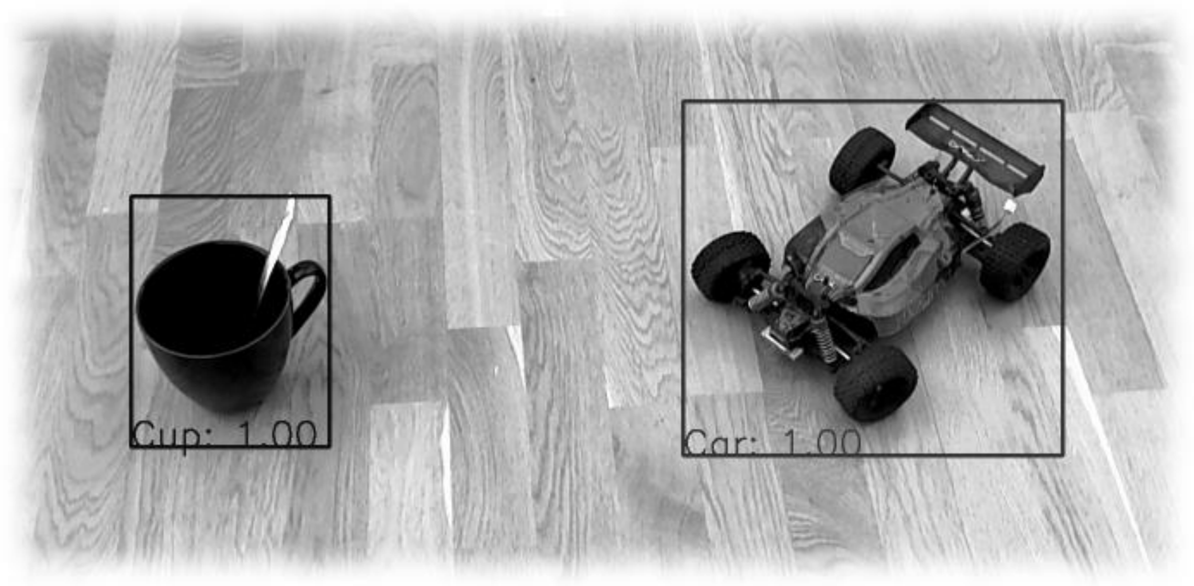
FMH606 Master's Thesis 2018

Industrial IT and Automation

Object Detection and Tracking on a Raspberry Pi using Background Subtraction and Convolutional Neural Networks



Torbjørn Grande Østby

Faculty of Technology, Natural sciences and Maritime Sciences

Campus Porsgrunn

Course: FMH606 Master's Thesis, 2018

Title: Object Detection and Tracking on a Raspberry Pi using Background Subtraction and Convolutional Neural Networks

Number of pages: 46 report + 19 appendix = 65

Keywords: Raspberry Pi, Object Detection, Convolutional Neural Network, Background Subtraction.

Student:	Torbjørn Grande Østby
Supervisor:	Ola Marius Lysaker
Co-Supervisor:	Joachim Lundberg
Availability:	Open

Summary:

Object detection and tracking are key features in many computer vision applications. Most state of the art models for object detection, however, are computationally complex. The goal of this project was to develop a fast and light-weight framework for object detection and object tracking in a sequence of images using a Raspberry Pi 3 Model B, a low cost and low power computer.

As even the most light-weight state of the art object detection models, i.e. Tiny-YOLO and SSD300 with MobileNet, were considered too computationally complex, a simplified approach had to be taken. This approach assumed a stationary camera and access to a background image.

With these constraints, background subtraction was used to locate objects, while a light weight object recognition model based on MobileNet was used to classify any objects that were found. A tracker that primarily relied on object location and size was used to track distinct objects between frames.

The suggested framework was able to achieve framerates as high as 7.9 FPS with 1 object in the scene, and 2.9 FPS when 6 objects were present. These values are significantly higher, more than 7 times for 1 object and 2.6 times for 6 objects, than those achieved using the mentioned state of the art models. This performance, however, comes at a price.

While the suggested framework was seen to work well in many situations, it does have several weaknesses. Some of these include poor handling of occlusion, a lack of ability to distinguish between objects in close proximity, and false detections when lighting conditions change. Additionally, its processing speed is affected by the number of objects in an image to a larger degree than what the state of the art models are. None of the mention models have deterministic processing speeds.

Preface

This report is the written by Torbjørn Grande Østby, a student of Master of Science in Industrial IT and Automation at the University of South-East Norway, as his master's thesis. It is the result of 4 months of work, which included studying a field in which the author had no prior knowledge.

It is assumed that the reader has a background in science or related fields. With this, well known problems and algorithms, such as the Kalman filter and assignment problems, that are mentioned in a general way or not cited.

The front-page illustration was created by the author.

The following software was used throughout this project:

- Microsoft Office 365
- Microsoft Visio 2013
- Microsoft Visual Studio 2017 w/ Python Development Tools
- Notepad ++ 7.5.4
- Python 3.5 and 3.6
- Spyder 3.2.6
- Tensorflow 1.4 and 1.6
- OpenCV
- NumPy
- Scipy
- Matplotlib
- Keras
- Scikit-image

Porsgrunn, 15th May 2018

Torbjørn Grande Østby

Contents

Nomenclature

CNN – Convolutional Neural Network

CPU – Central Processing Unit

DPM – Deformable Parts Model

FLOPS – Floating Point Operations Per Second

FPS – Frames Per Second

GPU – Graphics Processing Unit

HOG – Histogram of Oriented Gradients

IOU – Intersect Over Union

PC – Personal Computer

R-CNN – Region-based Convolutional Neural Network

ReLU – Rectified Linear Unit

RGB – Red Green Blue

RMSE – Root-Mean-Square Error

RPN – Region Proposal Network

SIFT – Scale-Invariant Feature Transform

SPD – Single Pass Detector

SSD – Single Shot multibox Detector

YOLO – You Only Look Once

1 Introduction

Object detection and tracking are key features in many computer vision applications, with uses in autonomous cars, medical diagnostics, surveillance, and industry automation, amongst others. It is a research area which has had a lot of progress in recent years.

Methods based on Convolutional Neural Networks (CNNs) have proven especially effective in this regard. With the increase in computational power of modern computers and hardware, and an increase in data availability, complex models are able to achieve both high levels of accuracy and low processing times. With the correct model and powerful hardware, real-time performance can be achieved.

The main goal of this project is to develop an object detection and tracking framework that is able to run on the Raspberry Pi 3 in real-time. A user-friendly visualization of the detection and tracking in images should also be given.

The Raspberry Pi is a low cost and low power computer. Due to its versatility and price, it has become popular in automation settings, especially amongst hobbyists. However, due to its low computational power, using unmodified state of the art architectures for object detection is likely to result in exceedingly long processing times.

A signed copy of the original task description can be found in Appendix A.

The rest of this chapter gives more information about the Raspberry Pi, and explains the terms real-time image processing, object recognition, object detection and object segmentation.

Chapter 2 gives a brief historical overview of object recognition and detection. This is followed by an in-depth explanation of CNNs, examples of various state of the art CNN-based object detector architectures, and an introduction to multiple object tracking.

Chapter 3 describes the implemented framework in detail, from detection and recognition, to tracking and data presentation.

Chapter 4 shows the results that were achieved when using the implemented object detection and tracking framework, and discusses its performance.

Chapter 5 contains a discussion about possible ways to improve the suggested framework, as well as issues that appeared during the project.

Chapter 6 concludes the project, and gives suggestions for further work.

1.1 The Raspberry Pi

The Raspberry Pi is a series small, low cost, and low powered computers developed by the Raspberry Foundation, a charity based in the United Kingdom. It was developed to promote teaching of computer science in schools and in developing countries. It has significantly less processing power than a regular PC, or even most modern smart phones, but due to its versatility and cost, it has become popular even outside the initial target audience. The size of the Raspberry Pi can be seen in Figure 1.1.

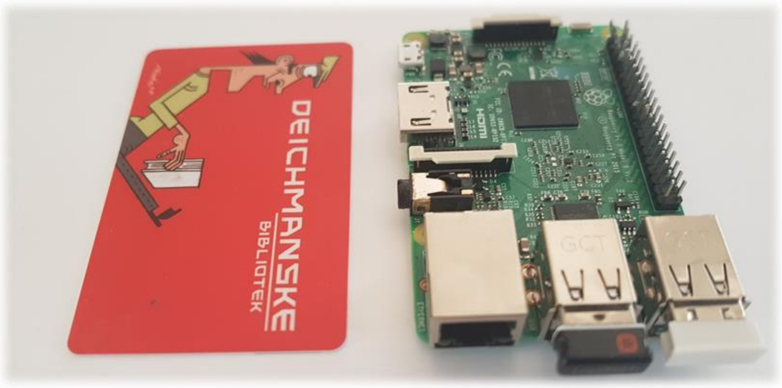


Figure 1.1 - The Raspberry Pi 3 Model B, with a size similar to a credit card.

The Raspberry Pi 3 Model B was used in this project. It boasts a higher CPU core count and speed compared to previous models. In certain specialized benchmarks it is able to achieve upwards of 5 billion FLOPS [1], however this performance does not reflect regular use. The Raspberry Pi's 3 Model B performance lags significantly behind current day laptop and desktop computers [2]. This poses a challenge, as most computer vision tasks are computationally expensive. Only having 1 GB of RAM, shared between CPU and GPU, is another factor which might prove a challenge.

A newer version of the Raspberry Pi, the Raspberry Pi 3 Model B+, was released 14.3.2018 [3]. It boasts some new and improved features, but uses the same SoC as the Model B, at slightly higher clock rates, but with the same amount of RAM. Performance using the Model B+ would likely have been slightly better, though this was not tested.

1.2 Real-Time Image Processing

Exactly what constitutes as real-time image processing can be somewhat unclear. In a digital signal processing context, it is required that the processing is completed, deterministically, within a given timeframe [4]. The lack of clarity stems from the question of how large this timeframe is. For online real-time video processing, the primary factor that impacts this timeframe, is the framerate of the video. Even so, video can be taken with a large variety of framerates. Modern smartphones often support framerates as high as 60 FPS, while some action cameras even support framerates of 240 FPS. Surveillance cameras, on the other hand, generally use much lower framerates, such as 10 FPS, 7.5 FPS, or even lower. In many cases, when the term is used in research papers about object recognition, it seems it only means "high FPS".

1.3 Object Recognition, Detection and Segmentation

Object recognition, object detection and object segmentation are three important concepts in computer vision. This chapter gives an explanation and example of what is meant by these three concepts.

According to the Dictionary of Computer Vision and Image Processing the term object recognition relates to "identifying which of several (or many) possible objects is observed in an image." [5, pp. 192-193]. In many research articles, however, the term is used to describe models able to classify whole images into one of a number of classes [6], [7], [8], [9]. The term image classification is often used synonymously with object recognition. An example of such classification can be seen in Figure 1.2. When used in this project, it is this latter, "classification of an image of an object", understanding of object recognition that is implied.

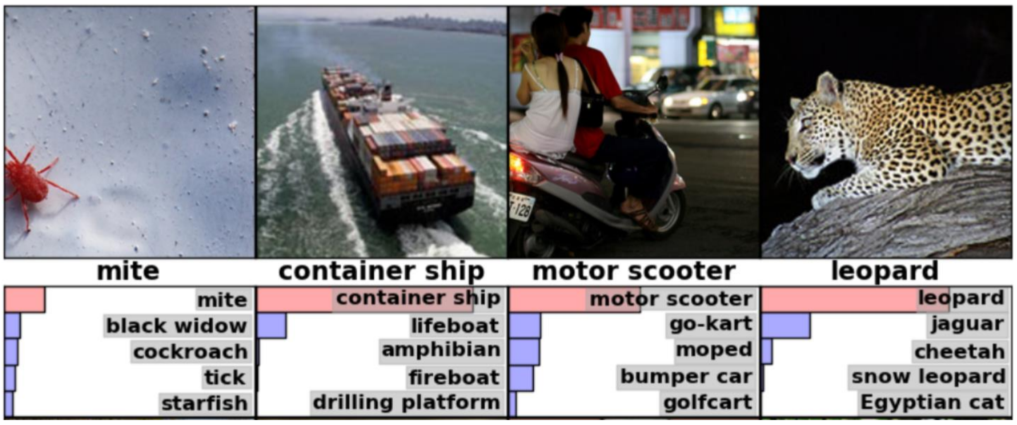


Figure 1.2- An example of object recognition, with AlexNet. Figure from [6, p. 8].

Object detection, on the other hand, is about identifying and locating one or more objects in an image, as implied by research articles that describe object detection models [10], [11], [12]. The Dictionary of Computer Vision and Image Processing gives the simple, and quite general, definition, that object detection is "The discovery of objects within a scene or image" [5, p. 192]. The location of a detected object is usually described by a bounding box, which is a rectangle used to bound the extremities of the object, and the identification is usually presented as a class probability score. An example of object detection can be seen in Figure 1.3. Models for object detection are discussed further in Chapter 2.

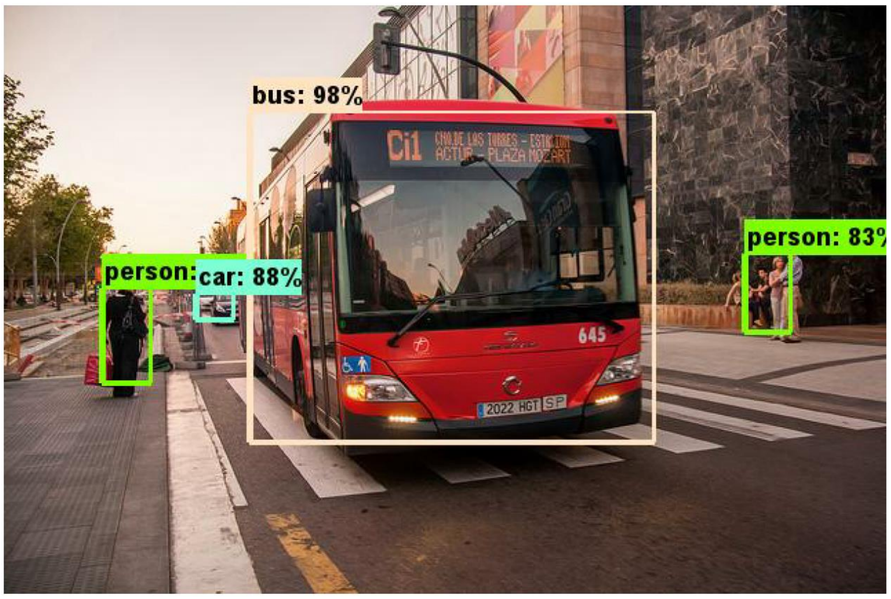


Figure 1.3 - An example of object detection, with MobileNet and SSD. Figure from [7, p. 7].

Finally, object segmentation pertains to "The separation of objects within a scene or image" [5, p. 193]. Segmentation differs significantly from detection, in that, where object detection uses a coarse bounding box to denote an objects location, object segmentation aims to predict the exact pixels that are associated with a given object. Needless to say, this is a more complex operation than object detection. An example of object segmentation can be seen in Figure 1.4.

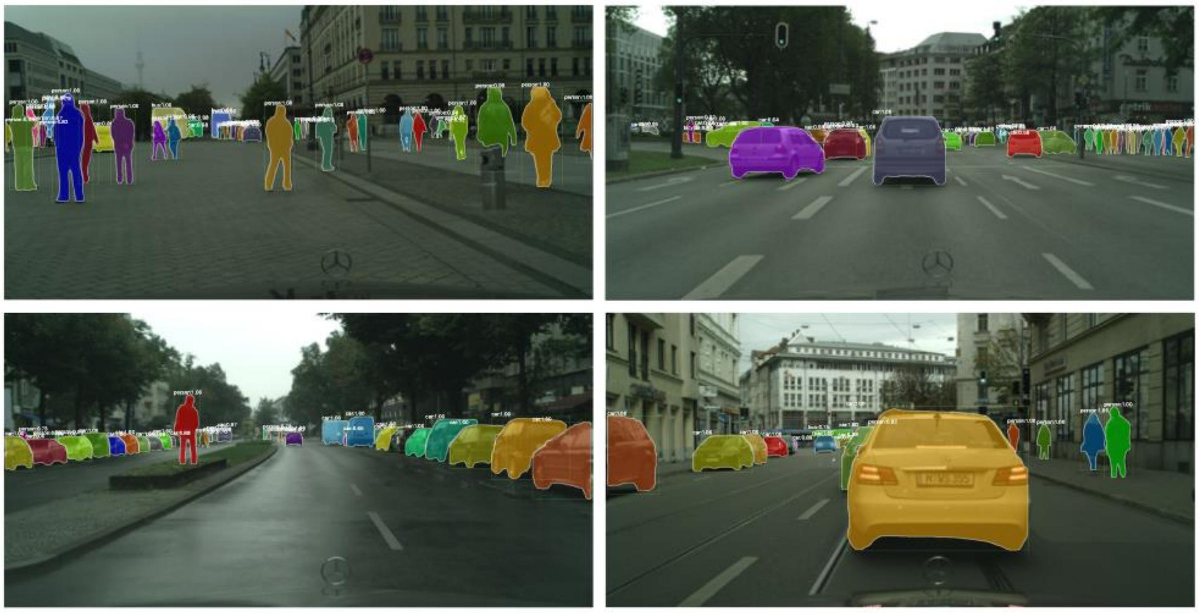


Figure 1.4 - An example of object segmentation, with Mask R-CNN. Figure from [13, p. 9].

2 Theory

In this chapter, a brief history of object recognition and detection is presented, followed by a detailed explanation of how CNNs work, examples of CNN-based feature extractors and CNN-based object detectors, and, finally, an explanation of how multiple object tracking works.

2.1 Brief History of Object Recognition and Detection

Object recognition and detection has long been a challenge in computer vision, and many different approaches have been taken in the attempt to overcome it. Some of these approaches include matching visual aspects of an object, such as edges, contours, and colour, with similar instances in an image, or using more specific features to do the same.

Before 2012 the primary method used for object detection was based on using feature matching. In many cases these features were hand-crafted. An example of a popular method is Scale-Invariant Feature Transform (SIFT) [14]. It is able to recognize known objects in images, and solved many of the problems with matching features with changing scale and rotation. A method using a similar approach is Histograms of Oriented Gradients (HOG) [15], though it is more interested in contours than specific feature points. HOG has been successfully used for challenges such as pedestrian detection, though it works less well for deformable objects and people in more varied poses. A solution to this was introduced with the Deformable Parts Model (DPM) [16]. Instead of using a single large template in order to locate an object, as HOG does, several templates are used for various object parts as well as the base object. Prior to 2012, DPM was the state of the art when it came to performance on object recognition benchmarks such as ImageNet.

Then in 2012 AlexNet [6], a CNN based object recognition model, entered the yearly ImageNet challenge. It outperformed its competition by a large margin, achieving 15.3 % top-5 error compared to 26.2 % from the second-best entry. This was in many ways the advent of CNNs, and since then CNNs have become immensely popular. CNNs, however, weren’t really anything new. An important example of this is LeNet from 1998 [17] , which is a model that was being used to read hand written digits in postal zip codes. The increase in the computational power of computers and the increase in available data have been used to explain the resurgence of CNNs in 2012 and since.

Today CNNs are the state of the art. They do, however, have their weaknesses. Capsule networks and CapsNet [18] are examples of an architecture whose performance is impacted less by various transformations of the input data compared to what CNNs are. Capsule networks, however, are still in their infancy, and are not used in this project.

2.2 Convolutional Neural Networks

This chapter gives an introduction to CNNs. First, an explanation of how CNNs work is given, before examples of CNN-based feature extractors and object detection architectures are introduced.

2.2.1 The Structure of Convolutional Neural Networks

CNNs are primarily made up of 3 main building blocks; convolutional layers, activation functions, and pooling layers. For object recognition a fully connected feed forward part is often also included.

2.2.1.1 Convolutional Layers

In a convolutional layer, convolutions are performed between the filters of the layer and the matrix input to the layer. This is similar to when a filter is used for edge detection. The difference is that for edge detection the filters are hand crafted, while for CNNs the filters are found through solving an optimization problem, and that CNNs usually contain a lot of filters. The values in these filters are called weights.

The filters are smaller than the input, and as such, these filters are moved across the input in a sliding window approach [19]. An example of the convolution and sliding window approach can be seen in Figure 2.1. The number of filters in a layer, the size of the filters, and the stride with which the filters are moved between convolutions, are user defined hyperparameters. In this context a hyperparameter is a parameter with a value that is set before a model is fit to the data, while other parameters are derived through training [20]. The output from a convolutional layer is often referred to as a feature map.

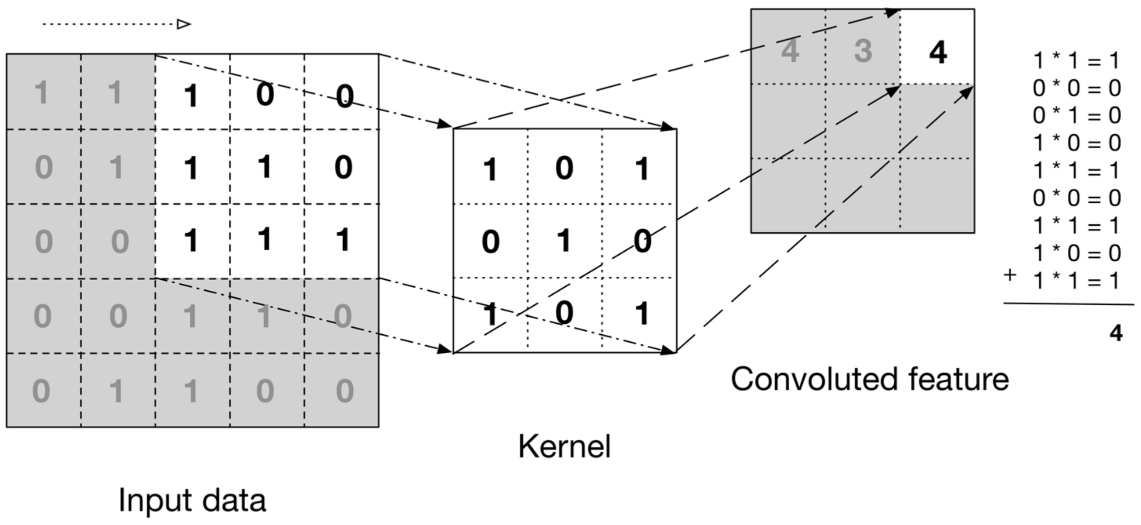


Figure 2.1 - The convolution operation. Figure from [19, p. 131].

For 3-dimensional input, such as an RGB-image with 3 channel layers, the filter size is generally defined by height and width, while the filter depth implicitly is equal to the depth of the input. Filters tend to be square with an odd numbered height and width, such as 3×3, 5×5, or 7×7, so that there is a centre pixel in the filter.

Because of the way the convolutions between the input and filter are performed, information contained near the edges and corners of the input is given less impact in the layer output. To alleviate this problem, zero padding of the input is often used. In such cases, often referred to as *same* padding, the input is padded with zeros so that the layer output has the same height and width as the input [19]. An example of this can be seen in Figure 2.2. Using no padding is often referred to as *valid* padding.

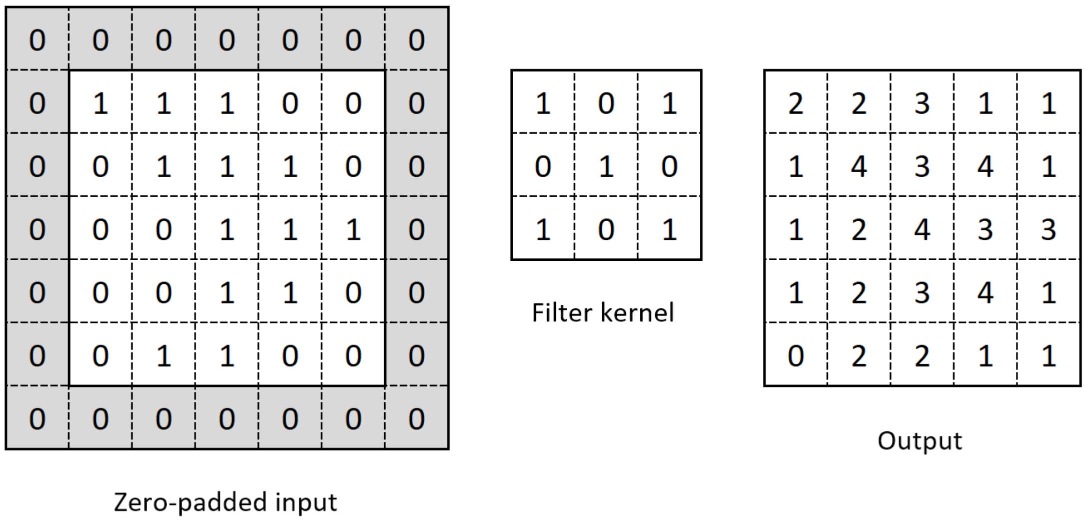


Figure 2.2 - Zero-padded input (left), with a 3 × 3 filter (mid), to achieve an output with the same size as the non-padded input (right).

2.2.1.2 Activation Layer

A convolutional layer is usually followed by an activation layer. A variety of activation functions exist, but for CNNs a Rectified Linear Unit (ReLU), shown in Figure 2.3, is generally used [19]. The activation function is applied elementwise, and is used to make the network nonlinear. This nonlinearity is what allows the neural network to model complex problems. The activation function also serves an important purpose when it comes to training the neural network. This is further explained in Subsection 2.2.3.

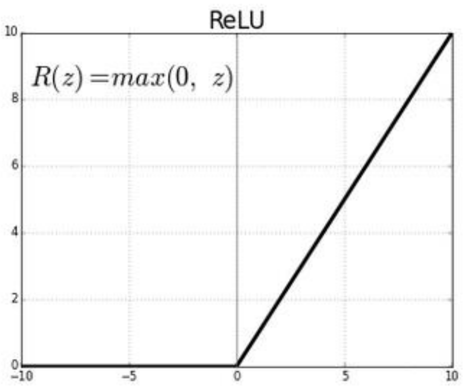


Figure 2.3 - The ReLU activation function. Figure from [21].

2.2.1.3 Pooling Layer

In the final building block, the pooling layer, the spatial size of the data is reduced, often with a pooling filter size of 2 × 2. This operation helps make the model more robust and less affected by small changes in the input data. This is useful when whether a feature is present is more important than its exact location [22].

The pooling operation finds a summary statistic of neighbouring locations, combining these into one value. Max pooling is perhaps the most used pooling operations, where the maximum value inside the pooling filter is used. An example of this pooling operation can be seen in Figure 2.4. Examples of other pooling operations that can be used include the average, the L2-

norm, or a weighted average. It should be noted that pooling is generally not done across the depth of the data.

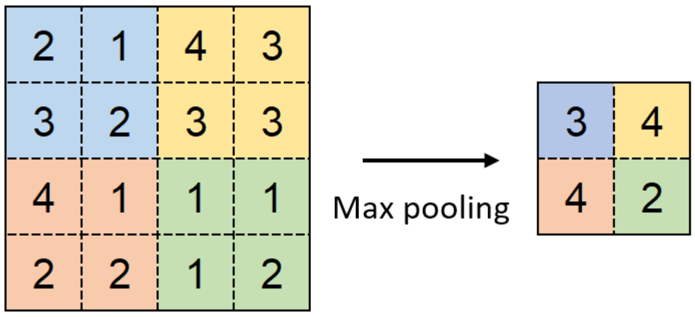


Figure 2.4 - An example of the max pooling operation, using a pooling size of 2 × 2.

2.2.1.4 Fully Connected Layers

The objective of fully connected layers is to map combinations of high level features to class probabilities. These layers are often added to the end of CNNs used for classification, and have more or less the same structure as a standard feed forward neural network. The output from the convolutional and pooling parts of the CNN is reshaped from 3D to 2D, before being fed to fully connected layers of neurons. The activation function used in these neurons will often be the same as the one used with the convolutional layers, except for the output layer. For classification, where the input only represents one class, the softmax activation function is used. The softmax function is a logistic function, but with the output squashed so that the sum of the output across all linked nodes equals 1.

2.2.2 Visualizing Convolutional Neural Networks

Understanding CNNs, however, is not that easy. Even knowing the mathematics that they are built on they can appear to be black boxes. One explanation of how neural networks work that is often given is how early layers detect simple features such as colours and lines, while later layers combine earlier features into more and more complex features. An example of this can be seen in Figure 2.5.

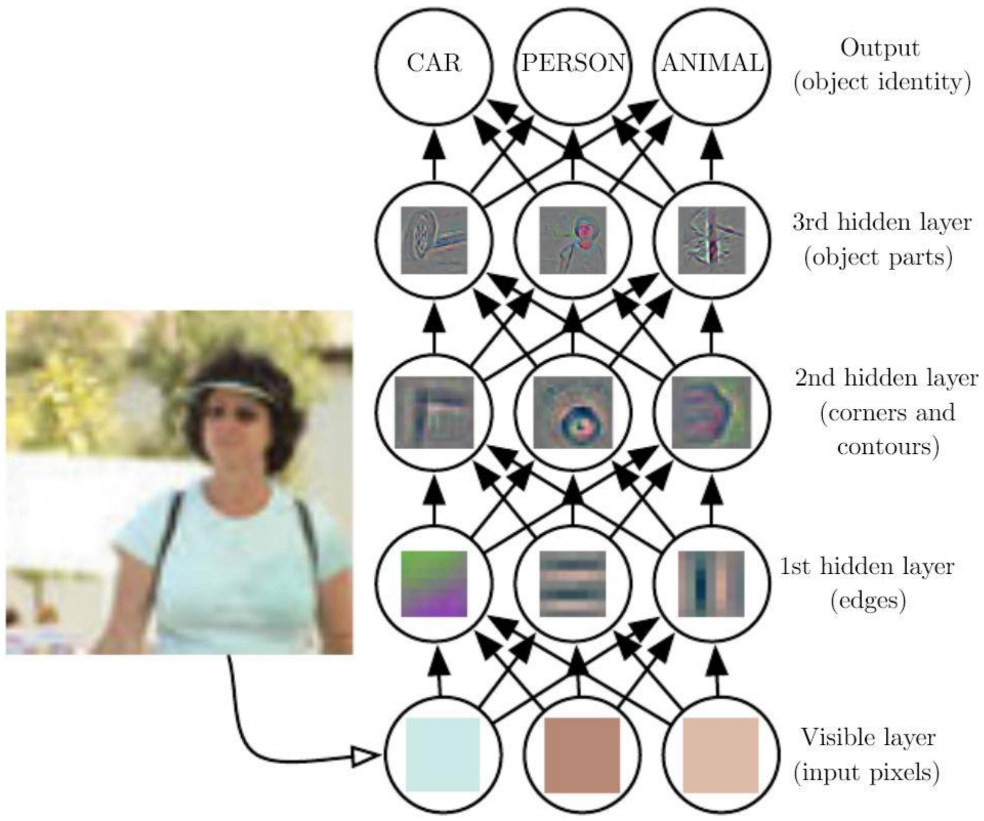


Figure 2.5 - An example of how early layers in a neural network detects simple features, such as colours and lines, while later layers combine these into more and more complex features. Figure by Zeiler and Fergus, here from [22, p. 6].

It is interesting to see what triggers various filters in different layers in a CNN, by calculating what kind of input produces a high output from a filter. An example by Chollet [23], displayed in Figure 2.6, shows this for some filters in the first 5 layers of VGG-16, a feature extractor that is described further in Subsection 2.2.4. From this it is clear that latter layers are triggered by combinations of features that triggers earlier layers.

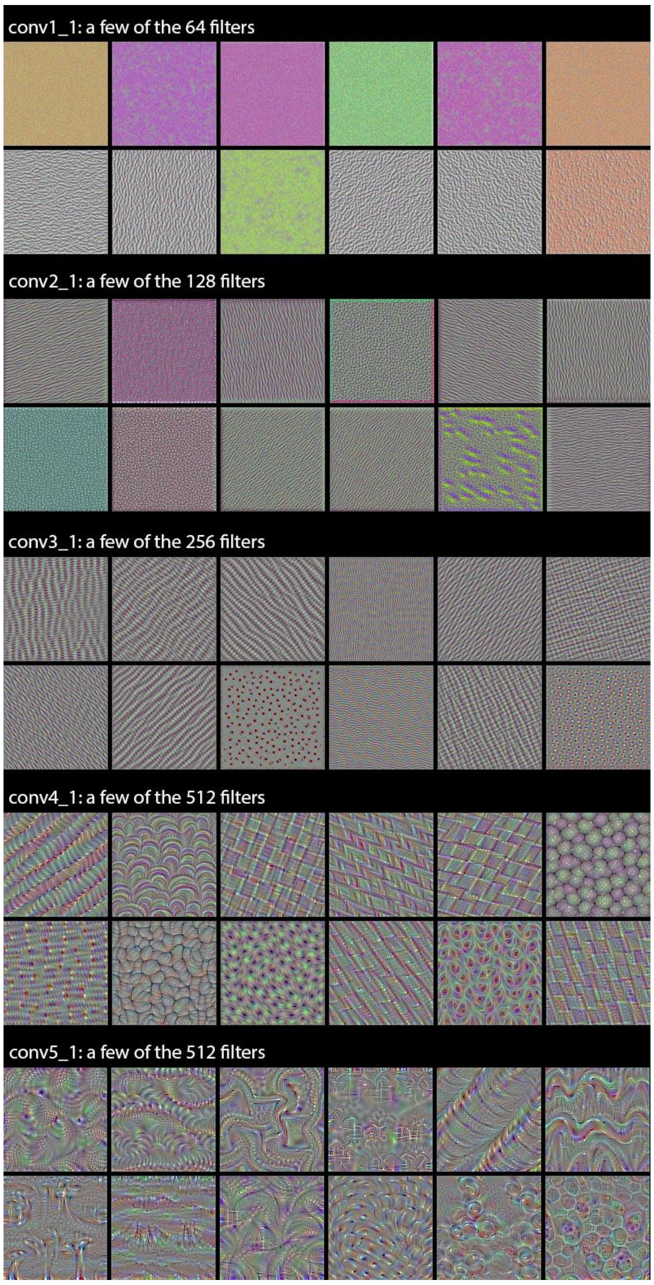

conv1_1: a few of the 64 filters
conv2_1: a few of the 128 filters
conv3_1: a few of the 256 filters
conv4_1: a few of the 512 filters
conv5_1: a few of the 512 filters

Figure 2.6 - A visualization of the preferred input to some of the filters in the first 5 layers of VGG-16. Figure from [23].

While CNNs are able to map images correctly to probable classes, they do not have the same concepts of what specific objects as humans do. They simply map combinations of various features, be it colour, textures, contours or others, to this probability [23]. This can be exemplified by generating synthetic images of what a CNN considers to be various classes. An example for bell pepper, lemon and husky can be seen in Figure 2.7. To some extent I can, as a human, agree that these are examples of bell pepper, lemon and husky, but they are not what I would visualize. That the image shown in Figure 2.8 is a magpie, however, is harder to swallow.

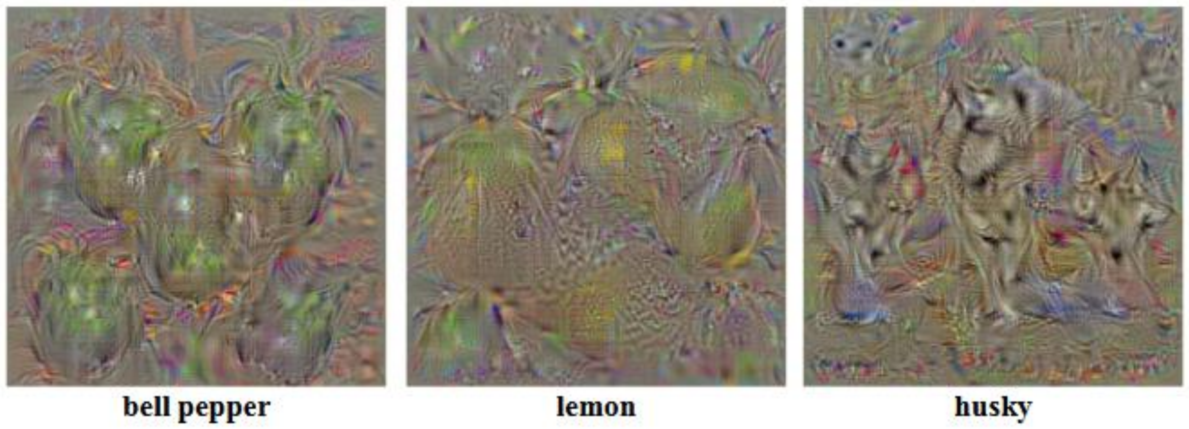


Figure 2.7 - Generated synthetic images of what a CNN considers to be bell pepper (left), lemon (middle), and husky (right). Figure from [24, p. 3].

Figure 2.8 - A generated synthetic image of a magpie, with 99.99 % confidence. Figure from [23].

Though they aren't perfect, there is no denying how effective CNNs are for object recognition. CNNs have state of the art performance in object recognition benchmarks. They are currently the best option that exists when it comes to being accurate on multi-class object recognition problems. Because of this, a CNN based model was the preferred method in this project.

2.2.3 Training Neural Networks

When it comes to training CNNs, this is usually done through supervised learning. With supervised learning, input variables and their corresponding output variables are known, and the objective of the training is to find the best possible mapping between these. This means that the supervised training of neural network can be solved as an optimization problem. The performance of the mapping between input and output is quantified by a loss, or objective function. Often used loss functions in machine learning are RMSE, entropy, L1-, and L2-norm.

This optimization problem is usually solved using gradient descent, or a version thereof. With backpropagation the gradient is propagated backwords through the various layers of the network, using the chain rule, and filter and neuron weights are updated [19]. Since neural networks often are trained on, and even require, very large datasets, calculating the gradient based on the whole dataset can be time consuming. Because of this, methods such as stochastic gradient descent and mini-batch gradient descent are often preferred. With stochastic gradient descent the gradient is calculated for backpropagation performed for each data sample, while for mini-batch gradient descent the same is done but based on a small batch of data samples.

As neural networks tend to be complex models, they are vulnerable to overfitting. Various regularization techniques are often used to prevent this from happening. One such technique pertains to the amount of data used for training. The more unique data one has, the less likely overfitting is to occur. In many cases, however, more data is not available. In some such cases data augmentation can be used. For images such augmentation can include horizontal or vertical flipping, performing various degrees of rotation, shifting hue and saturation, blurring, sharpening and cropping [25]. Another regularization technique is called dropout. With dropout randomly selected nodes in a neural network are disabled during training. This is done to prevent neurons from co-adapting too much, and is proven to be an effective regularization technique [26]. These are just two of a number of techniques.

During this project some simple data augmentation was used, as well as dropout.

2.2.4 Notable Feature Extractors

From 2012 and on research has yielded many new feature extractor architectures, with ever increasing accuracy values on benchmark datasets. Two notable feature extractors, LeNet and AlexNet, have already been mentioned in Chapter 2.1. This chapter will introduce some more feature extractors, and discuss how they differ.

While all the mentioned features extractors are built using the same basic building blocks mention in Subsection 2.2.1, the size and the number of filters used, as well as the number of layers, differ significantly. Some of these feature extractors also introduce unique structures and layers.

VGG [27] is notable in that it started using stacked layers of 3×3 filters, rather than the 9×9 and 11×11 filters used in AlexNet. In the article describing VGG it is argued that this makes the network more discriminative, that it reduces the number of parameters, and that it imposes some amount of regularization. This approach is also used by later feature extractors. The perhaps most commonly used version of VGG is VGG-16, which has 16 layers.

With Network In Network (NIN) [28] it was suggested that 1×1 convolutions could be helpful by combining higher level features after convolutions are performed. GoogLeNet [8] utilizes such 1×1 convolutions to reduce the computational complexity of operations that would otherwise be too expensive, in its Inception modules, where 1×1, 3×3 and 5×5 convolutions are performed in parallel. GoogLeNet requires a lot fewer operations compared to VGG.

With Inception V2 [29], batch-normalization layers were introduced. In these layers the output of a convolutional layer is normalized. Having all layers respond in the same range of values is something that helps during training.

ResNet [9] introduced the idea of using a bypass to skip layers. This serves two purposes. Firstly, it allows the combination of lower and higher lever features. Secondly, and perhaps more important, it makes training more efficient, allowing networks to get even deeper and more complex.

Needless to say, many of these networks have been improved upon, and many of the mentioned innovations have been combined, of ResNet and Inception V4 are good examples.

One final feature extractor that must be mentioned, is MobileNet [7]. MobileNet was designed with mobile devices in mind. It utilizes some of the mentioned innovations, such as 1×1 convolutions and batch-normalization. While it may not have as high accuracy as the newer feature extractors like Inception V3 and ResNet, it is able to achieve results similar to

GoogLeNet and VGG-16 with a lot fewer operations and parameters. This makes MobileNet a very interesting feature extractor for use in this project.

2.2.5 Convolutional Neural Network Object Detection Meta-architectures

With CNNs, a lot of progress has been made, not only with object recognition, but also with object detection. Multiple (meta)-architectures for object detection have been developed and iterated upon. The term meta-architecture is used to refer to object detection architectures that use similar approaches for detection [30].

One of the first proposed meta-architectures, was R-CNN [10]. R-CNN uses a selective search algorithm to find region proposals. An image crop of each region is then taken and passed through a CNN to extract features. Finally, a support vector machine is used to decide whether the crop contains an object, and what kind of object it is. Two issues with R-CNN is that it is slow, often with many duplicated computations, and that, because it is built up of 3 separate parts, it can be difficult to train [31].

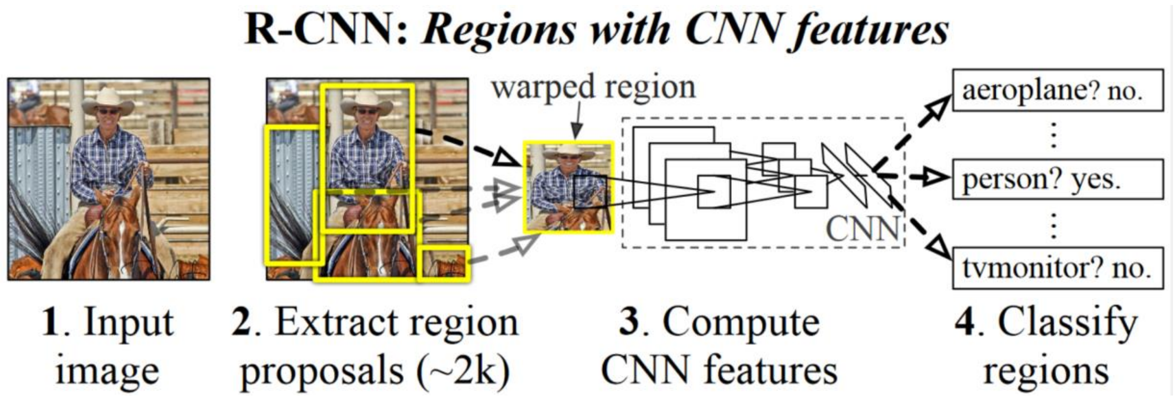


Figure 2.9 - An overview of the R-CNN architecture. Figure from [10, p. 1].

To alleviate these issues Fast R-CNN [32], and later Faster R-CNN [33], was developed. For Fast R-CNN the main change was in that the whole image is passed through the CNN once, before crops are taken from the resulting feature space. This way, features contained in areas of overlapping regions are calculated just once. Additionally, the three parts of R-CNN were joined and trained as one. An overview of the Fast R-CNN architecture can be seen in Figure 2.10. For Faster R-CNN, the main change was in how region proposals are found. Instead of using selective search, a CNN called a Region Proposal Network (RPN) is used. This network predicts regions of interest based on features calculated by a feature extractor, features that are calculated when classifying the region content anyway, and results in a significant speedup.

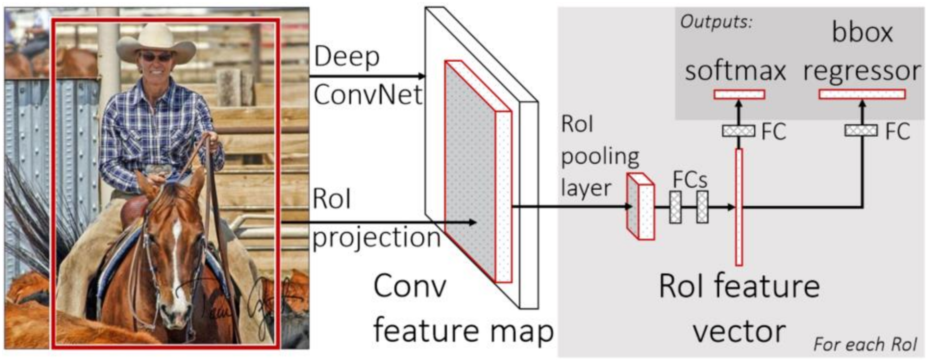


Figure 2.10 - An overview of the Fast R-CNN architecture. Figure from [32, p. 2].

A different meta-architecture, is an architecture where an image is passed once through a single CNN. While this meta-architecture is referred to as Single Shot Detector in [30], the term Single Pass Detector (SPD) will be used here, to differentiate between the SPD meta-architecture and the Single Shot Multibox Detector (SSD) architecture. The three most notable SPD architectures are Single Shot Multibox Detector [11] (SSD), You Only Look Once [34] (YOLO), and RetinaNet [35]. In these architectures coordinates for bounding boxes, classification of the content in these boxes and the confidence that an object is contained in the box, are calculated in a single pass through the network. This output is usually filtered by applying a threshold to the box confidences, and applying non-max suppression to overlapping boxes. Due to only using a single network, and only a single pass through this network for a given image, such architectures tend to be faster than Faster R-CNN.

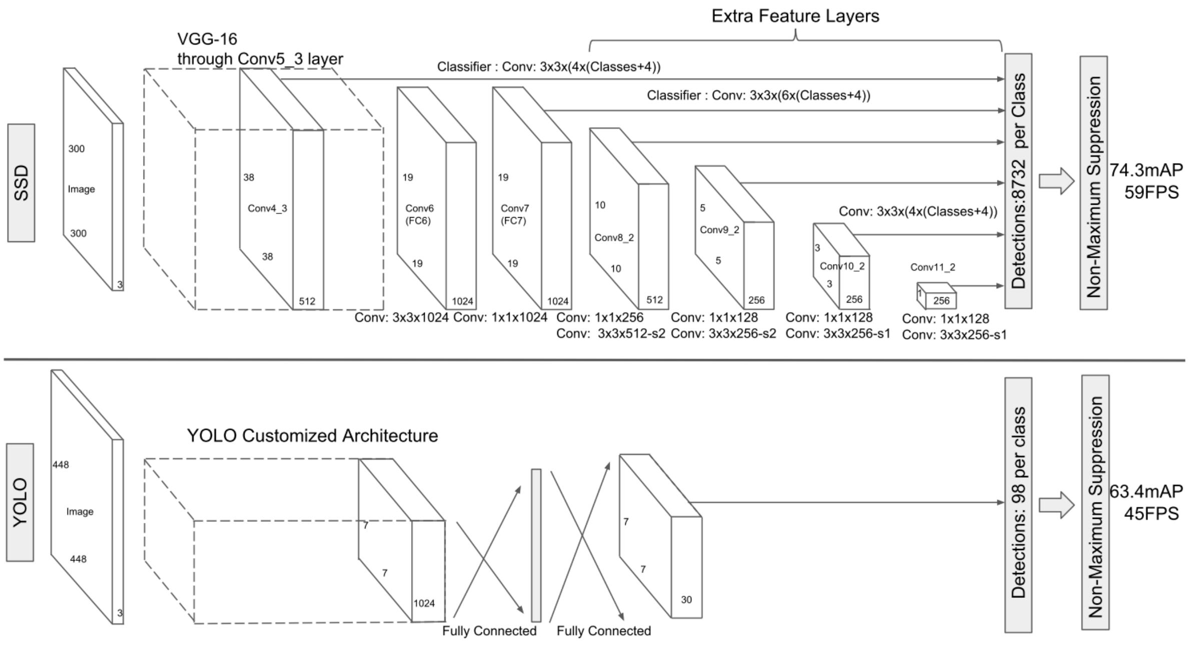


Figure 2.11 – An overview of a version of the SSD architecture (top) and the YOLOv1 architecture (bottom). Figure from [34, p. 4].

An important thing to note, is that all the object detection architectures mentioned above can make use of any feature extractor. Since CNNs are important in all of them, the choice of feature extractor will have a large impact on performance, both with regard to accuracy metrics and to speed. Figure 2.12 shows the accuracy and calculation time for various combinations of meta-architectures and feature extractors. It should be noted that all these benchmarks are performed using a NVIDIA Titan X GPU, a card with upwards of a thousand times the computational power of a Raspberry Pi.

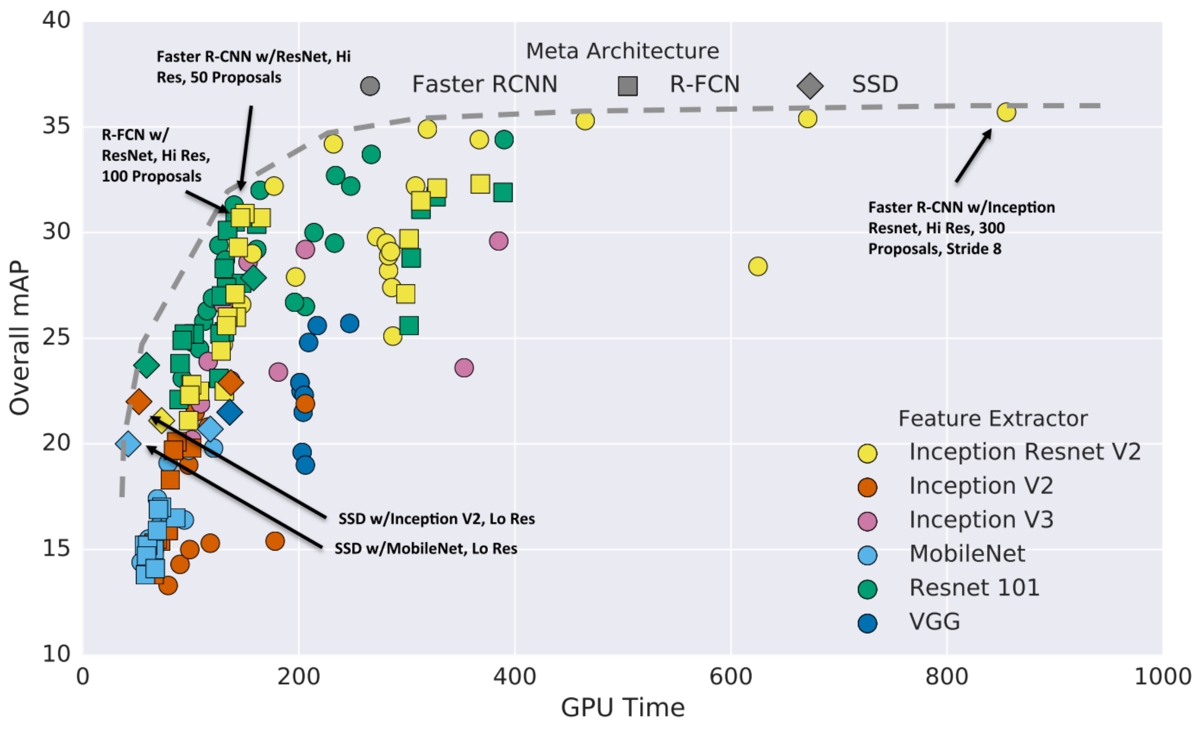


Figure 2.12 - Accuracy vs time, with marker shapes indication meta-architecture and colours indicating feature extractor. Figure from [30, p. 8] .

It should be noted that when [30] was published, YOLO was in its first version, and performed worse than SSD in all metrics, and as such, its performance is not shown in Figure 2.12. With YOLOv2 [36] and YOLOv3 [37], YOLO has seen significant improvements and seems to outperform SSD in many cases.

For use in this project, however, speed and computational complexity is perhaps the most important metric. While the most lightweight YOLO model, Tiny YOLO, reports an impressive framerate, of 244 FPS, one pass through the model requires 5.41 billion floating point operations [12]. SSD using MobileNet and an input size of 300, on the other hand, requires only 1.2 billion multiplications and additions [7]. However, compared to the low computational power of a Raspberry Pi, even these light weight models seem heavy. Because of this, a different approach has to be taken in the implemented object detection framework.

2.3 Multiple Object Tracking

Multiple object tracking in video pertains to localizing and identifying all objects of interest in a video and keeping the identities of these objects consistent between frames [38]. An example of this can be seen in Figure 2.13. Tracking can be very challenging, especially considering that objects can temporarily be occluded or leave the field of view. Objects crossing paths is another challenging aspect of object tracking.

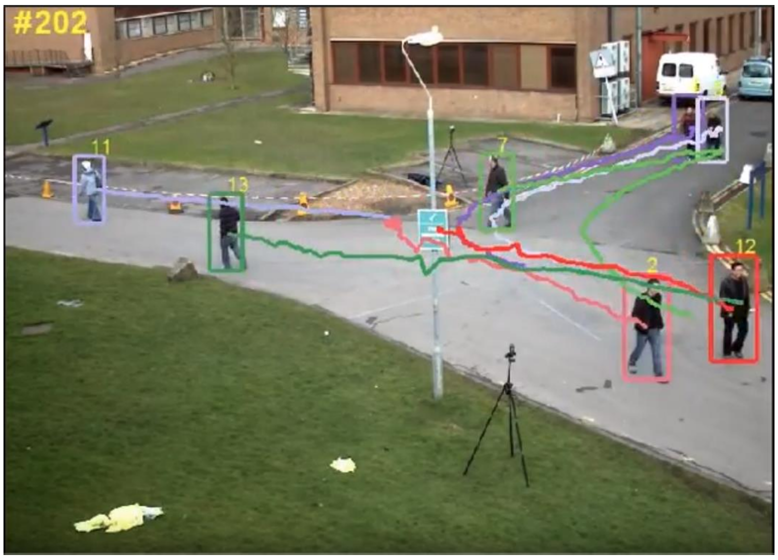


Figure 2.13 - An example of object tracking, here of pedestrians. The identity of individuals is indicated by colour and a number above their bounding box. A trail for each individual, showing their previous locations, is also included. Frame from [39].

A distinction is often made between online and offline trackers. While online trackers only have information from the current and previous frames, offline trackers are able to use information from both previous and future frames. A distinction can also be made between single and multi-class trackers. While having more classes provide more distinguishing features, it means all objects must be classified, preferably correctly.

With the advent of deep learning and CNNs, tracking-by-detection has grown popular. In this approach, all objects are first localized using an object detector. Association between objects between frames can then be made using information about this localization and other features of the objects. An example of this can be seen in Figure 2.14. In some cases, an estimator, such as a particle filter or the Kalman filter, is used to predict object features in the next frame, and the association is made between this prediction and the features collected from the next frame [38].

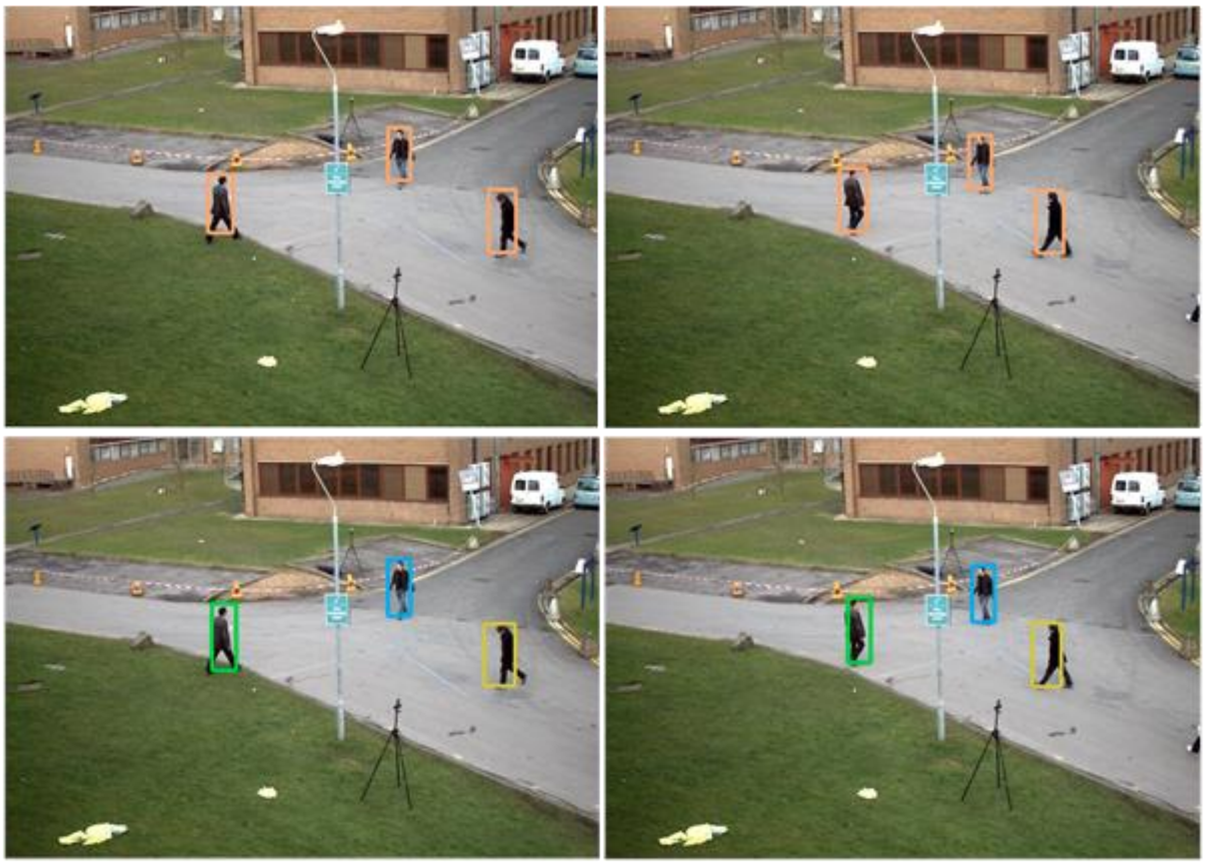


Figure 2.14 - An example of tracking by detection, here of pedestrians. The top pictures show detections in two different frames. The bottom pictures show these detections associated with individuals, across frames, as indicated by the colour of the bounding box. Frames from PETS 2009 dataset, here in [40].

Given that each object in a frame corresponds to a tracked object, the association between these becomes an assignment problem. In order to find the optimal association, a cost function calculated from one or more similarity measure is used. The solution of this optimization problem can then be found, for instance by using the Hungarian algorithm.

One often used similarity measure is the amount of overlap between bounding boxes, often labelled intersect over union (IOU). An example of what is meant by intersect and union is shown in Figure 2.15. IOU is the ratio between these, and is calculated according to equation (2.1) [41], where a and b refers to two different bounding boxes. A different, and quite self-explanatory, measure is the object class. Two objects belonging to the same class will be more similar than two objects belonging to different classes.

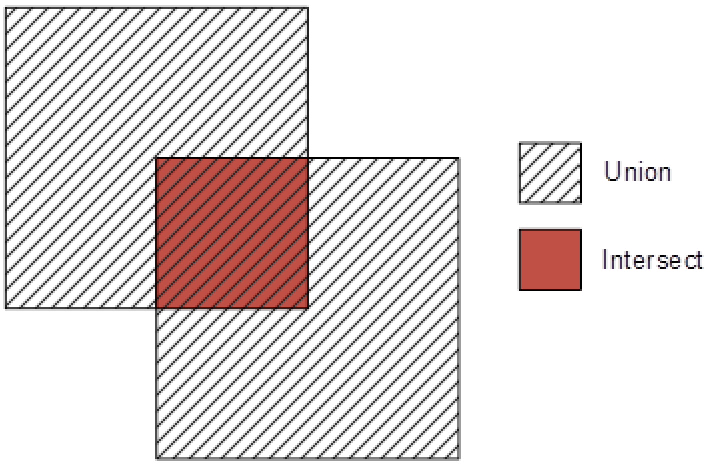


Figure 2.15 - Intersect and union between two boxes.

$$IOU(a,b) = \frac{Area(a) \cap Area(b)}{Area(a) \cup Area(b)} \tag{2.1}$$

Looking at the result chart of the MOT17 challenge [42], a benchmark or framework for evaluating various tracking algorithms, one tracker in particular stands out when it comes to speed. The IOU tracker reports to process 1522.9 FPS, a value much higher than any of the other trackers, though with slightly worse scores in other metrics compared to the most accurate tracker. The IOU tracker implements a simple algorithm, solely relying on the IOU measure, with some filtration based on confidence scores and track lengths [41]. Due to its speed and simplicity, this tracker is very relevant for the detection and tracking framework that will be implemented in this project. It does, however, come with several constraints, is very reliant on correct detections of objects.

3 Implementation

In this chapter, the implemented object detection and tracking framework is described and explained. First, an overview of the framework is given, before each used sub process is described in detail. Code for the implemented framework can be found in Appendix B.

3.1 Overview

The perhaps largest challenge in this project is the low computational power of the Raspberry Pi. State of the art object detectors perform with great accuracy, and are able to process images at high framerates, as mentioned in Subsection 2.2.5. However, this performance is achieved when using specialized hardware, with computational power many times that of the Raspberry Pi. While these object detectors can be implemented on a Raspberry Pi, the inference time will be excessively long. Even with Tiny-YOLO, a lightweight YOLO model, and code optimized for the CPU on the Raspberry Pi, the prediction time clocks in at about 1.3 seconds [43].

In order to create a framework able to achieve higher processing speeds than this, an approach other than that of using an already established object detection meta-architecture had to be taken. The object detection problem had to be simplified, so that a simpler approach for detecting objects could be used. With this, the following problem constraints were introduced:

- Stationary camera
- Background image is available
- Only 5 different classes: Ball, Car, Cup, Person, Unknown

These constraints allow for background subtraction to be used in order to find the location of possible objects in an image, a process that is further described in Chapter 3.2. Any such object can then be classified using a small object recognition model, as described in Chapter 3.3. Objects are then tracked between consecutive frames, as described in Chapter 3.4. Finally, data collected through object detection and tracking is presented as described in Chapter 3.5. An overview of this framework can be seen in Figure 3.1. The suggested framework is to some extent inspired by the R-CNN meta-architecture described in Subsection 2.2.5.

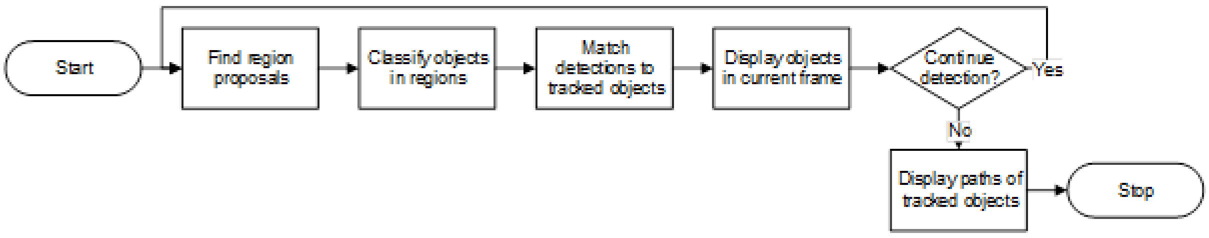

Start
Find region proposals
Classify objects in regions
Match detections to tracked objects
Display objects in current frame
Continue detection?
Yes
No
Display paths of tracked objects
Stop

Figure 3.1 - An overview of the object detection and tracking framework implemented in this project.

All examples used in this chapter are based on the images shown in Figure 3.2. Examples of results on more complex images are given in Chapter 4.

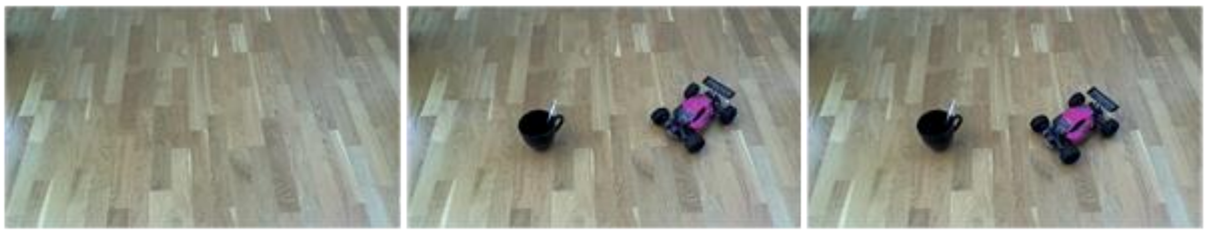


Figure 3.2 - Background image (left) and two consecutive frames (middle and right) used in examples in this chapter.

3.2 Finding Regions of Interest

The goal when finding regions of interest is to identify areas where an object is likely to be, or is, present. In our case, the coordinates of the bounding box around any such objects are sought, as shown in Figure 3.3.

To find these regions of interest in an image, the foreground in the image is segmented from the background using background subtraction. The image is scaled down to a smaller size before it is blurred. The background is then subtracted, and a binary image created. The regions of interest are calculated based on connected pixels in the binary image. This process, and the reasoning behind it, is explained in-depth in this chapter.

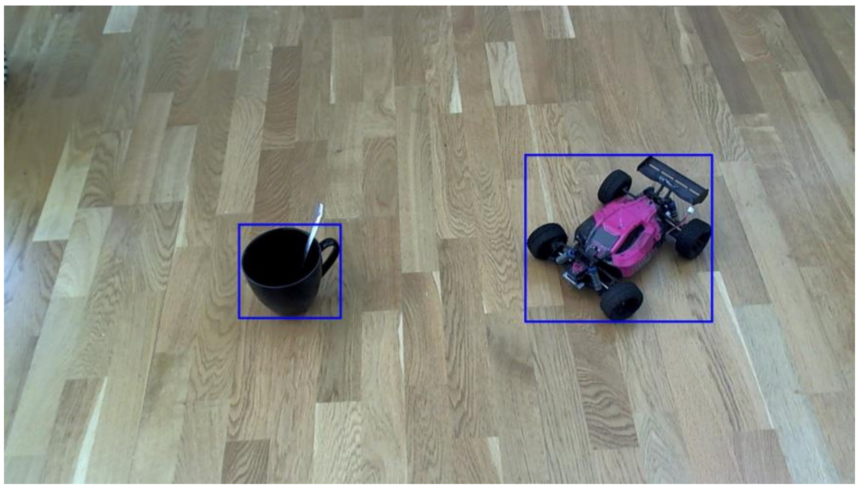


Figure 3.3 - The bounding boxes, for which the coordinates are sought, for the two objects in this example image.

The primary reason for reducing image size is to reduce the computational cost of the other operations that are performed when finding the regions of interest. For instance, reducing the height and width of an image by a factor of 0.1, reduces the number pixels by a factor of 0.01. Since the downsized image is just used for finding areas that are different in the current image compared to the background image, and not to classify the difference, the fidelity of the image is not overly important. Even so, how much it is possible to downscale an image without losing important information is dependent on the image resolution, and the relative size of any object of interest. Figure 3.4 shows an example where the height and width of an image, with a resolution of 1920 by 1080, is downscaled by a factor of 0.2, 0.1 and 0.05. With the reduction in size it gets increasingly difficult to identify what objects are present in the image. However, it should in all three cases be clear that objects are present.

While it gets increasingly difficult with the increased reduction in size, to identify what object is present in the image, it should in all three cases be clear that an object is present. Throughout this project, downscaling was done by a factor of 0.1.

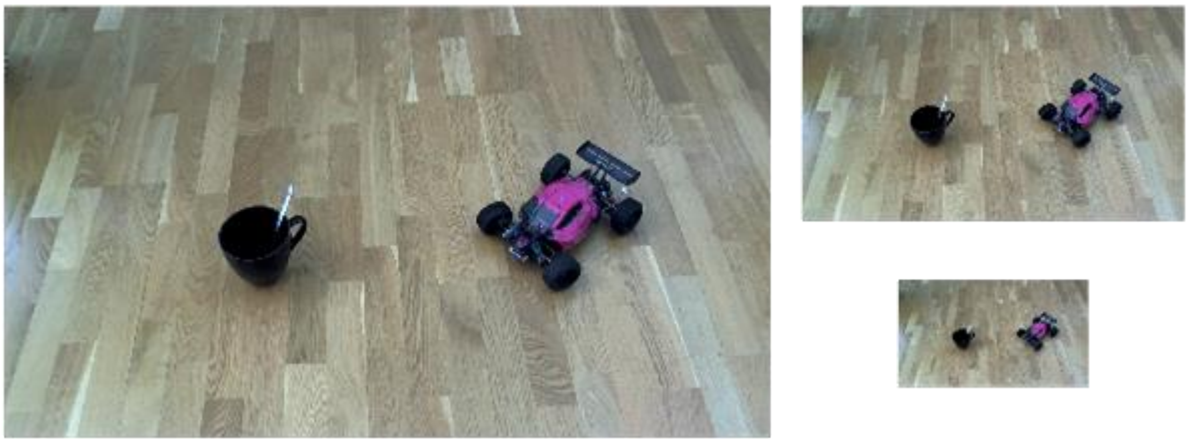


Figure 3.4 - Image with width and height downscaled by a factor of 0.2 (left), 0.1 (top right), and 0.05 (lower right). It is clear that an object is present in all three cases.

Blurring is done to reduce the impact of noise and any unwanted sharpness in an image. Several methods for blurring images exist, though maybe most notable are gaussian blur, median blur and box blur. An example of the result of using these three blurring methods can be seen in Figure 3.5. Using OpenCV and the same kernel sizes, box blur is notably faster than the other two methods. Since computational efficiency in this case is more important than image fidelity, the faster option was the reasonable choice. Furthermore, one can argue whether blurring the image is necessary when the image is downscaled significantly, as the downscaling process also reduces image fidelity. An example of the final result if no blurring or downscaling is performed, can be seen in Figure 3.7. Here, many small and fine-grained spots or flecks are present all over the image. This result is unwanted as it increases calculation time and presents a very high number of regions of no interest.

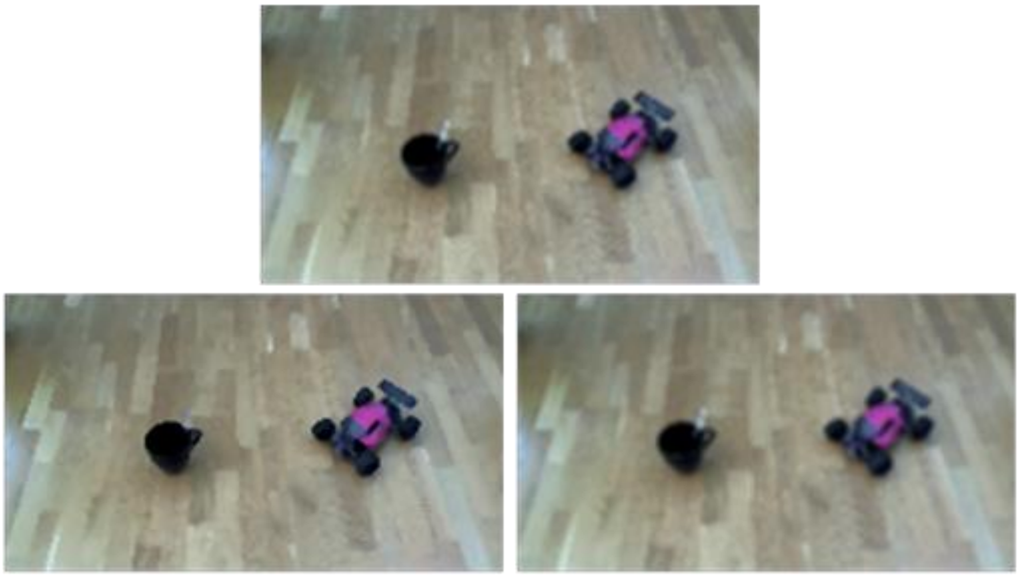


Figure 3.5 - Image blurred using three methods: gaussian blur (top), median blur (bottom left), box blur (bottom right).

When the image has been downscaled and blurred, the absolute difference between the present image and the background image, which also has been downscaled and blurred, is calculated. The calculation is done elementwise. The resulting differential image is then converted to a binary image by evaluating which pixel values are above a set threshold. An example of a differential image and the corresponding binary image can be seen in Figure 3.6.

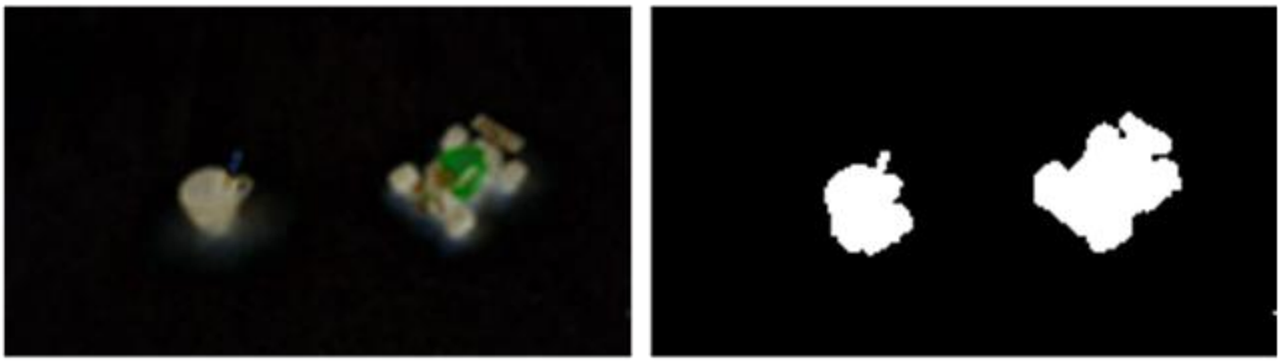


Figure 3.6 – Absolute difference between current frame and background image (left), converted to binary image through thresholding (right).

Once the binary image has been created, the regions of interest can be found by evaluating interconnected pixels. Two neighbouring pixels are considered to belong to the same region if they have the same value. The minimum and maximum row and column indices for each region are calculated, giving the coordinates for the bounding box around the region.

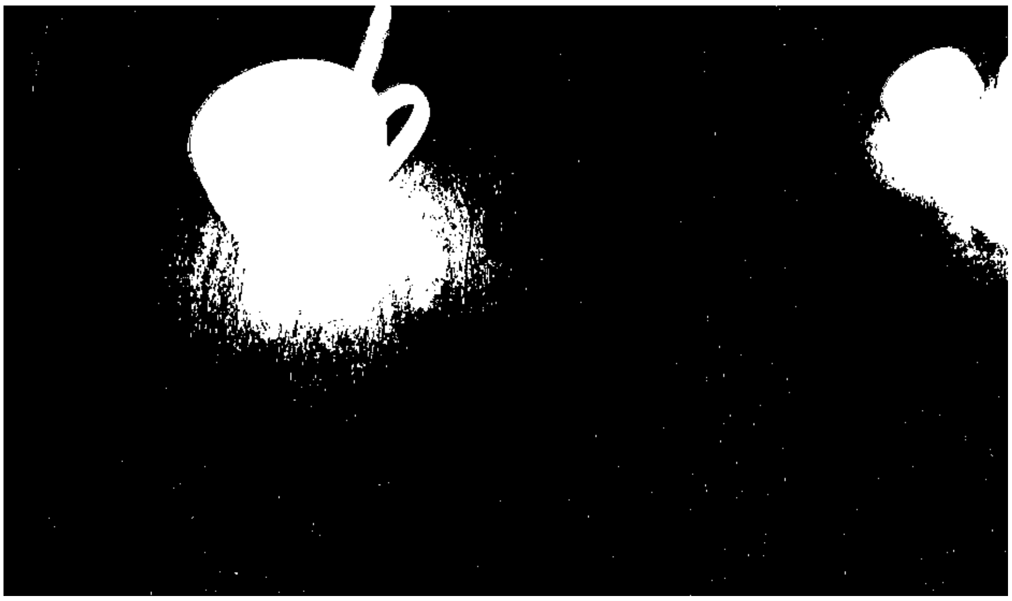


Figure 3.7 - A crop of the end result if no blurring or downscaling is performed. The image shows many small white flecks or spots not in connection to the objects that are present in the image.

In addition to performing blurring and downscaling to prevent small false negative detections, a simple filter was implemented. The filter simply discards bounding boxes with a height or width lower than a set value.

Some experimentation was done as to whether using a colour space other than RGB would yield benefits for the background subtraction, such as making the process less vulnerable to changes in lighting and image exposure. No notable difference could be seen, and it seemed that it only introduced the additional calculation cost of converting colour space. However, it should be mentioned that this could be due to the controlled environment in which the images and videos used were taken.

While traditional background subtraction, by subtracting a known background image, is used in this project, other approaches for segmenting the foreground from the background can be used instead. This could be using the differential between two or more consecutive frames for identifying movement, or more complex algorithms based around Bayesian segmentation or Gaussian mixture models. As long as the method used is able to identify the regions of interest and find the bounding boxes for these regions, it could replace the method described in this chapter.

With potential objects having been located, these objects can be classified, as described in the next chapter.

3.3 Classifying Region Content

When regions of interest in an image have been found, the content of the regions can be classified. A crop of each region, with some padding added to make sure the whole object is in the crop, is taken from the full-sized image. The crops are then scaled to fit the input size expected by the object recognition model used to classify the region content. An example of such scaled crops can be seen in Figure 3.8. The pixel values in these crops are also scaled to be between -1 and 1. Finally, the crops are input to the recognition model, which produces a probability score for each predefined class for each crop.

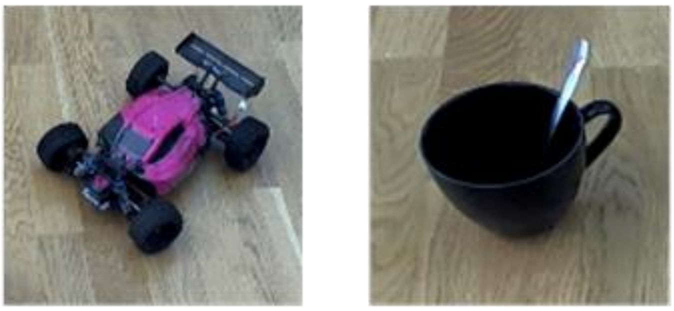


Figure 3.8 - Scaled crops taken from the full-sized image.

The neural network used to classify objects was built around the MobileNet feature extractor. As discussed in Subsection 2.2.4, MobileNet is a fast and efficient feature extractor, albeit with a poorer accuracy performance than other larger and slower feature extractors. Its speed and efficiency made MobileNet the natural choice in this project.

Instead of building and training the neural network from scratch, which is a very time-consuming process, the smallest and most lightweight model of MobileNet, with weights trained on the ImageNet dataset, was repurposed and retrained to classify the five classes used in this project. The fully connected layers of the original model where replaced, while the convolution and pooling layers remained the same, before the model was retrained. The model was retrained on a manually, and somewhat arbitrarily, selected subset of the ImageNet and Coco datasets. Approximately 1300 images were used for each category. Some pictures of RC-cars had to be substituted into the car category in order to achieve descent classification accuracy on these. Transfer learning, where a model trained on a general dataset, such as ImageNet, is repurposed and retrained for use in more specialized cases has proven to be an effective method [44].

The network expects a 4-dimensional array as input. The first dimension is used to index an image or crop, with a size equal to the number of images in the array. The second and third dimensions are used to index image height and width, respectively, both having a size of 128. Finally, the fourth dimension is used to index the channels in the RGB colour space, with a size of 3. In other words, the expected input has the dimensions $N_I \times 128 \times 128 \times 3$, where N_I, the number of images, can vary.

The output of the network is a 2-dimensional array, with the first dimension corresponding to the first dimension of the input, and the second dimension corresponding to the classes used by the network.

When objects have been located and classified, they can be tracked, as explained in the next chapter.

3.4 Tracking Objects

With objects having been detected, found and classified in a frame, it must be established whether these objects are instances of objects tracked from the previous frame or if they represent new objects. This is done by matching detections in the current frame with detections in the previous frame, using a cost function subject to optimization with the Hungarian algorithm. An example of such detections in consecutive frames, can be seen in Figure 3.9

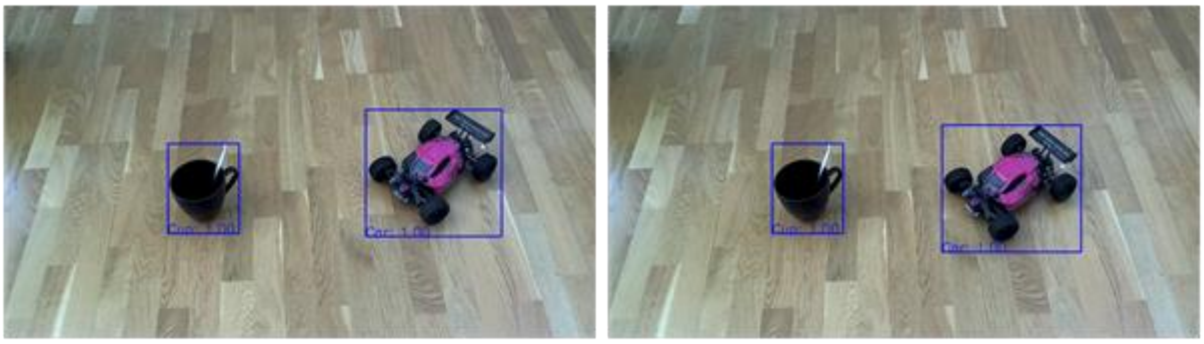


Figure 3.9 - Objects having been detected and classified in two consecutive frames, frame 1 (left) and frame 2 (right).

The IOU-tracker mentioned in Chapter 2.3 makes the foundation for the tracker used in this project. However, the MOT challenge is a single class tracking problem, while this project presents a multi-class tracking problem. Because of this, the similarity measure of object class is used in addition to IOU. A weighted sum approach is used to combine these two measures, with IOU having a weight of 1, and same class categorization adding a value of 0.1 when classes are the same. An example of a cost matrix built for the frames in Figure 3.9, can be seen in Figure 3.10. Here A and B refer to two already tracked objects, and a and b refers to two new detections. IOU is calculated as in equation (2.1), and SC as in (3.1), where a and b refer to two detections, and $Class(\cdot)$ is the classification of the object. As both a high IOU and same class categorization suggests a good match between objects, and the cost function is subjected to maximization.

$$SC(a,b) = \begin{cases} Class(a) = Class(b) & 1 \\ Class(a) \neq Class(b) & 0 \end{cases} \tag{3.1}$$

	A	*B*
a	*1•IOU(a,A) + 0.1•SC(a,A)*	*1•IOU(a,B) + 0.1•SC(a,B)*
b	*1•IOU(b,A) + 0.1•SC(b,A)*	*1•IOU(b,B) + 0.1•SC(b,B)*

	A	*B*
a	0,65	0
b	0	1,1

Figure 3.10 - An example of a cost matrix used to match detections with tracked objects, with values based on the detections in Figure 3.9.

Once solved, each assignment is checked to see if it is a good match. This is done by evaluating whether the assignment cost is above a set threshold. Assignments with a cost higher than the threshold are assumed to be correct. Unassigned detections in the current frame, and assignments with a low cost, are assumed to be instances of new objects, and are set to be tracked as such. Unassigned detections in the previous frame, or with assignments with low cost, are assumed to be objects that no longer are within the field of view, and tracking of these objects is stopped.

Each tracked object is assigned a unique ID. It is also given a colour, which is used when annotating images. While not unique, these colours are assigned sequentially from a list of colours. Additionally, the following information is stored for each tracked object for each frame where the object is detected:

- Frame number
- Bounding box coordinates
- Class probability score

This data can be presented to the user, as described in the next chapter.

3.5 Presenting Detection and Tracking Data

When the object detection and tracking procedures have been completed, the collected data is presented to the user. Detections are displayed for each frame, with a bounding box drawn around each object and the classification class confidence score annotated in the lower left corner of the box. Colours are used to indicate whether an object is the same one tracked in the previous frame. An example of this can be seen in Figure 3.11.

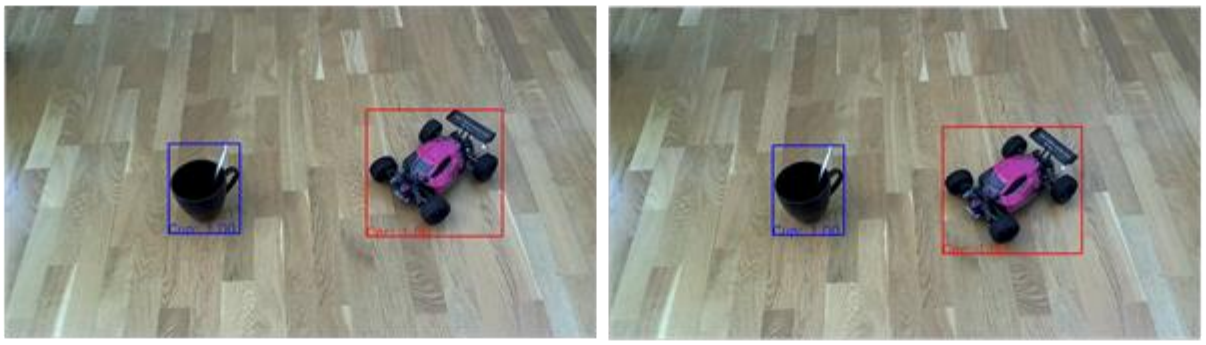


Figure 3.11 - Two consecutive frames, showing detections using bounding boxes, annotations and colours.

A history of where tracked objects have been located, and their classification when in that location, can also be displayed. This information is shown as a line plot, with bounding box centres used to describe object location, marker shapes used to show classification, and colours used to differentiate between tracked objects. This plot is overlaid on the background image in order to give the plots some context. An example of this data presentation, for the example images used in this chapter, can be seen in Figure 3.12.

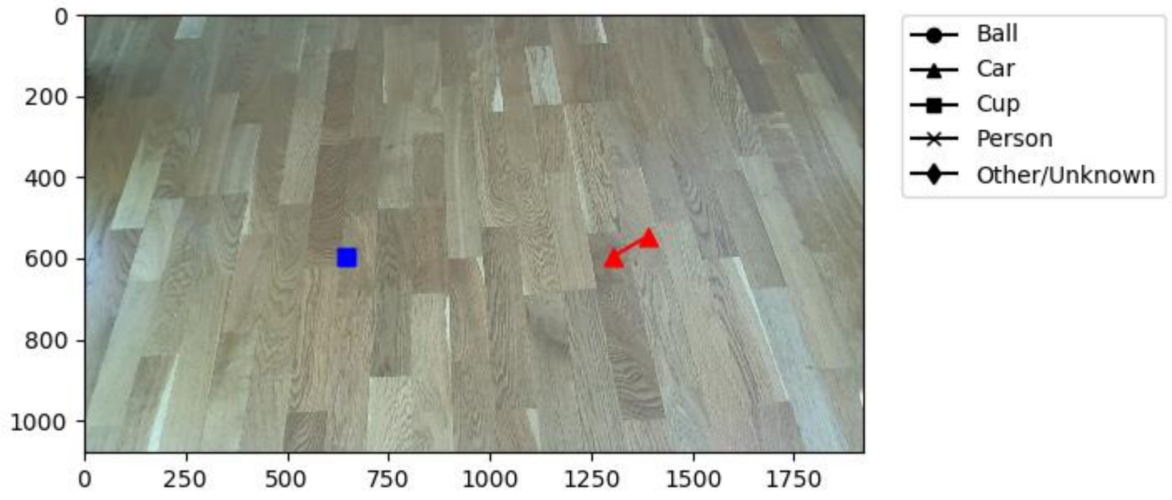


Figure 3.12 - A plot showing the history of tracked objects, with bounding box centres being used to describe object location. Mark shapes are used to describe object classification and colours are used to differentiate between tracked objects.

4 Results

In this chapter, results from the suggested framework are shown and discussed. First, the performance of the object localization is evaluated, before the object recognition model and tracking algorithm are assessed. Finally, the processing speed of the framework is reviewed. As no benchmark dataset suitable for the implemented framework was available, the presentation of the performance results is to a large degree anecdotal.

4.1 Object Localization Performance

Background subtraction was the chosen method for locating objects in images. Given the constraints of a stationary camera, and a known background image, this is seen to work well in many cases. An example of this can be seen in Figure 4.1. The method, however, is not without flaws.

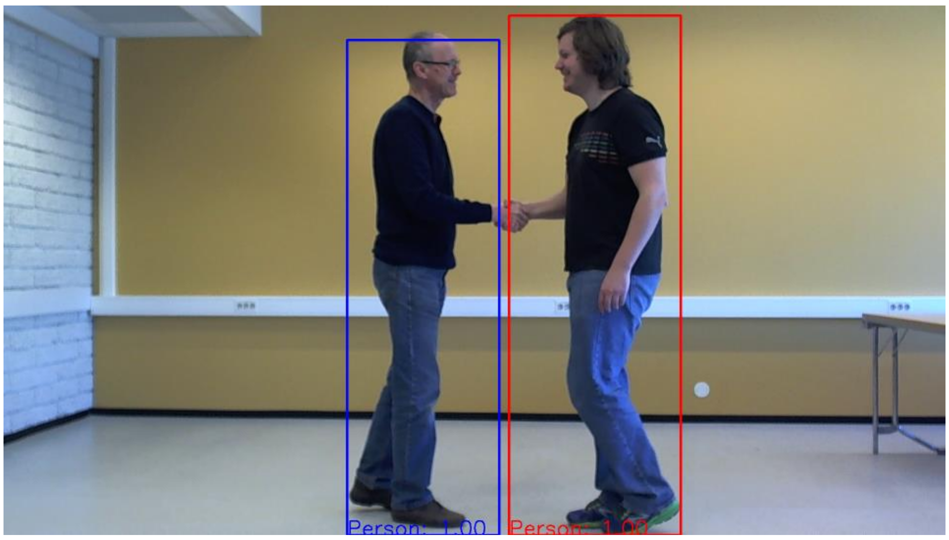


Figure 4.1 - An example of when the suggested framework works well.

One issue is if the lighting in a scene, or if the exposure to the camera's image sensor, changes. This will cause new images to differ significantly from a previously taken background image. In a best-case scenario this is likely to lead to false positives, while in a worst-case scenario it might render the suggested method unusable. An example of such false positives can be seen in Figure 4.2.

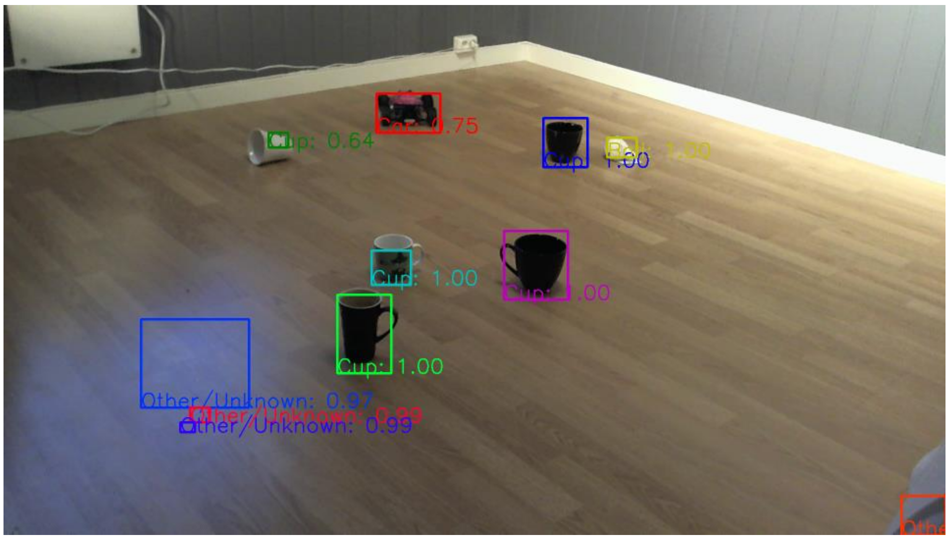


Figure 4.2 – An example of false detections of objects due to changing in light or camera exposure.

Another issue, which will always be present, is that the suggested method for locating objects is poor for handling occlusion. If only one part of an object is visible, while the rest is occluded, the method might work fine. However, if multiple non-connected parts of an object are visible, each part will be detected as a separate object, something that can be seen in Figure 4.3.

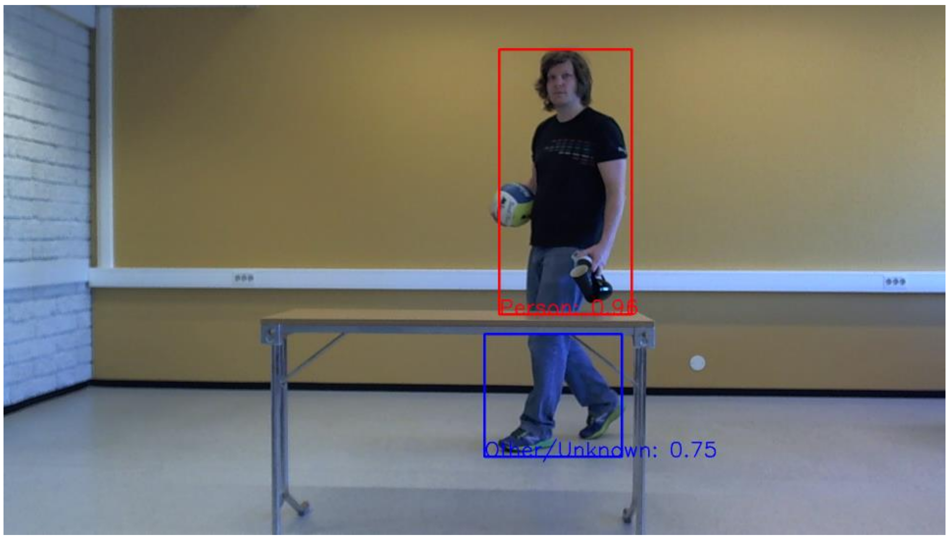


Figure 4.3 – An example of occlusion causing one object to be detected as two.

Finally, if two objects get too close to one another other, they will be understood to be one object. An example of this can be seen in Figure 4.4. This lack of ability to differentiate between objects in close proximity severely impacts the rest of the object detection model, as it makes object recognition more difficult, with multiple objects being present in the image crop. It will also affect object tracking, as certain objects will no longer be detected.

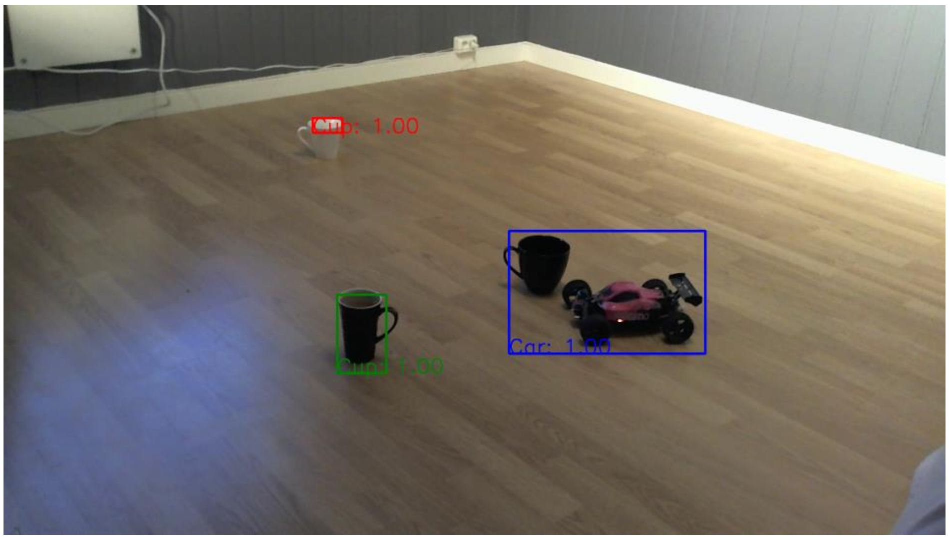


Figure 4.4 - An example of objects in close proximity being detected as one object.

All these issues were expected, as a price paid in the attempt to achieve as high processing speeds as possible. The speed performance of the suggested framework is described in more detail in Chapter 4.4.

4.2 Object Recognition Performance

The classification performance of the CNN used to classify objects, was evaluated on a small set of test images. An example of the classification of 9 different images, of which none were used for training or validation, can be seen in Figure 4.6. Except for the top left of these example images, the classification is near perfect. The exact reason for why the volleyball in the top left image is classified as a cup is not clear. Using a more closely cropped image of this ball, however, produces the correct classification with close to 100 % confidence. During training, the object recognition model was able to achieve an accuracy of 91.2 % on the validation dataset.

In photos and videos taken of fast moving objects, the problem of motion blur appeared. An example of two images with such blurring can be seen in Figure 4.5. This blurring affected the classification of the objects significantly, often with the object being classified as “unknown”. Using slightly blurred images in the training dataset might have made correct classification easier for cases where some blurring was present. In images with severe blurring, however, correct classification cannot be expected.

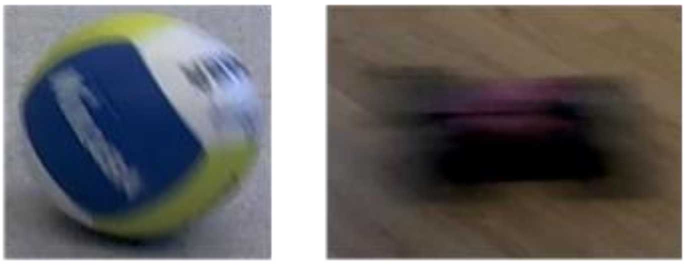


Figure 4.5 - Two images with motion blur, which makes classification difficult.

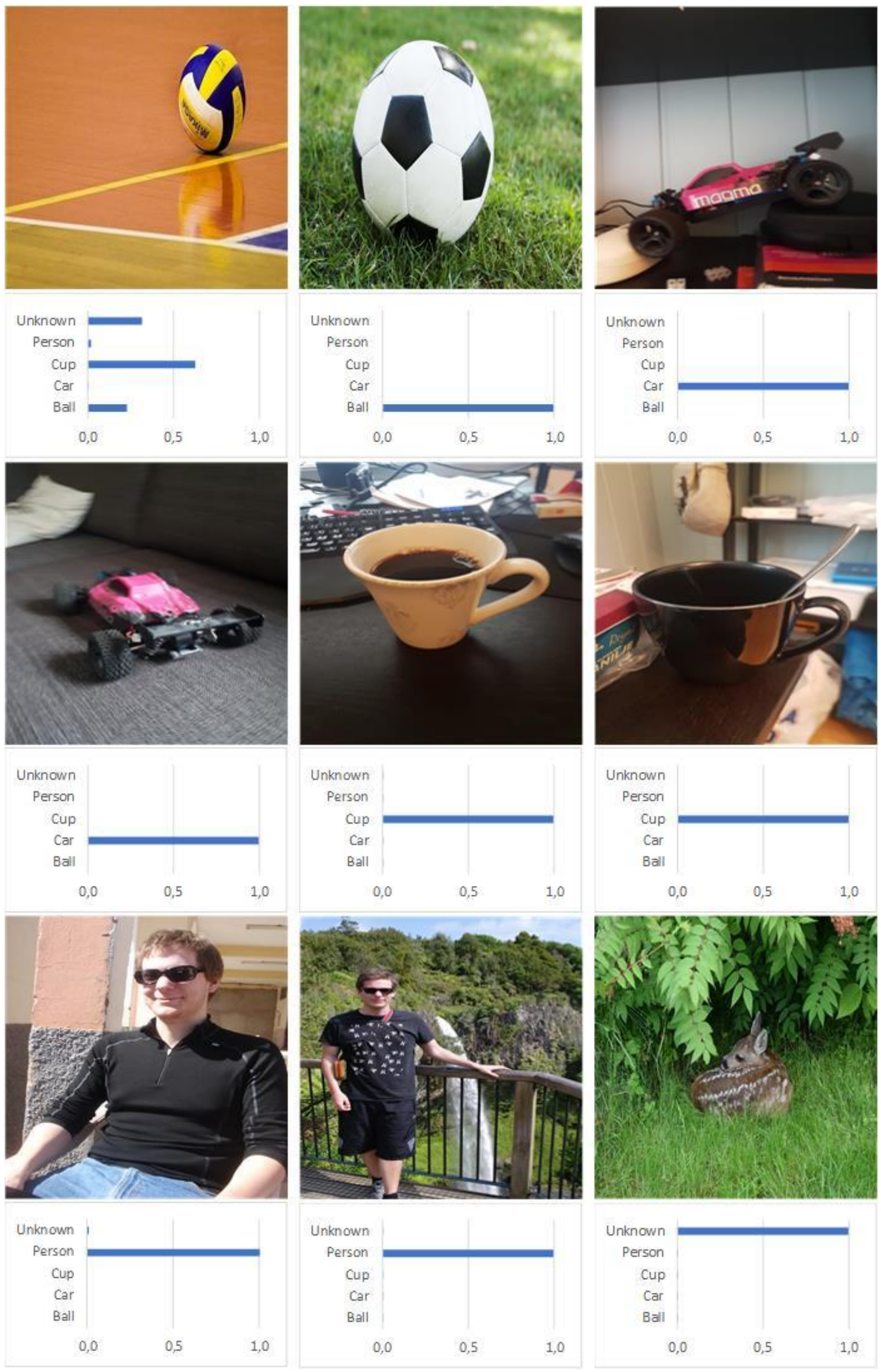


Figure 4.6 - The classification output from the object recognition model, for 9 different images.

4.3 Tracking Performance

The implemented tracking algorithm is highly reliant on correct detections. Needless to say, if detection fails, so will tracking. Tracking of moving objects is also reliant on how fast an object moves, and at which framerate images are taken. Figure 4.7 shows the tracking history for an RC-car entering from the right, using video with framerates of 7 FPS and 5 FPS.

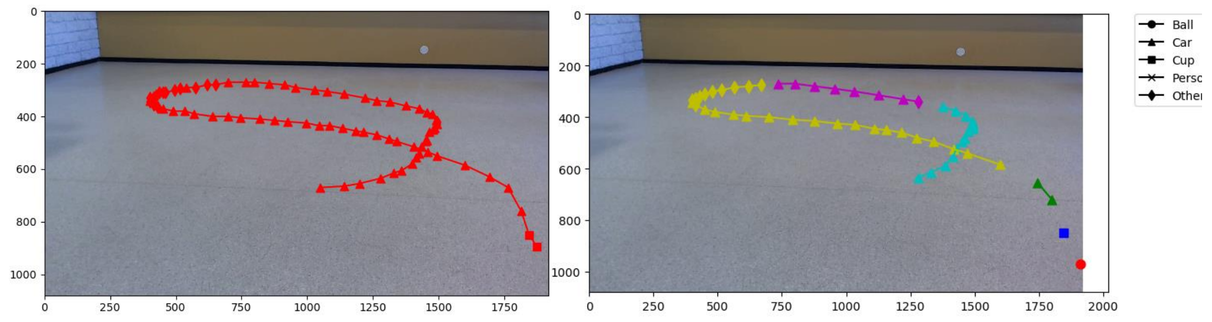


Figure 4.7 - Plots showing the tracking history for an RC-car driving in a circle, using videos with 7 FPS (left) and 5 FPS (right). It is clear that proper tracking starts to break down when objects move too far between frames.

This example shows that, when objects move too far between frames, due to a high speed, low framerates, or a combination of these, the tracking of these objects starts to break down. If the object moves too far, detections of this object are no longer understood to be instances of the same object. This can be seen going from green to yellow in Figure 4.7. Changes in the classification of the object further exacerbates this, as seen in going from red to blue to green, and from yellow to magenta, and from magenta to turquoise. This is to be expected, due to the high reliance on IOU to associate detections between frames.

4.4 Framework Speed Performance

The speed of the suggested framework was measured by timing the various components of the framework over a number of iterations. A set of images containing 1-6 objects, shown in Figure 4.8, were used, with predictions being performed on each image 1001 times. The average timings over 1000 iterations could then be calculated. The first time object recognition is performed is always notably slower than the rest, and is, for this reason, not included in the average timings. The results are presented in Figure 4.9 and Figure 4.10, with values shown in Table 4.1.

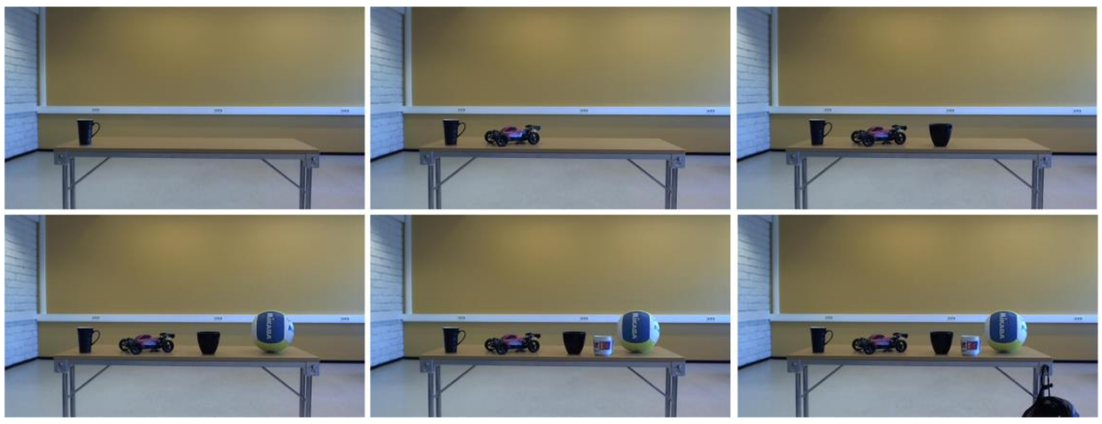


Figure 4.8 - The set of images, containing 1-6 objects, used to analyse the speed of the suggested framework.

As can be seen from these results, the primary contributor to the total processing time is the classification process. The time spent on classifying regions increases significantly with increased number of objects, which is to be expected, as calculations in this process are performed per object.

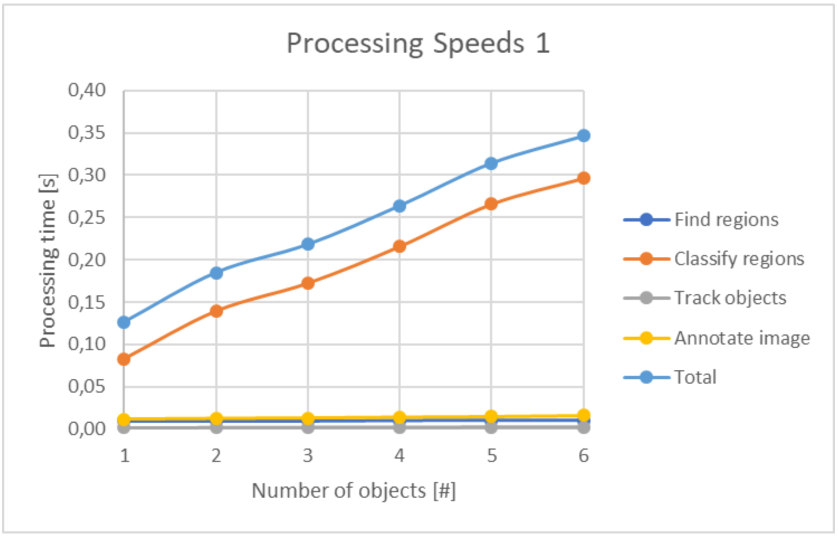


Figure 4.9 - Processing speed for all parts of the framework, as well as the total.

Tracking, on the other hand, did not notably affect the processing speed. While time spent on tracking does increase with the number of objects, this increase is so small that, when compared to the other timings, it is nigh imperceptible.

The time it takes to find regions of interest does not increase notably with an increased number of objects, either, which is to be expected. Most of the calculations done in this process are done regardless of how many objects are present, and only a small number of calculations are done per region found.

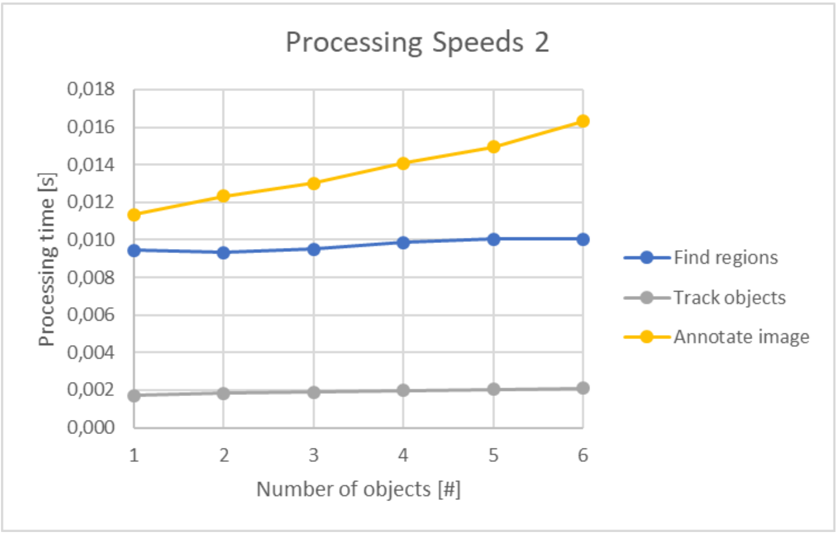


Figure 4.10 - Processing speed for the three fastest parts of the framework.

The only real surprise, was how long it took to annotate an image by drawing bounding boxes and writing class confidence scores. This process should not be computationally expensive, and the fact that it takes longer to annotate an image than to find regions of interest is astonishing. Preliminary investigations suggest this has to do with how a copy of the original image is being created. Compared to the classification process, however, annotating does not contribute that much to the total prediction time. The time it takes to annotate an image scales with the number of objects, which is to be expected.

Table 4.1 - The timings, in seconds, for the various components in the framework, calculated for 1-6 objects.

	1 object	2 objects	3 objects	4 objects	5 objects	6 objects
Find regions	0,009	0,009	0,010	0,010	0,010	0,010
Classify regions	0,083	0,139	0,172	0,216	0,266	0,297
Track objects	0,002	0,002	0,002	0,002	0,002	0,002
Annotate image	0,011	0,012	0,013	0,014	0,015	0,016
Measured total	0,126	0,185	0,218	0,264	0,314	0,346

The suggested framework is significantly faster than Tiny-YOLO and SSD300, assuming processing time for Tiny-YOLO to be as reported in [43] and processing time for SSD300 with MobileNet to be as measured using djmv's OpenCV implementation [45]. Processing times for these frameworks can be seen in Table 4.2. It should be noted that tracking is not included for Tiny-YOLO and SSD300.

Table 4.2 - Processing time and framerates for 3 different object detection frameworks.

	Ours, 1 object	Ours, 6 objects	Tiny-YOLO	SSD300 with MobileNet
Processing time [s]	0,126	0,346	1,2	0,95
FPS [Hz]	7,9	2,9	0,8	1,1

When only one object is present, the suggested framework is almost 10 times as fast as Tiny-YOLO and 7 times as fast as the implementation of SSD. If 6 objects are present, these values drop to 3.6 and 2.6, respectively, which is still a significant speedup. This speedup, however, does not come without a cost. The suggested framework comes with significant constraints, and is severely impacted by the number of detected objects, something the other methods are not.

While running the suggested framework on a Raspberry Pi it was noted that the CPU usage fluctuated around 54-60 %, across all cores. This observation was made using htop. The framework itself is not multi-threaded, though some of the used libraries, TensorFlow for instance, are. The fact that CPU usage was not capped out during inference suggests that something was bottlenecking the CPU. The performance of the suggested framework would improve if this bottleneck could be avoided. Some speculation of how this could be done is performed in Chapter 5.

5 Discussion

In this chapter, various possible ways of improving the speed of the suggested framework are discussed.

5.1 Why use Python

A very reasonable question to ask, is: “Why would you ever use Python, when you are working with a low power device, and speed is of the essence?”. It is a well-known fact that Python has a high overhead, especially when compared to languages such as C++. C++ would in many ways have been a more logical language to use when implementing the suggested framework.

The reason for using Python is quite simple. At the start of this project the author had little experience with deep learning and neural networks, and great high-level machine learning libraries, Keras in particular, exist for Python, and not for C++. Additionally, the online community for machine learning that uses Python is really quite great. A number of good tutorials and guides exist. The though was to use Python initially, for then to port the framework over to C++. Sadly, there wasn’t enough time within the timeframe of the project to do this.

It should be noted, however, that the most computationally expensive operations are done using highly optimized libraries written in C and C++, such as NumPy, OpenCV and TensorFlow. An attempt was made to make the impact of the increased overhead from Python as small as possible, for instance by using vectorized computations in order to avoid loops in Python. So, while porting the framework to C++ is likely to provide a speed increase, this increase might not be as large as one would hope.

5.2 Quantization and Network Pruning

The largest contributor to the calculation time of the suggested network, by far, is object recognition. So, is there any way of speeding this up? After all, one of the most light-weight CNN architectures is already used.

One approach would be to use an even less complex CNN. This could be achieved by building and training a new model, with fewer layers etc, from the ground up. A different approach would be to remove layers that aren’t important. It is this latter approach that is referred to as network pruning.

Network pruning is proving to be quite effective. In a recent article [46] it is shown that the number of floating point operations in a model can be reduced by approximately 80 %, while only losing 3.4 percentage points in accuracy. It should be noted that these values are for a network with a more complex feature extractor than the one used in this project, and because of this that pruning is unlikely to be as effective here. Even so, it would have been interesting to see what results could be achieved by pruning the implemented object recognition model.

Another approach would be to use a quantized network [47]. Quantization is a method for representing values in a model using lower resolution representation, for instance 8-bit, without notably affecting the networks accuracy. One of the benefits of using 8-bit data representation, compared to the standard 32-bit, is that it uses 25 % of memory bandwidth. If memory bandwidth was what bottlenecked the CPU, as noted in 4.4, this could be a possible solution. In some cases, calculations based on 8-bit fixed point data representation are also faster, enabling the use of more specialized hardware such as SIMD architectures.

Due to time constraints these approaches could not really be explored in this project.

6 Conclusion

In this project, an attempt was made to develop an object detection and tracking framework able to run in real-time on a Raspberry Pi 3 Model B.

Multiple state of the art object detection algorithms, Tiny-YOLO and SSD300 with MobileNet in particular, were considered and discarded due to their computational complexity. As an alternative, a scheme of using background subtraction to locate objects, which could then be classified by light weight object recognition model, was suggested.

This required a stationary camera, and it was assumed that a background image was available. The object recognition model used the MobileNet feature extractor, and was adapted from a model trained on the ImageNet dataset. Tracking was performed using a modified IOU-tracker, which associated detections between frames based on location and object classification.

The suggested framework was able to achieve framerates as high as 7.9 FPS when 1 object was detected, and 2.9 FPS when 6 objects were present. These values are significantly higher, more than 7 times for 1 object and 2.6 times for 6 objects, than those achieved using state of the art models. This increase in speed, however, does not come without a price.

While the suggested framework was seen to work well in many situations, it does have several weaknesses. Some of these include poor handling of occlusion, a lack of ability to distinguish between objects in close proximity, and false detections when lighting conditions change. Additionally, the processing speed of the suggested framework is not deterministic, as its processing times are highly dependent on the number of detected objects. This is true for state of the art models as well, though to a much smaller extent.

6.1 Suggestions for Further Work

Much of the accuracy performance of the suggested framework stands or falls with the success of the background subtraction method. Chapter 3.2 briefly mentions alternative methods for background subtraction. It would have been interesting to see how the suggested framework would have been affected by the various approaches to background subtraction.

The current implementation is in Python, a language known for its overhead. Porting the framework to C++, as discussed in Chapter 5.1, is likely to increase the speed of the suggested framework.

The largest contributor to the calculation time of the suggested framework is object recognition. Two possible ways of speeding up this process, through quantization and network pruning, are discussed in Chapter 5.2. Whether the object recognition process could be sped up is something that should be looked further into.

References

[1] R. Longbottom, “Roy Longbottom's Raspberry Pi, Pi 2 and Pi 3 Benchmarks,” 5 2017. [Online]. Available: http://www.roylongbottom.org.uk/Raspberry%20Pi%20Benchmarks.htm#anchor24b. [Accessed 3 4 2018].

[2] M. Larabel, “Raspberry Pi 3 Model B+ Benchmarks,” 22 3 2018. [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=raspberrypi-3-bplus&num=1. [Accessed 20 4 2018].

[3] E. Upton, “Raspberry Pi 3 Model B+ on sale now at $35,” The Raspberry Pi Foundation, 14 3 2018. [Online]. Available: https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/. [Accessed 20 4 2018].

[4] S. M. Kuo, B. H. Lee and W. Tian, Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications, Chichester: John Wiley & Sons, Incorporated , 2013.

[5] R. B. Fisher, T. P. Breckon, K. Dawson-Howe, A. Fitzgibbon, C. Robertson, E. Trucco and C. K. I. Williams, Dictionary of Computer Vision and Image Processing, Chichester: John Wiley & Sons Ltd, 2014.

[6] A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” 2012. [Online]. Available: https://www.cs.toronto.edu/~fritz/absps/imagenet.pdf. [Accessed 7 2 2018].

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, A. Weyand, M. Andreetto and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” 17 4 2017. [Online]. Available: https://arxiv.org/pdf/1704.04861. [Accessed 5 2 2018].

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, “Going deeper with convolutions,” 17 9 2014. [Online]. Available: https://arxiv.org/abs/1409.4842. [Accessed 7 2 2014].

[9] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” 10 12 2015. [Online]. Available: https://arxiv.org/abs/1512.03385. [Accessed 8 2 2018].

[10] R. Girshick, J. Donahue, T. Darrell and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” 22 10 2014. [Online]. Available: https://arxiv.org/abs/1311.2524. [Accessed 13 4 2018].

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu and A. C. Berg, “SSD: Single Shot MultiBox Detector,” 29 12 2016. [Online]. Available: https://arxiv.org/abs/1512.02325. [Accessed 15 1 2018].

[12] J. Redmon, “YOLO: Real-Time Object Detection,” [Online]. Available: https://pjreddie.com/darknet/yolo/. [Accessed 1 5 2018].

[13] K. He, G. Gkioxari, P. Dollár and R. Girshick, “Mask R-CNN,” 24 1 2018. [Online]. Available: https://arxiv.org/abs/1703.06870. [Accessed 8 5 2018].

[14] D. Lowe, “Object recognition from local scale-invariant features,” in *Proceedings of the Seventh IEEE International Conference on Computer Vision,* Kerkyra, Greece, 1999.

[15] N. Dalal og B. Triggs, «Histograms of oriented gradients for human detection,» i *CVPR'05,* San Diego, CA, USA, 2005.

[16] P. F. Felzenszwalb, R. B. Girshick, D. McAllester and D. Ramanan, “Object Detection with Discriminatively Trained Part-Based Models,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume: 32, Issue: 9),* pp. 1627-1645, 9 2010.

[17] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE (Volume: 86, Issue: 11),* pp. 2278-2324, Nov 1998.

[18] S. Sabour, N. Frosst and G. E. Hinton, “Dynamic Routing Between Capsules,” 7 11 2017. [Online]. Available: https://arxiv.org/abs/1710.09829. [Accessed 10 2 2017].

[19] J. Petterson and A. Gibson, Deep learning: A practitioner's approach, O'Reilly Media, Inc, 2017.

[20] Wikipedia, “Hyperparameter (machine learning),” 23 4 2018. [Online]. Available: https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning). [Accessed 14 5 2018].

[21] S. Sharma, “Activation Functions: Neural Networks,” 6 9 2017. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed 15 5 2018].

[22] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[23] F. Chollet, “How convolutional neural networks see the world: An exploration of convnet filters with Keras,” The Keras Blog, 30 1 2016. [Online]. Available: https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html. [Accessed 8 2 2018].

[24] K. Simonyan, A. Vedaldi and A. Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps,” 19 4 2014. [Online]. Available: https://arxiv.org/abs/1312.6034. [Accessed 10 2 2018].

[25] L. Taylor and G. Nitschke, “Improving Deep Learning using Generic Data Augmentation,” 20 8 2018. [Online]. Available: https://arxiv.org/abs/1708.06020. [Accessed 16 4 2018].

[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research,* pp. 1929-1958, 15 6 2014.

[27] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” 10 4 2015. [Online]. Available: https://arxiv.org/abs/1409.1556. [Accessed 27 2 2018].

[28] M. Lin, Q. Chen and S. Yan, “Network In Network,” 4 3 2014. [Online]. Available: https://arxiv.org/abs/1312.4400. [Accessed 15 5 2018].

[29] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” 2 3 2015. [Online]. Available: https://arxiv.org/abs/1502.03167. [Accessed 14 5 2018].

[30] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama and K. Murphy, “Speed/accuracy trade-offs for modern convolutional object detectors,” 25 4 2017. [Online]. Available: https://arxiv.org/abs/1611.10012. [Accessed 16 1 2018].

[31] D. Parthasarathy, “A Brief Histroy of CNNs in Image Segmentation: From R-CNN to Mask R-CNN,” 22 4 2017. [Online]. Available: https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn-to-mask-r-cnn-34ea83205de4. [Accessed 27 4 2018].

[32] R. Girshick, “Fast R-CNN,” 27 9 2015. [Online]. Available: https://arxiv.org/abs/1504.08083. [Accessed 28 4 2018].

[33] S. Ren, K. He, R. Girshick and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” 6 1 2016. [Online]. Available: https://arxiv.org/abs/1506.01497. [Accessed 28 4 2018].

[34] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” 9 5 2016. [Online]. Available: https://arxiv.org/abs/1506.02640. [Accessed 12 1 2018].

[35] T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, “Focal Loss for Dense Object Detection,” 7 2 2018. [Online]. Available: https://arxiv.org/abs/1708.02002. [Accessed 1 5 2018].

[36] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” 25 12 2016. [Online]. Available: https://arxiv.org/abs/1612.08242. [Accessed 12 1 2018].

[37] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement,” 8 4 2018. [Online]. Available: https://arxiv.org/abs/1804.02767. [Accessed 27 4 2018].

[38] S. Murray, “Real-Time Multiple Object Tracking - A Study on the Importance of Speed,” 2 10 2017. [Online]. Available: https://arxiv.org/abs/1709.03572. [Accessed 7 3 2018].

[39] M. A. Naiel, M. O. Ahmad, M. Swamy, J. Lim and M.-H. Yang, “Online multi-object tracking via robust collaborative model and sample selection,” 8 2016. [Online]. Available: https://www.youtube.com/watch?v=lnAUnU596UE. [Accessed 8 5 2018].

[40] MOT Challenge, “PETS09-S2L1,” MOTChallenge.net, 6 3 2009. [Online]. Available: https://motchallenge.net/vis/PETS09-S2L1/gt/. [Accessed 8 5 2018].

[41] E. Bochinski, V. Eiselein and T. Sikora, “High-Speed Tracking-by-Detection Without Using Image Information,” 23 10 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8078516/. [Accessed 16 1 2018].

[42] MOT Challenge, “MOT17 Results,” MOTChallenge.net, N/A. [Online]. Available: https://motchallenge.net/results/MOT17/. [Accessed 2 5 2018].

[43] digitalbrain79, “Darknet with NNPACK: README.md,” 9 11 2017. [Online]. Available: https://github.com/digitalbrain79/darknet-nnpack. [Accessed 7 4 2018].

[44] M. Huh, P. Agrawal and A. A. Efros, “What makes ImageNet good for transfer learning?,” 30 8 2016. [Online]. Available: https://arxiv.org/abs/1608.08614. [Accessed 24 3 2018].

[45] djmv, “MobilNet_SSD_opencv,” 9 5 2018. [Online]. Available: https://github.com/djmv/MobilNet_SSD_opencv/blob/master/sample_img.py. [Accessed 11 5 2018].

[46] Q. Huang, K. Zhou, S. You and U. Neumann, “Learning to Prune Filters in Convolutional Neural Networks,” 23 1 2018. [Online]. Available: https://arxiv.org/abs/1801.07365. [Accessed 19 2 2018].

[47] TensorFlow, “Fixed Point Quantization,” www.tensorflow.com, 29 3 2018. [Online]. Available: https://www.tensorflow.org/performance/quantization. [Accessed 14 5 2018].

Appendices

HSN University College of Southeast Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Real Time Image Processing with Raspberry Pi.

HSN supervisor: Ola Marius Lysaker (with co-supervisor Joachim Lundberg)

Task background:

Raspberry Pi is a computer built on a single circuit board, roughly at the same size as a credit card. A Raspberry Pi is a low cost computer (total cost is around 400 NKR) and it operates at 1.2 GHz, with 1 GB of RAM. Since its introduction in 2012, Raspberry Pi has become increasingly popular in home automation, in industrial automation, and to promote the teaching of basic computer science in schools. The objective of this master thesis is to develop a framework for automatic detection and tracking of objects based on real time images and information processing. Object detection and tracking are key features in many applications, and are important in various computer vision tasks such as surveillance cameras, autonomous cars, cameras mounted to a flying drone, among others.

Task description:

1. Make a literature review on relevant topics (e.g. real time image processing).
2. Process simple test images using Raspberry Pi.
3. Process images from a web camera (or other sensors), and detect and track objects in real time using Raspberry Pi.
4. Implement a user-friendly visualization/representation of the detection and tracking in the images.

A background in programming (python/matlab) is preferred. Basic knowledge in image processing, pattern recognition and real time computing is also valued, but not needed.

Student category: IIA students

Practical arrangements:

None

Signatures:

Student (date and signature): 29/1 - 2018 [illegible]

Supervisor (date and signature): 29/1 - 2018 Marius Lysaker

Address: Kjølnes ring 56, NO-3918 Porsgrunn, Norway. Phone: 35 57 50 00. Fax: 35 55 75 47.

Appendix B – Code for the Implemented Framework

- Main.py
- NeuralNetwork.py
- ObjectDetector.py
- Tracker.py
- TrainNeuralNetwork.py

```python
import json
import cv2
import time
import numpy as np

from ObjectDetector import ObjectDetector
from ImagePreprocessing import resize, resize_absolute
from NeuralNetwork import NeuralNetwork

def detect_using_image(n_iterations):
    # Load settings
    with open("settings.json", "r") as json_settings:
        settings = json.load(json_settings)
    test_image = cv2.imread(settings["image"]["test_image_path"])
    # create detector
    detector = ObjectDetector(settings, log_length = log_length)
    log_index = 0

    for i in range(n_iterations):
        start_time = time.time()
        print("Analyzing image")
        # Perform inference and show image
        output_image = detector.analyze_image(test_image)
        output_image = resize(output_image, 0.5)
        cv2.imshow("Output image", output_image)
        cv2.waitKey(1)
        log_total[log_index] = time.time() - start_time
        log_index = increment_log_index(log_index)
        print("Complte process took: {} s".format(time.time() - start_time))
    print_calculation_times(detector)
    detector.present_tracking_history()
    cv2.waitKey(0)

def detect_using_webcam():
    # Load settings
    with open("settings.json", "r") as json_settings:
        settings = json.load(json_settings)
    # Take background image
    time.sleep(2)
    cap = cv2.VideoCapture(0)
    ret, background = cap.read()
    # create detector
    detector = ObjectDetector(settings, background, log_length = log_length)

    while (True):
        # Read camera
        ret, frame = cap.read()
        if not ret:
            break
        # Perform inference and show image
        output = detector.analyze_image(frame)
        output = resize(output, 1)
        cv2.imshow("Output", output)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    cap.release()
    detector.present_tracking_history()

def detect_using_video(bg_from_vid):
    # Load settings
    with open("settings.json", "r") as json_settings:
        settings = json.load(json_settings)
    # Load background
    background = cv2.imread(settings["image"]["background_path"])
    # Load video
    cap = cv2.VideoCapture(settings["image"]["test_video_path"])
    # Create detector
    detector = ObjectDetector(settings, background, log_length = log_length)
    frame_number = 0
    while(cap.isOpened()):
        # Read vid
        ret, frame = cap.read()
        if not ret:
            break
        if frame_number == 0 and bg_from_vid:
            # Take alternative background iamge
            cv2.imwrite("bg_vid.jpg", frame)
            detector = ObjectDetector(settings, frame)
        # Perform inference
        output = detector.analyze_image(frame)
        output = resize(output, 0.5)
        cv2.imshow("Output", output)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
        frame_number += 1
        print(frame_number)
    cap.release()
    detector.present_tracking_history()

def print_calculation_times(detector):
    print("Average calculataion times over {} frames:".format(log_length))
    print("Region proposals: {}".format(np.mean(detector.find_regions_log)))
    print("Classifications: {}".format(np.mean(detector.classify_log)))
    print("Tracking: {}".format(np.mean(detector.track_log)))
    print("Annotating: {}".format(np.mean(detector.annotate_log)))
    print("Total: {}".format(np.mean(log_total)))

def increment_log_index(log_index):
    if log_index == log_length - 1:
        log_index = 0
    else:
        log_index += 1
    return log_index

if __name__ == "__main__":
    log_length = 1
    log_total = np.zeros((log_length))
    detect_using_image(1)
    detect_using_webcam()
    detect_using_video(True)
```

```python
import numpy as np
from keras.applications.mobilenet import MobileNet, preprocess_input, relu6, DepthwiseConv2D
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model, load_model
from keras.layers import Dropout, Flatten, Dense, BatchNormalization
from keras.models import model_from_json
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint

class NeuralNetwork:

    def __init__(self, architecture=None, model_path=None):
        self.model = None
        if model_path is not None:
            self.load_model(architecture, model_path)

    def load_model(self, architecture, model_path):
        """ Loads neural network model and weights from file

        Args:
            model_path (str): path to model .json file
            weights_path (str): path to weights .h5 file
        """
        self.model = load_model(model_path, custom_objects = {'relu6':relu6, 'DepthwiseConv2D': DepthwiseConv2D})

    def declare_model(self, n_classes):
        """ Declares model using static declaration. Used when no model file exists.

        Args:
            n_classes (int): Number of classes
        """
        # Import MobileNet feature extractor without fully connected layer
        base_model = MobileNet(input_shape = (128, 128, 3), alpha = 0.25, depth_multiplier = 1, include_top = False, weights = "imagenet")
        # Generate new fully connected layer
        x = Flatten()(base_model.output)
        x = Dense(128, activation='relu')(x)
        x = Dropout(0.5)(x)
        x = BatchNormalization()(x)
        x = Dense(32, activation='relu')(x)
        x = Dropout(0.5)(x)
        x = BatchNormalization()(x)
        predictions = Dense(n_classes, activation='softmax')(x)
        self.model = Model(input=base_model.input, output=predictions)
        # Compile model
        self.model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['categorical_accuracy'])
        print(self.model.summary())

    def train_model(self, x_train, y_train, x_val, y_val, n_epochs = 15, batch_size = 32):
        """ Trains neural network model on data

        Args:
            x_train (4d tensor): training input to model
            y_train (2d tensor): known output for training data
            x_val (4d tensor): validation input to model
            y_val (2d tensor): known output for validation data
        """
        # Preprocess input for mobilenet (input size already correct)
        x_train = preprocess_input(x_train.astype(np.float32))
        x_val = preprocess_input(x_val.astype(np.float32))
        # Set random seed for reproducability
        seed = 5
        # Declare generator for image augmentation
        data_gen = ImageDataGenerator(horizontal_flip=True,
                                      width_shift_range=0.1,
                                      height_shift_range=0.1)
                                      #shear_range = 0.1,
                                      #zoom_range = [0.3, 0])
        data_gen.fit(x_train, augment=True, seed=seed)
        image_gen = data_gen.flow(x_train, y_train, batch_size = batch_size)
        # Recompile model with optimizer learning rate decay
        learning_rate = 0.001
        decay = learning_rate / n_epochs
        optimizer = Adam(lr = learning_rate, decay = decay)
        self.model.compile(optimizer=optimizer,
          loss='categorical_crossentropy', metrics=['categorical_accuracy'])
        # Create checkpoint to save best model
        checkpoint = ModelCheckpoint("train_checkpoint.h5",
          monitor='val_categorical_accuracy', verbose=1, save_best_only=True,
          mode='max')
        # Train model
        self.model.fit_generator(image_gen,
                        steps_per_epoch = int(x_train.shape[0] / batch_size),
                        epochs=n_epochs,
                        verbose=1,
                        validation_data=(x_val, y_val),
                        workers=3,
                        max_queue_size=20,
                        callbacks=[checkpoint])

    def predict(self, images):
        """ Predicts classification of supplied images

        Args:
            images (4d tensor): Images to classify [image, y, x, d]
        Returns:
            2d tensor of class predictions
        """
        model_input = preprocess_input(images.astype(np.float32))
        model_output = self.model.predict(model_input)
        return model_output

    def save_model(self, model_path, weights_path):
        """ Saves neural network model and weights to file

        Args:
            model_path (str): path to model .json file
            weights_path (str): path to weights .h5 file
        """
        with open(model_path, "w+") as json_file:
            json_file.write(self.model.to_json())
        self.model.save_weights(weights_path)
        print("Model saved")

```

```python
import cv2
import numpy as np
import json
import time
from skimage import measure
from keras.models import model_from_json
from keras.applications.mobilenet import relu6, DepthwiseConv2D, MobileNet
#from keras.models import load_model
import ImagePreprocessing
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from NeuralNetwork import NeuralNetwork
from Tracker import Tracker


class ObjectDetector:
    def __init__(self, settings, background = None, log_length = 1):
        """ Class constructor

        Args:
            settings (dict): Dictionary containing settings
            background (array): Background image, image loaded from file if None
            log_length (int): Length of timings logs
        """
        # Store settings in object
        self.settings = settings
        self.image = np.zeros((settings["image"]["height"], settings["image"]["width"]))
        # Load background image from file, if not provided
        if background is None:
            background = cv2.imread(settings["image"]["background_path"])
        self.background_fullsized = background
        self.background_image = ImagePreprocessing.resize(background, settings["preprocessing"]["downscale_factor"])
        self.background_image = ImagePreprocessing.blur(self.background_image, settings["preprocessing"]["blur_kernel_size"])
        #self.background_image = cv2.cvtColor(self.background_image, cv2.COLOR_BGR2YCrCb)
        # Load neural network
        self.neural_network = NeuralNetwork(settings["neural_network"]["architecture"],
                                            settings["neural_network"]["model_path"])
        # Load class labels
        with open(settings["neural_network"]["labels_path"], "r") as json_labels:
            self.labels = json.load(json_labels)
        # Hard-coded marker types, to fit number of classes
        self.class_markers = ["o", "^", "s", "x", "d"]
        # Initialize tracker
        # Hard-coded list of colours.
        color_list_dummy = [(0,0,255),
                            (255,0,0),
                            (0,127,0),
                            (0,191,191),
                            (191,0,191),
                            (191,191,0),
                            (50,255,0),
                            (255,50,0),
                            (50,0,255),
                            (255,0,50),
                            (0,50,255)]
        self.tracker = Tracker(color_list_dummy, settings["tracker"]["same_class_reward"])
        # Initialize arrays used to log calculation times
        self.log_length = log_length
        self.log_index = 0
        self.find_regions_log = np.zeros((log_length))
        self.classify_log = np.zeros((log_length))
        self.track_log = np.zeros((log_length))
        self.annotate_log = np.zeros((log_length))

    def analyze_image(self, image):
        """ Detects and classifies objects in image
        Args:
            image (array): image to be analyzed
        Returns:
            Image annotated with bounding boxes and class confidences
        """
        # Find region proposals
        start_time = time.time()
        bounding_boxes = self.find_region_proposals(image)
        self.find_regions_log[self.log_index] = time.time() - start_time
        print("Finding bounding boxes took: {} s".format(self.find_regions_log[self.log_index]))
        start_time = time.time()
        # Classify content in regions
        start_time = time.time()
        classifications = self.classify_region_content(image, bounding_boxes)
        self.classify_log[self.log_index] = time.time() - start_time
        print("Classifying {} crops took: {} s".format(bounding_boxes.shape[0], self.classify_log[self.log_index]))
        # Track objects
        start_time = time.time()
        tracks = self.tracker.track(bounding_boxes, classifications, self.settings["tracker"]["iou_threshold"])
        self.track_log[self.log_index] = time.time() - start_time
        print("Tracking objects took took: {} s".format(self.track_log[self.log_index]))
        # Draw bounding boxes
        start_time = time.time()
        annotated_image = self.draw_bounding_boxes(image, tracks)
        self.annotate_log[self.log_index] = time.time() - start_time
        print("Annotating images took: {} s".format(self.annotate_log[self.log_index]))
        # Increment log index
        self.increment_log_index()
        return annotated_image

    def classify_region_content(self, image, bounding_boxes):
        """ Classifies content in bounding box regions using neural network model

        Args:
            image (array): Image to perform detection on
            bounding_boxes (array): array containing upper left and lower right corner coordinates of bounding box rectangle (y1, x1, z, y2, x2, d) in second dim
        Returns:
            2d array of class probabilities
        """
        # Declare array for holding input data
        network_input = np.zeros((len(bounding_boxes), self.settings["neural_network"]["input_size"], self.settings["neural_network"]["input_size"], 3))
        crop_index = 0
        # Loop through bounding boxes, crop image, scale, and add to input array
        start_time = time.time()
        for bounding_box in bounding_boxes:
            x1 = np.maximum(0, bounding_box[1] - self.settings["bounding_box"]["padding"])
            x2 = np.minimum(1919, bounding_box[3] + self.settings["bounding_box"]["padding"])
            y1 = np.maximum(0, bounding_box[0] - self.settings["bounding_box"]["padding"])
            y2 = np.minimum(1079, bounding_box[2] + self.settings["bounding_box"]["padding"])
            crop = image[y1:y2,x1:x2]
            network_input[crop_index,:,:,:] = ImagePreprocessing.resize_absolute(crop, self.settings["neural_network"]["input_size"], self.settings["neural_network"]["input_size"])
            #cv2.imshow("debug", network_input[crop_index,:,:,:,].astype("int"))
            #cv2.waitKey(0)
            crop_index += 1
        print("Croping images took {} s".format(start_time - time.time()))
        # Classify content
        start_time = time.time()
        classifications = self.neural_network.predict(network_input)
        print("classifying images took {} s".format(start_time - time.time()))
        return classifications

    def draw_bounding_boxes(self, image, tracks):
        """ Draws bounding boxes on image

        Args:
            image (array): Image to annotate
            tracks (list): list of currently tracked objects
        Returns:
            Annotated image
        """
        # Copy image
        annotated_image = np.array(image)#image.copy()
        # Loop through bounding  boxes, drawing each on the copied image, add class text
        for i in range(len(tracks)):
            # Create text annotation
            class_index = np.argmax(tracks[i].classification_track[-1][:])
            annotation = "{0}: {1:.2f}".format(self.labels[str(class_index)],
                                               tracks[i].classification_track[-1][class_index])
            # Draw bounding box
            cv2.rectangle(annotated_image,
                          (tracks[i].bounding_box_track[-1][1],
                            tracks[i].bounding_box_track[-1][0]),
                          (tracks[i].bounding_box_track[-1][3],
                            tracks[i].bounding_box_track[-1][2]),
                          color = tracks[i].color,
                          thickness = self.settings["bounding_box"]["box_thickness"])
            # Write annotation
            cv2.putText(annotated_image,
                        annotation,
                        (tracks[i].bounding_box_track[-1][1],
                            tracks[i].bounding_box_track[-1][2]),
                        cv2.FONT_HERSHEY_SIMPLEX,
                        fontScale = self.settings["bounding_box"]["font_scale"],
                        color = tracks[i].color,
                        thickness = self.settings["bounding_box"]["font_thickness"])
        return annotated_image

    def find_region_proposals(self, image):
        """ Finds region proposals using background subtraction

        Preprocesses image, through color space change, resizing, background subtraction, bluring, and thresholding, to generate region proposals

        Args:
            image (array): Image (differential, threshold) to find region proposals from
        Returns:
            Array of bounding boxes/region proposals (y1, x1, z, y2, x2, d)
        """
        # Preprocess image
        resized_image = ImagePreprocessing.resize(image, self.settings["preprocessing"]["downscale_factor"])
        blured_image = cv2.blur(resized_image, tuple(self.settings["preprocessing"]["blur_kernel_size"]))
        diff_image = ImagePreprocessing.remove_background(blured_image, self.background_image)
        threshold_image = ImagePreprocessing.threshold(diff_image, self.settings["preprocessing"]["threshold"] * 255)
        ## Show image for debug
        #cv2.imshow("debug", dilated_image)
        #cv2.waitKey(1)
        # Get region proposals
        labels = measure.label(threshold_image)
        regionprops = measure.regionprops(labels)
        bounding_boxes = [prop.bbox for prop in regionprops] # (y1, x1, z, y2, x2, d)
        bounding_boxes = np.array(bounding_boxes) * (1 / self.settings["preprocessing"]["downscale_factor"])
        bounding_boxes = self.filter_bounding_boxes(bounding_boxes)
        return bounding_boxes.astype("int")

    def filter_bounding_boxes(self, bounding_boxes):
        """ Filters bounding boxes based on minimum box size and maximum intersect/union (IoU)

        Args:
            bounding_boxes (array): array containing upper left and lower right corner coordinates of bounding box rectangle
        """
        if len(bounding_boxes) == 0:
            return bounding_boxes
        # Filter out bounding boxes with size smaller than minimum size in settings
        height_filter = (bounding_boxes[:, 2] - bounding_boxes[:,0] > self.settings["bounding_box"]["min_size"])
        width_filter = (bounding_boxes[:, 3] - bounding_boxes[:,1] > self.settings["bounding_box"]["min_size"])
        box_filter = np.bitwise_and(height_filter, width_filter)
        return bounding_boxes[box_filter,:]

    def increment_log_index(self):
        """ Increments the log index
        """
        if self.log_index == self.log_length - 1:
            self.log_index = 0
        else:
            self.log_index += 1

    def present_tracking_history(self):
        """ Creates a plot for all tracked objects history
        """
        # Set background image as plot background for context
        plt.imshow(cv2.cvtColor(self.background_fullsized, cv2.COLOR_BGR2RGB))
        # Create list of markers for plot legend
        marker_list = [mlines.Line2D([], [], color="black", marker=m) for m in self.class_markers]
        # Create line and scatter plots for each active track
        for track in self.tracker.tracks_active:
            self.plot_track(track)
        # Create line and scatter plots for each active track
        for track in self.tracker.tracks_finished:
            self.plot_track(track)
        # Add legend and show plot
        plt.legend(marker_list, list(self.labels.values()), bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0)
        plt.show()

    def plot_track(self, track):
        """ Plots line and scatter plot of a specific tracked objects history

        Args:
            track (track object): Object containing a tracked objects history
        """
        # Set color for track
        color = tuple([v/255 for v in reversed(track.color)])
        # Convert list of box coordinates to an array, and calculate centre point for each position
        box_coords = np.array(track.bounding_box_track)
        x_center = (box_coords[:,3] + box_coords[:,1])/2
        y_center = (box_coords[:,2] + box_coords[:,0])/2
        # Find class with highest confidence in each frame
        classifications = np.array(track.classification_track).argmax(axis=1)
        classes = np.unique(classifications)
        # Create a scatterplot for each classification type the object was classified as, in order to get correct markers
        for c in classes:
            index_mask = classifications == c
            plt.scatter(x_center[index_mask], y_center[index_mask], marker=self.class_markers[c], color=color, s=60)
        # Plot line to show positions
        plt.plot(x_center, y_center, color=color)
```

```python
import numpy as np
from scipy.optimize import linear_sum_assignment
from itertools import cycle

class Tracker:

    def __init__(self, color_list, same_class_reward):
        self.colors = cycle(color_list)
        self.tracks_active = []
        self.tracks_finished = []
        self.frame_index = 0
        self.incremental_id = 0
        # Variables holding data from previous frame,
        # to not have to calculate again for next frame
        self.previous_boxes = np.empty((0,0))
        self.previous_area = np.empty((0))
        self.previous_classifications = np.empty((0,0))
        self.box_to_track_map = []
        self.same_class_reward = same_class_reward

    def track(self, boxes, classifications, threshold):
        """ Matches bounding boxes in previous and current frame, using Hungarian Algorithm

        Args:
            boxes (array): new bounding boxes
            classifications (array): classifications of content in provided bounding boxes
            threshold (scalar): threshold used to determine whether detections are instances of the same object.
        """
        # compute the area of the bounding boxes
        if len(boxes) > 0:
            area = (boxes[:,3] - boxes[:,1]) * (boxes[:,2] - boxes[:,0])
        else:
            area = np.zeros((0))

        # Find number of previous and current boxes
        n_new = boxes.shape[0]
        n_prev = self.previous_boxes.shape[0]

        # If boxes exist
        if n_new > 0 and n_prev > 0:
            # Find all maximum coordinate combinations between new and previous boxes
            yy1 = self.get_max_coordinate_block(boxes[:,0], self.previous_boxes[self.box_to_track_map,0], n_new, n_prev)
            xx1 = self.get_max_coordinate_block(boxes[:,1], self.previous_boxes[self.box_to_track_map,1], n_new, n_prev)
            yy2 = self.get_min_coordinate_block(boxes[:,2], self.previous_boxes[self.box_to_track_map,2], n_new, n_prev)
            xx2 = self.get_min_coordinate_block(boxes[:,3], self.previous_boxes[self.box_to_track_map,3], n_new, n_prev)
            # Calculate intersect between new and previoius boxes
            w = np.maximum(np.zeros((n_new,n_prev)), xx2-xx1)
            h = np.maximum(np.zeros((n_new,n_prev)), yy2-yy1)
            intersect = np.multiply(w,h)
            # Create 2d arrays from area and previous area
            area_block = np.repeat(area[:,None],n_prev,1)
            prev_area_block = np.repeat(self.previous_area[None,self.box_to_track_map],n_new,0)
            # Calculate intersect over union between new and previous boxes
            iou = intersect / (area_block + prev_area_block - intersect + 1)
            # Find cases where classification of previous and new boxes match
            class_match_block = self.get_class_match_block(classifications, self.previous_classifications[self.box_to_track_map,:], n_new, n_prev)
            # Create cost matrix
            cost_matrix = iou + class_match_block*self.same_class_reward
            # Solution using hungarian algorithm, maximize cost
            col_ind, row_ind = linear_sum_assignment(-cost_matrix.T)
            # Filter out instances where value is below threshold
            ind_mask = cost_matrix[row_ind,col_ind] >= threshold
            row_ind = row_ind[ind_mask]
            col_ind = col_ind[ind_mask]
        else:
            row_ind = []
            col_ind = []

        # Generate list of indices of new boxes not assigned to tracks
        not_assigned_row = [i for i in range(n_new) if i not in row_ind]
        not_assigned_col = [i for i in range(n_prev) if i not in col_ind]

        # Add box matches to current tracks
        for i in range(len(row_ind)):
            self.tracks_active[i].add_frame_data(self.frame_index,
                                                    boxes[row_ind[i],:],
                                                    classifications[row_ind[i],:])
        # Move non-matched tracks from active to finished, then delete them
        for i in range(len(not_assigned_col)-1, -1, -1):
            self.tracks_finished.append(self.tracks_active[not_assigned_col[i]])
            del self.tracks_active[not_assigned_col[i]]

        # Add non-matched boxes to active tracks
        for i in range(len(not_assigned_row)):
            self.tracks_active.append(Track(self.incremental_id,
                                            self.frame_index,
                                            boxes[not_assigned_row[i],:],
                                            classifications[not_assigned_row[i],:],
                                            next(self.colors)))
            # Increment next id
            self.incremental_id += 1

        # Update box_to_track_map
        self.box_to_track_map = list(row_ind) + not_assigned_row
        # Move boxes to previous boxes, area to previous area, and increment frame index
        self.previous_area = area
        self.previous_boxes = boxes
        self.previous_classifications = classifications
        self.frame_index += 1
        return self.tracks_active

    def get_max_coordinate_block(self, new, prev, n_new, n_prev):
        """ Creates a 2d block containing the max coordinates between new and previous bounding boxes

                    A,       B
            a  max(a,A) max(a,B)
            b  max(b,A) max(b,B)

        Args:
            new (array): New bounding boxes
            prev (array): Previous bounding boxes
            n_new (int): Number of new boxes
            n_prev (int): Number of previous boxes
        Returns:
            2d array containing max coordinates between combinations of new and previous boxes
        """
        new_block = np.repeat(new[:,None],n_prev,1)
        prev_block = np.repeat(prev[None,:],n_new,0)
        max_coords = np.maximum(new_block, prev_block)
        return max_coords

    def get_min_coordinate_block(self, new, prev, n_new, n_prev):
        """ Creates a 2d block containing the max coordinates between new and previous bounding boxes

                     A,       B
            a  min(a,A) min(a,B)
            b  min(b,A) min(b,B)

        Args:
            new (array): New bounding boxes
            prev (array): Previous bounding boxes
            n_new (int): Number of new boxes
            n_prev (int): Number of previous boxes
        Returns:
            2d array containing min coordinates between combinations of new and previous boxes
        """
        new_block = np.repeat(new[:,None],n_prev,1)
        prev_block = np.repeat(prev[None,:],n_new,0)
        max_coords = np.minimum(new_block, prev_block)
        return max_coords

    def get_class_match_block(self, new, prev, n_new, n_prev):
        """ Creates a 2d block containing information of whether combinantions of bounding boxes had the same class

        Args:
            new (array): New bounding boxes
            prev (array): Previous bounding boxes
            n_new (int): Number of new boxes
            n_prev (int): Number of previous boxes
        Returns:
            2d array containing information of whether combinantions of
              bounding boxes had the same class
        """
        new_block = np.repeat(new.argmax(axis=1)[:,None],n_prev,1)
        prev_block = np.repeat(prev.argmax(axis=1)[None,:],n_new,0)
        match_block = new_block == prev_block
        return match_block


class Track:
    """ Class containing tracking information
    """
    def __init__(self, identifier, frame_index, bounding_box, classification,
      color):
        self.frame_track = [frame_index]
        self.bounding_box_track = [bounding_box]
        self.classification_track = [classification]
        self.color = color
        self.identifier = identifier

    def add_frame_data(self, frame_index, bounding_box, classification):
        """ Adds data for current frame to an objects track

        Args:
            frame_index (int): Index of current frame
            bounding_box (1d array): Bounding box in current frame
            classification (1d array): Classification scores in current frame
        """
        self.frame_track.append(frame_index)
        self.bounding_box_track.append(bounding_box)
        self.classification_track.append(classification)


```

```python
import cv2
import numpy as np
import os
from keras.utils import to_categorical

from NeuralNetwork import NeuralNetwork
from ImagePreprocessing import resize_absolute

def load_data(data_path, val_portion, network_input_size):
    """ Loads image data from folders (requires correct structure)

   |-Data (top level directory)
      |-Class 1 folder
      .   |-image 1
      .   |-image 2
      .   |-image n
      |-Class 2 folder
      .   |-image 1
      .   |-iamge 2
      .   |-image n
      |-...
      |-Class n folder

    Args:
        data_path (str): Path top level data directory
        val_portion (float): Portion of loaded data to use as validation (remainder used as training)
    Returns:
        Lists containing training data, validation data, and number of classes
    """
    print("Loading data...")
    x_train = []
    y_train = []
    x_val = []
    y_val = []
    class_index = 0
    # Traverse directory
    for dir_name, sub_dirs, files in os.walk(data_path):
        x = []
        y = []
        if len(files) > 0:
            # Read files
            for file in files:
                image = cv2.imread(dir_name + "\\" + file)
                resized_image = resize_absolute(image, network_input_size, network_input_size)
                x.append(resized_image)
                y.append(class_index)

            # Calculate number of files in class to use for validation
            n_val = int(len(files)*val_portion)
            val_indices = np.random.choice(range(len(files)), n_val, replace = False)
            # Add validation data to validation lists, add training data to trainingn lists
            x_train.extend(np.delete(x, val_indices, axis=0))
            y_train.extend(np.delete(y, val_indices))
            x_val.extend([x[i] for i in val_indices])
            y_val.extend([y[i] for i in val_indices])
            # Increment class index
            class_index += 1
    n_classes = class_index
    x = None
    y = None
    x_train = np.array(x_train)
    x_val = np.array(x_val)
    print("Data loaded!")
    return x_train, y_train, x_val, y_val, n_classes

def reshape_image_list(x, network_input_size):
    """ Reshapes list of images to 4d array, with correct shape (dimensions) for use with neural network

    Args:
        x (list): List of images
        network_input_size (int): input height and width expected by neural network
    Returns
        Reshaped array, ready for use in training
    """
    # Declare array to hold reshaped data
    x_reshaped = np.zeros((len(x), network_input_size, network_input_size, 3))
    # Resize images
    for i in range(len(x)):
        x_reshaped[i,:,:,:] = resize_absolute(x[i], network_input_size, network_input_size)
    return x_reshaped

if __name__ == '__main__':
    # Settings
    np.random.seed(5) # For reproducability
    val_portion = 0.25
    data_path = 'E:\\Download\\Datasets\\Master\\'
    model_path = 'E:models\\Mobilenet_128_model.json'
    weights_path = 'E:models\\Mobilenet_128_weights.h5'
    n_epochs = 50
    batch_size = 128
    network_input_size = 128
    # Load and reshape data
    x_train, y_train, x_val, y_val, n_classes = load_data(data_path, val_portion, network_input_size)
    #x_train = reshape_image_list(x_train, network_input_size)
    y_train = to_categorical(y_train)
    #x_val = reshape_image_list(x_val, network_input_size)
    y_val = to_categorical(y_val)
    # Create neural network
    neural_network = NeuralNetwork()
    neural_network.declare_model(n_classes)
    # Train neural network
    neural_network.train_model(x_train, y_train, x_val, y_val, n_epochs, batch_size)
    neural_network.save_model(model_path, weights_path)
```