

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

Neural-based navigation system for 4x4 Jaguar Platform

Diploma thesis

**Submitted in partial fulfillment of the requirements
for the degree of *Engineer*
in the domain of *Electronics, Telecommunications and Information
Technologies*
study program *Applied Electronics***

Thesis Advisor(s)

**Prof. Dr. Ing. Corneliu BURILEANU,
As. Univ. Drd. Ing. Ana Neacșu**

Student

Alexandru-Mădălin Costea

Year 2020

Statement of Academic Honesty

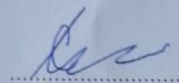
I hereby declare that the thesis *Neural-based navigation system for 4x4 Jaguar Platform*, submitted to the Faculty of Electronics, Telecommunications and Information Technologies, University POLITEHNICA of Bucharest, in partial fulfillment of the requirements for the degree of *Engineer* in the domain Electronics and Telecommunications, study program *Applied Electronics* is written by myself and was never before submitted to any faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand that plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, June 2020.

Student: Alexandru-Mădălin Costea



.....

University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology
Department **EAIT**

Anexa 1**DIPLOMA THESIS**

of student **COSTEA F. Alexandru-Mădălin , 441F-ELA.**

1. Thesis title: Neural-based Navigation system for 4x4 Jaguar Platform

2. The student's original contribution will consist of (not including the documentation part) and design specifications:

The purpose is to design and implement a collision avoidance system, using image based obstacle detection trained with deep neural networks. The system will be validated on a 4x4 platform that will automatically move in a controlled environment.

To fulfill this purpose the following objectives will be achieved:

- creating a data set consisting of images containing 3 classes of objects (red – obstacles, blue – pass through objects and the robot itself)
- training different Convolutional Neural Networks architectures for the obstacle detection
- implementing a path detection algorithm, to find the shortest way from the starting point to the destination chosen by the user, given the obstacles.
- programming the 4x4 platform to move according to the algorithm's output

The system will work as follows:

A Kinect sensor will provide environmental images, that represent the input of the CNN. The network will be able to identify the coordinates of the three types of objects aforementioned. The path finding algorithm will compute the shortest path to a destination given by the user, considering the objects detected. This path will be transmitted to the robot that will move accordingly.

3. Pre-existent materials and resources used for the project's development:

Hardware: 4x4 Jaguar Platform (has it's own router), Kinect Xbox One Software: PyKinect Library, Kinect SDK, OpenCV Library, TensorFlow, IBM Cloud Annotations, Navigation System Code Commands for 4x4 Jaguar Platform

4. The project is based on knowledge mainly from the following 3-4 courses:

Medical Imaging, Decision and estimation in information processing, Object Oriented Programming and

5. The Intellectual Property upon the project belongs to: U.P.B.

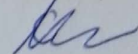
6. Thesis registration date: 2019-11-28 14:29:44

Thesis advisor(s),
Prof. dr. ing. Corneliu BURILEANU

Ana NEACSU

Departament director,
Prof. dr. ing Sever PAȘCA

Student,



Dean,
Prof. dr. ing. Mihnea UDREA

Validation code: **f2bd38c5a0**

Table of Contents

List of figures	iii
List of tables	iv
List of abbreviations	v
1. Thesis Technologies	3
1.1. Hardware	3
1.1.1. Jaguar 4x4 Mobile Platform	3
1.1.2. Kinect Xbox One	3
1.2. Software	4
1.2.1. PyKinect2	4
1.2.2. OpenCV 4	4
1.2.3. Keras	5
1.2.4. XML	6
1.2.5. IBM Annotation Tool	6
2. Neural Networks	7
2.1. Concepts of Neural Networks	7
2.2. Modified Tiny YOLO	15
2.2.1. How does YOLO work?	15
2.2.2. Modified Tiny YOLO Concepts	19
2.2.3. Results of Modified Tiny YOLO	25
2.2.4. YOLO Comparison	26
2.3. Patch Model	28
2.3.1. Dataset for Patch Model	28
2.3.2. The Keras model	29
2.3.3. Results of the Patch Model	31
2.4. Advantages and disadvantages	32
3. Pathfinding System	35
3.1. Pathfinding algorithms	35
3.2. Helpful functions	37
3.3. Robot commands	39

4. Conclusions	41
References	45

List of figures

1.1. Deep Learning Framework Power Scores 2018 [1]	6
2.1. Fully Connected Layers [2]	8
2.2. Activation Functions	10
2.3. Activation Functions	11
2.4. Convolutional Layer [3]	12
2.5. Types of data fitting [4]	15
2.6. YOLO's Simple Output	16
2.7. Image Grid	17
2.8. Intersection over Union [5]	17
2.9. Depthwise Separable Convolution	20
2.10. Filter Correlation along Depth Axis [6]	21
2.11. Filter Variance [6]	22
2.12. BSConv comparison with Standard Convolution Layer [6]	23
2.13. Unconstrained BSConv	23
2.14. Subspace BSConv	24
2.15. Modified Tiny YOLO Results	26
2.16. Tiny YOLO Results	27
2.17. Patch Dataset	29
2.18. Patch Model Results	31
2.19. Patch Model Results	31
2.20. Shipping yard [7]	34
2.21. Market images	34
3.1. Types of Distances	36
3.2. Direction Detector	38
3.3. Path Creation	40

List of tables

2.1.	Main features of used NNs	27
2.2.	Layers of used NNs	27
2.3.	Accuracy of used NNs	27
2.4.	Parameters of used NNs	28

List of abbreviations

DNN = Deep Neural Network
CNN = Convolutional Neural Network
YOLO = You Only Look Once
SDK = Software Development Kit
pip = pip installs packages
CV = Computer Vision
API = Application Program Interface
GPU = Graphics Processing Unit
SSE = Sum Squared Error
SGD = Stochastic Gradient Descent
ReLU = Rectified Linear Unit
LeakyReLU = Leaky Rectified Linear Unit
R-CNN = Region with Convolutional Neural Network
SSD = Single Shot multibox Detector
ResNet = Residual Network
VGG = Visual Geometry Group
IoU = Intersection over Union
NMS = Non-Max Suppression
BN = Batch Normalization
DSC = Depthwise Separable Convolution
BSConv = Blueprint Separable Convolution
PCA = Principal Component Analysis
PC1 = Principal Component 1
IoT = Internet of Things
RGB = Red Green Blue
HD = High Definition

Introduction

Purpose

This thesis proposes to design and implement a collision avoidance system, using image-based obstacle detection trained with **Deep Neural Networks** (DNNs). The system will be validated on a **4x4** platform that will automatically move in a controlled environment.

The system works as follows: A **Kinect** sensor provides environmental images, that represent the input of learning system, namely a **Convolution Neural Network** (CNN). The network is able to identify the coordinates of the three types of different objects that will be detailed later. The path finding algorithm computes the shortest path to a destination given by the user, considering the objects detected. This path is then transmitted to the robot that will move accordingly.

Objectives

The main objectives of the thesis are the following:

- to create a data set consisting of images containing 3 classes of objects (red – obstacles, blue – pass through objects and the robot itself);
- to train different **Convolutional Neural Networks** architectures for the obstacle detection, in the quest of finding the best trade-off between efficiency and accuracy;
- to implement a path detection algorithm, that finds the shortest way from the starting point to the destination chosen by the user, given the obstacles;
- to control the **4x4** platform to move according to the algorithm's output.

Motivation

In the automotive industry there are a lot of companies that started researching self-driving cars. These cars are equipped with a variety of sensors to transmit data to the control system. While a fully autonomous vehicle is desired, at the moment, such a system is hard to implement and expensive. Other methods have to be developed to make use of an autonomous vehicle or robot. These methods would need to reduce the number of sensors, the amount of data and compute processing power needed. Of course, reducing the aforementioned items would require compromises in other areas of the system.

The *Neural-based navigation system for 4x4 Jaguar Platform* achieves a basic form for an easier to implement and less expensive system. The only sensor used for pathfinding is the **Kinect Camera**, that gives a top-down view of the robot. Given the fact that there is only one sensor, the amount of data processed is kept at a minimum (just the images taken by the camera). The system makes use of a CNN and a pathfinding algorithm **A*** connected to the results of the CNN. A modified version of **Tiny YOLO** and a new model **Patch Model** is used for the CNN as it provides a good accuracy as well as low requirements in terms of processing. A comparison between **Tiny YOLO**, **Modified Tiny YOLO** and the **Patch Model** will be done later in the paper. The same motivation can be presented for the **A*** algorithm.

This type of system presents limitations. As it is, the camera is fixed, thus, it can provide information for a single area. A fully autonomous robot would be able to perform in all environments. A solution for this, would be a drone companion but, this is beyond the scope of this paper, and it comes with its own problems. A mobile camera, or a larger field of view can be simple solutions but, in the end, the mapping will still be done for a local area. Another limitation of the top-down camera is that the system will not be able to see the robot if the robot goes under a foreign object (i.e. a bridge). Reviewing the lesser amount of sensors and data, means the system will know less about the surroundings of our robot. The more the system knows about its surroundings the more functions the robot can have (i.e. picking up an object).

As it can be seen, achieving a simpler system comes with its compromises. Although it has its limitations, some advantages can be seen, other than making it simple. A top-down camera allows the system to see a lot more of the environment in which the robot performs. While an autonomous system has to calculate the route in real time when the robot is moving, a top-down camera can provide a clear map with the fastest route to the destination.

A variety of sensors can be attached to the robot, keeping in mind cost efficiency, to provide more functionality to it. An example would be an infrared sensor to give precise distance information for the robot, or a frontal camera for performing different actions if the robot has a robotic arm, as is the case for the **Jaguar 4x4 Platform**. Various other software programs can be implemented for the type of image provided, such as, an algorithm to calculate distance between corners of different objects.

A simpler system could benefit more users, provided they can make use of it with its current limitations.

Chapter 1

Thesis Technologies

1.1 Hardware

The hardware used for this project is composed of: Jaguar 4x4 Mobile Platform [8], Kinect Xbox One [9] and a laptop. The first two will be briefly introduced.

1.1.1 Jaguar 4x4 Mobile Platform

The *Jaguar 4x4 Mobile Platform* is a robot that has many functionalities, thanks to the many sensors that it's equipped with and the robotic arm attached to it. A few of these sensors will be useful for moving the robot according to the data created by the pathfinding algorithm. These sensors provide yaw, acceleration and gyro information.

Other than sensors, the robot has wireless control, since it comes with its own *Wireless Router WRT802G 802.11N*. Information transmission will be done through this router using TCP/IP model and the `Socket Library`.

The robot itself is commanded by the *PMS5006 Controller*, so specific commands need to be sent and picked up by the router. A `C#` program has been written by Tudoroiu Mihai-Cristian for his undergraduate thesis [10] in which he models the movement of the robot in order to be able to receive the commands given by an external client and move at a safe speed, since the motors of the 4x4 platform are powerful. The presented thesis makes use of this program such that it can send the specific commands required by the PMS5006 controller.

1.1.2 Kinect Xbox One

The Kinect device [9] also has a variety of functionalities thanks to its Full HD camera, depth sensors, motion tracking and voice commands. For the project only the camera will be used.

The camera is attached to a metallic rail bolted into the ceiling of the laboratory. With the help of this rail the camera can be moved up or down. Thanks to the way the Kinect is attached to the rail it has two more degrees of movement allowing the user to position the camera correctly.

Images that are captured will have a resolution of 1920x1080 pixels and a field of view of 135 degrees. The image is processed to fit into the software system created afterwards.

1.2 Software

Most of the code is written in the **Python** programming language, thus, the architectures and libraries used, are for this language. The version used for this project is **Python 3.7+**.

In the the following subsections, the most important libraries will be briefly presented, with their description and usefulness for the project.

1.2.1 PyKinect2

This **Python** library [11] allows the creation of applications to control the *Xbox One Kinect*. It does not allow for the full control of the device and presents the following functions: color, depth, body and body index frames. These functions satisfy the need of the current project.

In order to use the library the computer must have the following prerequisites: **Python 2.7.x** or **3.4** and higher, **NumPy**, **comtypes**, Kinect for Windows SDK 2.0, Kinect sensor and adapter.

To install **Python** packages **pip** is used, acronym for "**pip** Installs Packages". This is a practical package-management system. **NumPy** is, according to its description, a package for scientific computing that contains: N-dimensional array object, sophisticated (broadcasting) functions, tools for integrating C/C++ and Fortran code, useful linear algebra, Fourier transform, and random number capabilities. This package is very useful for Machine Learning. On the other hand, **comtypes** is a lightweight **COM** package. It allows to define, call, implement custom and dispatch-based **COM** interfaces in pure **Python**. Both of these packages were installed using **pip**.

Kinect for Windows SDK 2.0 is used in developing applications for Kinect devices on Windows 10. The manufacturing of the Kinect for Windows has been discontinued, as presented on the official Microsoft website but, the SDK still proves itself to be useful in this project, while also being a must for the **PyKinect2** library. This kit enables developers to create applications that support gestures and voice recognition, using the Kinect sensor technology. The Kinect sensor and adapter are the hardware prerequisites. **PyKinect2** being itself a **Python** package, has been installed through **pip**, with the following command in command prompt:

```
pip install pykinect2
```

All these packages helped in creating the **Python** application that transmits real-time images to the Neural Network.

1.2.2 OpenCV 4

Open Source Computer Vision Library is an open source computer vision and machine learning software library. It contains several thousands algorithms that range from face recognition and object detection to extracting 3D models and removing red-eyes from an image. It was developed in order to accelerate the rate of innovation and implementation of machine perception in commercial products [12].

In this project it is used for image processing, as it offers multiple functions for image manipulation such as: reading images and/or video, resizing, image thresholding and displaying windows. The main functions used from the **OpenCV** library are:

- **Core functionality (core)** – a compact module defining basic data structures, including the dense multi-dimensional array **Mat** and basic functions used by all other modules.
- **Image Processing (imgproc)** – an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), color space conversion, histograms, and so on.

- **Video Analysis (video)** - a video analysis module that includes motion estimation, background subtraction, and object tracking algorithms.

Full functions list is detailed in [12].

The OpenCV package was also installed using `pip`:

```
pip install opencv-Python
pip install opencv-contrib-Python
```

The package is originally written in C++ but, it has been ported to other programming languages. The two packages, `opencv-Python` and `opencv-contrib-Python` represent the main modules and the contributed modules, respectively.

1.2.3 Keras

In order to create a neural network architecture, the **Keras** API has been used. It does not process low-level operations, such as tensor operations, it only provides high-level models and functions to offer a quick and stable method for testing and creating deep neural networks. In order to process tensor operations a back-end engine, a tensor manipulation library is required. **Keras** is tied to this back-end framework. The officially supported frameworks are: **Theano**, **CNTK** and **TensorFlow**. The chosen back-end for this project is **TensorFlow**.

A short description of Keras backends

Firstly, it needs to be mentioned that - Yoshua Bengio, announced that major development of the **Theano** framework, would be ceased, 15th of November 2017 being the release of **Theano** 1.0.0, the last major release. Currently, **Theano** is at version 1.0.4, with the PyMC team continuing its maintenance.

CNTK [13] is presented by Microsoft and it stands for Cognitive Toolkit. It is an open-source library for deep learning. It describes neural networks as a series of computational steps via a directed graph. **CNTK** is a commercial grade architecture and it comes with many of the functions that popular back-ends provide, but it does not provide the same integration of **Keras** that **TensorFlow** does.

TensorFlow is developed by Google. The framework has many ideas based on the **Theano** framework, many of these have been improved. It uses dataflow graphs to represent computation, shared state, and the operations that mutate that state. It has been released as an open-source project, and it has become widely used for machine learning research. At the moment **TensorFlow** 2.0 has been released, which provides the best support for the **Keras** API. The **Keras** library is implemented directly into **TensorFlow** and can be called as a **Python** module for use in neural networks. In [14] can be found the **TensorFlow** dataflow model and the demonstration of the compelling performance that **TensorFlow** achieves for several real-world applications.

Keras can use other types of backends as well, or can use multiple backends, one example could be **MXNet**. Although the use of different backends can have an impact on performance, the difference between back-ends for **Keras** while building simple CNNs (e.g. the project requires a simple CNN to perform one specialized task, obstacle detection, with three classes) is negligible. The reason behind this is because most of the GPUs will use **cuDNN** to process the heavy work.

Why Keras?

The **Keras** framework guides itself after the following principles: ease of use, fast experimentation, maintained flexibility. Thus, for creating a 'normal' CNN, a flexible and easy to use framework is useful for experimenting with different models and hyper-parameters. On the other hand, **Keras** is built inherently in **Python**, providing excellent support for the current thesis as it is created using the **Python** programming language.

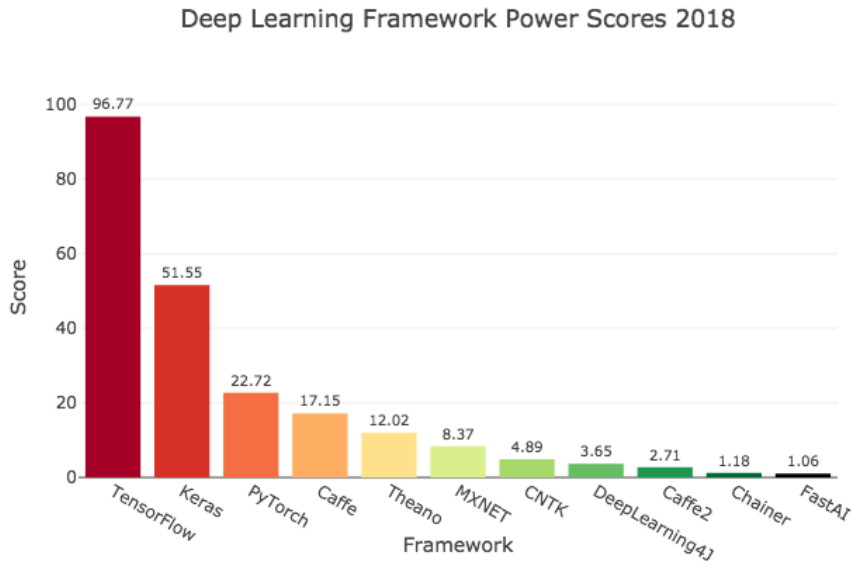


Figure 1.1: Deep Learning Framework Power Scores 2018 [1]

Other than its principles and use as technology, **Keras** has a great adoption in the industry and research community which provides also great community support and multiple examples for multiple uses.

In the figure 1.1 it can be observed that **TensorFlow** is also one of the most adopted architectures, which means that it also takes advantage of great community support.

Compared to **TensorFlow**, apart for using it as a back-end, **Keras** is not as verbose and it is more streamlined. Simple functions take care of a lot of the steps that were required in **TensorFlow**.

1.2.4 XML

XML is a data storage and transport system. It was used to store annotation information for the used neural networks. It works on a parent-child system, without any visual information, just text. This system works great for the presented project as different data can be stored in a single file, each with its own category, such as class name or bounding box coordinates.

Two different libraries were used to work with **XML** files: **ElementTree** and **minidom**. Both of them were used since the **ElementTree** library provides a more efficient way of creating **XML** files whilst **minidom** provides an easy way of reading through the storage file, through tag searching (a tag can be a parent of child and provides information on all data under it).

XML files are also used in the popular **VOC Datasets** on which the structure of the annotations is based on: folder, filename, size, class and bounding box as main nodes.

1.2.5 IBM Annotation Tool

The **IBM Annotation Tool** is a very useful product, as it provides an online tool for creating annotations for a dataset of images – e.g., upload images, choose between image classification or object detection, click and drag to make boxes over different objects, with a before chosen label. Not only the tool allows the creation of bounding boxes for the dataset but it can also be exported in many different formats such as the one used for **YOLO** based on **txt** files or the one used in this project in the **VOC XML** format.

Chapter 2

Neural Networks

2.1 Concepts of Neural Networks

The software component of this thesis is based on the concept of *machine learning*. It employs some powerful and complex learning techniques called *neural networks*.

Neural networks are mathematical structures that can map changing input data to specific outputs. Neural networks mimic the function of the brain and represent one branch of *Artificial Intelligence*. Lately, these learning methods have become ubiquitous tools, successfully in an ever-increasing number of domains – e.g. medicine, autonomous driving, Human Computer Interaction (HCI), computer vision etc.. Neural networks can be used for many tasks, ranging from regression to complex multi-modal classification.

A neural network is structured on *layers*, each composed of a number of learning units, called *neurons*. The more layers, the deeper the network – hence, the name of *Deep Neural Networks*. Depending on their position in the network, 3 types of layers can be distinguished: input, hidden and output. The input layer is used to ensure the correct dimension of input data and has no active role in the learning process; the hidden layers are where the neural network learns, by updating the parameters after each iteration. The output layer assures the right dimension of the output – e.g. the number of classes in a classification task.

To train a neural network you need to have a dataset containing multiple examples and the associated desired output, called *ground truth*. The learning mechanism is employing an iterative optimization algorithm that minimizes a *cost function*, that measures how close is the output of the network from the ground truth.

Fully connected layers

The most general types of layers are fully connected layers. Each neuron in such a layer $l \in \{1 \dots m\}$, where m is the total number of layers, excepting the input one, is connected to all neurons in the previous and next layer as seen in figure 2.1. The outputs of such a layer are given by the following formulas:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \quad (2.1)$$

$$A^{[l]} = \sigma(Z^{[l]},) \quad (2.2)$$

where $W^{[l]}$ are weights matrix of layer l , $A^{[l-1]}/A^{[l]}$ is the input/output matrix and $b^{[l]}$ signifies the bias parameter. σ denotes a non-linear component of the network called the *activation function* (e.g. sigmoid function)

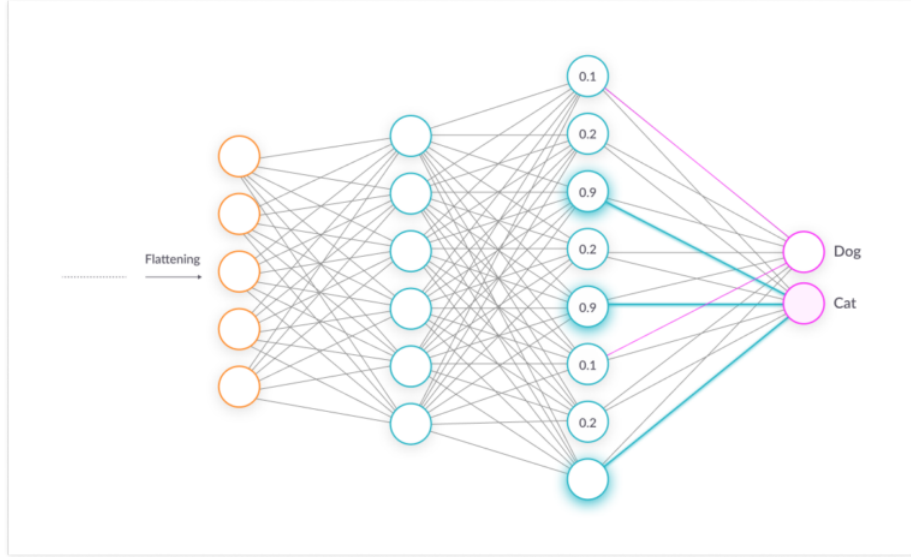


Figure 2.1: Fully Connected Layers [2]

Loss Functions

In order for the network to learn, the weights and biases need to change according to a cost function. The task of the optimizer is to minimize this loss by changing the parameters accordingly. There are many types of loss functions, two of which are explored in this project, the *Sum Squared Error* (SSE) and the *Sparse Categorical Cross Entropy*. These losses represent the error found between the estimated value \hat{y} and the true value y over m number of examples.

SSE is used in the loss of the YOLO architecture and has the following formula:

$$\mathcal{L} = \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2 \quad (2.3)$$

SSE is sensitive compared to other losses and can be impacted more by outliers in the data. It can generally be seen in linear regression models.

The sparse categorical cross entropy has the following formula:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})) \quad (2.4)$$

It can handle multiple classes and the labels provided to it are of integer type as opposed to simple *categorical cross entropy* which takes one-hot encoding types of labels. The sparse version is considered quicker since it only needs an integer as opposed to a whole vector.

Optimization Algorithms

Through backpropagation, the weights and biases of the network (its main parameters) are changed with each iteration, in an effort to minimize the loss function. The formula through which this happens varies by choosing different optimization algorithms such as: **Adam**, **RMSProp**, **Gradient Descent**, **SGD** (i.e. Stochastic Gradient Descent) and other variations. The algorithm used in this paper is the **Adam optimization algorithm** [15] as it works best in most neural networks. This algorithm makes use of **Gradient Descent with Momentum** and **RMSProp**.

The backpropagation of the network is an algorithm that for each iteration computes the gradients $\frac{\partial \mathcal{L}}{\partial w}$, $\frac{\partial \mathcal{L}}{\partial b}$ with respect to all weights and biases. In order to make use of backpropagation,

the aforementioned optimization algorithms are used and impact the weights and biases through the following general formulas:

$$W := W - \alpha \nabla \mathcal{L}(W) \quad (2.5)$$

$$b := b - \alpha \nabla \mathcal{L}(b) \quad (2.6)$$

Where $\nabla \mathcal{L}(W)$ and $\nabla \mathcal{L}(b)$ represent the gradient with respect to weights and biases, respectively. Different algorithms present different variants of these formulas, like the concepts used for the Adam algorithm.

First, in order to make sens of these concepts, the exponentially weighted moving averages must be explained.

The formula for this concept is the following:

$$v_t = \beta_{t-1} + (1 - \beta)\theta_t \quad (2.7)$$

The v_t term represents the the moving average at iteration t . The exponential moving average makes use of the previous average at iteration $t - 1$ with modifier β representing how smooth or how precise the average can be, taking values between $(0, 1)$ (i.e. a modifier of 0.9 can represent an average over 10 iterations as given by equation $\frac{1}{1 - \beta}$). The term θ_t represents the current value at iteration t . This formula will not give the best average but it is used in deep learning as it is very memory efficient and as it is the building block for **Gradient Descent with Momentum** and **RMSProp**, algorithms that speed up the search for the minimum of the loss.

The gradient of the loss function with respect to W and b will be denoted as follows: $\nabla \mathcal{L}(W) = dW$ and $\nabla \mathcal{L}(b) = db$.

Putting Gradient Descent with Momentum

$$V_{dW} := \beta_1 V_{dW} + (1 - \beta_1) dW \quad (2.8)$$

$$V_d := \beta_1 V_{db} + (1 - \beta_1) db \quad (2.9)$$

together with RMSProp,

$$S_{dW} := \beta_2 S_{dW} + (1 - \beta_2) dW^2 \quad (2.10)$$

$$S_{db} := \beta_2 S_{db} + (1 - \beta_2) db^2 \quad (2.11)$$

will give the final parameter equations for W and b ,

$$W := W - \alpha \frac{V_{dW}}{\sqrt{S_{dW}} + \epsilon} \quad (2.12)$$

$$b := b - \alpha \frac{V_{db}}{\sqrt{S_{db}} + \epsilon} \quad (2.13)$$

making up the **Adam** optimization algorithm.

The symbol $:=$ represents that the value in question is updated with the new value represented by the equation.

It has to be noted that α is the learning rate and $V_{dW/b}$ and $S_{dW/b}$ are initialized with zero.

Activation functions

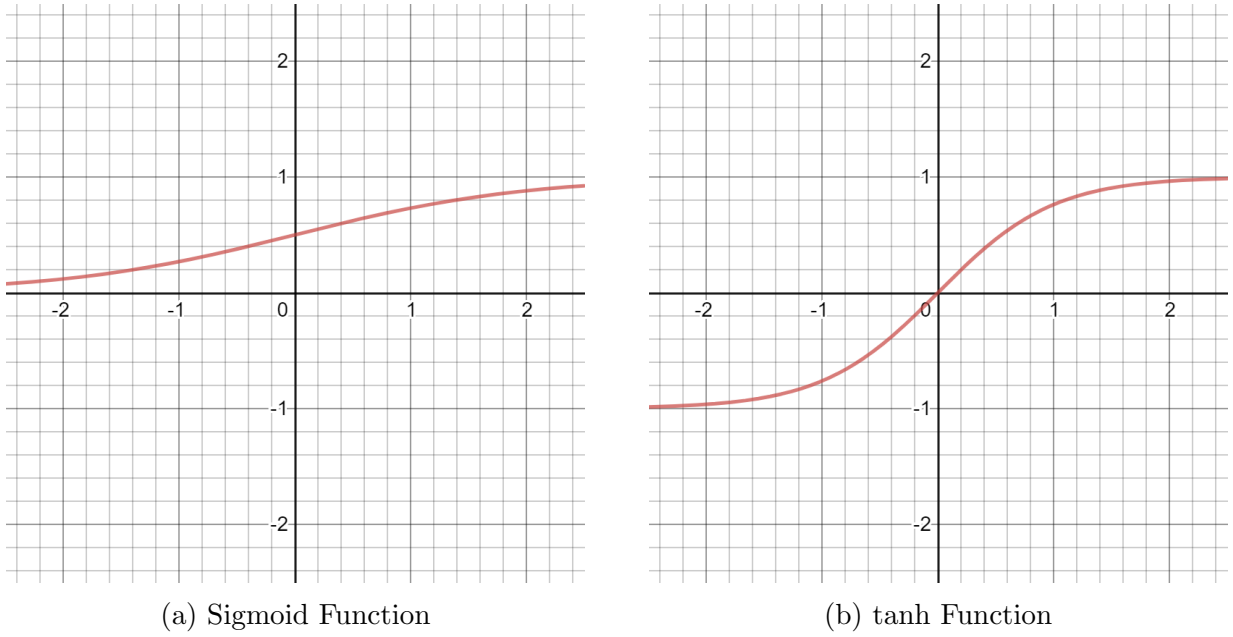


Figure 2.2: Activation Functions

Another concept that has to be explained, is the activation function. An activation function can be correlated with the firing of a neuron. There are many types of activation functions, the ones that are used in this project are: ReLU (i.e. Rectified Linear Unit), LeakyReLU and Mish. As mentioned before, the introduce the non-linear component of the neuron.

To better show what activation functions do, the following functions will be presented briefly: sigmoid, tanh, ReLU and Mish. The **sigmoid** function is displayed graphically in figure 2.2a with the following formula:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.14)$$

As it can be seen, the function tends to 0 for negative numbers and to 1 for positive number. It is centered in 1/2, unlike the next function that has zero-mean.

The **tanh** function has its graph representation in figure 2.2b with the following formula:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.15)$$

It can be seen the tanh function provides a better mean with a center in zero but, because it's a function that goes between -1 and 1 and because it saturates really quickly, at these limits, the slope provided by it is very small, thus, gradient descent will perform very slow.

The **ReLU** function has its graph representation in figure 2.3a with the following formula:

$$f(x) = \max(0, x) \quad (2.16)$$

The ReLU function provides a much better slope and it is one of the most used activation functions, with many alternative variants, such as the LeakyReLU or Mish.

Mish [16] is a smooth non-monotonic activation function. It is similar to Swish [17] and an improvement over the general ReLU activation function. Its graphical representation can be found in figure 2.3b and it has the following formula:

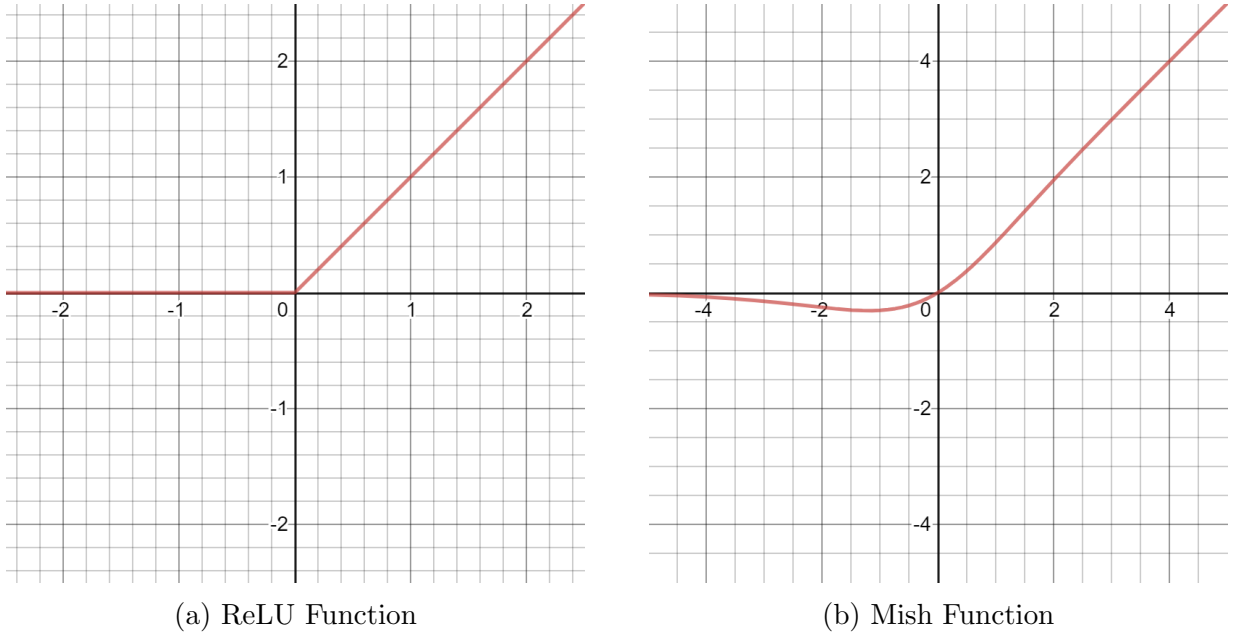


Figure 2.3: Activation Functions

$$f(x) = x \tanh(\zeta(x)) \quad (2.17)$$

Where $\zeta(x)$ is the softplus activation function:

$$\zeta(x) = \ln(1 + e^x) \quad (2.18)$$

The Mish activation is bounded below and unbounded above, thus providing a degree of regularization and avoiding vanishing gradients, respectively. Being a smooth non-monotonic function it preserves small negative values, which provide better expressivity. Also, as opposed to ReLU, Mish is continuously differentiable.

Convolutional neural networks

In computer vision, the most common networks are **Convolutional Neural Networks** or CNNs. These networks make use of convolutional layers.

The convolutional layers allow for deeper networks and lower number of parameters with filters that extract different features from the former layer.

It can be observed in figure 2.4 the calculations that are being done. A convolution is done between the kernel (i.e. the 3×3 matrix) also named a convolution filter, in this case a Sobel filter (e.g. a filter that extracts the vertical edges) that is applied as a sliding window across the 11×11 matrix, which represents the input features. This type of matrix convolution is used in image processing and as it can be seen a dot product is performed between the kernel and the window, the total sum of these products gives the result of the convolution for that window. This calculation is repeated along the matrix until it can no longer slide the window further.

The following equation represents the shape of the output of such a convolutional calculation for each layer $l \in \{1 \dots m\}$.

$$n^{[l]} = \left\lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \quad (2.19)$$

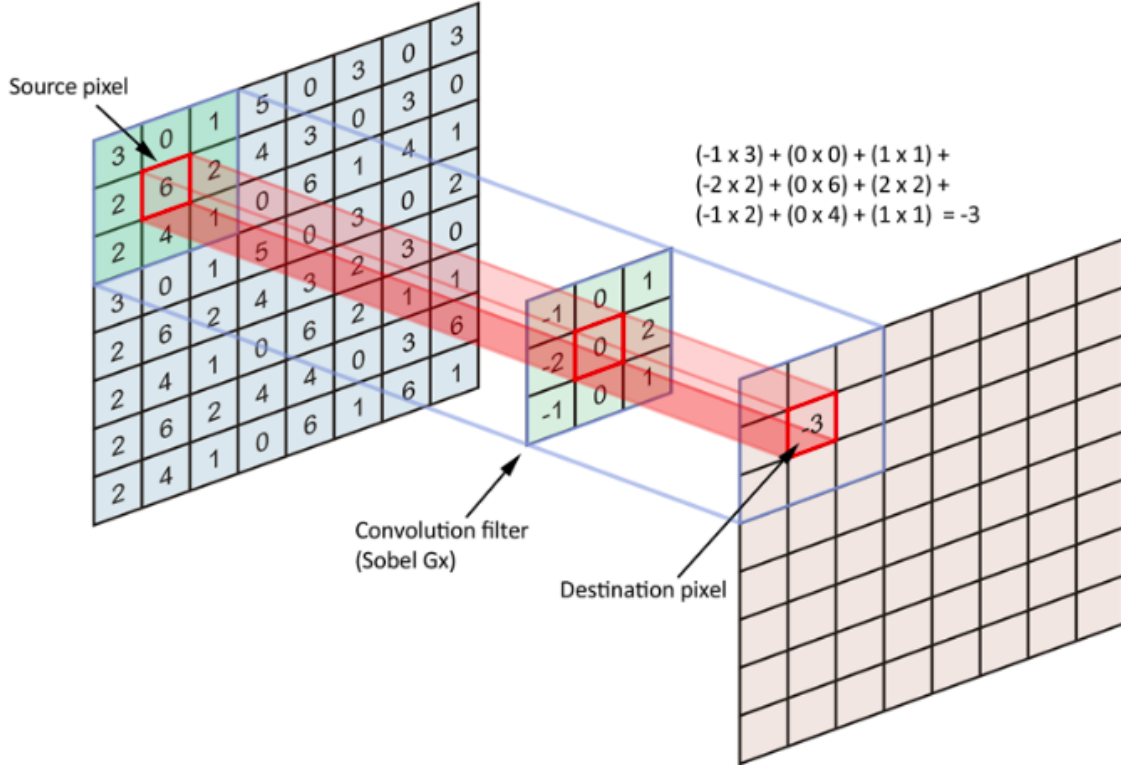


Figure 2.4: Convolutional Layer [3]

The notations used represent the following concepts:

$f^{[l]}$ – filter size

$p^{[l]}$ – padding

$s^{[l]}$ – stride

$n^{[l-1]}/n^{[l]}$ – input/output matrix size

The following equalities can be observed: $f^{[l]} = 3$ $p^{[l]} = 0$ $s^{[l]} = 1$ $n^{[l-1]} = 11$ – figure 2.4

While f or n are self-explained p and s are not.

Padding (p) is used to put an extra set of numbers around the input matrix. In most cases those numbers do not affect the result in any significant way. Generally zeros or neighbouring numbers are used. Padding is used, in general, to keep the dimensions of the input matrix, as a convolution will downsize the image passing through layers. It can also be observed that the edge information is not as used as a center point information is. Thus it can be said that a bit of information is lost along the way. There are two types of padding: "valid" and "same". Valid padding means no padding is added to the input matrix, while same padding means padding is added in order to keep the same dimensions of the input matrix. The formula for how much to pad a matrix, for same padding, is:

$$\frac{s \times (n - 1) - n + f}{2} \quad (2.20)$$

The layer number is no longer necessary for this equation since the size of the input will be equal to the size of the output.

Stride (s) is the number by which the window is sliding across the input matrix.

A convolution over a plain has been covered but, in neural networks, convolutions are usually made over volumes. A well known example of such a volume is a colored image. This image has 3 channels – red, green and blue with matrices where each number represents the intensity of the pixel in that particular channel. When applying a convolutional layer, multiple filters are used. Most used dimensions are: 16, 32 and more filters, but other user-chosen numbers

can be used as well. The final output shape of the convolution will be: $n_h \times n_w \times n_c$. Where n_h and n_w represent the size for height and width, and n_c represents the number of channels.

Each filter will have a size of $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$. The number of channels of the filter is given by the input, in order to make the correct computation. The number of output channels $n_c^{[l]}$ is given by the number of filters applied.

All of these concepts tied together, represent the convolutional layer in a neural network. There can be fully convolutional networks or networks with convolutional layers that end in a few fully connected layers.

The number of parameters to be learned in a convolutional layer are:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]} \quad (2.21)$$

The computational cost of such a layer is the following:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]} \times n^{[l]} \times n^{[l]} \quad (2.22)$$

Other than the conventional layers and the activation layers, there are different layers that help in specific directions such as regularization or normalization.

In order to understand the use of regularization, first the concepts of *underfitting* and *overfitting* have to be explained. These two can be understood in terms of high bias or high variance, respectively. Having a high training error (i.e. the model does not fit the data good enough) means a high bias, thus, underfitting. Having a big difference between the training error and the test error means there is high variance, thus, overfitting. In figure 2.5 a better graphical intuition can be gained. In order to address each of these problems, there are few steps that can be taken. For underfitting the best approaches are getting a bigger network and training the data for longer. For overfitting the best approaches are getting more data or include some regularization techniques in the training process. Both of them can also be addressed through modifying the neural network architecture, but this leads to a lot of experimentation, while the methods mentioned above are proven steps to enhance your performance.

As it can be seen regularization can be used to address overfitting. There are different types of regularization such as ℓ_2 regularization and dropout which are used in this project and which will be explained further.

The ℓ_2 regularization is an addition to the cost function with the Frobenius norm. The function looks like this:

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m \text{loss}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L ||w^{[l]}||_F^2 \quad (2.23)$$

Where λ is the regularization parameter and:

$$||w^{[l]}||_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{i,j}^{[l]})^2 \quad (2.24)$$

Through backpropagation, this leads to the following update of the weights of each layer l :

$$\begin{aligned}
 w^{[l]} &:= w^{[l]} - \alpha[\nabla\mathcal{L}(w^{[l]}) + \frac{\lambda}{m}w^{[l]}] \\
 &:= w^{[l]} - \frac{\lambda\alpha}{m}w^{[l]} - \alpha\nabla\mathcal{L}(w^{[l]}) \\
 &:= (1 - \frac{\lambda\alpha}{m})w^{[l]} - \alpha\nabla\mathcal{L}(w^{[l]})
 \end{aligned}$$

This is why ℓ_2 regularization also gets the name *weight decay*.

The reason why ℓ_2 regularization works, is because it manages to make the fitting more linear, by decreasing the value of weights, thus, some neurons along the network have a lesser impact on the result.

Another way to implement regularization is *Dropout*. This type makes use of a probability to eliminate a number connections from the network, resulting in a smaller network. Because dropout eliminates nodes at random, it spreads the weight impact across, since the node that calculates with the help of the inputs can not rely on any one feature, having a similar effect to ℓ_2 regularization but, with different scaling. One of the disadvantages of dropout is that it reduces the networks capacity, or it is thinning the network, thus, a wider network may be required during training.

Other than regularization there is normalization. In this thesis, a layer named *batch normalization* is used. In neural networks normalization and standardization (which is usually referred to as normalization, as well) are used to bring the input data to a certain smaller scale (e.g. from 10 – 10,000 to 0 – 1), as pre-process step. This normalization helps by reducing the training time and reduces the chance of exploding gradients, because of input features having very large values, which will cascade across the network. The cascade effect is still not completely avoided; some neurons tend to have a much higher impact than others, batch normalization is applied as standardization to the inputs of a layer for each mini-batch, also reducing the number of training epochs needed. This technique adds two more learnable parameters μ and σ as explained bellow:

$$z = \frac{x - \mu}{\sigma} \tag{2.25}$$

Where x represents the input, μ is the mean and σ is the standard deviation. It then multiplies by an arbitrary parameter g and adds another arbitrary parameter b , giving the final form:

$$\hat{x} = zg + b \tag{2.26}$$

One last concept that will be needed for this project is *transfer learning*. If the training dataset is small (e.g. 100 images like the case of this project), transfer learning can be a very good mechanism for getting good results. The idea behind it is that some architectures are trained on very large datasets, such as **ImageNet**, and many of the low end features have been learned. Features such a lines, corners or other low level ones. Through transfer learning, the new network can borrow the learned weights and train just the high level features. The high level features can be just the last layer used for classification, or multiple layers down if the training dataset is larger and it has the confidence that it can improve accuracy. For transfer learning the inputs have to be the same (e.g. images for images, audio for audio). The concept for using just part of the network is known as freezing. Freezing means that the first layers of the network, those that hold the low level features, are frozen or better said, not trained. Data is propagated only forward through the frozen layers and the training process affects only the

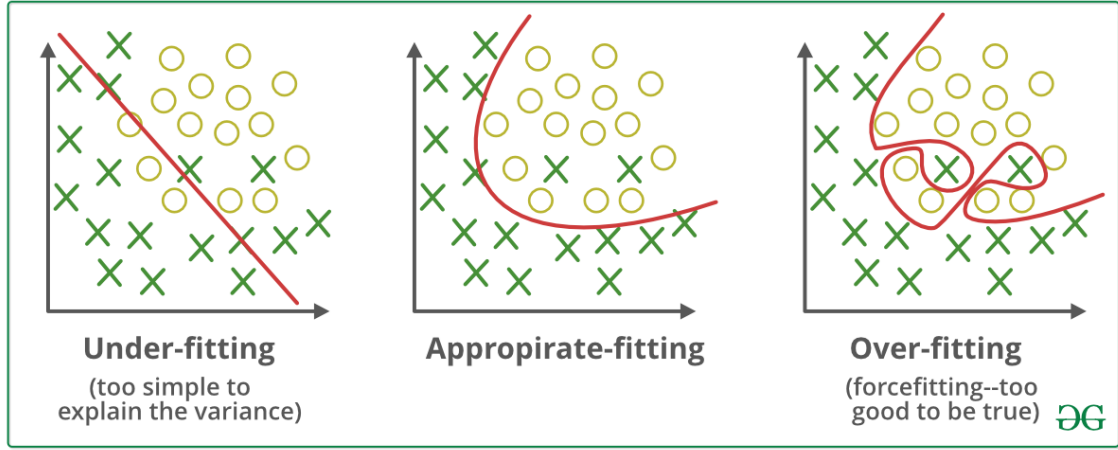


Figure 2.5: Types of data fitting [4]

last layers,

Hyperparameters

As mentioned before the W and b parameters are modified while training such that the neural network will learn to do the given task. In order to achieve the best performance, and the best parameters, various *hyperparameters* have to be tuned. These hyperparameters are the variable elements of the network, such as: α (i.e. learning rate), β (i.e. exponentially weighted averages modifier), number of layers, number of neurons, kernel size, number of filters and others. All these moving parts of the neural network have to be tuned in order to achieve the best performance.

The description of neural networks and the notations were inspired by Andrew Ng's series on Deep Learning [5].

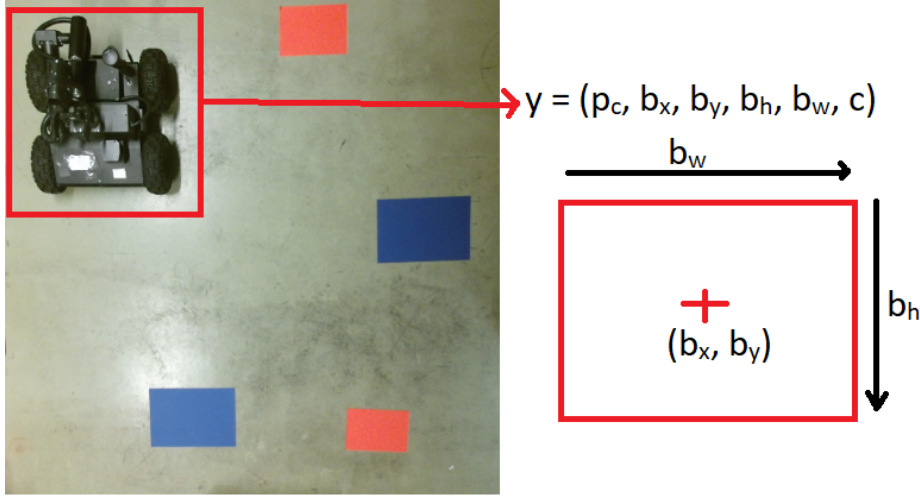
2.2 Modified Tiny YOLO

The first type of neural network used for developing the navigation system, was the object detection network YOLOv3 [18]. Since then, version 4 [19] has been released, by a new team, on 23rd of April, this year. These versions are the state of the art networks in object detection. YOLO provides great accuracy at a high framerate. The chosen version for this project is Tiny YOLO, a small version that came together with the YOLOv3 network. The reason for choosing this version, is that even though, YOLOv3 provides great results, it is expensive in terms of computational and storage resources. Thus, in order to have a network that works on mobile platforms or low-end hardware Tiny YOLO was best suited for this.

Tiny YOLO was created with embedded systems in mind. While the accuracy of Tiny YOLO is lower of mAP – 33.1 compared to YOLOv3 of mAP – 55.3 it is much faster and requires less computational power. The number of FLOPS and FPS of Tiny YOLO are FLOPS – 5.56 Bn and FPS – 220 compared to YOLOv3 FLOPS – 65.86 Bn and FPS – 35.

2.2.1 How does YOLO work?

There are two types of object detection networks: one-stage detectors and two-stage detectors. First, a few two-stage detector examples: Fast R-CNN, Faster R-CNN or R-FCN. These type of detectors, use a sliding window technique to classify and localize objects in a window. The latest versions of R-CNN, such as Faster R-CNN, make use of region proposals and segmentation, in order to classify only where necessary. On the other hand, one-stage detectors, such as: YOLO,



$p_c = 1$: confidence of an object being present in the bounding box

$c = 2$: class of the object being detected (here 2 for "Jaguar")

Figure 2.6: YOLO's Simple Output

SSD or RetinaNet; are end-to-end neural networks, that look only once at the features of the image and provide all the results at once. That is one of the reasons why YOLO stands for - "You Only Look Once". The one-stage detectors provide great speeds while almost retaining the accuracy of two-stage detectors.

YOLO takes as input the matrices of an image representing its pixels. These are taken through what is called a backbone network such as: Darknet, ResNet or VGG16. The backbone is responsible for learning and classifying the image. The second part of an object detector is the head which is responsible for taking the result of the backbone and transforming those into information such as: confidence, class and bounding-box position. Because the YOLO architecture is used, the Tiny YOLO head will be used for the implemented system.

YOLO's simple output is $y = (p_c, b_x, b_y, b_w, b_h, c)$ and can be observed in figure 2.6.

The number of classes can vary from task to task, in this project there are a number of 3 classes (i.e. Red, Blue, Jaguar). Thus, it can be said that each bounding box is represented by 8 numbers, 3 classes plus 1 confidence number and 4 position numbers.

YOLO also makes use of anchors, which are height/width ratios that represent different classes. For example a human may not have the same ratio as a car – skinny and tall as opposed to wide and short. Anchors are very useful when two objects overlap in a image allowing the user to detect both correctly. After encoding for example an image of shape $[m \times 416 \times 416 \times 3]$, the final output would be of shape $[m \times 13 \times 13 \times 3 \times 8]$.

As stated before 8 represents the bounding box, 3 is the number of anchors, 13×13 represents the grid by which YOLO detects objects and m is the number of the example to be detected.

From the specified grid if an object is found in one of the grid cells, it will be used for detecting the object. A representation of the grid can be seen in figure 2.7, where the cell painted in red is the grid cell in which part of the object was found.

The network does this for all cells and through training it will get closer and closer to the ground truth. Due to the fact that there are many cells and anchors, when the network will detect, there will be more bounding boxes than necessary. There are two ways, to clear them out and get the best results. First is a threshold with regards to confidence, in this case $p_c * c$,

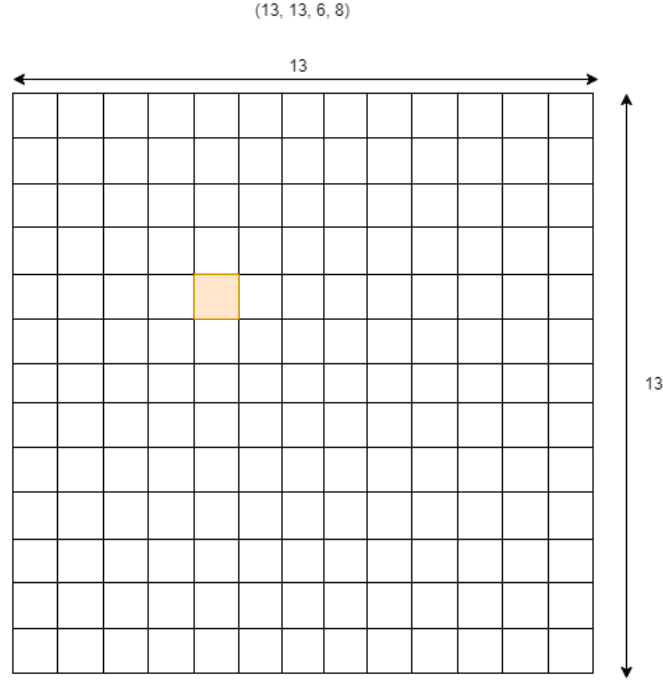


Figure 2.7: Image Grid

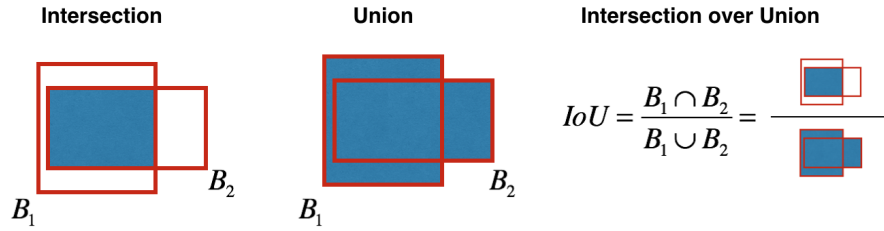


Figure 2.8: Intersection over Union [5]

this equation makes use of `Python` broadcasting. Usually, if this confidence for a box is lower than 0.5, that box is eliminated; this threshold can be chosen by the user. Another way to eliminate the remaining boxes is through *non-max suppression*. Non-max suppression oversees boxes that overlap each other by a certain threshold and that share the same class. It selects the box with the highest score and iteratively removes the boxes that significantly overlap (i.e. over the specified threshold). In order to do this it makes use of IoU (Intersection over Union) seen in figure 2.8. The IoU is the parameter that has to be above the set threshold.

Another concept that has to be taken into consideration is that YOLO predicts across multiple scales [18]. Previous versions of YOLO had trouble with small objects thus, it now makes use of upsampling at the end of the network similar to pyramid networks. It makes use of previous feature maps and upsamples it by 2. At the end these two outputs are concatenated and then the final features are computed with 2 more convolutional layers. Giving the network two grids for computing features, the original 13×13 one and a 26×26 one, after upsampling.

With all these concepts detailed, the backbone of the the `Modified Tiny YOLO` can be presented. Only the backbone is presented, as this is the part of the network that has been significantly modified.

The original `Tiny YOLO` architecture, has the following structure:

```

1 def tiny_yolo_body(inputs, num_anchors, num_classes):
2     '''Create Tiny YOLO_v3 model CNN body in keras.'''
3     x1 = compose(
4         DarknetConv2D_BN_Leaky(16, (3,3)),

```

```
5         MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
6         DarknetConv2D_BN_Leaky(32, (3,3)),
7         MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
8         DarknetConv2D_BN_Leaky(64, (3,3)),
9         MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
10        DarknetConv2D_BN_Leaky(128, (3,3)),
11        MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
12        DarknetConv2D_BN_Leaky(256, (3,3)))(inputs)
13    x2 = compose(
14        MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
15        DarknetConv2D_BN_Leaky(512, (3,3)),
16        MaxPooling2D(pool_size=(2,2), strides=(1,1), padding='same'),
17        DarknetConv2D_BN_Leaky(1024, (3,3)),
18        DarknetConv2D_BN_Leaky(256, (1,1)))(x1)
19    y1 = compose(
20        DarknetConv2D_BN_Leaky(512, (3,3)),
21        DarknetConv2D(num_anchors*(num_classes+5), (1,1)))(x2)
22
23    x2 = compose(
24        DarknetConv2D_BN_Leaky(128, (1,1)),
25        UpSampling2D(2))(x2)
26    y2 = compose(
27        Concatenate(),
28        DarknetConv2D_BN_Leaky(256, (3,3)),
29        DarknetConv2D(num_anchors*(num_classes+5), (1,1)))([x2,x1])
30
31    return Model(inputs, [y1,y2])
```

Parameters for Tiny YOLO:

```
=====
Total params: 8,680,864
Trainable params: 8,674,496
Non-trainable params: 6,368
-----
```

In this case `DarknetConv2D_BN_Leaky` represents a Convolutional Layer with ℓ_2 kernel regularizer, a Batch Normalization Layer and a LeakyReLU activation, while a simple `DarknetConv2D` is just a Convolution Layer with ℓ_2 kernel regularizer. Either of the two have as inputs the number of filters and the kernel size, as in the case of the first one, with a number of 16 filters and a kernel size of 3×3 .

As it can be seen from the output convolutions the final shapes are $13 \times 13 \times 24$ and $26 \times 26 \times 24$. This time having the last two dimensions flattened (i.e. a shape of $13 \times 13 \times 3 \times 8$ to a shape of $13 \times 13 \times 24$).

With the help of various modifications the number of parameters will be reduced in the Modified Tiny YOLO version.

The modified version has the following structure:

```
1 def bsctiny_yolo_body(inputs, num_anchors, num_classes):
2     '''Create Tiny YOLO_v3 model CNN body in keras.'''
3     x1 = compose(
4         _bs_conv(filters=32, strides=(1,1)),
5         MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
6         _bs_conv(filters=32, strides=(1,1)),
7         MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
8         _bs_conv(filters=64, strides=(1,1)),
9         MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
10        _bs_conv(filters=128, strides=(1,1)),
11        MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
12        _bs_conv(filters=256, strides=(1,1))
13        )(inputs)
14    x2 = compose(
15        MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='same'),
16        _bs_conv(filters=512, strides=(1,1)),
17        MaxPooling2D(pool_size=(2,2), strides=(1,1), padding='same'),
```

```

18         _bs_conv(filters=1024, strides=(1,1)),
19         _bs_conv(filters=256, strides=(1,1), kernel_size=(1,1)))(x1)
20     y1 = compose(
21         _bs_conv(filters=512, strides=(1,1)),
22         DarknetConv2D(num_anchors*(num_classes+5), (1,1)))(x2)
23
24     x2 = compose(
25         _bs_conv(filters=128, strides=(1,1), kernel_size=(1,1)),
26         UpSampling2D(2))(x2)
27     y2 = compose(
28         Concatenate(),
29         _bs_conv(filters=256, strides=(1,1)),
30         DarknetConv2D(num_anchors*(num_classes+5), (1,1)))([x2,x1])
31
32     return Model(inputs, [y1,y2])

```

Parameters for Modified Tiny YOLO:

```

=====
Total params: 168,263
Trainable params: 155,263
Non-trainable params: 13,000
-----

```

As it can be seen the number of parameters has been drastically reduced. The **Modified Tiny YOLO** structures achieves a reduction in parameters of 98.1% from the original number of parameters. It can also be observed that the number of *Non-trainable parameters* has increased; this will be discussed later. The overall structure of the architecture has not been modified, but now the training is approached differently. The numbers of filters have been increased from 16 to 32 for the first layer. The most striking difference is the one between **DarknetConv2D_BN_Leaky** to **_bs_conv**, which stands for *blueprint separable convolution* [6]. With the help of this type of convolution the number of parameters has been reduced, while getting a better inference and accuracy.

2.2.2 Modified Tiny YOLO Concepts

In order to understand *Blueprint Separable Convolutions*, *Depthwise Separable Convolutions* must be acknowledged first. The concept of depthwise separable convolutions was taken from the implementation of MobileNetv1 [20].

Depthwise separable convolutions make use of a depthwise convolution and a pointwise convolution which is a 1×1 convolution. A depthwise convolution applies convolution to each channel of the input. The pointwise convolution creates a linear combination of the result for the number of filters that need to be applied, in a extra-kernel approach.

It can be recalled that a normal convolution applies a filter to an input feature map and then combining the results to create a new feature map, having finer more complex details. The concept of depthwise separable convolution splits this act in two, by applying the filter through the depthwise convolution and combining the results with a pointwise convolution.

In subsection 2.1 the computational cost of a convolutional layer was presented as:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]} \times n^{[l]} \times n^{[l]} \quad (2.27)$$

Taking first the cost of a depthwise convolution:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n^{[l]} \times n^{[l]} \quad (2.28)$$

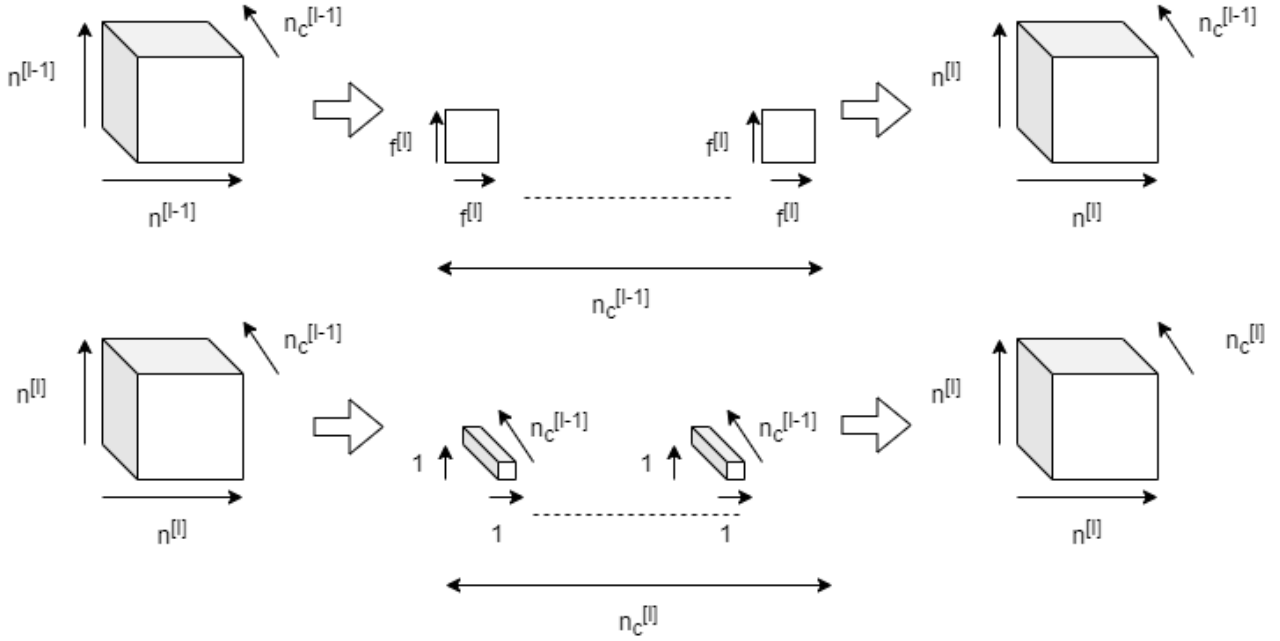


Figure 2.9: Depthwise Separable Convolution

Combined with the cost of the pointwise convolution that follows it:

$$n_c^{[l-1]} \times n_c^{[l]} \times n^{[l]} \times n^{[l]} \quad (2.29)$$

It can be seen that a smaller cost is achieved, without affecting the overall results of the network:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n^{[l]} \times n^{[l]} + n_c^{[l-1]} \times n_c^{[l]} \times n^{[l]} \times n^{[l]} \quad (2.30)$$

The amount of the reduction is of:

$$\frac{f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n^{[l]} \times n^{[l]} + n_c^{[l-1]} \times n_c^{[l]} \times n^{[l]} \times n^{[l]}}{f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]} \times n^{[l]} \times n^{[l]}} = \frac{1}{n_c^{[l]}} + \frac{1}{f^{[l],2}} \quad (2.31)$$

Using a 3×3 depthwise convolution achieves a reduction in parameters of 8 to 9 times smaller (i.e. from 8,680,864 to 1,085,108) It can be observed that already a significant drop in the number of parameters has been achieved without affecting the network's accuracy.

To better understand how the number of parameters is reduced a graphical representation will be shown in figure 2.9.

It can be observed from the figure that the depthwise convolution passes through the input one channel at a time creating a new filtered map with the same number of channels, then the pointwise convolution makes the combination of the previous result, giving the final feature map across the desired number of channels.

The number of parameters can be computed with the help of the following equations:

For the depthwise convolution the number of trainable parameters is:

$$n_c^{[l-1]} \times f^{[l]} \times f^{[l]} \quad (2.32)$$

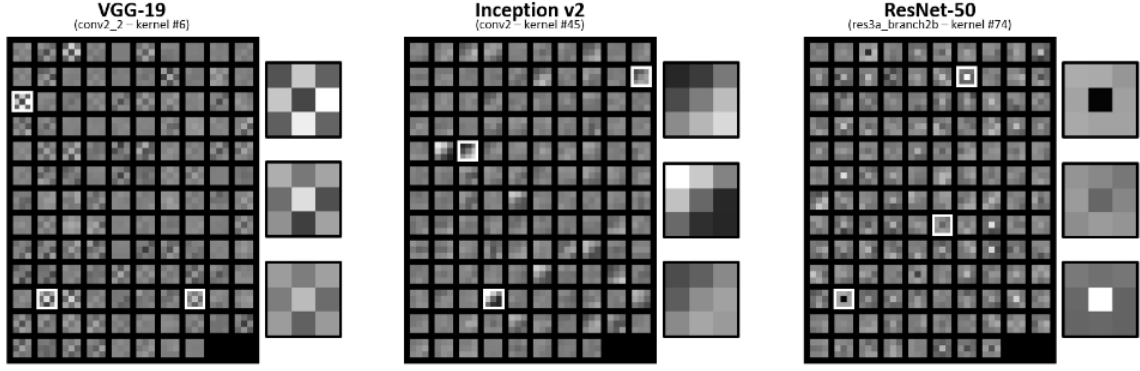


Figure 2.10: Filter Correlation along Depth Axis [6]

For the pointwise convolution the number of trainable parameters is:

$$n_c^{[l-1]} \times n_c^{[l]} \quad (2.33)$$

The reduction compared to a normal convolution will be the same as it was for the computational cost:

$$\frac{n_c^{[l-1]} \times f^{[l]} \times f^{[l]} + n_c^{[l-1]} \times n_c^{[l]}}{n_c^{[l-1]} \times n_c^{[l]} \times f^{[l]} \times f^{[l]}} = \frac{1}{n_c^{[l]}} + \frac{1}{f^{[l],2}} \quad (2.34)$$

In the MobileNet architecture there are two more concepts: *width multiplier* and *resolution multiplier*. Two more hyperparameters that allow to change the width of the layers and the resolution of the input but they are not used in this version of Modified Tiny YOLO. While they do help in achieving a much smaller network, they significantly affect its accuracy.

In order to bring the number of multiplications-additions and parameters to an even lower amount a new version of separable convolutions was issued - *the blueprint separable convolutions* [6].

These type of convolution still make use of a separation of general convolutions by using pointwise and depthwise convolutions. The difference between these and depthwise separable convolutions is that they first make use of the blueprint created by the pointwise convolution and then apply the filter of the depthwise convolution.

Further on, *blueprint separable convolution* and *depthwise separable convolution* will be referred to as BSConv and DSC respectively.

While DSC makes use of cross-kernel correlations BSConv makes use of intra-kernel correlations. The team behind BSConv [6] showed intra-kernel correlation along the depth axis. Observing that the slices for a filter have the same $f^{[l]} \times f^{[l]}$ 'blueprint' with variations like a negative scale or simply an inverted version. They determined the variance of the filter kernel through PCA. The correlation and variation can be seen in figures 2.10 and 2.11. With the axis of 2.11 being 'filter count' on the O_y axis and 'filter kernel variance explained by PC1 [%]' on the O_x axis with the same networks being tested.

To better represent how to BSConv operates a few graphical representation will be shown to compare and to go through the steps it takes in figures 2.12, 2.13 and 2.14.

There are two types of BSConvs proposed by [6]. The unconstrained one which is just the reverse of DSC but has a bigger cost in terms of computation as opposed to DSC, while retaining

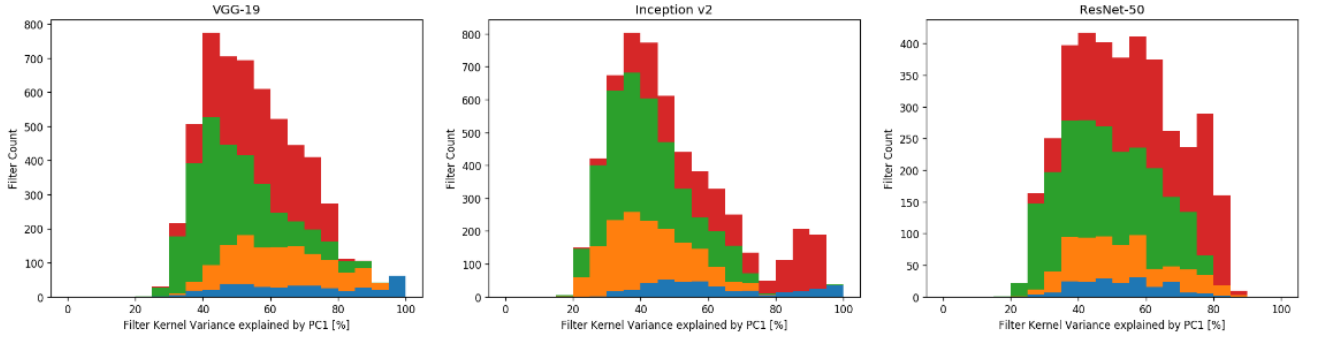


Figure 2.11: Filter Variance [6]

almost the same number of parameters of DSC. The second type is the **Subspace BSConv** that provides much better results.

In order to see the difference the cost and number of parameters formulas will be deduced: First layer computational cost for **Unconstrained BSConv**:

$$n^{[l-1]} \times n^{[l-1]} \times n_c^{[l-1]} \times n_c^{[l]} \quad (2.35)$$

Last layer computational cost for **Unconstrained BSConv**:

$$f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l]} \quad (2.36)$$

This gives a total reduction in computational cost for the **Unconstrained BSConv** of:

$$\frac{n^{[l-1]} \times n^{[l-1]} \times n_c^{[l-1]} \times n_c^{[l]}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}} + \frac{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l]}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}} \quad (2.37)$$

After simplification the final result will be:

$$\frac{n^{[l-1]} \times n^{[l-1]}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]}} + \frac{1}{n_c^{[l-1]}} \quad (2.38)$$

First layer computational cost for **Subspace BSConv**:

$$n^{[l-1]} \times n^{[l-1]} \times n_c^{[l-1]} \times \frac{n_c^{[l]}}{\alpha} \quad (2.39)$$

Second layer computational cost for **Subspace BSConv**:

$$n^{[l-1]} \times n^{[l-1]} \times \frac{n_c^{[l]}}{\alpha} \times n_c^{[l]} \quad (2.40)$$

Last layer computational cost for **Subspace BSConv**:

$$f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l]} \quad (2.41)$$

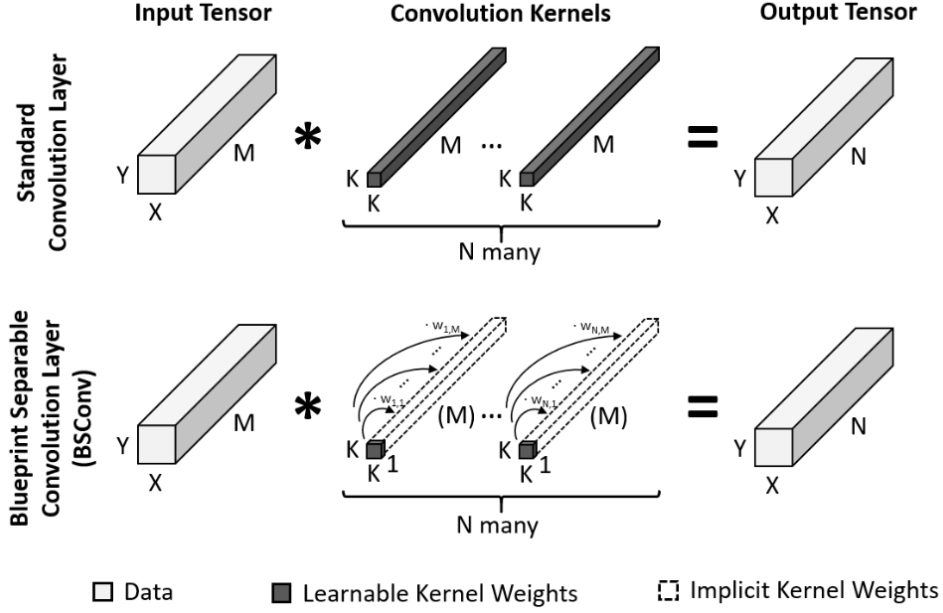


Figure 2.12: BSConv comparison with Standard Convolution Layer [6]

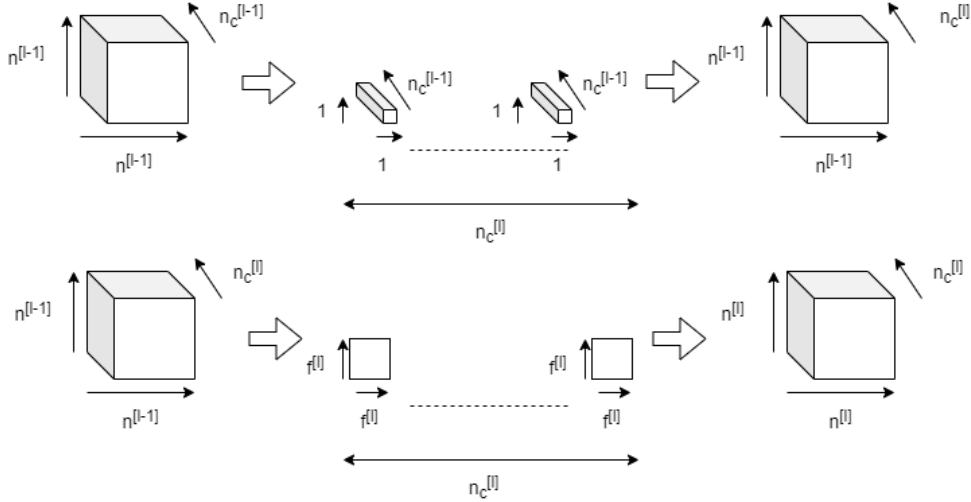


Figure 2.13: Unconstrained BSConv

This gives a total reduction in computational cost for the **Subspace** BSConv of:

$$\begin{aligned}
 & \frac{n^{[l-1]} \times n^{[l-1]} \times n_c^{[l-1]} \times \frac{n_c^{[l]}}{\alpha}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}} + \frac{n^{[l-1]} \times n^{[l-1]} \times \frac{n_c^{[l]}}{\alpha} \times n_c^{[l]}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}} \\
 & + \frac{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l]}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}}
 \end{aligned} \tag{2.42}$$

After simplification the final result will be:

$$\frac{n^{[l-1]} \times n^{[l-1]}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]}} \times \frac{1}{\alpha} + \frac{n^{[l-1]} \times n^{[l-1]} \times \frac{n_c^{[l]}}{\alpha}}{f^{[l]} \times f^{[l]} \times n^{[l]} \times n^{[l]} \times n_c^{[l-1]}} + \frac{1}{n_c^{[l-1]}} \tag{2.43}$$

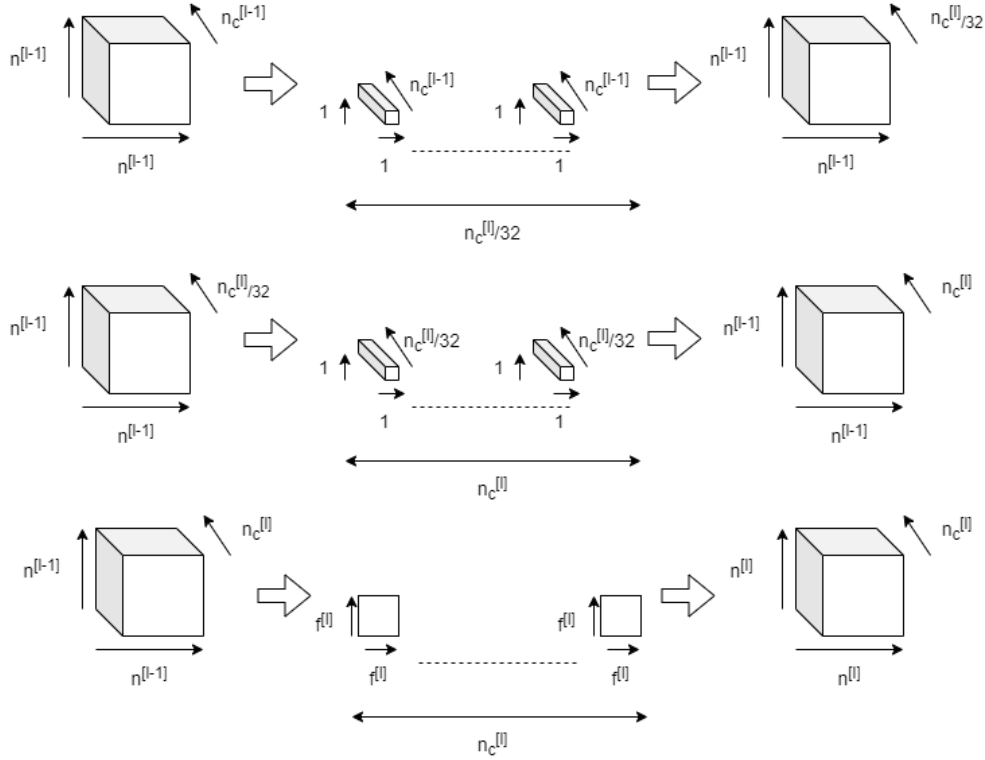


Figure 2.14: Subspace BSConv

It can be observed that the computational cost for the **Subspace BSConv** decreases much quicker than **DSC** and it is a lot smaller than that of **Unconstrained BSConv**.

The α parameter used in the **Subspace BSConv** is a positive integer number chosen by the user, divisor of $n_c^{[l]}$. This creates a subspace that reduces the number of parameters used by the normal pointwise convolution and outputs the desired $n_c^{[l]}$ number of filters. The number 32, as seen in the figure 2.14 is the α chosen in the **Modified Tiny YOLO** architecture, giving the best results.

The number of parameters for the **Unconstrained BSConv** remains similar to **DSC** as it can be seen in the following equations:

For the first layer – pointwise convolution:

$$n_c^{[l-1]} \times n_c^{[l]} \quad (2.44)$$

For the last layer – depthwise convolution:

$$n_c^{[l]} \times f^{[l]} \times f^{[l]} \quad (2.45)$$

With the reduction being of:

$$\frac{n_c^{[l-1]} \times n_c^{[l]}}{n_c^{[l-1]} \times n_c^{[l]} \times f^{[l]} \times f^{[l]}} + \frac{n_c^{[l]} \times f^{[l]} \times f^{[l]}}{n_c^{[l-1]} \times n_c^{[l]} \times f^{[l]} \times f^{[l]}} = \frac{1}{n_c^{[l-1]}} + \frac{1}{f^{[l]} \times f^{[l]}} \quad (2.46)$$

It can be observed that the number of parameters for the **Unconstrained BSConv** is slightly higher than that of **DSC** because the first term is divided by the channels of the previous layer. The reason why it is just slightly higher is because the number of filters usually increases very quickly, thus, the first term has only a small influence on the overall result.

Until now the **Unconstrained BSConv** does not seem to be much better than the **DSC**, it has a bigger computational cost and almost the same number of parameters. The reason behind

the higher computational cost is that it first uses a pointwise convolution and as presented in both [20] and [6] papers, over 95% of the parameters can be found there and more than 70% of the computational cost is represented by the pointwise convolutions. Having the pointwise first means more computations have to be performed since the input has more features to be computed (i.e. instead of having the smaller $n_c^{[l]} \times n_c^{[l]}$ feature map it has the original $n_c^{[l-1]} \times n_c^{[l-1]}$ which is almost always bigger and has to combine those features into a new number of channels). It does not hold only disadvantages, since the feature map given, first uses a pointwise, the depthwise convolution can fully use the feature maps – as stated by the **BSConv** team – ”feature maps from the first regular convolution can be fully utilized by the depthwise convolution via the preceding pointwise distribution. In contrast, each kernel of the first depthwise convolution of the original **MobileNetV1** model can only benefit from a single feature map, leading to limited expressiveness” [6].

On the other hand the **Subspace BSConv** has a much lower number of parameters as it can be seen from the following sequence of formulas:

For the first layer - pointwise convolution:

$$\frac{n_c^{[l]}}{\alpha} \times n_c^{[l-1]} \quad (2.47)$$

For the second layer - pointwise convolution:

$$n_c^{[l]} \times \frac{n_c^{[l]}}{\alpha} \quad (2.48)$$

For the last layer - depthwise convolution:

$$n_c^{[l]} \times f^{[l]} \times f^{[l]} \quad (2.49)$$

With the reduction being of:

$$\begin{aligned} & \frac{\frac{n_c^{[l]}}{\alpha} \times n_c^{[l-1]}}{n_c^{[l-1]} \times n_c^{[l]} \times f^{[l]} \times f^{[l]}} + \frac{n_c^{[l]} \times \frac{n_c^{[l]}}{\alpha}}{n_c^{[l-1]} \times n_c^{[l]} \times f^{[l]} \times f^{[l]}} + \frac{n_c^{[l]} \times f^{[l]} \times f^{[l]}}{n_c^{[l-1]} \times n_c^{[l]} \times f^{[l]} \times f^{[l]}} = \frac{1}{n_c^{[l-1]}} + \\ & + \frac{n_c^{[l]}}{\alpha \times n_c^{[l-1]} \times f^{[l]} \times f^{[l]}} + \frac{1}{\alpha \times f^{[l]} \times f^{[l]}} \end{aligned} \quad (2.50)$$

It can be observed that the reduction in parameters is much bigger and that is why the number of parameters of the **Modified Tiny YOLO** is 155,263, while the number of parameters for **Tiny YOLO** is 8,674,496. Moreover, because of the advantage mentioned for the **Unconstrained BSConv** the **Modified Tiny YOLO** gets better accuracy results compared to **Tiny YOLO** when training from scratch and similar results or even slightly better when **Tiny YOLO** makes use of *transfer learning*.

On the other hand, the number of non-trainable parameters has increased. It's because convolution function is now being formed out of three layers and has two more batch normalization layers for the first two pointwise convolutions. These layers do not have an activation function attached, only the final depthwise layer does.

2.2.3 Results of Modified Tiny YOLO

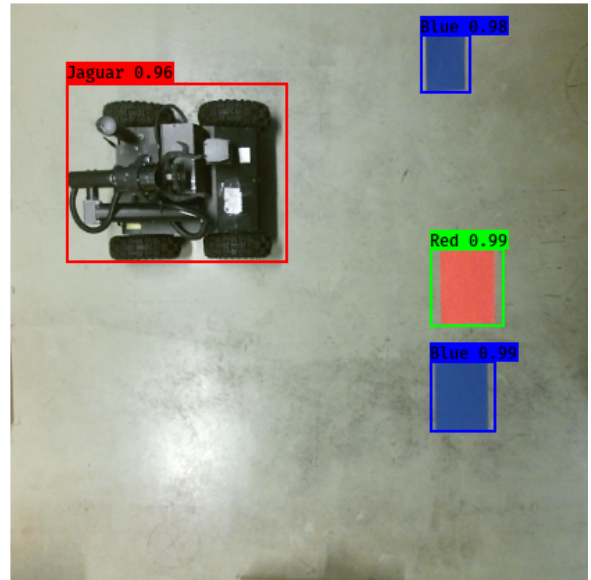
With a dataset of just 100 images the network achieves an accuracy of above 96%. An example can be seen in figure 2.15a, whereas visual results are depicted in figure 2.15b. As a result of

```

(416, 416, 3)
2020-06-11 15:45:16.410394: E tensorflow_core: Subshape must have computed
3372036854775807 over shape with
2020-06-11 15:45:16.642657: E tensorflow_core: Subshape must have computed
3372036854775807 over shape with
Found 4 boxes for img
Jaguar 0.96 (40, 57) (199, 186)
Red 0.99 (302, 176) (355, 232)
Blue 0.98 (295, 22) (331, 64)
Blue 0.99 (302, 257) (349, 308)
1.7806585000000013

```

(a) Detection Results



(b) Detection Image Output

Figure 2.15: Modified Tiny YOLO Results

the low number of parameters the output weights are only 894 KB compared to the 34 MB of the normal architecture. The size of the weights is given for the **Keras** implementation of the network, not the original one implemented in **C**, with **.h5** format file.

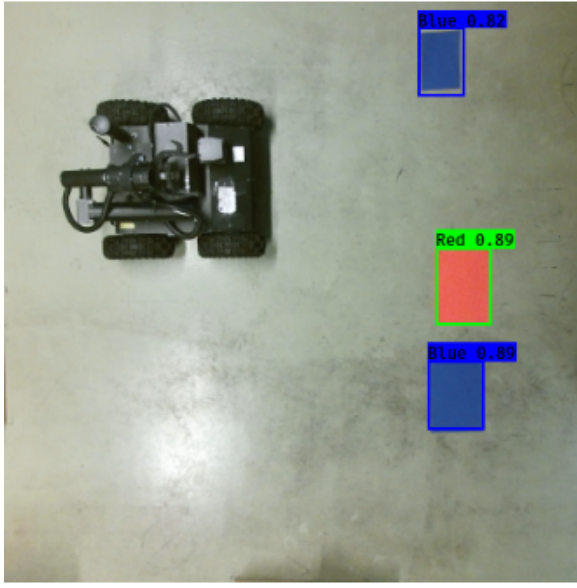
Training **Tiny YOLO** from scratch on the normal architecture gets lower accuracy for detecting *Red* and *Blue* classes and it is having trouble detecting the *Jaguar* class as it can be seen in figure 2.16a. Testing on other different images the normal architecture struggles with red and blue objects as well. On the other hand using transfer learning the network does get better results, but it still struggles with finding the 4x4 Jaguar Platform as it can be seen in figure 2.16b, also the accuracy for the red and blue objects is lower than that of **Modified Tiny YOLO** version.

All three networks were tested over 100 epochs with a batch of 8 and a learning rate of 0.01 for the first 50 epochs and a learning rate of 0.001 for the last 50 epochs.

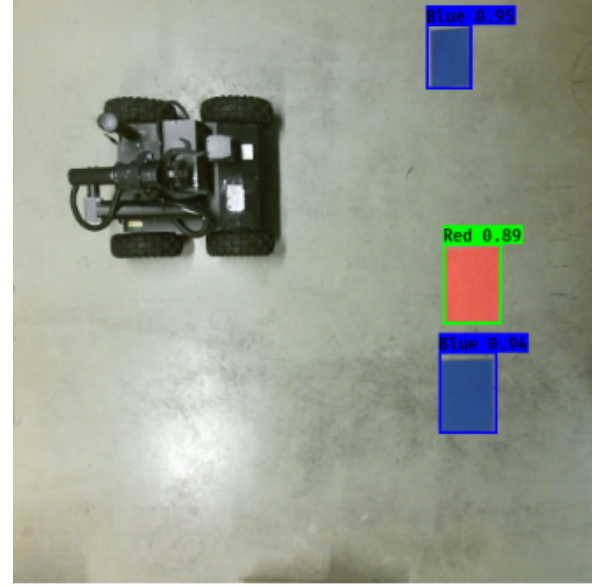
2.2.4 YOLO Comparison

As it can be seen, the **Modified Tiny YOLO** model is a big improvement over the standard architecture of **Tiny YOLO**. The areas where the architecture was improved is in its number of parameters and accuracy.

There are a lot of features, methods and techniques that improve networks and that also improve inference. More and more of these techniques are generally applicable. The *Blueprint Separable Convolution* and the *Depthwise Separable Convolution* are clear examples of this. More and more teams are providing small, well performing networks and are improving and creating different methods constantly. As opposed to early days when the main focus of building neural networks was on how to build the largest network with the most parameters, due to the rise of IoT and mobile devices, small networks that are able to work on low end or small devices are needed more and more.



(a) Detection Image Output



(b) Detection Image Output with Transfer Learning

Figure 2.16: Tiny YOLO Results

Architecture	No. of Layers	Loss Function	Optimizer
Tiny YOLO	20	SSE	Adam
Tiny YOLO w/ transfer learning	20	SSE	Adam
Modified Tiny YOLO	42	SSE	Adam

Table 2.1: Main features of used NNs

Architecture	No. of Conv	No. of Pool Layers	No. of Depthwise Conv	No. of Pointwise Conv
Tiny YOLO	13	6	-	-
Tiny YOLO w/ transfer learning	13	6	-	-
Modified Tiny YOLO	36	6	11	22

Table 2.2: Layers of used NNs

Architecture	Red	Blue	Jaguar
Tiny YOLO	0.89	0.89	-
Tiny YOLO w/ transfer learning	0.89	0.95	0.51
Modified Tiny YOLO	0.99	0.99	0.96

Table 2.3: Accuracy of used NNs

The accuracy for each class has been calculated by taking the values predicted for the test images and making the average.

Architecture	No. of trainable params	No. of non- trainable params
Tiny YOLO	8,674,496	6,368
Tiny YOLO w/ transfer learning	8,674,496	6,368
Modified Tiny YOLO	155,263	13,000

Table 2.4: Parameters of used NNs

The tables provide some interesting views on how the networks compare. As it can be seen the number of layers for the **Modified Tiny YOLO** is larger, but the types of layers are different. Although the layers are of different types their main objectives are the same, making a convolutional layer. It can be seen that the approach of the **BSConvs** is better by lowering the number of parameters and increasing the accuracy, making it even better than **Tiny YOLO** with transfer learning.

2.3 Patch Model

The **Patch Model** is a very different architecture compared to **YOLO**. This architecture was build from scratch as a two-stage detector, with a completely different strategy. It is trained purely as a classifier, and afterwards a post-processing step is employed before it is fed to the path-finding algorithm.

2.3.1 Dataset for Patch Model

To train such structure, a different type of dataset had to be built. This dataset is composed of 16×16 pixels images, basically splitting the images from the original dataset into a 26×26 grid and labeling each patch of 16×16 pixels with its corresponding class. This type of dataset was achieved by making, first a patch annotation script and after that a patch split script.

The patch annotation script takes the **XML** annotation of each image and slides a window across the shape of the image. If the window is inside the bounds of any object then it labels that patch with the objects name, and if does not find itself inside the bounds of any object then it labels that patch as background. In order to make sure that the patch is inside the bounds of an object since the objects do not match the grid perfectly, all corners were taken into account and multiple checks were done in order to asses if any corner of the patch is inside the bounds of an object. As an example: if the bottom right corner of the patch is taken as assessment point then – if x_{max} of the patch is bigger than x_{min} and smaller than x_{max} of the bounding box and if y_{max} of the patch is bigger than y_{min} and smaller than y_{max} of the bounding box then the patch is inside the object's borders. For clarity: (x_{min}, y_{min}) – top left corner, (x_{min}, y_{max}) – bottom right corner, (x_{max}, y_{min}) – top right corner and (x_{max}, y_{max}) – bottom right corner. To verify a good representation of the target object is captured, another parameter was added, the **RANGE_INT**. This range is a variable that is added or subtracted from x_{min} , y_{min} , x_{max} or y_{max} , thus, making the bounding box smaller and making sure the annotation captures more of the object and not a large part of background and just a small part of the object.

The information is then stored in another **XML** file with the following information: all new 676 bounding boxes with their positions and class names, together with the image path from which all the bounding boxes were created.

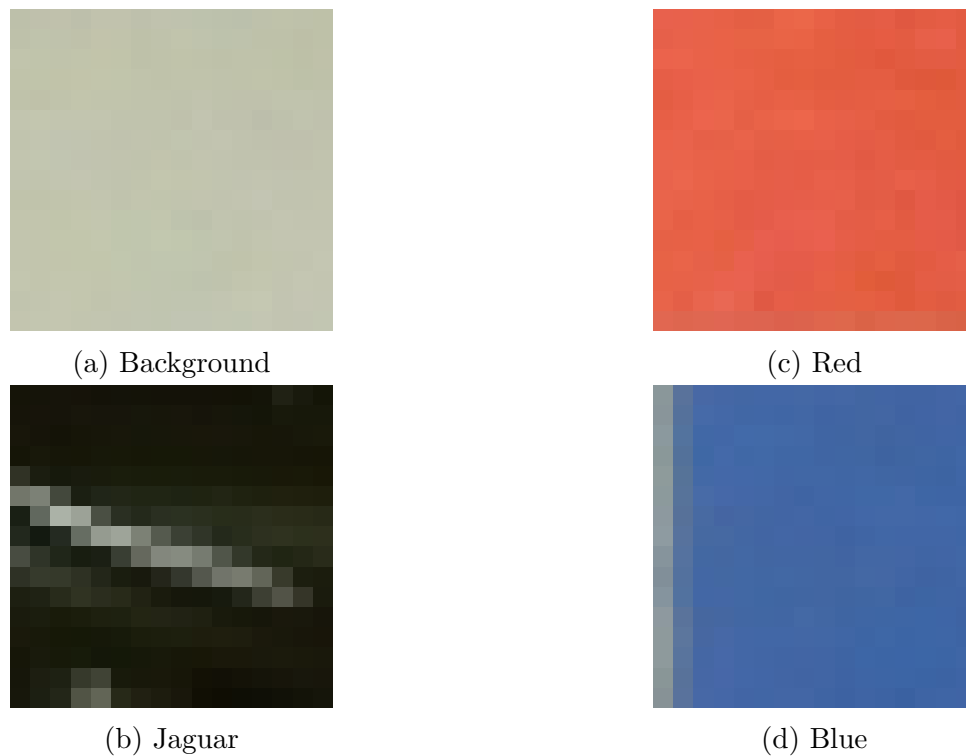


Figure 2.17: Patch Dataset

Because the number of 676 bounding boxes is very large and because a large percent of them represent background, the split for background is done separately, as it would take too much time to process all the annotations and image splitting.

Image splitting is the second part of morphing the dataset according to the patch model. Reading the newly created annotations in the XML format, the splitting script makes use of the path of the image and the class names; for each class name the exact bounding box is cut from the image and stored as a new smaller image in a folder representing that class (e.g. Red), thus, created 4 folders: Background, Red, Blue and Jaguar with new, hundreds of 16×16 images each. This forms the final dataset for the patch model.

The results can be seen in figure 2.17.

Although this is the final dataset, it still needs to be pre-processed in order to input it into the network. To do this all images are read by class name and added as `numpy` arrays to an x training set, at the same time, the class name is added to another list y , which represents the labels of the training set. The `numpy` array is then reshaped with the following shape $(-1, 16, 16, 3)$, where -1 is representative for the number of examples passed (i.e. if there are 164 images then -1 is representative for 164), 16 and 16 represent the size of the image by width and height and 3 is the number of channels as these pictures are colored, using RGB encoding. These sets are then saved in two `.pickle` files for further use in a neural network. `Pickle` is a `Python` method used for serializing and de-serializing information. In this case the information is converted in memory as a byte stream and saved in the `.pickle` file, that can be later de-serialized for use in a program. A pickle serialization should not be confused with compression, which encodes data to occupy less space on disk.

2.3.2 The Keras model

The neural network was built in `Keras` and it has the `X.pickle` input normalized by 255 as pixels have a range of $[0, 255]$. Multiple variants were experimented with, giving the final model

which looks like this:

```
1 model = Sequential()
2 model.add(Conv2D(32, (3, 3), input_shape=X.shape[1:], kernel_regularizer=l2(5e-4)))
3 model.add(Activation('relu'))
4 model.add(MaxPooling2D(pool_size=(2, 2)))
5
6 model.add(Conv2D(64, (3, 3), kernel_regularizer=l2(5e-4), padding='same'))
7 model.add(Activation('relu'))
8 model.add(Dropout(.2))
9
10 model.add(Conv2D(128, (3, 3), kernel_regularizer=l2(5e-4), padding='same'))
11 model.add(Activation('relu'))
12 model.add(Dropout(.2))
13
14 model.add(Conv2D(256, (3, 3), kernel_regularizer=l2(5e-4), padding='same'))
15 model.add(Activation('relu'))
16 model.add(Dropout(.2))
17
18 model.add(Flatten())
19
20 model.add(Dense(4))
21 model.add(Activation('softmax'))
```

It can be seen that the network has 4 convolutional layers with an increasing number of filters. Although creating a model is an experimental process where libraries like **Keras Tuner** provide tools for even faster experimentation, some of the parameters can be explained. First of all **Keras Tuner** was not used in this network, a manual approach was used, with the help of for loops and lists and with the help of the **TensorBoard** library for comparison between networks.

The number of 4 convolutional layers has been reached by first starting with just one layer. The more layers a network has, the more complex features it can encode in its weights.

Along the convolutional layers there are activation functions of type ReLU. In this case a LeakyReLU is not needed as it does not get any significant improvements, thus, a simple ReLU function was chosen.

A pooling layer was also added after the first convolutional layer to downsample the image and summarize the feature map created by the convolutional layer. This is useful for the robustness of the feature map.

The last is a dense layer (i.e. a fully connected layer) with a softmax activation function in order to output the four types of classes with values ranging from 0 to 1. In order to add the dense layer the input from the previous layers has to be flattened.

There is one more important feature for the patch model, the addition of the regularization technique of dropout. After training the model with just convolutional layers and the final dense layer, according to the validation dataset, the model was overfitting. In order to stop it from overfitting ℓ_2 regularization was used but its impact was limited. Dropout improved the generalization of the networks, thus leading to better results. The problem that overfitting posed in this situation was that it was overfitting for the 4x4 Jaguar Platform class and it would not detect it properly, most of the time mistaking it for background. While the dropout layers did not make the model perfect, the jaguar class is a lot more visible.

The prediction script

The image is split in a grid, a 26×26 grid with 16×16 patches of pixels. The prediction makes use of a sliding window technique instead of an end to end prediction of all patches. The grid gives the number of predictions per image. Each patch is classified using the model and the image is re-stitched using the predicted patches. Each classified patch is replaced by specific color of the label: white for background, black for the Jaguar Platform, red for Red Obstacle and blue for the Blue Pass Through Object.

Position data

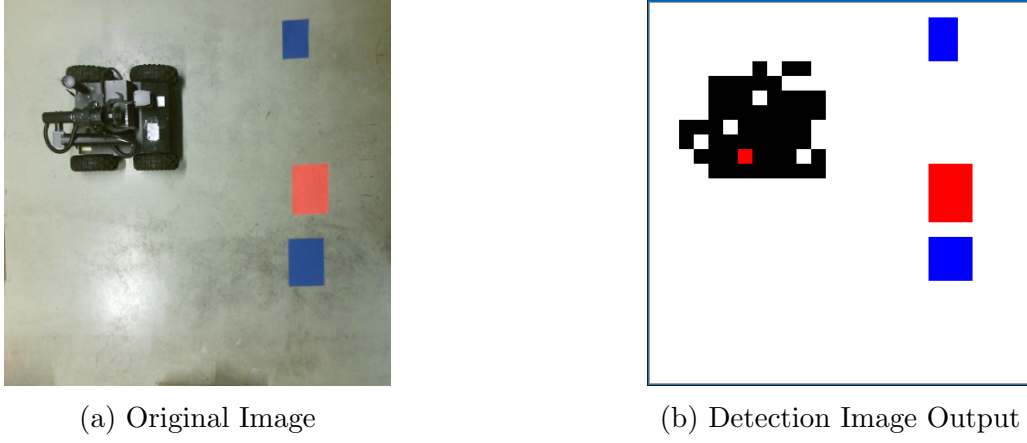


Figure 2.18: Patch Model Results

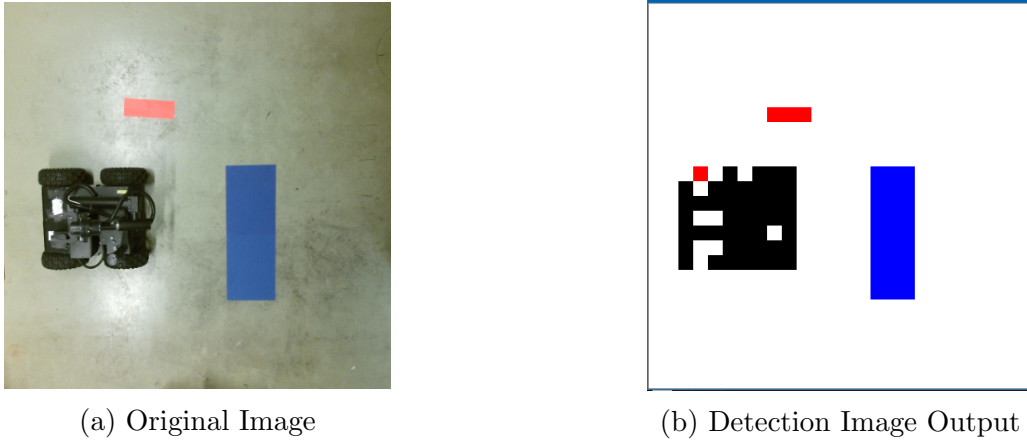


Figure 2.19: Patch Model Results

As opposed to the YOLO model, the **Patch Model** does not provide bounding boxes and, thus, does not provide x_{min} , y_{min} , x_{max} and y_{max} from the start. These positions have to be computed after the prediction for each patch. In order to do this dictionaries and lists are used to store the positions for each object. For getting the positions (x_{min}, y_{min}) and (x_{max}, y_{max}) the first patch that is representative of a label is taken and the last patch that is representative of the same label is also taken. The (x_{min}, y_{min}) of the overall bounding box is the (x_{min}, y_{min}) of the first patch and (x_{max}, y_{max}) of the overall bounding box is the (x_{max}, y_{max}) of the last patch for the represented label.

While building this approach a problem was encountered. If there are just singular representations of each class then the script performs correctly. On the other hand, if there are multiple objects of the same class, the first patch and the last patch will not correspond to the same object. To split the objects into separate lists, the algorithm looks for the next patch on O_y axis, if the next patch represents the same label then it represents the same object, if does not represent the same label then the list for the current object is closed and the list for the next object is opened. In the case of background patches, no lists are created, as there is no need for bounding boxes representing the background.

2.3.3 Results of the Patch Model

With this approach the system can give the same output data as the YOLO model but, it also performs segmentation on the image, trying to achieve an accurate representation of the objects, not in terms of a bounding box but, in terms of the actual shape of the object (e.g. round

object, non-uniform objects like the *Jaguar 4x4 Platform*).

The **Patch Model** still, does not detect the *Jaguar* class perfectly, sometimes mistaking it for the *Red* class or the *Background* class. This happens due to the very low resolution of the images in the dataset making it hard for the network to make a difference between the two classes and the *Jaguar* class. While most of the *Jaguar 4x4 Platform* is of a dark black, depending on how the light comes across it, some parts of it might look close to white due to the to its reflective (metallic) material. Not only this, the *Jaguar 4x4 Platform*, also has white patches on it, making it even harder to make the perfect shape. In most of the tests, a good enough contour can be seen on the final detection image.

The training of the network is very fast due to the low resolution images, but on the second stage of detection, in the prediction stage, the system can be slow. It is slow because not only does it have to do a prediction, it has to do a prediction for each 16×16 patch for the test image, compute the x_{min} , y_{min} , x_{max} , y_{max} positions and recreate the image for the segmented view.

The number of parameters of the **Patch Model** is:

```
=====
Total params: 438,596
Trainable params: 438,596
Non-trainable params: 0
-----
```

As it can be seen the number of parameters is larger than that of the **Modified Tiny YOLO**. The blueprint separable convolution modification was experimented with in the **Patch Model** as well but, with worse results than the standard convolutions.

The overall accuracy of the **Patch Model** is of 98% but, it is clear that there are still a considerable amount of false positives. Some results of the **Patch Model** can be seen in figures 2.18 and 2.19.

In the end the **Patch Model** provides more details about the image that it's testing but, it is slower and prone to errors.

2.4 Advantages and disadvantages

Comparing the two created networks **Modified Tiny YOLO** and **Patch Model**, a few advantages and disadvantages can be seen. Both of them have been used in the pathfinding algorithm, with various degrees of success.

The **Modified Tiny YOLO** model provides a great accuracy in classifying objects as well as providing a correct sized bounding box. As specified in the objective of the thesis, the model is smaller in terms of parameters, it works on a small dataset and it provides enough information for creating a path for the robot in use.

At the moment, this architecture, is trained and tested in a controlled environment. The performance might decrease in a new one. An example would be a t-shirt with blue patches on it; the network identifies those patches as the blue objects given in the dataset. What this means, is that the objects provided in the dataset are not very unique, having just a few major features, the most important feature is of the object being color. With this in mind, if the system is to be applied in a public environment, such as an airport, a bigger more general dataset must be used for training, in order to provide uniqueness to the classes it predicts. On the other hand, if the network is used in a storage facility, there are not many factors that can be out of ordinary. A good example can be a shipping yard, as it can be observed in figure 2.20. This environment is what can be called a controlled environment as opposed to

what can be seen in figure 2.21 which represents a top down view of a market. In a market, when it is empty it can be seen that there a lot of objects that occupy its space, of various sizes and of various shapes, which can be problematic. But a market is not built to be empty (i.e. without customers). In figure 2.21b can be seen a busy market, with many customers shopping. These customers are always moving, blending in with the background and bringing in various other objects that may not be found in the dataset, such a place is very difficult to process correctly by any neural network. On the other hand the shipyard is less cluttered and organized. Most of the containers are of a specific shape or color. In the presented image the robot that may be controlled is the crane that picks up a certain container, or the truck onto which it is loaded. For the crane there are not many obstacles that have to be avoided but, there might be areas where the crane must not place containers, areas which are detected by the network. For the truck, on the other hand, there are obstacles. The simple approach of detecting these obstacles from a top-down perspective and controlling the truck in the same manner can provide a more efficient process. There are already automated processes in modern shipyards, that take advantage of artificial intelligence such as, LBCT - Long Beach Container Terminal [21].

The **Patch Model** provides similar informational output to that of the **Modified Tiny YOLO** model but, can provide better segmentation of the objects and more precise shape. While the **YOLO** architecture creates a bounding box around the object, there are moments where certain edges or corners are out of the borders of the bounding box or considering a tight space the borders can be larger than the object itself and the robot might not consider that path due to it not being wide enough for the robot to pass through. Detecting the position of the robot and then processing the image with the segmentation provided by the **Patch Model** can provide a great precision in regards to what areas are passable or not.

This network as it stands now, does have some major problems. Due to the low resolution of the input data it can misclassify certain patches, which can be very problematic when it classifies a passable spot as non-passable, or worse the other way around leading to a crash. While the precision of the network is vastly improved, the lack of accuracy greatly diminishes its usability.

The other problem that the **Patch Model** encounters is its very low speed. In a fast moving environment, which can be assumed most real environments are, the lack of speed of a network will not detect changes in a scene in time for the system to make adjustments.

In order to apply this type of network a controlled environment is a must, where objects have a clear general uniqueness so that they can be accurately detected, like the red and blue objects are detected in the test images of this project. To provide that great precision either the resolution of the patches has to be lowered or the resolution of the images that model is applied to has to be increased, such that a bigger grid can provide more information. A good lighting has to be provided, with materials that are not prone to reflections. More images will lead to a better generalization of the dataset and can improve the accuracy of the detection.

Which architecture to use?

As the advantages and the disadvantages of the networks were presented, it can be seen that one architecture can be more useful than the other in different cases.

The **Modified Tiny YOLO** model can be used in more general environments, providing a high accuracy and a good precision for bounding boxes and its small number of parameters and speed is useful considering real time control.

The **Patch Model** is good for precision segmentation and can be useful in tight spaces where every centimeter counts and its fast training speed means it can be adapted very easily to the situation at hand.

Both of them suffer from generalization problems, but the **Patch Model** more so than the

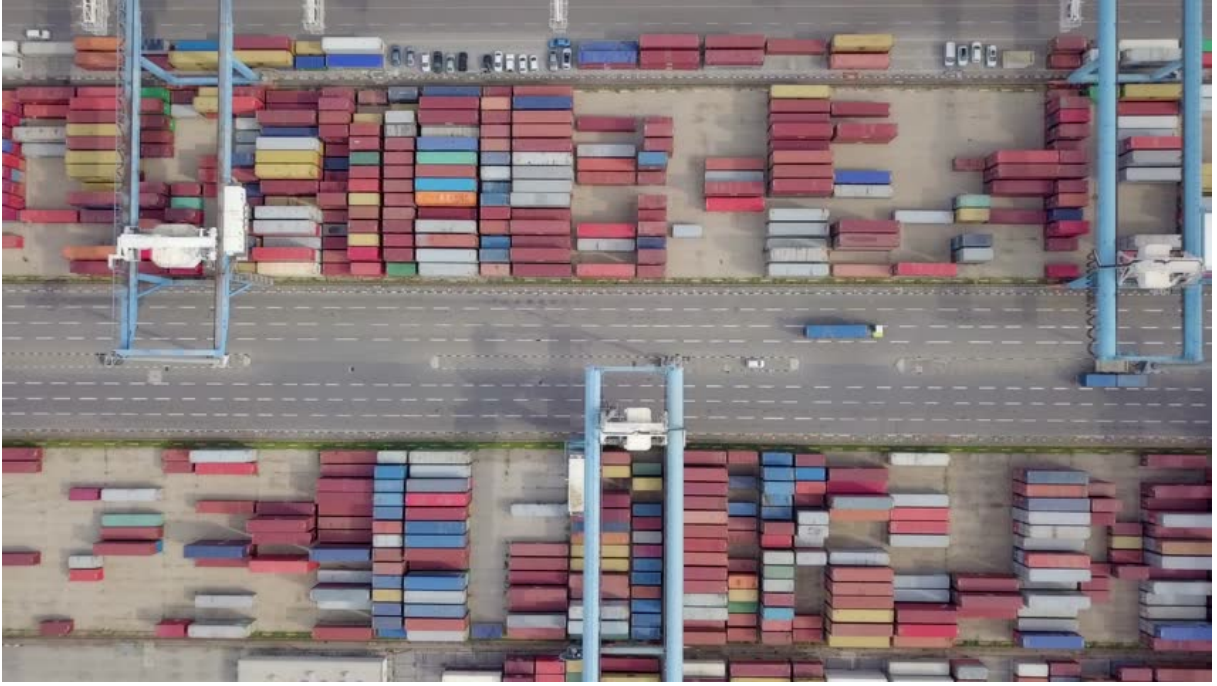
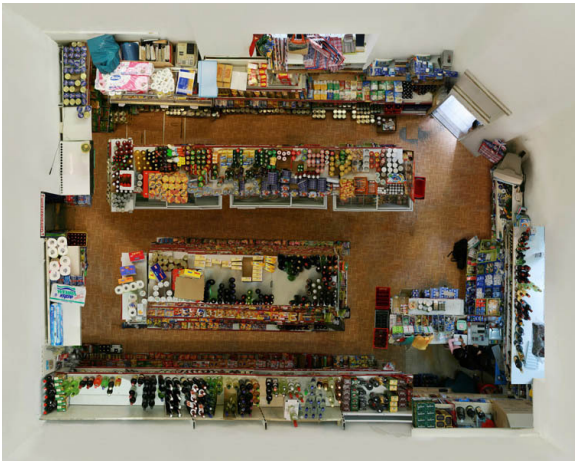


Figure 2.20: Shipping yard [7]



(a) Top down market portrait [22]



(b) Busy market

Figure 2.21: Market images

Modified Tiny YOLO model, with the YOLO architecture having a much better accuracy and detection speed. The Patch Model is also very dependent on the environment and the contrast and uniqueness of the objects in the dataset, so although it is fast to train, it is confined by these constraints to a small number of applications.

It can be concluded that the Patch Model needs to be improved but, it does provide some useful features, while the Modified Tiny YOLO model achieves its objective and can be deployed in different environments.

This is the reason why, moving forward in this paper, the model used for pathfinding is the Modified Tiny YOLO model.

Chapter 3

Pathfinding System

The pathfinding system is the second part of the neural based navigation system. Taking the output of the neural network, it must find the shortest path to the objective given by the user, while avoiding the obstacles along the way. The pathfinding system must be fast, accurate and able to send the correct commands to the 4x4 Jaguar Platform.

3.1 Pathfinding algorithms

When it comes to pathfinding there are multiple algorithms that can be used with different features and approaches. The ones that will be presented in this thesis are the Dijkstra Algorithm and the A* Algorithm [23], [24].

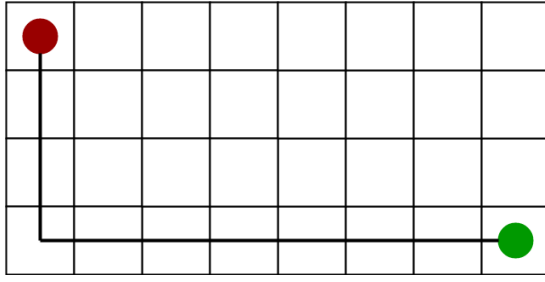
In order to understand these algorithms, the first concept that needs to be explained is *heuristics*. Heuristics are techniques for solving problems in a quick way. These techniques are not the best solution but, the fastest one. They are usually approximations that give a good enough result in order to reach the object in the shortest way.

The heuristics of pathfinding give the cost for following a certain path. Dijkstra algorithm does not use heuristics, whereas A* algorithm does. The A* algorithm can be viewed as a Dijkstra algorithm with heuristics.

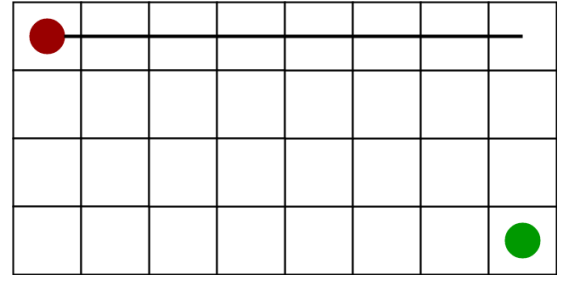
Both algorithms use a cost of movement by which they find the shortest path. Dijkstra has a cost of movement from the current cell to the very next one, while A* has a cost $f = g + h$ where g is the cost from the current cell to the next one and h is the cost from the next cell to the final destination. In this case h is the approximation or the heuristic because the algorithm does not know yet the shortest path to the objective, since there might be obstacles in the way (e.g. walls, humans or others).

The Dijkstra algorithm has two lists one that includes the cells use for the shortest path to the objective (generally called the closed list) and one that has the other neighbouring cells with their costs (generally called the open list). The initial cell is initialized with a cost of zero, while all the other initialized with infinity. As the algorithm goes through the cells it checks the neighbouring cells and adds them to the open list with their representative costs, if a shorter path to a cell is found the cost for that cell is updated or the other way around if a cell is in the open list already with a lower cost the current one is skipped. With the next iteration, starting from the cell that had the lowest cost the algorithm adds the next cells in the path with their costs and so on until they reach the end goal where the algorithm stops and then it backtracks the steps it has taken to get there.

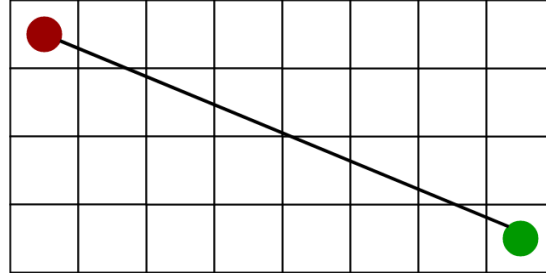
The A* algorithm has the same approach but the cost of each cell is seen differently. A*



(a) Manhattan Distance [24]



(b) Diagonal Distance [24]



(c) Euclidean Distance [24]

Figure 3.1: Types of Distances

can also be seen as a Dijkstra algorithm with direction. Detailing the h term a few types of heuristics can be seen such as:

Manhattan distance which is the sum of the absolute differences between the current cell's x and y values and the objective's x and y values:

$$h = |c_x - o_x| + |c_y - o_y| \quad (3.1)$$

Diagonal distance is the maximum between the Manhattan distance's differences:

$$h = \max(|c_x - o_x|, |c_y - o_y|) \quad (3.2)$$

Euclidean distance is the direct distance from the current cell to the objective using the standard distance formula:

$$h = \sqrt{|c_x - o_x|^2 + |c_y - o_y|^2} \quad (3.3)$$

A graphical representation of them can be found in figure 3.1.

The A* algorithm works the following way. It has two points, a start point and an end point. These points have no parent cell. Two lists are created an open list that at the beginning has only the start cell and a closed cell which initially is empty. The algorithm checks the open list for the cell with lowest cost (in this case f). The one with lowest cost is popped out of the open list and appended into the closed list. From this cell, children cells are then created in all 8 directions. If any of these cells are out of bounds or an obstacle, they are ignored. If they are clear cells then they are created as new objects with the current cell as parent (in this initial case the starting cell). All of these children are then appended into a children list which will be iterated with a `for` loop in order to first check if they are in the open or closed list already, in which case they will be skipped, if they are not, the cost f will be computed using the one of the heuristics presented (the Manhattan distance is used in this project) and they will be appended to the open list. The cycle repeats this time by checking the open list for the child

with the lowest cost, popping it out of the list and appending it to the closed list, this child then becoming the current cell from which the algorithm moves further into the other new 8 direction until the end cell is reached. If the end cell is reached, a new list will be created, the path list, in which all the cells that lead to it, will be added by going from parent to parent ending with the starting cell and finally returning the path in a reversed list.

Between the two algorithms it is clear that the A* algorithm is faster because it has a direction as opposed to the Dijkstra algorithm which can take unnecessary paths just because they have a low cost but not knowing that they may lead to a completely different direction than the objective. Thus, the algorithm that is used for the pathfinding system is the A* algorithm.

3.2 Helpful functions

In order for the A* pathfinding to create the correct path to follow, for the 4x4 Jaguar Platform, two helpful functions were created. The functions are named *Jaguar Direction* and *White Objects*.

Jaguar Direction

The 4x4 Jaguar Platform needs to be facing the start of the path, meaning that even though the found path commands the robot to move left, the robot may be facing right, thinking that it is facing in the correct direction it will move forward but, instead of moving towards the objective it is moving in the opposite way. In the initial form of the system the starting point of the pathfinding algorithm was the center point of the robot but, in doing so, the above problem was encountered. The direction is important because in order for the robot to have the shortest path to the objective it is better for it to be facing that objective directly. Suppose the following scenario. The robot is facing right and the objective is at the bottom of the image. Normally, the pathfinding algorithm would start creating a path from the front of the robot which is facing right, meaning extra pixels will have to be identified for the path. By finding the direction of the robot and then facing it towards the objective the system will get a correct starting point and path, and a shorter one, as well. Facing the objective fixes the problem of moving in the opposite way from the objective. Even if the direction of the robot is found and starting point is set at the front of it, the path would go through the robot towards the objective but, the robot will not, so, facing the objective is both shorter and it also solves the problem of moving away from the objective.

In order to create this script a cropped image of the robot had to be created. With the help of the output of the neural network in the form of an XML file the positions of the bounding box are known. The bounding box cropped out of the original image and then a few algorithms are used in order to detect the front of the robot, which way is it facing and how it should be turned towards the objective if needed.

For detecting the front of the robot, a few unique features of the the robot were used. The robot has a white patch on its front, so this patch was used to detect the position in the image's matrix. The cropped imaged was split in 4 sections as seen in figure 3.2. The algorithm checks if the white patch is in any of these sections. By detecting in which section the patch is, the algorithm has a rough estimation of where the front might be. To detect the correct section, first it checks if there are any white spots either in the left part of the robot or the right part of the robot. Then it checks if there are any white spots on the top or on the bottom of the robot. If there are white spots in both left and right parts the left and right variables are evaluated as false. If white is detected only in one direction, that one is validated as true and the other as false. The same steps are applied to the top and down sections. The white patch can not be in both opposite directions, thus setting them as false leads to less parameters to deal with, meaning that the algorithm will know from the start the final direction without having to choose if its

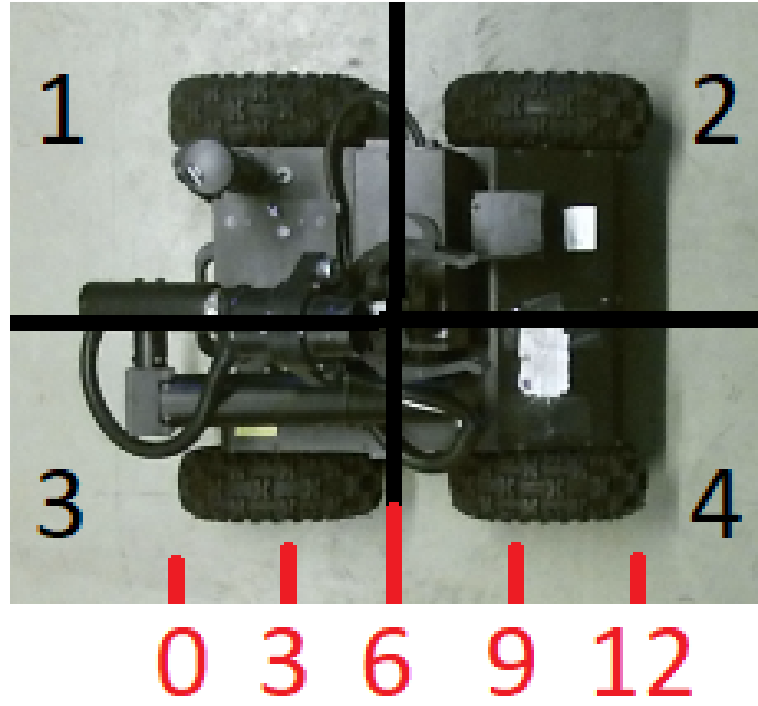


Figure 3.2: Direction Detector

direction is vertical or horizontal. If there are still two directions set at true, – i.e. in order to check if it is down or right, the connected components are mapped on the cropped image. The x position of the white pixel is subtracted from half the length of the cropped image, giving the algorithm the information of how close to the edge or the middle of the image the pixel is. If the absolute value of this difference is bigger than a quarter of the length of the robot the left or right variable is set as false else the top or down variable is set as false, leaving just one direction as the true direction. The difference works because the split in half and then in quarters makes a symmetrical representation. For more clarity if half the length is 6, then we can see all 4 quarters as being in the intervals: 0 – 3, 3 – 6, 6 – 9 and 9 – 12. If the point is of value 8 and it is subtracted from 6 it would give -2 , that as absolute value is 2. In order to get 2, the value 4 can also be subtracted from 6. The value of 2 is closer to the edge – i.e. lower than the quarter length, meaning that the white patch is closer to the middle. This invalidates the horizontal direction, leaving just the vertical one. The reverse is true, as well. After identifying the direction, the representative position of x and y is also given (i.e. not as the position of the pixel but as the maximum center position of either direction).

After some testing while, this approach works, if the white patch is not visible enough, the result might be wrong or it might not find the front even if it gives a wrong direction. Another approach was experimented with, one in which the robot is still split in 4 section but, this time these sections are triangles, each triangle representing now the top, bottom, right and left of the image as if that triangle would be the front of the robot. In order to reduce the number of checks if the width of the robot is bigger than its height, then only left and right are checked and top and bottom are checked if the reverse is valid. While this approach is simpler, the results are similar. In order to be sure that the system calculates the right result the image has to be uniformly lit and the white patch has to be clearly visible.

After the direction the robot is facing is found the part of the system where this direction is compared to the position of the objective begins.

First, two subtractions are computed in order to get the sign for O_x and O_y axis. After getting the signs, depending on whether these signs are positive or negative, the algorithm can tell in which direction the objective is compared to the front of the robot. These directions are

noted as: SE, NW, SW, NE, S, N, W and E.

The robot being a rectangle, changing its orientation might not give to correct position of the new front. In order to orient the robot towards the objective the center of the robot is calculate as variables x_c and y_c . From x_c and y_c two more values are calculated named r_x and l_x , representative of either the value on the O_x axis when the robot needs to be facing left or right, or the value on the O_y axis for when the robot needs to be facing top or bottom. In this algorithm the r_x and l_x values are used only for the O_x axis since it is sufficient. These values offer three sections on the O_x axis – e.g. smaller than 0, interval 0 – 12 and bigger than 12, in order to know in which section the robot finds itself, another two values are computed, these values are represented by the differences: $dr_x = r_x - objective_x$ and $dl_x = l_x - objective_x$; where $objective_x$ is the position on the O_x axis for the objective. Depending on the signs of the results, if dr_x is positive and dl_x is negative the final orientation of the robot is given by the orientation given in the last paragraph. If dr_x is negative the final direction is right, if dl_x is positive, the final direction is left. At the end, the function returns the final direction, the position of the start dot for the pathfinding algorithm and the angle by which the robot must turn in order for the front of it be at the position specified by the dot.

White Objects

The pathfinding algorithm will work on binary images. So, for the sake of simplicity, all the obstacles that are marked as free to pass (i. e. the blue ones) will be also painted black.

The **White Objects** function addresses one more problem. The path is painted starting from the center front point of the robot, thus, the robot moves only on this line. The problem here is even if the robot does "avoid" the obstacle, it does not do it correctly because it might be possible that half of it will still hit the obstacle. Another problem that stems from this one is that even though the algorithm may find a path through two obstacles the width of this path might not be large enough for the robot to pass through. There are multiple methods to address this problem, such as, clearance-based A*, where another parameter is added to the cost function, the clearance as specified in its title. Depending on the how close to an obstacle a cell is, it will have a certain clearance cost that will be added to the overall cost. While this approach works, the algorithm is much slower.

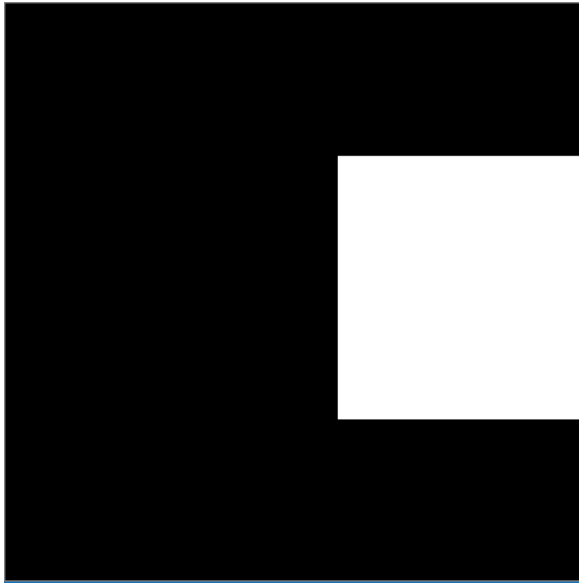
A simpler approach is implemented. The area of the white space that is painted on image is enlarged by half of the length of the robot in each direction as seen in figure 3.3a. This way the robot will not step on the obstacle and will not encounter a situation where the path is not wide enough for it to pass through.

The final path with the avoidance area created by the **White Objects** function can be seen in figure 3.3b.

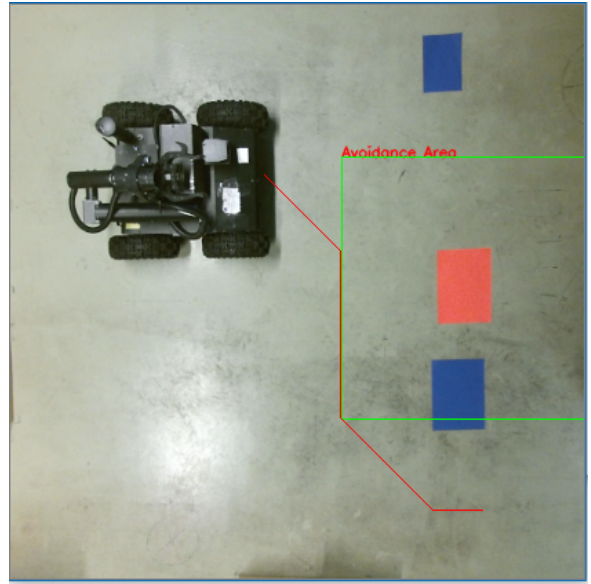
3.3 Robot commands

After the path has been created, it has to be transmitted to the robot to perform the necessary moves to arrive at the objective. A new script was created for this called **Jaguar Move**. This script makes use of the **socket** library and the program **Jaguar Base Controller**, written in C#, used in [10] for transmitting the correct format of the command and for the motors to have a good enough speed in order for the robot to not crash or not to be too slow.

With the help of the script and the **Jaguar Base Controller** the system connects to the robot's router and transmits a series of commands. In order for these commands to be transmitted they would have to be created from the path that was given by the A* algorithm, which give a list of positions. The script sees the direction in which the robot has to move for a certain portion of the path. For example it has to move left, then for as long as the y position on O_y axis does not change, then the robot, because it is facing that direction, moves



(a) White Objects Binarization



(b) Final Path

Figure 3.3: Path Creation

forward for how many steps there are until the y position changes. The same idea can be applied if the robot is moving towards the bottom of the image, only this time the change in direction is given by a change in x of the O_x axis. All these steps are saved in a list and with the help of this list, three more lists are created (directions, distances, angles) that will provide the parameters for the robot commands. The steps saved in the list are saved as directions: top, bottom, right, left, bottom-right, top-left, top-right, bottom-left. For each step that is the same as the previous one, the number of steps is increased. For each step that is different than the previous one, the previous direction is appended to the directions list and the angle for the turn is appended to the angles list. If the direction is not on a diagonal path, then the distance is appended by multiplying the number of steps with a constant that was deduced by measuring the width or height (since they are equal) of the area that was captured by the Kinect and dividing by the length of image (the number of pixels on one axis), giving the value in centimeters for 1 pixel, thus giving the value of the constant, of 0.85 cm. In order to get the constant for a diagonal move this value was multiplied by $\sqrt{2}$ giving 1.2 cm. These values were lower to 0.5 and 1 since the robot would not be able to break well enough in order to stop in the exact spot given by the algorithm. Of course, if the direction is changed, the number of steps would be reinitialized to 0. This part is done in a **for** loop and in order to get the last command parameters this part of the script is done one more time separately.

An example of the parameters for final path in figure 3.3b:

```
Directions: ['bottom-right', 'bottom', 'bottom-right', 'right']
Angles: [45, 90, 45, 0]
Distances: [27, 60, 33, 13]
```

After the parameter lists are filled, a **for** loop for constructing commands is created. The commands have the following format - "**sendCommand(angle, distance)**". The distance remains the same as it is given in the list but, the angles need to be processed further. As they are in the initial format of the list, the angles represent the direction in which the robot needs to be facing. The robot moves by the amount of degrees it has received. Thus, every turn is represented in angles by the difference between the current angle and previous one. While connected to the robot with the help of the **Jaguar Base Controller** program, these commands are sent one by one, with the final command "**endCommand**" that ends the connection.

Chapter 4

Conclusions

There are many approaches of creating autonomous robots and the applications are vast. In order for these robots to be more accessible, less expensive systems need to be created. Having the possibility of implementing an autonomous system at home, will open the market to new opportunities and applications.

Objectives Review

The main product of this thesis, is a neural-based navigation system with a top-down view. This product started with the following objectives in mind:

- to create a data set for the neural network
- to train different Convolutional Neural Networks architectures for the obstacle detection
- to implement a pathfinding algorithm
- to control the 4x4 platform according to the algorithm

The motivation behind them was that a system such as this could provide autonomous control with less resources. While researching the desire for such a system, a few criteria were set along the way, such as:

- lower number of sensors
- smaller computational cost
- similar results to autonomous obstacle avoiding systems

To do this, sensors had to be used in a new way, that's why there is a top-down view. In order to have small computational cost, it lead to modifications to some of the networks or algorithms, and all of these had to work together in such a way, that the limitations, imposed by the made modifications, would not significantly affect the obstacle avoiding system quality.

Personal Contributions

The navigation system as a whole achieves its initial objectives and overall purpose.

Thus, a dataset consisting of 100 images has been created, with the help of the Kinect camera and the IBM Annotation Tool for the YOLO architecture. The dataset consists of 3 classes: Red, Blue and Jaguar. Another dataset was created for the **Patch Model** with 4 classes (with the added Background class). The dataset for the **Patch Model** was created from the original dataset by creating 3 new scripts, used for: recreating the annotations, processing the images based on the new annotations and encoding these images in a format recognized by the neural network.

These two dataset were fed into two neural networks. One network that is based on the **Tiny YOLO** architecture, where the contribution can be seen in the implementation of the **Blueprint Separable Convolutions** in order to reduce the number of parameters and to increase accuracy. The other network was created from scratch, together with the detection system, specifically for the approach of patch recreation.

A pathfinding system inspired by the A* algorithm was implemented. This includes the A* algorithm itself, with a focus on speed in mind. This algorithm was fitted with some helpful function, for detecting the robots front facing direction and in order to rotate it to face the direction of the objective. Another functionality added to the algorithm consist in making sure the robot fits between two obstacles, or that it has enough clearance to go around one. The system is completed by the script that translates the created path into commands that the robot can understand and perform. The final piece is the link with movement system, that is separated from the pathfinding system. The movement system was developed by Eng. Mihai-Cristian Tudoroiu in his undergraduate thesis [10].

Navigation System Results

The presented system manages to lower both the number of images in the dataset and the number of parameters needed for training a network, while increasing the accuracy of the system for the **Tiny YOLO** architecture. It can be seen that there are many tools to improve networks performances and to make them accessible to low end devices. Also, it was shown that different types of approaches can work, and can provide new features, such as the **Patch Model**. This model can be improved further to reduce errors and accelerate the detection process. By using the patch model with the improvements done to the YOLO model, it was seen that the tools used to improve that architecture are not generally valid for every architecture, especially when dealing with images of a very low resolution, from which not many features can be extracted.

The system also manages to lower the number of sensors needed for driving the robot to its destination, but some problems still remain – e.g. the robot has to always be in the field of view of the camera and the camera vision of the robot should not be obstructed by any elements on the map. Although these problems occur, if the automation is done in a controlled environment, this method can significantly simplify a system that makes use of autonomous robots. The device used for capturing images was an Xbox Kinect, but any camera can be used as long as it can send the data to the network, such as a mobile phone.

Future Developments

The system can have various improvements, starting with the main problems and ending with adding new features.

Beginning with the **Modified Tiny YOLO**, the architecture could be ported to **TensorFlow Lite**. In doing so the network can be implemented on mobile devices. Linked with the a mobile application can be built, in order to place the destination of the robot using the touch screen display. In this application multiple analysis features can be implemented (i.e. battery voltage, speed or others). A type of this application already exists, created by the Dr Robot team for the **4X4 Jaguar Platform**. If the application can be modified, the network can be implemented as one of its features. Atop of porting the network to **TensorFlow Lite**, it can also be modified further to test if it can provide better inference on the mobile devices, or low end devices.

Considering the **Patch Model**, as it was stated both in its description and in chapter 4 it needs further development in order to lower its error for classification. If this error can be lowered this type of network can provide more information to the system and make it more robust. Making the patch model, into an end-to-end architecture, similar to **YOLO**, or modifying the **YOLO** architecture to perform the task of the **Patch Model**.

For the pathfinding system, the algorithm can be improved in order to provide a faster response, trying to achieve a real-time movement of the robot as opposed for having a buffer time to calculate the route, albeit a small one. Different heuristics can be tested, such as a clearance cost to get a better precision when avoiding obstacles.

Taking into account the top-down view and the fix camera, a mobile camera can taken into consideration, such as a drone companion, but this type of companion would still have to be controlled in some manner, in order to fly safely and not crash itself in different obstacle that might appear in the area (e.g. a tree, a building or other). If the air zone is mostly clear, the drone companion could follow the robot, acting as the robot's eyes. In order for the robot to reach a location outside the area of its vision, geographical coordinates can be given to it acting as the direction of the objective, while it continuously creates a path ahead of itself avoiding obstacles.

As the **4X4 Jaguar Platform** has also other sensors equipped to it, as well as a robotic arm, different scripts can be created for them to add more functionality to the system. With the help of these developments this neural based navigation system, could be a better autonomous system, with still some limitations to consider but, with much vaster functionality if set in a different environment, not needing a controlled environment anymore.

All software resources of the project are available online on a public **GitHub** ¹ repository.

¹<https://git.speed.pub.ro/costeamadalinalalexandru/neural-based-navigation-system-for-4x4-jaguar-platform>

References

- [1] Jeff Hale. Deep learning frameworks ranking computed based on 11 data sources across 7 categories, 2018.
- [2] missinglink.ai. Fully connected layers in convolutional neural networks: The complete guide.
- [3] Vinicius C. Costa. Understanding the structure of a cnn, 2019.
- [4] Geeks for Geeks. Underfitting and overfitting in machine learning.
- [5] Andrew Ng, Kian Katanforoosh, and Younes Bensouda. Deep learning specialization, 2017.
- [6] Daniel Haase and Manuel Amthor. Rethinking depthwise separable convolutions:how intra-kernel correlations lead to improved mobilenets, 2020.
- [7] StockStudio Aerials. Shutterstock shipyard.
- [8] Dr Robot. Jaguar 4x4 wheel - user guide, 2018.
- [9] Microsoft. Kinect for windows.
- [10] Tudoroiu Mihai-Cristian. Sistem de evitare automată a obstacolelor cu robotul jaguar 4x4 pe baza procesării unor secvențe video, 2019.
- [11] vladkol. Pykinect2, 2016.
- [12] OpenCV Team. Opencv 4.1.1 documentation, 2019.
- [13] The microsoft cognitive toolkit documentation, 2017.
- [14] Martín Abadi, Ashish Agarwal, Paul Barham, and et. al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [15] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization, 2017.
- [16] Diganta Misra. Mish: A self regularized non-monotonic neural activation function, 2019.
- [17] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Swish: A self-gated activation function, 2017.

- [18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.
- [19] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.
- [20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.
- [21] Long Beach Container Terminal (LBCT LLC.). Long beach container terminal, 2016.
- [22] Menno Aden. Room portraits.
- [23] Geeks for Geeks. Dijkstra’s shortest path algorithm.
- [24] Geeks for Geeks. A* search algorithm.