the level of the histogram represent the DNN certainty of recognizing a specif scene.
Now we start rotating the camera (or robot), to have a transition from the first scene to the



(a) Starting Position (b) First step rotation (c) Final step rotation

Fig. 6.5 DNN output over rotation movement

second scene. We start rotating to the right, and we can see in real time the DNN output. At some point of the rotation we will start looking at a scene that is a transitions scene between scene one and scene two. This is figure 6.5b. Now here it is very important to notice the almost equal level of the two scene. Note that the scale automatically changed, so both classes have a 50% certainty. By continuing the rotation we will end up in the new scene. This is represented by figure 6.5c.

This results perfectly shows the new relation between our DNN and the real geometrical world. All the geometrical information, not only are preserved in the whole network, but are intricately learned and associated to the final certainty output of our network. This is only possible thanks to the new proposed architecture: geometrical information are essential to achieve this result.

Chapter 7

Conclusions

In this thesis we proposed a procedure to enable a robot or vehicle to autonomously navigate in an indoor environment, relying on visual information. We proposed many novel approaches to different tasks needed to create a working system. In this chapter we will summarize the most relevant ones.

7.1 Data Labeling and Mapping and Localization

Thanks to the use of VMs, we proposed a method that follows the *lazy teacher* approach: we automated as much as possible the procedure of labeling of the data. We designed a new VM, focusing on the needs of our task. This new VM design enables the detection of it from a further distance than other designs, greatly reducing the false positives errors and holds only the essential information needed to generate labels for a visual based scene recognition dataset. Thanks to this approach we combined the benefits of natural landmarks and artificial landmarks: using simple and inexpensive artificial landmarks we created a dataset, that was used to train a DNN that relies on the natural landmarks to robustly recognize different locations. This combined approach enables the creation of big dataset in a fast, efficient and inexpensive way. In fact, while the data recording is done with simple video recording, the label generation procedure is fast and almost fully automatic. This procedure also enables a dynamic creation of the dataset: both for fine tuning or adding new scenes, there is no need to repeat the whole procedure, but by simply adding new VMs in the new area it is possible to increase the existing dataset.
Moreover thanks to this approach we focused on a non grid-like map, but on a graph map instead that takes inspiration from human navigation. The use of key areas, denoted as Regions, enables a fast and memory efficient map creation process. While this approach loses precision compared to a grid based map approach, it has the robustness of human-like

navigation: simply by looking at the surroundings it is possible to have a reliable likelihood of being in a certain region. This can be increased by performing simple *tests* to maximize the certainty of the system, like looking around. This is possible thanks to the structure of the Regions that are composed of Nodes, and their intrinsic geometry information.
Moreover this approach solves the *same door* localization problem in Navigation: if there are two very similar areas in the map, like being in front of one of many doors in the area, the difficulty of knowing which door are we close to. Thanks to a graph map approach we know the connections between regions: knowing the region we come from, and the movement we performed, we can greatly reduce the possible regions we arrived at. Moreover by defining all the Nodes with the visual information from the front, left and right side of the robot we can solve the *same door* localization problem, and highly increase the robustness of the localization system.

7.2 A novel Deep Neural Network Architecture

[1] In this thesis we proposed a novel Deep Convolutional Neural Network architecture, that does not rely on the geometrical invariance property. While using layers like Pooling and weight optimization like the one used in INN V3 greatly increases the performance of DCNN for scene recognition tasks, it is not the optimal choice in every field. Here we focused on navigation, where geometry is essential for any navigation system. We showed that it is possible to create high performing DCNNs that can also maintain the geometrical properties of the input, giving an intrinsic knowledge of the geometry of the environment to the network itself.
Since the proposed DNN is a key component of a visual based navigation system, we focused on experimenting on datasets which represent real world scenarios. The whole pipeline consists of an almost autonomous way of generating labels, to create in a fast way a dataset of the environment. This is used to train the proposed DNN, which can generalize the training data in a meaningful way: a direct correlation between geometrical movement and the DNN output probability. This is only possible when the DNN relies on geometrical properties. It is also important to notice that we experimented with removing the VMs from the environment, and the DNN performances remain the same; but no extensive experiments were performed with this focus.
To conclude, we show that it is possible to use convolution layers to create deep architectures, while maintaining all the geometrical properties of the data. This can be applied in all the

[1] This chapter is part of a waiting for review conference paper

fields where the position, or geometrical properties of the features are as important as the features themselves.

7.3 Future Work

The work we proposed focused on the localization and mapping systems. These are combined with the classic Navigation System through ROS. Thanks to this modular nature of this work, it is possible to implement it side by side with other systems. In fact one of the key elements of this work was the possibility to be ready to use for most applications. As an example, when a precise localization is needed, the system proposed can work in parallel with a grid based map approach: while using our localization for the global map, once a desired region is reached, it is possible to switch to a classical laser based localization, for a more precise position in the region. This combination allows the possibility to create laser maps of only specific areas of interest, while using the proposed graph map approach for the totality of the environment.
Another focus of this whole system and general procedure is to be as independent as possible from hardware specifications. In fact it is possible to use this procedure on a completely different robot or vehicle, as long as a camera is available. This is possible thanks again to the modularity approach: the localization system and the graph map are independent from the rest of the system, and can be used in combination with other systems.

The new DNN architecture proposed, showed that is possible to use DCNNs for gemetrically based data. Moreover it can be applied as the core localization subsystem for a visual based navigation system. This would rely on the geometrical properties of the real world environment to possibly create a human like approach to navigation. If we consider that humans are able to perfectly navigate in any environment with solely visual information, then the proposed approach would fit the need of a human like navigation system. To accomplish this, more considerations and research needs to be done, focusing on the system itself: the use of 3D cameras, stereoscopic cameras and more.
On a more general level this approach shows potential in applications where geometrical structure matters. As an example we could consider FMRI scans: these visual data are directly correlated to the brain areas. Another example could be visual games: from chess to Atari games. Given the ability of the proposed DCNN to rely on the positions of the features and to generalize the geometrical properties of the data, the possible benefits are clear.

Moreover is possible to experiment and build further and more sophisticated localization behavior. As an example the possibility to use **Long Short Term Memory** (LSTM) Networks on top of the localization system for more robustness. Moreover this would allow a real continuous learning procedure: using as input of this network output tensor of the scene recognition system, the graph map status and the movement performed, it is possible to predict the reached Region. This will enable the system to continuously learn the possible hardware errors, or even small environment modifications.
On another topic, we introduced a new DNN architecture, that shows the importance of geometrical information of the input

References

[1] Hobart R Everett and Douglas W Gage. From laboratory to warehouse: Security robots meet the real world. *The International Journal of Robotics Research*, 18(7):760–768, 1999.

[2] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 163–168. IEEE, 2011.

[3] Jose Guivant, Eduardo Nebot, and Stephan Baiker. Autonomous navigation and map building using laser range sensors in outdoor applications. *Journal of robotic systems*, 17(10):565–583, 2000.

[4] Dario Floreano and Robert J Wood. Science, technology and the future of small autonomous drones. *Nature*, 521(7553):460, 2015.

[5] Alberto Elfes. Sonar-based real-world mapping and navigation. *IEEE Journal on Robotics and Automation*, 3(3):249–265, 1987.

[6] Salah Sukkarieh, Eduardo Mario Nebot, and Hugh F Durrant-Whyte. A high integrity imu/gps navigation loop for autonomous land vehicle applications. *IEEE Transactions on Robotics and Automation*, 15(3):572–578, 1999.

[7] Stephen Se, David Lowe, and Jim Little. Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *The international Journal of robotics Research*, 21(8):735–758, 2002.

[8] Stephen Se, David Lowe, and Jim Little. Vision-based mobile robot localization and mapping using scale-invariant features. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 2051–2058. IEEE, 2001.

[9] Laurent George and Alexandre Mazel. Humanoid robot indoor navigation based on 2d bar codes: Application to the nao robot. In *Humanoid Robots (Humanoids), 2013 13th IEEE-RAS International Conference on*, pages 329–335. IEEE, 2013.

[10] Reginald G Golledge. Path selection and route preference in human navigation: A progress report. In *International Conference on Spatial Information Theory*, pages 207–222. Springer, 1995.

[11] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[12] Riccardo Cassinis, Fabio Tampalini, and Roberto Fedrigotti. Active markers for outdoor and indoor robot localization. *Proceedings of TAROS*, pages 27–34, 2005.

[13] Alan Mutka, Damjan Miklic, Ivica Draganjac, and Stjepan Bogdan. A low cost vision based localization system using fiducial markers. *IFAC Proceedings Volumes*, 41(2):9528–9533, 2008.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[16] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962, 2003.

[17] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jurgen Schmidhuber. Convolutional neural network committees for handwritten character classification. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 1135–1139. IEEE, 2011.

[18] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[19] Jun Yang, Yu-Gang Jiang, Alexander G Hauptmann, and Chong-Wah Ngo. Evaluating bag-of-visual-words representations in scene classification. In *Proceedings of the international workshop on multimedia information retrieval*, pages 197–206. ACM, 2007.

[20] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[21] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[22] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[23] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.