NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Real-Time Navigation of Unmanned Vehicle based on Neural Networks Classification of Arrow Markings

Smaragda D. Reppa

Supervisor: Stathes P. Hadjiefthymiades, Professor

ATHENS

JULY 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αυτόνομη πλοήγηση μη επανδρωμένου οχήματος σε πραγματικό χρόνο, με χρήση νευρωνικών δικτύων για την κατηγοριοποίηση σημάνσεων

Σμαράγδα Δ. Ρέππα

Επιβλέπων: Ευστάθιος Π. Χατζηευθυμιάδης, Καθηγητής

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2022

BSc THESIS

Real-Time Navigation of Unmanned Vehicle based on Neural Networks Classification of Arrow Markings

Smaragda D. Reppa

S.N.: 1115201600143

SUPERVISOR: **Stathes P. Hadjiefthymiades, Professor**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αυτόνομη πλοήγηση μη επανδρωμένου οχήματος σε πραγματικό χρόνο, με χρήση νευρωνικών δικτύων για την κατηγοριοποίηση σημάνσεων

Σμαράγδα Δ. Ρέππα

Α.Μ.: 1115201600143

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Ευστάθιος Π. Χατζηευθυμιάδης,** Καθηγητής

ABSTRACT

Since the 1970s, autonomous robots have been in daily use at any altitude, for deep-sea and space exploration as well as in almost all aircraft. The last decades, an increasing interest has been recorded on the exploitation of unmanned vehicles in fields such as environmental monitoring, commercial air surveillance, domestic policing, geophysical surveys, disaster relief, scientific research, civilian casualties, search and rescue operations, archeology, maritime patrol, seabed mapping, traffic management, etc. Regardless the domain (i.e., aerial, ground or surface) that they belong to, the key elements that distinguish them as the leading edge of their technology are the provided degree of autonomy (i.e., the ability to make decisions without human intervention), the endurance and the payload that they can support.

A complicated task which is a prerequisite in robotic missions is the autonomous navigation of the robots. An autonomous mobile robot constructs a robust model of the environment (mapping), locates itself on the map (localization), governs the movement from one location to the other (navigation) and accomplishes assigned tasks. Going a step further the autonomous and self-driving impose a new research area where vehicles can monitor the road marks or the traffic signs and take the proper decisions for their navigation in space. The image classification in real-time navigation is latency sensitive and resource consuming task.

In this Thesis, we aimed to navigate an unmanned vehicle on an unknown path by recognizing and following arrow markings and signs. For the image recognition, a CNN was used, which is trained with a dataset of arrow markings. For dataset creation, a Turtlebot 2 was used along with a raspberry pi camera. A benchmarking of convolutional neural networks, i.e. VGG-16 and VGG-19, is presented. Finally, the real-time experiments were executed with a Turtlebot2 in real-time navigation and discussed.

SUBJECT AREA: Image Processing, Convolutional Neural Networks

KEYWORDS: neural networks, convolutional neural networks, image processing, path planning, navigation, unmanned vehicle, turtlebot, ROS

ΠΕΡΙΛΗΨΗ

Από τη δεκαετία του 1970, τα αυτόνομα οχήματα χρησιμοποιούνται ευρέως, για εξερεύνηση βαθέων υδάτων και διαστήματος καθώς και σε όλα σχεδόν τα αεροσκάφη. Τις τελευταίες δεκαετίες, έχει καταγραφεί αυξανόμενο ενδιαφέρον για την εκμετάλλευση μη επανδρωμένων οχημάτων σε τομείς όπως η παρακολούθηση του περιβάλλοντος, η αέρια επιτήρηση για εμπορικές πτήσεις, η αστυνόμευση, οι γεωφυσικές έρευνες, η αντιμετώπιση φυσικών καταστροφών κι εντοπισμός θυμάτων, η επιστημονική έρευνα, οι επιχειρήσεις έρευνας και διάσωσης, η αρχαιολογία, η περιπολία σε θάλλασες, χαρτογράφηση βυθού, διαχείριση κυκλοφορίας κ.λπ. Ανεξάρτητα από τον τομέα (δηλαδή εναέριο, επίγειο ή επιφανειακό) στον οποίο ανήκουν, τα βασικά στοιχεία που τα διακρίνουν ως αιχμή της τεχνολογίας τους είναι ο παρεχόμενος βαθμός αυτονομίας (δηλ. ικανότητα λήψης αποφάσεων χωρίς ανθρώπινη παρέμβαση), την αντοχή και το ωφέλιμο φορτίο που μπορούν να υποστηρίξουν.

Ένα πολύπλοκο έργο που αποτελεί προϋπόθεση στις αποστολές αυτών των οχημάτων είναι η αυτόνομη πλοήγηση τους. Ένα αυτόνομο κινητό ρομπότ κατασκευάζει ένα ισχυρό μοντέλο του περιβάλλοντος (χαρτογράφηση), εντοπίζεται στον χάρτη (τοποθέτηση), ελέγχει τη μετακίνηση από τη μια τοποθεσία στην άλλη (πλοήγηση) και εκτελεί εργασίες που έχουν ανατεθεί. Βλέποντας ένα βήμα παραπέρα, η αυτόνομη και αυτόνομη οδήγηση επιβάλλει έναν νέο ερευνητικό χώρο όπου τα οχήματα μπορούν να παρακολουθούν τα οδικά σήματα ή τα σήματα κυκλοφορίας και να λαμβάνουν τις κατάλληλες αποφάσεις για την πλοήγησή τους στο διάστημα. Η ταξινόμηση εικόνων στην πλοήγηση σε πραγματικό χρόνο είναι μια εργασία που απαιτεί χρόνο και καταναλώνει πόρους.

Σε αυτή τη διατριβή, στοχεύουμε να πλοηγήσουμε ένα μη επανδρωμένο όχημα σε άγνωστο μονοπάτι αναγνωρίζοντας και ακολουθώντας τα σημάδια και τις πινακίδες με βέλη. Για την αναγνώριση εικόνας χρησιμοποιήθηκε ένα συνελικτικό νευρωνικό δίκτυο, το οποίο εκπαιδεύεται με ένα σύνολο δεδομένων από σημάνσεις βέλους. Για τη δημιουργία δεδομένων, χρησιμοποιήθηκε ένα Turtlebot 2 μαζί με μια κάμερα raspberry pi. Παρουσιάζεται μια συγκριτική αξιολόγηση των συνελικτικών νευρωνικών δικτύων, π.χ. VGG-16 και VGG-19. Τέλος, τα πειράματα σε πραγματικό χρόνο εκτελέστηκαν με Turtlebot2 σε πλοήγηση σε πραγματικό χρόνο και συζητήθηκαν.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: π.χ. Επεξεργασία Εικόνας, Συνελικτικά Νευρωνικά Δίκτυα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: νευρωνικά δίκτυα, συνελικτικά νευρωνικά δίκτυα, επεξεργασία εικόνας, σχεδιασμός διαδρομής, πλοήγηση, μη επανδρωμένο όχημα

Αφιερώνω αυτήν την πτυχιακή εργασία στην οικογένειά μου για την αδιάκοπη στήριξή τους κατά την εκπόνησή της, αλλά και καθ'όλη τη διάρκεια των σπουδών μου.

ΕΥΧΑΡΙΣΤΙΕΣ (ή AKNOWLEDGMENTS)

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή, Ευστάθιο Χατζηευθυμιάδη, που μου έδωσε την ευκαιρία να αναλάβω το συγκεκριμένο θέμα και να ασχοληθώ με τον τομείς της ρομποτικής και των νευρωνικών δικτύων. Θα ήθελα να ευχαριστήσω, επίσης, την διδάκτορα Κυριακη Παναγιδη για τη συνεργασία και την πολύτιμη βοήθειά της καθ'όλη τη διάρκεια εκπόνησης της εργασίας.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In recent years, the immense development and use of unmanned vehicles seems to overwhelm the technological community. Unmanned vehicles are used more and more in many applications because of their rapid and cost-effective deployment. They can be used not only for reconnaissance, but also as communication platforms. Compared with satellite or sensor platform, an unmanned vehicle has simple system construction, high speed and low lag communication capability. As an auxiliary infrastructure, it provides reliable wireless links for remote users to realize safe and reliable transmission of information.

For all these reasons, the autonomous navigation of unmanned vehicles is a main challenge due to the complexity and the dynamic nature of the environment around as the interaction between themselves, persons or any unannounced change in the area. For instance, a vehicle should navigate itself on the road following a specific path marked by arrow markings, which are numerous, indicating a set of rules. The autonomous navigation can be enhanced by the use of machine learning and more specifically the use of Convolutional Neural Network (CNNs). A CNN 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 architecture of a CNN is analogous to that of the connectivity pattern of Neurons in the human brain; individual neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. A collection of such fields overlap to cover the entire visual area.

This research proposes a methodology that uses CNN techniques to detect arrow markings in a path that show directions captured by a raspberry-pi camera and to classify these arrows accordingly in order to follow a specific trajectory. A pre-processing function was implemented in CNN to enhance the classification of the images captured in the arrow markings classes. A popular image recognition architecture for CNNs were deployed which is named as Visual Geometry Group (VGG). VGG is the basis of deep ground-breaking object recognition models with multiple layers. Developed as a deep neural network, the VGG also surpasses baselines on many tasks and datasets beyond ImageNet. The “deep” refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers. A series of experiments were conducted, in which i) a benchmarking of VGG-16 and VGG-19 models with different hyperparameters and ii) a real-time navigation with an unmanned vehicle were performed. The real-time experiments were executed with a Turtlebot2 running ROS with a raspberry pi camera installed. In Section 2 a Robotic Operating System (ROS) used for the controlled UGV is presented. In Section 3 Neural Networks and Convolutional Neural Networks are described. In Section 4 related work of this thesis is discussed. In Section 5 the problem, the challenges and our solution are presented and in Section 6 we present and explain the experiments conducted and the conclusions are following in Section 7.

2. UNMANNED VEHICLES

2.1 Definition of Unmanned Vehicles

An unmanned vehicle is the vehicle that operates without a human presence on board. They can be either remote-controlled guided vehicles or autonomously guided vehicles. An unmanned system is equipped with the required data processing units, sensors, automatic control, and communications systems to carry out missions without the need for human intervention. Unmanned aircraft, ground robots, undersea explorers, and other unusual structures are examples of unmanned systems [1].

The autonomous vehicles can sense their surroundings using sensors or maps that depict the area. The usage of a map, on the other hand, demands prior knowledge of the environment, suggesting that the unmanned vehicle or other device must first explore the world and produce the necessary map before operating autonomously in this environment. Unmanned vehicles, on the other hand, may explore their surroundings using a range of sensors such as cameras, lidar, and temperature sensors. Many scientists have been working on developing new algorithms or optimizing current ones for unmanned vehicles to operate autonomously and examine their surroundings with sensors.

Unmanned vehicles, depending on the environment they are being navigated in, include UAVs, or unmanned aircraft aerial vehicles, also known as drones, UGVs, or unmanned ground vehicles, and USVs, or unmanned surface vehicles, which operate on the water's surface and are also known as surface drones.

2.1.1 Unmanned Ground Vehicles

Unmanned ground vehicles (UGVs) are vehicles that function without the presence of a human onboard. The vehicle's operation can be managed via teleoperation by human actors or it can be completely autonomous thanks to specially implemented algorithms. UGVs can be utilized in a variety of situations where the presence of human operator would be inconvenient, hazardous, or even impossible [1].

Today's UGVs come in a wide variety. They are used for military purposes, space, civilian or commercial applications. Surveillance, reconnaissance, and target acquisition are examples of military applications. Agriculture, mining, and construction constitute the industrial purposes.

Figure 1 - Autonomous UGV in public transportation

Figure 2 - Military UGV for explosive disposal

Unmanned cars that work autonomously and can be used for transportation as a regular car are now being researched by scientists all around the world. It all started in 1921, when the goal was to build a remotely controlled car, but in recent years, the goal has shifted to creating a self-driving car that will be used for human transportation. The safety of passengers on board a self-driving automobile is critical. As a result, the construction of the algorithm, particularly for navigation, must be properly designed. Sensors such as LiDAR, GPS, and cameras are required for these algorithms.

Figure 3 - Self-driving car

An unmanned aerial vehicle (UAV), also known as a drone, operates without a human pilot, crew, or passengers. A UAV can be fully autonomous, or controlled from a human operator usually on the ground or in another vehicle [1]. UAVs [2] are becoming increasingly popular in a variety of applications, civilian, commercial, military, and aerospace applications. Civilian applications include real-time monitoring, traffic monitoring, wireless coverage, remote sensing, search and rescue, cargo delivery, security and surveillance, precision agriculture, civil infrastructure inspection, or even aerial photography, while military applications include intelligence or reconnaissance missions, demining of an area.

Figure 4 - Italian Selex ES Falco used by several military for reconnaissance and surveillance

Figure 5 - Drone used in agriculture

2.1.2 Unmanned Surface Vehicle

Unmanned surface vehicles (USVs) are boats that operate autonomously without humans on board. In most cases USVs are controlled remotely by a human actor that usually is on the ground. USVs can be used in military missions, in oceanography and seaweed farming. Furthermore, USVs are the future of cargo shipment.

Figure 6 - Unmanned naval boat

2.2 Robot Operating System

With technology progressing so fast, robotics has become a more and more essential part of people's lives. Robots can be used in many situations for many purposes, as in

dangerous environments (including inspection of radioactive materials, bomb detection, and deactivation), industry, or where humans cannot survive like in space or underwater). Aside from this progress, robots still present significant challenges for software developers, such as code reusability and computational distribution. A proposed solution to these challenges is a software platform called Robot Operating System (ROS).

2.2.1 Definition of ROS

ROS is an open-source, meta-operating system for robots. As it is a meta-operating system, it does not replace, but it runs alongside a traditional operating system. It provides a set of software libraries and tools, that allow someone to obtain, write, run code across multiple different terminals, in order for a robotics application to be built. Also, it provides services as of those of an operating system, such as message exchanging between processes, low-level device control, hardware abstraction, package management.

Its open source feature allows the reusability of code by other users and requires another open source operating system underlying. Currently only runs on Unix-based platforms like Mac – OS and Linux. Other advantages are that it supports peer – to – peer communication through a reliable mechanism that enables communication between different components and that its simulators improve time and quality of code testing.

In addition to the above, there are more reasons why it is so popular. ROS is very light, it gives you the opportunity to control multiple robots simultaneously while they communicate with each other, it has great simulation tools, such as RViz and Gazebo. It provides many libraries that allow you to use several languages, other than C++ and Python that are the most popular and the chance for nodes of different languages to be able to communicate.

ROS comes in versions which are called distributions. Each one was named using adjectives that start with successive letters of the alphabet.

2.2.2 ROS packages

Software in ROS is organized into packages. A package is a directory that could contain a collection of ROS nodes, a ROS - independent library, a dataset, configuration files, a third-party piece of software, external libraries and a manifest which is an xml configuration file called “package.xml”. The goal packages aim to achieve is to provide a useful functionality so that software can be easily reused and maintained.

2.2.3 ROS stack

Stacks are the primary mechanism in ROS for distributing software. A stack is a collection of packages. Practically it is a directory, that among others, contains a “stack.xml” file. While the goal of packages is the reusabillity of code by storing it into collections, the goal of stacks is the simplicity of code sharing. It can comprise packages that all provide a certain functionality, such as a navigation stack or a manipulation stack.

Unlike an ordinary software library, these stacks can also provide this functionality via ROS topics and services or add functionalities, during execution time. A stack can also declare dependency another stack or more. Every stack has a manifest, a file with essential information about the stack and dependencies to other stacks.

2.2.4 ROS catkin

This is the ROS build system. It is a set of tools, used to create executable programs, interfaces, scripts and libraries so that they can be reused and maintained.

2.2.5 ROS nodes

In ROS, a node is an instance of an executable. It may represent a sensor, wheel motor, processing or monitoring algorithm, etc. Every node that starts running declares itself to the Master. All nodes are combined into a graph and their communication is mostly based on topics. A robotic system usually includes many nodes. Nodes also have a node type, that simplifies the process of referring to a node executable on the filesystem. These node types are package resource names with the name of the node's package and the name of the node executable file.

The use of nodes in ROS is very beneficial to the overall system, as code's complexity has reduced in comparison to monolithic systems. Details about implementation are hidden better as the nodes disclose less information to the rest of the graph and alternate implementations, even in other programming language.

A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

2.2.6 ROS topics

Topics are named buses that nodes use in order to communicate with each other. This communication is achieved through messages. A node that is interested in a specific kind of data subscribes to the relevant topic and consumes data, published by other nodes over the same topic. Therefore, nodes exchanging data over a topic don't know whom they are communicating with. There can be multiple publishers and subscribers to a topic as long as they are named differently.

Topics are intended for unidirectional, streaming communication. Each topic has a specific type, which means that a specific type of message should be used for node communication through this topic.

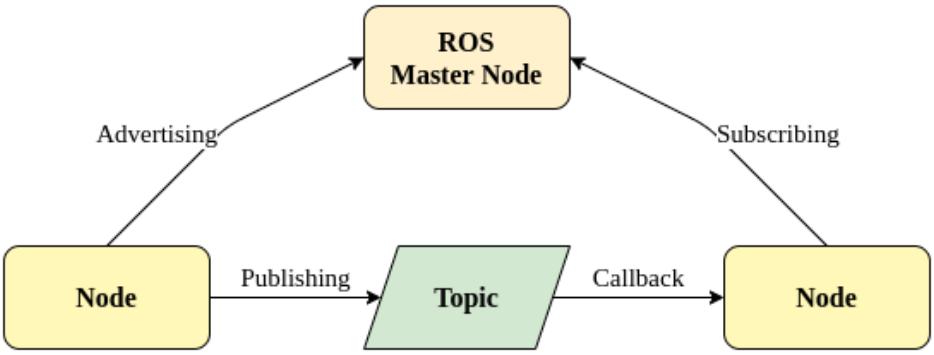


Figure 7 - ROS Topics and MasterNode Communication

2.2.7 ROS services

Services are another kind of communication for nodes. A service is a pair of messages, a request and a reply. A node offers a service under a string name and a "client" node calls the service by sending the request message and waiting for the first one to reply. They are defined using srv files.

Like topics, services also have types. The name of the type that is the package resources the same with the name of the .srv file.

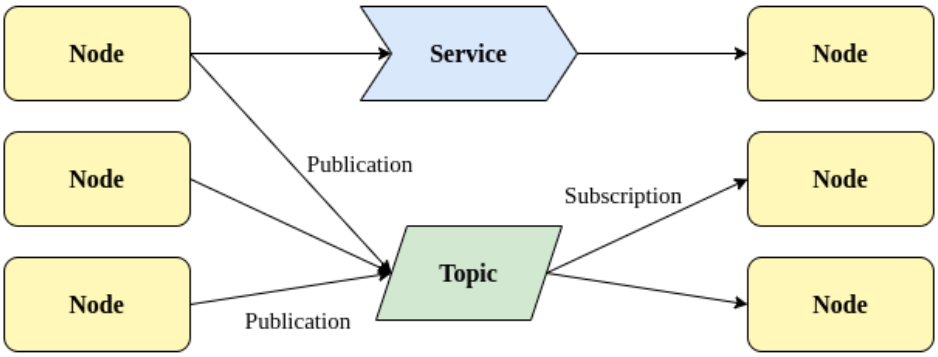


Figure 8 - Topics and Nodes Pub/Sub Service

2.2.8 Master nodes

The Master is a node which declares and registers all other nodes, both very essential for ROS's use. It is basically a primary node, that allows all other nodes to locate each other, communicate and exchange data. Master node is activated by using the command "roscore" before any other code execution and it has to be running until the end of every execution and action taking place.

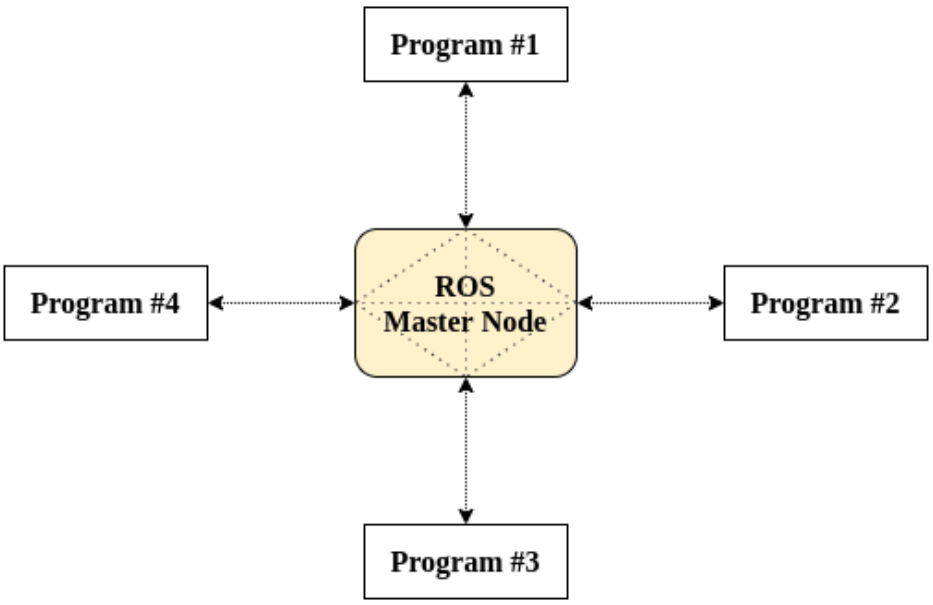


Figure 9 - Master Node

2.2.9 ROS messages

As mentioned before, ROS nodes communicate with each other. That happens via ROS messages and through a topic. A publishing node constructs a message, publishes it on to atopic and other nodes subscribing to this topic can receive this message. Nodes can also exchange a request and response message through a service call.

A message is a simple data structure. A .msg file is a simple text file stored in the msg subdirectory of a package, that specifies the data structure of a message. Standard types such as integer, floating point etc. are supported and so are arrays of those types. A user can also create their own type of message by declaring a .msg file.

2.2.10 Launch files

ROS provides a tool for simultaneously launching multiple ROS nodes. This is achieved through launch files, xml files using a .launch extension. Roslaunch can execute one or more of these files that specify details about paraneters, nodes needed etc.

2.3 Sensors

Sensors are required for robot systems to operate as expected in any environment and accomplish specific tasks. A LiDAR or a Kinect camera, for example, would be regarded vital in a robot's navigation system. In this thesis a raspberry pi camera is used for picture capturing, in order for the robot to follow a path.

2.4 Turtlebot

TurtleBot [turtle] is a low-cost, mobile, programmable robot with open-source software, was created by Melonee Wise and Tully Foote at Willow Garage in November 2010. It consists of a mobile base, multiple sensors, but does not have a processor. So, it needs an external unit which runs ROS such as a laptop or a rasberry pi.

It is very popular amongst software developers. TurtleBot can do real-time obstacle avoidance and autonomous navigation using the conventional TurtleBot components. It can construct a map using standard Simultaneous localization and mapping (SLAM) techniques and can be operated remotely using a laptop or an android-based smartphone.

Figure 10 - Turtlebot UGV

2.5 Simulators

A well designed simulator is an essential tool for every developer who is involved in the field of robotics. It enables the user to test their algorithms, design robots, projects, environments, perform testing, debugging, work on their innovative ideas rapidly and efficiently, in general. For this thesis Gazebo and RViz where used.

Gazebo offers the ability to simulate populations of robots in complex indoor and outdoor environments. It is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Best of all, Gazebo is free with a vibrant community.

Rviz is a three-dimension visualization tool for ROS applications. It provides a view of your robot model, capture sensor information from robot sensors and replay captured

data. It can display data from sensors such as a camera, from 3D and 2D devices including pictures and point clouds. Rviz simulator is available by using the command “roslaunch rviz rviz”.

3. IMAGE PROCESSING TECHNIQUES AND NEURAL NETWORKS

3.1 Image Processing and Unmanned Vehicles

Sensors fitted on autonomous vehicles allow them to interact with their surroundings. Autonomous vehicles are equipped with a variety of sensors such as light detection and ranging (LiDAR), infrared, sonar, inertial measurement units, and so on, as well as a communication subsystem to improve sensing accuracy. Visual sensors are typically employed to gather images, while computer vision, signal processing, machine learning, and other techniques are utilized to acquire, process, and extract data in order to model visual scenes for navigation, 2D and 3D scene reconstruction, object and obstacle detection or avoidance, tracking, recognition, control, and inference. The control subsystem analyzes sensory data to determine the best navigation path to its goal and an action plan for completing tasks [14].

A few of the challenges autonomous vehicles come up against are blind spots, unknown environments, non-line-of-sight scenarios, poor sensor performance due to algorithmic complexity, sensor errors, limited energy, limited computational resources, human–machine communications, size, and weight constraints, weather conditions. Several algorithmic approaches have been implemented to respond to these challenges, including sensor design, processing, control, and navigation [14].

In reference of image processing, there are multiple techniques such as principal or independent component analysis, point feature matching, linear filtering, neural network and so on.

3.2 Artificial Neural Networks

Neural Networks, are a subset of machine learning and are at the heart of deep learning algorithms [3]. They are also known as Artificial Neural Networks (ANNs) or Simulated Neural Networks (SNNs). Their goal is to recognize underlying relationships in a data set by imitating the behavior of a human brain and its neurons, thus their name and structure.

Artificial neural networks consist of node layers, an input layer, an output layer, and one or more hidden layers in between. In an ANN, each node of layer connects with every node of the next layer. Each node, or artificial neuron, is connected to another node and has an associated weight and threshold value. If a node’s output exceeds the specified threshold, that node is activated, and data is sent to the next layer of the network. Otherwise, no data is sent to the network’s next layer. The basic structure of an ANN is shown in Figure 11.

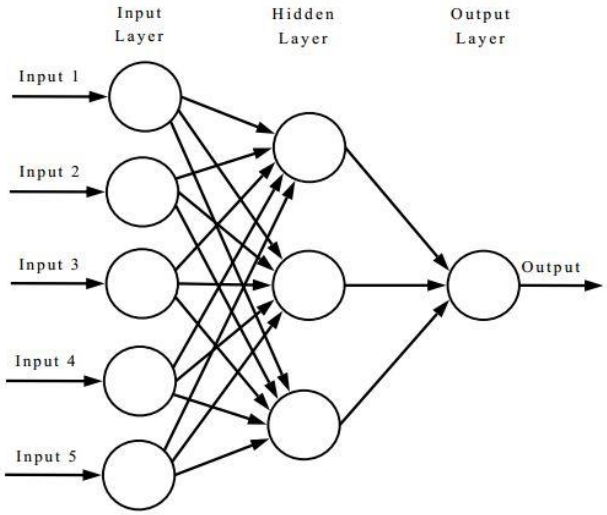


Figure 11 - Artificial Neural Network structure

In image processing tasks, supervised and unsupervised learning are the two main learning approaches, supervised and unsupervised learning [4].

Learning with pre-labeled inputs that serve as targets is referred to as supervised learning. There will be a set of input values (vectors) and one or more related defined output values for each training example. The purpose of this type of training is to lower the overall classification error of the model by correctly calculating the output value of each training example.

On the other hand, unsupervised learning is characterized by the absence of labels in the training set. The criterion for success is the ability of the network to lower or enhance an associated cost function. It's worth mentioning, though, that most image-based pattern recognition tasks rely on supervised learning for classification.

There are different categories of neural networks, used for different purposes. The most common types are Multi-Layer Perceptrons (MLPs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). For the purpose of this thesis, a CNN has been used.

3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs or ConvNets) were already used at the end of 90’s but have become extremely popular in our days because of the impressive impact they have had on the area of computer vision [5]. A CNN differentiates from an ANN in two ways.

CNNs are most commonly applied on analyzing and identifying visual imagery such as digital images or photographs [4]. This allows us to encode certain features in the architecture to recognize specific elements in the images. For example, a face consists of eyes, ears, a nose, a mouth. If we are trying to recognize a face, we look for those elements. Sometimes an element might be missing, such as an ear that is hidden behind the hair, but we still classify it as a face considering the rest of the features. Although at first, we need to know how these features look like. We also need to know those features’ relative size, where they should be placed. For all those operations a CNN is needed, because each of its layers learns a different level of abstraction [5].

In addition, as the word “convolutional” indicates, the mathematical operation of convolution is employed in at least one of its layers instead of the general matrix multiplication [4]. The main purpose of a convolutional layer is to detect features such

as edges, lines, color drops, etc. This means that once the network has learned a characteristic at a specific point in an image, it can detect it again in any other part of it. On the other hand, a densely connected neural network doesn't have this property and has to learn the characteristic again if it appears in a new location of the image. So, this is why convolution is a very interesting property in this case of networks [3].

In conclusion, CNNs are very good feature extractors when a dataset consists of images. This is, why in this thesis CNNs were chosen to be used.

3.4 CNN architecture

A CNN consists of three types of layers. Convolutional layers, pooling layers and fully connected layers. Despite the fact that a CNN just requires a few layers, there is no one-size-fits-all approach to creating a CNN architecture. That being said, throwing a few layers together and expecting it to work would be unrealistic. CNNs, like other types of ANNs, tend to follow a common architecture, as proven by reading related literature. Figure 12 illustrates a common architecture in which convolutional layers are stacked, accompanied by pooling layers and both being repeated before being fed forward to fully - connected layers [4].

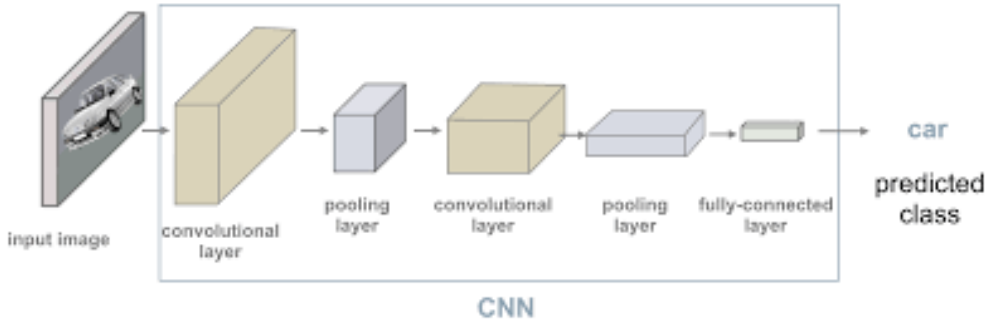


Figure 12 - Convolutional Neural Networks

As shown in Figure 13, another frequent CNN architecture is to stack two convolutional layers before a pooling layer. This is highly recommended since stacking many convolutional layers enables for the selection of more complex properties of the input vector. In this thesis is used a model that stacks at least two convolutional layers before a pooling layer, named VGG.

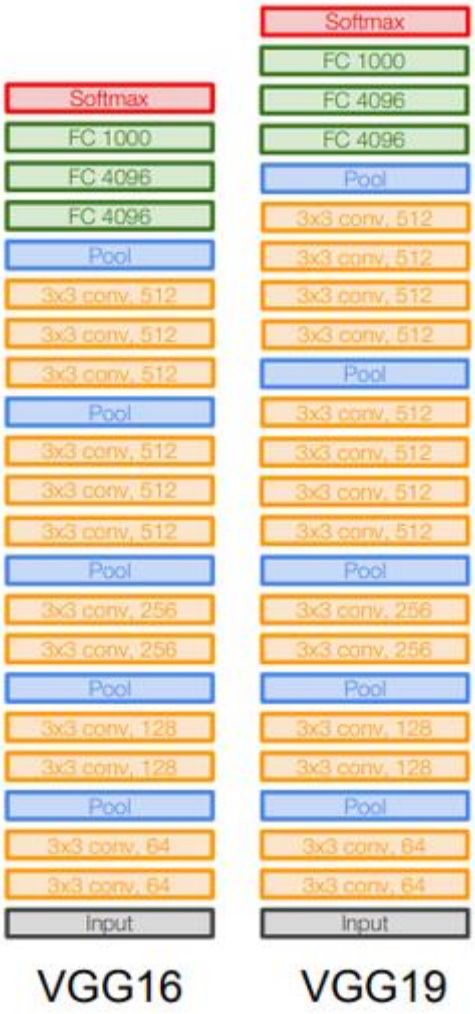


Figure 13 - VGG architecture

The abbreviation VGG stands for Visual Geometry Group. It is a typical deep Convolutional Neural Network (CNN) design with numerous layers. The term "deep" describes the quantity of layers, with VGG-16 or VGG-19 having 16 or 19 convolutional layers, respectively. Innovative object identification models are built using the VGG architecture. It is also remains one of the most often used image recognition architectures today.

Small convolutional filters of size 3 × 3 are used to build the VGG network and its input is of size 224 × 222. Thirteen convolutional layers and three fully connected layers constitute the VGG-16, while vgg19 has three more convolutional layers [6]. 1 × 1 convolution filters are used to transform the input linearly. The next part is a ReLU unit, shortening training time. Rectified linear unit activation function, or ReLU, is a piecewise linear function that, if the input is positive, outputs the input; otherwise, the output is zero. In order to maintain the spatial resolution after convolution, the convolution stride is kept at 1 pixel. All the hidden layers in the VGG network, also, use ReLU. There are three fully connected layers at the end the VGGNet. The first two levels each have 4096 channels, while the third layer has 1000 channels with one channel for each class that it can predict.

3.4.1 Convolutional layer

The convolutional layer, as its name suggests, plays a fundamental role in a CNNs operation. The convolution operation's goal is to extract high-level characteristics from

the input image and compress the images into a format that is easier to process while preserving elements that are important for obtaining a successful prediction.

Conventionally, the first convolutional layer is responsible for capturing low - level information such as edges, color, gradient direction, and so on. By adding layers the architecture adjusts to the high-level characteristics as well, giving a network that understands the photos in the dataset in the same way that people do. For example, a second layer can learn patterns composed of basic elements learned in the previous layer [4].

Convolutional layers can also learn spatial hierarchies of patterns by retaining spatial relationships, which is another feature of them. As mentioned before, a first convolutional layer, can learn basic elements like edges, while a second convolutional layer can learn patterns made out of the basic elements learnt in the prior layer. And so on, until it is able to recognize extremely complicated patterns. Convolutional neural networks are able to learn very complex and abstract visual concepts as a result of this [3].

The convolutional layer is defined by a few parameters that focus around the use of learnable kernels. The spatial dimensionality of these kernels is usually low, yet they spread along the entire depth of the input. When data is passed through a convolutional layer, the layer convolves each filter across the input's spatial dimensionality to create a 2D activation map (Figure 14) [4].

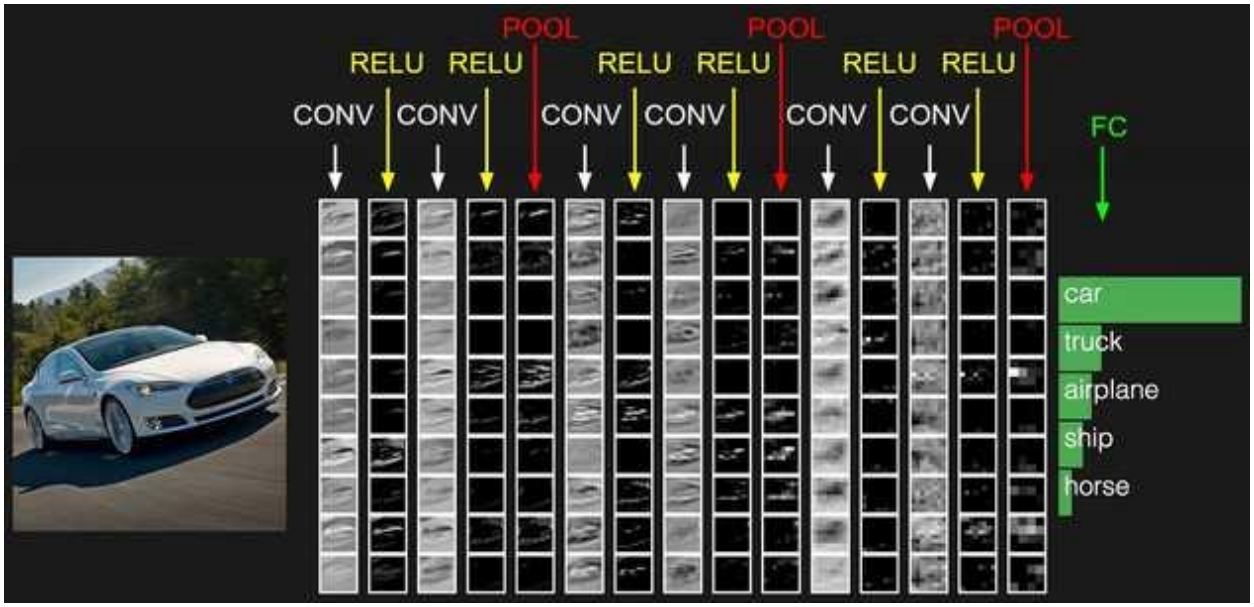


Figure 14 - Activation maps of an image

The scalar product is calculated for each value in that kernel as we move through the input. In a convolutional layer depicted in Figure 15, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation [8]. The network will then learn kernels that will 'fire' when they observe a specific feature at a specified spatial position in the input. These are referred to as activations [4].

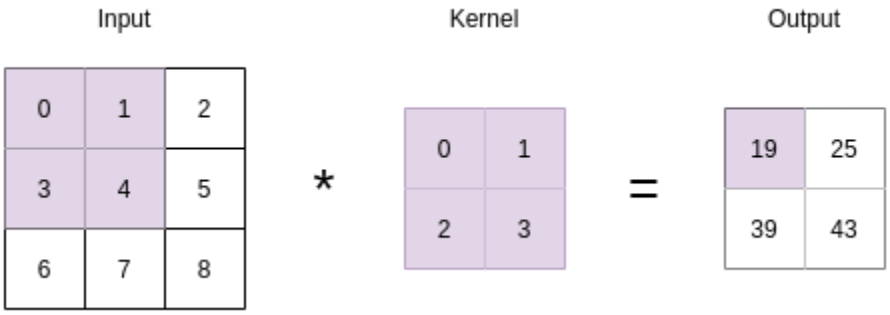


Figure 15 - Convolutional layer

The colored segments are the input and kernel tensor elements used for the output computation as well as the first output element which comes from the following calculations:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$

This sum is usually weighted. The output size is slightly less than the input size along each axis. We can only compute the cross-correlation for locations where the kernel fits entirely within the image because the kernel has a width and height higher than one. If the input tensor's dimensions are input_height and input_width and the kernel's dimensions are kernel_height and kernel_width, the output size is determined [8] by:

$$(input\_height - kernel\_height + 1) \times (input\_width - kernel\_width + 1)$$

Every kernel will have its own activation map, which will be stacked along the depth dimension to generate the convolutional layer's whole output volume.

As mentioned before ANN's layers are fully connected with each other as shown in Figure 11. In CNN's, on the other hand, every neuron in a convolutional layer is only connected to a small fraction of the input volume in order to not result in a model too large to train efficiently. This fraction's dimensionality is referred to as the receptive size of the neuron. The depth of the input is almost always equal to the magnitude of the connectivity through the depth.

Through the optimization of the model's output, convolutional layers can also greatly lower the model's complexity. Three hyperparameters, the depth, the stride, and the padding setting, are used to optimize these [4].

Size and number of filters per layer

The size of the kernel (kernel_height x kernel_width) that holds information from spatially close pixels is usually 3x3 or 5x5. It has been observed that as the size of the filter gets bigger, the model's accuracy is increased and model training lasts longer.

The number of filters that tells us the number of characteristics that we want to handle is usually 32 or 64. As more filters are added in a layer, the accuracy is increased and the training lasts longer.

Stride

This hyperparameter indicates the number of steps in which the filter kernel moves (or slides). Through stride we determine the depth around the spatial dimensions of the input. If we set the stride to 1, for example, we will get a massively overlapped receptive field with extraordinarily big activations. Setting the stride to a higher number, on the

other hand, will lower the distance traveled and decrease the size of the information that will be passed to the next layer [4].

Padding

The operation produces two types of results: one in which the dimensionality of the convolved feature is lowered when compared to the input, and the other in which the dimensionality is either increased or unchanged. This is accomplished by using Valid Padding in the first case and Same Padding in the second.

We discover that the convolved matrix is of dimensions 5x5x1 when we augment the 5x5x1 image into a 6x6x1 image and then apply the 3x3x1 kernel over it. As a result, the name "Same Padding" was coined. When we execute the same operation without padding, however, we get Valid Padding, which is a matrix with the same dimensions as the Kernel (3x3x1).

3.4.2 Pooling Layer

The Pooling layer, like the Convolutional Layer, is responsible for downsizing the Convolved Feature's spatial size. Through dimensionality reduction, the computational power required to process the data is lowered. It's also beneficial for extracting rotational and positional invariant dominant features, which helps to maintain the model's training process properly [4][5].

Pooling is divided into two types: maximum pooling and average pooling. The maximum value from the Kernel-covered area of the image is returned by Max Pooling. Average Pooling, on the other hand, returns the average of all values from the Kernel's section of the image.

Max Pooling serves as a noise suppressant as well. It removes all noisy activations and conducts de-noising as well as dimensionality reduction. Average Pooling, on the other hand, just conducts dimensionality reduction as a noise suppression approach. As a result, Max Pooling may be said to outperform Average Pooling.

Typically, the stride and filters of the pooling layers are both set to 2 2, allowing the layer to stretch throughout the input's spatial dimensions. Overlapping pooling can also be used, with the stride set to 2 and the kernel size set to 3. Because of the destructive nature of pooling, having a kernel size greater than 3 will usually result in a significant reduction in model performance.

It's also worth noting that, in addition to max-pooling, CNN architectures may include general-pooling. Pooling neurons in general pooling layers are capable of performing a variety of common operations such as L1/L2-normalization and average pooling.

3.4.3 Fully Connected Layer

The fully connected layer is composed of neurons that are directly linked to the neurons in the two adjacent layers, but not to any layers in between. This is similar to how neurons are placed in standard ANN models. (Figure 1). [4] This is usually added as the last layer of the network and used for classification. It is a (typically) low-cost approach of learning non-linear combinations of high-level information represented by the convolutional layer's output.

4. RELATED WORK

4.1 Image processing and classification using neural networks

Given that there is a dataset of images, needed for an autonomous vehicle to be employed. The images of this dataset need to be processed and categorized. Firstly, to gather, process, and extract information, visual sensors are used, and computer vision algorithms, signal processing, machine learning, and other approaches are used. Several algorithmic approaches have been implemented to address these issues, including sensor design, processing, control, and navigation. The goal of [7] is to give up-to-date information on the needs, algorithms, and major issues associated with the use of machine vision-based navigation and control methods in autonomous vehicles.

Quinonez et al [8] were looking to address a simple pattern recognition problem with a backpropagation learning neural network. Authors trained a Multi-Layered Perceptron, instead of a CNN that is used in this thesis, using the Scaled Conjugate Gradient algorithm, in order to detect four photos. The goal of this project was to provide a secure way for guiding a robot from an initial location to a target position via an obstacle-free path within a known or unknown environment while attempting to proceed with minimum cost. The initial position and the destination/target must have been indicated by the environment.

Authors in [9] propose an end-to-end deep learning-based approach for indoor mobile robot localization via image classification using a small dataset, retrieved from a real apartment setting using the MYNT EYE camera. Topological map information is collected from the robot workspace's floor plan. The topological map's navigable region is then extracted, and each of them is categorized as a robot workstation. In this way, the mobile robot is able to autonomously localize and traverse throughout the topological map. The dataset mentioned is utilized to train a CNN-based classifier model, which is subsequently used to localize and navigate the mobile robot or even detect potential obstructions in the robot's path. A very remarkable fact is that the authors present an acceptable cost function to address the CNN classifier model's limited generalization capability, which exists due to the dataset's small size. Unlike other well-known loss functions, the suggested loss considers not only the chance of the input data being allocated to its true class, but also the probability of the input data being allocated to other classes. The results indicated that the proposed system is efficient, effective, and generalizable.

4.2 Arrow marking detection and preprocessing

In order to create a dataset, the photos taken by the robot will need to be preprocessed. That is locating and extracting the arrow markings, so that they can later be classified.

Authors in [10] present a general geometric approach using curve-based prototype fitting. This suggests a two-step procedure which includes a preprocessing and a recognition step. The first step seems to be an interesting approach when it comes to extracting a marking and processing it. It consists of setting a region of interest (ROI) that will be framing the arrow marking, extracting the connected components of it and contouring the arrow’s outer geometric shape, and finally, performing a plausibility check by reconstructing the local environment and reprojecting the extracted contour pixel list on the road surface.

On the other hand, instead of extracting the arrow of the road image, the entire picture could be processed. The research work in [11] attempts to navigate a robot using a camera and identifying arrows. A track is made with various turns for the robot to move

around. On every turn, a right or left arrow is provided for robot to recognize the arrow symbol by capturing images using a wireless camera. There is an initial starting point and the final destination point, which the robot has to reach. Unlike [10], it doesn't extract the arrow marking from the picture of the road, but processes the entire photo as it is taken by the robot's camera. This photo's pre-processing includes converting it to a binary image and removing noise and small objects from it.

Similar tactic is used by [8], meaning that they also used the entire image, not just the arrow shape. Although before converting it to binary image, a median filter was applied on the images. Also, after these two steps, there was an attempt to reduce the image noise by removing objects smaller than 300 pixels.

Like [8] and [11], the work in [12] processes the entire image. Shojaeipour, Haris, and Khairir presented a method to navigate a mobile robot, which decides the shortest path for the robot to follow to reach its destination while avoiding obstacles in its path. For that goal they used image processing techniques, such as converting the input image from RGB to greyscale, finding the edges of objects in the image, and removing noise from it.

Another research work, that also has a little different goal than the previous ones is described in [13]. It presents an image-based omni-directional mobile robot guidance system in an indoor space with installed artificial ceiling landmarks. Just like the previous ones, this too is using image proccesing techniques in order to process the ceiling landmarks that include color space transformation, histogram equalization, color detection, filtering, object tracking and recording.

5. AUTONOMOUS NAVIGATION WITH THE USE OF NEURAL NETWORKS AND IMAGE RECOGNITION

5.1 Problem Definition

In this thesis, we aimed to navigate a robot on an unknown path by recognizing and following arrow markings and signs. For the image recognition, a CNN was used, which we trained with a dataset of arrow markings that we made. For dataset creation, a Turtlebot was used along with a raspberry pi camera. An image pre-processing technique was developed, in order to make network training and image classification more accurate.

5.2 Challenges

5.2.1 Vehicle's computational power and communications

Real-time navigation is a project that demands multiple resources. Usually an unmanned vehicle is equipped with a resource-constrained computer like a raspberry pi. The vehicle's raspberry pi used for this thesis doesn't have a strong processing power. Therefore, we needed a stronger device to manage and execute the pre-processing of an image and the image prediction using a neural networks. Consequently, a server-client model was created, where the Turtlebot behaved as a client and the other computer had the role of the server. These two communicated via Python sockets.

5.2.2 Navigation conditions

In real-time navigation, the vehicle's surroundings matter a lot. In this case, the colour of the ground and the lights or daylight can affect the vehicle's navigation accuracy and overall, the navigation's success. The darker the color of the floor, the less we had to modify and adjust our image pre-processing technique. As for the environment's lighting, daylight and its reflection to the floor's surface could cause a big problem to the image pre-processing, which, in turn, could cause a problem to the neural network's image prediction.

5.3 Dataset creation

The first step of our research is to make our own dataset, using a raspberry pi camera, physically supported by a structure made of LEGO Mindstorms.

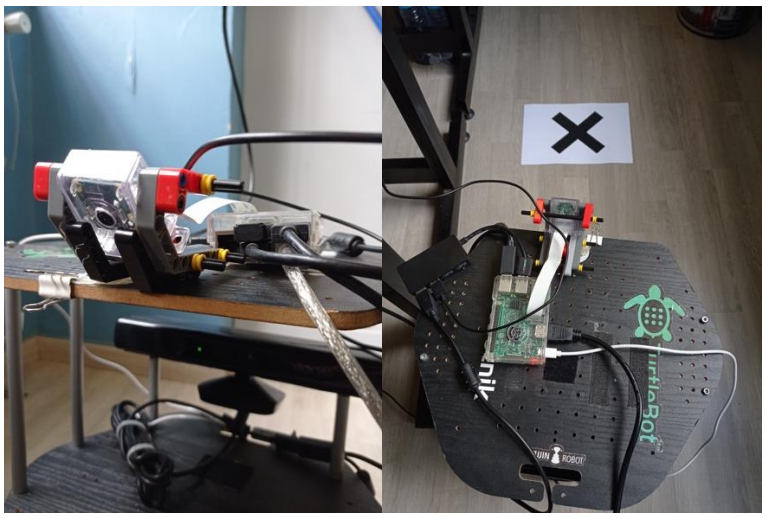


Figure 16 - Mindstorm structure for camera (left)

This structure is used to hold the camera in a certain angle towards the floor. For each navigation class, i.e. abort, go front, go backwards, go right and go left specific signs with arrows in paper were used. The abort class was represented by an X. Photos (n=100 samples) for each class were captured by the raspberry camera to create the training datasets.

Python functions were used to preprocess the images. We start by using a function that locates the sign in an image, in order to capture the photo and then classified by our model. This function uses area contour, binary masks, noise filters. After the sign is located and the area is cropped, we use a python script that flips vertically / horizontally the image, in order to augment the dataset.

5.4 Pre-Processing

As mentioned, we captured images with our raspberry camera to create a dataset. The same camera for the vehicle's real-time navigation was used. The dimensions of these images were bigger than those needed by a VGG model and contain a lot of background noise. Hence, it was needed to develop a function eliminating the background noise in our images and reduce the size. As a result the credibility of model training and real-time arrow classification could be increased.

To achieve this, an image pre-processing function was developed. This function aimed to locate a white area in each image containing a black sign. The function, as shown in Figure 17**Error! Reference source not found.** below, contained Gaussian noise filter, bitwise operations between images and contouring.

```python
def image_preprocessing(filepath):
    image = cv2.imread(filepath) #reads image
    image = cv2.resize(image, (672,672), interpolation = cv2.INTER_AREA) #resize photo in 672x672

    # blur = cv2.medianBlur(image, 5)
    blur = cv2.GaussianBlur(image,(5,5),0)
    gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)

    # find white areas
    (thresh, binary_white) = cv2.threshold(gray, 170, 255, cv2.THRESH_BINARY)

    # fill (closing) of dark regions in white areas
    kernel = np.ones(360)
    binary_white = cv2.morphologyEx(binary_white, cv2.MORPH_CLOSE, kernel)

    # mask only white rois from initial image
    masked_white_rois = cv2.bitwise_or(gray, 255-binary_white)

    # find black areas inside white areas of image
    masked_white_rois = cv2.medianBlur(masked_white_rois, 5)
    (thresh, binary_black) = cv2.threshold(255-masked_white_rois, 190, 255, cv2.THRESH_BINARY)

    contours = cv2.findContours(binary_black, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = contours[0] if len(contours) == 2 else contours[1]
    result = image.copy()
    cntr_area = [cv2.contourArea(cntr) for cntr in contours]
    if not cntr_area:
        print(filepath)
    cntr_indx = np.argmax(cntr_area)
    # for cntr in contours:
    cntr = contours[cntr_indx]
    x,y,w,h = cv2.boundingRect(cntr)
```

Figure 17 - Image pre-processing function

Therefore, this function located the signal in the photo, marked the area and cropped it, as inFigure 18. Then, the output of this function was the main input for model training or predicting the class of a sign in real-time navigation.

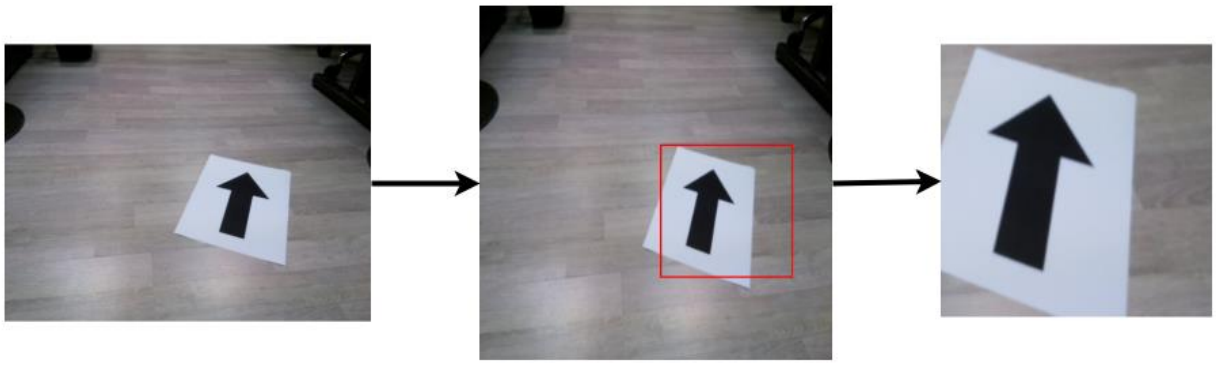
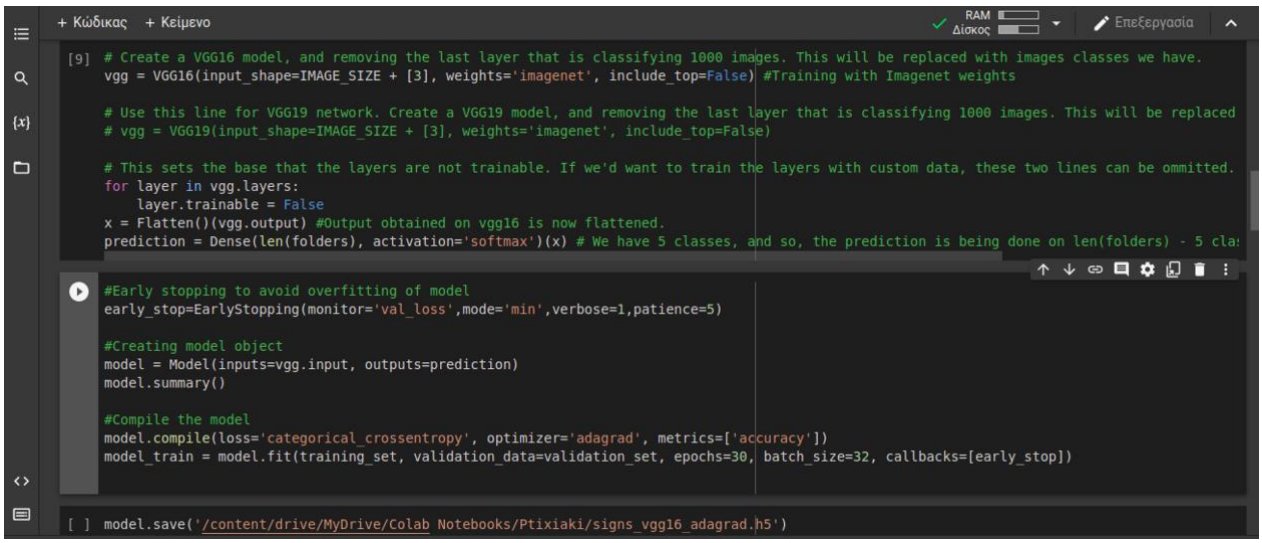


Figure 18 - Arrow markings pre-processing

5.5 Model

For this thesis, the VGG model was used. It was imported from a python library and its last layer, a dense layer responsible for classifying up to 1000 images of different classes, was removed. In the removed layer's place, a fully connected layer is added, in order to classify images of five categories. Training dataset was comprised of the images that were captured by the raspberry camera at the beginning of our research and then were processed by our image pre-processing function, i.e cropped and resized in 224x224. Resizing the in 224x224 was necessary, because VGG16 and VGG19 input dimensions are 224x224 exactly.

While training the model, a few parameters could be changed, such as the optimizer which was used. Hence, we trained multiple different models, that vary on those parameters.



```
[9] # Create a VGG16 model, and removing the last layer that is classifying 1000 images. This will be replaced with images classes we have.
    vgg = VGG16(input_shape=IMAGE_SIZE + [3], weights='imagenet', include_top=False) #Training with Imagenet weights

    # Use this line for VGG19 network. Create a VGG19 model, and removing the last layer that is classifying 1000 images. This will be replaced
    # vgg = VGG19(input_shape=IMAGE_SIZE + [3], weights='imagenet', include_top=False)

    # This sets the base that the layers are not trainable. If we'd want to train the layers with custom data, these two lines can be ommitted.
    for layer in vgg.layers:
        layer.trainable = False
    x = Flatten()(vgg.output) #Output obtained on vgg16 is now flattened.
    prediction = Dense(len(folders), activation='softmax')(x) # We have 5 classes, and so, the prediction is being done on len(folders) - 5 cla

    #Early stopping to avoid overfitting of model
    early_stop=EarlyStopping(monitor='val_loss',mode='min',verbose=1,patience=5)

    #Creating model object
    model = Model(inputs=vgg.input, outputs=prediction)
    model.summary()

    #Compile the model
    model.compile(loss='categorical_crossentropy', optimizer='adagrad', metrics=['accuracy'])
    model_train = model.fit(training_set, validation_data=validation_set, epochs=30, batch_size=32, callbacks=[early_stop])

[ ] model.save('/content/drive/MyDrive/Colab Notebooks/Ptixiaki/signs_vgg16_adagrad.h5')
```

Figure 19 - Training a VGG model

5.6 Navigation code

A python script was created responsible fot the navigation of Turtlebot in space. The script was compatible with ROS libraries and topics like rospy, rospkg, geometry_msgs, nav_msgs.

```
9   counter = 0
10  oldX = 0.0
11  oldY = 0.0
12  PI = 3.14
13  rotation = None
14
15  # Class responsible for moving the Turtlebot
16  class RobotMovement:
17
18      def __init__(self):
19          rospy.init_node("move_robot")
20          self.vel_publisher = rospy.Publisher("/cmd_vel_mux/input/navi", Twist, queue_size=1)
21          self.move = Twist()
22
23
24      def stop(self):
25          self.move.linear.x = 0.0
26          self.move.angular.z = 0.0
27          while True:
28              if self.vel_publisher.get_num_connections() > 0:
29                  self.vel_publisher.publish(self.move)
30                  break
31
32
33      def move_forward(self, dist):
34          self.move.linear.x = dist
35          self.move.angular.z = 0.0
36          # Wait to connect to topic
37          while True:
38              if self.vel_publisher.get_num_connections() > 0:
39                  self.vel_publisher.publish(self.move)
40                  break
```

Figure 20 - Navigation code init, stop, forward

```
def rotate(self, clockwise=True):
    speed = 90
    angle = 90
    #Converting from degrees to radians
    angular_speed = speed*2*PI/360
    relative_angle = angle*2*PI/360

    #We won't use linear components
    self.move.linear.x = 0
    self.move.linear.y = 0
    self.move.linear.z = 0
    self.move.angular.x = 0
    self.move.angular.y = 0

    # Checking if our movement is CW or CCW
    if clockwise:
        self.move.angular.z = -abs(angular_speed)
        print("Rotating clockwise")
    else:
        self.move.angular.z = abs(angular_speed)
        print("Rotating counter clockwise")

    # Setting the current time for distance calculus
    t0 = rospy.Time.now().to_sec()
    current_angle = 0
    while(current_angle < relative_angle):
        while True:
            if self.vel_publisher.get_num_connections() > 0:
                self.vel_publisher.publish(self.move)
                break
        t1 = rospy.Time.now().to_sec()
        current_angle = angular_speed*(t1-t0)

    #Forcing our robot to stop
    self.move.angular.z = 0
    while True:
        if self.vel_publisher.get_num_connections() > 0:
            self.vel_publisher.publish(self.move)
            break
```

Figure 21 – Navigation code rotate

```
def move_backwards(self, dist):

    speed = 90
    angle = 180
    #Converting from angles to radians
    angular_speed = speed*2*PI/360
    relative_angle = angle*2*PI/360

    #We wont use linear components
    self.move.linear.x = 0
    self.move.linear.y = 0
    self.move.linear.z = 0
    self.move.angular.x = 0
    self.move.angular.y = 0
    self.move.angular.z = -abs(angular_speed)
    print("Rotating 180 degrees")

    # Setting the current time for distance calculus
    t0 = rospy.Time.now().to_sec()
    current_angle = 0
    while(current_angle < relative_angle):
        while True:
            if self.vel_publisher.get_num_connections() > 0:
                self.vel_publisher.publish(self.move)
                break
        t1 = rospy.Time.now().to_sec()
        current_angle = angular_speed*(t1-t0)
    time.sleep(5)

    #Forcing our robot to stop
    self.move.angular.z = 0
    self.vel_publisher.publish(self.move)

    self.move.linear.x = dist
    self.move.angular.z = 0.0
    while True:
        if self.vel_publisher.get_num_connections() > 0:
            self.vel_publisher.publish(self.move)
            break
```

Figure 22 - Navigation code backwards

While the robot moves and follows the path, its coordinates are monitored using the following function in Figure 23 and then noted in a .txt file of this form in Figure 24. Then this file, is read by a Matlab script, which depicts the robot's trajectory as in Figure 53 and Figure 56.

```
19  # Getting odometry data
20  def PositionCB(data):
21      global oldY, oldX, rotation
22      oldX = data.pose.pose.position.x
23      oldY = data.pose.pose.position.y
24      rotation = data.pose.pose.orientation
25
```

Figure 23 - Odometry function

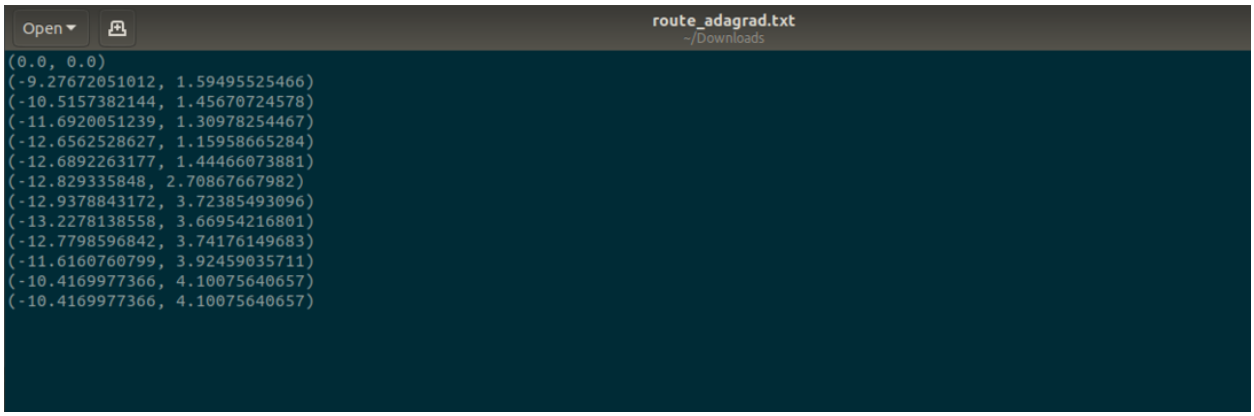


Figure 24 - Robot's coordinates after each move

5.7 Socket Communication

Turtlebot and server communicated via Python sockets using the python library “socket”. Only one socket was used for this communication. First, the client captured an image with the raspberry camera and sent it to the server through the socket.

```
53  # 3. Snap photo
54
55  sending_image = 'send_images2/send{counter}.jpeg'.format(counter=counter)
56  camera.capture(sending_image)
57  print("Snapshot taken.")
58
59  # 4. Send photo to Maestro4
60
61  file = open(sending_image, 'rb')
62  file_data = file.read(BUFFER_SIZE)
63
64  while file_data:
65      client.send(file_data)
66      file_data = file.read(BUFFER_SIZE)
67  client.send(b"edima")
68
69  file.close()
70
71  # 5. Receive result from Maestro4
72  recv_data = client.recv(BUFFER_SIZE)
73  print(recv_data)
74
```

Figure 25 - Client implementation

The server received this image, preprocessed it, classified it using a VGG model and sent an answer to the client about the classification, through the same socket.

```
    # 6. Predict class of photo

    y_pred=model.predict(np.expand_dims(cropped_image, axis=0))
    y_pred=np.argmax(y_pred,axis=1)
    # print(y_pred)


    # 7. Send result to Turtlebot

    if y_pred == [0]:
        client_socket.send(b"0")
        print("Stop.")
        break
    elif y_pred == [1]:
        client_socket.send(b"1")
        print("Go backwards.")
        time.sleep(3)
    elif y_pred == [2]:
        client_socket.send(b"2")
        print("Go forward.")
        time.sleep(2)
    elif y_pred == [3]:
        client_socket.send(b"3")
        print("Turn left.")
        time.sleep(5)
    else:
        client_socket.send(b"4")
        print("Turn right.")
        time.sleep(5)

```

Figure 26 - Prediction of model from server side

Then, the client received the answer about which class the arrow belongs to and moved accordingly. This process was repeated until an “abort” sign was detected and the socket closed.

6. EXPERIMENTS

For our research, we started by conducting experiments involving our neural networks. Based on these results, the parameters of a neural network were fine tuned that would better fit our purpose, which was for the robot to follow an unknown path of arrows. We further studied the behavior of these parameters and two neural networks with different parameters were trained and compared regarding their performance while the robot was following a path.

6.1 Experiments on CNN

For experimenting and training with our neural network, Google Colab was used. We considered Python Notebooks to be very useful for this purpose, along with Colab's GPU. The experiments were conducted using the dataset we made on both VGG16 and VGG19 models.

6.1.1 Epochs

Experiments were conducted concerning the model's behaviour when we trained it with different number of epochs, starting with ten (10) epochs and adding 10 on each training, until we reached one hundred (100) epochs. For the purpose of this experiment, a batch size was set to 32, loss function is Categorical Crossentropy and Adam is used as optimizer.

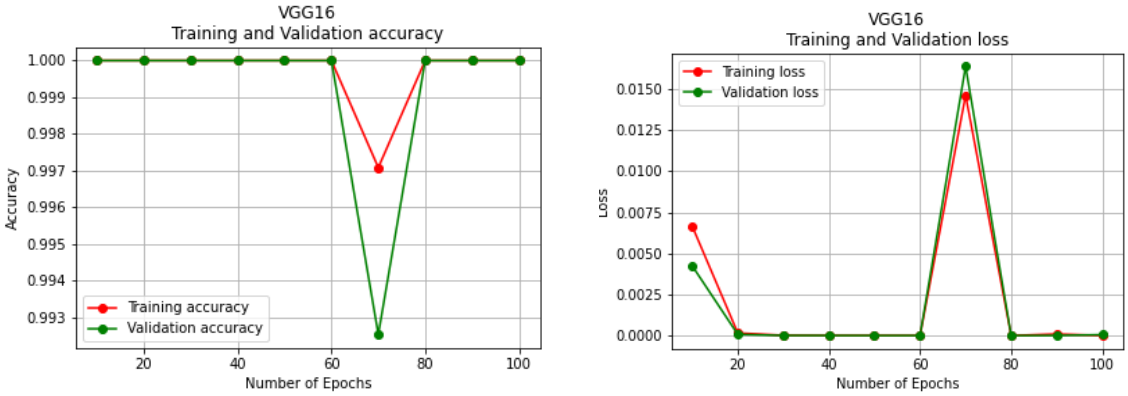


Figure 27 - accuracy (left) and loss (right) estimation of VGG-16 model and, batch size=32

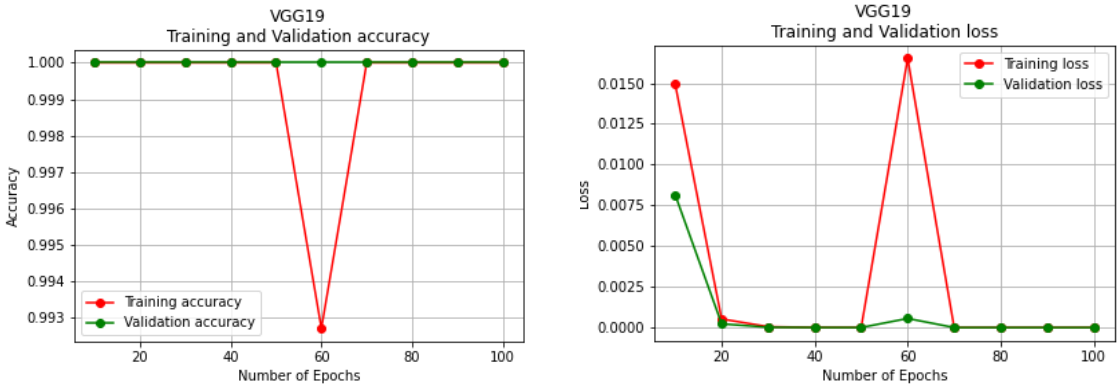


Figure 28 - accuracy (left) and loss (right) estimation of VGG-19 model

In all graphs, a peak was spotted, which was created from early stopping in comparison to a training of a model with 60 epochs. In loss graphs of Figure 27 and Figure 28 as it is shown that the loss decreased at first, while number of epochs was increased. That is

expected because the more the epochs, the less the loss is. Also, due to the fact that we have a small dataset, the loss can easily reach the value of zero, while accuracy can reach the value of one. Therefore, it was decided that the models that will be used for navigation, will be trained for 25 epochs.

6.1.2 Batch size

In the following figures we conducted experiments by changing the batch size while the rest parameters where epochs=30, loss function= “Categorical Crossentropy” and optimizer= Adam.

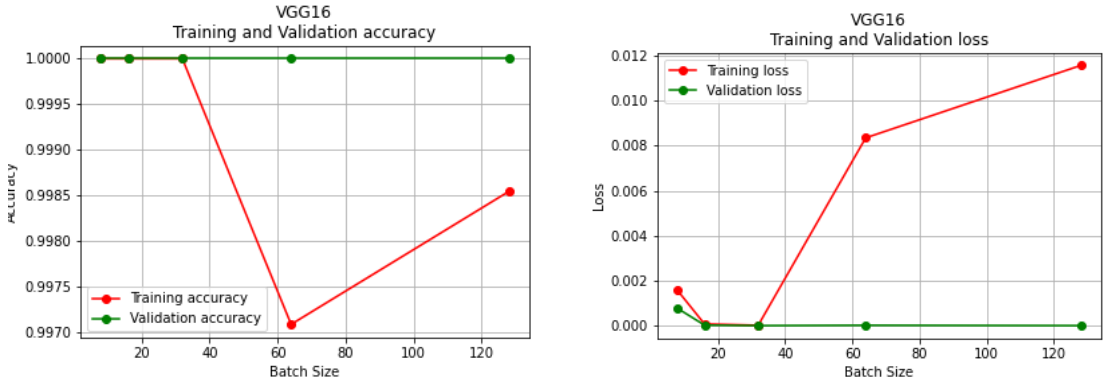


Figure 29 - accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=30

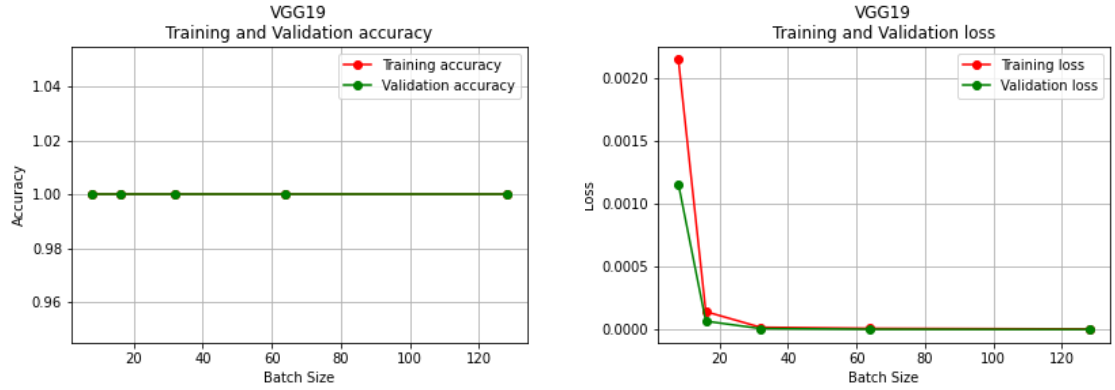


Figure 30 - accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=30

For VGG16 as shown in Figure 29 we can spot an expected behavior. As batch size increased, accuracy slightly decreased and loss increased. Due to the fact that batch size becomed bigger, the training in a certain number of epochs would be less accurate. As for VGG19 in Figure 30, accuracy had the value of one, because of our small dataset. Loss tends to decreased, but if we noticed better, the decrement was very small. It was decided that for the models used for navigation, batch size would be set to 32 while they were trained.

6.1.3 Loss function and Optimizer

For our research, we also experimented with loss functions and optimizers. As loss functions we only used two:

- Categorical Crossentropy and
- Mean Squared Error.

Categorical crossentropy is used in multi-class classification tasks, where an example can only belong to one out of many alternative categories, and the model is expected to

decide which one, while Mean Squared Error loss is calculated as the average of the squared differences between the predicted and actual values.

As optimizers we used four:

- Adagrad,
- Adam,
- RMSprop() and
- Stochastic Gradient Descent.

All possible combinations were made. The Gradient Descent algorithm calculates the gradient for the entire dataset and updates the values in the direction that is opposite to the gradients until a local minima is discovered. As long as the Gradient Descent generates large updates for uncommon parameters and tiny updates for frequent parameters, Adagrad is better suitable for a sparse data set. Adaptive Moment Estimation is known as Adam. This one too, determines numerous learning rates. Adam is quicker, more effective, and works well in practice. The basic idea behind RMSprop, which stands for Root Mean Square Propagation, is to keep a moving (discounted) average of the gradients' square and divide the gradient by the average's root.

6.1.3.1 Loss function = Categorical Crossentropy and VGG16

The following figures concern VGG16 models trained with Categorical Crossentropy as loss function.

Categorical Crossentropy and Adam

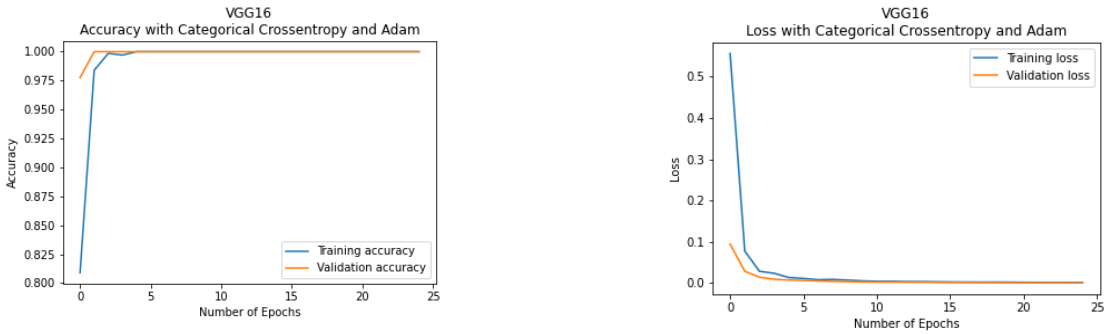


Figure 31 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size = 32

Categorical Crossentropy and Adagrad

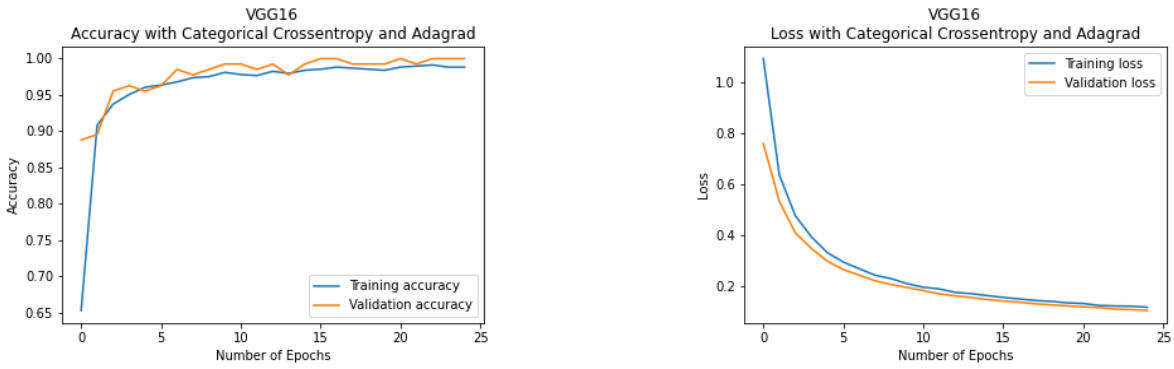


Figure 32 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size= 32

Categorical Crossentropy and RMSprop

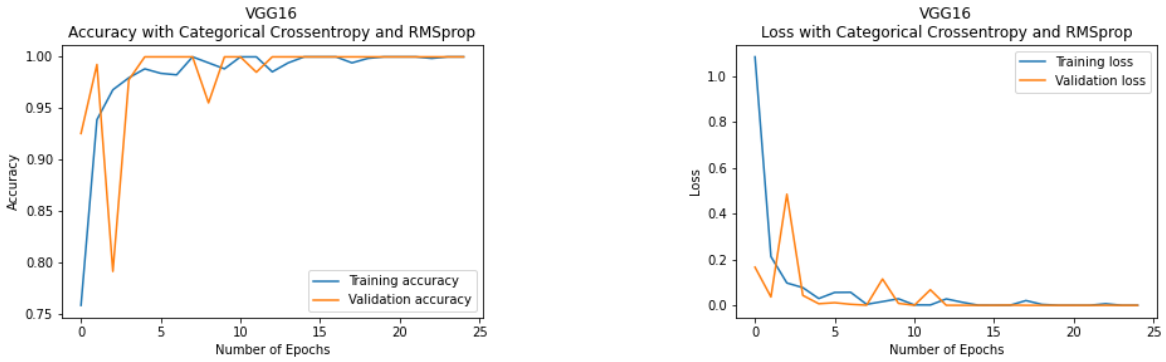


Figure 33 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size= 32

Categorical Crossentropy and SGD

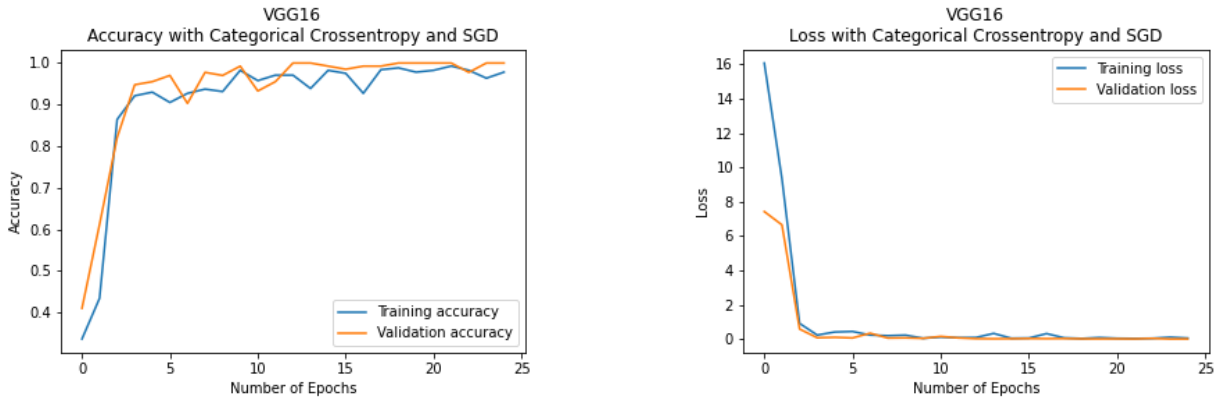


Figure 34 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size=32

It can be pointed out that Categorical Crossentropy along with Adam and Adagrad seem to behave nicely, meaning that their figures' curves (Figure 31 and Figure 32 respectively) seem to be smooth. Also, their loss tends to decrease and their accuracy to increase as the number of epochs increase, which is expected. On the contrary, Categorical Crossentropy with RMSprop or SGD (Figure 33 and Figure 34 respectively) don't seem to be good combinations as it seems from their figures that they should not be very stable.

6.1.3.2 Loss function = MSE and VGG16

The following figures concern experiments conducted on VGG16 models when MSE is used as loss function.

MSE and Adagrad

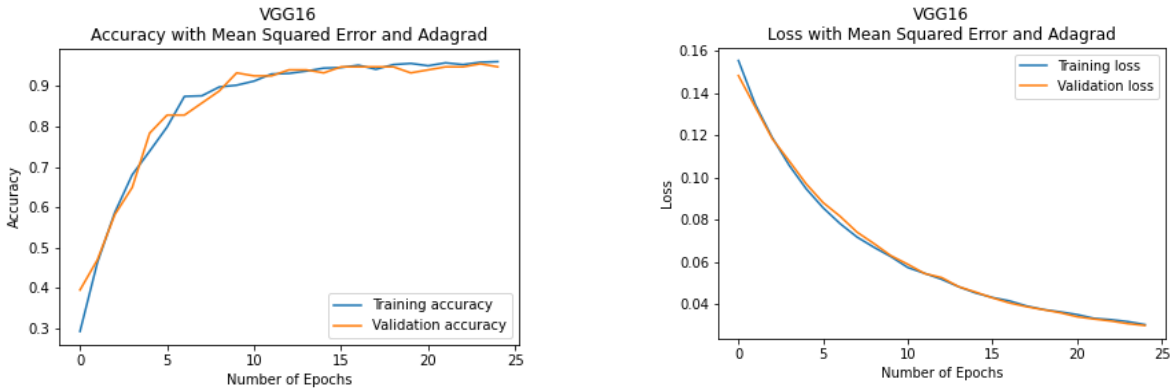


Figure 35 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size=32

MSE and Adam

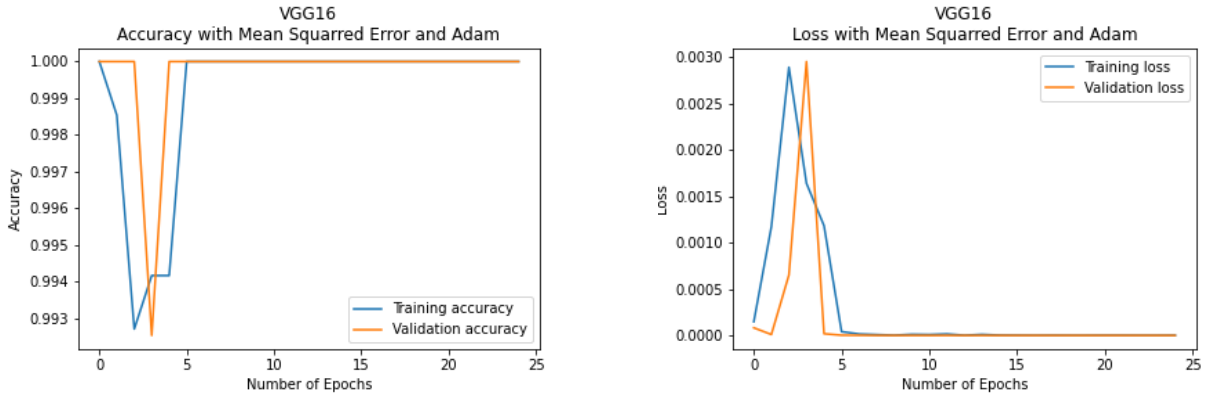


Figure 36 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size=32

MSE and SGD

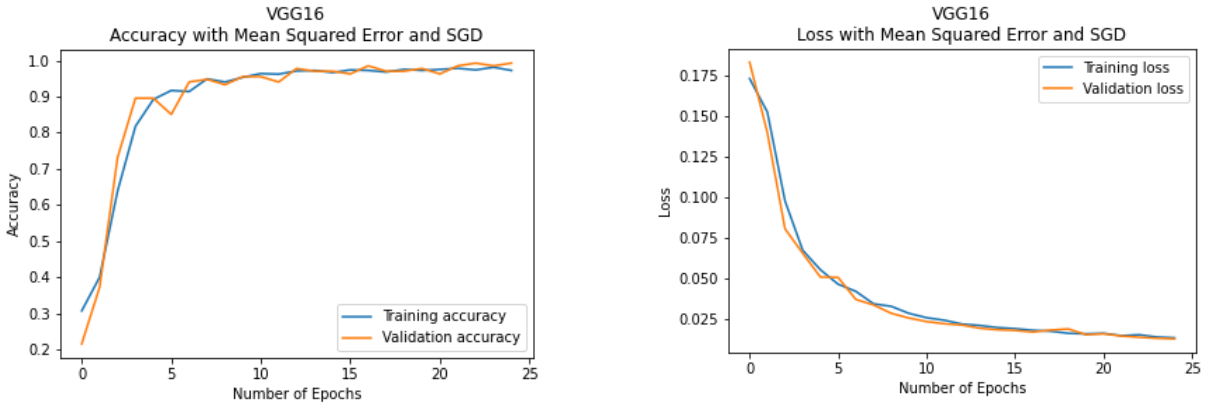


Figure 37 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size=32

MSE and RMSprop

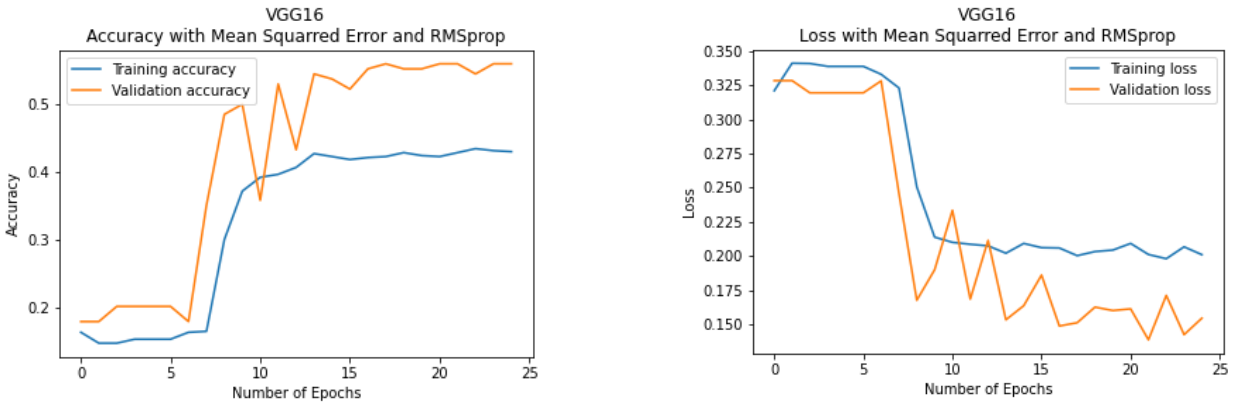


Figure 38 - Accuracy (left) and loss (right) estimation of VGG-16 model and, epochs=25 and batch size=32

On the one hand, MSE with SGD (Figure 37) and Adagrad (Figure 35) seem to behave as expected. Their figures' curves seem to be smooth. Also, their loss tends to decrease and their accuracy to increase as the number of epochs increase, which is required. On the other hand, models with MSE with RMSprop (Figure 38) or Adam (Figure 36) don't seem to be good combinations as it seems from their figures that they should not be very stable.

6.1.3.3 Loss function = Categorical Crossentropy and VGG19

The following figures result from experiments conducted on VGG19 and Categorical Crossentropy.

Categorical Crossentropy and Adam

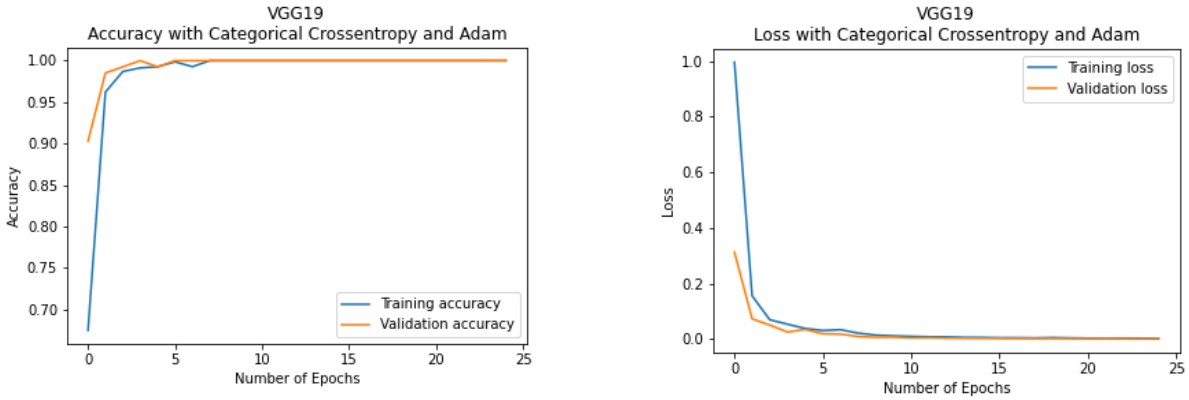


Figure 39 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

Categorical Crossentropy and Adagrad

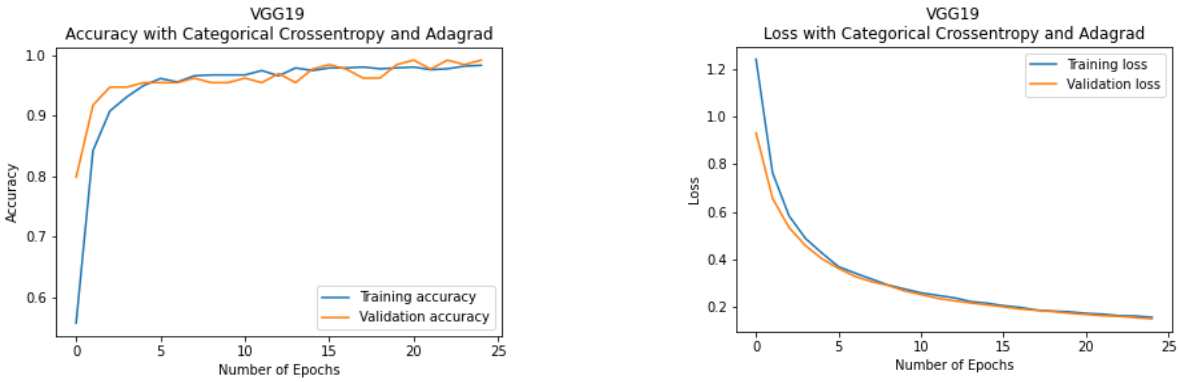


Figure 40 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

Categorical Crossentropy and RMSprop

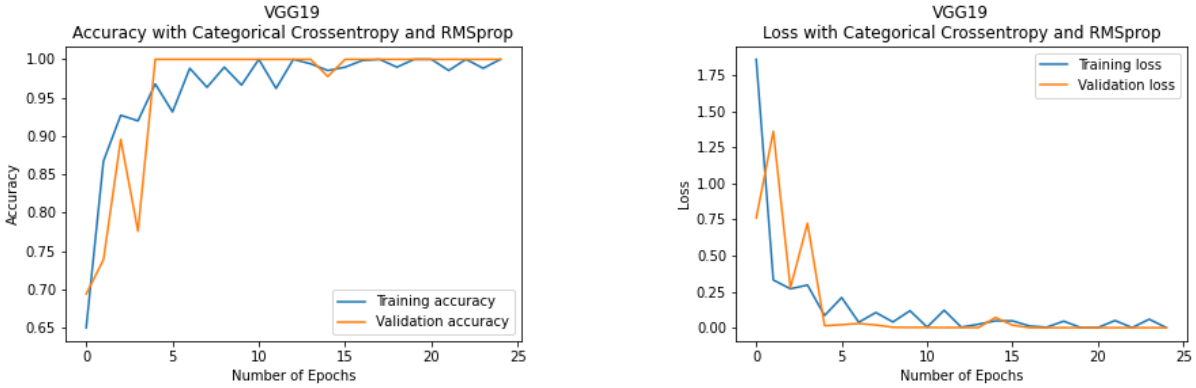


Figure 41 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

Categorical Crossentropy and Stochastic Gradient Descent

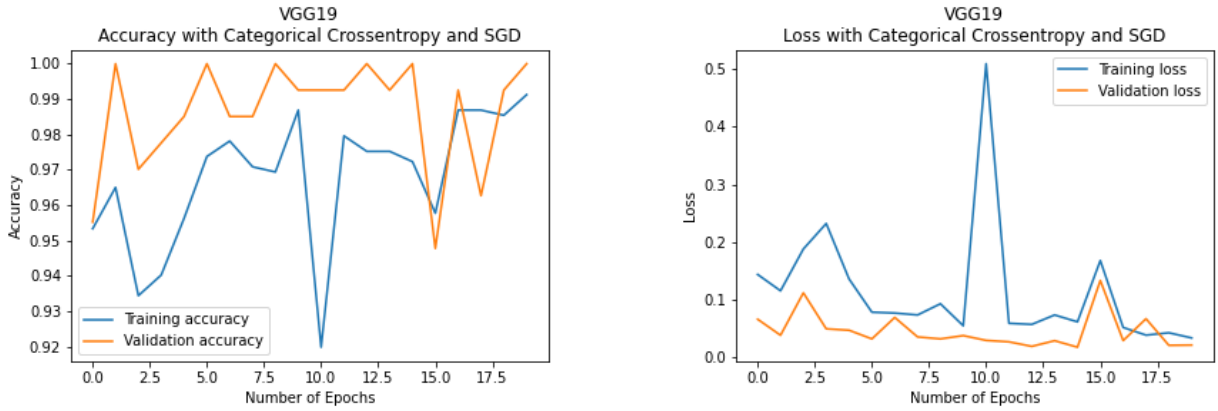


Figure 42 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

Adam (Figure 39) and Adagrad (Figure 40) models' curves seem to be smooth. Their loss tends to decrease and their accuracy to increase as the number of epochs increase unlike RMSprop or SGD (Figure 41 and Figure 42 respectively), whose diagrams are not at all stable.

6.1.3.4 Loss function = MSE and VGG19

The figures that follow concern VGG19 models trained with MSE.

MSE and Adagrad

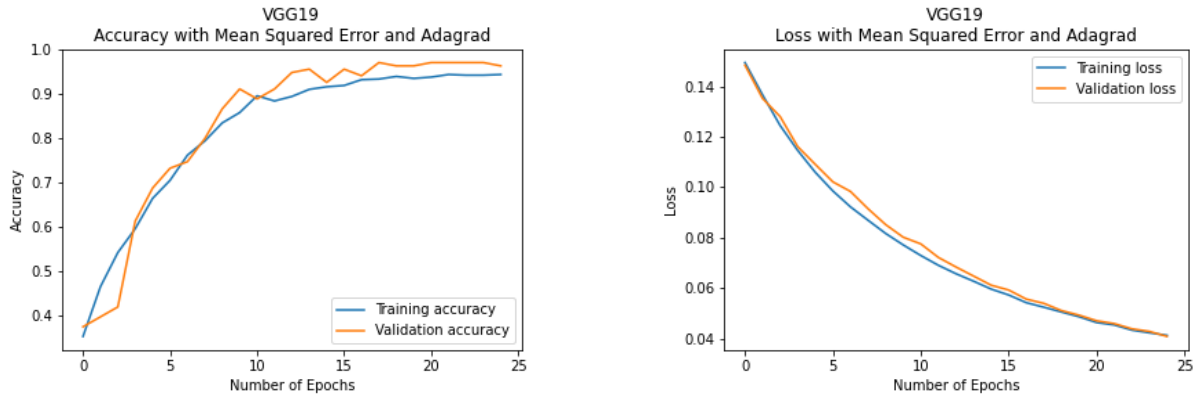


Figure 43 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

MSE and Adam

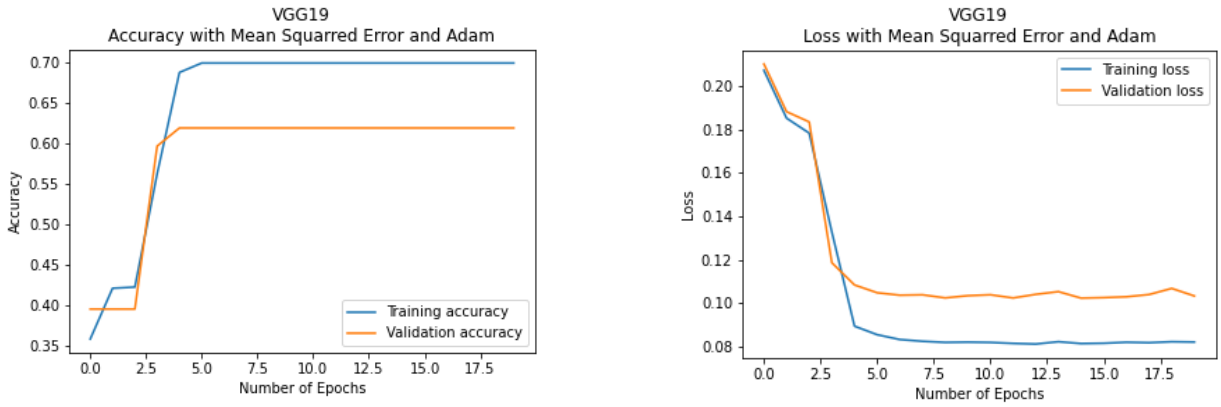


Figure 44 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

MSE and Stochastic Gradient Descent

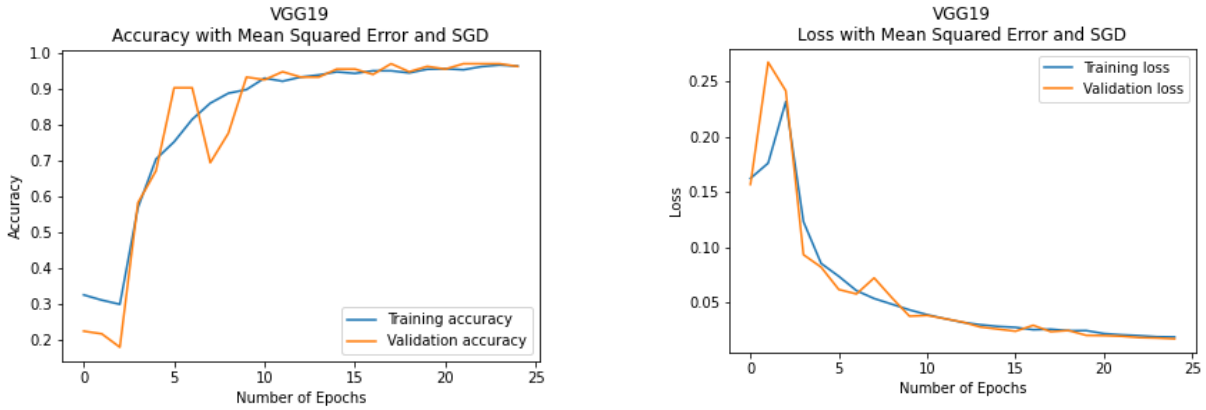


Figure 45 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

MSE and RMSprop

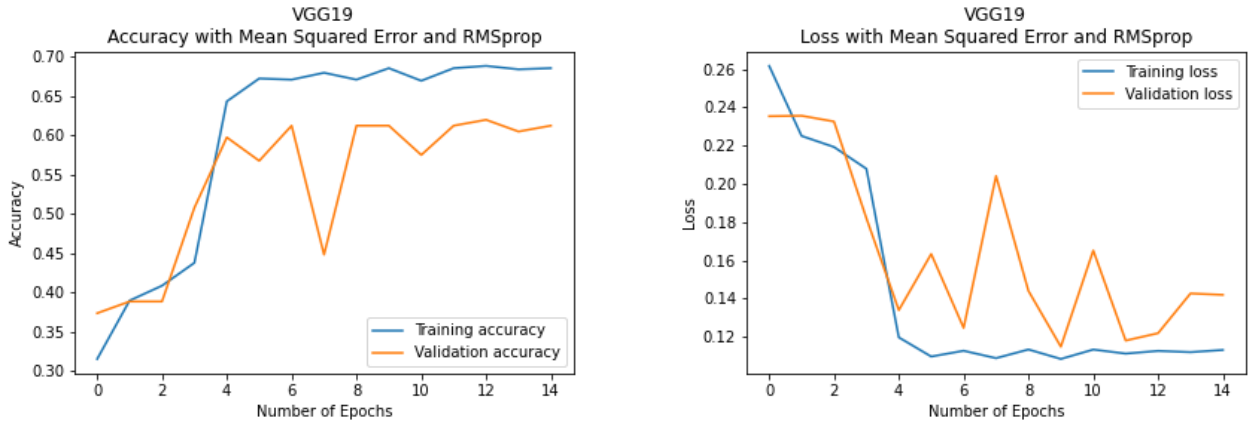


Figure 46 - Accuracy (left) and loss (right) estimation of VGG-19 model and, epochs=25 and batch size=32

MSE with Adam and Adagrad seem to function as required, based on Figure 45 and Figure 44, while models with SGD or RMSprop curves (Figure 45 and Figure 46 respectively) aren't stable.

In conclusion, models trained with 25 epochs and 32 batch size are chosen for the real-time navigation of our vehicle. Apparently, categorical crossentropy seems to suit better this purpose, along with Adam and Adagrad as optimizers, according to the conclusions made based on all figures. Therefore, we trained 2 models and compared their performance on the second phase of our real time navigation set experiments. A VGG16 with Categorical Crossentropy and Adam and a VGG16 with Categorical Crossentropy and Adagrad are discussed in the following sections.

6.2 Path following

6.2.1 Set up experiments

First, we set up a path of arrow markings on the floor. Figure 47 shows the path's top view, where the green circle represents Turtlebot at its starting point.

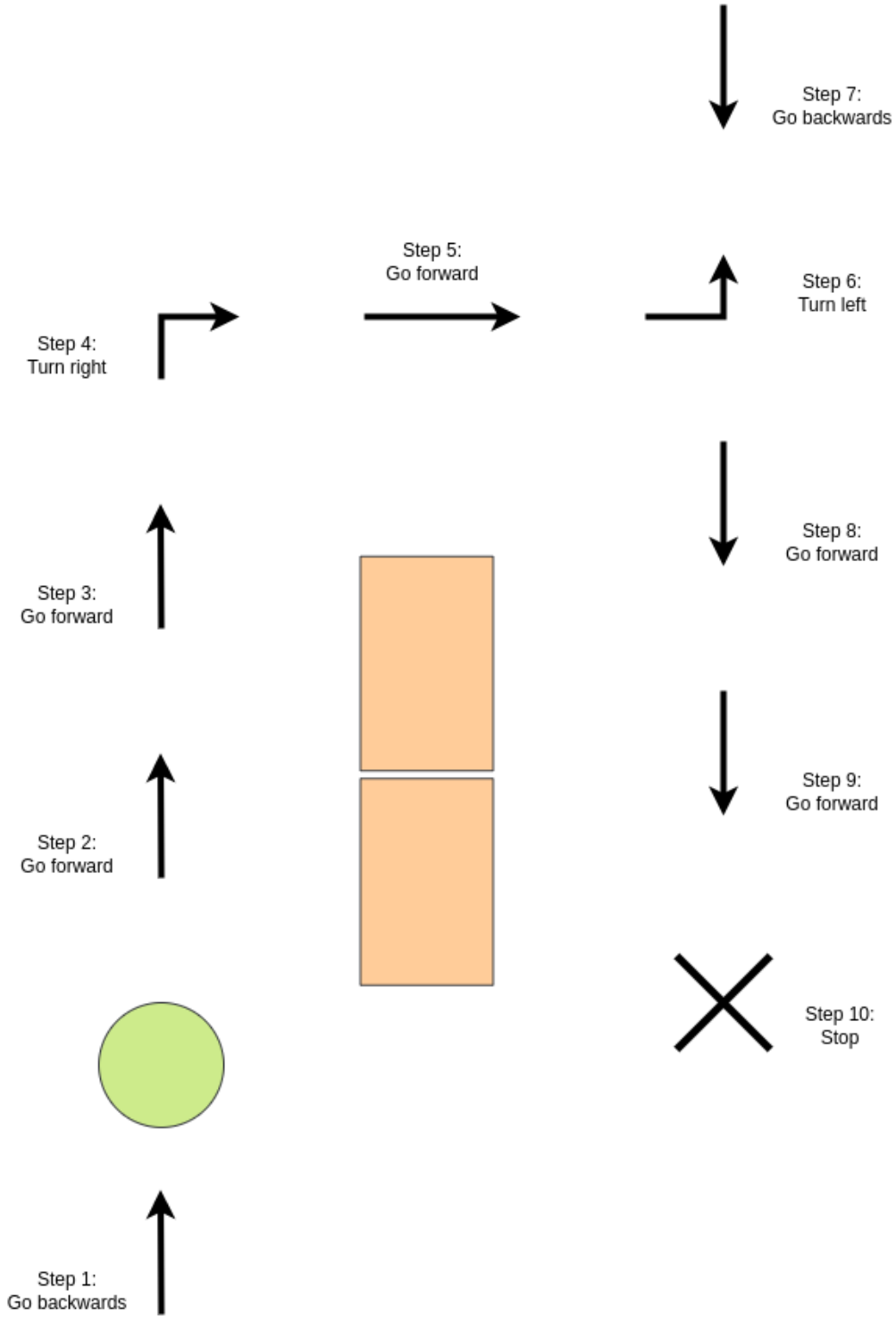


Figure 47 – Map's top view

Turtlebot starts facing the arrow of step 1. Then, it has to follow the rest of the steps. Also, the distance between each marking is very specific, because our robot's moves are also very specific. For example, the marking that follows a "go forward" marking is placed after 1.10 m or the marking that follows a "go backwards" marking is placed after 1.70 m. As shown, all kinds of arrow markings were used.

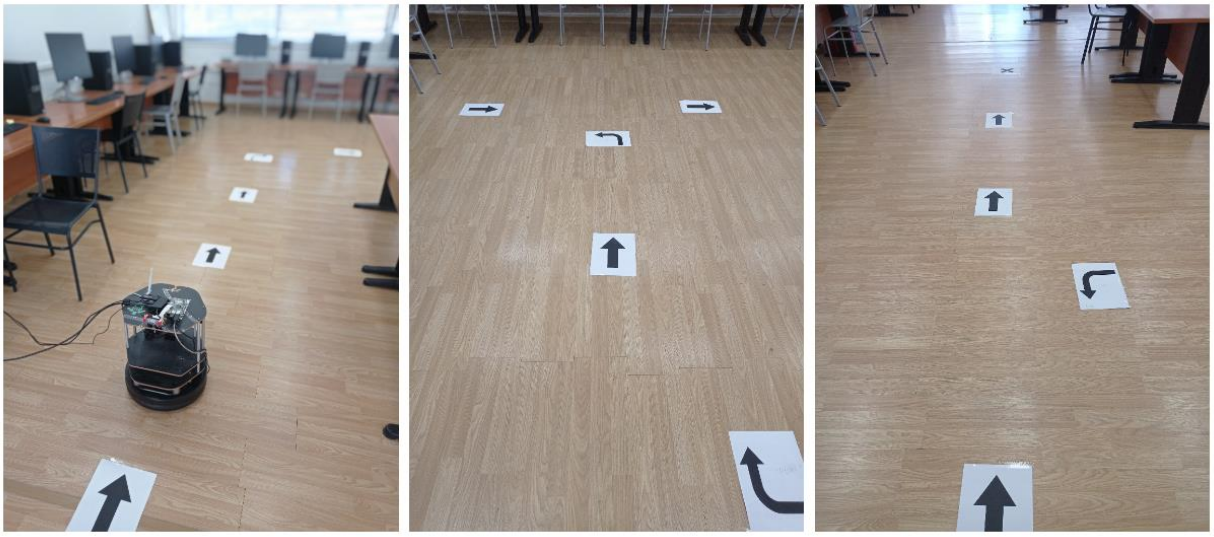


Figure 48 – Path in space

The robot has to follow this path and classify the pictures captured with the two different models we decided on the previous chapter.

As mentioned before it is very crucial that we created the best possible environment for our navigation to be successful. We deprived the room of daylight and turned all the lights on, so that the lighting is stable and doesn't depend on whether there is sun outside or not. After all, the sunlight reflects on the floor and makes the pre-processing part a lot harder and less successful.

6.2.2 Experiment execution

Before starting navigation experiments, we ran a few tests regarding image pre-processing. We adjusted a few parameters in the function in order to ensure that the markings will be spotted in the most efficient way possible. As a result, the image classification will be more credible. Then, the experiments on navigation can begin.

First, we execute the server's code. That will load the chosen VGG model and the set up socket. We wait until the server prints out a message that is ready to accept requests from a client.

```
root@b520896b4c5f:/home/tensorflow# python3 experiment_server.py
Loading VGG model...
I can now accept requests from clients.
```

Figure 49 – Executing the server

Then, we deploy the client's code. The client captures the first image sends it via socket to the server. The server receives the image, spots the marking, crops its area and resizes the resulting image as 224x224, so that it can fit the VGG's input dimensions, while the client waits for an answer. The server predicts the class that single image belongs to and sends the result, as an answer, back to the client. The client receives this answer and moves accordingly. That procedure is repeated until an "abort" sign is located.

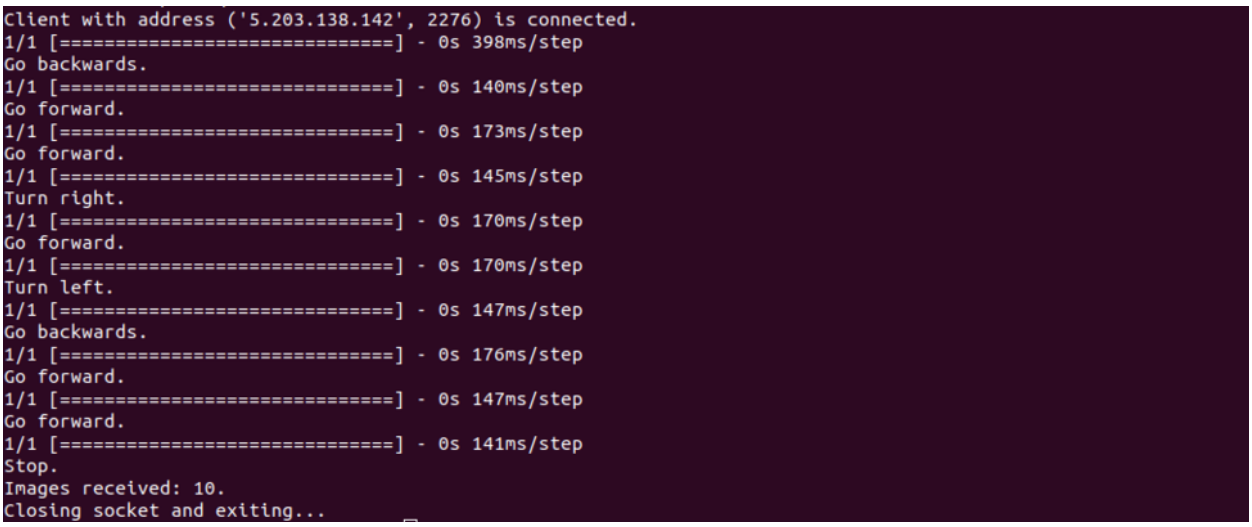


Figure 50 – Server messages about classification results

The entire procedure that was just described will be done twice, once for each model that we have chosen.

6.2.3 Experiment results

6.2.3.1 VGG16 with Categorical Crossentropy and Adam

We performed one experiment with a model trained for 25 epochs, with batch size 32, categorical crossentropy as a loss function and Adam as optimizer. This model is used by the server for image classification.

The robot had to follow the route on Figure 47 and as it turned out, the navigation was successful. It classified all 10 pictures correctly. In figure are shown all images that were received by the server. Tenth image, should contain the entire marking. That marking represents “abort”. All other images depict markings correctly.

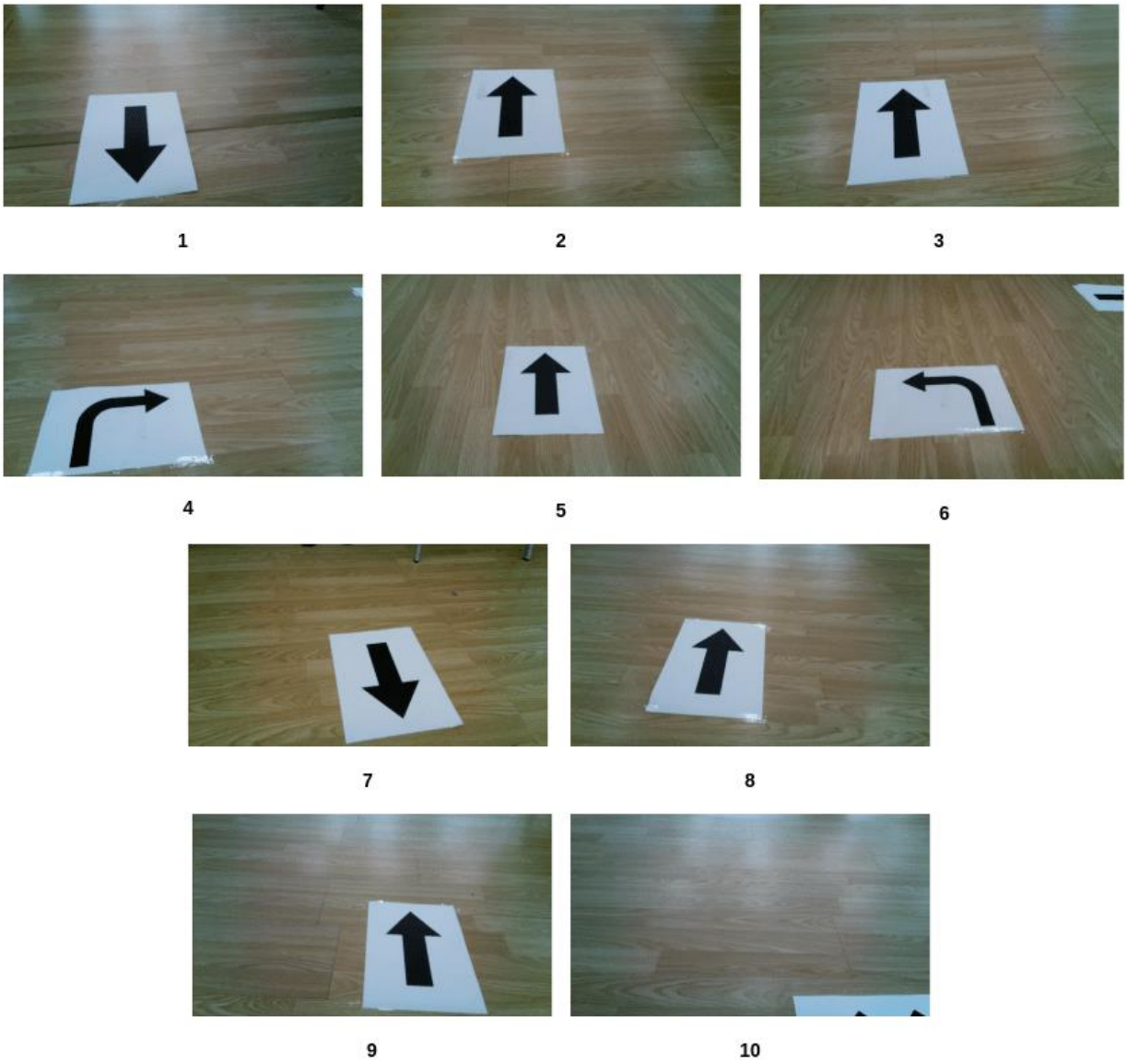


Figure 51 – Received images by server

The images that resulted from pre-processing and are destined for classification, are shown inFigure 51. The tenth image of this figure comes from the tenth image of Figure 52, which was captured from the robot inaccurately. This is the best possible outcome of pre-processing for this image. Even though it is not a credible result, it was classified correctly leading the navigation to end successfully.

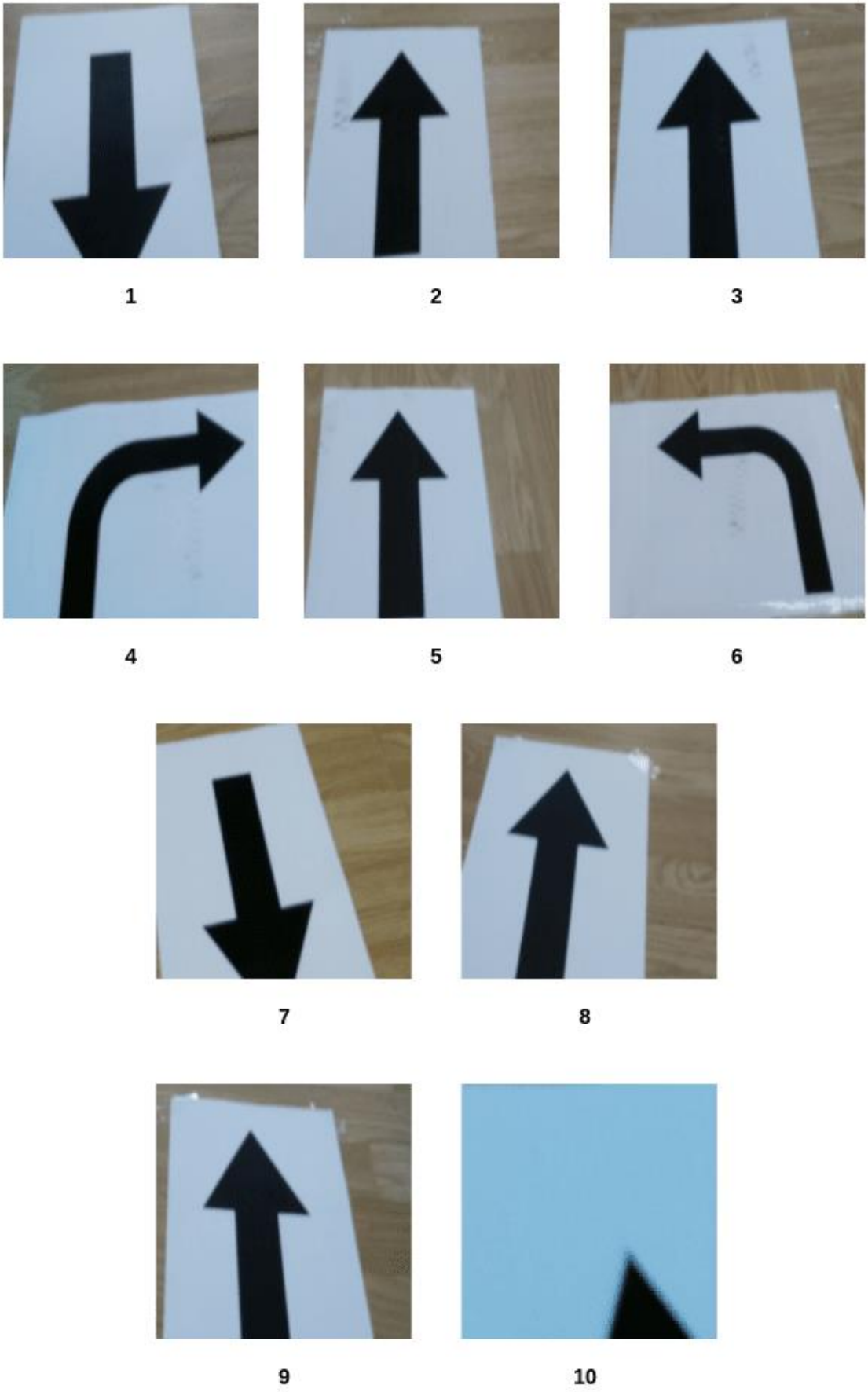


Figure 52 – Images after pre-processing

While the robot navigates, its coordinates are noted after each move. In the following figure (Figure 53), the robot's trajectory is depicted. Each dot represents every set of coordinates noted.

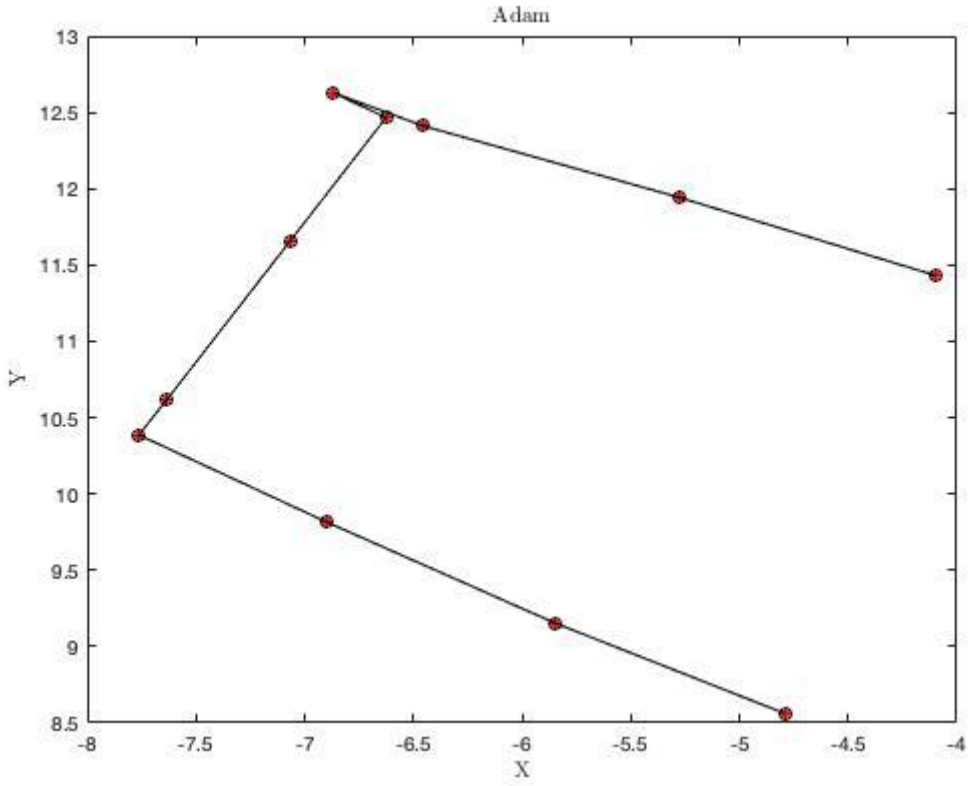


Figure 53 – Robot's trajectory in space, classification model trained with Adam

6.2.3.2 VGG16 with Categorical Crossentropy and Adagrad

A second experiment was conducted, using a model trained for 25 epochs, with batch size 32, categorical crossentropy as a loss function and Adagrad as optimizer. This model is used by the server for image classification.

The robot had to follow the route on Figure 47 and at the end, it followed the path correctly, just like in the previous experiment. It classified all 10 pictures correctly. In Figure 54 are shown all images that were received by the server. Apparently, all markings were captured in an efficient way by the raspberry camera.

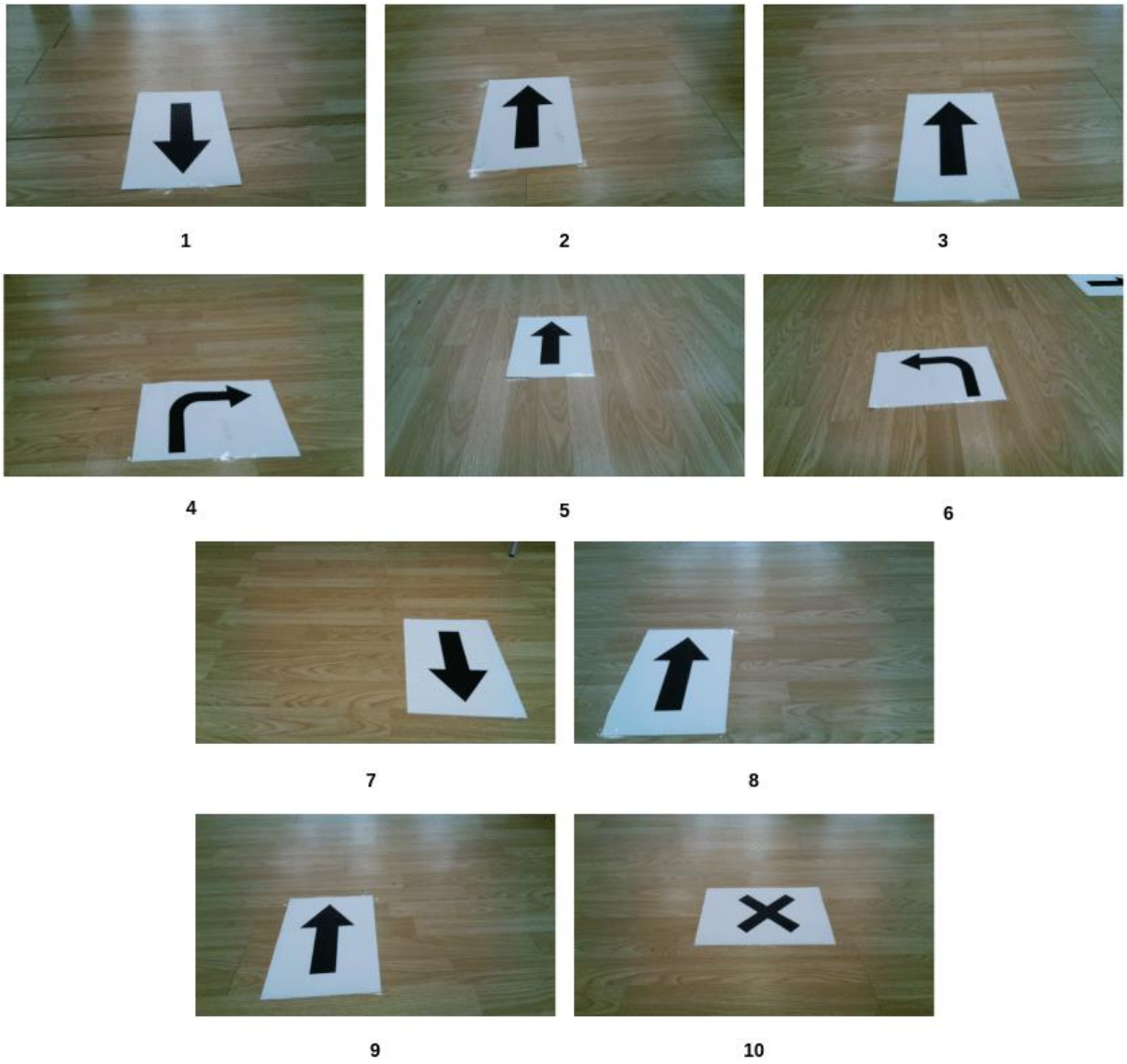


Figure 54 – Received images from server

In Figure 55, the images that resulted from pre-processing and are destined for classification, are shown. Since all images were captured correctly from the robot, pre-processing results were very efficient, as expected. Therefore, each image from Figure 55was successfully classified by our model and the navigation terminated successfully.

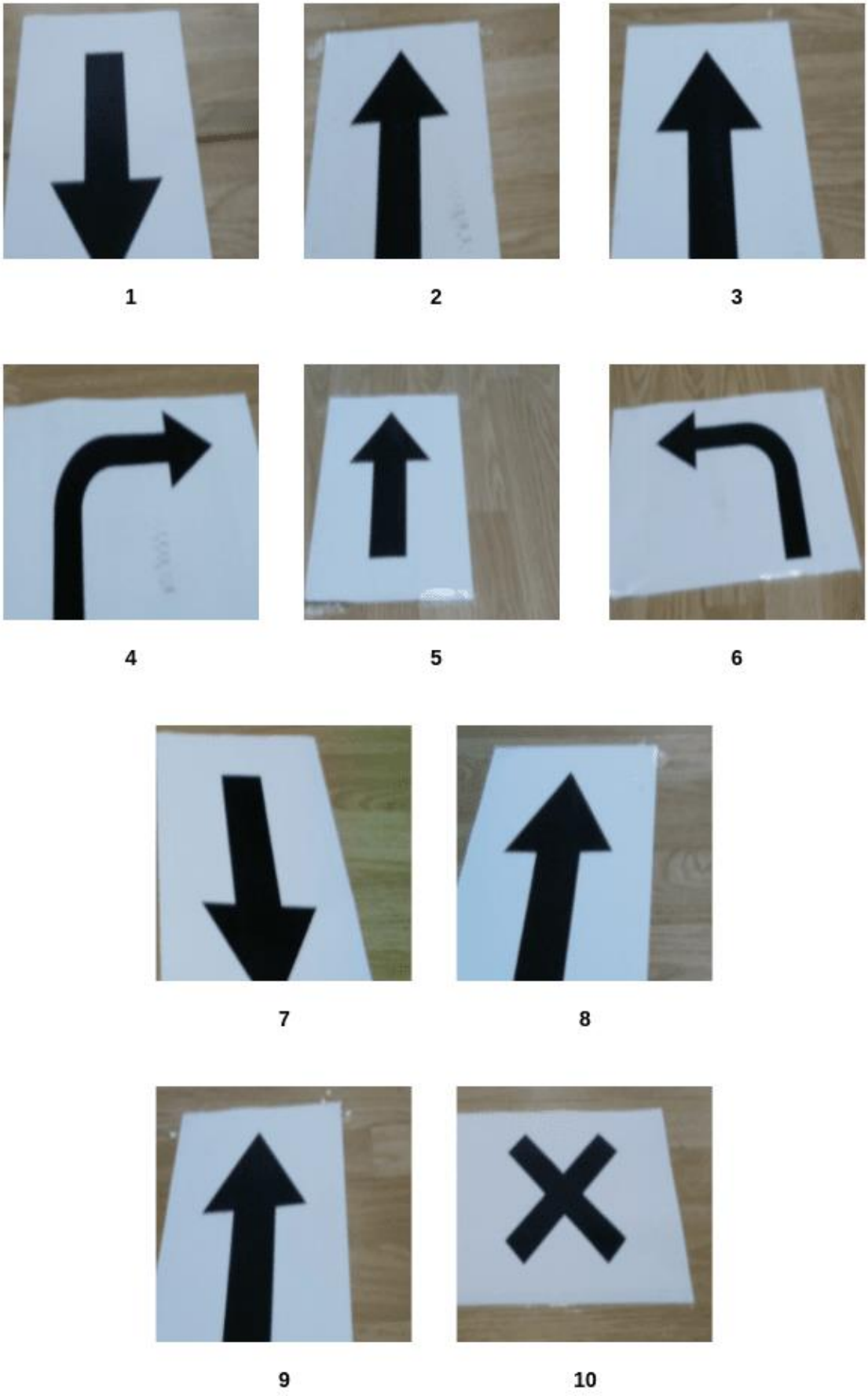


Figure 55 – Images after pre-processing

During this experiment, too, the coordinates of the robot were noted after each move. The following figure (Figure 56) depicts its trajectory, from the beginning until the end. Each dot represents each set of coordinates noted.

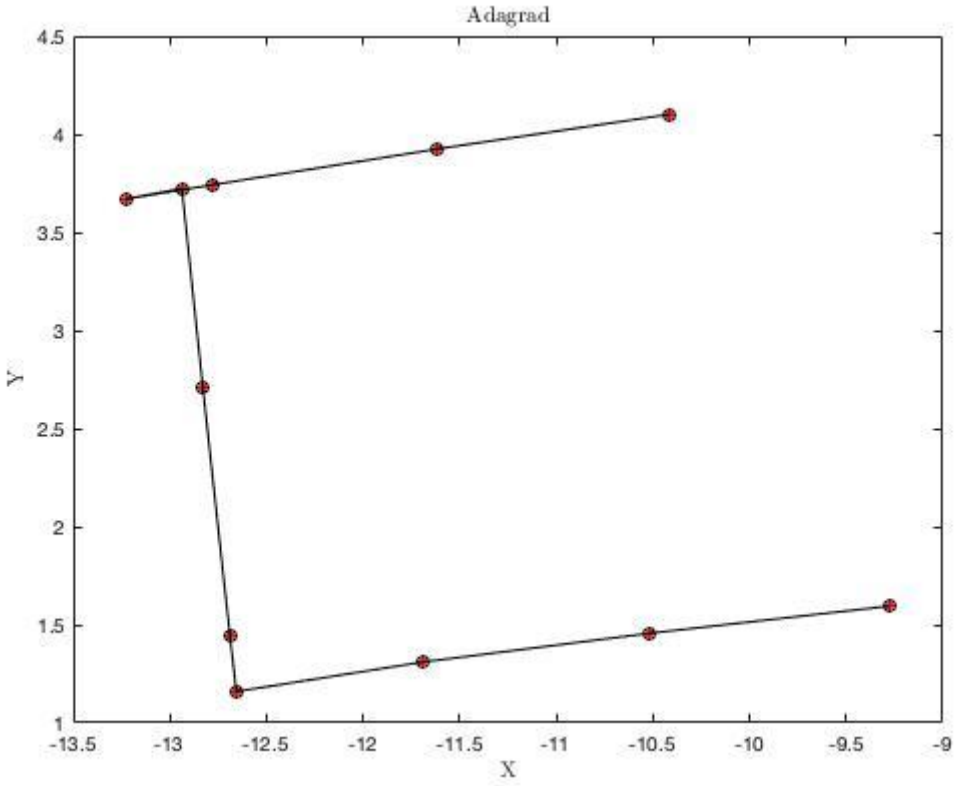


Figure 56 - Robot's trajectory in space, classification model trained with Adagrad

7. CONCLUSION

The enormous growth and use of autonomous vehicles in recent years appears to have overwhelmed the technology community. Due to the fact they can be deployed quickly and efficiently, unmanned vehicles are being employed more and more in a variety of applications. The complexity and dynamic nature of the environment around unmanned vehicles, as well as their interaction with other people and objects, as well as any sudden changes in the environment, make autonomous navigation of these vehicles a major challenge for a variety of reasons.

For instance, a vehicle should drive itself along the road by obeying the various arrow signs that indicate a set of restrictions. There are multiple ways, provided by machine learning, to achieve this, one of them being the use of Neural Networks. More specifically, with the use of a Convolutional Neural Network, a deep learning algorithm that is ideal when a dataset is comprised of images. An analogy can be drawn between the architecture of a CNN and the connectivity network of neurons in the human brain; individual neurons only respond to stimuli in a specific area of the visual field known as the receptive field. The entire visual field is covered by a series of such fields that overlap.

This thesis has proposed a methodology that included the use of a image processing technique and a popular CNN, named VGG, in order to detect arrow markings in images captured by a raspberry-pi camera and later classify these arrows accordingly and follow the path that was planned. A series of experiments were conducted, starting with a benchmarking of VGG-16 and VGG-19 models with different hyperparameters. This showed the different behaviors of models trained with different combinations of hyperparameters, which lead to the choice of certain hyperparameters to be ideal for the training of 2 models, used in the second phase of experiments. As for the second phase, one navigation took place using each model, concluding that not only the hyperparameters of a model are important in a successful navigation, but also precise moves of the vehicle and a pre-processing technique, whose parameters are adjusted to the environment’s condition’s (i.e floor color, lighting of space).

ABBREVIATIONS - ACRONYMS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
ConvNet	Convolutional Neural Network
GPS	Global Positioning System
LiDAR	Light Detection And Ranging
MLP	Multi-Layered Perceptron
MSE	Mean Squared Error
ReLU	Rectified Linear Unit
RMSprop	Root Mean Square Propagation
RNN	Recurrent Neural Network
ROS	Robot Operating System
SGD	Stochastic Gradient Descent
SLAM	Simultaneous Localization And Mapping
SNN	Simulated Neural Network
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
USV	Unmanned Surface Vehicle
VGG	Visual Geometry Group

REFERENCES

[1] Caska, Serkan & Gayretli, Ahmet. (2014). A survey of UAV/UGV collaborative systems. 453-463.

[2] H. Shakhatreh et al., "Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges," in IEEE Access, vol. 7, pp. 48572-48634, 2019, doi: 10.1109/ACCESS.2019.2909530.

[3] Hardesty, Larry (14 April 2017). "Explained: Neural networks". MIT News Office. Retrieved 2 June 2022.

[4] O'Shea, Keiron and Nash, Ryan. An Introduction to Convolutional Neural Networks, arXiv, 2015, doi: 10.48550/ARXIV.1511.08458

[5] Jordi TORRES.AI, Convolutional Neural Networks for Beginners using Keras & TensorFlow 2, Retrieved 2 July 2022

[6] Afzal, Muhammad Zeshan & Kölsch, Andreas & Ahmed, Sheraz & Liwicki, Marcus. (2017). Cutting the Error by Half: Investigation of Very Deep CNN and Advanced Training Strategies for Document Image Classification. 10.1109/ICDAR.2017.149.

[7] Tawiah TA-Q. A review of algorithms and techniques for image-based recognition and inference in mobile robotic systems. International Journal of Advanced Robotic Systems. November 2020. doi:10.1177/1729881420972278

[8] Quiñonez, Yadira & Ramirez, M & Lizarraga, Carmen & Tostado, I & Bekios-Calfa, Juan. (2015). Autonomous Robot Navigation Based on Pattern Recognition Techniques and Artificial Neural Networks. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 9108. 320-329. 10.1007/978-3-319-18833-1_34.

[9] Foroughi, F.; Chen, Z.; Wang, J. A CNN-Based System for Mobile Robot Navigation in Indoor Environments via Visual Localization with a Small Dataset. World Electr. Veh. J. 2021, 12, 134. https://doi.org/10.3390/wevj12030134

[10] G. Maier, S. Pangerl and A. Schindler, "Real-time detection and classification of arrow markings using curve-based prototype fitting," 2011 IEEE Intelligent Vehicles Symposium (IV), 2011, pp. 442-447, doi: 10.1109/IVS.2011.5940451.

[11] Park, Jongan & Rasheed, Waqas & Beak, Junguk. (2008). Robot Navigation Using Camera by Identifying Arrow Signs. 382-386. 10.1109/GPC.WORKSHOPS.2008.41.

[12] Shojaeipour, Shahed & Haris, Sallehuddin & Khairir, Muhammad. (2009). Vision-Based Mobile Robot Navigation Using Image Processing and Cell Decomposition. 5857. 90-96. 10.1007/978-3-642-05036-7_10.

[13] Shih, Ching-Long & Ku, Yu-Te. (2016). Image-Based Mobile Robot Guidance System by Using Artificial Ceiling Landmarks. Journal of Computer and Communications. 04. 1-14. 10.4236/jcc.2016.411001.

[14] Tawiah TA-Q. A review of algorithms and techniques for image-based recognition and inference in mobile robotic systems. International Journal of Advanced Robotic Systems. November 2020. doi:10.1177/1729881420972278