Norwegian University of
Science and Technology

Computer Vision and Deep Learning in Autonomous Drones

Markus Teigen Pike

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Frank Lindseth, IDI

Norwegian University of Science and Technology
Department of Computer Science

Norwegian University of
Science and Technology

TDT4900 - Computer Science, Master Thesis

Computer Vision and Deep Learning in Autonomous Drones

Markus Teigen Pike

June 2016

MASTER THESIS
Department of Computer and Information Science
Norwegian University of Science and Technology

Supervisor: Professor Frank Lindseth

Abstract

In this thesis we want to create a deep learning based object detection solution that is able to run locally on an autonomous drone. The goal of the drone is to herd a group of ground robots correctly within the International Aerial Robotics Competition (IARC). The main problem this thesis want to address is how to reliably detect these ground robots through cameras mounted on the drone. It all has to be performed in real time and in four different directions at the same time with limited computational power. The proposed solution is to first create a large dataset of detection examples using images recorded at the IARC 2016 competition and second to use the dataset to train the fastest and most accurate detection neural networks available like YOLO (You Only Look Once) and SSD (Single Shot Multibox Detection). The final result is a dataset of 6100 detection images with labels for each of the three different types of robot in the competition as well as several detection networks that are able to run in real-time onboard the drone with varying degree of accuracy.

Contents

1 Introduction 1
1.1 Background and motivation . . . 1
1.2 Goals and research questions . . . 2
1.3 Thesis structure . . . 3

2 Background 4
2.1 Ascend NTNU . . . 4
2.1.1 International Aerial Robotics Competition (IARC) . . . 4
2.1.2 Why Ascend NTNU is relevant . . . 5
2.1.3 Hardware configuration . . . 6
2.1.4 Topic definition within Ascend NTNU . . . 7
2.1.5 Existing image data . . . 8
2.2 Deep learning models . . . 9
2.2.1 Convolutional Neural Networks (CNNs) . . . 9
2.2.2 AlexNet . . . 10
2.2.3 Googles Inception model . . . 11
2.2.4 VGGNet . . . 12
2.2.5 ResNet . . . 12
2.2.6 Comparison of classification networks . . . 13
2.3 Object Detection Networks . . . 13
2.3.1 Region-based Convolutional Neural Networks (R-CNN) . . . 15
2.3.2 YOLO - You Only Look Once . . . 16
2.3.3 SSD - Single Shot Multibox Detection . . . 18
2.3.4 YOLOv2 . . . 18
2.3.5 Comparison of detection networks . . . 20
2.3.6 Dlib and Max-Margin Object-Detection (MMOD) . . . 20
2.4 Semantic segmentation . . . 21
2.4.1 Sliding window . . . 22
2.4.2 Early stage of CNN image segmentation . . . 22
2.4.3 FCNs - Fully Convolutional Networks . . . 23
2.4.4 Deconvolutional networks for semantic segmentation . . . 24
2.4.5 SegNet . . . 26
2.4.6 DeepLab . . . 27
2.4.7 Comparing segmentation networks . . . 28
2.5 Speed-up Techniques . . . 29
2.5.1 Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks . . . 29

2.5.2 Deep Feature Flow for Video Recognition 29
2.6 Recurrent neural networks 30

3 Methods and implementation **32**
3.1 Detection network methods 32
3.1.1 R-CNN based detection networks 32
3.1.2 YOLO 33
3.1.3 Fast segmentation network 33
3.1.4 SSD 33
3.1.5 YOLOv2 34
3.1.6 Deep feature flow extension 34
3.1.7 MMOD and dlib 35
3.1.8 Conclusion 35
3.2 Detection dataset methods 35
3.2.1 Template matching 36
3.2.2 Background subtraction 37
3.2.3 Sliding window segmentation 37
3.2.4 Location extraction from segmentation images 38
3.2.5 MMOD and dlib 41
3.2.6 Conclusion 41
3.3 Detection dataset implementation 41
3.3.1 Robot segmentation 41
3.3.2 Robot localization 48
3.4 Detection network implementation 58
3.4.1 Network training 58
3.4.2 Adding color 59
3.4.3 Network testing 59

4 Results **60**
4.1 Detection database 60
4.1.1 Robot segmentation 60
4.1.2 Localize robots 61
4.1.3 The final detection dataset 64
4.2 Detection network 67
4.2.1 Network training 67
4.2.2 Network testing 69

5 Discussion **77**
5.1 Detection dataset 77
5.1.1 Robot segmentation 77
5.1.2 Robot localization 79
5.2 Detection network 82
5.2.1 Network training 82
5.2.2 Network testing 83
5.3 Methods that should have been considered 84
5.3.1 dlib 85
5.3.2 Non-maximum suppression 85

6 Conclusion and future work 86
6.1 Conclusion . . . 86
6.1.1 Detection dataset . . . 86
6.1.2 Detection network . . . 86
6.2 Future work . . . 87
6.2.1 Tracking . . . 87
6.2.2 Deep feature flow . . . 87
6.2.3 A more streamline way of training the detection part of SSD . . 87
6.2.4 Trimming the detection edges . . . 88

Chapter 1

Introduction

1.1 Background and motivation

Over the last few years there has been great technological advances in the field of parallel processing and the development of powerful graphical processing units (GPUs). Processing power has been increasing exponentially while the required power consumption has stayed the same[1]. Many of these units are intended for the user market to enable high resolution gaming experiences, which means that high performance parallel computing units can be bought by anyone for a low price. Even though they are somewhat geared towards optimizing graphical computations they are still general enough to be used in a wide array of other parallelizable tasks. This has had a great impact on the research field of deep learning. It is now possible for anyone to apply advanced deep learning techniques to their work in regular labs or even at home.

Deep learning is a type of machine learning that has proven useful in a range of tasks that are considered simple for humans but very hard for computers to perform. Some examples are natural language processing and image analysis like object classification, object detection and segmentation. This opens up to a range of possibilities in the pursuit of creating autonomous systems. Previous attempts have mainly used a combination of simple sensory input like laser scanners and radar to understand the state of the world around it. However, it can be difficult to get a complete and accurate state estimation without a more thorough way of sensing. A more robust solution would be able to make sense of imagery input from cameras. A 360 degree live video stream would provide a human pilot with nearly all the required information to properly control most vehicles. Recent discoveries in deep learning has been able to replicate the pattern recognition abilities of humans to a certain degree and allowed for a more general and robust way of processing images. A software system capable of understanding the world around it using live images would be able to give a much better estimate of the world than a purely sensory based solution would. This idea is finally realizable through deep learning and the availability of powerful GPUs which has sparked a revolution in the development of autonomous vehicles.

An intriguing domain for autonomous research can be found in the air. Different types of multicopters, commonly referred to as “drones”, has especially proven sus-

ceptible to autonomous control because of the unrestricted environment they operate in. Compared to larger vehicles like cars we see that drones are much smaller, far less dangerous and there are fewer laws restricting their use in public. Drones have therefore far fewer requirements to fulfill and will only need a fraction of the computational power required by a car to drive safely by itself. As a result of the advances in GPU technology and deep learning there even exists credit card sized embedded systems that are optimized for deep learning techniques. These can be installed on the drone and work as an onboard computer to improve its computational capabilities.

Research within the field of autonomous drones exploded when radio controlled versions became popular consumer products in 2010. Since then several adaptations with autonomous flight capabilities have been released mainly intended for capturing aerial video shots. However there also exist purely academical endeavors at achieving autonomous flight like the International Aerial Robotics Competition (IARC)[7]. Drones in the competition require reliable state estimation of different types of ground robots moving within a 20x20m arena. This will in turn be used to navigate the arena accordingly.

1.2 Goals and research questions

The aim of this thesis is to research in which ways deep learning and computer vision can be used to extract robot position data from raw live images taken onboard a drone to enable it to fly autonomously within the IARC competition. The processes that will be considered to obtain localization data from raw images are several newly developed methods for object detection and localization in images. The top performers on these types of problems are based in deep learning and the first step in this thesis will be to research state of the art deep neural networks approaches that enable efficient and accurate object detection. The main effort of setting up and using a customized detection network nowadays is the creation of a detection dataset. As a result of this we will focus on the acquisition of reliable training data before creating the localization network itself. So although this thesis will focus on a deep learning approach to object detection, it will still involve a large amount of more traditional computer vision techniques to pave the way for a deep learning solution to perform at full capacity. When the required training set is completed we will shift focus to testing the different deep learning approaches to find the one with the best combination of speed and accuracy for an onboard detection system to be used within the IARC competition. Hence we want to answer the following research questions:

- First, how can reliable and general training data be created for a deep learning based object detection system without any prior information about position or size of the objects?
- Second, how can object detection be done reliably in real-time on an embedded system with limited computational potential?

1.3 Thesis structure

This thesis is structured as follows. This introductory chapter presents a high level description of the problem we want to solve and explains why this topic is relevant today. The second chapter will first define the concrete goal we want to achieve through this thesis and how it directly relates to our research goals. Secondly it will look at different techniques within computer vision and deep learning that could be adopted as parts of the final solution. The third chapter considers how theory in chapter two will benefit the solution and how it will be implemented and used in practice. Chapter four will reveal the results, while chapter five will be used to discuss the outcome more in detail. Chapter six will conclude the thesis and discuss which methods and implementation strategies should be emphasized in future work as well as what could have been done differently to improve the final results of this thesis.

Chapter 2

Background

The background portion of this thesis will cover a variety of deep learning topics. The amount of information that is available on image analysis using deep learning has grown immensely the last couple of years and it shows no sign of stopping. This means that there are no single book that can be read to learn all that is needed, but rather a large number of scientific papers that must be sifted through to find the ones that are relevant. This process is time consuming and comprehensive which means that quite a large potion of this thesis will be focused on unveiling and discussing state of the art deep learning approaches. A large number of topics were examined but many of them turned out to be fruitless in relation to the research goals. In this chapter we will go through the relevant topics and assess them to find out which strengths and weaknesses they possess.

2.1 Ascend NTNU

2.1.1 International Aerial Robotics Competition (IARC)

Ascend NTNU is a non-profit student organization created to represent NTNU in the International Aerial Robotics Competition[7]. The organization was founded in 2015 and participated in the competition for the first time in 2016. IARC is an international contest that has been ongoing since 1994, and its goal is to push the technology of aerial robotics forward. It challenges the contestants to solve problems that are considered hard or impossible at the time, and whenever someone eventually solves it a new and harder problem is introduced. The competition is currently on its seventh mission which challenges contestants to create a fully autonomous drone with the ability to herd a set of ground robots in a specific direction.

The tournament is held indoors on a 20x20m white grid with one boundary edge colored green and the opposite boundary edge colored red as seen in Figure 2.1. The two remaining boundary edges have a wider white line than the rest of the grid. There are 10 ground robots with a diameter of 34cm that all spread out from a given starting position and navigate the grid autonomously and semi-randomly. The robots have a

simple forward motion but will reverse direction every 20 seconds and have up to 20 degrees of angle noise applied to their trajectories at 5 second intervals. There is a switch on top of each robot which can be activated by landing on it which results in a 45 degrees clockwise turn. Another option is to land in front of it and block its path, in which it will turn 180 degrees. There are also 4 extra ground robots with 2 meter vertical tubes mounted to them that must be avoided when navigating the airspace. Whenever a ground robot reaches the edge of the arena it will be permanently removed from the competition. The current mission is to guide as many ground robots as possible to the green edge within a time frame of 10 minutes or until all the robots have exited the arena.

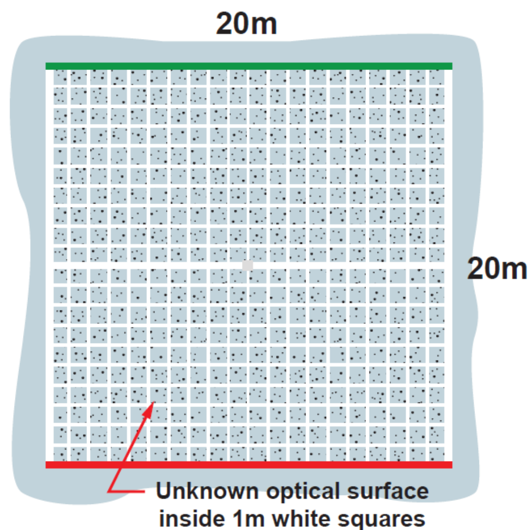


Figure 2.1: The IARC arena. The grid is 20x20 meters where each cell is 1x1m. The grid lines are all white and the same width except for the boundary edges which are all thicker than the rest. Two of the opposing boundary lines are colored red and green.

2.1.2 Why Ascend NTNU is relevant

This thesis is generally interested in exploring the world of autonomous drones and how deep learning techniques can be applied to create robust and time efficient solutions for autonomous flight. To enable any sort of practical testing within this field it is essential to have an actual test drone available. There exists several good options from the consumer market that have pre-installed features that make them easy to control and navigate, even for people who are completely new to drones. The problem is that hardly any of these are open source, and even though most of them have a camera pointing forward there is usually no simple way to attach extra cameras, sensors or additional computational power. With this in mind we consider three approaches to acquire a fully functional test drone:

1. Build a drone from scratch using parts or a kit from a hobby shop.

2. Buy a ready-to-fly drone and attach extra computing power and sensors/cameras.
3. Join Ascend NTNUs perception team which solely focus on the visual computing task of autonomous flight while leaving the complex work of building and maintaining a drone to the designated hardware team.

First, let's consider the option of building a drone from simple parts or a kit. The greatest concern with this approach is the difficulty of the build process without any previous experience with drones. It is easy to make small but expensive (in both money and time) mistakes that require new parts which halts the desired research process. The same goes for a solution where extra computing power and sensors/cameras is attached to an existing drone. This setup should be simpler, but mistakes within this process could be substantially more expensive. This solution will also require the drone to have a flight controller with an open source API which only a selection of ready-to-fly drones have.

For multiple reasons the far and away most appealing solution is to join the Ascend NTNU team and set the goal of this paper to improve their existing autonomous system using computer vision and deep learning. First of all, their approach to autonomous flight using only image and sensory inputs is very relevant for this thesis. Second, the level of resources, knowledge and support this master thesis would receive from a collaboration with Ascend is highly desirable and would significantly decrease the time it would take to acquire a functional test unit as well as maintaining it.

2.1.3 Hardware configuration

Ascend NTNU has already competed once in this competition and therefore have a fully functional semi-autonomous drone. They even have a designated group of people that maintains and improves it constantly. The drone is build from scratch and uses a carbon fiber case and four detachable boom arms. Every component has been joined together neatly in a way that would hardly be possible for a first time drone builder. Some of the more exposed parts around the individual motors are custom made and 3D-printed which means they can easily be replaced if they are damaged. At its center there is a Pixhawk flight controller and four Nvidia Jetson TX2 cards as its onboard processing power. This requires a customized carrier board to get the necessary connection interface for all the units in a small and lightweight package. The flight controller has an IMU (Inertial Measurement Unit) which measures the angle of the drone in all directions. The input devices used is a 360 degree LIDAR (Light Detection And Ranging) that scans the arena horizontally for obstacles, one fisheye camera pointing straight down, and four cameras pointing forward, backwards, left and right. There is also a sideline computer that can be used for heavy computational tasks, but the delay of transferring data back and forth makes onboard computing much more responsive and is usually preferred. The drone will however constantly stream images and sensor data to the sideline computer so they can be used to improve the system after the competition is over. The way core computational components, cameras and sensors are connected to each other is shown in Figure 2.2. Note that the Jetson TX2 embedded system was launched by Nvidia in March 2017 which made it impossible to

perform any tests on the system until May when Ascend was able to acquire them. Up until this point we will use the predecessor Jetson TX1 to perform experiments and tests.

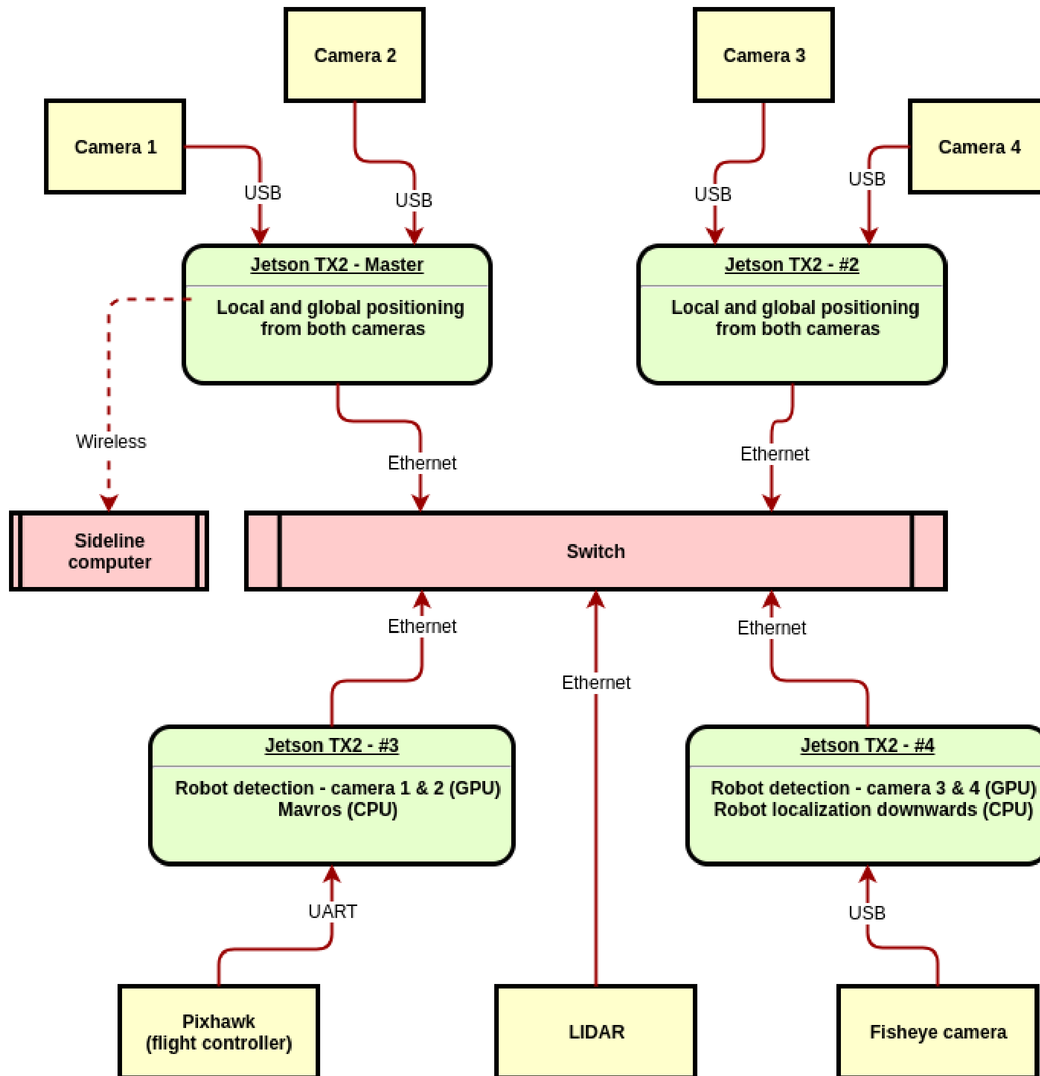


Figure 2.2: The different computational components and sensors onborad the drone that Ascend will use in IARC 2017.

2.1.4 Topic definition within Ascend NTNU

This thesis is officially in collaboration with Ascend NTNU and will focus on using computer vision and deep learning to facilitate the autonomous controls of a drone competing in the IARC competition. To see how this would work we have to take a look at the internal structure of Ascend. The team is split into several smaller groups each focusing on one specific task:

- Hardware - This group focus on building the drone and making sure it has all the required and desired components installed. They ensure it is modular as well as robust to enable different hardware setups more easily and allows it to crash a few times without destroying any vital components.
- Perception - Tries to make sense of all the sensory input from different sources on the drone. This group has a special focus on input from the five onboard cameras and tries to capture the state of the world around it. The world that is expected to be perceived is confined to the grid within the IARC competition. By creating a state model of the world that is as reliable as possible Perception allows for the Planning group to make better decisions.
- Planning - Planning gets the state model from the Perception group and applies it in a search for the best possible path for the drone. The optimal solution should guide as many ground robots as possible to the green side of the grid within a 10 minute period.
- Control - Control gets the preferred path from the Planning group and makes sure the drone safely gets to the next location without crashing in any of the tower robots that might obstruct the path. This group communicates directly with the hardware on the drone.
- Admin - The admins make sure communication flows between the other groups and that everyone has a clear common goal. They also take care of any administrative tasks and budget handling.

This thesis will focus on the perception part of Ascend. More precisely it will research whether or not deep neural networks can be adopted to detect and localize ground robots in images taken onboard the drone in real-time. This topic definition prompts several challenging issues like how localization should be performed, how accurate will it be detecting robots of different sizes with different resolution in the image, and how to do it all in real-time. The greatest challenge seems not to be whether detection and localization can be done, but whether it can be done fast enough onboard a drone with limited computational power. For this solution to be of any use for the Ascend project it has to perform well and fast. A lot of the research in the following sections has been selected because of its commitment to being computationally cheap compared to other methods.

2.1.5 Existing image data

Because of Ascends participation in IARC 2016 they have video recordings from multiple runs in the arena of about a minute each. The data consists of a video stream from each of the four onboard side-cameras and one from the fisheye camera pointing directly down. This means that there exist about 50.000 images from the competition we want to optimize for that can be processed and used to either train or test different approaches. The drone was equipped with a LIDAR (Light Detection And Ranging) that constantly measured the flying altitude as well as an IMU (Inertial Measurement Unit) which recorded the roll, pitch and yaw of the drone at all times. The team also

maintained a complete log of all sensor data and image recording with corresponding timestamps which makes it possible to find the altitude and angle every image was recorded at.

2.2 Deep learning models

Deep learning is quite a wide term for any machine learning based on deep graphs that is able to learn complex concepts without specific guidelines from a programmer. It should be able to learn solely by examples and adapt accordingly. The graph represent a hierarchy of simpler concepts that can be learned individually. The more complex overall concept is then learned by combining these smaller parts. The most common example of a deep learning model is the feedforward deep network or multilayer perceptron. A multilayer perceptron is just a mathematical function mapping some set of input parameters, e.g. an image, to a set of output values. MLPs will be also be the focus point of this thesis as we want to create a network that can detect and localize ground robots in a given image. This section will describe different deep learning models that are useful to either directly or as a concept to create something else from.

2.2.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks have been around for almost 20 years. They were first successfully implemented in the 1990's by Yann LeCun [22]. He created LaNet which could classify different types of numbers with high precision. Even though it has existed for a long time it was only recently popularized due to improvements in graphics cards and interfaces that allow networks to train on them in parallel. The creation of AlexNet sparked the interest in 2012 and several improvements has been proposed since which demonstrates the power of CNNs. We will soon take a more detailed look at AlexNet and a few of its descendants to see in which ways they might be useful to this thesis.

CNNs get their power from the convolutional layers that each has a set of learnable filters. Every filter will through training learn to activate for a specific type of feature. These features are sometimes run through a max pooling layer to down-sample the image representation to allow for assumptions to be made about the different regions of the image at different scales. Convolutional layers in combination with max-pooling layers make it easier to define the abstract form of an object and learn a more general representation of it. The output they create is called a feature map which gives us information about which features were present at different positions in the image. Specific combinations of these features then makes it possible to classify the image or parts of it into different categories. This last step is often done through a small number of fully connected layers at the end of the network. A much more thorough explanation of DNNs and CNNs can be found in Andrej Karpathy's excellent lecture notes from the Stanford computer science class CS231n [19].

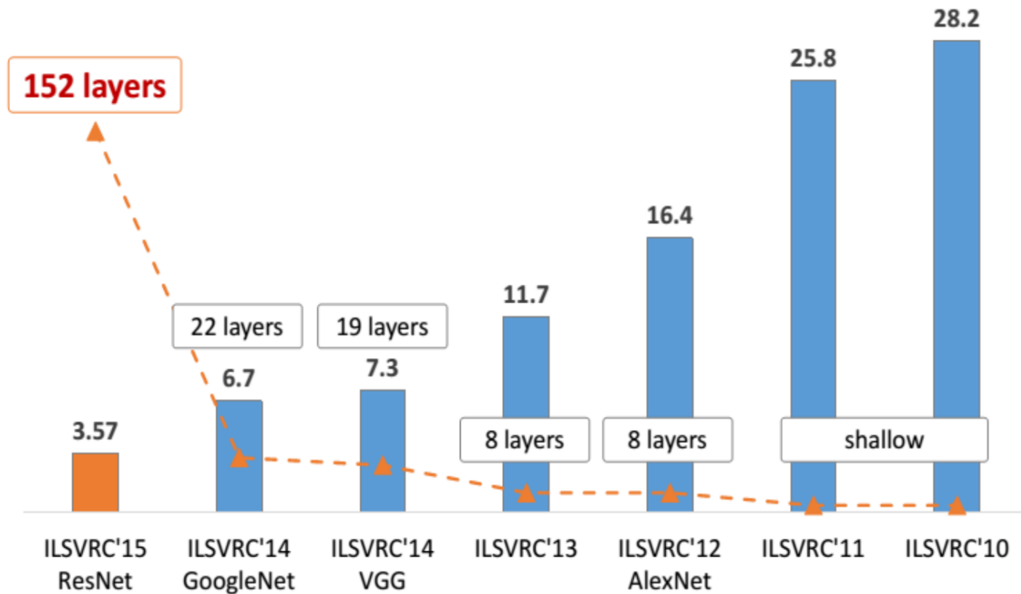


Figure 2.3: The evolution of the winning contributions in ImageNet[40] competition over the past 6 years. The bars represent the top-5 error for each of the approaches. The graph can be found in the presentation slides of the ResNet paper[17].

2.2.2 AlexNet

AlexNet[21] was introduced as a contribution in the 2012 ImageNet[40] competition which in its own words "*evaluates algorithms for object detection and image classification at large scale*". We will soon see that ImageNet has been an exceptional place for ground breaking CNNs to get recognition. AlexNet significantly outperformed the other computer vision algorithms in 2012 and became an instant success. It was created to be able to differentiate between 1.2 million images into 1000 different classes and had only a 16% error rate in the competition compared to the 26% of the runner-up. This sparked a huge interest in CNNs which led to several ensuing changes to the architecture to improve the accuracy further. AlexNet was created by connecting 5 convolutional layers followed by 3 fully connected ones. Because of its quite dense structure this sums up to 60 million parameters and 500,000 neurons which we will soon see is quite a lot for such a shallow network.

AlexNet is of interest to this thesis because it could provide a somewhat overpowered but *reliable* way to classify images to decide if it contains a ground robot or not. This would enable a slow program that could detect which parts of an image contains robots and which parts does not without many errors. This segmentation could further be used to create reliable data for a network that can actually localize the different robots.

2.2.3 Googles Inception model

A team from Google also participated in ImageNet 2014 and won with a network they called GoogLeNet[44]. GoogLeNet has been adapted and improved since and the newest version is commonly known as the Inception v4 model[43]. Structurally it is very different from the standard model that AlexNet uses. It uses a layer setup called an inception module which dramatically reduces the number of required parameters. Despite its deep structure it uses 12 times fewer parameters than AlexNet while still being significantly more accurate. The only problem with this approach is that it is hard to fully understand and even harder to modify. Just to fine tune it require the use of three different loss functions in stead of one. Its full structure is shown in Figure 2.4.

The Inception model can be useful in two different ways. First, in the same way we intended to use AlexNet, which could possibly gain even more accurate segmentation results. However, considering that this model is designed to differentiate between 1000 different classes while we only need two, it might not introduce any noteworthy increase in performance compared to the AlexNet model or other cheaper solutions. Second, we could use a pre-trained version of it as a starting point for some other type of network. It is possible to remove only the final output layer of the Inception model and append a completely new network structure as a replacement. This is commonly referred to as *transfer learning*[15, Section 15.2]. Each convolutional layer in the Inception model will describe features in the input image that gets more general the deeper the layer is. The last convolution layer will therefore output activations for a large number of general features. In the original model these are used in a final fully connected layer to determine the class of the image. We could instead use the feature activations as a starting point for a segmentation network or other types of useful networks that could benefit from having features as input.

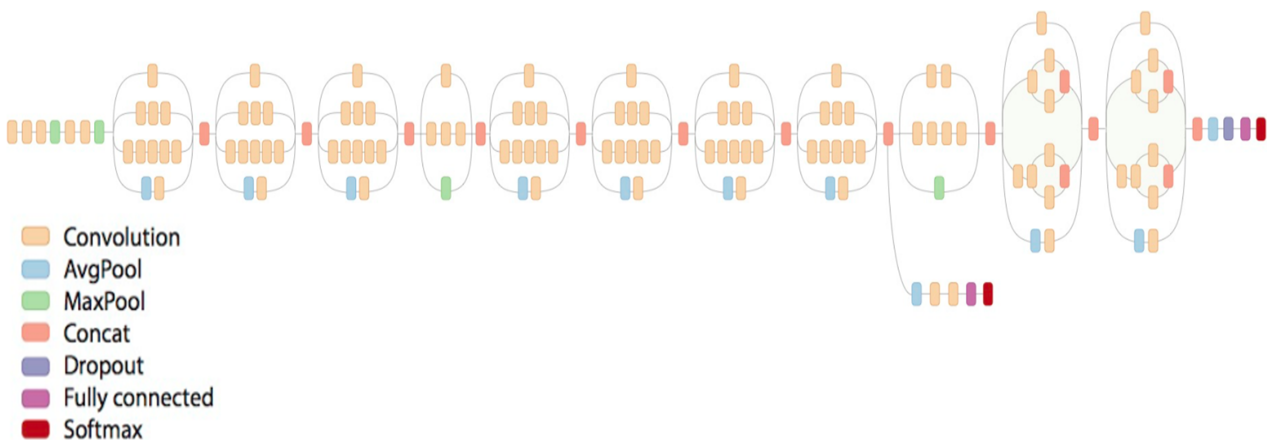


Figure 2.4: Architecture of Googles Inception v3 Model

2.2.4 VGGNet

VGGNet[42] was the runner up in ImageNet 2014. It was originally developed as an extension and improvement to the AlexNet model from 2012 but is much deeper and more efficient than its predecessor. Its main contribution was to show the relationship between network depth and performance accuracy. 16 layers was considered very deep and difficult to train at the time because the convergence of the deeper layers was hard to control. The creators of VGGNet solved this by splitting the net into smaller pieces and train them as a pre-training process. This was time consuming but the results were impressive. There now exist initialization methods[29] that renders the pre-training process useless and makes the VGGNet much simpler to train from scratch.

VGGNet is a rather large network with about 140 million parameters. About 100 million of those parameters are located in the first fully connected layer shown in blue in Figure 2.5. There exist methods that reduce this layer significantly without reducing the overall performance of the network. VGGNet is also popularly used as a feature extraction network where only the convolutional layers are used to create feature maps for other types of networks. This means that we are left with about 15 million parameters. However, the main reason it it used is because of its combination of simplicity and accuracy. It builds directly on the AlexNet model and stacks simple 3x3 convolutional layers after each other with an occasional max pooling layers to down-sample the features. This makes it easy to understand, modify and use. Because of its modularity it comes in several sizes, but a depth of 16 or 19 are by far the most common.

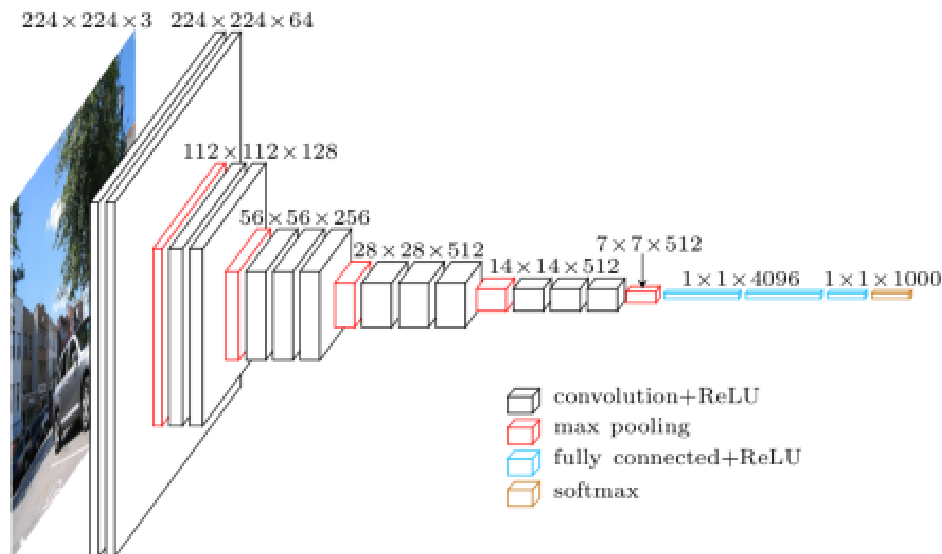


Figure 2.5: Architecture of VGG16

2.2.5 ResNet

ResNet[17] is yet another winner of the ImageNet competition, this time in 2015, which reinforced the trend we have seen so far - deeper it better. Does this mean that we

can simply stack more layers on top of each other and add more computational power to increase the performance of the network? Unfortunately not. This is caused by the fact that the magnitude of gradient flow during backpropagation decreases drastically after about 20 layers which renders the network unable to learn. ResNet addresses this problem by creating something called a “network-in-network” architecture. The overall difference is that ResNet consist of a large collection of shallow networks placed after each other with an occasional pooling layer. This allows the training process to bypass the vanishing gradient problem. Another effect of this architecture that is much easier to change after training. If you remove a single layer from a trained VGGNet you will have a very significant drop in performance and the netwrok is basically useless. With ResNet you can remove any of the deeper layers with a accuracy drop of about 0-5%. The initial layers will have more of an impact, but still much less than in a more traditional network. Veit et al. published a paper[45] that tries to explain in a more intuitive way how and why ResNet works and is worth a read for more in depth information.

2.2.6 Comparison of classification networks

A strictly numerical comparison between different types of classification networks in terms of number of layers, speed and accuracy can be seen in Table 2.1. All of the structured discussed here are used in different applications for different reasons. However, the two methods that seem to be used the most in terms of feature map extraction for more advanced networks to build on are VGGNet and ResNet. VGGNet is often preferred to GoogleNet even though GoogleNet has fewer parameters and faster computation time. The reason ins that VGGNet is slightly more effective and has a more straight forward architecture with a single loss function compared to the three used by GoogleNet. This makes it a lot simpler to modify. ResNet offers a variety of sizes which makes it possible to adjust the speed to accuracy ratio for different purposes. The choice of method is very dependent on the application and we will therefore asses them more later in more contextual settings.

2.3 Object Detection Networks

Neural networks are remarkably powerful for a specific set of tasks but are unfortunately limited to very static architectural structures. They have a constant number of input nodes, and a constant number of output nodes which cannot be easily modified without requiring a substantial amount of retraining. Neural networks have therefore primarily been used in tasks like image classification that take an image of fixed size as input and outputs a probability distribution over a fixed set of possible categories. A classification network can however easily be adapted to become a detection network for *a single object* where the last layer learns to output the coordinates of a bounding box in addition to the classification output. A solution like this can be implemented as long as there exist sufficient amounts of good data to train on, which can be hard to obtain. The real challenge arise when we want to detect an arbitrary number of objects in an image. It is no longer desirable to use a constant number of outputs bounding boxes because of

Network	Layers	Inference time (ms)	Top-1 error (%)	Top-5 error (%)
AlexNet	8	14.56	14.56	19.80
GoogleNet	22	39.14	-	10.07
VGG16	16	128.62	27.00	8.80
VGG19	19	147.32	27.30	9.00
ResNet-18	18	31.54	30.43	10.76
ResNet-50	50	103.58	24.01	7.02
ResNet-101	101	156.44	22.44	6.21
ResNet-200	200	296.51	21.66	5.79

Table 2.1: Comparison between different types of classification networks in terms of number of layers, speed and accuracy [18]. The speed is measures in milliseconds for a forward and backwards pass on a Pascal Titan X GPU with cuDNN 5.1 installed. Top-1 error is a measurement of how often the top output prediction is incorrectly labeled while top-5 error is the percent of times the correct label is not even in the top 5 output predictions. The top-1 and top-5 errors are based on images from the ImageNet 2012 validation set where the center of the image has been cropped out and passed through the network. For VGG-16 and VGG-19 there only existed data which instead use multiple crops at different locations in the image. All models perform better when using more than one crop at test-time which gives the VGG models a slight advantage.

the irregular number of observable objects. One solution is to create a combination of networks that first find possible object locations and then run a classification network for each of the proposed regions. R-CNN networks are good examples of this. The second solution is to create a network that outputs a constant number of detections for all possible regions in the image and threshold the results afterwards. The second solution is what we define as a single-shot detection network where YOLO, SSD and YOLOv2 are the most acclaimed examples of this. Instances of both approaches will be presented and discussed.

A large amount of papers and articles on object detection techniques for multiple objects using deep learning are constantly being published in an effort to find the most efficient and accurate method. They are performing increasingly well and the accuracy some of them exhibit would be sufficient to create a great detection and localization system for ground robots. However, computation speed is not always prioritized and many of the solutions are performing far to slow to be run on a Nvidia Jetson system in real-time. The main focus of the following sections will therefore be on fast alternative methods of multiple object detection.

2.3.1 Region-based Convolutional Neural Networks (R-CNN)

R-CNN[12] is a good example of how a fixed size neural network can be used to output a varying number of object detections. It proposes a solution that first takes an image as input and extracts a set of about 2000 class independent region proposals from it. The proposed regions are then sent through a CNN to extract features before a support vector machine (SVM) identify which class, if any, it belongs to. Figure 2.6 is taken from the R-CNN paper and shows the different steps in the process. R-CNN performs well in terms of classifying the correct objects, but can have some localization errors. However, the bigger problem is that it takes about 20 seconds to run detection on a single image using a powerful desktop GPU. This is far from fast enough for us to consider.

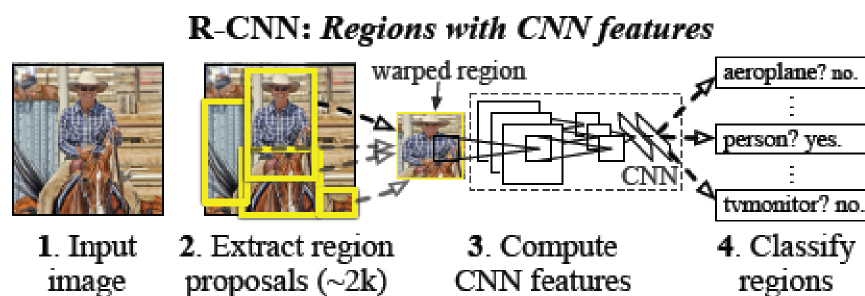


Figure 2.6: Structure of the R-CNN network. Image is found in [12].

Fast R-CNN

The paper on Fast R-CNN[13] claims that R-CNN is so slow because it performs a full forward pass in the CNN for each of the 2000 region proposals without ever reusing

any of the computed values. It proposes the use of Spatial Pyramid Pooling networks (SPPnets) to speed up this process. This method computes the convolutional feature map for the entire image once and uses it directly to classify different regions instead of recalculating only the region specific features for every proposal. This did improve performance by a factor of 10 which means we are down to about 2 seconds computation time per image, or 0.5 frames per second (FPS). This is still not fast enough for a real-time system, especially not for one running on a small embedded system on a drone.

Faster R-CNN

Faster R-CNN[37] recognizes that the Fast R-CNN method significantly increased performance at the CNN but neglects the still very slow creation of region proposals. Despite using GPU-accelerated CNNs the regions are created using methods implemented on the CPU. Faster R-CNN introduces Region Proposal Networks (RPNs) to solve the problem that share convolutional layers with the already existing CNN and greatly reduces the time it takes to create proposals. This vastly improves performance and the final result achieves about 7 frames per second. This is within the limits of real-time performance, but considering the restricted hardware specs this project will make use of compared to the powerful GPUs used in the papers we would prefer an even faster solution.

2.3.2 YOLO - You Only Look Once

YOLO[34] is a method coming out of University of Washington that introduces a single neural network that predicts bounding boxes and their classifications directly from a full sized image. It tries to take a step back from the region based methods where you potentially try to classify the same region several thousand times. Its goal is instead to only look at every part of the image once, hence the name. To realize this they start by splitting each image into a 7x7 grid where each cell is responsible for predicting a few things. First they predict a small set of bounding boxes which each has the cell as their center. The number of predictions for each cell is a small constant, which in the paper was set to 2. Second it should predict confidence values for each of the bounding boxes. This reflects how sure the YOLO network is that the bounding box covers some object. If the cell does not contain any objects it will still propose the same set of bounding boxes, but the confidence values should be low. By combining the results from each of the 7x7 cells we get something that looks a lot like the bounding box predictions described in R-CNN methods, but with less overlap. Compared to the R-CNN which has about 2000 possible bounding boxes to classify individually, we now have 7x7x2=98 (2 for each cell in the grid) boxes with corresponding confidence values. Next, the network tries to classify each of the bounding boxes. This is not done by running it individually for each box, but rather for each of the cells. Because every bounding box is centered in one of the cells we can safely assume that the object within the bounding box is the same as the one defined for the cell. The final output is then all the bounding boxes with corresponding confidence values and class probabilities. To find the correct boxes and their classes we simply threshold the confidence values and discard all the weak probability boxes. The process YOLO goes through is shown

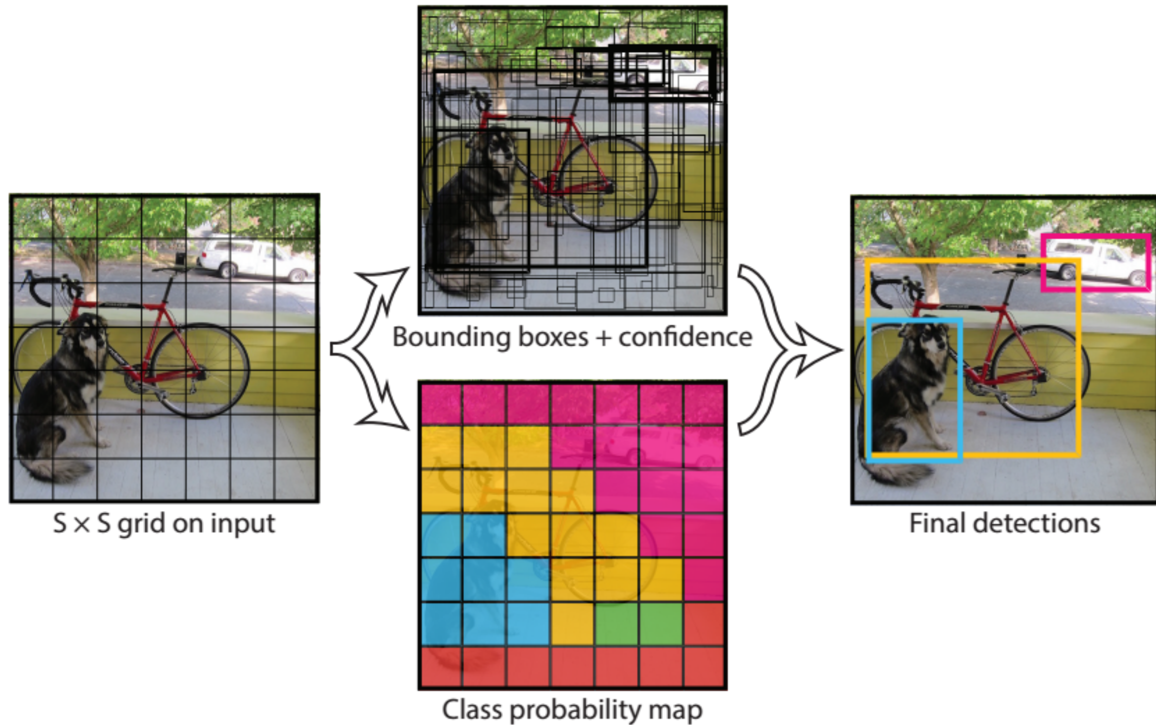


Figure 2.7: The leftmost image shows how the input image is divided into a 7x7 grid. The upper image shows how the network predicts a given number of bounding boxes their confidence for each cell in the grid. The lower image shows which classifications are most likely in the different cells. Each color represent one of the 20 categories. The image to the right shows the final output after being thresholded. Image is found in [34].

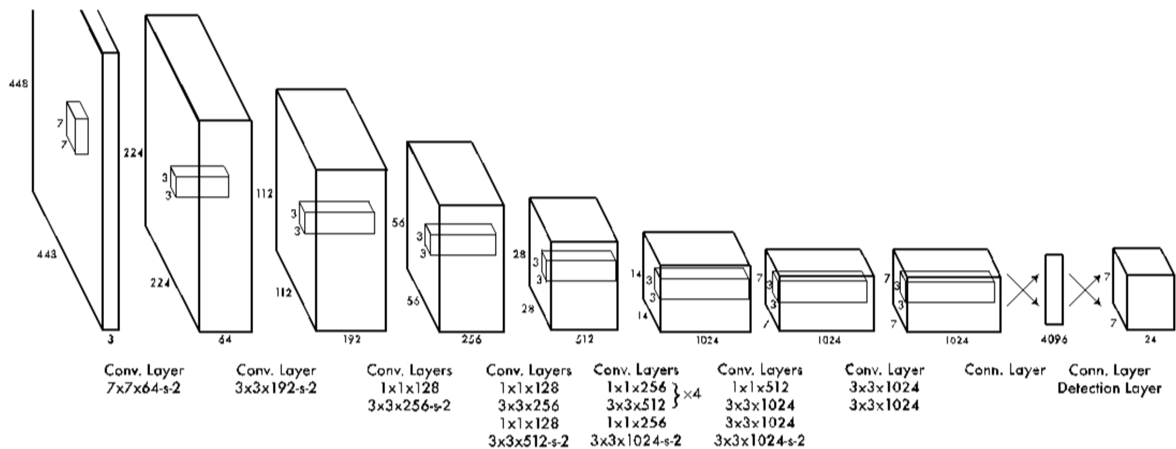


Figure 2.8: This is the full architecture of the YOLO network. Image is found in [34].

in Figure 2.7. In the paper they use 20 classes and each bounding box is defined by 5 meta-data values. This gives a total of 7x7x(2x5+20) = 1470 output nodes. For a modern neural network this is not whole lot, especially for a object detection network. The final architecture of YOLO is shown in Figure 2.8. The paper also claims to run

at 45 frames per second on a high end graphics card which is a good starting point for realtime applications. Another important observation stated in the paper is that "*Compared to state-of-the-art detection systems, YOLO makes more localization errors but is less likely to predict false positives on background*". This should be taken into account when considering which method to go for.

2.3.3 SSD - Single Shot Multibox Detection

SSD[25] is a detection network that like YOLO produces a fixed-size collection of detection outputs where the highest values are selected and presumed to be valid object detections. The initial layers of SSD are based on one of the standard classification networks mentioned in Section 2.2 where the last fully-connected layers are removed so that it outputs a set of feature maps. In the paper they have chosen to use the VGG16[6] network with a few moderations proposed in [6] to truncate it and make it perform faster without any significant drop in accuracy. They then add a set of convoluional feature layers at the end of the base network which gradually decrease in size to enable feature extraction at multiple different scales like we can see in Figure 2.9. Each map is of size $m \times n \times p$ where $m \times n$ is the number of feature cells in the map and p is the depth of the feature vector for each of them. The feature maps of each layer is then used for two things: input to the next layer, and directly to predict detections. For each cell we can then output a constant number (set to 4 in the paper) of detection predictions relative to that cells position in the image to allow for bounding boxes of slightly different sizes and shapes. The initial layers have a large m and n values which splits the image into smaller pieces which again results in detection of small objects. The cells of the deeper layers covers a larger part of the image and allows for larger objects to be detected.

The main improvements SSD has over YOLO is that it is fully convolutional and that it detects at multiple different scales. Fully connected layers are extremely expensive because they require such a large number of connections which slows the down computation significantly. SSD only uses convolutional layers which means it can use all that extra computation to focus on a much larger number of detections for each position in the image. In Figure 2.9 we see that YOLO outputs 98 predictions per class while SSD is able to output 8732 and still run at the same number of frames per second. This gives SSD a huge advantage when detecting objects of varying sizes and makes it much more prone to location error.

2.3.4 YOLOv2

YOLOv2 [35] is a renewed version of YOLO as a response to the release of SSD. SSD did a few things better than YOLO which made it the better detection network option at the time. However, YOLO had a lot of potential for improvements, and some of them were even addressed in the SSD paper[25]. We will take a look at these differences and how they were addressed to create YOLOv2.

There are a few differences between the first and the second version that both managed to speed up the process and make it more accurate at the same time. Some of

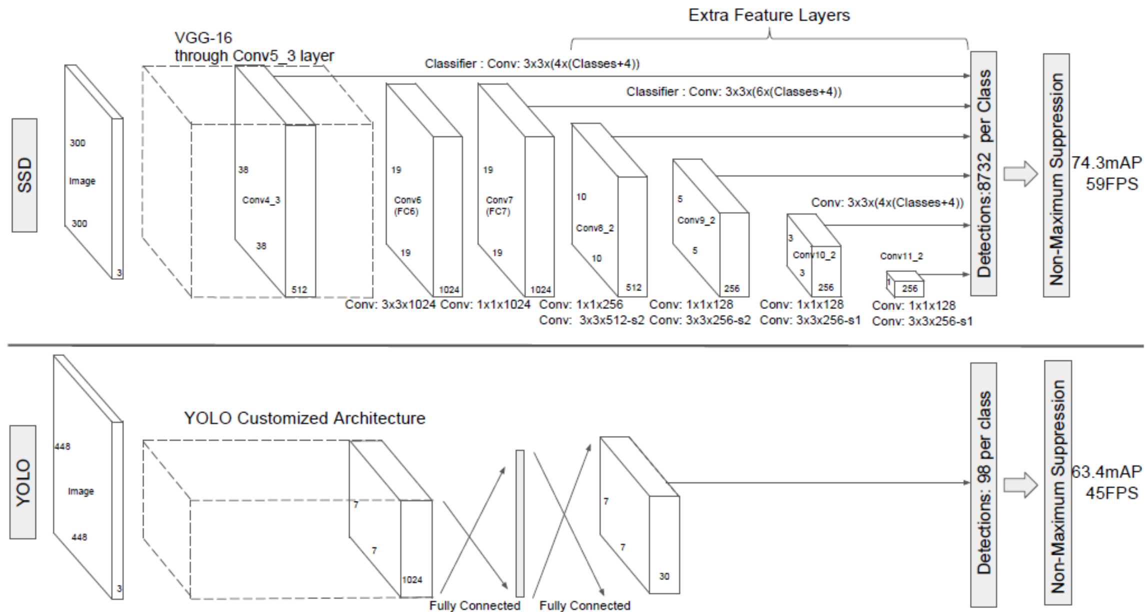


Figure 2.9: The upper part of the image show the architecture of the SSD network an how it scales won the features from the base network to create feature maps of different sizes. Features are divided into a set of cells which contain a feature vector, e.g. the features from VGG16 contain 38x38 cells and that each have a feature vector of size 512. Each cell is then used directly to predict bounding boxes and confidence values for objects that approximately fit the cell. The same is done for each of the extra convolutional feature layers to provide features at different scales in the image. The first layers detect small objects in the image while the deeper layers combine smaller features to find the larger object in the image. In the lower part of the image we see the comparison to the YOLO architecture. YOLO might seem like a simper approach, but the fully connected layers severely hurt performance and the lack of scaling makes it more prone to location error. Image is found in [25].

the more technical fine tuning done will be briefly mentioned while the full description is available in the paper[35] and their impact on accuracy is shown in Figure 2.10. First and most importantly the network was converted to a fully convolutional model where the final detections were calculated directly from the feature maps using anchor boxes which works much like the predefined detection outputs from each feature map cell in SSD. This did not improve the accuracy of the network directly but enables it to run much faster. In difference to SSD they did implement a k-means clustering method to learn the most effective setup for these output boxes which increased the performance by a staggering 5%. Another weakness of the first version was its use of a pre-trained classifier at 224x224 that was simply upscaled to 448x448 to enable larger detection resolution. This was done to avoid having to retrain a 448x448 network from scratch which can take a long time, even for companies like Google with access to large amounts of computational power. The difference in v2 is that they fine tuned

the upscaled network on full scale images before implementing it which made it able to adjust a lot better than the purely upscaled version. The rest of the improvements and the full transformation from YOLO to YOLOv2 can be seen in Figure 2.10.

Finally the paper mentions a few steps used to improve both classification and detection through training the network to work specifically well with the classes in the ImageNet and COCO datasets. This is done by training the YOLOv2 detector network on both classification data and detection data. This creates a detection network that can not only learn the 1000 classes in the detection dataset but also the almost 9000 classes specified in the classification dataset.

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figure 2.10: The path from YOLO to YOLOv2 found in [35]. Most of these design choices lead to improvements in mPA while switching to a fully convolutional network and using anchor boxes mainly improved the computation speed of the network. The paper also mentions that switching to the anchor box approach increased recall without changing mAP and at the same time cut computation by 33%.

2.3.5 Comparison of detection networks

In Table 2.2 we see the numerical comparison between the most promising methods we have looked at so far. This table will be used for further evaluation of the methods in Chapter 3.

2.3.6 Dlib and Max-Margin Object-Detection (MMOD)

Davis E. King released a paper in 2012 called Max-Margin Object-Detection[20] that described how to optimize the training process of sliding window detection methods. In the paper he uses a HOG detector in combination with the proposed MMOD function to calculate loss. It also proposes an algorithm that optimizes the loss function over a full sized image to maximally improve the detection for every possible window in that image. This is a very powerful method because it enables detectors to become very

Network	VOC 2007 mPA	Framed per second
R-CNN with T-net [12]	58.5	0,05
Fast R-CNN with VGG16 [13]	70.0	0.5
Faster R-CNN with VGG-16 [37]	73.2	7
YOLO [34]	63.4	45
SSD 300x300 [25]	74.3	46
SSD 512x512 [25]	76.8	19
YOLOv2 416x416 [35]	76.8	67
YOLOv2 544x544 [35]	78.6	40

Table 2.2: Comparison between the most promising detection networks mentioned in Section 2.3. mPA stands for mean average precision and is used as an accuracy measurement in object detection where a higher score is better [9]. These tests were created using VOC 2007+2012 training sets and the VOC 2007 testing set to calculate the mPA.

good with only a small number of training examples. The detector is trained to find objects of a constant size but can simply be run on multiple image scales to enable location of different sized objects. Davis E. King is also the creator of an open source machine learning toolkit called dlib which recently implemented the MMOD approach where the HOG detector was replaced by a convolutional network. It was able to create a descent face detector from a training set of only 4 images each containing between 4-6 faces. More impressively it was able to match the performance of a state of the art R-CNN face detection approach trained on 160.000 images using only 4600 images itself. It also claims that 640x480 images can be processed in 45ms which amounts to 22 FPS on a GPU if they are processed one at the time. If the images are processed in batches it takes only 18ms per image which gives us 55 FPS. The only drawback is that this seems to be measured on a single scale meaning it only looks for object of a constant size. The MMOD method addresses several issues that are important to this thesis and could prove very useful.

2.4 Semantic segmentation

Semantic segmentation is a form of segmentation that not only tries to distinguish different parts of the image but also make a prediction of what those parts consist of. In context of this thesis this is interesting because an image that is able to segment out the positions of robots from the background could be a great starting point for a localization algorithm. This will require a good location extraction method that above all must be accurate. The approaches described in this section can be useful in two different ways:

1. A slow but accurate approach that can transform the existing dataset from IARC 2016 into a detection dataset with bonding boxes around all visible robots within the images. A detection dataset will enable us to train one or more of the detection networks that will be discussed in Section 2.3.

2. As a stand alone solution to the robot detection research question we want to address in this thesis. For this to work we need a fast segmentation approach in combination with a way to quickly convert the segmentation image into location data for each of the robots.

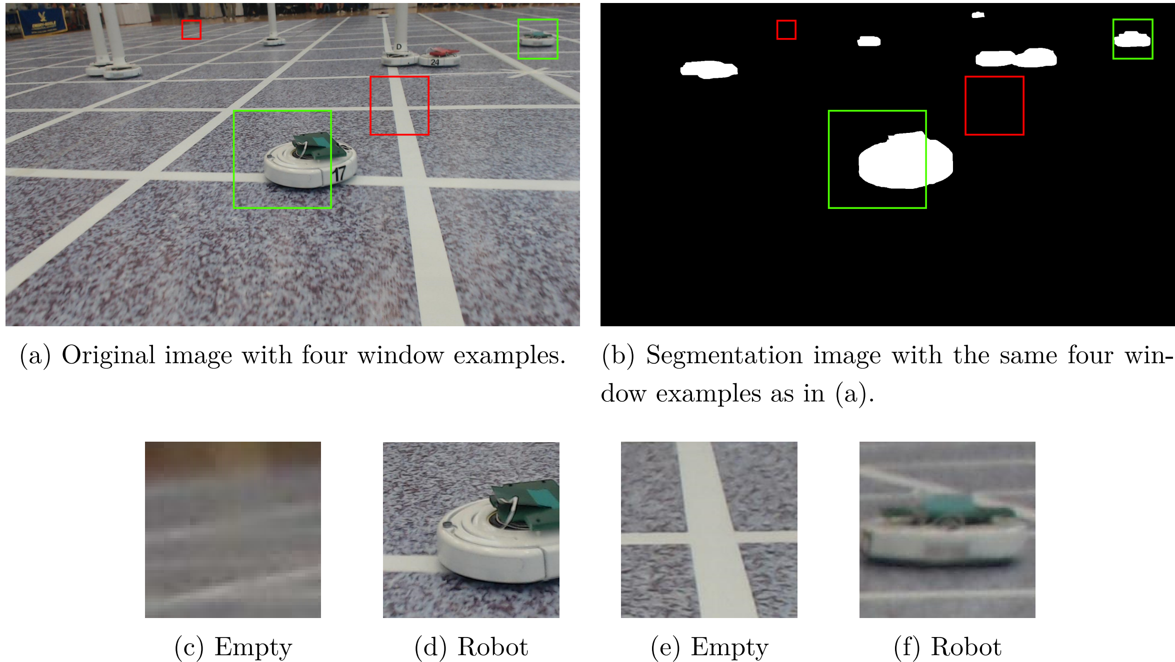
We will go through techniques that facilitate both of these approaches to see which have the greatest potential. The main reason a segmentation and extraction combination might be more attractive than a complete detection network solution is the fact that many detection approaches are slow doe to the large number of locations it has to consider possible detections. The field of image segmentation using neural networks has has a greater focus on running an entire image through the network once to get a sense of where objects are located within it and therefore have the potential of running faster, even though it requires extra computation at the end to properly extract the object locations.

2.4.1 Sliding window

The simplest solution to consider is a sliding window operation where the image is divided into possibly overlapping windows before each of them are classified through a classification network. The results of the individual window classifications are then mapped to an output image which creates a semantic segmentation of the scene. An example of how this segmentation would look like along with four example windows is illustrated in Figure 2.11. A more practical example of this approach will be discussed in Section 3.3.1.

2.4.2 Early stage of CNN image segmentation

In 2013 Yann LeChun was once again involved in the development of a new type of neural network. This paper [23] proposes a solution that first scale down the image 3 times and run each of the scales through a CNN. It also runs a separate segmentation on the image which vastly over-segments the image meaning is divides the image into too many objects rather that too few. The over-segmentation is done in three different ways using: 1. Superpixels, 2. Conditional random field over superpixels, 3. Multilevel cut with class purity criterion. An illustration of this setup is shown in Figure 2.12. It takes approximately 0.7 seconds to process a 320x240 image. Even though this is too slow to be viable it still emphasizes the importance of being able to detect at different scales and how feature maps can be used directly as identifiers of different region classes.



(a) Original image with four window examples.

(b) Segmentation image with the same four window examples as in (a).

(c) Empty

(d) Robot

(e) Empty

(f) Robot

Figure 2.11: Example of how windows are classified in a sliding window approach. (a) shows the original image with four example windows. The green windows should be classified as robots while the red windows are empty. The same windows and how they would be used to create a segmentation image is shown in (b). Cutouts of the windows are shown in (c)-(f). To go from (a) to (b) we slide the window across all pixels in (a) and set the center pixel to 1 if the classification is a robot and 0 if it is empty. (b) represent the ideal segmentation which might be hard to recreate exactly, but the output of a sliding window should look very similar if the classifier is trained well. In this example we use different sizes for the windows to make sure they fit the robot sizes. How this can be realized will be discussed further in Chapter 3.

2.4.3 FCNs - Fully Convolutional Networks

A fully convolutional network [41] can be created by replacing the final fully connected layers in a standard model like AlexNet or Inception with convolutional layers that output a classification for each pixel at a given resolution. The architecture for this process is shown in Figure 2.13. According to the paper [41] FCN can output a 10x10 segmented image from a 500x500 input image in 22ms on a high end desktop graphics card. In comparison AlexNet use approximately 1.2ms to classify a single 227x277 image. Meaning that the FCN effectively classifies 100 different segments of the image 5 times faster than it would take AlexNet to go through them all. This is not terribly impressive considering this has to run on a Jetson card and that the output resolution must be significantly higher that 10x10 to ensure that even the smallest robots in the image gets detected. Still, the part that is most computationally expensive in convolution based networks is the (usually pretrained) convolutional layers that create feature maps. If this is true for FCNs as well then upscaling the output layer would

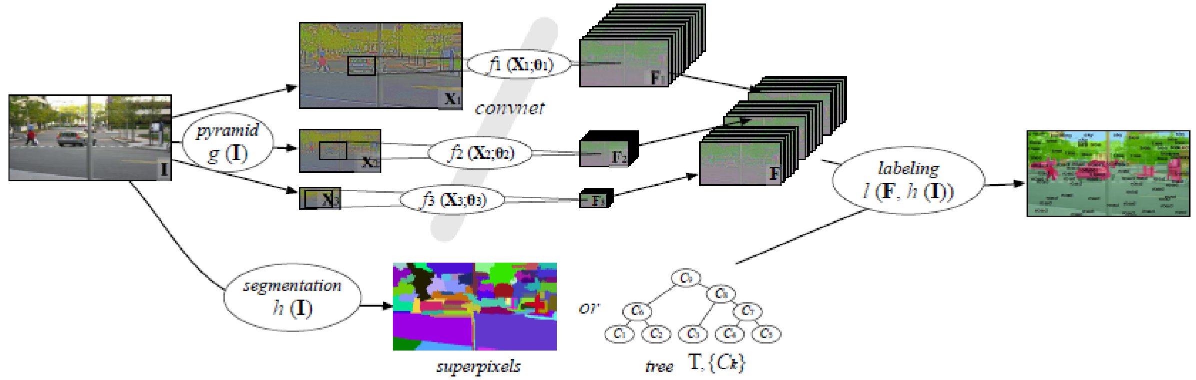


Figure 2.12: This is a diagram of the complete scene parsing system created in [23]. The input image is first transformed through a Laplacian pyramid to create different scales of the image. These images are fed through different convolutional networks which each creates a set of feature maps. The feature maps from the smaller scales are then upsampled to match the size of the larger feature maps before they are all concatenated. Each feature vector then represent a large contextual window at different scales around each pixel. In parallel, some kind of segmentation is performed. Several different methods are tested in the paper. Lastly the results are combined to create the final segmentation output. The paper does not focus on a sole method for this last step but rather try out several options.

not effect the inference time by too much and FCN might just run fast enough to get a few frames per second on a Jetson system. However this leads to a new problem. The network can only handle single scale semantics within the image due to its fixed-size receptive field. The reception field denotes the resolution at which the segmentation network is able to operate. In the case of FCN it is restricted by the small size of the final convolutional layer which is necessary to capture larger features in the image but makes it hard to define the smaller ones. Therefore, the object that is too small for the receptive field may be fragmented, mislabeled or completely ignored.

2.4.4 Deconvolutional networks for semantic segmentation

Deconvolutional networks [30] are based on much of the same principle as FCNs [41]. We take out the final fully connected layer in a standard classification network and replace it by something else. This time the replacement is not only a layer or two, but a large multi-layered deconvolutional part that consists of unpooling layers and deconvolutional layers as shown in Figure 2.14. The only difference in the convolution half of the network is that locations of maximum activations selected during pooling operations are recorded to achieve a more accurate upsampeling later on. The deconvolutional half is in turn trying to smoothly upscale the features to a segmentation image. The unpooling layers have the opposite effect of normal max-pooling layers and scale the feature maps up instead of down while the deconvolutional layers undo

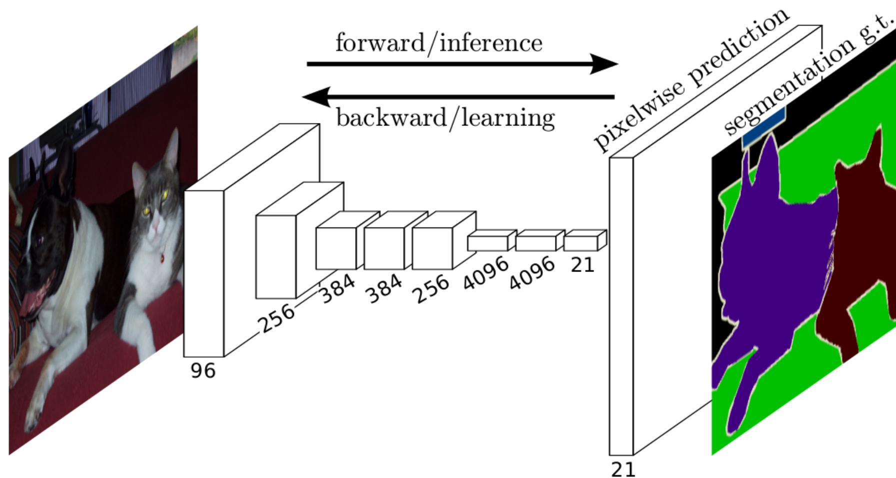


Figure 2.13: Illustration of the fully convolutional network architecture from [41]. The first part consist of convolutional layers that extract a set of features from the image. These features are used to predict the object class at the same resolution as the final feature map before it gets upscaled to the final output segmentation image.

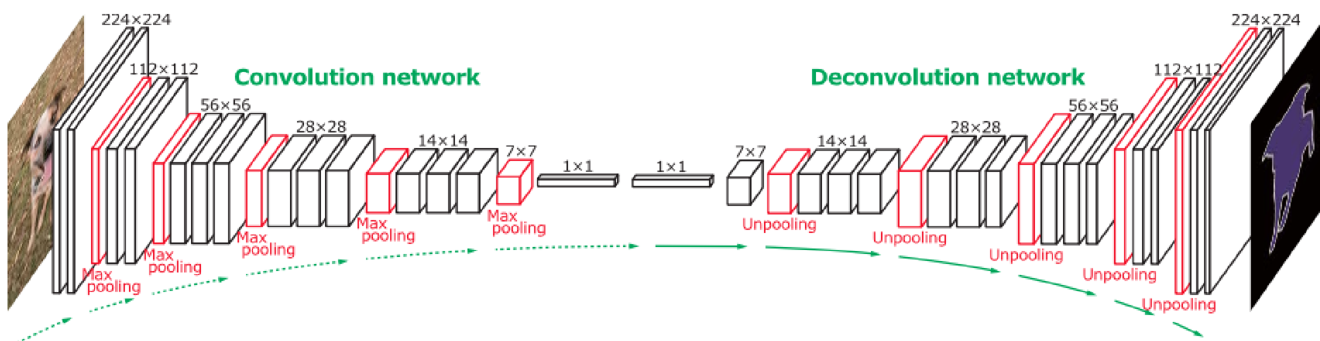


Figure 2.14: Diagram of the deconvolution network in [30]. The first half of the network is structured like most convolutional classification networks, but the last layer of size 1x1xP (where P is the number of dimensions) is treated as a convolutional layer even though it is a fully connected one. The second half of the network starts unpooling the feature maps to expand their receptive fields smoothly which mitigates many of the downsides with FCN [41].

the downscaling done by convolution. An illustration of how this is done is shown in Figure 2.15.

Deconvolution is proposed as a method of fighting the main weakness of FCN: its restrictive receptive field and coarse level of detail in the convolutional layer just before the output layer. The training is done in two stages, one for the convolution part and one for the deconvolution part. The paper describes how the convolution part was

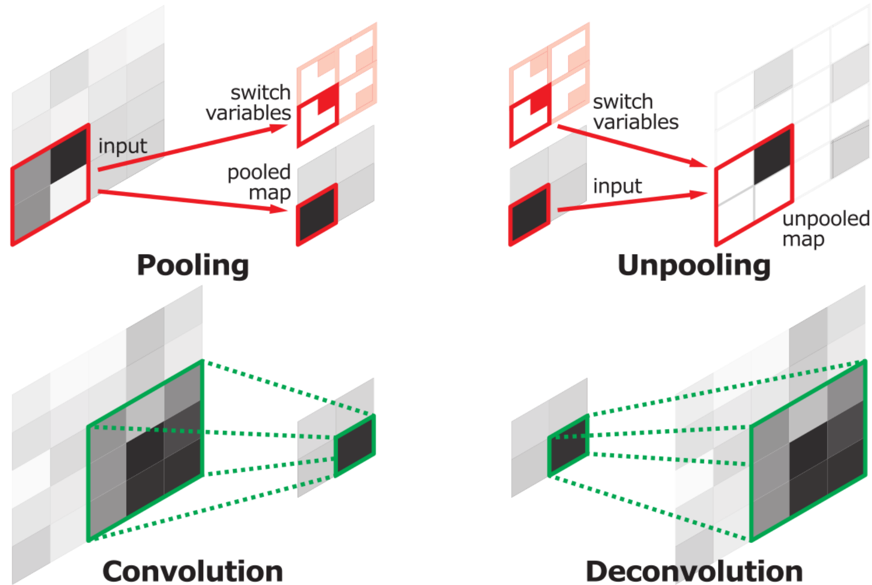


Figure 2.15: Illustration of unpooling and deconvolution operations done by the deconvolutional network in [30].

trained on a GTX Titan graphics card for 2 days while the last part was trained for 4 days.

A deconvolutional network could indeed be of use to this thesis as a step to segment out the ground robots from surrounding background. It performs almost 10% better than FCN on the VOC2012 dataset [8] but has no recorded data on inference speed which could be a bad sign considering its large number of convolution and deconvolution steps. Another important notice is that the deconvolutional network puts a lot of effort into refining the edges of the detected objects. This is not going to be a focal point in the case of robot detection because the extraction of robot positions from a segmentation image is most likely going to focus on the placements and sizes of “robot blobs” rather than their exact contour.

2.4.5 SegNet

SegNet is described in the paper [2] as a deep convolutional encoder-decoder architecture for image segmentation. It has a lot of similarities to the Deconvolutional network but has improved in some critical areas. First, it refers to its convolution part and deconvolution part as encoder and decoder respectively where the encoder part is a VGG16 network with all of the fully connected layers removed. SegNet is fully convolutional compared to the deconvolutional network which keeps fully connected layers in between the convolution and deconvolution. Considering that these hold about 90% of the parameters of the entire network we get a network that is a lot smaller and faster. Second, SegNet has replaced the trainable unpooling layers with what is calls upsampling layers. It performs the same operation but in a more simplistic way that requires less training and still yield the same result. SegNet claims to process 360x480

images giving pixelwise segmentation in real-time which sounds like an ideal starting point for this thesis.

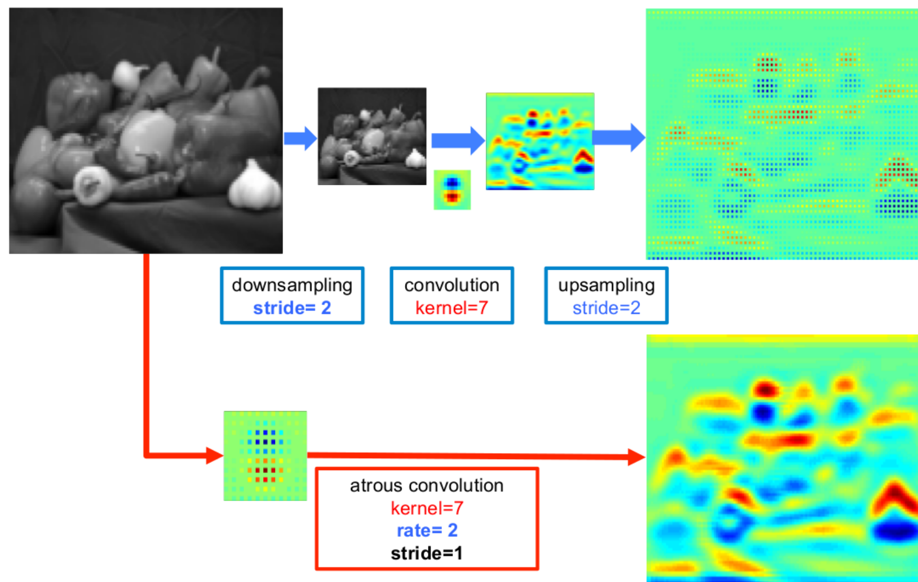


Figure 2.16: Illustration of atrous convolution in 2D from [6]. The top row shown how feature extraction with standard convolution on low resolution features is performed. The image has to be downscaled to fit the features and enable convolution before upscaling the output image again. In the bottom row we see how atrous convolution with a $rate = 2$, meaning it upscales the features by a factor of 2 before convolving it with the image and outputs a full resolution segmentation image.

2.4.6 DeepLab

DeepLab[6] tries to increase the spatial resolution of the features after several convolution and max-pooling layers without implementing and deconvolutional layers. Their solution is something called *atrous convolution*, or convolution with upsampled filters and can be applied either before or after training. Atrous convolution is simply the process of upscaling the feature extractions to the desired resolution by filling in zeros in between every feature. This does not increase the resolution of the features them self but increases the final resolution and outputs a smooth segmentation image. In Figure 2.16 we see how this looks like with a simple convolution example. The features can therefore be created without the need of upscaling which makes the network much faster. To increase performance we can perform this step on feature maps of various sizes from the earlier convolution layers in something the authors like to call *atrous spatial pyramid pooling*. It exploits multi-scale features by employing multiple overlapping filters with different rates to classify a single center pixel. The final step introduced in the DeepLab paper is a object boundary improvement mechanism called fully connected Conditional Random Field (CRF). These might not be necessary in our

Network	Forward pass(ms)	Backward pass(ms)	mIoU	Model size (MB)
FCN	317.09	484.11	49.83	539
DeconvNet	474.65	602.15	59.77	877
SegNet	422.50	488.71	60.10	117
DeepLab	110.06	160.73	53.88	83

Table 2.3: The performance[2] of the different segmentation network approaches discussed in Section 2.4. To measure performance we use the mean of intersection over union (mIoU) also known as the Jacard index which is the most commonly used metric in benchmarking[2]. The time estimates were calculated as an averaged over 10 iterations with a batch size of 1 and an image size of 360x480. The dataset used in the performance test is the CamVid[4] test set.

case because of we are not really interested in the contours of the objects but rather the centers. Figure 2.17 shows the full DeepLab process and we see that the steps leading up to the CRF might be just as useful in our case. This might even speed up the process by some small extent.

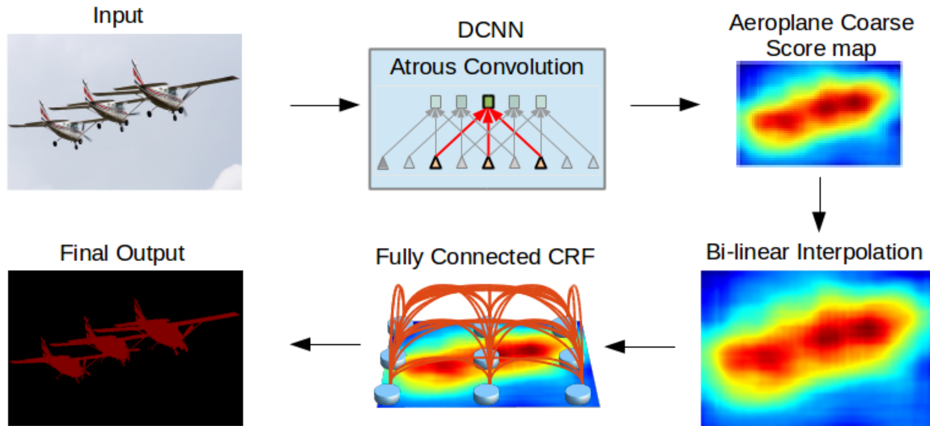


Figure 2.17: Illustration of the full DeepLab procedure.

2.4.7 Comparing segmentation networks

A comparison between the different segmantetion networks in terms of inference speed, accuracy and size can be seen in Table 2.3. Keep in mind that the different networks will perform differently on different datasets, so the mIoU performance measure ratio between the networks might vary.

2.5 Speed-up Techniques

There exist several techniques that have been developed to speed up various parts of the deep learning process which could be utilized in this thesis to increase performance. Some are aimed at the training process to make it more efficient like [27] which is intended to speed up the *training process* of segmentation networks that uses max-pooling layers. We will take a look at some of the methods that focus on speeding up the *run time* performance of the network, which in our case is much more interesting.

2.5.1 Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks

The sliding window approach described in Section 2.4.1 would apply a classification network to every patch in a larger image and calculate the feature maps for each of them. For every overlapping part of these patches we end up recalculate the same feature maps at a slightly different position. The paper [14] is based on a different method using "extended maps" which is a collection of all the feature maps from all patches in an image. The clue is that this approach never calculate the same feature map twice, like overlapping windows in the sliding window method usually does. This can be implemented easily if max-pooling layers are not present. However, modern neural networks owe much of their power to max-pooling layers combined with convolutional layers. Max-pooling layers decrease the size of the feature maps and therefore lose information about certain locations in the image. E.g. if we use a 2x2 max-pooling operation we lose information about every other patch. This information loss increases for every new max-pooling layer. As a counter measurement we can change the max-pooling layers to output several fragments that together cover the entirety of the image. This is shown in Figure 2.18 where a 2x2 filter is represented by 4 smaller maps that combined performs max-pooling on all parts of the input fragment. We can then combine the final set of fragments in different ways to represent the feature map that would be outputed at any patch in the input image. The authors of the paper used this to create a per-pixel segmentation method that could segment a 512x512 image in 15 seconds running on a non-optimized matlab CPU implementation. In comparison a normal sliding window approach on the same task running on highly optimized Cuda code on a graphics card took 492s which is 32 times slower. This goes to show how much time can be saved by only extracting features from the input image once. Implementing a method using extended maps in Cuda could result in a real-time segmentation algorithm.

2.5.2 Deep Feature Flow for Video Recognition

The SSD paper states: *"Note that about 80% of the forward time is spent on the base network (VGG16 in our case). Therefore, using a faster base network could even further improve the speed, which can possibly make the SSD512 model real-time as well"*. Deep Feature Flow [46] is a method for replacing the slow per frame computation in video segmentation and object detection. This method takes advantage of the spatial

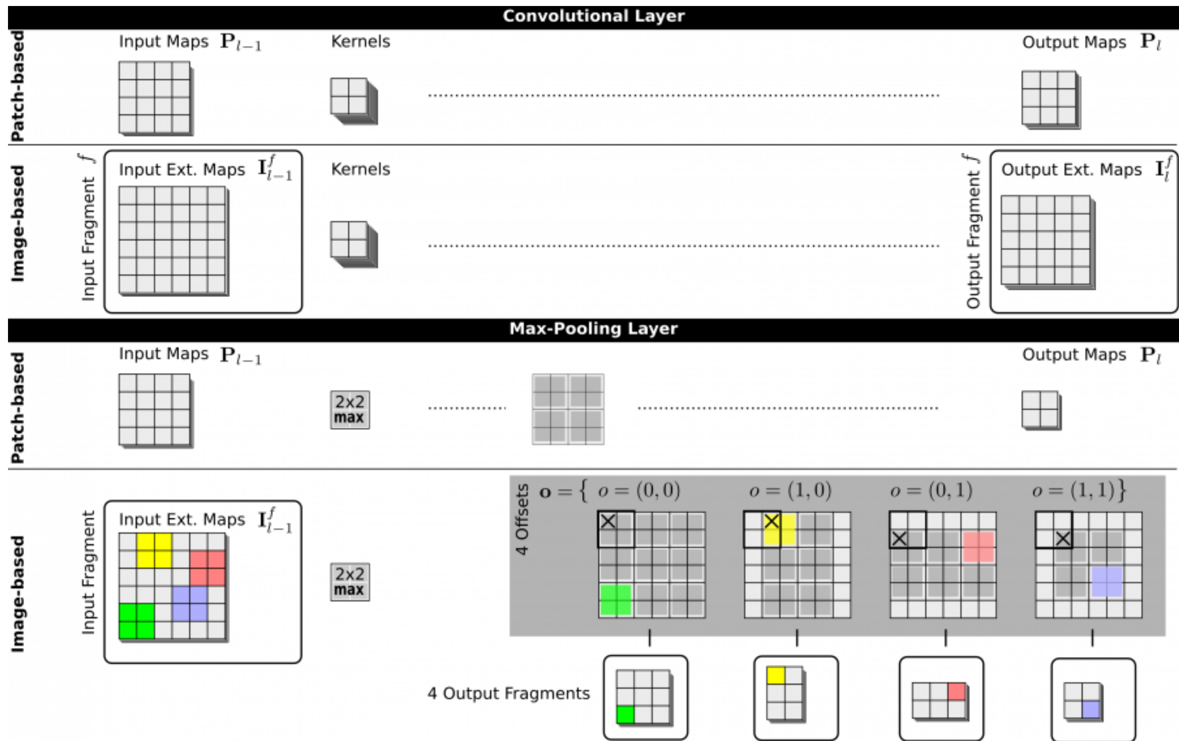


Figure 2.18: Patch-based and image-based forward propagation techniques for convolutional and max-pooling layers.

relationship between each frame in the video. Instead of calculating the feature maps for every new frame in the video from scratch it uses the feature maps at time t in combination with optical flow to *estimate* how the feature maps at time $t+1$ will look like. The new feature maps are calculated by running a Feature Flow Network instead of a standard CNN like VGG-16 or Inception v3. This proves to be much more efficient since the main time consumer of most detection and segmentation networks is feature extraction.

The first frame at t=0 is calculated as it normally would in a per frame solution. The feature maps for the frame at t=1 is calculated by first finding the optical flow between frame 0 and 1, then propagating it with the feature maps extracted at t=0 through a given propagation function. Optical flow can be calculated in a variety of ways but the default in the paper is a neural network approach called *FlowNet* [10]. This improves the performance of a ResNet-101 based detection method from 4.05 FPS to 20.25 FPS. The tested network architectures are well documented but no code seems to be available yet. It also seems that this solution can be sped up even further at the cost of accuracy.

2.6 Recurrent neural networks

Recurrent neural networks (RNNs) are based on a network structure that feeds its own output into itself again with or without the addition of extra data. This gives the network a few very powerful traits. RNNs are very modular and can run for an

unspecified number of iterations, accept an unspecified number of inputs and produce an unspecified number of outputs. The neural network approaches we have looked at so far all have a strictly fixed sized input, a fixed sized number of operations and a fixed sized output. Even the detection networks have a fixed sized number of detections in each image, we just segment out all the ones that have a low confidence value. The second trait is RNNs ability to form a memory. The network has an internal state that changes every time an input is passed through which makes it able to “remember” what it has done previously and build on it. Recurrent neural networks have had huge success with problems that are naturally sequential like language and video processing, but can also be used in processes that are not naturally sequential. We will take a look at a few examples.

One of the fields that could benefit from RNNs loose coupling is object tracking, and Deep Traking[31] is trying to prove just that. It performs tracking by estimating how the sensory output will look like in the next stage. The model they use is very simplified compared to the one we use in the regard that its input data is a simple 1 dimensional vector with LIDAR readings. It detects obstacles in front of the LIDAR sensor and translates this into a 2D image. It then performs tracking by estimating how the 2D image will look like in the next step based on knowledge about the way the objects in front of it moves. This is almost impossible in our case because we use a 2D image as input with a lot of noise. In addition we have a moving camera which disrupts the flow of the dynamics within the frames. It would be almost impossible to estimate how the full image would look like in the next frame because of all the different variables that effect the outcome. The only way it might work was if the noise were removed through e.g. segmenting out the robots. This would however require a lot more computation. The final nail in the coffin for this method is that it requires a steady time step between each frame to be able to make predictions. Our algorithm will run on a system that must handle more than one program at the time. Even though the other processes will run on the CPU of the Jetson TX2 card we must be able to handle minor delays between frames. Many tracking algorithms like DeepTracking will have a hard time making accurate predictions without a steady interval between frames. Other methods like [28] are more robust in this regard. It even solves the problem of data association and trajectory estimation, which is very promising. The only problem is that the field of tracking using RNNs is still in its infant stage and even the most promising methods still lack in terms of accuracy compared to state of the art tracking approaches.

Chapter 3

Methods and implementation

3.1 Detection network methods

The research goals of this thesis can be achieved in a number of ways so first we have to consider all the different options and find out which methods to pursue further. In this section we will go through the methods and how they can be combined to reach these goals. We will review the deep learning methods mentioned in Chapter 2 to see how they can contribute as either stand alone solutions or in combinations with other methods. Some of these methods were not discovered in the first reserach phase of the thesis which means the conclusion of this section changed during the course of the process. We will separate between the methods that were chosen to be pursued in the initial stage and the methods that were gradually introduced later that improved on those choices. The goal of this section is to find a solution that can perform detection on a single Jetson TX2 with input from four cameras simultaneously. Ideally we would like to achieve at least 4 FPS on the Jetson system to get an updated detection for each camera once every second.

3.1.1 R-CNN based detection networks

We can clearly see in Table 2.2 that even though a lot of improvements have been applied to the first R-CNN[12] implementation to create Faster R-CNN[37], it is still rather slow. 7 frames per second on a high end GPU will most likely result in about 1-2 FPS on a Jetson TX2 which is too slow. There are however two ways to improve the solution to make it viable. One way is to extract a smaller number of region proposals without dismissing any of the proposals that actually contain robots. The other course of action is to implement a faster way to classify the regions. Considering that no one has created methods that achieve this even with the large focus R-CNNs have had in the research field it is not very likely that we will be able to make an improvement that speeds up the method enough to reach the required FPS limit.

3.1.2 YOLO

YOLO certainly meets the speed requirement to perform in real-time and according to its creators it has a very descent accuracy on larger detection datasets like VOC[9] and COCO[24]. The main concern regarding YOLO is its accuracy when detecting small objects. The paper admits it lacks in location precision and the image is only divided into a 7x7 grid where two detections are extracted from each cell. The final network might have to detect multiple robots overlapping each other which could become an issue. One option to fight this is to increase the number of cells to something larger than 7x7, but this has the potential of increasing the speed of the forward pass by a critical amount. Decisions like this will have to be reevaluated when we know more about the accuracy of YOLO on the robot detection task. This is however one of the methods we would like to implement for further testing.

3.1.3 Fast segmentation network

The main reason segmentation networks are a part of this discussion is their ability to find parts of the input image that contain objects of interest in a single run of a network. We see in Table 2.3 that the networks are not as fast as we would have liked, but DeepLab has the most potential with about 9 frames per second on a high end GPU. DeepLab i especially inclined because we can remove the CRF step in the process to make it slightly faster. In addition to that we could scale down the input image size to speed it up even more. By implementing these tweaks we might get a segmentation process that can meet our requirements in terms of speed even on a Jetson system. So far the YOLO approach has deemed itself the most promising in terms of speed to accuracy but there are some lingering questions about its localization precision. We have therefore considered a combination of segmentation and location extraction to be a valid candidate as a final solution alongside YOLO. This does however require a fast way to extract robot positions from the segmentation image. Different methods for that will be discussed further in Section 3.2 when we analyze methods for creating a training dataset.

It was with these two main candidates we started creating the training dataset that will be described in Section 3.2 Other methods like SSD and YOLOv2 had at this point not gained much attention in the research field of object detection which meant they were not discovered until later in the process. A result of this was a heightened focus on creating both a segmentation dataset for DeepLab as well as a detection dataset for YOLO. The implications of this will be discussed further in Chapter 5.

3.1.4 SSD

The segmentation networks approach was mainly a topic when this thesis had not yet found any other detection networks than YOLO that sought out to perform detection on an entire image in one step instead of sifting through a large set of candidate positions. It seemed we were not the only ones that consider the convenience of such an approach, hence the creation of the SSD network. It captures exactly what we

wanted to implement but in a more direct approach which creates detection directly from the feature maps instead of separating the process into steps of segmentation and location extraction. Table 2.3 shows that SSD is both faster and more accurate than both YOLO and most likely any segmentation based solution that can be created from DeepLab. The full source code is also available online which makes it quite simple to train and test. SSD is a prime candidate for robot detection and the final goal of this thesis will be to enable training and testing of SSD with different input sizes to see which setup best achieves fast and accurate robot detections.

3.1.5 YOLOv2

YOLOv2 is, according to its creator, even faster and more accurate than SSD. The only problem with this approach as it relates to this thesis is that it was released after the research on YOLO was complete, so v2 wasn't discovered until the very end of the thesis. The source code for the project is however easy to set up so it will be tested and reviewed as a possible final solution.

3.1.6 Deep feature flow extension

The deep feature flow method proposed in [46] is not a standalone solution, but can improve the efficiency of a network like SSD which utilizes a base network to create feature maps. Section 2.5.2 described how a FlowNet[10] solution can find the feature map of a frame by combining the optical flow in the image sequence with the feature map from last frame. This can speed up object detection in video sequences as long as the base network is run for every few iteration to make sure the feature map is up to date with all the required features that might have appeared in the newer frames. In the paper they use a step value of 10 which means every tenth frame is generated by the slow but accurate base network. The video that is used for benchmarking is shot in 30 FPS which means that the base network has to run $30/10 = 3$ times per second to make sure the feature map stays updated. Lats say the final result of this thesis manages to run at 12 FPS on a Jetson TX2, then that leaves 3 FPS for each of the four cameras. The base network would then have to be run at each frame to make sure the feature map stayed updated and there would not be any time left for the FlowNet to run. However it might still be worth implementing if the required step size turns out to be different when performing detection on input from a somewhat slowly moving drone or if the final solution achieves a higher frame rate than 12 FPS. If we assume that a detection can be performed twice as fast using the FlowNet compared to the VGGNet model used in SSD then we only need a SSD frame rate of 18 FPS to achieve 24 FPS in a combined solution because there will be enough time to run the FlowNet once after every VGGNet detection. VGGNet will take 1/18 of a second to run while a FlowNet will be half of that at 1/36 of a second. $12 \times 1/18 + 12 \times 1/36 = 1$ second.

This is an extension rather than a full solution and will therefore be considered after the detection solution has been implemented to possibly improve the speed and obtain a higher FPS count.

3.1.7 MMOD and dlib

The MMOD algorithm and its CNN implementation in dlib has some really promising features, but also a few of concern. The most obvious being that it looks for objects within bounding box of a predefined size. A robot detection algorithm needs to be able to detect robots of a wide range of sizes which means that the MMOD based network will have to run multiple times for each image at different scales to achieve accurate localization and bounding box size. This could really hurt performance speed. The implementation of this method will be reevaluated when we know more about the general performance of the other methods like SSD have been tested.

3.1.8 Conclusion

The initial conclusion of this section was to try both the YOLO network as well as a combination of segmentation and location to see which performed the best. At the time YOLO was the fastest end-to-end solution we could find and was therefore a prime candidate for a final solution. The only hesitation resided in the possibility of a segmentation network in combination with a fast location extraction being even faster and/or more reliable. Because of the fast moving research field we had to adapt halfway through the process when SSD was discovered and selected to be the most promising option. SSD will be thoroughly tested as a stand alone solution and reviewed on how well it is able to detect robots in real-time with limited access to computational power. This will be the primary real-time detection objective going forward. YOLOv2 was introduced very late in the process and testing it will thus be a secondary goal to see how it compares to SSD. The number three priority will be to evaluate whether dlib and deep feature flow are worth implementing.

3.2 Detection dataset methods

Now that we know approximately how we want our final solution to look like we can start addressing what it requires to run optimally. The first step in creating any sort of detection network is to create a large amount of training data that represent as many realistic real world scenarios as possible. We know that we want to create an approach that takes camera images as input and outputs either a segmentation image or a set of detections. Most neural networks are developed and tested using one of the many standard datasets from various image processing contests like VOC2012[8] or ILSVRC2015[40] (ImageNet 2015). However, segmentation and detection of ground robots of different colors with an occasional pole mounted to them is far from a standard task and will require a self-made dataset of robots and their location from various angles and directions. There exists a considerable amount of images taken during Ascends participation in the IARC competition last year that will be very similar to the scenarios we want to prepare the detection network for. Figure 3.1 shows an image extracted from the dataset. However, none of the existing images contain information of any robot locations. The initial focus of this thesis will therefore be on a ground robot detection solution in which detection accuracy is far more valued than computation speed in

order to create a reliable training dataset using images from last years competition. This will in turn be used to train the methods discussed in Section 3.1. The creation of a reliable robot detection dataset will be a large part of this thesis considering the importance of accuracy in this stage and the lack of existing methods that could have sped up the process.

Figure 3.1: Example of a typical image recorded at the 2016 IARC competition.

The most straight forward approach would be to simply label all the images manually. However, this would take a very long time and be very tedious so the focus will be on creating an automatic solution that uses computer vision approaches to reliably label the images. We will first review a variety of basic object detection methods that have been used in the computer vision field for years to see if we are able to quickly create a detection set. These usually rely on some form of pattern recognition either in the shape of edge detection, color matching or feature matching. However if these methods fail we will concentrate on killing two birds with one stone and create a two step detection method that first segments out the robots from the image and then find the specific location of each robot based on the segmentation image. This will leave us with a dataset of segmentation images as well as a robot detection dataset which allows testing of both approaches.

3.2.1 Template matching

The most straightforward solution would be a feature detection method like Harris corner detector[16] or scale-invariant feature transform (SIFT)[26] to extract local feature descriptors from the input image and use an algorithm like RANSAC[11] to match them with descriptors we know represent a ground robot. There are two reasons why this method is likely to fail. First, there has to exist a very general set of images to search through for matching descriptors. These have to be general enough to enable matching all types of robot from all possible angles without being so general that it spawns false detections in the background surrounding the arena. Second, when the objects we are looking for are so small compared to the entire image we might not get

a large enough set of descriptors of the robots to properly identify them without also including a very large amount of unwanted descriptors. This can slow down the process significantly and also have a negative effect on the detections. As the final nail in the coffin for this approach: last years Ascend NTNU team did experiment with feature detection methods and the results were very poor. That is part of the reason why this thesis is in collaboration with Ascend in the first place.

3.2.2 Background subtraction

Another viable type of object tracking that is used in a vast variety of application is based on subtracting frames at different times to extract objects that move between frames. We know that the ground robots will be moving most of the time and could possibly be extracted form the fixed ground grid underneath. This approach is usually only used when the camera position is statically positioned because it makes the process of tracking which parts of the image are moving and which are stationary straightforward. We know that the cameras onboard the drone will be moving, so for this solution to work we would have to track the movement of the camera and subtract it from the movement of the entire image to extract only the robots movement. This approach does introduce problems. First, robots that are in the process of turning will appear still in relation to the ground. Second, this solution will at best create a segmentation image where ground robots are separated from the background. Segmentation images can be created more reliably in other ways which will be considered next.

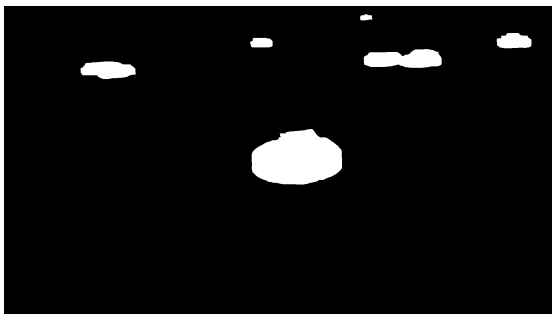
3.2.3 Sliding window segmentation

A more primitive alternative to using a multi object detection network is to train a much simpler classification network to learn the difference between ground robots and background. This can in turn be used in a sliding window operation as explained in Section 2.4.1. This solution will presumably not be fast enough to be implemented as a final solution but will hopefully prove reliable. Its main use would be to create a dataset of images where the robots and the background are separated. The separation between background and robot should be a manageable task for most standard deep classification networks as they have proven able to perform very well on more general classification tasks like the ImageNet challenge[40]. This solution will still require a large amount of training data, but this data is much easier to create compared to detection data as we can simply take still images of robots from different angles and use them directly. The steps of this approach will be to first create a CNN robot classification network, then use that network as a sliding window to create a segmentation image, then lastly retrieve the robots locations from the segmentation image (which will be discussed further in Section 3.2.4).

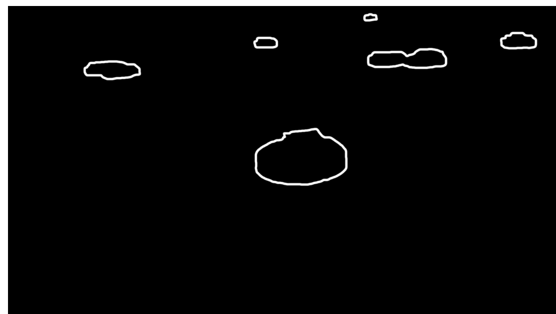
3.2.4 Location extraction from segmentation images

Segmentation images are valuable because they only include the parts of the image that contain objects of interest. This means that we do not have to worry about unforeseen color combinations or patterns that might occur in the background that can create false positives. Extracting location data from a segmentation image should therefore be simpler in cases that have a large potential for noisy background and uncertainty about exactly how the objects looks like from every angle. However segmentation image also tend to be a very simplified version of the original image. The only information remaining is whether or not a group of pixels (which this thesis will refer to as a “blob” from now on) are part of an object or not. The methods we are interested in tries to use this to their advantage. They search for specific shapes in the image and thrive on the fact that the only information available is the shape and size of the different blobs. Finding the location of blobs that only contain a single object is rather arbitrary. The real challenge presents itself when there can be more than one object hiding in a single blob and we have to be able to detect it. We will quickly take a look at a few viable options to get a sense of how location extraction from segmentation images could be realized.

(a) Original image



(b) Ideal segmentation image



(c) Edge detection image

Figure 3.2: The original image in (a) with its corresponding manually created ideal segmentation in (b). Image (c) shows the detected edges of the segmentation image.

Generalized Hough transform:

The first approach we might consider is a generalized Hough transform that is set to detect robot shapes within the image. This would require an edge detection method to allow the Hough transform to focus on the edges alone. Figure 3.2(c) shows the segmentation image in (b) after a Sobel edge filter has been applied to it. The next step is to define a shape that the generalized Hough transform should look for in the image. This could become a problem. The generalized Hough transform can be specified to be rotation and scale invariant, but the shape of the object it looks for still has to be reasonably specific to properly detect objects. In Figure 3.2(c) there is no single shape that perfectly captures the robots. They are all oval shaped, but the robots in the back are stretched more to the sides than the one in front. We also see that the two entangled robots to the left can easily be mistaken for a single larger robot. It is therefore hard to create a general shape to search the image for.

Meanshift and Camshift:

Meanshift is an algorithm that works to find a window of maximum pixel density in images representing probability distributions. It is typically used in combination with a histogram backprojection which is used to find the individual probability for each pixel in an image. This is just like the segmentation image created by the sliding window in this thesis but the histogram backprojection works similarly to performing template matching for each pixel and is therefore much less reliable. The Meanshift algorithm is started in a section of the image that contains more than just black pixels. It has a constant sized search window and for each iteration it moves its center to the area of maximum pixel density. This process and how it would work on a typical robot segmentation image is illustrated in Figure 3.3. One of the constraints of Meanshift is its use of a static window size that will have difficulties matching the robots correctly when it is unaware of the sizes they will be.

One way to avoid the restriction of the static window size is the implementation of a Camshift[3] (Continuously Adaptive Meanshift). This is in many ways an addition to the Meanshift algorithm meant to adjust the size of the final output window. The first step is to apply Meanshift in a normal fashion. Next, it sets the size of the window equal to

$$s = 2 \times \sqrt{\sum_x \sum_y W(x, y)} \tag{3.1}$$

where $W(x, y)$ represent the pixel value at position (x, y) in the window W. It then performs Meanshift again on the adjusted window size and repeat the entire process until convergence. The result is a self adjusting window that finds the local maximum pixel density.

The Camshift algorithm might have a hard time separating between some of the more entangled robot segmentations but it is quite simple to set up and test and should be considered a possible solution. The Meanshift is in most cases less adaptable to the Camshift but it might just be perfect in the case of robot detection when there exist an estimate for how large a robot will be at any point in the image. If this is true we

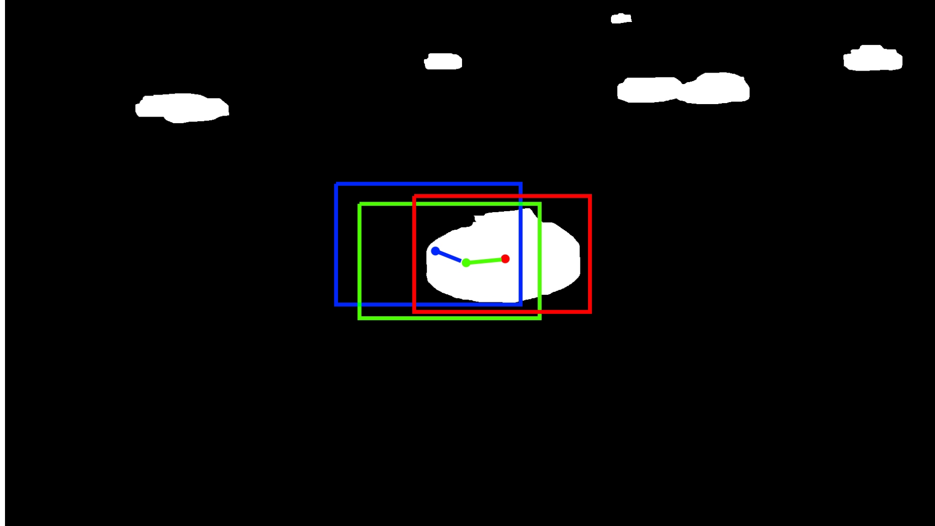


Figure 3.3: The initial image is the ideal segmentation of Figure 3.2(a). First the blue dot is defined as the starting position for the Meanshift algorithm and the blue rectangle is its search area. It finds that the average position of all white pixels within the window is located a little down and to the right from its current location. It then moves its center to this position and transitions to the green state. The new green search area is not perfect either and the center of maximum pixel density is still a little to the right. It moves again and enters the red state. When searching the red search area it finds that the red center is almost perfectly positioned at the center of maximum pixel density and the algorithm therefore terminates with the location of the red rectangle as its return value. The window now nicely fits around the robot and can be used as a bounding box representation of the robot.

might just be able to create a perfectly fitted search area and enable a more accurate location.

Large sized filters:

The expected output of the location extraction process is the center positions of each robot. So the objective of the search is to find the local centers of mass in each segmentation image. This can be achieved by applying something like an averaging filter that is the same size as the robot it looks for. This is pretty much like evaluating every position in the image like the Meanshift algorithm and then run an evaluation on all of them to get the bigger picture. However the filter approach is more adaptable because of the different types of filters that can be applied. This does however require a way of knowing the sizes of the robots that the filter is trying to expose the centers of. We will get beck to whether this is possible in Section 3.3.1.

3.2.5 MMOD and dlib

The fast learning abilities of the MMOD loss function could be a great feature to inspect further. By manually creating a small dataset of detections we could train the dlib CNN to at least a certain degree of accuracy. It would then be tested on a number of new images where the results get manually evaluated and minor mistakes get corrected quickly and used for further training. There are no speed requirements in this step either so a large number of scales can be run for each image to create the perfectly fitted bounding boxes. This method was unfortunately discovered right after the dataset was created, otherwise it could have proved a far simpler approach to robot detection that didn't require a large detection dataset to train.

3.2.6 Conclusion

Considering the assumption that standard methods for object detection like template matching are not sufficiently reliable in detecting small scale ground robot, we have to create the dataset in some other fashion. The process could be simplified by separating the problem into a segmentation part and a location extraction part as described in Section 3.2.3 and 3.2.4 respectively. This approach benefits from creating both segmentation data as well as detection data which could be useful considering that a segmentation solution has the potential to be just as good or better than an end-to-end detection network (see Section 2.4). Camshift and Meanshift are both interesting methods that should be tested as a possible simple solution to the location extraction problem. If the results return fruitless then we will explore the large sized filters to enable a more controlled extraction of the pixel mass centers.

The late discovery of dlib means it will not take part in the creation of the dataset. However it might prove much more potent then the solution chosen here and will therefore be discussed as a possible improvement to the total process in Section 5.3.

3.3 Detection dataset implementation

3.3.1 Robot segmentation

The aim of this section is to carry out the segmentation methods methods that were concluded to have the most potential in Section 3.2.6.

Creating a CNN robot classifier

The solution we want to focus on is based on a simple classifier that can categorize a 64x64 image as either a robot or not. The classifier would be used to determine where robots appear in a larger image by sliding a 64x64 window over it and map the classification result between 0 and 1 as probability of whether or not the window contains a robot. Because of its intended use in a sliding window approach we want the

CNN to more specifically classify whether its *center pixel* is part of the ground robot or not and will train it accordingly. This solution will presumably not be fast enough to be used as a final solution but will hopefully prove reliable. Its main use would be to create a dataset of segmentation images where the robots and the background are separated.

The selected architecture for this CNN classifier is the AlexNet network. There are several reasons to why it was chosen. First, it has proven itself able to differentiate well between a large number of classes [21]. Even though many of the CNNs mentioned in Section 2.2 outperforms AlexNet when it comes to accuracy it should prove more than precise enough for the binary classification task of ground robots. Second, compared to the other networks in Section 2.2 it is not very deep. The reason this is an advantage is the small image size we want to classify. With only 64x64 pixels as input we have to modify all of the other networks drastically to avoid reducing the image to nothing when running it through all the max pooling layers. Third, this network will be run a very large number of time for each image and the speed of AlexNet compared to GoogleNet, VGGNet and ResNet (see Table 2.1) will improve the computation time. Finally, AlexNet was a ground breaking CNN back in 2012 and is therefore already implemented in almost all existing deep learning frameworks. It is simple to set up and will require almost no time to start the training process.

With the network architecture settled we shift focus to the acquisition of training data. Deep learning famously require a large amount of training data to properly converge to a decisive solution that is still general enough to avoid overfitting. The goal of the network is to reliably classify two types of images: robot images and empty images. The dataset containing empty images need to represent a wide variety of scenarios to make sure that all parts of the arena are classified correctly including the background outside of the 20x20m grid. Figure 3.1 shows a typical image from a real world scenario and how cluttered the image can be. The image is taken from the dataset described in 2.1.5, which also contains several images without any robots in visible range. These images can simply be split into smaller 64x64 pixel images and be used as training data. We will also introduce some overlap between neighboring windows to make sure we capture the different features we want to learn at different positions within the extracted image. It is important to create this data from images that capture a large number of different scenarios that could present itself during a live run. We will therefore select images that are expected to produce the greatest challenges to correctly classify for every possible window. This will make up the *empty image* part of the dataset.

Next, we need to establish a data set of robot images. This time we do not have the luxury of a large base of existing images to separate into thousands of unique samples. There is no trivial way to reliably extract only robot images from the IARC data set without performing the very time consuming process of manually cropping them out, so we have to find a better alternative. One possibility is to take a series of images of the different types of ground robots with varying camera angles and backgrounds to make the network learn what the essence of a ground robot looks like. A set of ground robots are available to the Ascend NTNU team and can easily be photographed in any setting and lighting. The network should be able to distinguish robots from empty images regardless of the surface the robot stands on which makes this a reasonable

approach to creating a dataset. Additionally, this will produce high resolution images which can easily be scaled down or modified to fit almost any network.

Approximately 400 images have been taken of the different types of ground robot to capture them from every possible angle. In the specialization paper [33] leading up to this thesis we applied different pre-processing techniques like blurring, scaling and rotating the images in an attempt to make the dataset larger and more general. The intention was to improve the overall performance without having to take even more images. Several methods like different kinds of cropping and rotation were tested but blurring was the only pre-processing technique that had enough impact to be used further. As a result we ended up with a dataset that consisted of 1600 robot images and 68000 empty images. The ratio between the two might seem uneven, but we have to consider that the empty images must cover a much larger number of scenarios than the robot images and will cover more than 95% of the images on average.

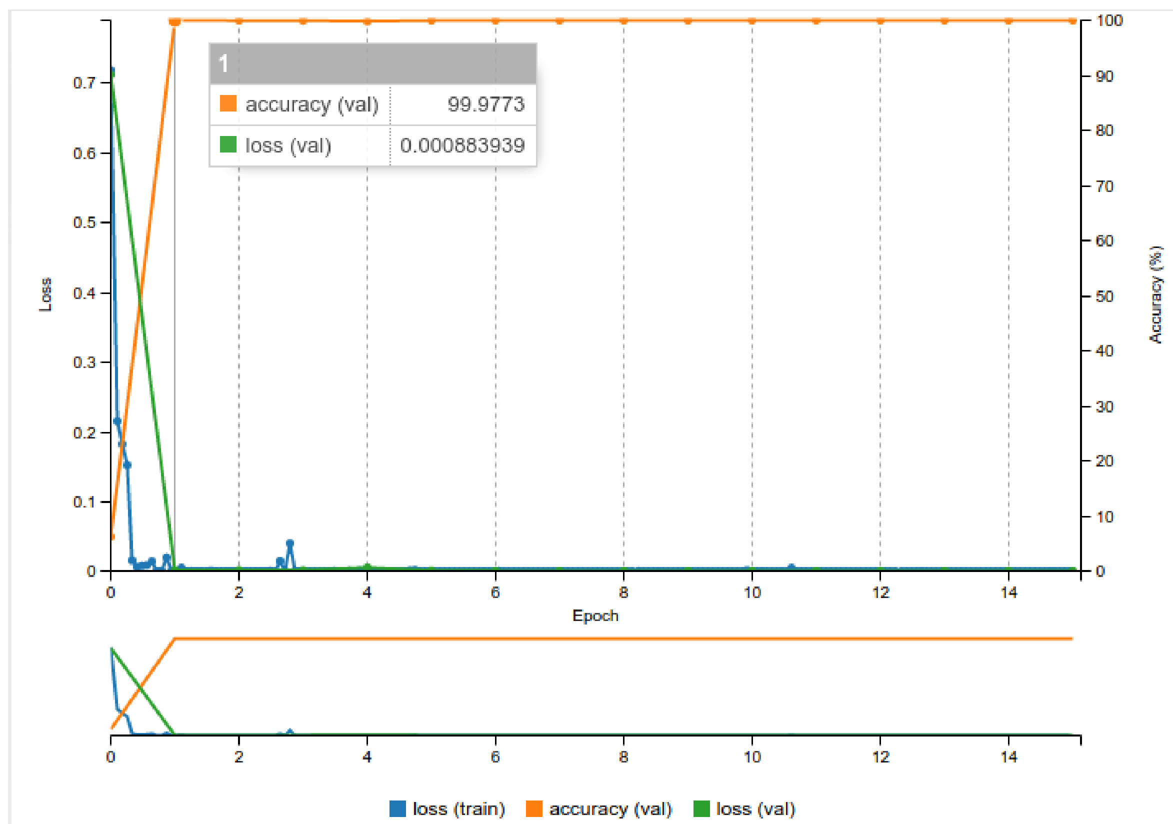


Figure 3.4: Graph from the learning procedure in Nvidia DIGITS showing the first 15 epochs of AlexNet training to distinguish empty images from robot images.

The next step is to train AlexNet on the created dataset to see how well it is able to learn the general visual concepts that make up ground robots. The dataset from the specialization paper that performed the best was randomly split into a training set and a validation set before being used as input to AlexNet. The result is shown in Figure 3.4. It shows a very high test score of 99.97% on the independent validation set after only one epoch. This could indicate that the network works extremely well extremely fast, but might also be a reason for concern. We are clearly not overfitting the images selected to be in the validation set which is usually a good thing. The problem is that

when it converges so quickly to an almost perfect solution it could suggest that the training and validation data are too similar without necessarily representing the real world scenario. If this is the case then we have a problem even though the network is not overfitting the training data because it could be overfitting the entire dataset as a whole. This was another topic addressed in the specialization paper and the answer had both a positive and negative side to it. The negatives being that the dataset of robot images was general enough for any robot placed directly in the middle of the image, but it struggled in a number of edge cases. The positive side to this is that AlexNet proved able to learn the visual model of a ground robot very quickly and will most likely be able to improve its edge case classification as long as it is given some edge cases to train on.

Bootstrapping

Bootstrapping is a method that can be used to localize and extract edge case scenarios when performing object detection. It is used in several papers like [39] and [36] to describe a system able to find errors performed by the network, label them correctly and put them back in the training set. A system like this would be able to find the edge cases we have struggled to define and and use them directly to train the network. Realizing that the final use case for this neural network is an image segmentation algorithm we can create a method for quickly labeling a large number of windows without having to go through them separately. First, we manually create what we call a binary *ideal segmentation image* which represent what we would like to see as output. An example of this is shown in Figure 3.2. This works as a ground truth and is used as a standard that the network output can be compared to. We can now match every window that is classified at a given location in the full sized image to the same location in the ideal segmentation. If the classification done by the network is different from the ground truth then the window is saved as either false positives of false negatives and added to the dataset for the next iteration of training. For this bootstrapping method to be tested, however, we still need a proper way of performing the sliding window. The topic of bootstrapping will therefore be revisited after the creation of the sliding window in Section 3.3.1.

Creating a sliding window

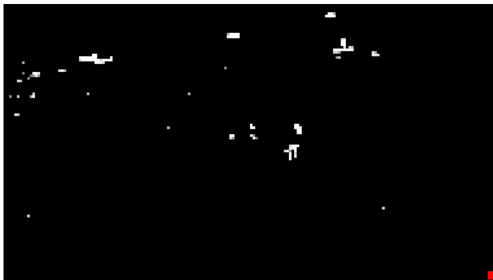
Now that we have a CNN robot classifier we can start to apply the sliding window approach to the input images. Even though AlexNet is the fastest network with the required accuracy it is far from fast enough to run for every pixel in the 1920x1080 pixel input image. The window will need to stride a few pixels to the side for each iteration. This will lead to a lower resolution output image which will be fine for most classifications, but could potentially be a problem for some of the smaller robots in the image. Another problem introduced by this approach is the consideration of an appropriate window size. Different objects in the image will have different sizes and require different sized windows to cover them completely. More specifically, the robots further away will be smaller than the robots closer to the camera. We will consider two different methods to counter this problem. In the specialization paper [33] we tried

to run the operation multiple times with different sized windows to detect robots at different sizes. For n different window sizes we got n segmentation images as output, each showing where robots of the approximate size of the window was found in the image. These n segmentation images were then combined and normalized as the final multiscale segmentation image. The result of this approach is shown in Figure 3.5. We see that (c) can find the smallest robot in the original image that is hardly visible in the very back. However, this also detects a large amount of false positives in the foreground where robots are much larger than the window size. (d) and (c) both detect the medium sized robots, but still produce false positives whenever the window size is too large or too small. (f) is mostly correct, but is far less accurate in detecting the smaller robots than the smaller window sizes. Finally, (g) shows the the combined segmentation on top of the original image to show which parts of the image it thinks contains robots. It certainly manages to emphasize the locations of the robot but at the cost of also producing a lot of noise. Most importantly, a large number of false positives are detected in the final output as a consequence of the various window sizes being completely disproportionate to the objects it is looking for in certain areas of the image. This approach also favors the medium sized robots as they can be detected in several different window sizes while the smallest and the largest robots only fit a single size each. From this we learned that independence of scale would be a very useful attribute as the classification network performs substantially better when the windows are approximately the same size as the robot its trying to detect. It is also worth mentioning that this is a very time consuming process as it needs to completely scan the image for every one of the n window sizes used. The combined problems of noise, uneven weight distribution, neglection of robots, and time consumption indicated that an approach using constant window sizes was not reliable enough.

All problems with the solution so far with regards to the sliding window can be solved by using a window size that correctly fits around the robots its looking for in any given position in the image. We have seen that AlexNet is able to learn to differentiate between ground robots and background reasonably well so far and with the help of a bootstrapping system it could be near perfect in its classification as long as the size of the robots are approximately constant. One way to realize this is to make use of the sensory data from the IMU and the LIDAR mentioned in Section 2.1.5 to make assumptions about the diameter of any ground robot given its pixel location in the image. The LIDAR gives us information about the altitude of the drone and the IMU gives us the roll and pitch of the drone which can be used to find the angle of any of the side cameras. The cameras are all positioned stationary at a 66 degree angle from the ground up and they all use the same focal length and pinhole structure. The drone does not make many sudden movements and the variation in roll and pitch are therefore minimal. This leads to the assumption that the images are taken more or less horizontally to the floor which again means that the size of a given robot is only dependent on its y position in the image. We will however use roll and pitch to fine tune the camera angle because even small changes here will have a large effect on the outcome. With a combination of these measures we can calculate the width of a single pixel in meters for any y value in the image. This is shown in Equation 3.2 where P is the width of a pixel in meters in the real world.

$$P = \frac{h}{(cos(\alpha) \times f) + (sin(\alpha) \times dy)} \tag{3.2}$$

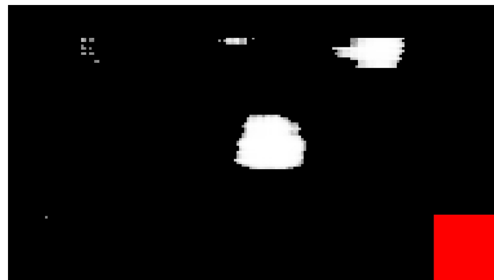
(a) Original image



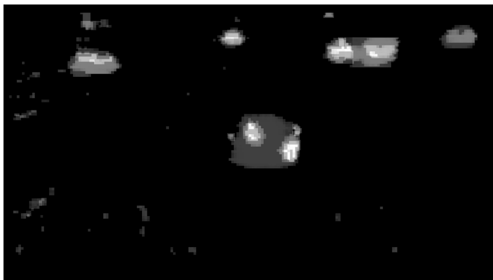
(b) Window size = 32

(c) Window size = 64

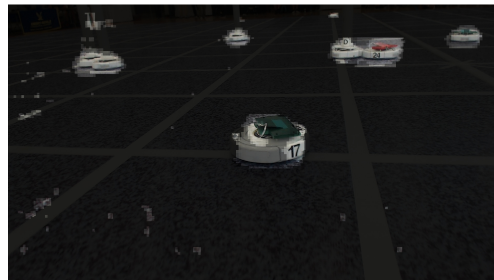
(d) Window size = 128



(e) Window size = 256



(f) Final normalized and combined output



(g) Combination of image (a) and (f)

Figure 3.5: (a) is a typical image taken during the 2016 IARC competition. The images (b)-(e) were created by running a sliding window of size 32, 64, 128, and 256 respectively for every tenth pixel in (a) and classify them as either a robot (white) or not a robot (black). In the lower right corner they all have a small red square to show the size of the window relative to the size of the robots in the image. (f) is a segmentation image created by adding the pixel values of (b), (c), (d), and (e) together and normalizing them. (g) is a combination of image (a) and (f).

The hight h is measured in meters above the ground, α is the angle of the camera from the ground up, f is the focal length of the camera, and dy is the distance in pixels from the halfway line in the image. The images from IARC 2016 has a resolution of 1920x1080 which means the halfway line is at $1080/2 = 540$. All ground robots also have a diameter of 34cm which means we can find the expected pixel diameter at any position in the image by Equation 3.3 where d is the diameter.

$$d = \frac{0.34m}{P} \tag{3.3}$$

The diameter d will be referred to later in this thesis as the *expected robot size.* This opens up to a sliding window that is much more customized to the size of any robot at any position in the image. Instead of scanning the image multiple times with fixed window sizes we can have a single run where the window size is the same as the expected diameter of a robot given the y coordinate we are searching at. The full description is shown in Algorithm 1.

Algorithm 1 Self adjusting sliding window

Precondition: Classification function *classify(image,x,y,d)* *map(class, segImage, x, y, diameter)* which maps a classification between 0-1 to the output segmentation image.

```
 1: function SLIDINGWINDOW(image)
 2:     y ← first y that gives diameter > 25        ▷ Smallest robot is 25 pixels wide
 3:     d ← diameter at y from Eq. 3.3
 4:     while (y + d) < image.height do
 5:         step ← d/4                              ▷ Each step is 1/4 of the window diameter
 6:         x ← 0
 7:         while (x + d) < image.width do
 8:             class ← classify(image)             ▷ class is a number between 0-1
 9:             windowClassificationList ← class
10:             segImage ← map(class, segImage, x, y, d)
11:             x ← x + step
12:         y ← y + step
13:         d ← diameter at y from Eq. 3.3
14:     return segImageorwindowClassificationList
```

It is worth mentioning that the *step* value is set to a quarter of the window size to make sure that each patch of the image is inspected from more than one perspective. The current value was chosen because of its favorable ratio between computation time and accuracy. There are two possible outputs from this method which will both be discussed later. The segmentation image results from this approach is presented in Section 4.1.1.

Improving the classifier

The fact that the classifier is able to run in a sliding window fashion and produce segmentation images opens up to a whole new way of evaluating its performance. We can now make use of a bootstrapping method that is able to detect errors made by the network in a much faster and more reliable way than a human could by going through the classifications manually. The main reason to do this is to filter out mistakes done by the classifier like the ones seen in Figure 4.1(d) where part of the mounted poles have been classified as robots just above some of the robot blobs and the segmentation boundaries tend to be misshaped. The first step in this process is to create a set of ideal segmentation images just like the one in Figure 3.2(b). These can be created manually and will serve as an overall ground truth depicting which pixels contain robots and which do not. They are easy to create and it will not take long to make only a few by hand. In addition we also create a set of images without any robots in them that all have ideal segmentation images that are completely black. The empty images are mainly used to improve the CNNs overall performance on the varying background outside the arena. They are not meant to target the edge cases specifically. We can see in Figure 4.2(b) that this is necessary to avoid false detections in areas that are outside the arena. The next step is to let the sliding window run on the original images with the use of the currently best performing CNN classifier to label every window. For each window we can now oversee whether the classification was correct or not by comparing it to the same window in the ideal segmentation image. This allows us to extract every window that is incorrectly classified, label them correctly and add them to the classification dataset. The new dataset will hopefully help the classifier to perform less background mistakes as well as distinguish between robot and background in edge cases. The bootstrapping process can also be performed several times on the same collection of images if the result is not good enough after only one iteration.

3.3.2 Robot localization

Now that segmentation images can be created reliably through the self adjusting sliding window technique, it is time for the actual localization of the robots. This step is necessary for both a segmentation based solution and an end-to-end robot detection network. A segmentation solution would have to perform this step as quickly as possible to avoid halting the detection process. This would be a vital part of the solution. However, considering the benefits of using an end-to-end detection network discussed in Section 3.1 we will mainly focus on creating a method that is reliable rather than fast.

The reason localization of robots is considered hard is that the segmentations of the robots are more or less just blobs of white pixels in areas where the classifier was really sure it saw a robot. The case is simple when these blobs only represent a single robot but the case is quite often that robots are close together and partially occluded by each other. Extracting two robots from these kinds of blobs without creating false positives in the single robot blobs will be the real challenge.

The final goal of this process is not just to find the centers of the robots, but to

create bounding boxes around them which is the standard approach to create a dataset for any of the major modern detection networks. The conversion from center points to bounding boxes will therefore be vital. However, the information required by Ascend is only the point position of every ground robot in the frame at any given time. This means that the bounding boxes do not need to fit perfectly as long as they are accurate enough for the detection network to learn where they should be placed. As long as the detection network is able to reliably position the boxes at the approximate same place relative to the center of every robot it finds then we can easily extract and utilize the exact center point afterwards.

Meanshift and Camshift

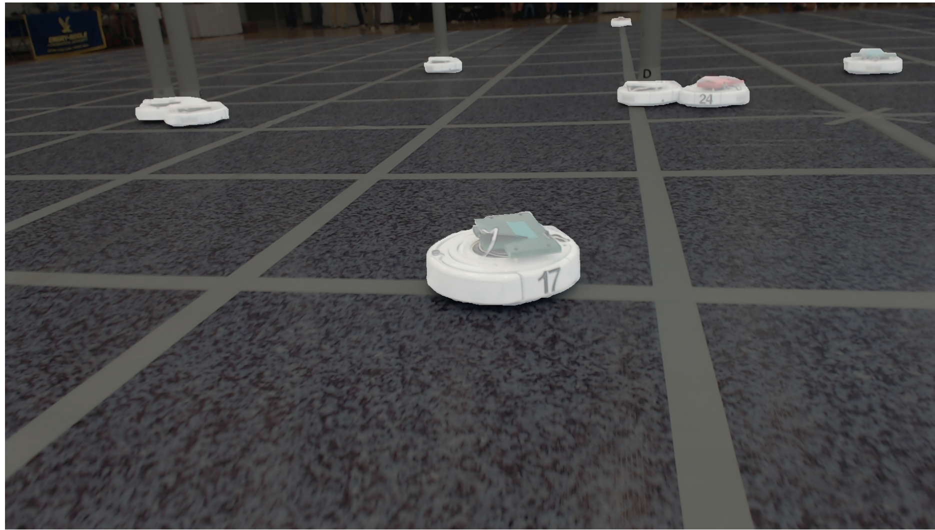
First we would like to try the Camshift algorithm to see if it is able to find the robot positions and sizes based solely on Equation 3.1. The second solution we want to explore is an custom version of Meanshift that uses a window with a constant ratio between its height and width, however the width is always calculated in the same manner as the self adjusting windows used with the sliding window. The initial placement of the window will be at a random location that holds a white pixel. When the Meanshift has converged we consider the window as either a robot detection or not and all pixels within the window are set to zero. For a window to be considered a valid detection it has to contain threshold percentage of white pixels that will have to be experimented with. The results of the two methods using Camshift and the custom Meanshift will both be reviewed in Section 4.1.2.

Robot sized filters

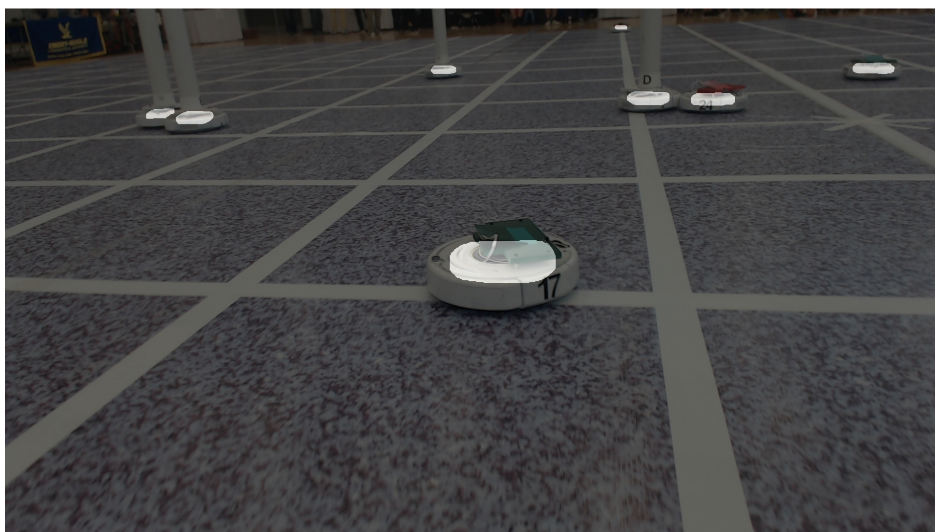
So far we have seen that the unknown number of robots hiding in the "robot blobs" can be hard to detect even for humans when only examining the blob by its shape. One option is to use what we know about the expected size of the robots at different positions to tell if a blob is too big to hold only one robot. The next step is therefore to use large sized filters to perform heavy blurring of the image to expose local maxima in the segmentation images. Blurring is usually used as a noise reduction measure in images but can also be re-purposed to find the center of ground robots. The underlying idea is to blur the areas of the segmentation image holding robots with a kernel sizes that fit the expected robot sizes to create peaks in the center of the blobs. These peaks can then be extracted as local maxima and used as candidates for ground robot centers.

The first attempt will use a simple box filter to get the average of the surrounding area. The size of the filter will be dynamic and change along with the y coordinate to always equal the expected robot size. Box filters work by equally averaging over every pixel within the square confines of the filter. The hope is that because of the filter size being equal to the expected robot sizes we will get a higher value in the middle of the blobs than on closer to the edge. In the simple case of blobs that only contain a single robot this will give us the center of mass which is most likely to be the center of the robot as well. In the case of blobs that are larger than the expected robot size we hope

to see a separation of mass centers that are easy to pinpoint and single out individually. The goal is to get a blurred version of the segmentation image as output where the center(s) of mass in each blob are represented as easily identifiable local maxima.



(a) Ideal segmentation.



(b) Center oriented segmentation.

Figure 3.6: These two images are a combinations of the original image from Figure 3.5(a) and different versions of segmentation. A rather standard robot segmentation is shown in (a) where the entire robot is separated from the background. In (b) we see a more center oriented version that only segment out the middle of the robots. This results in a much clearer separation between the different robots even though they are very close to each other or even partially occluding each other.

The results of the box filter method is shown in Section 4.1.2 and the discussion in Section 5.1.2 concludes that the process needs to run faster and at the same time be able to better distinguish between robots that are partially occluded by each other. We will consider a few changes to the process to address these issues.

First we will alter the detection network to be more specific in its classification of robots. In Section 3.3.1 we addressed the lack of edge cases that had been introduced to the classifier during training. This helped us improve the overall segmentation output to look more like the ideal segmentation we wanted to see. This did however turn out not to be such a great starting point for localization after all. The problem is that this leads to the rather large sized blobs that leaves no or little indication of where one robot starts and the next begins. Figure 5.1(d) and (e) in the Chapter 5 show that this is true no matter how well this boundary segmentation is performed. To counteract this we will retrain the CNN classifier to only label almost perfectly centered windows as robots and everything else as background. The goal is to get smaller robot blobs in the segmentation image which again are easier to separate by a filter afterwards. An example of the wanted segmentation image is shown in Figure 3.6. To create a CNN that classifies like this we will have to gather a lot more edge cases that represent the division between what qualifies as a robot center and what does not. One way to do this is to manually capture and add images that we think represent these cases well. This will be done to some extent, but most of the images will be created by using the already established bootstrapping procedure that has previously been used to improve on the accuracy of the classifier. The only difference between this round of bootstrapping and the last is that we want to accept fewer and more centralized robot classifications than before. This means that all the false positives that were found in our last round of bootstrapping are still relevant while the false negatives will have to be discarded because they might represent windows that are outside of the robot center. The starting dataset will therefore be the original dataset with the addition of the false positives found when bootstrapping the classifier. With the dataset in place we need to create the center oriented segmentation images. We have seen before that only a few segmentation images can result in large changes to the classifier so creating between 10-20 images will hopefully be enough without taking too much time to create manually.

This method will find a lot of edge cases, however some of the windows that are very close to the center border could be classified differently in similar cases due to human inaccuracy when creating the ground truth. This could confuse the classifier network more than it will help and we would therefore like to have a measurement of how correct a classification is compared to a small area around the center pixel. The result could then be thresholded to sort out the windows that are borderline correct and only extract the ones that are properly incorrect. How “correctness” can be measured will be explained after Gaussian filters with dynamic kernel size have been introduced. By retraining AlexNet on the new dataset we will hopefully get a more center oriented classifier that can be used in a sliding window to better separates the large robot blobs into smaller ones.

The second change we do is to the filter itself. Now that the classifier is more focused on the center of the robot we get a greater separation between robots even though their centers are positioned relatively close together. This means that larger blobs that contain two robots are more likely to be shaped like the centers of the two robots with a connecting bridge between them. In Figure 3.7(a) we see how this works in practice and that rather than looking like one large blob it now looks like two separate blobs that are overlapping each other. This feature can be exploited by

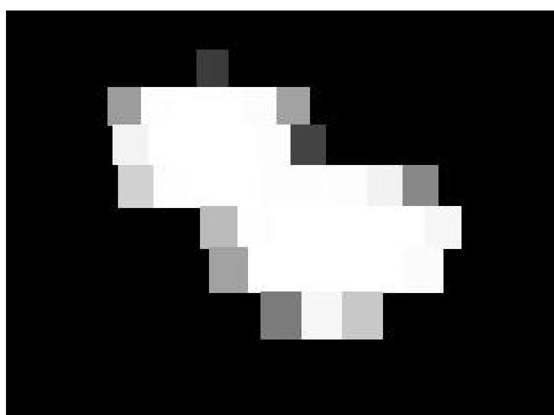
replacing the box filter with a Gaussian filter which is illustrated in Figure 3.8. The round shape of the weight distribution is preferable in this scenario because it will target round objects to a larger degree. This means that even though two blobs are overlapping each other we will be able to source them out. Figure 3.7(b) shows that as long as the Gaussian filter is fitted for the size of the individual blobs it will create higher intensity values at the centers rather than in between. By adjusting the sigma value we can make sure that not only the center pixels are contributing, but rather the entirety of the blob shape. At the same time we want the center to contribute the most. The goal is to create a bell over each robot in the shape of a Gaussian where the local maxima will represent the center point. The maximas can then be extracted.

With the notion of Gaussian functions in place we can take step back and a look at how it can be used in bootstrapping to evaluate the "correctness" of a classification. The first step is simply to apply the dynamic Gaussian filter to the ideal segmentation image and normalize the result between zero and one. A one will then represent a neighborhood where every pixel is classified as a robot while a zero is a completely empty neighborhood. This will blur its edges and give them values ranging from zero to one instead of the hard binary format of the ideal segmentation. In edge cases we can now find the correctness of an edge case by comparing it to the blurred classification. Cases that are close to the segmentation edge will have a value of approximately 0.5 while the cases further away will be closer to either one or zero depending on whether the surrounding pixels are robot centers or not. A threshold can then be set which requires cases to have a difference of e.g. 0.9 between classification and blurred classification for them to be categorized as a misclassification and added to the new dataset. The lower the threshold, the more edge cases are included. We can also experiment with separated thresholds for the false positives and the false negatives if we want to extend or reduce the center edge by a small amount.

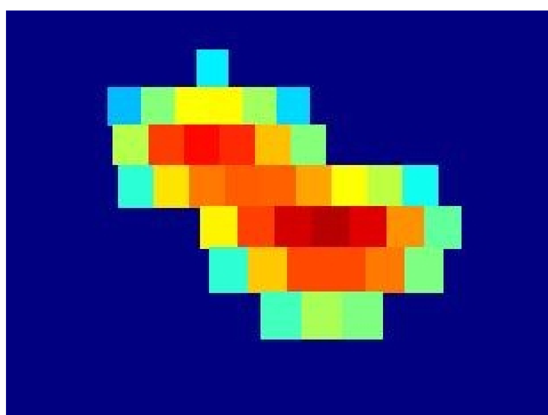
The third change relates to the resolution of the output image. The data outputted from the sliding window is in reality just a list of classifications for the different windows in the image. The output images are created setting each pixel to the value of its closest prediction. This conversion is mainly done to enable visual analysis of it. Rather than applying the filter to the full size image we can get approximately the same result by applying it directly to the classifications. Each image run through the sliding window will output between 8000-35000 window classifications depending on the hight and angle of the camera when the image was captured. In addition we can skip every window that is classified as empty because we can assume all robot centers are classified correctly. This brings the number of computations down to a few hundred per image. In comparison a full sized image consists of $1920x1080 = 2.073.600$ pixels. For each image we can therefore reduce the number of operations from 2 million to only a few hundred. The position of every window within the image is known which means it is easy to calculate the distance between each of them. For each window we can create the set D containing the distance d to every window within the expected robot radius r. The intensity value $I(d)$ for each window will then be given by Equation 3.4 where $G(d)$ is given by Equation 3.5.

$$I(D) = \sum_{d}^{D} G(d) \tag{3.4}$$

(a) Two overlapping robots.



(b) Segmentation of the two overlapping robots in image (a).



(c) The image from (b) after the application of a Gaussian filter.

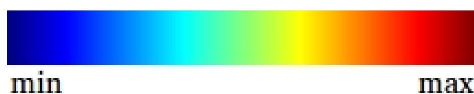
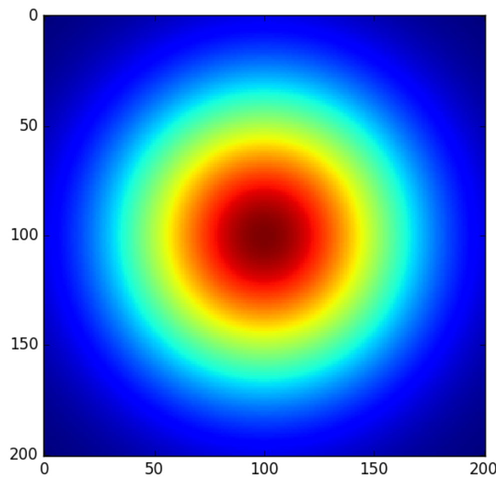


Figure 3.7: Image (b) shows a segmentation output that makes it easy to see where both of the robots within the blob are positioned. Image (c) shows how a Gaussian filter can be applied to the segmentation image in (b) to get high intensity values in at the center of the robots.

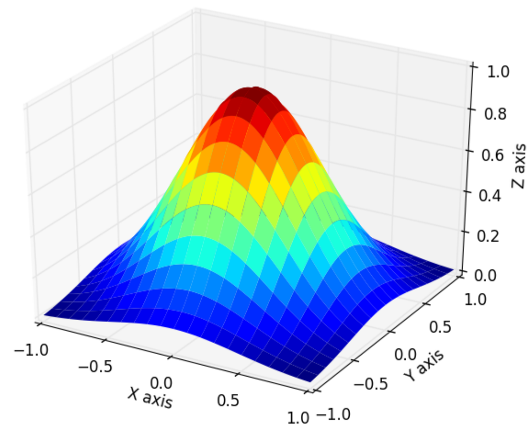
$$G(d) = e^{-\frac{1}{2}(\frac{d/r}{\sigma})^2} \tag{3.5}$$

By applying this method we will get an output that is also a list of window intensities which can be converted to an image or simply used as is. The results will all be presented as images while the robot localization algorithm will use the intensity list to avoid unnecessary additional computation.

Extraction of local maxima can be done in a number of ways. After the convolving the Gaussian filter over the segmentation image we will shift our focus to a global scale. The largest intensity value in the filtered image will be considered as a candidate robot and evaluated. If the point passes the evaluation then it will be saved as a center point, if not it will be discarded. Either way we will reduce the value of the point and every



(a) Heatmap of a Gaussian filter.



(b) 3D visualization of a Gaussian filter.

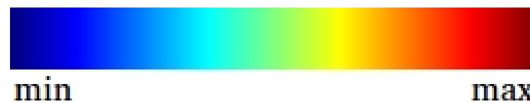


Figure 3.8: (a) and (b) visualize how a Gaussian filter looks like and how it will weigh different parts of the image. The sigma value is 3 in both images.

surrounding window within a radius equal to the expected robot radius. The next global maximum will then be evaluated in the same way until we reach a threshold intensity limit that is considered too low to represent anything of value. The output will be a list of center points where we know the expected robot size, and we can create the appropriate bounding boxes. The full process is describe in Algorithm 2 while the evaluation function and the reduction of the surrounding windows will be described in the next paragraph.

The first point to address is how to reduce the values of windows surrounding a potential detection point. The reason we do this in the first place is to not only remove the highest intensity window from the image but also remove the surrounding windows that are likely to be almost as high. If they were left in we would most likely get a bunch of overlapping detections for the first robot before even getting to any of the others. We are looking for local maxmas, so whenever one is found we have to reduce the local area to be able to find the next. One way to do this is to simply set every window within the expected robot radius to zero and be done with it. This would work if our evaluation only cared for high intensity values and accepted everything over a certain threshold. However this proves to create a lot of false positives because the imperfections of the classifier would leave single and quite high value windows just outside the removal radius that would then be evaluated as a detection later on. Another problem with this approach is that robots with their center positioned close to another robot would be incorrectly located just outside the other robots radius because its center intensity would be completely removed. What we really want to remove here is the Gaussian shaped blob that represented the robot we just evaluated. One solution is therefore to

Algorithm 2 Slow robot detection

Precondition: We need the following functions: $findWindowNeighbors(window)$, findMaxIntensityWindow(windowIntensityList)

```
1:  function RobotDetection(image)
2:      windowClassificationList = slidingWindow(image)          ▷ From Algorithm 1
3:      for window ∈ windowClassificationList do
4:          neighbors = findWindowNeighbors(window)
5:          windowIntensityList ← I(neighbors)                    ▷ From Eq. 3.4
6:      maxIntensityWindow = findMaxIntensityWindow(windowIntensityList)
7:      while maxValueWindow > threshold do
8:          if evaluateWindow(maxValueWindow) == True then
9:              robotDetectionList ← maxValueWindow
10:         reduceNeighborhood(windowIntensityList)
11:         maxValue ← findMaxValue(windowIntensityList)
12:     return robotDetectionList
```

reduce all surrounding windows by their Gaussian weight. So by using Equation 3.5 we can simply subtract $G(d)$ from the intensity value where d is the distance to the candidate point. This should remove the blob for a single robot completely without effecting any overlapping blobs too much which should solve the problem of removing close by centers. The problem of outliers will have to be solved using the evaluation function.

A high intensity value is not always enough for a window to qualify as a robot detection. The reason is that there might appear smaller groupings of false detections that must be removed without removing robots that the classifier had a hard time labeling correctly or robots that were partially occluded. This is mostly going to be a problem when evaluating outliers and occluded robots where the overlapping robot has already been detected and had its neighboring windows reduced. The proposed solution will for every candidate window see if the Gaussian distribution still holds even though some of the neighbors might be reduced. If most of the neighborhood had been reduced significantly then we can safely assume that we are looking at an outlier. At the same time this will allow partially occluded robots to pass because even though their intensity values might have been reduced they will still get a higher Gaussian score within its radius than any isolated high value windows will. An example of the process can be seen frame by frame in Figure 3.9 where blobs are evaluated, saved as detections and removed. The goal of this step is to create a more center oriented segmentation to better separate and locate robots that partially occlude each other. We will use center oriented segmentation images in the bootstrapping algorithm as a standard for how we think the result should look like, but the only result that really matter is the final robot localization that is extracted from the segmentation images. We will therefore evaluate the performance of the center oriented approach in combination with Algorithm 2 and its use of Gaussian filters to extract these locations.

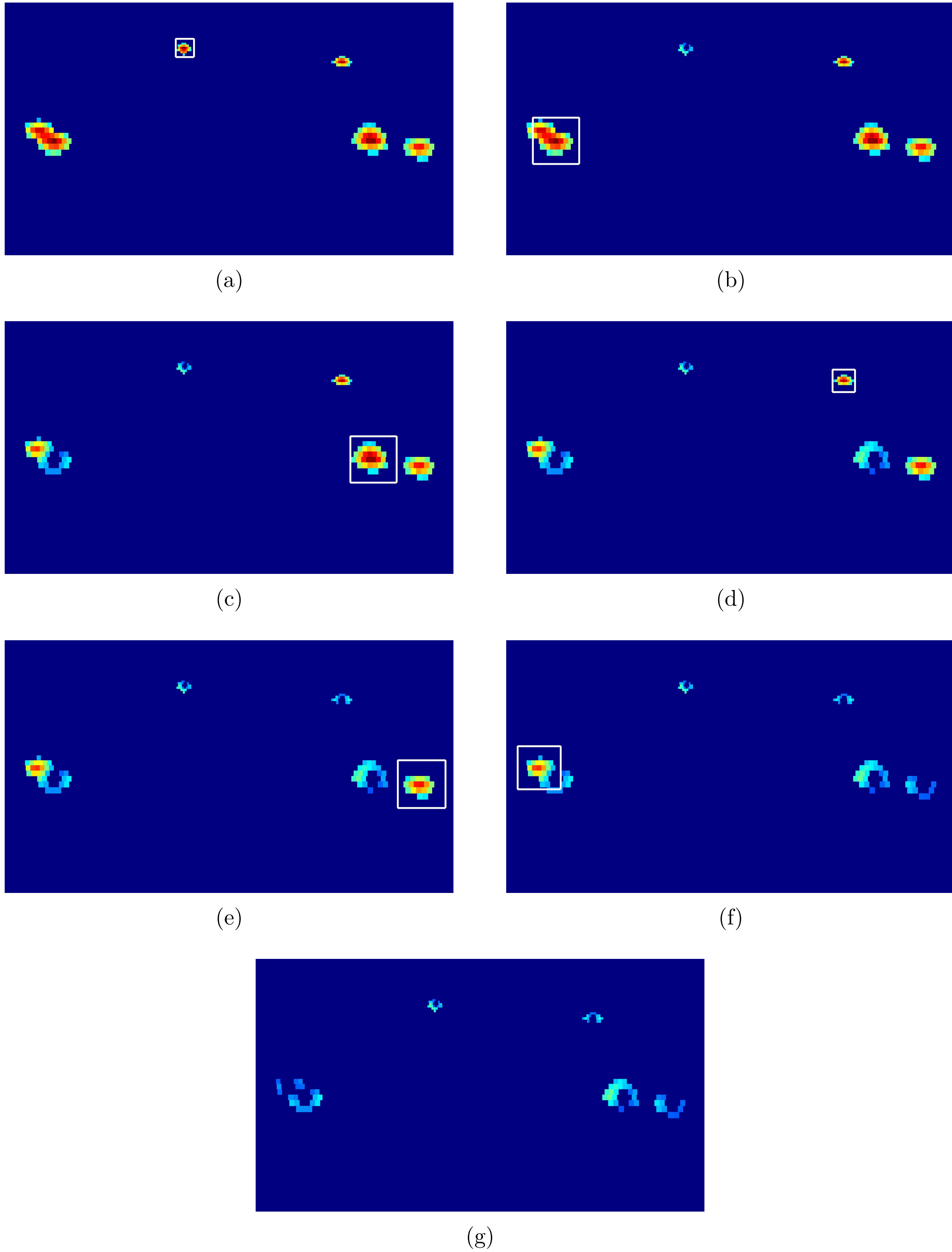


Figure 3.9: Each image represent a step in the final localization procedure. The boxes are drawn to show where the highest intensity values are found. Once they are evaluated we reduce the neighborhood and look for a new candidate detection. In (g) there are no more intensity values that exceed the threshold limit and the process is terminated. The candidates that passed the evaluation are saved as detections.

As mentioned in the introduction to this robot localization section we do not need the bounding box to be exactly at the limit of the robot in every direction as long as its center is always at the center of the robot. As of now we do have information about how wide the robots will be, but not how high. The hight will depend on the angle and hight of the camera as well as its relative distance to the robot. The hight of every robot could be worked out, but will not be necessary as long as the bounding boxes are fitted tightly enough at the sides. We will therefor assume that every bounding box is square and that the width and hight are always equally large. This will be a sufficient amount of information for the detection network to generalize on the visual traits of the robots and localize them. Modern detection networks are able to differentiate between a large amount of different objects and in our case it only has to recognize one. This gives us room to use training data that is not a hundred percent accurate in terms of the bounding box placement without hurting the final result. It will however not be a problem to adjust the height at a later stage should there be any indication of accuracy loss as a consequence of this assumption.

The full procedure

Up until this point we have describes the different steps of a slow robot detection algorithm that should be able to create a sizable robot detection dataset from the 2016 IARC images. For the sake of completeness and overview we would like to sum up the complete process in the following steps:

1. First, train a classification neural network to reliably differentiate between images containing a robot and images that do not. The network is trained on 64x64 pixel images and gives out a confidence value between 0 and 1 for each of the two possible classifications: Robot or Background.

2. Divide the full sized image from IARC 2016 into a fixed number of somewhat overlapping windows. The size of the windows depends on the windows y value in the image: The further away, the smaller the windows. These window sizes are estimates of how wide a ground robot will be at a certain distance to the drone given the hight and angle of the camera. As long as these measurements are accurate we can estimate the size of the window to precisely fit any given robot at any distance.

3. The windows now have different sizes depending on their y value in the image and must be resized to fit the already trained classification network which takes images of size 64x64.

4. Each window is run through the classifier to see if it represents a robot or simply background. These values are stored and can be presented as a segmentation image or a where white spots are the parts classified as robots while black represents background.

5. The results of each window are then blurred using a Gaussian function within a neighborhood with a size of the expected robot diameter to create local maximum in the middle of the robot blob.

6. Finally the windows with the highest values are extracted and evaluated as either a robot detection or a false reading. For every center window that is extracted we reduce the surrounding windows with the same Gaussian as before to avoid detecting the same robot multiple times.

This full process was run on every fifth image in the IARC 2016 image set which amounted to about 6000 images. This was done to prevent several images that look almost the same to enter the detection dataset.

3.4 Detection network implementation

3.4.1 Network training

With a detection dataset in place we can finally start training a the networks. The SSD network is a very complete solution that has everything we need included in the code. Testing the network is as simple as converting the dataset to a format it accepts, adjust the batch size and learning rate to account for the memory available on the GPU in use, and then simply start the training. The detection dataset will be split into a training set containing data from five of the runs from IARC 2016, and a validation set containing one final run. All the runs look rather similar to the naked eye, but they all contain different combinations of robot placements and flight patterns. If the validation set was created by randomly selecting frames from the entire dataset we would get two sets that contain almost exactly the same information with only minor changes between the frames. By separating the two sets by different runs we get a much better impression of whether the training process is overfitting or not because the validation set will only contain frames from a run that the network has never seen before.

The SSD code has a large set of preprocessing parameters that can be set to make the training process more general. As a first iteration of the training process we will not touch any of these as the SSD paper argues very well for why they are set the way they are. If there are signs that any of them can be set differently to improve the result after this first iteration then they will be tested next.

There are several adjustments that can be made to the structure of the network at a whole to not only change how well training goes but rather the ratio between speed an accuracy. By changing the input size of the network we can increase the speed by making it smaller and less accurate or make it larger to enable slow but accurate detection. The first step in the process will be to test the two sizes proposed in the SSD paper to see how well they both perform on the Jetson TX2 system. The two first networks to be trained will therefore be of size 300x300 and 512x512 respectively. In an attempt to find a middle ground between the SSD300 and the SSD512 we also create a version with input size of 420x420. This will have almost exactly The results are accounted for in section 4.2.1.

3.4.2 Adding color

Now that we have a network that performs well we can start to focus on improving it. Specifically we want it to learn the difference between the tree types of robots that exist within the competition. As of now there are no difference between green and red robots, but this will be of importance in future missions. Something that always will be important however is to locate the robots that have poles mounted to them so that the drone can avoid them during flight.

The only real change that needs to be applied to the dataset is the specific color for each detection in stead of just saying it contains a robot. Through this thesis we have become quite familiar with classification networks and how accurate they can be. The proposed method is therefore to train a classifier to differentiate between the different colors and run it for every already detected robot to label them with colors. The classifier can be realized by taking the now rather large dataset of robots cutouts, divide them by color and train an AlexNet classifier to differentiate between them. There is no need this time to for the classifier to be able to detect background as it already knows it will only get robot images as input. All that remains after the dataset has been relabeled into different colors is to adjust the number of possible output detections in the detection network and retrain it.

3.4.3 Network testing

Once the networks have completed training they can be installed on the Jetson cards and tested. The accuracy of the network has already been tested through the validation set in the training process so there already exist pretty good numbers on what to expect in that area. With that being said it is hard to tell exactly how well the network performs based only on mAP score, so a more visual will be done by running the network on the validation run to see how it performs.

Chapter 4

Results

This chapter will present the results that have been collected during the different steps described in Chapter 3 for the creation of a robot detection and segmentation dataset from scratch using simple images along with information about the height and angle they were taken. This includes going through the creation of segmentation images using a sliding window technique before taking a look at the outcome of different robot localization approaches. Next we will present the result of the training and testing on the several different network setups described in Section 3.4.1 and 3.4.3 respectively. This thesis is very much based in visual output and review of that output. This chapter will therefore include images and illustrations to present the result as well as links to videos that might give a more complete view of the behavior of different detection networks. A deeper discussion of the results will be saved for Chapter 5.

4.1 Detection database

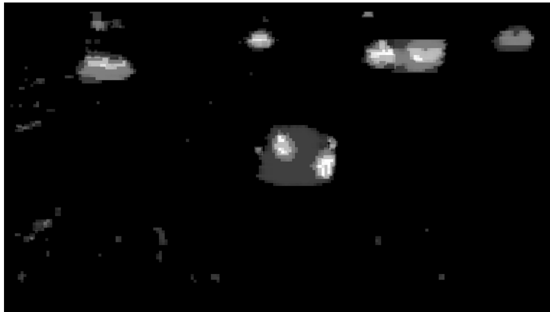
4.1.1 Robot segmentation

The two different approaches to the sliding window that uses constant sized windows and self-adjusting window sizes are shown in Figure 4.1(c) and Figure 4.1(d) respectively. The images were created as a way of visually comparing the different approaches to address eventual pros and cons they might have. When comparing the outputs to the ideal segmentation we get 96.98% for the fixed size and 98.17% for the self adjusting. Despite the increase in performance we still face mistakes in terms of classifying white poles and random objects outside the arena as robots like the ones we see in Figure 4.2(b).

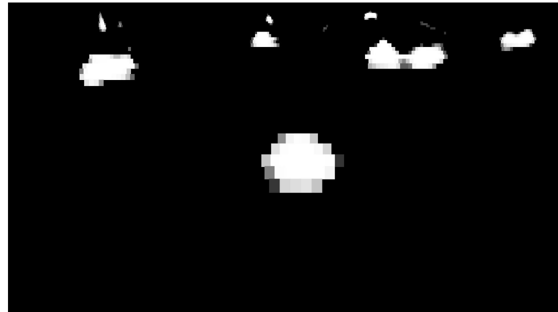
The process used to improve the classifier described in Section 3.3.1 was first focused on removing the misclassifications in the background before addressing the accuracy of the robot segmentations. The bootstrapping process was applied in increments and lead to minor improvements each time. The final result is shown in Figure 4.1(e) and required seven iterations where the last two had a few additional ideal segmentation images to utilize. The result was a 99.13% match with the ideal segmentation on an

(a) Original input image.

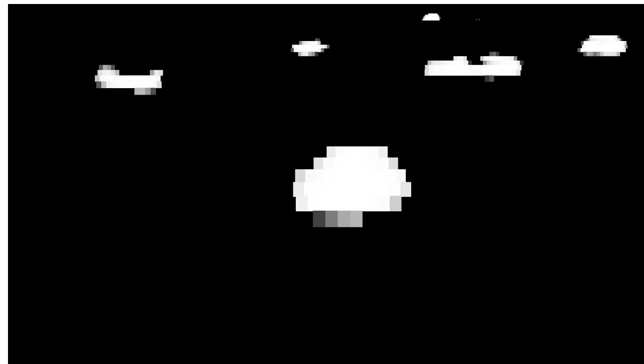
(b) Ideal segmentation of (a).



(c) Robot segmentation performed by the fixed size sliding window on (a).



(d) Robot segmentation performed by the the self-adjusting sliding window on (a).



(e) The final segmentation result after several iterations of bootstrapping using a set of hand made ideal segmentation images. Correctness percentage: 99.13%

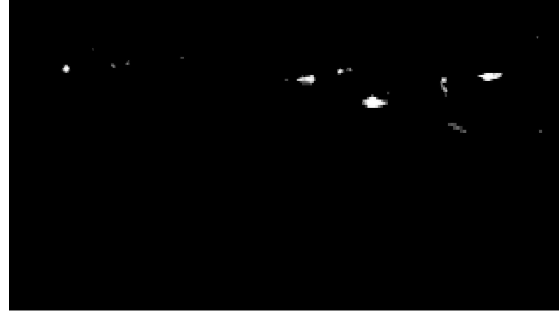
Figure 4.1: Comparison of the different segmentation methods proposed to enable robot detection.

image that was not included in the bootstrapping process.

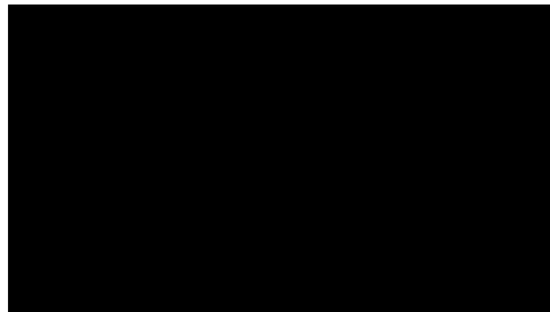
4.1.2 Localize robots

This section will review the results of the different localization methods that was proposed earlier in Section 3.3.2. In contrast to the normal grayscale representation of segmentation images used in this chapter so far we will now use a heatmap represen-

(a) Original input image.



(b) Robot segmentation performed by the the self-adjusting sliding window on (a) using the first iteration of the CNN classifier.



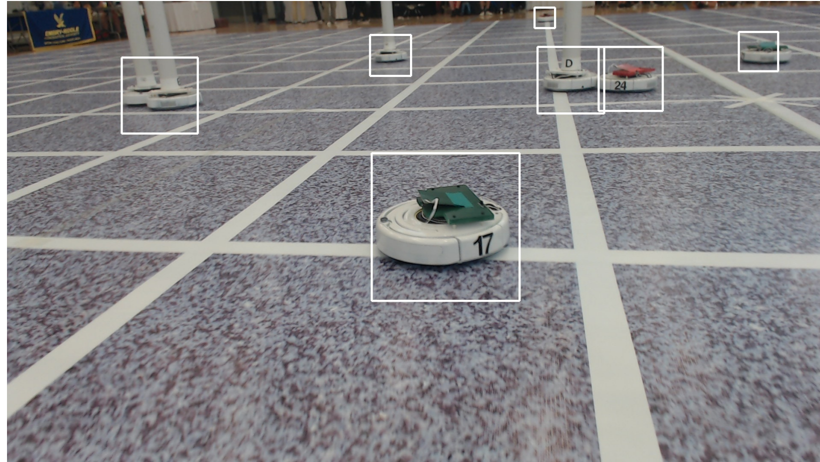
(c) Robot segmentation performed by the the self-adjusting sliding window using the fine tuned CCN classifier.

Figure 4.2: This figure shows how the CNN classifier in the self adjusting sliding window makes mistakes in images that contain a large amount of background noise. The first iteration of the classifier is shown in (b) while (c) shows the result after the final iteration of improving the classifier.

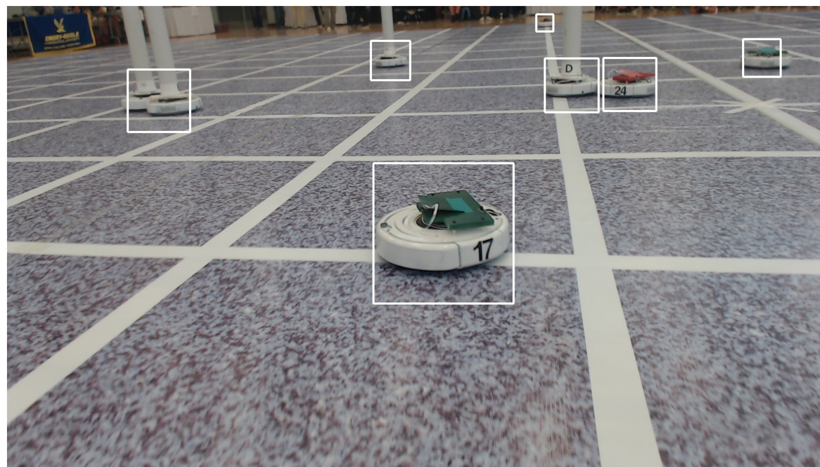
tation to better distinguish between small differences in the images. An example of this is shown in Figure 4.4(c) which also displays the colormap used to represent the different intensities.

Meanshift and Camshift

Figure 4.3 shows the output detections from the application of Camshift and Meanshift on the segmentation images. Camshift fits its detections around the entire blob while the custom Meanshift is limited to only use the expected robot size in its detections. They function well as blob detectors but have a hard time separating the overlapping robots from each other.



(a) Camshift applied to the ideal segmentation image in Figure 4.1(b).



(b) Meanshift applied to the ideal segmentation image in Figure 4.1(b).

Figure 4.3: Results from the Camshift and Meanshift algorithms attempting to extract robot locations from the ideal segmentation image in Figure 4.1(b).

Robot sized filters

The application of a box filter fitted to the expected robot sizes is shown in Figure 4.4(c) along with the colormap used. This shows that the filter approach is able to highlight centers of the robots but still struggles to properly separate robots that are overlapping each other. The next step is therefore to test the Gaussian filter approach to see if that can correct the problem.

Up until this point we have mainly used the image in Figure 4.1(a) to represent the results because it had not yet been used in any way to train the classifier and would therefore represent an impartial scenario. It was involved in the second round of center oriented bootstrapping which makes the classifier specifically trained to perform well on this image. To avoid overfitting the set of bootstrapped images we will instead use an image that has never been trained on.

The Gaussian filter will be evaluated in combination with the other methods proposed in Section 3.3.2 as a complete localization solution. The most interesting steps

(a) Original input image.

(b) Sliding window segmentation of (a).

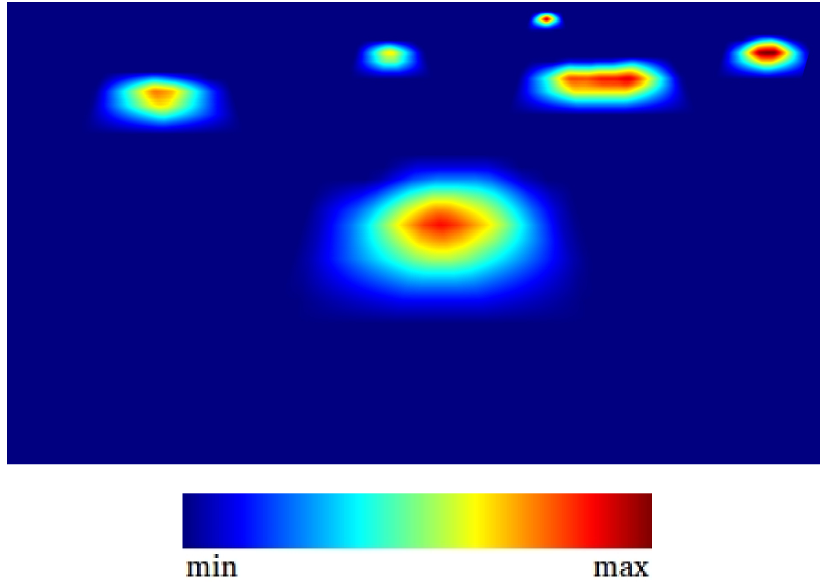


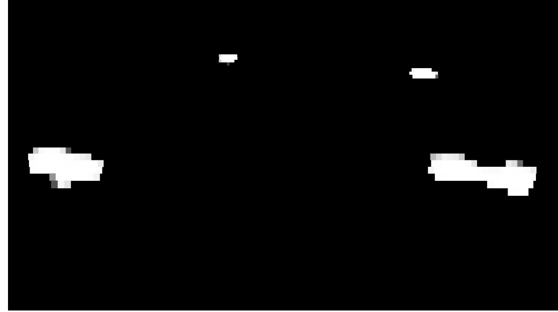
Figure 4.4: The full process of segmenting and applying a box filter to the original image in (a). Image (b) shows the sliding window segmentation, while the output of the box filter method proposed in 3.3.2 is shown in (c).

from the bootstrapping process has therefore been included in Figure 4.5 with its Gaussian counterparts in Figure 4.6 to see how well they are able to separate the different robot centers. This shows that adjusting the correctness threshold allowed us to create a segmentation output that properly divide overlapping robots into separate center locations when the Gaussian filter is applied to it. The detection output based on the final solution can be seen in Figure 4.7.

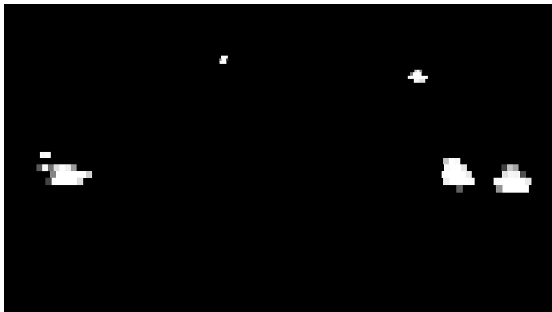
4.1.3 The final detection dataset

The full dataset was created by running the sliding window segmentation over 7100 images and the output was correctly able to detect all robots and localize them 83% of the time. The detector primarily had difficulties with images that were very blurry or had some error in the hight and angle meta data which led to incorrect robot size calculations. As a result the final dataset required a manual inspection to remove the erroneous images. This was unfavorable but the process only required a yes or no

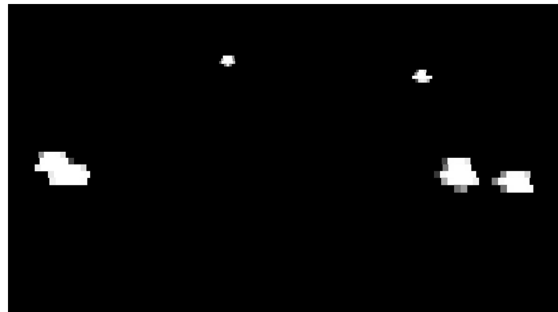
(a) Original image.



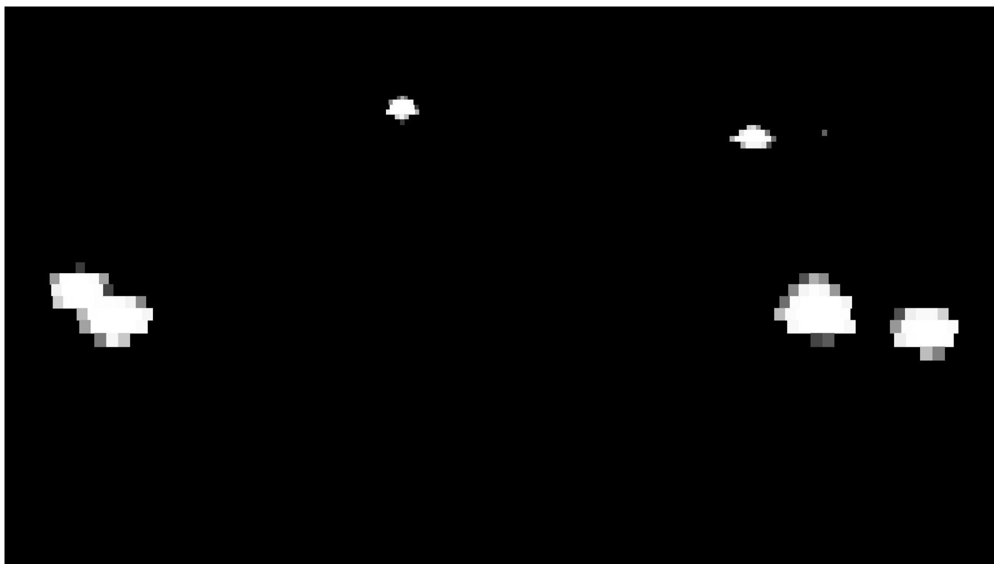
(b) Original segmentation output from the self adjusting sliding window.



(c) The segmentation result after two iterations of bootstrapping with the handmade center oriented ideal segmentation as the optimal standard.



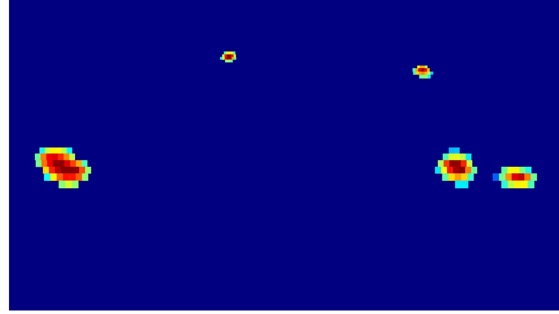
(d) Segmentation after two iterations of bootstrapping where the correctness threshold for false positives were set to 0.9 while the threshold for false negatives remained at 0.5.



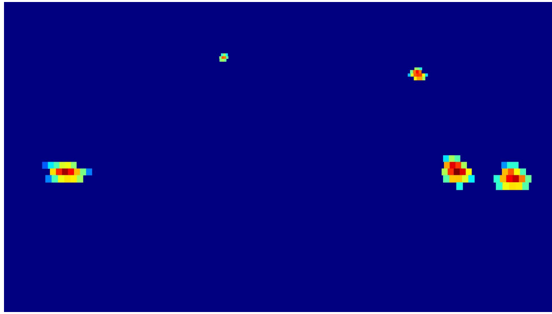
(e) Final and most successful segmentation output. The correctness threshold for false positives were set to 0.8 while the threshold for false negatives remained at 0.5.

Figure 4.5: The different steps of creating a CNN classifier that makes the process of extracting robot positions as straightforward as possible.

(a) Original image.

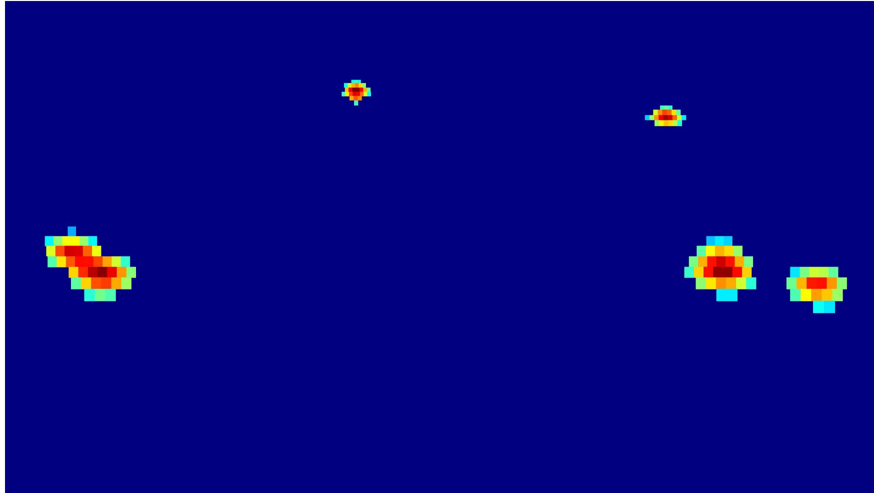


(b) Original sliding window segmentation output after the application of a Gaussian filter.



(c) Gaussian convolution of the overly centralized segmentation image in Figure 4.5(c).

(d) Gaussian convolution of the segmentation image in Figure 4.5(d).



(e) Gaussian convolution of the final segmentation image in Figure 4.5(e).

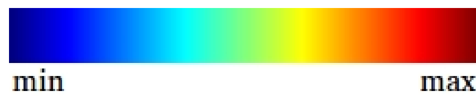


Figure 4.6: Illustration of the images in Figure 4.5 look like after the Gaussian filter has been applied to each of them. The intention is to make it simpler to separate the centers of overlapping robots for each step without neglecting any of the other robots. This is achieved in (e) which leads to the final detections shown in Figure 4.7.

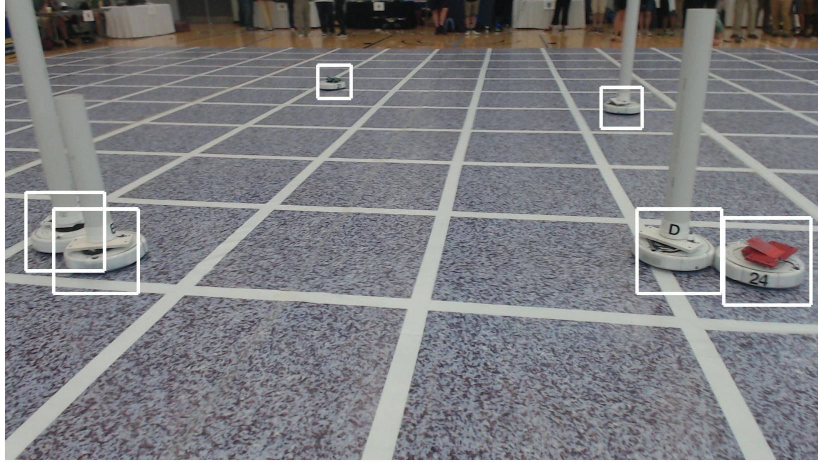


Figure 4.7: Final output of the localization algorithm.

answer for each image to whether it should be included or not. Some of the images that only had minor mistakes were corrected and added as well. The final output was a detection dataset of 6100 images with bounding boxes for all robots.

The next step was the addition of color. The AlexNet classifier had a 99.4% validation score after training and was therefore considered accurate enough to be applied without any manual inspection. The final output was a new version of the dataset with labels of the three different types of robot: tower, green, and red.

4.2 Detection network

4.2.1 Network training

After investing a significant amount of effort in creating a robot detection dataset we can finally review how an end-to-end detection network is able to generalize over it. The network was trained using a powerful computer with a state of the art Nvidia Titan X graphics card. Its 12GM memory capacity allowed for relatively large batch size and the 11 TFLOPs of processing power provided a much improved training time compared to most other desktop computers. The first network we wanted to was the 300x300 SSD with the reduced VGGNet as its feature extraction model. The batch size was set to 32 with a learning rate that started at 0.001 and decreased to 0.0001 after 10000 iterations. None of the other values in the SSD code were changed to avoid tampering with the combination that had seemed to work well for the creators of the SSD model itself. The model was run for a total of 16000 iterations which took 43h 14min and the result is shown in Figure 4.8. The graph shows that the network is able to learn throughout the training period without overfitting because the accuracy of the validation set is either increasing all the way until it settles in the final iterations with an mAP of 83.78.

The same process was run for a network with 512x512 input size. This required a smaller batch size of 8 and starting learning rate of 10^{-4} because of limited memory

on the graphics card. The model was run for a total of 32000 iterations and the result is shown in Figure 4.9. The complete training took a total of 24h 12min. The network increase its validation scores somewhat more sporadically than the 300x300 network but we see in the training graph that it too manages to learn throughout the process without overfitting and settles with a mAP of 90.55.

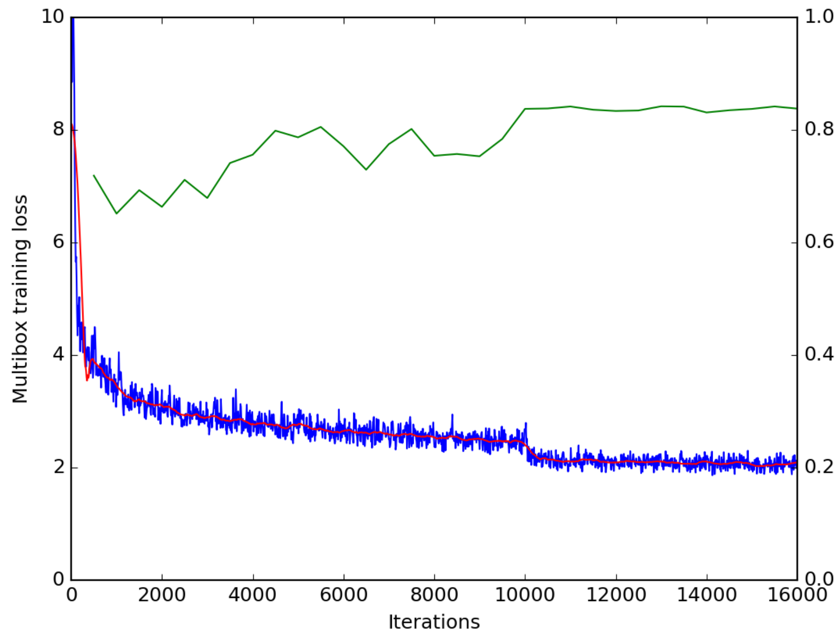


Figure 4.8: Training graph of an SSD network where the input image size is set to 300x300. The blue line represents the multibox training loss for every 10th iteration. The red line is a filtered average of this loss to better visualize if the loss has converged or is still decreasing. The final error after 16.000 iterations is about 2.0. The green line is the validation mAP calculated at every 500th iteration. The final result after 16.000 iterations is a mAP of 83.78. We see that after 10.000 iterations there is a drop in loss and a stabilization in the validation scores. The reason is that the multi step learning rate is dropped from 10^{-3} to 10^{-4} which makes the network able to settle on a more stable solution.

Next we perform the same training procedure on the colored dataset. The training graphs of the 300x300, 420x420 and 512x512 networks are all included in Figure 4.10. We see that the colored version requires a few more iterations to converge and that the final mAP are slightly lower. A more detailed view of the setup and the final result for each network can be seen in Table 4.1.

The last network to be trained is the YOLOv2 net of input size 416x416. The training graph is shown in Figure 4.11.

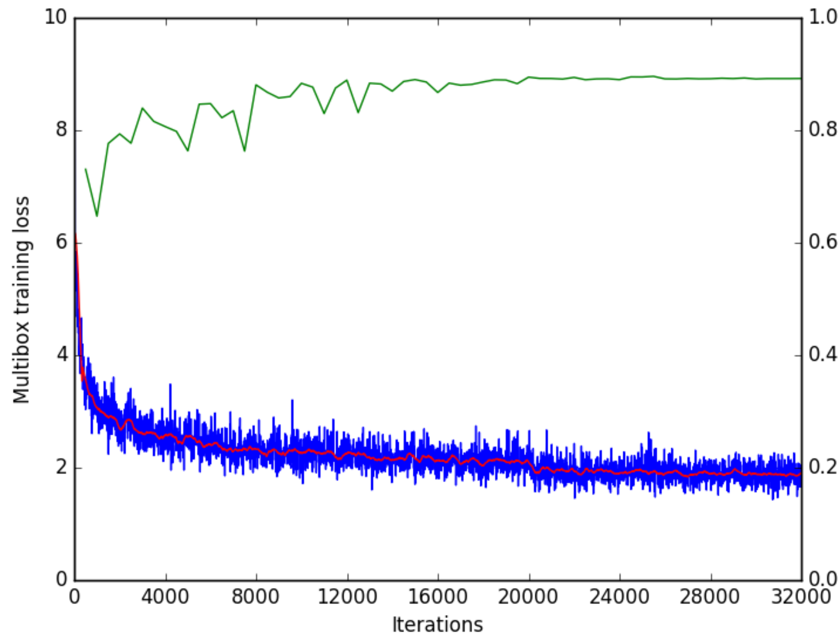


Figure 4.9: Training graph of an SSD network where the input image size is set to 512x512. The blue line represents the multibox training loss for every 10th iteration. The red line is a filtered average of this loss to better visualize if the loss has converged or is still decreasing. The final error after 32.000 iterations is about 1.9. The green line is the validation mAP calculated at every 500th iteration. The final result after 32.000 iterations is a mAP 90.55. After 20.000 iterations there is a visible drop in loss and a stabilization in the validation scores. This is caused by a drop in learning rate which happens after 20000, 26000 and 30000 iterations where the learning rate becomes 10^{-5}, 10^{-6}, and 10^{-7} respectively.

4.2.2 Network testing

Non-colored detection testing

Figure 4.12 shows a few selected frames from the validation run to showcase what went right and what is lacking in the first iteration of the 300x300 network while a full video of the validation run can be found here:

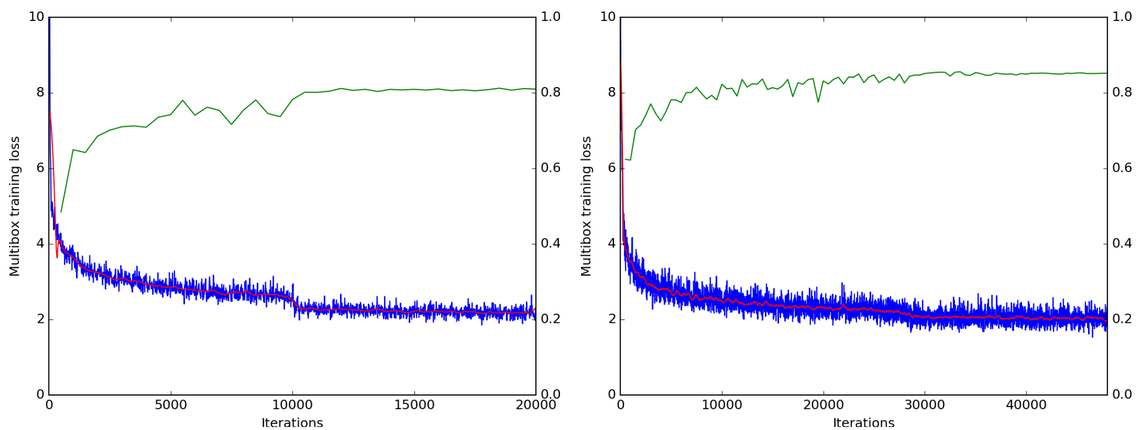
- `https://www.youtube.com/watch?v=MrzvE8vgIkA`

The next network to be tested was trained on the same dataset but the input images were scaled to 512x512 instead. A selection of the frames can be seen in Figure 4.13 and here is a link to the full video of the validation run:

- `https://www.youtube.com/watch?v=rXfx7FQtS-w`

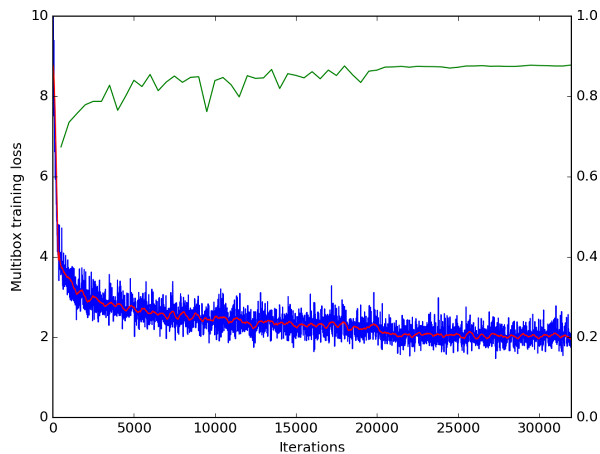
Net	Input size	LR	LR steps	BS	Iter	mAP	Loss
SSD [25]	300x300	10^{-3}	[10.000,18.000]	32	20.000	80.98	2.1
SSD [25]	420x420	10^{-4}	[28.000, 38.000, 46.000]	12	48.000	85.16	2.0
SSD [25]	512x512	10^{-4}	[20.000, 26.000, 30.000]	8	32.000	87.83	2.0

Table 4.1: Output values from training the SSD network with different input sizes on the colored robot detection dataset. LR is the starting learning rate while LR steps is a list of the different iterations where the learning rate drops to 10% of its previous value. BS is the batch size for each network which had to be adjusted to fit the GPU memory for the computer running the training process. Iter is the final number of iterations before training was terminated, while mAP and loss refers to the accuracy and loss of the network at that time.



(a) Training graph for SSD 300x300 on the colored dataset.

(b) Training graph for SSD 420x420 on the colored dataset.



(c) Training graph for SSD 512x512 on the colored dataset.

Figure 4.10: Training graphs for the different SSD input sizes that were tested.

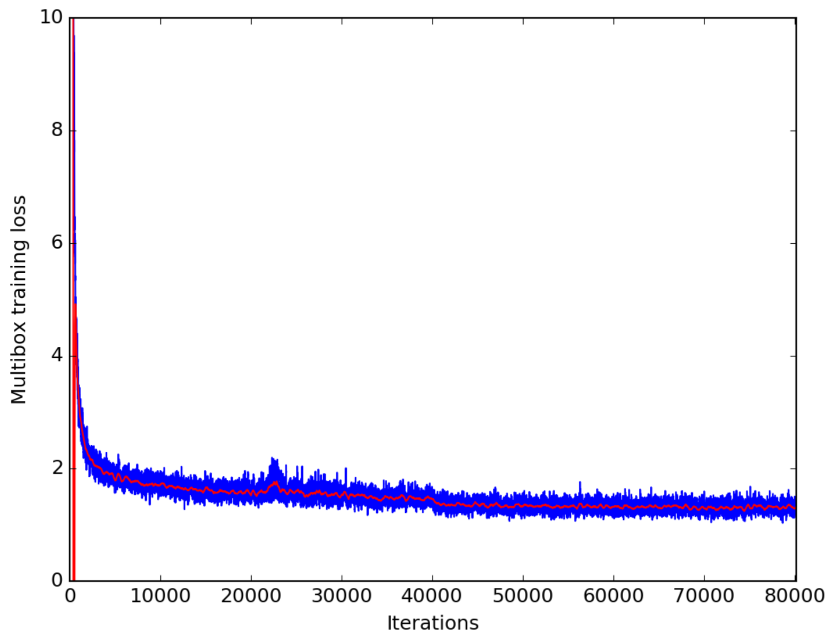


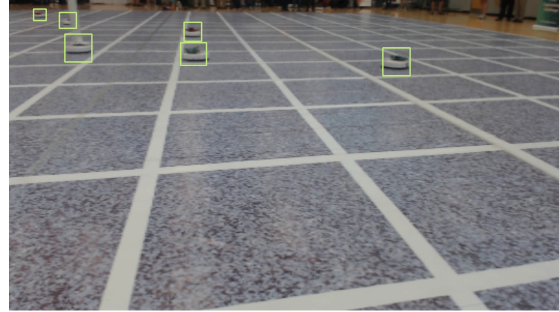
Figure 4.11: Training graph of an YOLOv2 network on the colored dataset with an input size of 416x416. The blue line represents the training loss for every iteration while the red line is a filtered average of this loss to better visualize if the loss has converged or is still decreasing. The final error after 80.000 iterations is about 1.3. YOLOv2 did not provide any simple means to calculate the validation score neither during training or after so the training was performed blindly. The learning rate was set to 0.001 and was set to 10^{-4} and 10^{-5} after 40.000 and 60.000 iterations respectively.

The speed and accuracy of the two different non-colored networks can be seen in Table 4.2 and show that the TX2 actually performs much worse than the TX1 on SSD. This is a very strange issue which seems to be caused by optimization techniques used in SSD that are supported on TX1 but not yet on TX2 as it was just newly released. The effects of this will be discussed further in Chapter 5.

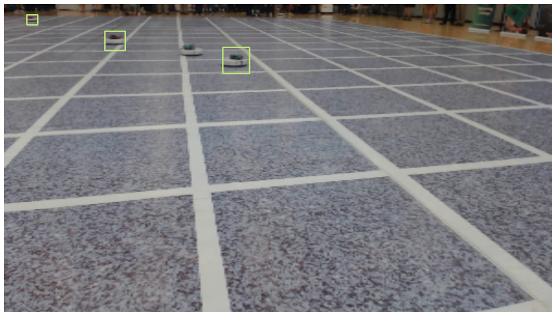
Network	Input size	mAP	FPS on Jetson TX1	FPS on Jetson TX2
SSD [25]	300x300	83.78	9.0	5.5
SSD [25]	512x512	90.55	3.5	2.5

Table 4.2: Test results of the two SSD networks trained to detect robots without separating between colors.

(a) Example of correct labeling even when the camera is at an angle.

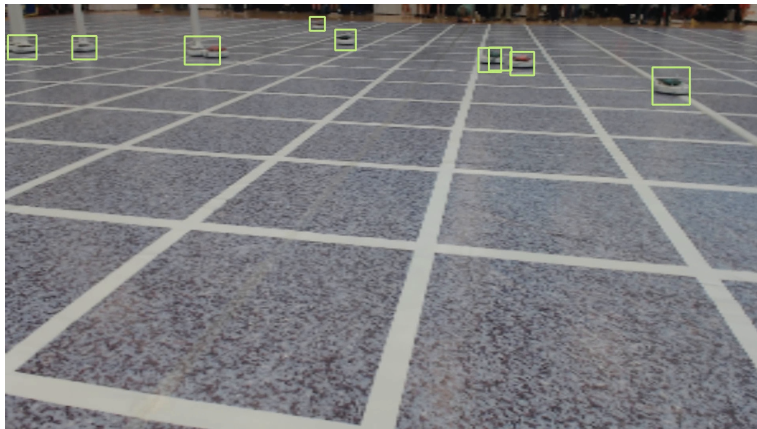


(b) Example of correct labeling even though one of the robots are occluded by a pole.



(c) Example of incorrect labeling. A clearly visible robot was not detected.

(d) Example of incorrect labeling. A robot was detected in the pattern on the floor.

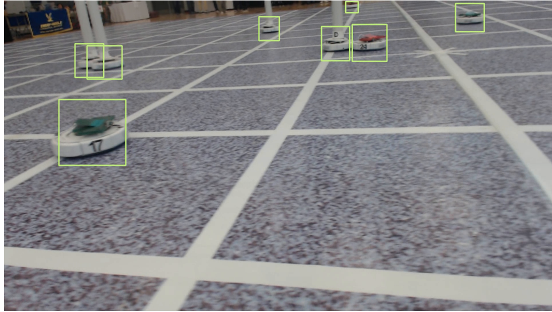


(e) Example of incorrect labeling. Two robots are detected as one to the left while two other robots to the right are detected as three separate ones.

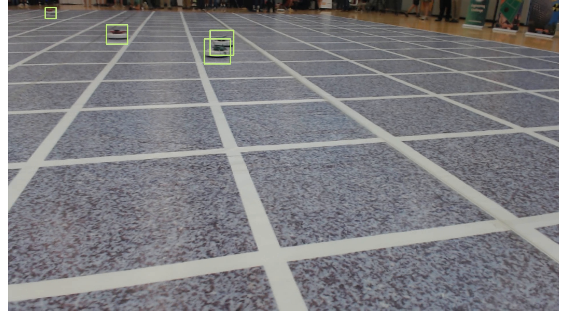
Figure 4.12: Examples highlighting the performance of the 300x300 SSD robot detection network in different scenarios in the validation set. Most of the mistakes are only present for a small number of frames.

Colored detection testing

Next we try train the colored versions of the SSD network. The same sizes as before are used to see if the results stay the same. Additionally we try to find a middle ground



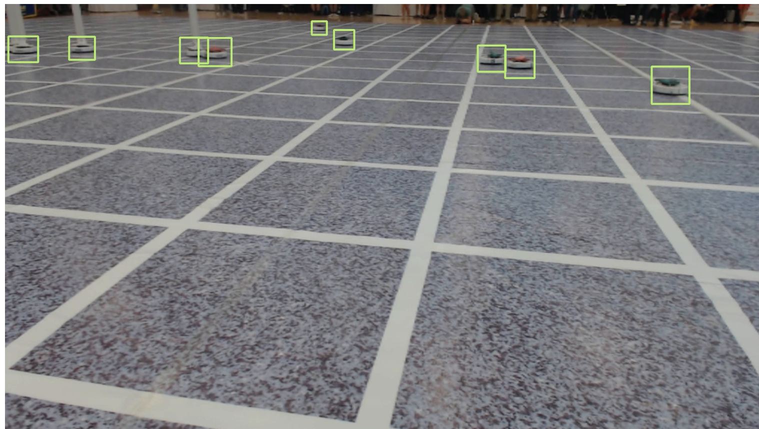
(a) Example of correct labeling. Even though the camera is at an angle it still detects the robot in the far back.



(b) Example of correct labeling of robots that are very close to each other.

(c) Example of incorrect labeling. All robots except one in the far back is detected in a very blurry image.

(d) Example of incorrect labeling. The network still finds ghost robots in the floor pattern, but significantly less frequent than SSD300.



(e) Example of correct labeling with several robots close to each other.

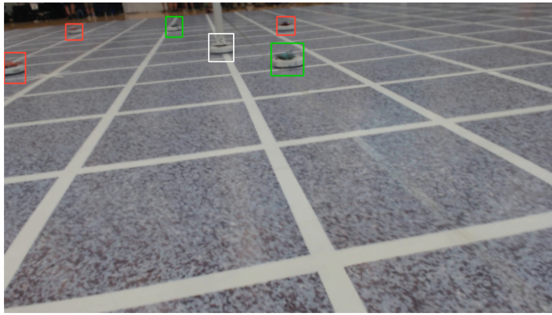
Figure 4.13: Examples highlighting the performance of the 512x512 SSD robot detection network in different scenarios in the validation set. Most of the mistakes are hardly ever present for more than a few frames.

between SSD300 and SSD512 in terms of performance and train a network with input size 420x420. Figure 4.14 shows the pros and cons of the SSD420 model as well as some of the differences between colored and non-colored networks in general. The colored models performs almost exactly as well as the non-colored versions in detecting the different robots. However, the addition of color has slightly decreased their overall mAP score. Lastly we have included a YOLOv2 robot detection network to compare its performance to SSD. Examples of its performance can be seen in Figure 4.15. The speed and accuracy of the different networks is shown in Table 4.3 while the full video of the result can be seen here for a more thorough investigation of the results:

- Colored SSD300: `https://www.youtube.com/watch?v=bDhBCKNrVgQ`
- Colored SSD420: `https://www.youtube.com/watch?v=adqlBLumwnw`
- Colored SSD512: `https://www.youtube.com/watch?v=ZFqJTsurVVw`
- Colored YOLOv2: `https://www.youtube.com/watch?v=a7gu9n59Mg0`

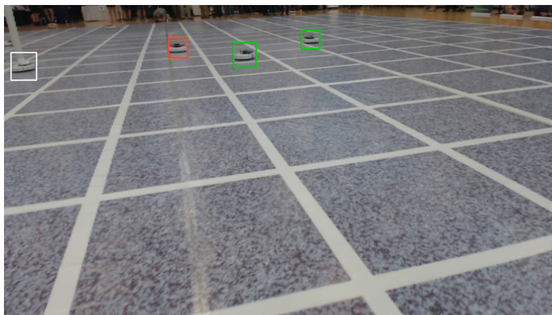
Network	Input size	mAP	FPS on Jetson TX1	FPS on Jetson TX2
SSD [25]	300x300	83.78	9.0	5.5
SSD [25]	420x420	85.16	5.5	3.0
SSD [25]	512x512	90.55	3.5	2.5
YOLOv2 [35]	416x416	-	11.5	19.0

Table 4.3: Results of the different detection networks run on Jetson TX1 and Jetson TX2.



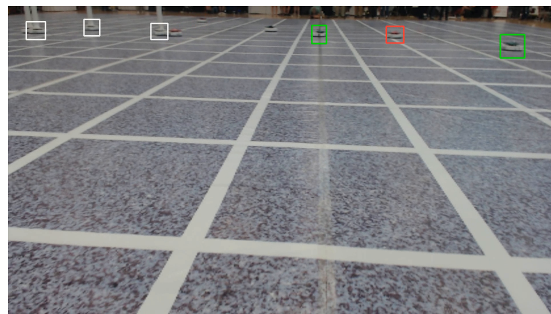
(a) Example of correct labeling even though the image is blurry. All colors labels are correct.

(b) Example of correct labeling. The colored version does not detect robots when the drone lands on the ground.



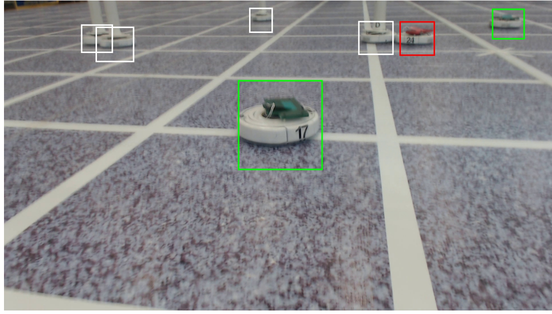
(c) Example of incorrect labeling. Misses the single robot in the back.

(d) Example of incorrect labeling. Two robots of different colors are detected at almost the same position to the right.

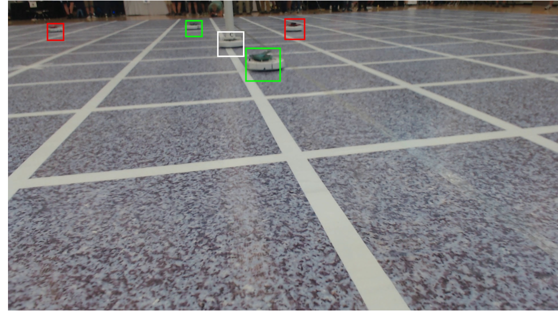


(e) Example of incorrect labeling. Three robots are undetected but the colors of the detected ones are all correct.

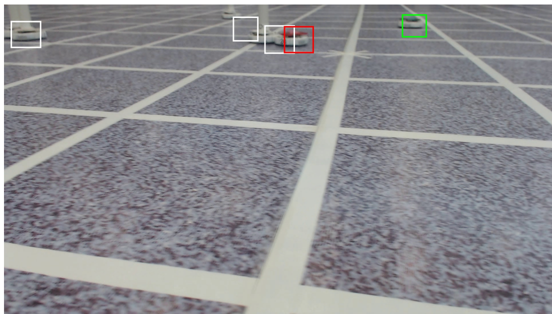
Figure 4.14: Examples highlighting the performance of the 420x420 SSD robot detection network in different scenarios in the validation set. The networks overall performance is almost as good as the SSD512 but performs the detections slightly faster.



(a) Example of correct labeling.

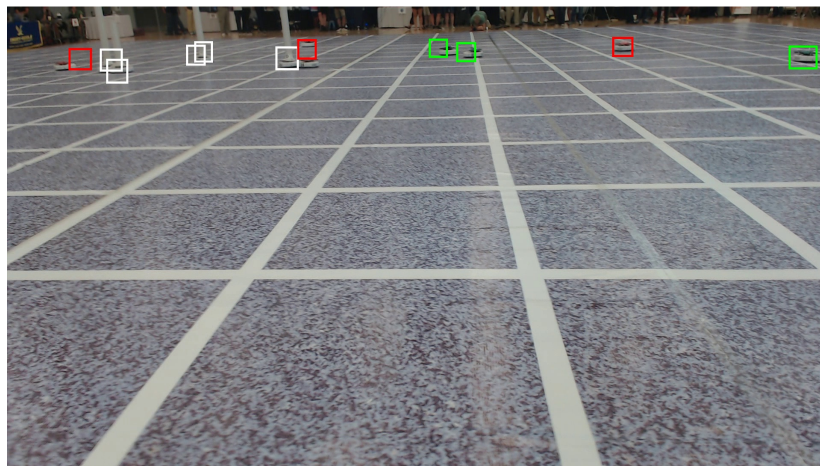


(b) Example of correct labeling.



(c) Example of incorrect labeling. YOLOv2 tends to blend objects that are close together like we see here.

(d) Example of correct labeling. Two robots in the back of the image was not detected.



(e) Example of correct labeling. The bounding boxes sometimes seems to slide of the detected robots like they do here.

Figure 4.15: Examples highlighting the performance of the 416x416 YOLOv2 robot detection network in different scenarios in the validation set.

Chapter 5

Discussion

This chapter will go through the results from Chapter 4 and discuss which areas the different approaches succeeded in and which they did not in accordance with the research questions of this thesis. We will end the chapter with a set of methods that had the potential to improve the current solution but were discovered too late to be implemented and tested.

5.1 Detection dataset

This section will address the methods that contributed to answering the first research question of the thesis: How can reliable and general training data be created for a deep learning based object detection system without any prior information about position or size of the objects?

5.1.1 Robot segmentation

A significant amount of time was spent trying to create a segmentation solution that was as reliable to the ideal robot segmentation as possible. The initial intention was to use this to create a dataset for a segmentation based neural network as the final solution. At that point in time we were not familiar with SSD or any other existing detection network with the required efficiency because of their fairly recent publication dates. This lead to an increased focus on segmentation in the initial stages of this thesis. The plan was always to enable extraction of location data from the segmentation images so that the solution would be useful independently of whether a detection or segmentation based approach would be selected. Being able to segment out the parts of the image that contained robots was therefore vital, however it might not have needed to focus so much on the perfect segmentation that it did.

The results in Section 4.1.1 show that the approaches for fixed size and self adjusting window sizes get a 96.98% and 98.17% match when comparing them pixelwise to the ideal segmentation. These solutions should be able to achieve quite high values

considering that 96.20% of the image is background. That means that the self adjusting window method is performing much better than the fixed size one even though they are only 1.19% apart in similarity percentage. The run time is also vastly improved as a single scan of an image in both cases take about the same amount of time, but the self adjusting solution only need a single scan compared to the four or more required by the fixed size depending on how many scales are included. This also leads to an output resolution that varies throughout the image. The determining factor of the resolution is the step size currently set to 1/4 of the expected robot size. This means that each row will have its own resolution which gradually decreases as we get further down the image. This effect is clearly visible in Figure 4.1(d) as the image looks more pixelated the further down we get. One way to improve the resolution would be to simply decrease the step size but this will also increase the computation time. This would have been preferable if the focus of the thesis had been a segmentation solution, but since the segmentation image would not be used directly by any network we were safe to assume that a step size of 4 was accurate enough. Later results proved to be correct.

In Figure 4.2(b) we see how the sliding window performs on an image that contains large amounts of background without having any actual robots to detect. The segmentation image reveals several areas where the classifier has incorrectly detected robots in parts of the image with about the same shape or color as a ground robot. If a collection of incorrect classifications becomes large enough it is likely to be mistaken for a robot and create a false positive in the final result. This would violate the premise set in the research question of creating a reliable detection method, and the classifier clearly needed to be more general, hence the introduction of the bootstrapping algorithm.

The bootstrapping required more time to perform than it might look like on paper. This was due to the larger than expected number of iterations required to truly specify the edge case scenarios. Each iteration of bootstrapping mainly improved the model in whatever way it most needed it, but at the same time it gave room for minor other issues to grow. The issues that emerged were however always much smaller than the ones that were fixed and after enough iterations over a sufficiently large set of ideal segmentations it managed to converge to a very stable solution.

Despite the fact that perfect segmentation wasn't required, we still managed to get remarkably close even though the output resolution from the self adjusting sliding windows were somewhat unreliable. The results in Figure 4.1 show a 99.13% correctness to the ideal segmentation and the background errors are almost completely removed as shown in Figure 4.2(c). The resolution could still have been improved by setting a smaller step size in Algorithm 1, and the accuracy of the classifier could have been improved even further by bootstrapping more example images or increase its image input size to something larger than 64x64. This also proves that the approach was made modular enough to be used in several types of segmentation tasks which later proved very advantageous. The main contributer to its modularity was the bootstrapping method which both simplified and sped up the process even though it required a few iterations to reach its full potential. All in all the robot segmentation was a success and deemed very useful as a starting point for a complete and reliable detection algorithm.

5.1.2 Robot localization

The starting point in the location process was the segmentation images created by the self adjusting sliding window in combination with the finely tuned CNN classifier. This introduced a few problems that initially had not been considered, like the difficulty of visually separating larger blobs into separate detections. This was an important step to address thoroughly to make the detection dataset general enough, otherwise the networks would most likely not be able to separate overlapping robots properly.

Meanshift and Camshift

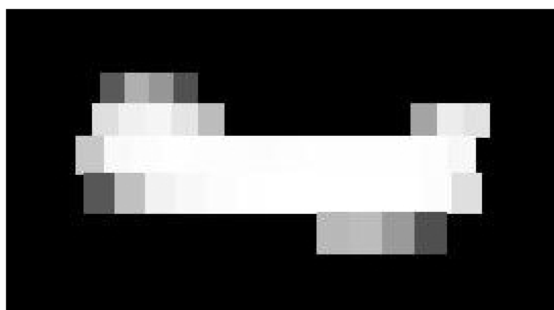
Both Camshift and the custom Meanshift method failed because they both looked for the maximum density of the entire blob, without any concern for the fact that there might be more than one robot enclosed within it. These approaches were unfortunately too greedy and did not attempt to optimize for more than one center location at the time which lead to critical errors and are thus not viable answers to our research question.

Box filter

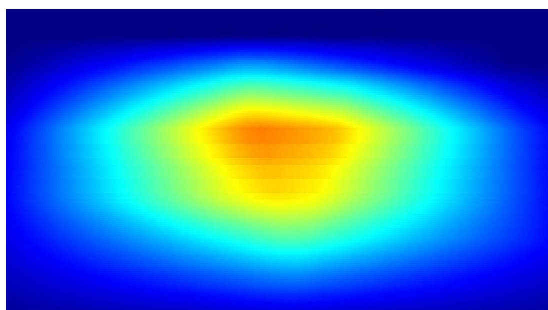
The first attempt at creating a filter with a kernel size equal to the size of the robots was the box filter approach. We see in Figure 4.4 that the blurring worked quite well on the blobs that only contain a single robot. Even the two robots that are close without overlapping each another created an output that made it possible to see where the two centers should have been located. However the blob to the far left contains two robots somewhat occluding each other and is therefore wider than the filter size. The intention behind this design choice was to enable separation between the different parts of the blob that belonged to the different robots by exposing two separated centers of mass. We see that it failed to do so because intensity values became centered within the entire block instead of being split into two as we needed it to, much like it did with the Meanshift and Camshift algorithms. Because of the square shape of the filter and the fact that everything in it was weighted equally we got a gathering of high intensity values in the middle area of the entire blob instead of at each of the centers individually. To illustrate this point we have extracted the region from Figure 4.4 holding two overlapping robots displayed in Figure 5.1 to show how it looks like in the original image, the segmented image, and the filtered segmentation in (a), (b), and (c) respectively. Another concern here was computation time. Even though the approach could have been optimized and made a great deal faster than the simple and straightforward implementation used for testing, it would have required a lot of work to make it run in real-time if the process were to run for every pixel in the full sized image.

The main reasoning behind this approach was to see if smoothing out the segmentation with a filter fitted to the robot size would make it easier to locate the center of the robots. In that regard it was successful. Not only did it prove that the concept was worth pursuing further but it also gave a great sense of which aspects had to be improved for it to yield the best results. In total we were faced with two main problems

(a) Extract from the original image in Figure 4.4(a).

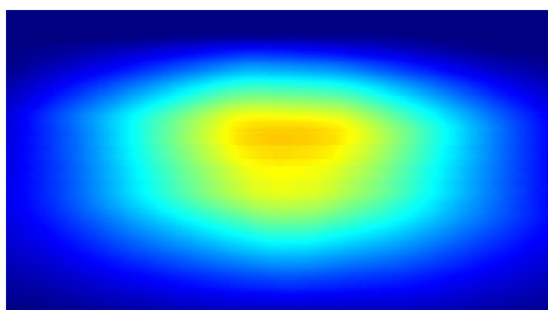


(b) Extract of the same region as in (a) from the segmentation image created by the self adjusting sliding window.



(c) Extract of the same region as in (b) after the box filter was applied to it.

(d) Extract of the same region as (a) from the ideal segmentation image.



(e) Extract of the same region as in (d) after the box filter was applied to it.

Figure 5.1: Extracted region from the image in Figure 4.4(a) to show how the application of a box filter looks like for a sliding window segmentation in (b) and the ideal segmentation image in (d).

from the box filter approach. First and most importantly, it does not separate robot centers that are close to each other well enough to make any assumptions about their actual positions. Second, the process is still quite time consuming and should be sped up significantly. It is also worth mentioning that the resolution at this point is equal to the original input image at 1920x1080 pixels which makes it easier to analyze as results but might not be necessary for the process to obtain accurate results.

Gaussian filter approach

Several improvements were implemented in a relatively large step here to address the problems with the box filter. The reason they were not tested more thoroughly as individual parts was a combination of two things. First, we thought the method was much more likely to produce a viable output after these steps had been combined. Testing the individual parts was therefore not likely to tell us much about the complete process. Second, we did not want to repeat the mistake of spending too much time and effort on a single module before combining it with the other parts. Too much focus had already been spent on creating a perfect segmentation algorithm when in reality a more center oriented approach was much more useful in the later stages. The changes were therefore implemented individually but tested and assessed as a single larger unit. The result seemed to benefit from this as the tests directly impacted the final output and were more to the point.

An important change in the evaluation process at this stage is that the bootstrapping process now can be controlled in a much greater sense than before by adjusting the correctness threshold for edge cases. The last approach only looked at the center pixel ground truth of every window to determine whether the classification was correct or not. Now we can evaluate the full process and choose to increase or decrease the correctness threshold of false positives and false negatives into the new dataset. This turned out to be an important asset.

All the systems required to create, test and automatically improve different center oriented datasets in combination with the full detection procedure have been put in place. This made it possible to simply try different thresholds for the sake of curiosity rather than analyzing in depth which exact value was most likely to result in the most progress before trying it out. We did of course make an effort to adjust the thresholds intelligently, but testing was cheap and the final classifier dataset is a combination of small improvements done by trial and error. The results of the process were presented in Figure 4.5. To fully understand why the different steps were made we also have to take Figure 4.6 into account. For the first few iterations represented by (c) in both figures we were mainly concerned with creating a classifier that could create a perfectly centered segmentation image. To achieve this we simply set the threshold for both false positives and false negatives to 0.5. This meant that all classifications that were even slightly in opposition within its neighborhood were relabeled and put back in the dataset. After two iterations we started to see a trend that was more damaging to our goal than it was advantageous. In Figure 4.5(c) we see that the classifier becomes too concerned with splitting the overlapping robots into two separate blobs. The one in the back is reduced to almost nothing which makes it very hard to distinguish from noise. In Figure 4.6(c) the blobs have completely been reduced to a single unit that only focus on the foremost robot. The problem is that the hard separation between close robots have been enforced too strictly. The small gap between close centers are labeled as false positives and put back into the dataset. To solve this we simply adjust the correctness threshold of the false positives to make it more restrictive. By setting it to 0.9 we make sure that no false positives are added unless its close neighborhood is mostly all empty space. The outcome of this change is shown in Figure 4.5(d) with its Gaussian equivalent in Figure 4.6(d). This is a step in the right direction, but

now the two blobs are slightly too connected which makes it hard to differentiate the two robots from each other. This is the same problem that occurred before the center oriented approach was introduced. We see in Figure 4.6(b) and (d) that they both posses the same high intensity bridge between the centers of the overlapping robots. The next attempt will keep the correctness threshold for false negatives at 0.5 while the threshold for false positives is set to 0.8, which will hopefully be a middle ground between the previously tested threshold values. The results were presented in Figure 4.5(e) and 4.6(e). We are now at a sweet-spot where the blobs are large enough to be evaluated properly and small enough to not be evaluated as a single large robot. The full process now looks like our intended goal in Figure 3.9 with the final output detections was shown in Figure 4.7. Several other threshold values were tested as well but the best result was achieved with a correctness threshold for false positives at 0.8. It is also important to notice that the image used here was not part of the images used for training and therefore represent an impartial result.

The final dataset

The steps of the Gaussian filter approach were impotent to discuss here because they show how the modularity of the bootstrapping algorithm was able to create a CNN classifier that in combination with the Gaussian filter approach finally answers our research question. With the amount of blur and inaccuracy in altitude data in some of the IARC 2016 images we consider a final result that correctly detects all robots within the images 83% of the time a success. With a supervised inspection and refinement of the dataset we now have a reliable dataset as well as a general one considering it contains examples of robot detections from multiple angles, multiple levels of occlusion and multiple scales.

5.2 Detection network

This section will address the methods that contributed to answering the second research question of the thesis: How can object detection be done reliably in real-time on an embedded system with limited computational potential?

5.2.1 Network training

The training process of the non-colored SSD networks was surprisingly simple. In Figure 4.8 and 4.9 we saw that the loss keeps declining while the validation accuracy improves until it stabilizes. Both SSD300 and SSD512 were able to converge on solutions with with final mAP values that were even better than the values advertised in the paper. The reason is that robot detection within a known environment is a much simpler problem than general object detection. We also see that the effect of the input size is much higher in the robot detection than the general detection. The difference in mAP between SSD300 and SSD512 in the case of robot detection is 6.8 while only 2.5 points separate the general detection versions from each other. That is most likely due

to the small size of some of the robots, which a larger input size is much more likely to pick up. Objects in the standardized detection datasets are usually not that small so the 2.5 point improvement is more likely to be a result of more accurate feature maps rather than the ability to detect smaller objects.

Training of the colored versions of the network behaved very similarly to the non-colored training, but required a few more iterations to properly stabilize. It does not seem that the addition of two more classes made the localization much harder considering that the network easily converged to solutions with high mAP scores. There is however a drop in mAP from the non-colored versions to the colored ones which will be examined further when we discuss the results of the network testing. The SSD420 network training required a large amount of iterations but resulted in an mAP score of 85.16, which is only 2.7 less than SSD512. This could prove to be a serious candidate for the final solution depending on how fast it performs.

It appears that YOLOv2 did not have an option to run a validation process during training which made it very hard to determine if it started overfitting or not. Neither did it have a simple tool for calculating the mAP, and because of the late discovery and implementation of YOLOv2 we did not have time to implement such a validation method our self either. We must therefore simply trust that the detection dataset is general enough for YOLOv2 just like it was for SSD and hope it prevents overfitting.

5.2.2 Network testing

All networks that were tested are clearly able to generalize over the training set and show impressive detection results on the validation set. They all struggle to some degree to detect robots far away from the drone because of the low resolution and often blurry appearance of the smaller robots, but apart from that the networks perform very well. Detections in the colored and non-colored versions are almost identical in terms of robot localization. The colored version does however not detect robots in the pattern on the floor, but apart from that there is not a significant difference. The reason the colored versions have a slightly lower mAP than their non-colored counterpart seems to be a result of detections being mislabeled rather than wrongly detected. When it comes to the comparison of different run times we will use the frames per second performance on the Jetson TX1 because of the unfortunate lack of support for SSD optimization on the TX2. This will be a high priority optimization issue for future work before the IARC 2017 competition.

The output from the SSD300 is mostly correct but it has a tendency to lose detections for a frame or two seemingly at random which creates a flickering appearance of the detections in the output video. The main concern regarding this is that when the network is run with a low update rate it could be unfortunate and get the same missed detection on the same robot several frames in a row. The low update rate would then make the robot invisible for some time to the rest of the Ascend system which could lead to collisions if the undetected robot happens to have a pole mounted to it. The detections do seem to be more reliable the closer they are to the drone so it is not very likely that the inaccuracies of the SSD300 will lead to collisions, but it would still contradict the research goal of creating a reliable real-time solution. The advantage of

SSD300 is its speed but the number of inaccuracies, especially on smaller robots is not sufficiently reliable as a final solution.

The validation video of SSD512 show a much more stable detection from frame to frame than SSD300. It rare loses a detection for more than a frame or two and generally detects robots further back with much better accuracy. The problem with SSD512 is its slow FPS count. On the TX1 it is able to run at 3.5 FPS which is not bad for such an accurate detection result. However, faster detection means a more up to date sate of the world around at any given point and we would like to sacrifice some accuracy for a slightly faster solution.

The final SSD input size to be discussed is the 420x420 network. The validation video and its mAP score show that the network is almost as accurate as SSD512 but is able to operate at a slightly higher speed. The main portion of errors contain missed detection of robots far away and wrongly labeled robots for a frame or two at the time. Out of the three sizes of SSD that have been tested in this thesis we can conclude that SSD420 is the one that best answers our research question by having the most favorable combination of accuracy and speed.

Finally we take a look at YOLOv2. The first feature that becomes clear when watching the validation video is that it is not as accurate as SSD in detecting blurry robots. It is able to detect most robots but the detections seem to prefer certain spots in the frame and stick to them even though the object slowly moves away. When the robot moves far enough the bounding box makes a jump-like motion to fit the robot again. Overall the detection is decent and it is able to detect the positions of robots better than SSD300 but it also has a harder time separating close detections from each other like we see in Figure 4.15(e). This is a step further away from our research goal in comparison to SSD420 because it might be faster, but it is also less reliable than we would like.

If the optimization problems with TX2 are not solved before the 2017 IARC competition we will be forced to perform one of the following steps: Either replace the two Jetson TX2 cards that run detection onboard the Ascend drone with TX1 and run the SSD420 network at 4 FPS on each of them, or simply run YOLOv2 on the TX2s. Considering that the drone has two Jetsons intended for the detection network it would be most beneficial to scale back to TX1s and run the SSD420 network which we have seen is far more reliable than the YOLOv2. This will hopefully not be the case but as of the final day of this thesis this is the best solution.

5.3 Methods that should have been considered

This section is made for the different methods that were discovered late in the process and therefore was not part of the method selection discussion. However, they are very promising and the thesis would most likely have benefited from considering them at an earlier stage.

5.3.1 dlib

The dlib library described in Section 2.3.6 have proved itself able to make object detections robustly using a considerably smaller training set than most other multi object detection methods based on machine learning. This could most likely have created a result just as good or better than the current solution and would most likely require far less preparation. The dlib method does not require height and camera angle to find robots of different sizes and would probably be more accurate than the current solution because of it. The data recorded for drone altitude and camera angle was not accurate enough and resulted in a much unnecessary supervised correction.

5.3.2 Non-maximum suppression

Non-maximum suppression (NMS) is a method that is able to find the local maxima of e.g. multiple overlapping detection bounding boxes to output the single most likely one. There exist several different approaches to NMS which can be simple greedy local optimization methods or more general ones like [38]. MMOD[20] is also an example of a type of NMS method that manages to evaluate every possible sub-sampling within a region to find the optimal local maximum. SSD uses non-maximum suppression on multiple scales to select the most probable detection from a large set of overlapping bounding boxes. This thesis could have attempted to use it in the same fashion to extract location data from the sliding window approach. This could possibly have been a simpler approach to the localization problem in Section 3.2.4.

Chapter 6

Conclusion and future work

6.1 Conclusion

6.1.1 Detection dataset

First research question: *How can reliable and general training data be created for a deep learning based object detection system without any prior information about position or size of the objects?*

This thesis proved it was possible to create a large detection dataset using only images along with the altitude and angle they were taken at to create a reliable detection network that was general enough for more advanced detection networks to train on them. The automatic labeling process segments out the parts of the input image that contain robots and uses that to localize robots by applying a Gaussian filter which highlights the center of mass for each robot. The success rate of the process was 83% which means it was not perfect but good enough. The dataset did require a supervised inspection of the final result to make sure it did not contain any errors but this solution was still highly preferable to labeling all of the 6100 images by hand. The automatic solution also makes it simple to expand the dataset when more IARC recordings become available. Later testing on the different detection networks also proved that the dataset was both general and reliable enough to be used for training without encountering the problem of overfitting the data.

6.1.2 Detection network

Second research question: *Second, how can object detection be done reliably in real-time on an embedded system with limited computational potential?*

This thesis has certainly proven that it is possible to perform accurate real-time detection using the computing power of a drone. The results even show that the accuracy/speed trade-off can be adjusted by simply changing the input image size of the detection networks to find just the right balance. The approaches that were tested

were SSD with the following input sizes: 300x300, 420x420, and 512x512. A YOLOv2 network with input size 416x416 was also tested and proved to be faster but less reliable compared to SSD. For the purpose of this thesis the final result was a SSD network with an input size of 420x420 which enabled fast enough computation as well as very accurate robot detections. The accuracy is not perfect but it is certainly impressive for a system running with multiple frames per second for multiple cameras onboard a drone.

6.2 Future work

6.2.1 Tracking

A possible improvement to the solution would be a tracking method that either enables fast tracking between the detection frames to create a higher FPS count or as a noise reduction technique on case of missed detections or false positives. An example would be to use a Kalman filter for tracking between detection frame while implementing the Hungarian algorithm to solve the data association problem. This could even enable the system to assign each of the robots and ID and track them as long as they stayed within the camera field of view.

6.2.2 Deep feature flow

Deep feature flow can be implemented in both SSD and YOLOv2 to release the feature extraction network from creating the features from scratch for every frame like mentioned in Section 3.1.6. This would improve the overall FPS but the frames would also be outputted with varying frequency due to the changes back and forth between the regular feature extraction network and the feature flow network.

6.2.3 A more streamline way of training the detection part of SSD

A large part of this thesis was spent creating multi-object localization data from scratch to enable SSD to run as efficiently as it did. A much improved solution would be to first train the last detection layers of the SSD network to recognize the different types of ground robots and their localization individually. This would mean training the convolutional prediction layers on 3x3xP (where P is the number of channels the predictor is supposed to accept) feature maps created from images that were either empty or contained an object. A solution like this would save a lot of time considering how much simpler it is to acquire data that only contains a single object.

The greatest challenge with this approach is how to perform localization. This could either be done through a similar method to the one proposed in Section 6.2.4 or possibly even using the weakly supervised method proposed in [32].

Keep in mind that this is a theoretical possibility and there might be restrictions that limit this approach that have not been considered here.

6.2.4 Trimming the detection edges

It is uncertain how much the small vertical shifts and the horizontal stretch of the robot bounding boxes in the dataset really impacted the final outcome of the detection network. The network did not seem to have much of a problem with the noise in the training data as it generalized well over it which mitigated the error in the output. However, training on data without the shifts and the much too large vertical stretch of the bounding boxes might have made it easier for the network to converge on a solution and made it more confident in its detections. Possible ways to solve this is to create a detection model like in [5] or a neural network solution that can localize single objects in an image. A single object localization network should be uncomplicated to create by simply replacing the final layer in a pre-trained classification network to output the bounding box of the object instead of the class probabilities. This requires a dataset to train on, but will most likely not require more than a few hundred examples which can be created manually or possibly bootstrapped in some fashion.

Bibliography

[1] RealHardTechX Team at Realhardtechx.com. Power requirements for graphics cards, 2016. [Online; accessed 28-November-2016].

[2] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *CoRR*, abs/1511.00561, 2015.

[3] Gary R. Bradski. Computer vision face tracking for use in a perceptual user interface, 1998.

[4] Gabriel J. Brostow, Julien Fauqueur, and Roberto Cipolla. Semantic object classes in video: A high-definition ground truth database. *Pattern Recognition Letters*, xx(x):xx–xx, 2008.

[5] Juan C. Caicedo and Svetlana Lazebnik. Active object localization with deep reinforcement learning. *CoRR*, abs/1511.06015, 2015.

[6] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. *CoRR*, abs/1412.7062, 2014.

[7] International Aerial Robotics Competition. Official rules for the international aerial robotics competition - Mission 7. 2017.

[8] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html, 2012.

[9] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

[10] Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. *CoRR*, abs/1504.06852, 2015.

[11] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[12] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 38(1):142–158, January 2016.

[13] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[14] Alessandro Giusti, Dan C. Ciresan, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. *CoRR*, abs/1302.1700, 2013.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[16] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[18] Justin Johnson. Cnn-Benchmarks. `https://github.com/jcjohnson/cnn-benchmarks`, 2017. [Online; accessed 09-May-2017].

[19] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, 2016. [Online; accessed 21-November-2016].

[20] Davis E. King. Max-margin object detection. *CoRR*, abs/1502.00046, 2015.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[23] Yann LeCun, Clement Farabet, Camille Couprie, and Laurent Najman. Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 35(undefined):1915–1929, 2013.

[24] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[25] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[26] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[27] Jonathan Masci, Alessandro Giusti, Dan C. Ciresan, Gabriel Fricout, and Jürgen Schmidhuber. A fast learning algorithm for image segmentation with max-pooling convolutional networks. *CoRR*, abs/1302.1690, 2013.

[28] Anton Milan, Seyed Hamid Rezatofighi, Anthony R. Dick, Konrad Schindler, and Ian D. Reid. Online multi-target tracking using recurrent neural networks. *CoRR*, abs/1604.03635, 2016.

[29] Dmytro Mishkin and Jiri Matas. All you need is a good init. *CoRR*, abs/1511.06422, 2015.

[30] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. *CoRR*, abs/1505.04366, 2015.

[31] Peter Ondruska and Ingmar Posner. Deep tracking: Seeing beyond seeing using recurrent neural networks. *CoRR*, abs/1602.00991, 2016.

[32] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Is object localization for free? – weakly-supervised learning with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[33] Markus Pike. Computer vision and deep learning for autonomous drones - specialization project. 2016.

[34] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[35] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

[36] Scott E. Reed, Honglak Lee, Dragomir Anguelov, Christian Szegedy, Dumitru Erhan, and Andrew Rabinovich. Training deep neural networks on noisy labels with bootstrapping. *CoRR*, abs/1412.6596, 2014.

[37] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[38] R. Rothe, M. Guillaumin, and L. van Gool. *Non-maximum Suppression for Object Detection by Passing Messages Between Windows.* ETH-Zürich, 2014.

[39] Henry A. Rowley, Shumeet Baluja, and Takeo Kanade. Human face detection in visual scenes. 1995.

[40] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[41] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1605.06211, 2016.

[42] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[43] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[45] Andreas Veit, Michael J. Wilber, and Serge J. Belongie. Residual networks are exponential ensembles of relatively shallow networks. *CoRR*, abs/1605.06431, 2016.

[46] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. *CoRR*, abs/1611.07715, 2016.