NTNU
Norwegian University of
Science and Technology

Tiny Overseer

A System for Autonomous Low-Altitude
Missions

Bjarte Sjursen

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Frank Lindseth, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

This thesis will demonstrate a complete system based on commercial quadcopter drones with the ability to perform autonomous unmanned missions. The system is based on state of the art object detection networks with an emphasis on SSD. The goal of the system is to perform surveillance and inspection over areas of interest. During missions, the drone is streaming video and other sensor data in real-time to an external server that performs inference using neural networks and predicts the geolocation of detected targets. We attain geolocation predictions from altitudes above 25 meters with errors in the range of 1 - 7 meters. The quadcopter applied during the research is a DJI Mavick Pro.

Sammendrag

I denne oppgaven presenteres et autonomt system basert på kommersielle droner. Systemet som er utviklet kan utføre ubemmanede oppdrag knyttet til inspeksjon og overvåkning gjennom bruk av nevralnettverk med kapabilitet til å utføre objektdeteksjon på bilder. Under utførelse av oppdrag strømmer dronen video og andre sensordata i sanntid til en ekstern server som eksekverer relevant analyse. Analysen som produseres inneholder geolokasjonsapproksimasjoner ved hjelp av en teknikk vi har utviklet. Approksimasjonene bygger på nevnte nevralnett, samt geodetiske teknikker utviklet på 70-tallet. Gjennom våre eksperimenter har vi observert en nøyaktighet på lokasjonsapproksimasjonene som er i størrelsesorden 1 - 7 meter. Våre tester med dronen er utført i stedshøyde mellom 20-40 meter. Dronen vi har benyttet under arbeidet er en DJI Mavick Pro.

Contents

List of Figures

List of Tables

Abbreviations and terms

API	Application Programming Interface
URL	Uniform Resource Locator
Waypoint	A geographical location used as part of a drone mission
NN	Neural network
CNN	Convolutional neural network
GUI	Graphical user interface
Lat	Latitude
Lng	Longitude
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
IoU	Intersection Over Union
mAP	Mean Average Precision
SDK	Software Development Kit

Chapter 1

Introduction

1.1 Motivation and project description

In the Era of Recombination, 240,000 to 300,000 years after the Big Bang, the first light in the Universe emerged. The Universe went from being totally opaque, to transparent. Photons made the universe visible for the first time. Fast-forward 9 billion years and life started to develop on Earth. A myriad of sophisticated organisms started to form and interact with the Universe. A key component in the formation of life, is its interplay with photons. In some cases to produce energy, in other cases to percept the world through vision. The perceptual stimulus given by vision gives the ability to assimilate information from the surrounding environment and understand the world.

In modern times there have been made attempts to emulate the ability of the mammalian visual system by the use of computers. A group of summer workers at the Massachusetts Institute of Technology started a project called the 'summer vision project' in 1966, with the goal of constructing a programmatic visual system with the ability to understand images. This project was one of the first stepping stones in the field of computer vision. The project explicitly stated a goal that is a key challenge in the discipline today: detect and label objects. Computer vision was originally meant to help enable the development of complex robotics with artificial intelligence.

Today there is a tremendous amount of interest in intelligent machines with the ability to autonomously perform tasks in the world as they can provide a cost-effective means of solving challenging problems such as agricultural analysis, power line inspection, and automated transportation. This thesis will look closer at techniques that can be employed in order to make vehicles smart and perform tasks autonomously. There will be an emphasis on techniques for aerial vehicles. The tools utilized in this paper will mainly be centered around commercial quadcopter drones and computer vision techniques with different flavours of convolutional neural networks. Exciting applications based on such tools are starting to appear. A recent paper out of Cambridge University presented a commercial drone system with remote neural network inference used for detection of violent behavior in crowds of people [21].

The thesis will explore the capabilities and limitations of commercial-grade quadcopter drone systems and demonstrate hands-on work. A series of experiments will be carried out to evaluate the performance in simulated and real-world environments.

1.2 Project goals and research questions

The primary goal of this thesis is to find out how one can utilize deep learning in the context of aerial vehicles to perform autonomous real-time missions. We divide the primary goal into two distinct sub-goals as follows.

1. Create a system to handle motoric actions performed by the vehicle.
2. Gather information through provided sensors to form a model of the environment related to the task at hand.

The research questions related to the goals at hand are as follows:

1. **RQ1**: Is it possible to use an affordable, commercial drone for autonomous flight?
2. **RQ2**: What infrastructure is needed to enable real-time autonomy?
3. **RQ3**: How can one use deep learning for inspection and surveillance?
4. **RQ4**: Is it possible to accurately map perceived objects to global coordinates using commercial drones?

1.3 Contributions

The contribution of this thesis is a complete system for executing autonomous drone-based missions. The missions range from single point inspection missions to area cover missions. The system created can approximate the geolocation of detected objects. We have moreover performed a quantitative analysis of remote neural network predictions for image-related tasks from mobile devices on TCP-network connections. Finally, we have created multiple bounding box annotated data-sets of cars from a birds-eye view.

1.4 Thesis outline

The organization of this paper is as follows. The introductory chapter has presented an overview of the questions to be answered as well as a description of the problem at hand. The second part of the paper will give some background on the various technologies that are used throughout the project, and this is going to be both regarding the computer vision technologies used as well as the software controlling the drone. Part three of the paper will present an architecture for the system. Part four will present results from executed experiments. The last two parts will give final thoughts and suggestions for future work.

Chapter 2

Background

The background chapter of the thesis will look at previous work related to the research questions at hand. The first few sections will cover the fundamentals of deep learning in addition to providing insights into how various object detection methods work. Following subsections will focus on the drone, its framework for control and surrounding tools. The final sections of this chapter will present important network programming paradigms in addition to the pinhole camera model.

2.1 Neural networks

In 1959 ADALINE was conceived. ADALINE is a system that was created to analyze binary patterns and predict the following bit. It is typically used to read bitstreams. Its main purpose is to eradicate echoes on phone lines. It holds a special role in the history of AI - it is the first commercially applied neural network [22].

A neural network (abbreviated NN) is a mathematical model with a great deal of inspiration from mammalian brain structures. NNs learn to solve problems by empirically examining data without task-specific programming. As the above story of ADALINE might reveal - neural networks are used for a wide variety of tasks demanding intricate pattern understanding [10].

The essential component of these computational structures is the artificial neuron. They are elemental mathematical objects that take multiple numerical input values and output a numerical output value. The artificial neuron conceptually has one side with several incoming edges in addition to one outgoing edge, all

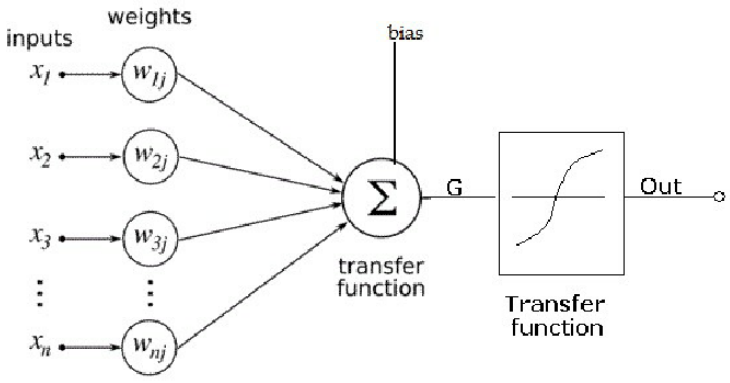


Figure 2.1: An illustration of the operations conducted by an artificial neuron [18].

of them with an associated edge weight. The mathematical operation performed by an artificial neuron is taking the dot product between the input parameters and the associated edge weights as depicted in figure 2.1. It then gives a scalar output value that feeds into an activation function.

A single neuron by itself has limited capabilities. To form neural networks one often group the neurons into layers as seen in figure 2.2. At the leftmost part of the network input values are flowing into the network, and on the very right, output values are flowing out. Figure 2.2 displays the most basic form of a neural network - the fully connected feedforward neural network.

Feedforward neural networks are great for tasks where the input is a vector of data. Certain kinds of input however allow for the exploitation of the intrinsic structures in the data. Images are good examples. Objects can be located at multiple locations in an image, yet they have the exact same appearance. This insight can be utilized to construct a new form of neural network that learns more abstractly what an object looks like. The architectural flavour of neural networks most commonly used for image analysis is called a convolutional neural networks.

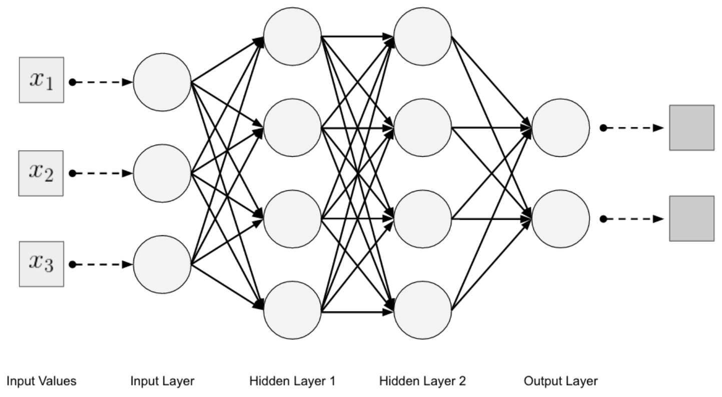


Figure 2.2: A figure illustrating how neurons are grouped into layers to form a neural network [14].

2.2 A convolutional neural network

The convolutional neural networks (CNN) is a kind of neural network which employs the mathematical operation of convolution in at least one of its layers. It is great for processing data with grid-like structures such as images. Successful applications of the CNN include image classification, object detection, and semantic segmentation. CNNs are preferred over fully-connected neural networks because the fully-connected neural network does not scale well when one use images as input, there are too many parameters for the network to learn.

One of the main features of CNNs is the convolution operation. The operation constructs an $n \times n$ kernel of numerical values and slides it over the input data, after which it computes the dot product among the region that the input covers and the values in the kernel. The output of the convolution is often called a feature map, and it is either of the same size or smaller than the input data, it can, however, create a larger number of elements in the third dimension. We depicted an example of how data propagates forward in CNN-based networks in figure 2.3.

There are three main groups of components in a network like this.

1. Input layer.
2. Feature extracting layers.
3. Classification layers.

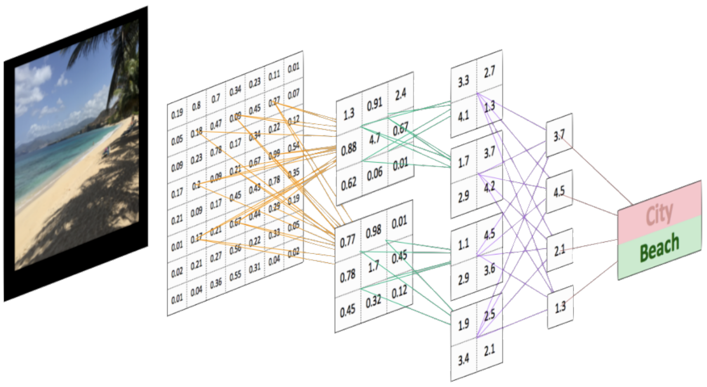


Figure 2.3: An illustration of how a CNN propagates input to output [14].

The major components of the groups above are the convolutional layer, the pooling layer, and the fully-connected layer. The pooling layer is another instance of an $n \times n$ numerical structure, and it is often used to down-sample the data. The pooling operation replaces the output of the net at a certain layer with a summary statistic of the nearby outputs, and typical examples include the average or the maximum. The pooling layer is slid over the output like the convolution layer. The fully-connected layer is generally used at the end of the network to perform classification.

Classic CNNs employ an alternating pattern of pooling and convolution operations, followed by fully-connected layers. The pattern produces a grid-like topology. For a more thorough walkthrough of NNs and CNNs, please see Goodfellow et al. [10].

2.3 Image classification models

The task of image classification revolves around classifying images with a label. If there is a picture of a cat, then the image is classified as a cat and so forth. CNNs are the most popular means of approaching this problem. The lifecycle of a CNN typically consists of two distinct parts.

1. Training.
2. Inference.

One can characterize the training of the networks as a process of optimizing the network loss function with regards to the parameters in the network. The process is complex and consists of partially differentiating the loss function with regard to all the weights in the network to find the best weight adjustment. This process is called backpropagation. One typically train the networks in a supervised offline fashion with labeled data.

In the inference-step one perform class prediction of labels from a given input image. If the model is trained and regularized well, it should perform the task easily. One furthermore commonly splits the labeled data into three sets, a training set, a test set, and an validation set. The validation set is a set of instances used to finetune the parameters of the network. Moreover, the test cases are not shown to network until the very end of the training process and are used to assess it. The most common metric to assess the performance of the network is accuracy. Accuracy indicates the proportion of correctly classified images from the test set. Each year there is competition within the visual computing community called the ImageNet Large Scale Visual Recognition Challenge. Accuracy is one of the most important metrics in this competition.

2.3.1 ImageNet Large Scale Visual Recognition Challenge

The Imagenet Large Scale Visual Recognition Challenge (ILSVRC) challenges participants to create image classification models with the ability to discriminate between 1000 different classes of objects. Since 2012 convolutional neural networks have been the top-performing models in the competition. In the following sections, we will discuss the inner workings of some of the top-performing ones: VGGNet and ResNet.

2.3.2 VGGNet

VGGNet made by Karen Simonyan and Andrew Zisserman is a convolutional neural network architecture from the University of Oxford [20]. In their paper titled 'Very Deep Convolutional Networks for Large-Scale Image Recognition,' they pointed a couple of interesting observations which have become common knowledge in the visual computing community. One of the most exciting observations they made was to point out that deep models tend to perform better than shallow ones, in some cases even with fewer parameters in the network. They were able to prove that a stack of three 3×3 convolution layers in succession have the same receptive field as a single 7×7 convolution filter and that a stack of two 3×3 convolution layers have the same receptive field as a single 5×5 convolution filter. One can create an arbitrarily sized stack of 3×3 convolution layers and receive a gradually bigger receptive field. It is a fascinating insight for a couple of reasons.

1. It adds extra nonlinearities to the network.
2. It reduces the number of parameters to learn.

In ILSVRC 2014 the model won the 2nd price for image classification and 1st place for object localization.

2.3.3 ResNet

At the time when Kaiming He made ResNet[11] he postulated a simple question - what happens when one continues to add layers to a plain convolutional neural network? He investigated this question and acquired some intriguing results. The observations made was that a 56-layer deep CNN had both worse training and testing error over a prolonged period compared to a 20-layer deep CNN. He hypothesized that the 56-layer CNN did not overfit, but rather that it would not converge during training, the training just took too much time. One can construct an argument for why a deep model should be able to perform at least as good as a shallower model. Copy the layers of the shallow model to the initial layers of the deep model, and set the following layers to the identity function. The deeper network will then output the same as the shallow one. Optimizing deeper networks are a lot harder than optimizing shallower ones. He used his findings to construct an architecture that makes training faster and easier by adding skip-layer connections to the network. The skip-layer connections are included in residual blocks consisting of two convolutional layers. The skip-layer connection is depicted in figure 2.4. The effect the skip connections have on the network is that it eases the training process for the network since we avoid the problem of vanishing gradients [5].

When the architecture was tested, it achieved several feats. In 2015 it won the ILSVRC with a 152 layer deep architecture with an impressive 3.5% error, he noted that this was better performance than what humans can do.

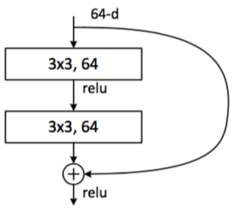


Figure 2.4: The residual block concept [11].

2.4 Object detection networks

The main consideration up until this point in the thesis has been around classical convolutional neural networks for image classification. The task at hand in this thesis does, however, require a higher level of granularity concerning neural network perception and output. To create an autonomous drone with capabilities related to inspection, surveillance and data generation one needs to know where objects of interest are located in images at hand. The task of locating and classifying multiple objects in an image is known as object detection, and there are several well known neural network designs created to do this. Object detection networks are engineered to take images as input and output bounding boxes. In the case of object detection, a bounding box is a rectangular box encapsulating an object of interest in a picture with an assigned label telling what it is plus a confidence score which indicates the certainty of the prediction. There have been a couple of high performing object detection networks in the past couple of years such as R-CNN, fast R-CNN, faster R-CNN, YOLO, and SSD. The following chapters of the paper will outline the inner workings of these architectures.

2.4.1 R-CNN, Fast R-CNN and Faster R-CNN

In 2014 a 3-stage object detection pipeline called R-CNN was proposed in the paper 'rich feature hierarchies for accurate object detection and semantic segmentation' [9]. The three stages of the system have the following areas of responsibility:

1. Propose regions of interest.
2. Compute features for the regions.
3. Classify regions.

A brute-force schema for detecting objects in an image could be to run a convolutional neural network over every possible location in the image at several scales and aspect ratios, and make it give a score for what it sees, essentially just doing image classification at all possible locations and scales in the image. That approach would however be exceedingly time-consuming and demand a lot of computation. This is the cause for why R-CNN has the first part of its pipeline generate regions of interest(ROI), which essentially are places where the system suspect an object is located. An algorithm called selective search is deployed for this task and it generates approximately 2000 regions of interest for a given image.

After generating the initial ROIs, the regions are transformed and given to a CNN that computes a set of features from them. One extract the computed features as a 4096-dimensional vector. The network used for this task consists of five convolution layers and two fully-connected layers at the end.

Succeeding the vector generation is the classification stage where a collection of support vector machines(SVM) are used to predict the label for the vectors. SVMs originates from the field of classical machine learning. There is one binary SVM trained for every class that needs to be classified. After the classification, one applies non-max suppression to remove overlapping bounding boxes.

R-CNN unfortunately suffers from being very slow at test time. It uses close to 50 seconds for every run. Each proposed region needs a full pass through the CNN in addition to the fact that the pipeline is complex and hard to finetune.

Work on R-CNN spawned two additional methods in following years. These methods were creatively named fast R-CNN [8] and faster R-CNN [17]. The main difference between fast R-CNN and classic R-CNN is in the way the features are extracted. Rather than running each region of interest through a CNN by itself, one runs the entire input image through a CNN once. The next step is to run the selective search algorithm and get the features from the desired regions of interest by looking at the output of the initial CNN forward pass. This means that one does not have to compute new features for every region of interest since those are all generated in one go. Following the ROI proposals of the projected feature map is a process called ROI-pooling. ROI pooling takes the variable sized ROIs and transform them all to a fixed size such that they fit the fully-connected layers at the end. This dramatically speeds up the entire pipeline by 25×, resulting in object detection done in 2 seconds. Another change worth mentioning is that the classification part is not done by binary SVMs any longer, a softmax classifier does that job on top of a fully-connected layer. A bounding box regression head and softmax classification head are both appended at the end of the network giving a 'two-headed' CNN. The beauty of this pipeline is that it is much easier to train than the classic vanilla R-CNN.

Succeeding fast R-CNN is faster R-CNN. The difference between fast R-CNN and faster R-CNN is that the region proposal strategy is changed into a part of the neural network. The dominant factor for the time consumption is the region proposal method in Fast R-CNN. This is changed into a region proposal network in the new version. The region proposal network takes the feature map produced by the first stage of the pipeline as input and produces proposed region by using an integrated part of the network instead of external methods such as selective search. By making use of the new method for region proposal, Faster R-CNN offers a 10× speed-up in comparison to fast R-CNN, which implies that faster

R-CNN is 250$\times$ faster than classic R-CNN. It is able to perform object detection at one image in 0.2 seconds. Dropping the selective search algorithm as region proposal strategy drastically improves the speed. In this method, the entire architecture is part of a CNN, making for a much sleeker pipeline. In addition, it is a lot easier to train than the preceding methods.

2.4.2 YOLO, YOLOv2 and YOLOv3

In 2016 Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi published a paper under the title 'You Only Look Once: Unified Real-time Object-detection' which depicted one of the fastest object detectors to date - YOLO [15]. There are two different versions of YOLO. The detector can process images at 155 fps for the smallest version and 45 fps for the larger one.

YOLO divides the input image into an $S \times S$ grid of cells. Each of the cells in the grid is responsible for predicting an arbitrary number of bounding boxes, B, which can be adjusted. Each of the bounding boxes is given by its x- and y-coordinates relative to the bounds of the cell as well as its width, w, and height, h, relative to the box. In addition to these parameters that describe size and location of the bounding box, the cells in the grid also predict a confidence score for how certain the detector is that it contains an object. The confidence scores show how certain the model is that the box encapsulates an object, and also how accurate it believes the predicted box to be. Each grid cell is also responsible for predicting C class scores that are multiplied with the confidence scores from the bounding boxes to generate bounding boxes with attached class predictions. The final output of the network is an $S \times S \times (B \times 5 + C)$ tensor containing all the data mentioned above.

The implementation is realized in the form of a convolutional neural network where the initial layers extract features while the fully connected layers at the end predict the output probabilities and coordinates. GoogLeNet was the inspiration of the object detection architecture. There are 24 convolutional layers initially followed by two fully-connected layers. Along with the 24+2-layer deep net there is a version with only nine convolutional layers. This is a faster model able to perform object detection at 155 fps.

In 2017 Joseph Redmon and Ali Farhadi released a paper titled 'YOLO9000'. They described modifications that mainly increased the accuracy of YOLO, and named the architecture YOLOv2 [16]. Among the updates they presented were batch normalization added to the convolution layers of YOLO, this alteration increased the mAP by 2 percent. Batch normalization essentially makes a layer able to learn more independently of other layers [10]. The authors also changed the training process. In YOLO one trains a deep neural network such as VGG16 for image classification with images of size 224×224, followed by end-to-end training with images of size 448×448. In YOLOv2 they added a fine tuning phase to the basic image classifier where they finetune the network with images of size 448×448 after they trained it on 224×224 images. The alteration further improved the mAP with 4 percent. Further improvements were made by adding standardized anchor boxes to help guess where objects are present, the precision was somewhat reduced but the recall improved. Other improvements were made by adding multi-scale training, direct location predictions and finer-grained features. The final results revealed an mAP of 78.6 for YOLOv2 while YOLO has an mAP of 63.4.

Recent updates to YOLO were presented April of 2018 in a paper titled 'YOLOv3: An Incremental Improvement' [Farhadi]. The changes they made focused on improving the speed as well as changing the the classification function to be multi-label since objects can have classifications that are non-exclusive. In the new architecture they added a feature pyramid network to help perform predictions at multiple scales. Furthermore, they changed the feature extractor into a 53-layer Darknet, instead of Darknet-19. Darknet-53 achieves the same classification accuracy as ResNet-152, however, it is 2 times faster. One of the great achievements of the new system is a significant improvement on detecting small objects. The authors of the paper observed great speed on the COCO test-set.

2.4.3 SSD

SSD - single shot multi-box detector is yet another powerful object detector, published in 2016, that has gotten a significant amount of popularity in recent years [13]. It is a real-time capable object detector. It is more accurate in its predictions than the first version of YOLO. The entire pipeline of the object detector is implemented as a single convolutional deep neural network architecture without the need for a separate object proposal stage. After the network has produced bounding box and class predictions it runs a non-maximum suppression, NMS. NMS is used because SSD generates a lot of bounding boxes in a forward pass, and it is hence important to trim away irrelevant ones. Bounding boxes with confidence scores below a certain threshold are discarded, and only top-N predictions are kept.

The center of SSD is a high-quality CNN architecture for image classification, VGG16, as seen in figure 2.5. The classification layers of the base model are removed and replaced by an auxiliary structure containing extra feature maps used to produce the detections. The auxiliary structure contains a series of convolution layers that are progressively decreasing in size. Having multiple sizes of the convolution layer at the end allows the network to perform high accuracy detection at multiple scales. An illustration of the architecture is given in figure 2.5. Each of the layers at the end produces a fixed number of bounding box suggestions. The way this works is by running a $3 \times 3 \times p$ small convolution kernel over the feature map produced by the convolution layer. The output of running the small kernel is either a score for a category or a shape offset.

To make the detector able to detect objects at multiple scales the constructed feature map cells have, b, pre-determined anchor boxes as illustrated in figure 2.6. The anchor boxes are carefully constructed to match common scales and aspect ratios of detected objects. Each of the anchor boxes are described by four coordinates. Hence the output of a given feature map with a grid size of dimensions $m \times n$ is a tensor of size $m \times n \times b \times (4 + c)$ values. c denotes the number of classes to predict.

Furthermore, SSD added hard negative mining, which is a method to restrict the number of negative examples to use during training. A lot of the bounding boxes produced during training will have a low intersection over union, and one should not use all of them as negative examples in the training phase. Adviced ratios from the paper is 3:1 in negative-positive ratio. The reason to use negative examples at all is in order to tell the network which instances are not objects of interest.

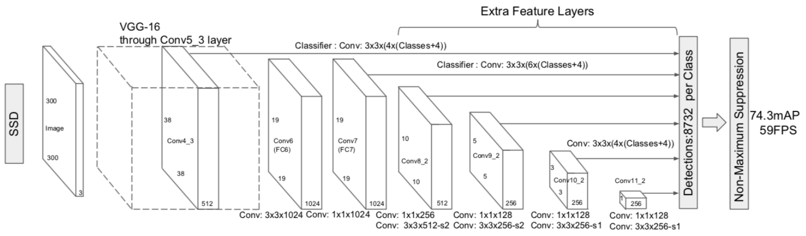
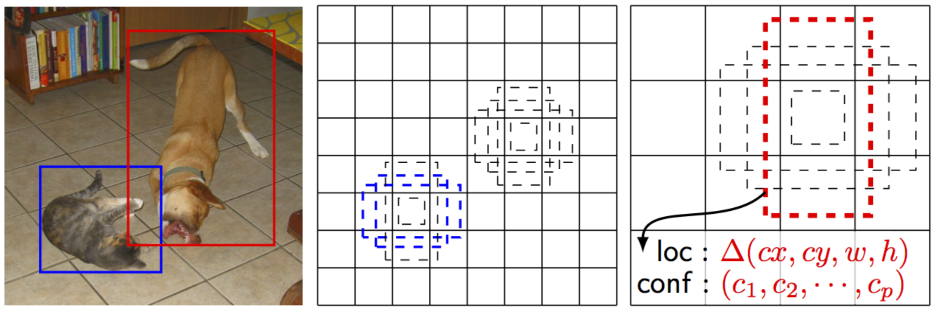


Figure 2.5: SSD Architecture [13].


loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$

(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.6: SSD Anchors [13].

2.5 Object detection software

The following sections of the paper will address how one can practically implement neural network architectures and object detectors. The First section will cover Googles deep learning framework known as Tensorflow and Keras.

2.5.1 TensorFlow

TensorFlow is a powerful software framework used for creating, training and running neural networks. The framework contains several levels of abstraction starting with TensorFlow Core which provides complete programming control. For most developers, the higher-level APIs are usually recommended. The essential data unit in the framework is the tensor, which is a multi-dimensional matrix. This is used to represent the networks themselves and their outputs.

Programs made in TensorFlow usually consists of three parts:

1. Creating a neural network.
2. Training a neural network.
3. Running inference using a neural network.

The system can run on both CPUs and GPUs. It is available for Linux, macOS, Windows and mobile computing platforms. Moreover, there is an abstraction layer called Keras that builds on top of TensorFlow that enable developers to iterate faster when developing deep neural nets.

Tensorboard for analysis

To better understand the progress of neural networks undergoing training, Google created a tool called Tensorboard. Tensorboard is a web-application that peeks into a log file produced by neural networks in the training phase. One can then easily access various metrics of the network. Typical measures one can access are a loss, learning-rate, and accuracy.

TensorFlow object detection API

TensorFlow additionally provides an Object Detection API. The API provides a wide range of capabilities related to object detection. One of the core features of the API are pre-trained object detection models. The available models are trained on various datasets such as KITTI and Coco. Training an object detector from scratch takes weeks to get desired results, therefore being able to utilize

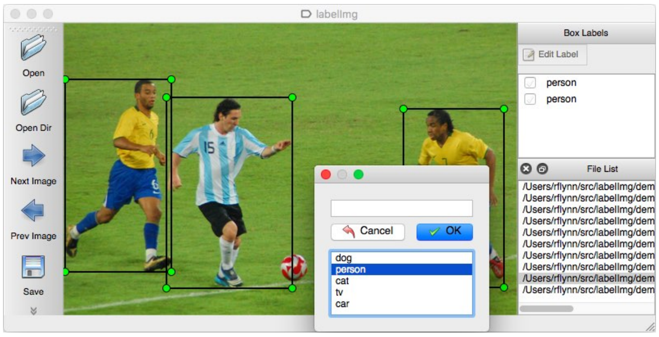


Figure 2.7: LabelImg graphical user interface for annotating images [3].

transfer learning and re-purpose object-detectors with a custom dataset is very time-saving. The models they provide are SSD and faster R-CNN in various flavors and configurations.

To create a custom object-detector using the API, one has to create a configuration file. In that file, one specifies which dataset one wishes to utilize, the hyperparameters related to training, in addition to the desired augmentations and regularizations.

2.5.2 LabelImg for creating custom datasets

LabelImg is a graphical image annotation tool built for Python. One import images, then annotate them with bounding boxes and finally save the data as a CSV file [3]. The interface of LabelImg can be seen in figure 2.7. The application is available on GitHub.

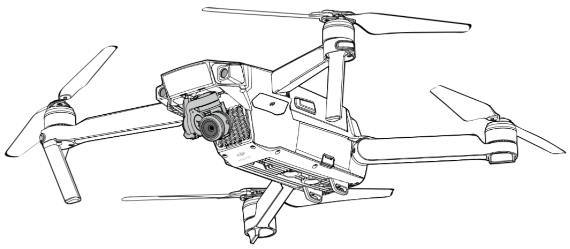


Figure 2.8: The DJI Mavick Pro

2.6 Quadcopter drones - hardware and software

The quadcopter drone used for this thesis is a DJI Mavick Pro model. The following sections will cover fundamentals of drone technology with an emphasis on the aerial vehicle at hand. Among the covered topics are hardware, software, and sensor technology.

2.6.1 Quadcopter drones

Quadcopter drones are multirotor helicopters that are lifted and propelled by four rotors. These aircraft are often created with two clockwise rotating and two counter-clockwise rotating propellers. By differing the rotational speed of each rotor, it is possible to get a desired total thrust and direction, making them very versatile airborne vehicles. At the end of the 2000s, electronic components such as accelerometers, global positioning systems, cameras and flight controllers dropped in price. The price drop spawned a boom of configurations of unmanned aerial vehicles. Drones have also proven to be a useful tool for university researchers to test and evaluate new ideas in a diverse number of fields.

2.6.2 DJI Mavick Pro

The drone utilized in this project is the DJI Mavick Pro as can be seen in figure 2.8. It is a premium commercial drone that one can obtain from most common electronic stores. We chose the model for this project for a couple of reasons:

1. It provides a great SDK for developers.
2. It has 5 on board cameras.

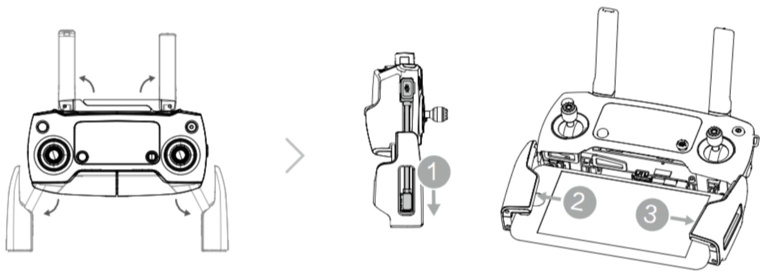


Figure 2.9: The DJI Mavick Pro controller

3. It provides Up to 27 minutes of flight time per charge.
4. It includes an obstacle avoidance system.
5. Under the right conditions it has an operating range of 7km.

To control the Mavic Pro one use the belonging controller depicted in figure 2.9. This controller can be connected to an Android or iOS smartphone for increased control and live view of the drone camera during flight. If one accepts a shorter range, one can also use a smartphone by itself, connecting it directly to the drone. The direct connection link will, however, limit the flying distance to 80 meters, and the altitude will be limited to 50 meters. The smartphone enables some exciting opportunities by letting software engineers create customized software via the provided SDK. We cover the SDK in a couple of sections.

2.6.3 Sensors

There are a number of sensors on the DJI Mavick Pro. Here is an overview:

1. Forward vision system consisting of a stereo vision setup with 2 camera sensors.
2. Downward vision system utilizing ultrasound in addition to a stereo vision setup of 2 camera sensors.
3. RGB camera used as main camera.
4. Accelerometers.
5. Gyroscopes.
6. Compass.

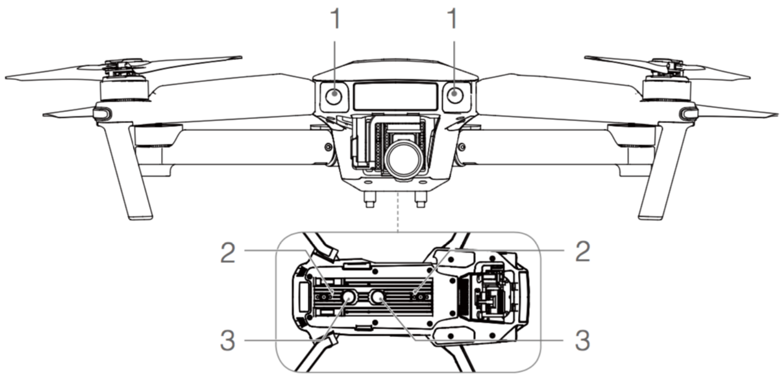


Figure 2.10: The DJI Mavick Pro Downward and upward vision system

7. Dual-band satellite positioning (GPS and GLONASS).

The forward and downward vision system is used to understand the nearby environment in 3D. By having a 3D-model of the nearby environment, DJI created an obstacle avoidance system. It continually scans for obstacles in front of it. These sensors allow the aircraft to dodge objects, maintain its current position, hover precisely and fly indoors or in other environments with weak GPS signal. The placement of the vision system is illustrated in figure 2.10. In this diagram [1] and [2] indicate the camera sensors, and [3] indicates the ultrasonic sensors used for downward vision. There are however a couple of limitations when using the obstacle avoidance system. We highlight the most critical limitations below.

1. The drone can not fly faster than 36 $\frac{km}{h}$.

2. The downward vision system is only functioning at altitudes between 0.3 to 13 meters.

3. It is not recommended to fly above bright (lux $>$ 100 000) or dark surfaces (lux $<$ 10).

The primary camera uses a 1/2.3 inch CMOS sensor with the ability to capture 4K video at 30 fps, and otherwise 12 megapixels still images. The camera is attached to a 3-axis gimbal that makes sure the camera is steady during flight. Thanks to the gimbal, the camera can look in a variety of directions.

2.6.4 API and capabilities

One of the most interesting aspects of this drone from a computer science point of view is the ability to tell the drone what to do programmatically. One realizes this through DJI' mobile SDK. The code is going to run from a smartphone that is connected to the drone, and not on the drone itself. The software development kit is available for both iOS and Android. The platform choice implies that one can write code for the DJI Mavick Pro with Swift, Objective-C or Java. There are a variety of capabilities provided through the SDK. It gives high and low-level flight control and access to the aircraft state through telemetry and sensor data. One can furthermore utilize the obstacle avoidance system, the camera, and the gimbal. The gimbal can be fixed upon a desired level of accuracy. One can get a live video feed of what the main camera of the drone is observing in addition to remote access to all the media stored on the drone. Since the project members are experienced iOS-developers with access to iOS devices, we chose to do the project with the iOS SDK.

Before one can start developing with the SDK, there are a couple of steps required. One needs to register as a developer at `developer.dji.com`. After which one has to generate an app key at the website and add to the info file in the Xcode project. The easiest way to set up a project with the DJI SDK is by using the iOS package manager Cocoapods.

2.6.5 Development workflow

When programming with the DJI SDK, the development workflow is as follows:

1. Connect smartphone to computer.
2. Write code to test.
3. Upload code to smartphone.
4. Unplug smartphone.
5. Turn on hand-held controller.
6. Turn on DJI Mavick Pro.
7. Wait for the hand-held controller to connect to the aircraft.
8. Plug your phone to the hand-held controller.
9. Run the code.

The beforementioned process is however quite tedious. It makes debugging difficult and time-consuming since every code iteration is going to demand the list of steps above to be repeated. Fortunately, there is a way to make the workflow more efficient. To aid in the development process DJI has created a Bridging app, peculiarly enough called SDK Bridge, that is available on the App Store. The Bridge SDK app enables code execution through a smartphone simulator on the computer. Look at figure 2.12 for a schematic overview of utilizing the bridging application. The only requirement for it to work is that the phone and the computer used for development is connected to the same wireless internet router and that the single line of code shown below is added to the app.

```
DJISDKManager.enableBridgeModeWithBridgeAppIP(”xxx.xxx.xxx.xx”)
```

The added SDK Bridge makes programming on the drone quite seamless. Now one can access the built-in debugger of Xcode and all the functionality of the IDE. Even with access to the debugging features of Xcode, running code out in the field is not practical, since it demands that one brings the drone to a location suited for flying. To help out with this issue, DJI created a simulator for the drone.

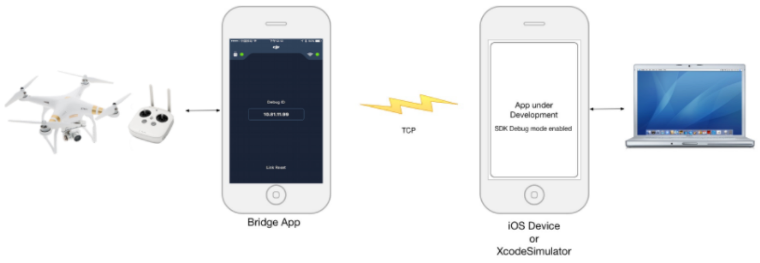


Figure 2.11: The workflow when the Bridge SDK app is connected.

2.6.6 DJI aircraft simulator

The aircraft simulator is an elegant tool to use during development cycles. To use it one has to download and install an application called DJI Assistant 2 from `www.dji.com/phantom-4/info#downloads` on a computer. To use it one connects the DJI Mavick Pro via USB to the computer with the Assistant installed on it and launches the simulator. The flight simulator is great when testing new software since there is no risk involved when testing in a virtual environment, and one can easily observe the behavior of the drone when testing new programs.

When in simulation mode, the aircraft can take control input from the remote controller or application code written with the SDK. It will simulate the aircraft behavior based on the inputs it receives, and it outputs state information based on the simulation. This state information includes velocity, acceleration, and orientation.

2.7 Network programming

The client part of the system presented in this thesis is running on a smartphone with modest computational power. Unfortunately, deep neural networks require significant processing, and as such the need for more computational capabilities arose. To meet the processor demand we chose to make use of external computing power. Off-device neural network inference involves intricate networking as the client needs to be able to communicate with external services. Our system requires fast delivery of images and neural network predictions. Finding readily available frameworks for streaming video in real-time was hard to come by for our platform. Consequently, we ended up creating a custom communication pipeline. The sections below will cover some of the fundamental technologies used for creating the communication pipeline between the drone client application and external services.

2.7.1 Client-server architecture

When two computers want to communicate, a typical pattern used for development is the abstract client-server model. In this model, the server has services which can be called upon by clients. One can either use published, and readily available communication protocols published in RFC such as FTP, or one can create custom protocols. In the implementation of the system used in this thesis, we created a custom communication protocol by utilizing functionality offered at the application layer of the network stack - sockets.

2.7.2 Sockets

Network sockets are endpoints for sending and receiving data over a computer network or by use internally in a computational node for inter-process communication [12]. Processes in a computer can refer to sockets using socket descriptors. Sockets can enable both persistent and non-persistent connectivity between two computers. To establish communication between two computers using this technique, the client has to specify the IP-address of the device it wishes to initiate contact with, in addition to the port number associated with the socket where the server is listening. There are three main types of sockets available as shown in the list below.

1. Datagram sockets utilizing the user datagram protocol (UDP).
2. Stream sockets utilizing the transmission control protocol. (TCP)
3. Raw sockets which are typically available in routers.

The ones that are of interest to this thesis is the TCP-based stream socket and the UDP-based datagram socket.

TCP

TCP is a connection-oriented protocol. It means that the two communicating entities have to perform a handshake to establish a TCP-connection. One of the endpoints attached to this connection is associated with the client socket address, and the other one is associated with the server socket address. When the connection is established, the entity that wishes to send data sends it to its socket. It is the job of the client to initiate this connection. From the viewpoint of the two communicating entities, they are connected by a solid pipe, everything they send in is guaranteed to come out the other end. TCP is reliable, it provides mechanisms to deal with creation and destruction of connections, in addition to proper error handling. TCP also provides reliable, in-order data transfer guarantee. The life-cycle of a TCP-connection is shown in figure 2.12.

UDP

UDP is a connectionless protocol. Because of this, every packet needs a destination address when being sent, that address comprises the IP-address of the host in addition to the port number associated with the socket. There is no guarantee that the data sent with UDP will make it to the receiving device, and there is no assurance about the ordering of the transmitted datagrams. UDP takes messages to be sent from the application process, attaches the source and destination addresses, and sends it to the network layer. However, the fact that the UDP protocol is so bare bones means that the application developer has a more granular level of control over the data transmission. UDP is a lot faster, and in many cases better when there is a need for real-time communication.

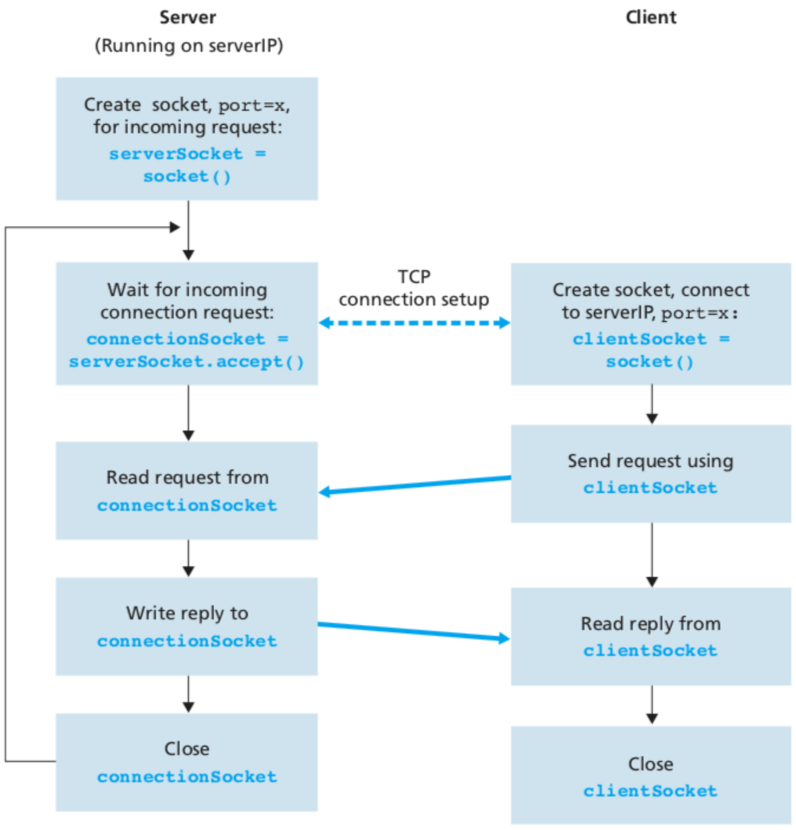

Server
(Running on serverIP)
Client
Create socket, port=x, for incoming request:
serverSocket = socket()
Wait for incoming connection request:
connectionSocket = serverSocket.accept()
TCP connection setup
Create socket, connect to serverIP, port=x:
clientSocket = socket()
Read request from connectionSocket
Send request using clientSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

Figure 2.12: The life cycle of a TCP connection [12].

2.8 Spherical Distance Calculations

Later in the thesis, we will present work related to estimation of global positions from image data. Those calculations are to a large degree built on top of former spherical distance calculation models. The most important one is known as Vincenty's formula.

2.8.1 Vincenty's formula

Thaddeus Vincenty was a Polish American who worked with geodetics, the study of accurately measuring and understanding the earth basic shape, its orientation in space and its gravity field [25]. The methods known as Vincenty's formulae are two iterative methods to calculate the distance between a couple of points on the surface of a spheroid, they are known as Vincenty's direct and indirect method. They assume the shape of the earth to be an oblate spheroid and are hence a lot more accurate than the distance calculations assuming a spherical Earth. An oblate spheroid is a shape obtained if one rotates an ellipse around its minor axis. The methods are widely used in geodesy because they have accuracy within the range $\pm 0.5mm$. The direct method is the most relevant one for this thesis and hence it will be presented in its entirety while we omit a presentation of the inverse method. The direct method computes a destination point, given an origin point, an orientation and a distance. Vincenty's inverse method takes two points and calculates the metric distance between them. The proper notation is as follows:

Notation
a: length of major semi-axis of the ellipsoid
b: length of minor semi-axis of the ellipsoid
f: flattening of the ellipsoid
ϕ_1: latitude of point 1
λ_1: longitude of point 1 on auxiliary sphere
ϕ_2: latitude of point 2
λ_2: longitude of point 2 on auxiliary sphere
$U_1 = arctan[(1-f)tan(\phi_1)]$: reduced latitude of point 1
$U_2 = arctan[(1-f)tan(\phi_2)]$: reduced latitude of point 2
$L = L_2$ - L_1: difference in longitude of two points
α: azimuth at equator
α_1: initial bearing (bearing: angle away from true north of distant point as observed from current point. North = 0°, East = 90°)
α_2: final bearing
s: ellipsoidal distance between the two points
σ: arc length between points on auxiliary sphere

Direct method

Vincenty's direct method calculates the location of a coordinate (ϕ_2, λ_2), given a distance (s) and a bearing (α_1) from another known point(ϕ_1, λ_1). Pseudocode for the method is available at [24]. The calculation goes according to these steps:

$$tan(U_1) = (1 - f) \cdot tan\phi_1 \tag{2.1}$$

$$cos(U_1) = \frac{1}{\sqrt{1 + tan^2(U_1)}} \tag{2.2}$$

$$sin(U_1) = tan(U_1) \cdot cos(U_1) \tag{2.3}$$

$$\sigma_1 = arctan(\frac{tan(U_1)}{cos(\alpha_1)}) \tag{2.4}$$

$$\sin(\alpha) = cos(U_1) \cdot sin(\alpha_1) \tag{2.5}$$

$$cos^2(\alpha) = 1 - sin^2(\alpha) \tag{2.6}$$

$$u^2 = cos^2(\alpha) \cdot \frac{a^2 - b^2}{b^2} \tag{2.7}$$

$$A = 1 + \frac{u^2}{16384} \cdot (4096 + u^2 \cdot (-768 + u^2 \cdot (320 - 175 \cdot u^2))) \tag{2.8}$$

$$B = \frac{u^2}{1024} \cdot (256 + u^2 \cdot (-128 + u^2 \cdot (74 - 47 \cdot u^2))) \tag{2.9}$$

$$\sigma = \frac{s}{b \cdot A} \tag{2.10}$$

$$cos(2\sigma_m) = cos(2\sigma_1 + \sigma) \tag{2.11}$$

$$\begin{aligned}\Delta\sigma = B \cdot sin(\sigma) \cdot (cos(2\sigma_m) + \frac{B}{4} \cdot (cos(\sigma) \cdot (-1 + 2 \cdot cos^2(2\sigma_m))))\\ - \frac{B}{6} \cdot cos(2\sigma_m) \cdot (-3 + 4 \cdot sin^2(\sigma)) \cdot (-3 + 4 \cdot cos^2(2\sigma_m))\end{aligned} \tag{2.12}$$

$$\sigma' = \frac{s}{b \cdot A} + \Delta\sigma \tag{2.13}$$

Repeat equations 2.11, 2.12 and 2.13 until the change in σ is negligible ($\approx 10^{-12}$). Then perform:

$$\begin{aligned}\phi_2\\ = arctan(\frac{sin(U_1) \cdot cos(\sigma) + cos(U_1) \cdot sin(\sigma) \cdot cos(\alpha_1)}{(1 - f) \cdot \sqrt{sin^2(\alpha) + (sin(U_1) \cdot sin(\sigma) - cos(U_1) \cdot cos(\sigma) \cdot cos(\alpha_1))^2}})\end{aligned} \tag{2.14}$$

$$\lambda = arctan(\frac{sin(\sigma) \cdot sin(\alpha_1)}{cos(U_1) \cdot cos(\sigma) - sin(U_1) \cdot sin(\sigma) \cdot cos(\alpha_1)}) \tag{2.15}$$

$$C = \frac{f}{16} \cdot cos^2(\alpha) \cdot (4 + f \cdot (4 - 3 \cdot cos^2(\alpha))) \tag{2.16}$$

$$\begin{aligned} L = \lambda - (1 - C) \cdot f \cdot sin(\alpha) \\ \cdot (\sigma + C \cdot sin(\sigma) \cdot (cos(2\sigma_m) + C \cdot cos(\sigma) \cdot (-1 + 2 \cdot cos^2(2\sigma_m)))) \end{aligned} \tag{2.17}$$

$$\lambda_2 = \lambda_1 + L \tag{2.18}$$

$$\alpha_2 = arctan(\frac{sin(\alpha)}{-(sin(U_1) \cdot sin(\sigma) - cos(U_1) \cdot cos(\sigma) \cdot cos(\alpha_1))}) \tag{2.19}$$

2.9 Image processing frameworks

The system developed in the thesis deals a lot with image data and utilizes various software frameworks to help process it. The following subsections will give a brief presentation of the most important ones.

2.9.1 OpenCV

The Open Source Computer Vision Library, OpenCV, is a software framework centered around computer vision and machine learning. It contains over 2500 optimized algorithms. The capabilities of the library are diverse, it can, for instance, detect and recognize faces, track moving objects, produce 3D point clouds from stereo cameras and much more. It is a framework utilized in both research and industry. The library is written natively in C++ but is available for a wide array of platforms such as iOS, Linux, Windows and Mac OS [4]

2.9.2 FFMpeg

FFmpeg is a collection of tools and libraries that are capable of processing multimedia content such as audio and video with related metadata. The main features are related to encoding, decoding, muxing and demuxing of data. The framework runs on Linux, Mac OS X, and many other platforms. It is made in the language C [2].

2.10 Infrastructure

Sophisticated neural networks have several requirements to work correctly, both regarding software and hardware. This section will cover infrastructure-related software applied to this thesis.

2.10.1 Docker

Containerization is the process of performing virtualization at the operating-system-level, Docker is a widely used platform for doing this. Dockers primary OS is Linux. By taking advantage of the resource isolation features in the Linux kernel, Docker can create independent containers running alongside each other without knowledge of each other's existence. Docker has two essential concepts: images and containers. An image is a combination of a file system and associated parameters. The image can, for instance, be an entire operating system with desired software frameworks, applications, and libraries. By creating an image, one can easily port software over to an entirely new computer and essentially duplicate pre-existing software environments. It is also common for large vendors of software to produce images for their systems. NVIDIA has created images that are ready with CUDA drivers and software for running GPU dependent code such as Tensorflow with neural networks.

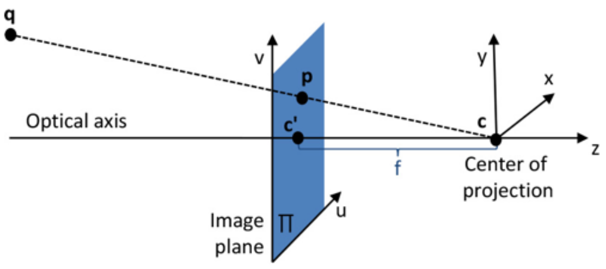


Figure 2.13: The pinhole camera model [19].

2.11 Camera calibration

Camera calibration is the process of measuring the internal camera parameters such as the focal length [19]. The most popular methodologies for camera calibration were proposed by Tsai [23] and Zhang [26]. Here it is assumed that a set of point correspondences between 2D and 3D points of a known reference object also called a calibration target are known. Popular calibration targets are checkerboards or rectangular grids of dots. To understand camera calibration one has to be familiar with the pinhole camera model.

The pinhole camera model is a modeled abstraction of a regular physical camera. The model represents a perspective projection of a 3D-point, **q**, in object space to a 2D-point, **p**, in image space. **c** is called the center of projection, which is the point where all the observed projections must pass. In a camera obscure, **c** is the hole where the light comes in. The line through **c** and **c'** is called the optical axis and the distance from **c** to **c'** is the focal length. We illustrate the model in figure 2.13. The perspective projection of points from object space to image space can be set up as following using homogeneous coordinates, where **M** is a 3×4 matrix representing the perspective projection.

$$\begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix} \propto M \begin{bmatrix} q_x \\ q_y \\ q_c \\ 1 \end{bmatrix} \tag{2.20}$$

The matrix **M** depends on both internal and external camera parameters. Internal parameters are intrinsic properties of the camera while the external parameters are related to the pose of the camera. The perspective projection matrix **M** is as follows.

$$M = K[R|t] \tag{2.21}$$

Where K is an upper triangular matrix containing the internal parameters of the camera, R is a 3×3 rotation matrix, and t is a translation vector. $[R|t]$ is a 3×4 matrix that in total describes the cameras pose relative to a defined world origin. The internal camera matrix K looks as follows.

$$K = \begin{bmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \tag{2.22}$$

When we perform camera calibration we are finding the values inside the matrix K. f_u, and f_v are the focal length of the camera scaled by the size of a pixel in the directions u and v. However, most modern cameras have square pixels so $f_u = f_v$ may be assumed. c_u and c_v is the offset in image coordinates for the principle point c'. s is a skew factor only non-zero if the image-directions u and v are not perpendicular.

When one calibrates a camera, one uses the camera to calibrate, take several images of a calibration target at multiple poses. The images of the calibration target are used as a baseline by Tsais calibration method. The point **q** in the world frame and the point **p** in image space are related to the perspective projection **M** as in equation 2.23. The reason one can set the relation between the points **p** and **q** to 0 is because they, when represented as vectors, point in the same direction, hence their cross-product equals 0.

$$\mathbf{p} \times \mathbf{Mq} = 0 \tag{2.23}$$

By performing various direct linear transformations such as SVD, one can compute **M** from several images relating 2D-points to 3D-points. From M we also have the internal matrix K. By figuring out the internal matrix K, we can also find the field of view of the camera. OpenCV has several functions in its toolbox to help with the camera calibration process [1]. OpenCV's implementation gives the field of view along the width, w, and height, h, of the image following the formulas below.

$$fov_y = 2archtan(\frac{h}{2f_y}) \tag{2.24}$$

$$fov_x = 2archtan(\frac{w}{2f_x}) \tag{2.25}$$

Chapter 3

Methods and implementation

The goal of this thesis is to create an autonomous quadcopter drone system with real-time analytic functionality for use in inspection and surveillance. Several tasks have to be done adequately to achieve that goal. First and foremost there is a need to create a system that enables the quadcopter drone system to fly autonomously without human intervention. Secondly, there is a need to deploy an object-detection system on a unit with sufficient computational power. In this part of the paper, we will outline how we approached these challenges.

3.1 Platform for autonomous drone operations - infrastructure and overview

The quadcopter drone SDK used in this thesis runs on mobile platforms such as iOS and Android. Running large neural networks on mobile devices is however not a viable option for this project. The reason is that it does not have sufficient computational power as can be seen in table 3.1. Empirical tests have previously revealed an attained frame rate of 1.7 fps when running shallow versions of YOLO on the client platform. Therefore the remaining option was to utilize external computational power, either through cloud-based infrastructure or separate servers. Consequently, the system ended up with two different components, a drone client used for controlling the quadcopter and extracting sensor data, and also a server component used for object detection. The system's structure will be presented in the sections to follow. We start by presenting the setup and architecture of the client.

3.1.1 Client architecture

We deployed the client part of the system on an iPhone 6 with iOS. The responsibilities of the client are to initiate missions, control the drone, and extract relevant sensor data to forward to the external server. The client has several capabilities. Among them is the ability to start point based missions, where one designated specific geolocations for the drone to fly to, in addition to area coverage missions, where one marks of an entire area on a map that shall be covered by the quadcopter. The client is also responsible for configuration of the mission-related parameters, such as quadcopter altitude, speed and heading during flight. One should furthermore be able to get feedback during mission execution, such as live video from the drone, in addition to feedback consisting of detections from the server. The client was broken down into separate component classes with unique responsibilities and structured according to the MVC-paradigm. See figure 3.1 for a class diagram displaying the client system in its entirety. The client was implemented in Objective-C++ to have access to POSIX socket networking code also used on the server. The development process ended up with the steps given in the undermentioned list.

1. Create graphical interface client application for the smartphone.
2. Connect client application to drone.
3. Create mission operator to manage drone missions.
4. Connect client to external server.

Graphical interfaces are natural starting points for development. The main GUI component of the client application is a map view used to steer the drone to different geographical coordinates. The map is used to select waypoints and also mark areas to cover to initiate missions.

Enabling the drone to talk to the client was the second part of the process. We performed this by calling functions in the DJISDKManager part of the API.

Mission operator implementation enabled the client to push control commands to the drone. We materialized it by implementing various protocols provided by DJI.

The first required protocol was the DJISDKManagerDelegate protocol. It sends the delegate class updated registration status and change of the aircraft status.

The second protocol of importance was the DJIFlightControllerDelegate. That

Table 3.1: Specifications of iPhone 6 used as platform for the drone application

	iPhone 6
Memory	1 GB RAM DDR3
Graphics	PowerVR GX6450 (quad-core graphics)
Processor	Dual-core 1.4 GHz Typhoon (ARM v8-based)

protocol allows the client to manipulate the aircraft, for instance, by sending control commands, adjusting altitude, velocity and so forth.

The third protocol that needed support was DJIBaseProductDelegate. That protocol qualifies the application to receive notifications from the drone when there is connectivity changes, or other internal components changes, plus one gets diagnostic information from the drone at regular intervals.

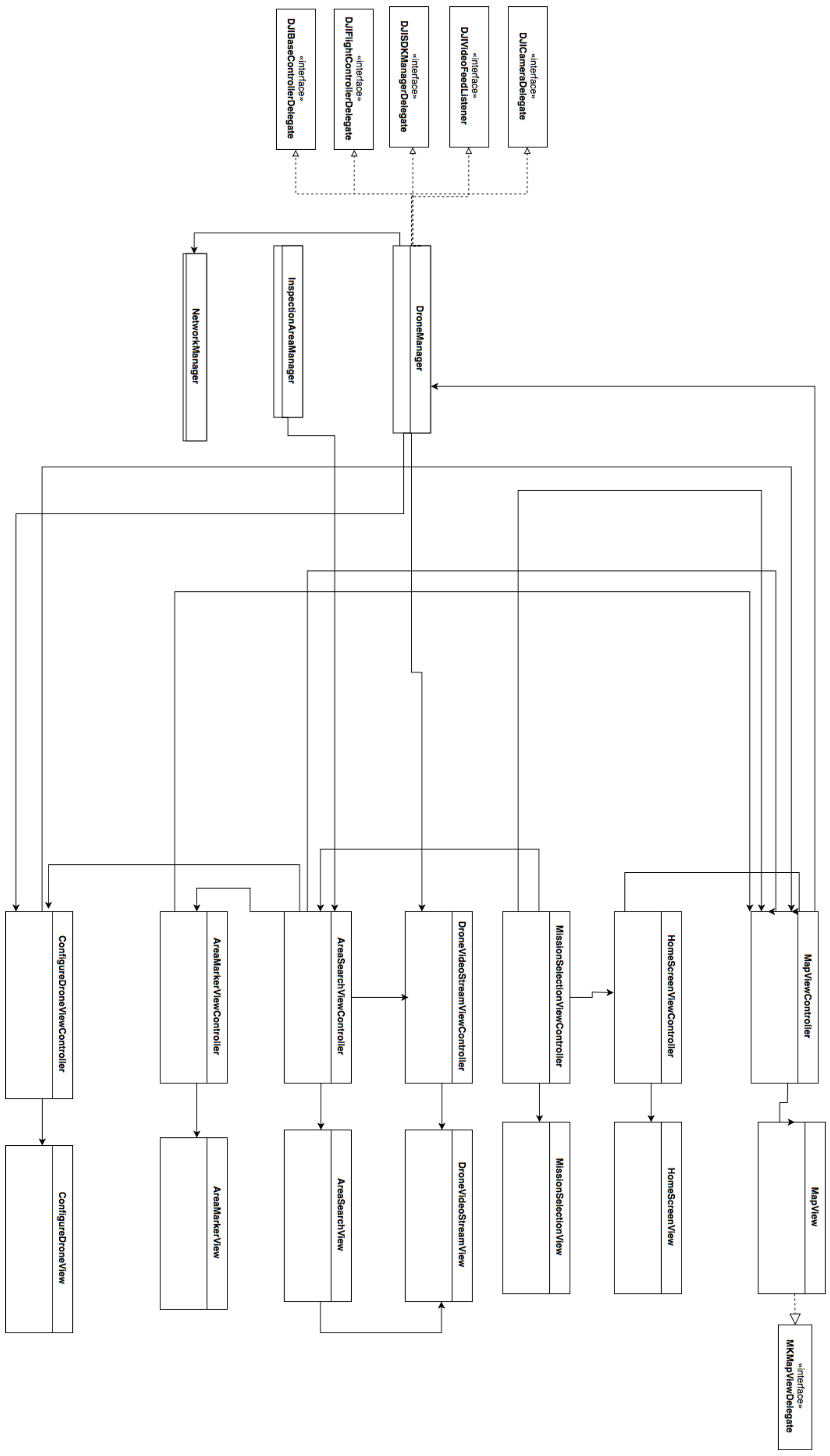


Figure 3.1: A class diagram depicting the architecture of the client side of the system.

Streaming video from DJI drone

An important requirement for the client is to extract the video stream from the drone and forward it to the external server for processing. The process of extracting it did, however, prove to be a lot more convoluted than first thought. In this section, we will briefly lay out the process.

DJI provides a nice framework called VideoPreviewer for its SDK to view the live video feed from the aircraft on the client application. However, it does not give access to individual video frames. It only enables implementation of a view to display video on the mobile device with decoding happening behind the scenes inside the framework. The design of the framework caused a problem for this project since it depends on being able to send images for processing to the external server. We went into a couple of dead-ends regarding the extraction of video during this project and spent roughly 1 month figuring out how we could decode the video. The answer lay in the multimedia library FFmpeg that we referenced in section 2.9.2.

What DJI makes available in the framework are h.264 encoded binary blobs of data that contains the video. Our initial idea was to send this binary blob of data to the server, perform decoding of the h.264 video and later perform the analysis on the decoded image. A video usually consists of a container which wraps streams of audio and video. The individual data elements inside that stream are called frames, and a codec encodes each of those frames. To decode the frames, one passes packets from the streams inside the video to a decoder and perform some action on the decoded image. The process is as follows:

1. Open video stream.
2. Read packet from video stream into frame.
3. If frame is valid then perform action on the frame otherwise go back to step 2.
4. Go back to step 2.

The problem was that DJI did not provide sufficient information to create a decoder, nor would they disclose how they did it through their developer forum. To create a decoder one needs an array of parameters, none of which were provided. Much time was spent trying to create a functioning decoder but to no avail. Our effort to create a decoder ended up as figure 3.2 shows. However, after working many hours with the FFmpeg library, we started to grasp the underlying concepts of decoders. What we ended up doing was modifying the VideoPreviewer framework from DJI and inject custom functions. We essentially had to hack their

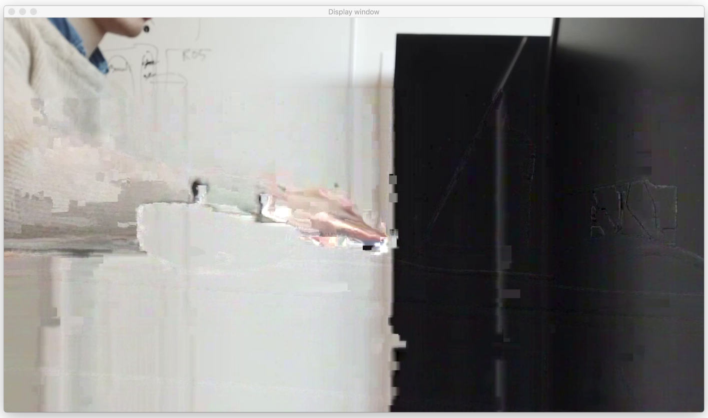


Figure 3.2: The attempt to create a decoder on the server did something correct, but was far from perfect.

framework to extract the desired image data. Deep inside the framework we discovered a place were the decoded AVFrames were being forwarded to their view classes, so we created a function there that forwarded that frame to our NetworkManager class. The NetworkManager used OpenCV to convert the AVFrame to a regular image, and that image was sent to the server for processing.

Waypoint-missions

One of the types of autonomous capabilities implemented for the quadcopter drone in this thesis is the waypoint-mission. In the waypoint-mission, the user selects waypoints on a map as destinations for the drone to cover during flight. One can alter parameters for the waypoint-mission such as drone heading, speed, altitude and actions at each waypoint. The waypoint-mission is depicted in figure 3.5.

Area cover missions

Another type of autonomous mission added to the system developed in this thesis is the area cover mission. In the area cover mission, the user marks an area on a map for the quadcopter unit to cover. In the area cover mission, it is also possible to modify parameters such as speed, altitude, heading, and waypoint actions.

The area cover missions are generated by taking the raw pixel coordinates of the marked area on the screen, transform the pixel coordinates to the map view coordinate system, and later receive geographical points indicating the coordinates of the corners. The corner coordinates of the marked area are handed to the InspectionArea-class which generate the missions. The algorithm for creating the waypoints for the area cover missions first calculates the necessary number of target points to cover the desired area, and after that, it calculates the coordinates of each individual waypoint in the mission.

Manual missions

The last mode of operation we added as part of the GUI is the manual mission. The manual mission lets the user fly the drone manually while having access to the capabilities created in the client application. The main usage of the manual mode for this project is for testing of the prediction abilities of the platform. The tests we performed were for instance to hover the drone over a car and run test out the object detection.

Graphical user interface

The client interface is touchscreen based and implemented for iOS-smartphones. Through the GUI the user can control the quadcopter in addition to perform object detection of images taken during missions. The main way to control the drone is through the map view, and as a consequence, the majority of the interface make use of it. By using satellite images, we make it easy to execute missions.

The main screen, as illustrated in figure 3.3, presents a couple of options for the user through a set of buttons. It enables the user to change the drones main camera rotation. It furthermore lets the user perform predictions of the current camera images. Moreover, it also lets the user annotate the current location of the smartphone as a pin on the visible map view. The location annotation is mostly for use in experiments. In the bottom right corner of the screen, a video view is added to display a stream of video from the drones main camera to give good feedback regarding what the drone is currently doing in the operations. The video view is created to be flexible such that it can be moved around and also set into full screen. This is depicted in figure 3.10 and 3.11. The last button lets the user select and initiate mission execution of different quadcopter missions.

After pressing the 'select mission' button the user is presented with three choices. A video showcasing the flow in the client GUI can be found at `https://youtu.be/N0tuARrqEHQ`.

1. Waypoint-mission.
2. Area cover mission.
3. Manual mission.

We show the process of initiating area cover missions in figures 3.6 and 3.7. After marking an area to cover and pressing 'start mission' the drone will execute coverage of the desired area.

Waypoint-missions lets the user select an arbitrary number of geographical coordinates to cover during a mission. The method for creating a waypoint-mission is presented in figure 3.5.

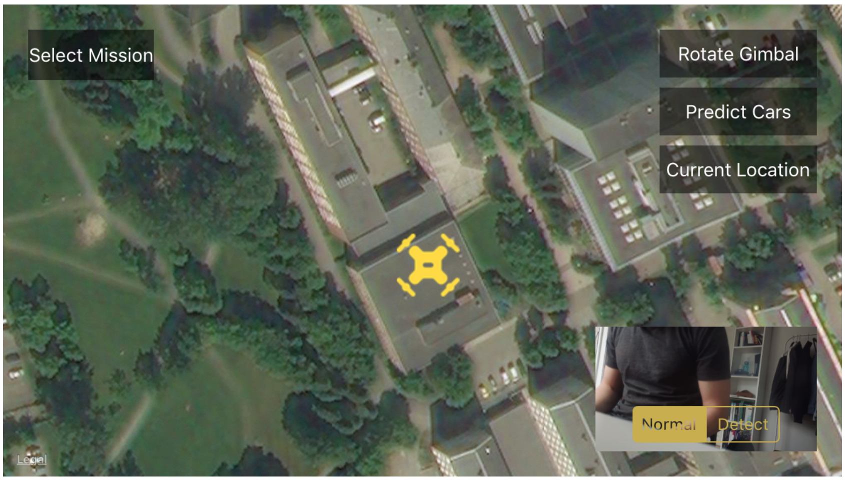


Figure 3.3: The main screen of the user interface. The video view is shown in the bottom right corner. The yellow drone icon denotes the current position of the drone.

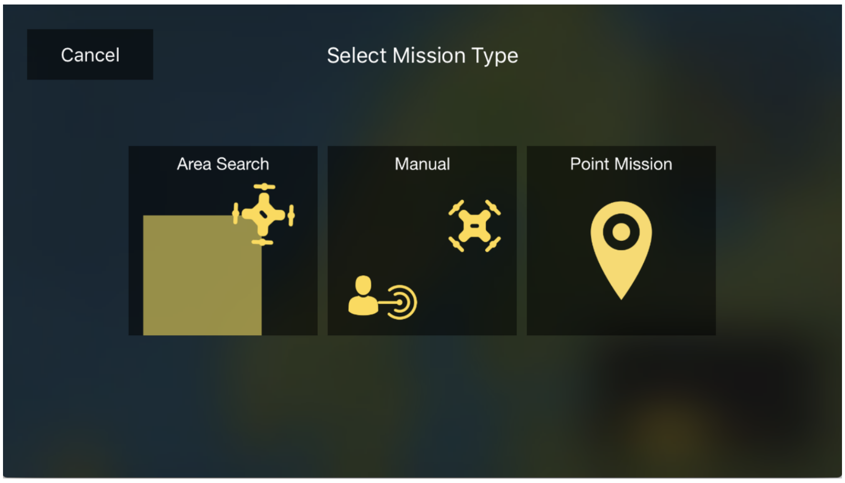


Figure 3.4: When *Select Mission* is pushed in the main screen, one arrives at the mission selection view shown here. From here one can choose which mode to operate the drone in.

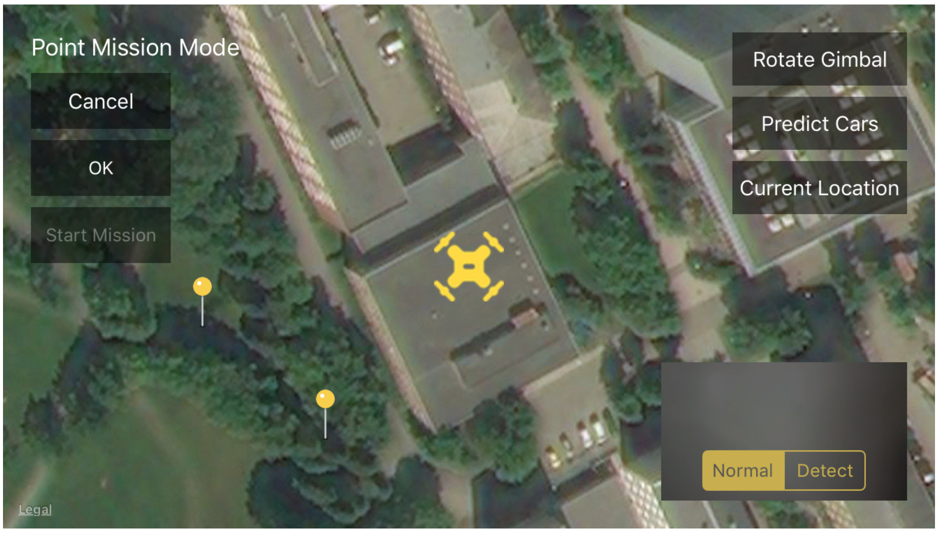


Figure 3.5: The waypoint-mission view with two points selected for the drone to cover in a mission.

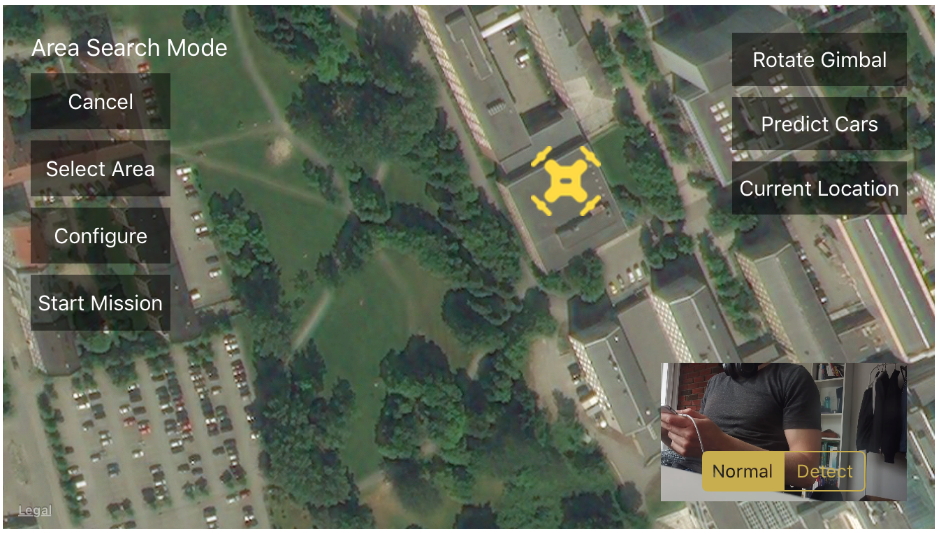


Figure 3.6: This is the area search mode view. From here the user can select an area for the drone to cover.

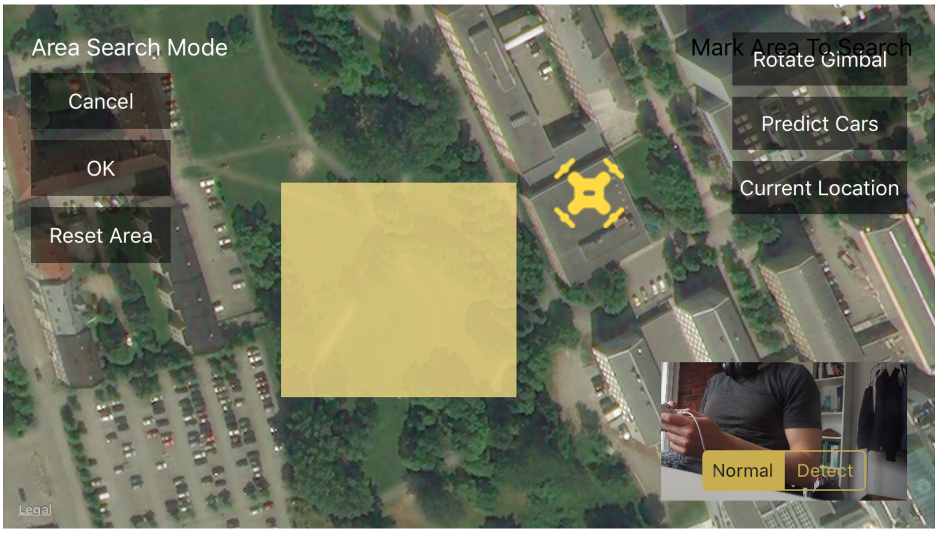


Figure 3.7: Marking an area for the drone to cover.

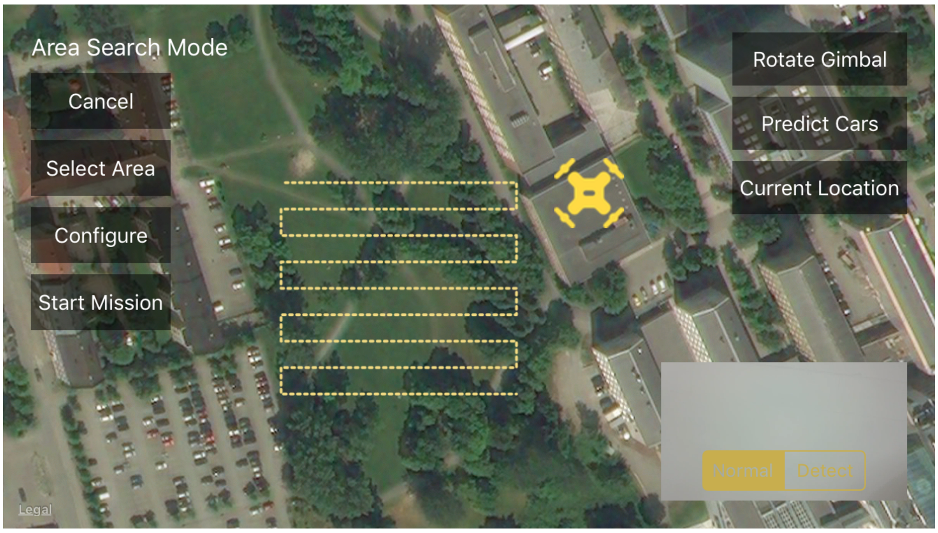


Figure 3.8: A mission created for the drone based on a designated area.

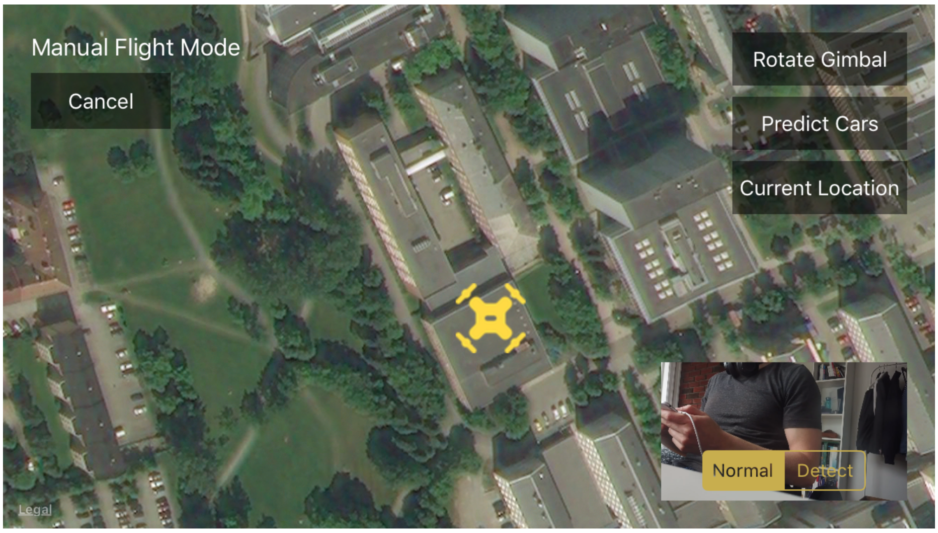


Figure 3.9: This is the manual flight mode view. This lets the user fly around manually.

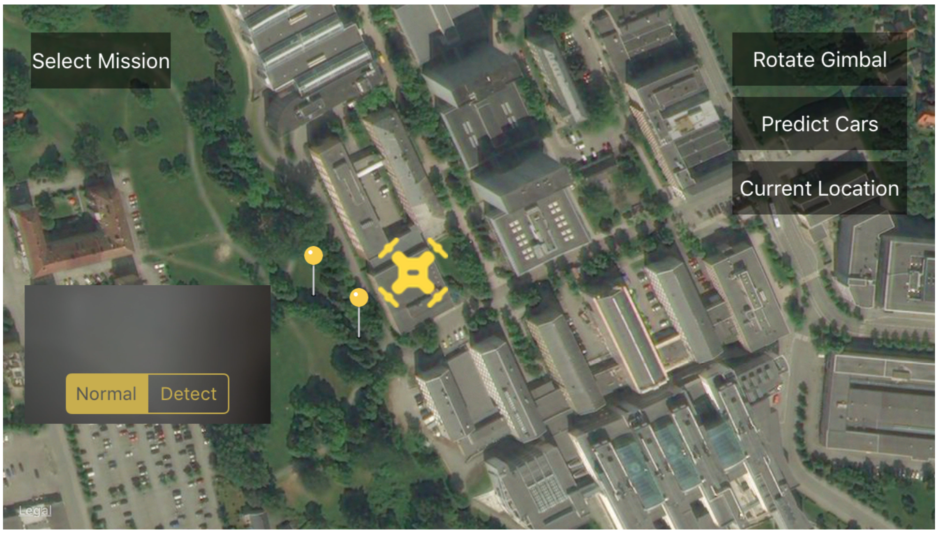


Figure 3.10: An illustration of the flexibility of the video view. It can be moved anywhere on the screen.

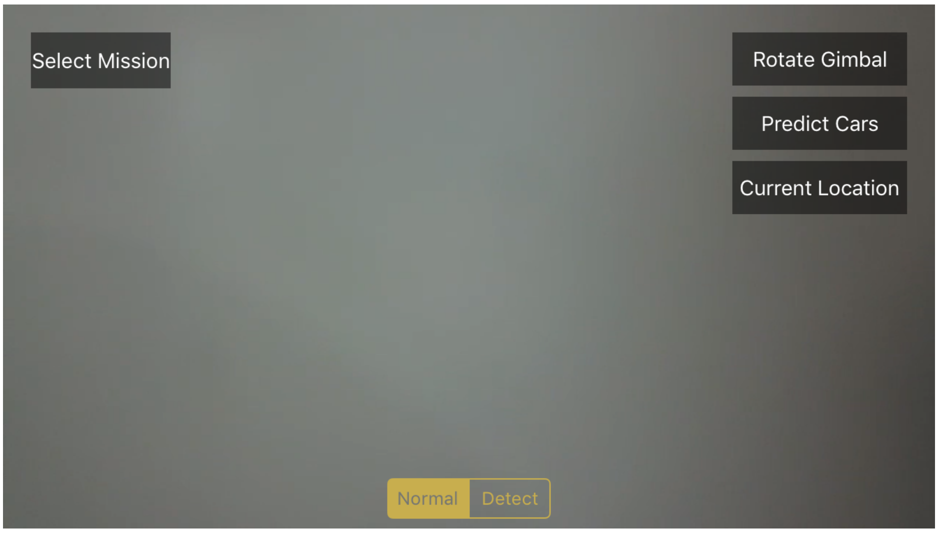


Figure 3.11: The video view in full screen. One can get in and out of full screen video view by tapping the video view.

3.1.2 Server architecture

When the client was fully functional, the focus shifted over to developing the server to be used for neural network inference. One of the available platforms provided by the Department of Computer Science at NTNU was workstation computers residing in the Visual Computing Laboratory at NTNU. The one used for this thesis was a custom built workstation with the specifications seen in table 3.2. We deployed Tensorflow object detection on this instance with GPU acceleration enabled. The object detection inference part of the code was implemented using Python.

After we had deployed the object detection system, it was still isolated from the client system. To enable communication between the two entities we turned the workstation into a server by opening up some ports and making the IP-address of the workstation public on the web. We set the communication up using POSIX sockets as discussed in section 2.7, to get real-time communication capabilities. By creating a code base in C++ for communication, we had highly portable networking code both supported by the client platform and the server. We embedded the C++ code into the iOS-client by creating it in a hybrid language known as Objective-C++ which a mix of Objective-C and C++. We furthermore created custom memory stream classes to wrap the data to be sent over the network, by

Table 3.2: Specifications of the utilized workstation computer at the Visual Computing Laboratory

	Workstation
Memory	32 GB RAM DDR3
Graphics	Nvidia TITAN X
Processor	Intel® Core^{TM} i7-6800K Processor

integrating the OpenCV library we got support for sending image data through these streams. We created one input memory stream for receiving data and one output memory stream for sending data.

We implemented object detection in Python because we needed access to Tensorflow Object Detection. However, we implemented the part of the server communicating with the client in C++ stimulated by a desire of maximizing network communication speed. For the two separate programs to be able to communicate a fair amount of research went into discovering the optimal inter-process communication schema. The solution became using shared memory. By using shared memory, we got the best of both worlds, quick communication and powerful object detection capabilities with no compromise. The way we did it was by creating a smart schema for the two processes to follow as can be seen in figure 3.12 and figure 3.13.

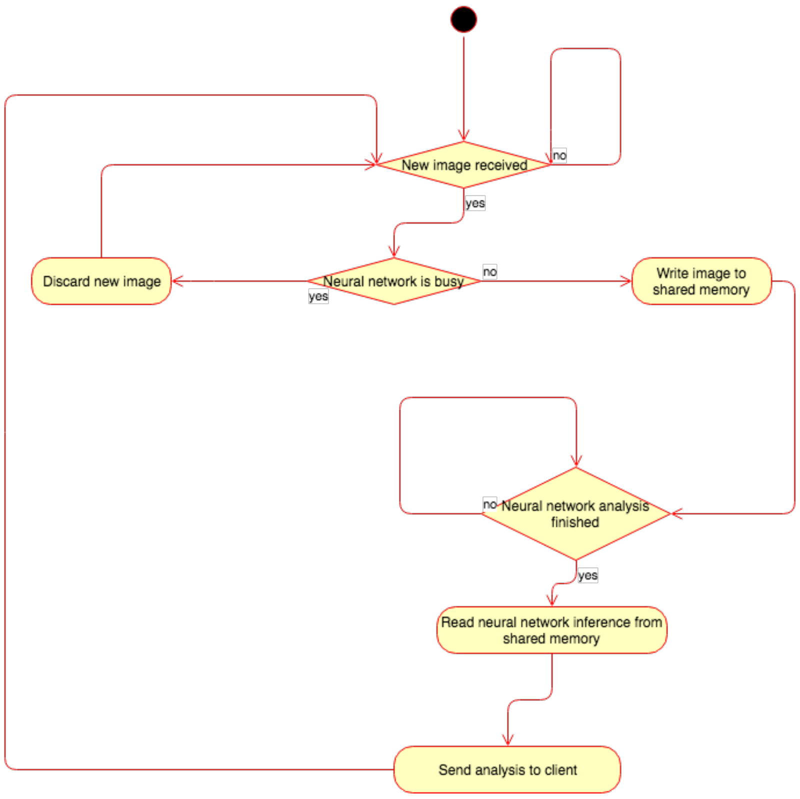

New image received
no
yes
Neural network is busy
yes
no
Discard new image
Write image to shared memory
no
Neural network analysis finished
yes
Read neural network inference from shared memory
Send analysis to client

Figure 3.12: The activity flow for the C++ component of the server.

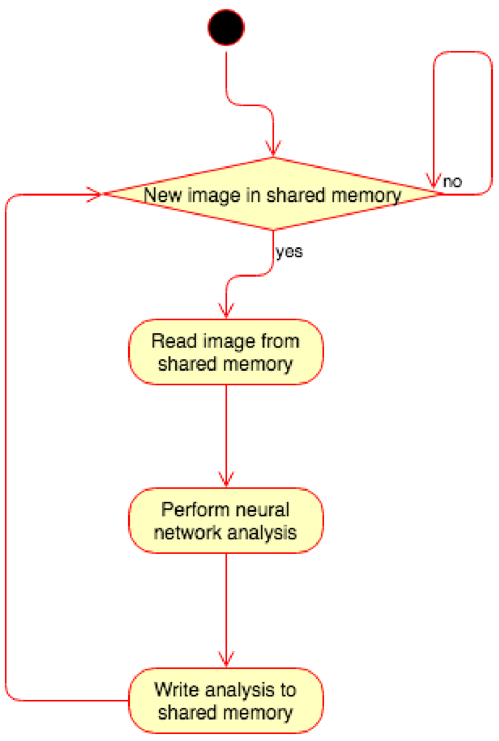

New image in shared memory
no
yes
Read image from shared memory
Perform neural network analysis
Write analysis to shared memory

Figure 3.13: The activity flow for the Python component of the server.

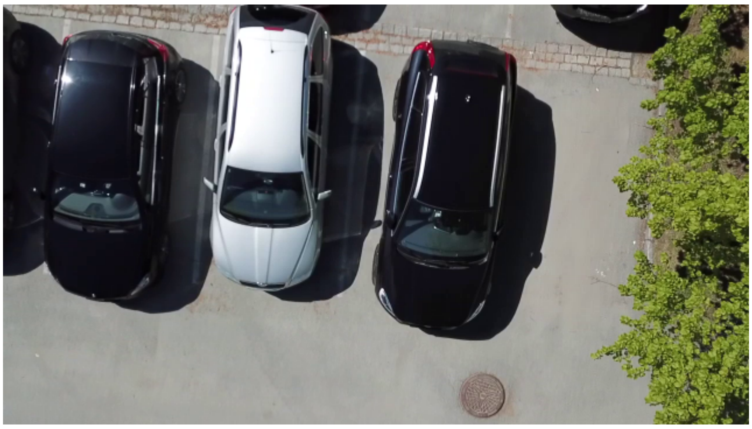


Figure 3.14: Sample of cars as seen from the quadcopter at 10 meters of altitude.

3.2 Object detection

With the infrastructure in place, the time had come to train and implement the neural networks that were going to perform the analysis on data from the aircraft. To showcase the capabilities of the system we wanted a neural network that was capable of detecting cars from a top-down aerial view. Sample images from the drone are shown in figure 3.14, 3.15 and 3.16. The aforementioned sample images show the images we need to run object detection on.

When we evaluated which object detector to utilize we considered both speed and accuracy, and after careful consideration, we found out that SSD would be the best fit for this project. SSDs fast inference time became the determining factor since the project aimed at achieving real-time performance. We deployed the model by using TensorFlows object detection API on the workstation referenced in table 3.2. We trained the model through several iterations on different datasets. We manually gathered data through various means, both aerial airplane photographies from online services, as well as data gathered during drone flight, were used in the process. The images were annotated using the LabelImg tool from section 2.5.2. The model we chose for this project was pre-trained on the Coco-dataset. Ideally, we would have utilized a model pre-trained on the KITTI-dataset, but we could not find a version of SSD in the TensorFlow object detection model zoo trained on that dataset. Training was monitored using TensorBoard. The following subsections will present how the object detector was trained during eight iterations. The results for each of the iterations will be presented in chapter 4.

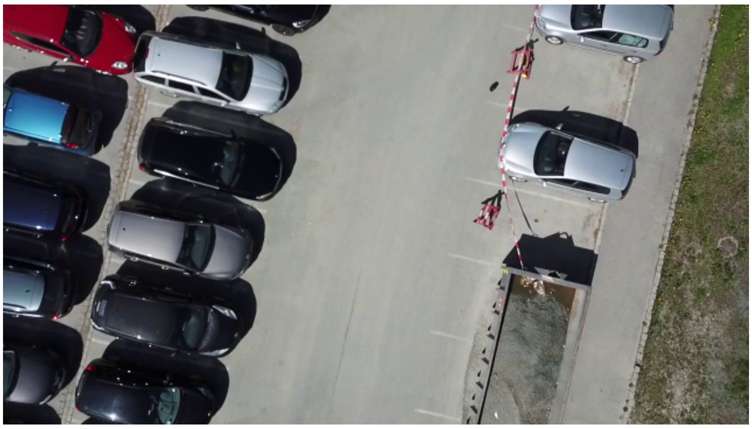


Figure 3.15: Sample of cars as seen from the quadcopter at 20 meters of altitude.

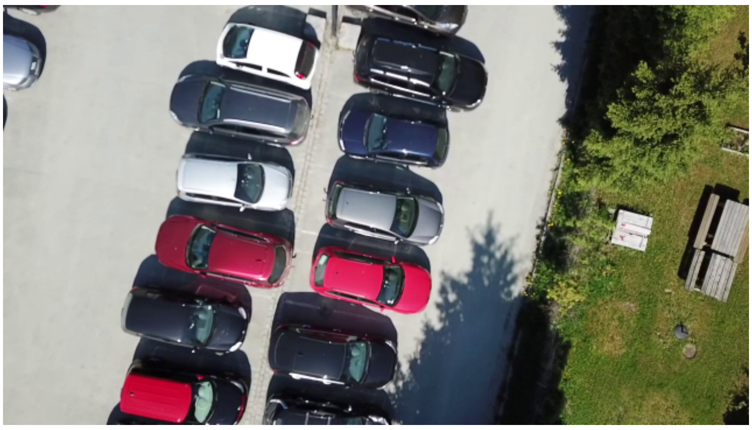


Figure 3.16: Sample of cars as seen from the quadcopter at 30 meters of altitude

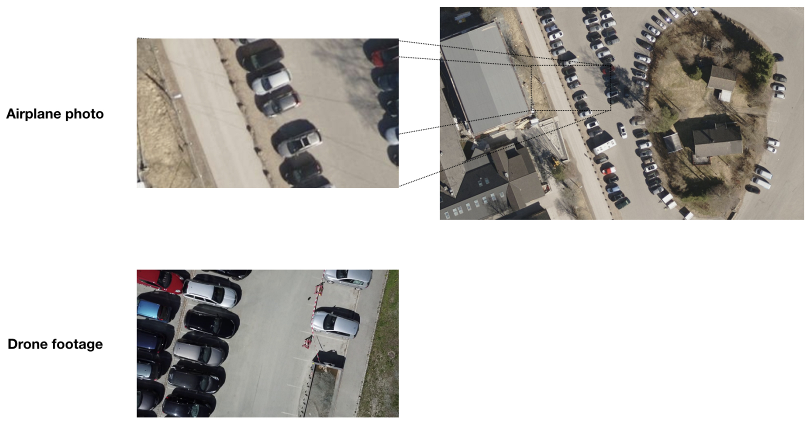


Figure 3.17: Sample airplane photo at full resolution with a subsection to illustrate how the cars look up close. Notice the similarity to the drone footage

3.2.1 1st iteration - training with airplane photos

In our first attempt at training an object detector, we used airplane photos that we scraped from `www.norgeibilder.no`. At a sufficiently high zoom level, the cars have a look that we hypothesized to be of close enough similarity to be utilized by our system. At maximum zoom, an image looks as in figure 3.17 which is not too dissimilar from the images from the drone. We scraped 500 flight originating photos of resolution 3000 $\times$ 1750. With this method, we acquired a large base of aerial view car images. It took approximately 20 hours to label all the cars in the images.

The established dataset of airplane photos ended up containing more than 4878 annotated cars. We split the data into a test set and a training set with respectively 450 training images and 50 test images, with 4398 labeled cars in the training set and 480 labeled cars in the test set.

Table 3.3: Training set 1 - unaltered airplane photos

	Training set	Test set
Size	450 images	50 images
Labeled instances	4398 labeled cars	480 labeled cars
Resolution	3000x1750	3000x1750

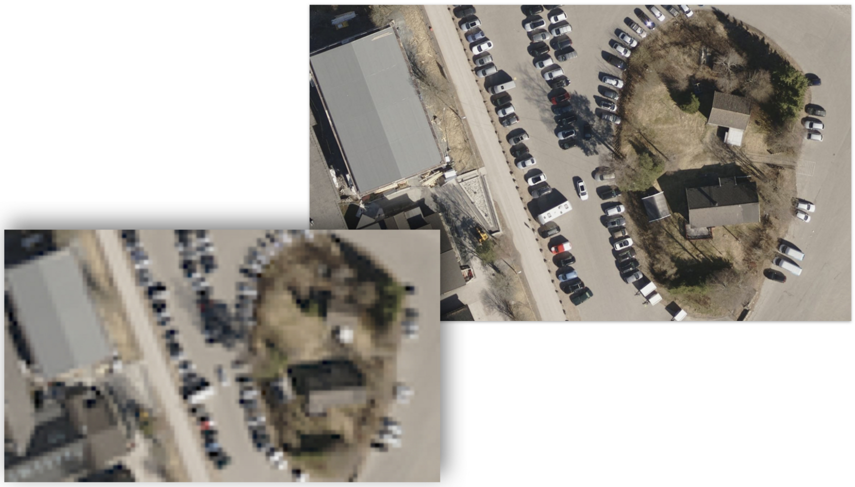


Figure 3.18: The second iteration contained down scaled airplane photos of cars.

3.2.2 2nd iteration - training with downscaled airplane photos

In the second iteration we modified the dataset. We downscaled the images from 3000×1750 to 600×350. To create new valid annotations we created a python script that read all the XML-files containing the annotations for the images in the original dataset and produced annotations for the downscaled images.

Table 3.4: Training set 2 - downscaled airplane photos

	Training set	Test set
Size	450 images	50 images
Labeled instances	4398 labeled cars	480 labeled cars
Resolution	600x350	600x350

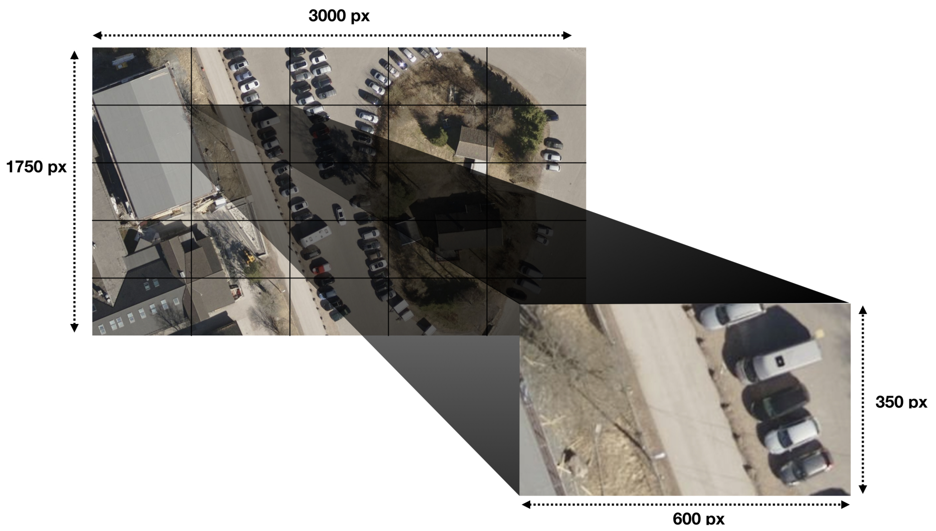


Figure 3.19: By tiling the image we increase the number of images, in addition to increasing the size of the cars.

3.2.3 3rd iteration - training with tiled airplane photos

In the third iteration of training the object detector, we altered the dataset once more. To make the cars in the dataset bigger we created a script that tiled the original 3000 × 1750 into 25 smaller images of size 600 × 350. As shown in figure 3.19. However, we did not keep all the 25 tiles of the images. We made the script only save a tiled portion of the image if a car was present. The dataset ended up with 2554 images and 5621 labeled cars, which is slightly more than the original dataset. The explanation for the presence of multiple cars is because some cars were present in multiple tiles. We created a training set with 5116 car annotations and 2325 images while the test set contained 229 images and 505 labeled cars

Table 3.5: Training set 3 - tiled airplane photos

	Training set	Test set
Size	2325 images	229 images
Labeled instances	5116 labeled cars	505 labeled cars
Resolution	600x350	600x350

3.2.4 4th iteration - training with tiled airplane photos and augmentations

In the fourth iteration of the training, we started looking at augmentations. Tensorflow object detection offers options for augmenting the training data by modifying parameters in a configuration file to make the object detector more general. The augmentations we introduced were as shown in table 3.7. We introduced scale augmentations to make the object detector more robust on images that were captured at multiple altitudes. We set a minimum scale to be 0.5 times the current image size, and a maximum scale to be 3.0 times the current image size. We furthermore wanted the object detector to more abstractly learn what a car looks like, and therefore added an augmentation that turns some images into grayscales. One of the causes for this is the fact that our training set contained many images of white and black cars and not enough of other car colors. Moreover, we perform random croppings of the images which make the object detector able to detect cars even when one only sees part of it in an image. The last augmentation we introduced was flipping of the images to increase the number of car orientations in the dataset.

Table 3.6: 4th iteration - augmentations with related hyperparameters

Random Scale	Min scale ratio: 0.5	Max scale ratio: 3.0
Random RGB to Grayscale	Probability: 50%	
Random Image Cropping		
Random Horizontal Flip	Probability: 50%	

3.2.5 5th iteration - training with tiled airplane photos and modified scale augmentations

In the fifth iteration, we made some modifications to the augmentation schema. We set the minimum scale ratio to 0.1 and the maximum scale ratio to be 4.0. We kept the other augmentations the same.

Table 3.7: 5th iteration augmentations with related hyperparameters

Random Scale	Min scale ratio: 0.1	Max scale ratio: 4.0
Random RGB to Grayscale	Probability: 50%	
Random Image Cropping		
Random Horizontal Flip	Probability: 50%	

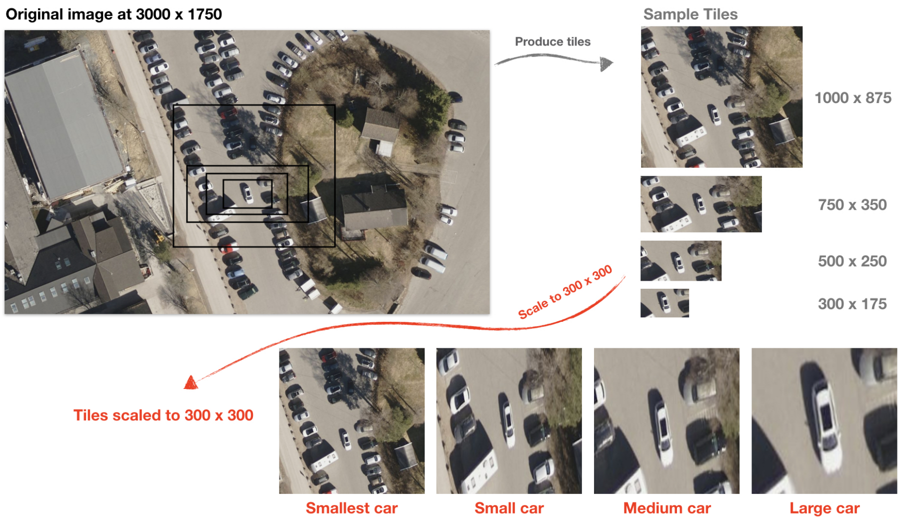


Figure 3.20: Dataset produced by tiling the original data with multiple tile sized and scaling all tiles to 300x300.

3.2.6 6th iteration - tiled multi-scale airplane dataset

In the 6th iteration we constructed another dataset. Instead of increasing the scale parameters in the training augmentation we created a dataset with images where the cars filled a varied amount of the pictures. We created a new Python script that extracted tiles from the original airplane photos at multiple sizes. The original airplane photos were of size 3000×1750. We created a tiled dataset where the sizes of the tiles were respectively $1000 \times 875, 750 \times 350, 500 \times 250$ and 300×175. When we had the tiles of various sizes, we scaled them all to the size that we eventually gave to SSD, which is 300×300. As can be seen in figure 3.20, the dataset is very diverse at this point with cars of many different sizes given to the detector at the training stage. We further kept the augmentation schema used in iteration number 4. After tiling, the dataset ended up with 15992 annotations of cars spread over 8148 images.

Table 3.8: Training set 4 - multi-scale tiled airplane photos

	Training set	Test set
Size	7334 images	814 images
Labeled instances	14346 labeled cars	1646 labeled cars
Resolution	300x300	300x300

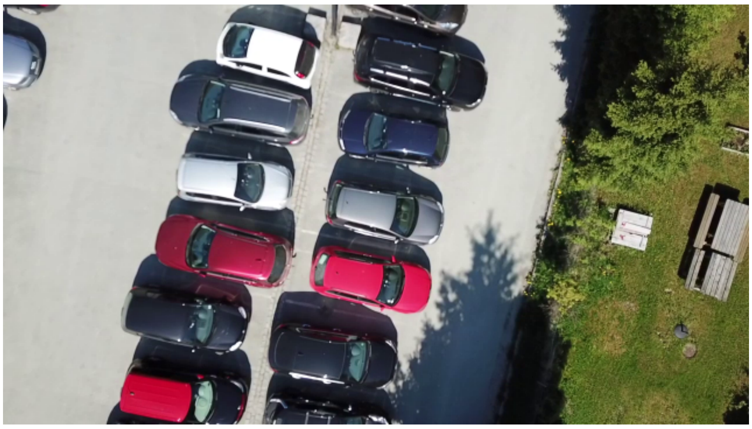


Figure 3.21: Sample drone image.

3.2.7 7th iteration - drone-based dataset

In the 7th iteration we created a dataset based on drone footage. To gather the footage we flew the quadcopter over different parking lots, at different times of the day, and at different altitudes to gather a large collection of images that we could use as a basis for car annotations. The images looked like the sample provided in figure 3.21. The first drone-based dataset contained 214 images with 1691 annotations of cars. This iteration used the same augmentations as we did in iteration 4.

Table 3.9: Training set 5 - drone-based dataset

	Training set	Test set
Size	199 images	15 images
Labeled instances	1451 labeled cars	240 labeled cars
Resolution	300x300	300x300

3.2.8 8th iteration - extended drone-based dataset

In iteration eight, we extend the drone-based dataset from the 7th iteration with more images containing very few cars, and also where the cars would not fill a significant proportion of the images. We extended the dataset by selecting a fraction of the airplane images, and manually selected scales and sections of the images that had the desired properties. This iteration also used the same augmentations as we did in iteration 4 and 7.

3.3 Image-derived geospatial data generation

This section will present the methods we used to predict geospatial locations from bounding boxes. The problem is visualized in figure 3.22.

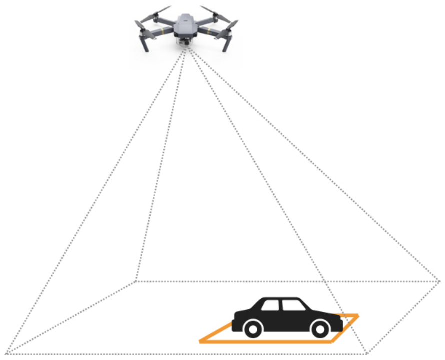


Figure 3.22: Graphical illustration of the drone and its view frustum containing objects of interest.

We have an aerial quadcopter drone flying at a certain altitude with a camera pointing straight down. The object detector creates bounding boxes around cars in the image. Figure 3.23 depicts what the drone in figure 3.22 sees. To borrow an expression from the world of computer graphics, let us call the pyramid depicting the drones vision for its view frustum. The far side of the view frustum is going to contain an image of what the quadcopter sees. One can however not map a bounding box from an image to global coordinates without some extra knowledge. The software development kit used for the quadcopter system provides us with a couple of parameters that we found interesting in order to estimate object coordinates. In real-time the drone gives us:

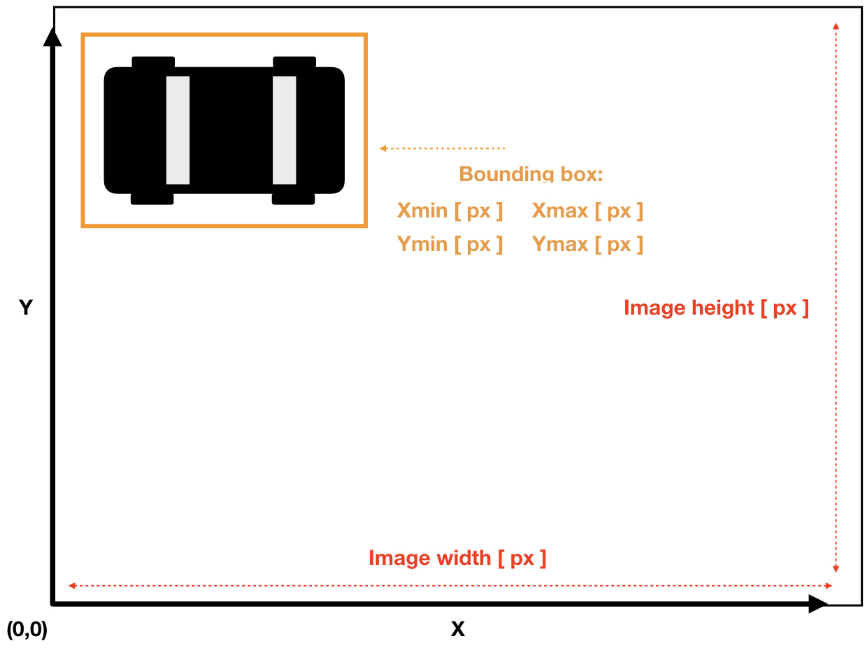


Figure 3.23: The top-down view of the drone.

1. Latitude and longitude of the drone location from its GPS.
2. The drones altitude above the ground measured by a barometric sensor. The altitude is measured relative to the point where the drone started its flight.
3. The drones heading with regard to true North, measured by the internal magnetic compass.

The simplifications we make in our approximation is that the drone is looking straight down, that the ground plane it is covering is perfectly flat and that the camera is located on the GPS-coordinate corresponding to the middle of the image.

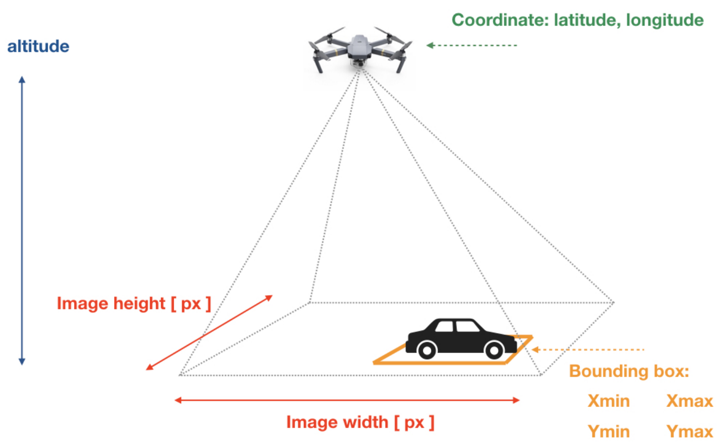


Figure 3.24: View frustum with known parameters mapped down.

When we add the image-related parameters, in addition to the parameters we have from the drones SDK the problem turns into the one shown in figure 3.24.

It provides an interesting starting point, but it is still not sufficient to approximate real-world coordinates. One can additionally extract the drones main camera field of view, both horizontal and vertical. By attaining that information, we could start approximating the width and height in meters of the view frustums far side. Later on, in this section, we will outline how we found the horizontal and vertical field of view from the drones primary camera. A simplified view of the problem is portrayed in figure 3.25.

When the view frustum of the drone is looked at from the side, then figuring out the side length of the view frustums far side turns into a trigonometric problem. By splitting the triangle in figure 3.25 in two identical right-angled triangles and denoting the field of view as fov, the problem can be stated as follows.

fov: field of view of the relevant side of the view frustum in degrees.
W: side length in meters of the far side of the view frustum corresponding to the width of the image
H: side length in meters of the far side of the view frustum corresponding to the height of the image.
Since H and W are calculated the same way, we will only present how it is done

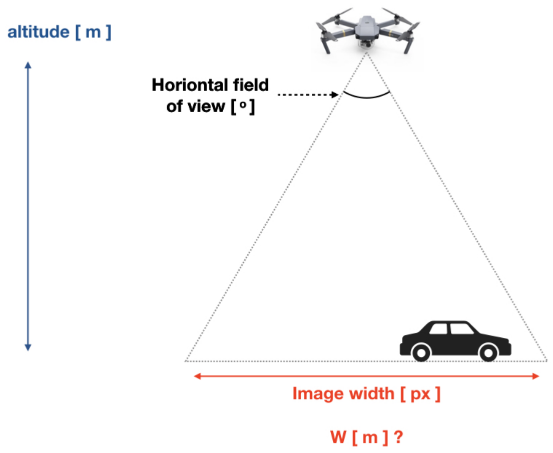


Figure 3.25: A side view illustration of the problem.

for W.
A: Altitude in meters above the ground for the quadcopter.
B: Hypotenuse side length in meters of the right-angled triangle in figure 3.26

$$tan(\frac{fov}{2}) = \frac{\frac{\frac{W}{2}}{B}}{\frac{A}{B}} \tag{3.1}$$

If one solves for W one ends up with the succeeding equation.

$$W = 2 \cdot A \cdot tan(\frac{fov}{2}) \tag{3.2}$$

W is denoting the side length in meters of the far side of the view frustum. By dividing W by the number of pixels, c_{pix}, on the corresponding side of the image, we have a ratio for meters per pixel here denoted by r.

$$r = \frac{W}{c_{pix}} \tag{3.3}$$

By using the ratio of meters per pixel, we can find the distance in meters between the cars location and the drones coordinate by using the Pythagorean theorem as shown in figure 3.27.

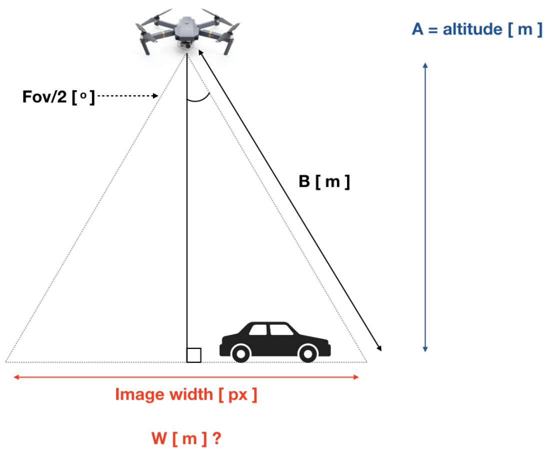


Figure 3.26: Turning the calculation into a trigonometric problem in order to find the length in meters of the wide side of the image.

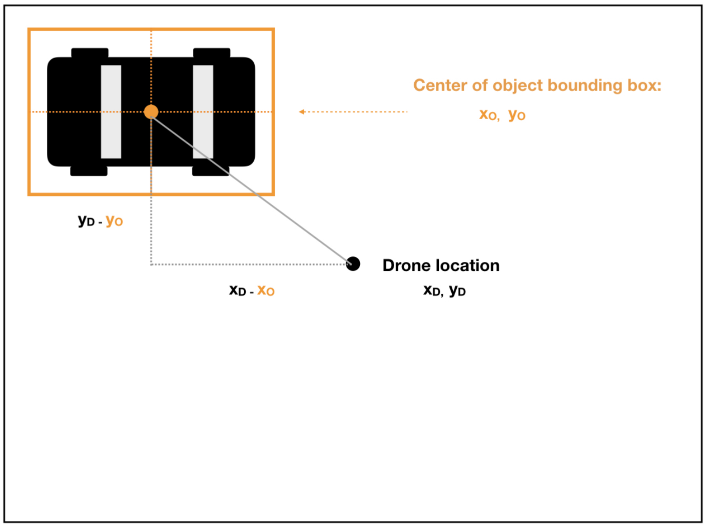


Figure 3.27: The pixelwise distance between the center of the bounding box and the camera location can be solved by using the Pythagorean theorem.

By defining the variables as follows:

x_D, y_D: pixelwise coordinates of the center of the image which is the approximation of the drones location.
x_O, y_O: pixelwise coordinates of the center of the bounding box, used as approximation for the object location.
d_{pix}: pixelwise distance between object and drone.

The pixelwise distance between the drone and the detected object can then be specified as shown in equation 3.4.

$$d_{pix} = \sqrt{(x_D - x_O)^2 + (y_D - y_O)^2} \tag{3.4}$$

By using the pixelwise distance d_{pix} from equation 3.4 and the meters per pixel approximation r from equation 3.3 we can estimate the distance in meters, d_m, between the drone and the object as shown in equation 3.5.

$$d_m = d_{pix} \cdot r \tag{3.5}$$

At this point, we have two out of three needed parameters to use Vincenty's method presented in section 2.8. To reiterate the usage of Vincenty's method, it outputs a global latitude and longitude coordinate given a starting point, a distance from that starting point, in addition to a bearing which is an angle compared to the true north. The starting point we input to the equation is the location of the drone, the distance is d_m from equation 3.5, the last parameter one needs is the bearing. Our starting point for approximating the bearing is the drones heading from the SDK. The problem is visualized in figure 3.28. Also, one has to figure out the angle, β, between the object-vector and the heading-vector as indicated in the figure. By figuring out β and adding that angle to the heading we have the true bearing in degrees of the object's location as seen from the drone.

To figure out the angle, β, one can perform some simple vector math. We can say that the heading we retrieve from the drone is in the direction of the vector pointing straight forward from (x_D, y_D). The object-vector, $\vec{v_o}$, can then be found by subtracting the drone coordinate, (x_D, y_D) from the bounding box center coordinate, (x_O, y_O), as seen below.

$$\vec{v_o} = \begin{bmatrix} x_O - x_D \\ y_O - y_D \end{bmatrix} \tag{3.6}$$

The heading vector, v_h, is simply 0 along the x-axis, and it can be set to the image height, h_i, subtracted by the drone coordinate, y_D along the y-axis as seen below.

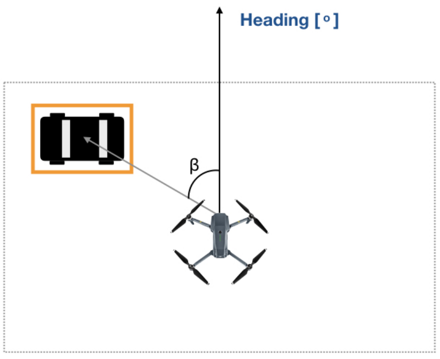


Figure 3.28: A visualization of the starting point for estimating the heading of the drone.

$$\vec{v_h} = \begin{bmatrix} 0 \\ h_i - y_D \end{bmatrix} \tag{3.7}$$

With v_h and v_o defined we can find the angle, β, by executing the formula below and solving for β.

$$cos(\beta) = \frac{\vec{v_h} \cdot \vec{v_o}}{\mid \vec{v_h} \mid \cdot \mid \vec{v_o} \mid} \tag{3.8}$$

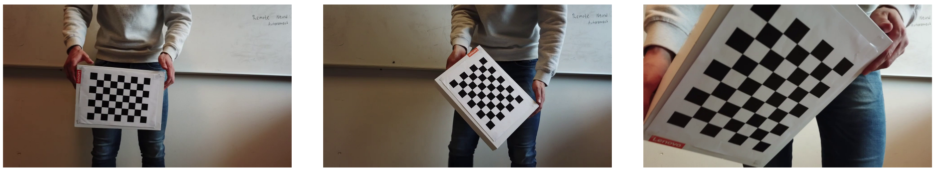
If we solve for β we get the following expression.

$$\beta = arccos(\frac{\vec{v_h} \cdot \vec{v_o}}{\mid \vec{v_h} \mid \cdot \mid \vec{v_o} \mid}) \tag{3.9}$$

The bearing that will be used as input to Vincenty's method, γ, will then be the drones heading, α, added to the angle β as seen in equation 3.10.

$$\gamma = \alpha + \beta \tag{3.10}$$

With all the relevant parameters in place, we can use them as input to Vincenty's direct method, which will output approximated global coordinate locations for objects detected during drone mission execution.



(a) Sample 1 (b) Sample 2 (c) Sample 3

Figure 3.29: Sample of the calibration images we used to find the intrinsic camera parameters.

3.3.1 Extracting intrinsic camera parameters

One of the crucial elements needed for the approximation of global coordinates to be possible is the cameras field of view. If one looks inside the user manual of the DJI Mavick Pro one can find out that the drones horizontal field of view is 78.8^o. The user manual does nevertheless not give the vertical field of view. Moreover, the field of view given in the manual assumes that the camera uses full resolution at 4000×3000 pixels. The resolution of the video that is streaming from the drone to the server is at 720p or 1280×720 pixels. Section 2.11 describes a technique for finding the cameras intrinsic parameters. We executed the calibration procedure by utilizing OpenCV. We took 30 images of a calibration target, a chessboard, of known size at various angles and poses as shown in the figure below. Using this calibration procedure we acquired both the vertical and horizontal field of view of the camera at 1280×720 resolution. We ended up with $fov_x = 64.900347^o$ and $fov_y = 39.381454^o$. The x-axis goes horizontally along the image, and the y-axis goes vertically along the image.

3.4 Evaluation

To evaluate our methods we did a number of experiments. The experiments can be divided into five separate parts:

1. Autonomous flight experiments to test the drones flight abilities.
2. Object detection experiments to verify the robustness of the object detectors created in the different iterations.
3. Geolocation prediction experiments to test the accuracy of the geolocation predictions.
4. Inference delay experiments to test the delay of the server side predictions.

5. Autonomous surveillance and inspection experiments to test the entire system.

We will briefly lay out how we executed the different experiments in the following subsections.

3.4.1 Autonomous flight experiments

To test the autonomous flight abilities of the drone we executed four experiments in total. The experiments we performed were a simulated waypoint-mission, a simulated area cover mission in addition to a real-world waypoint-mission and a real-world area cover mission. The purpose of the simulated experiments are to ensure that the drone has correct behavior before taking the aircraft into the field. The benefit of using a simulator for the experiments is that the risk is minimal and iterations can be performed rapidly.

3.4.2 Object detection experiments

The object detectors were quantitatively evaluated by using them on a validation set constructed from drone images taken at multiple locations, at several altitudes and in different lighting conditions. We furthermore qualitatively evaluated the object detectors by empirically observing their performance on a drone video.

3.4.3 Geolocation prediction experiments

The geolocation prediction experiments were performed by manually flying the drone over areas containing cars. The locations of the cars were marked by using the GPS of the smartphone while the phone was held over relevant cars. The smartphone annotated car locations would afterwards be compared to the geolocation predictions produced by the system to see how big the discrepancies between the two were. The distance between the predicted points and the annotated point is calculated using Vincety's indirect method, which was referenced in section 2.8.

3.4.4 Inference delay experiments

To assess the time it takes to perform inference from the client we need to test the round-trip delay. The round-trip delay is measured from the image data is sent from the client to the server, until the predictions are received. We use built in functionality of the client to execute these tests. The experiments will be performed on both Wi-Fi network connections as well as 4G network connections. The experiments are set up to use one TCP-connection, with connection setup and connection tear-down for every prediction.

3.4.5 Autonomous surveillance and inspection experiments

The final experiments are meant to test the entire system for use in an autonomous surveillance context. Two experiments are executed, the first experiment use the waypoint-mission as foundation, and the second experiment use the area cover mission as foundation. During the execution of the autonomous missions the drone performs geolocation predictions. In the experiments we start by annotating relevant car locations by using a smartphone. In the waypoint-based mission, the drone is instructed to fly over the relevant cars and perform predictions. In the area cover based surveillance mission, we mark an area for the drone to cover, the drone will periodically predict object locations as it flies. The predicted locations will at the end be compared to the smartphone annotated locations. Only predictions with more than 90 percent confidence would be displayed. The experiment altitude was set to 25 meters for both kinds of experiments.

Chapter 4

Results

This chapter presents the results obtained from the various stages of the project. We will start by presenting the autonomous flight experiments conducted with the drone, after which we will present the object detectors' performance through different iterations. The following section will lay out the experiments to evaluate the precision of the image-derived geospatial data predictions. The last parts of this chapter will demonstrate a quantitative analysis of server prediction delay, in addition to a full surveillance experiment where the entire system is tested.

4.1 Autonomous drone flight experiments

The performance of the drones autonomous flight abilities will be laid out in the following sections.

4.1.1 Simulated waypoint-mission

The first experiment was aimed at executing a waypoint-mission in the simulator. In the experiment, we marked two waypoints inside the client interface. The altitude was set to 20 meters and the speed was set to the range 8 to 10 $\frac{m}{s}$. The drone was instructed to return to the home point after visiting the two waypoints. The total mission execution time was 1 min and 55 s. The status of the drone and its behavior was carefully monitored during the flight as seen in figure 4.1. The drone visited both waypoints with expected actions, and the experiment was a success. A video that that documents the experiment is available at `https://www.youtube.com/watch?v=qoVBbKcjU4Y`.

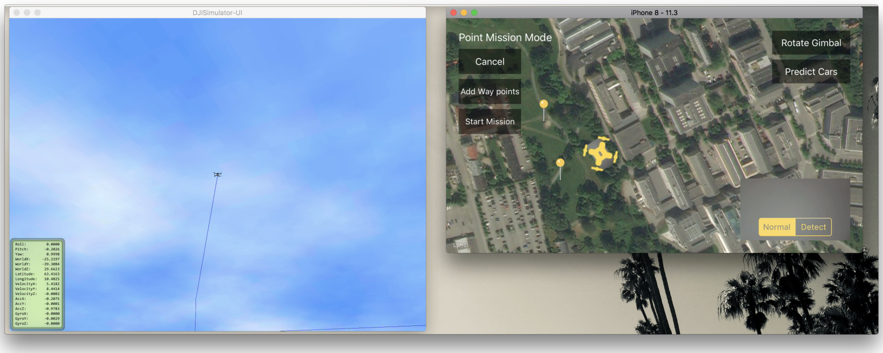


Figure 4.1: A screen shot of the simulator and the application in action during the simulated way waypoint-mission.

4.1.2 Simulated area cover mission

The second experiment was conducted to verify that the drone was covering the area marked in the mobile interface. During the experiment, we set the altitude to 20 meters with speed in the range between 8 to 10 $\frac{m}{h}$. The drone was furthermore instructed to return to the home point after finishing the mission. The total mission execution time was 4 min and 30 s. It covered the area it was instructed to cover, and the mission was a success. A video documenting this experiment is available at `https://www.youtube.com/watch?v=71cNrbGE41Y`.

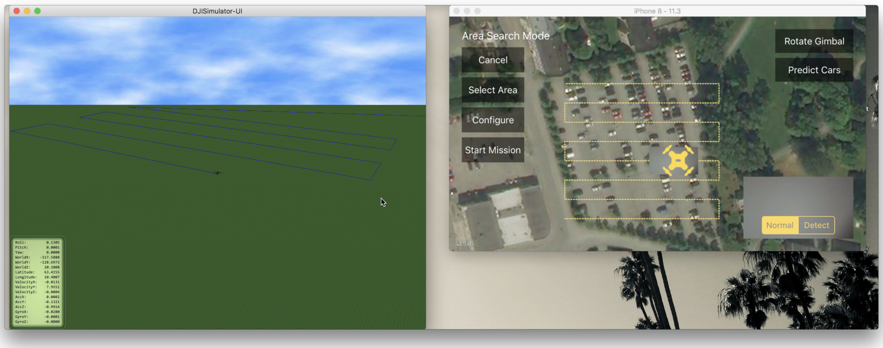


Figure 4.2: Screen shot from the execution of the simulated area cover mission.

4.1.3 Real-world waypoint-mission

The third experiment conducted had the goal of ensuring the quadcopter performed the waypoint-mission as instructed out in the field. The drone was brought to an open park area near the Kristiansten Fortress in Trondheim. The area was chosen as it is ideal for performing experiments with the quadcopter due to the low density of tall objects that can interfere with drone flights. The mission was initiated in a fashion similar to the one performed in the simulator. Two waypoints were selected, as shown in figure 4.3, and the drone was observed during the mission. The experiment set the altitude to 20 meters at speed in the range between 8 to 10 $\frac{m}{h}$. The total mission execution time was 1 min and 33 s. The quadcopter performed all the desired actions, and the real-world waypoint-mission was a success.

4.1.4 Real-world area cover mission

The last flight control experiment we conducted was the real-world area cover mission. The purpose of the mission was to assess the area coverage performed by the drone. This experiment was too conducted at the Kristiansten Fortress. An area was marked in the graphical interface as seen in figure 4.4. During the experiment, we set the mission altitude to be 20 meters at speed in the range between 8 to 10 $\frac{m}{h}$. The total mission execution time was 4 min and 31s. The drone covered the desired area with designated actions and accomplished its goals perfectly.

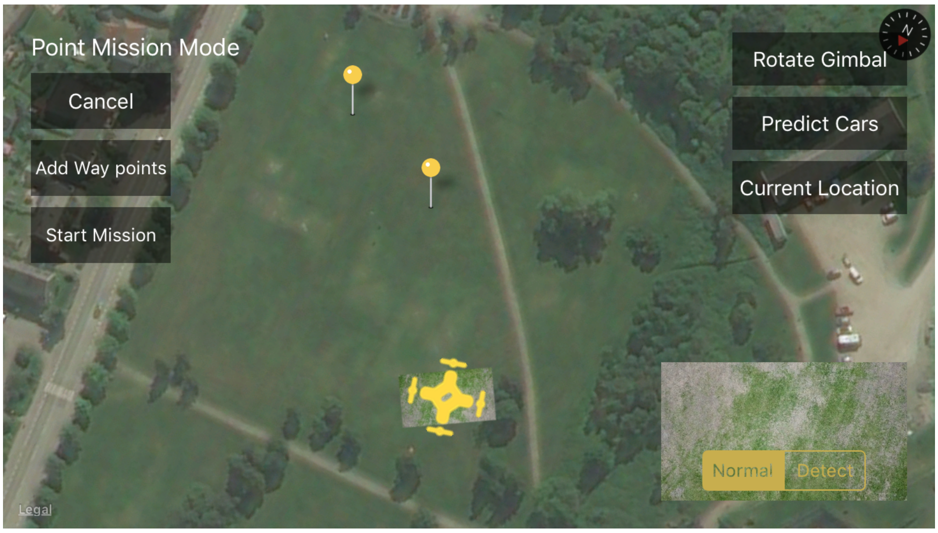


Figure 4.3: Screen shot from the execution of the waypoint-mission out in the field.

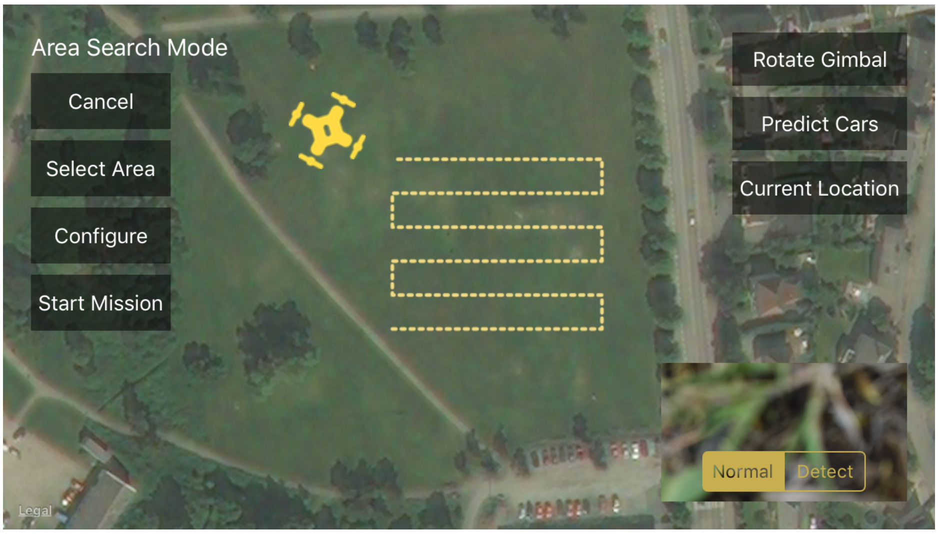


Figure 4.4: Screen shot from the execution of the area cover mission out in the field.

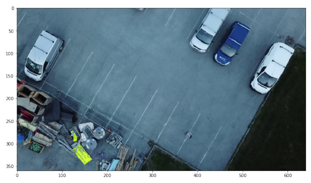
4.2 Object detection network performance

This section will lay out the results obtained from the iterations performed after training the object detectors. To reiterate what we described in chapter 3, we trained multiple version of SSD that were pre-trained on the Coco dataset. The mAPs after running the object detectors on the validation set is provided in table 4.1. Images illustrating the object detectors' prediction abilities are shown on the next pages.

Table 4.1: Results after running the object detectors on the validation set.

Object detector	**mAP with 0.5 IoU**
1 - trained with airplane photos	0
2 - trained with downscaled airplane photos	0.000013
3 - trained with tiled airplane photos	0.277539
4 - trained with tiled airplane photos and augmentations	0.334974
5 - trained with tiled airplane photos and modified scale augmentations	0.271160
6 - trained on multi-scale airplane dataset	0.333887
7 - drone-based dataset	0.845358
8 - extended drone-based dataset	0.839170

(a) Iteration 2: sample 1


(b) Iteration 2: sample 2

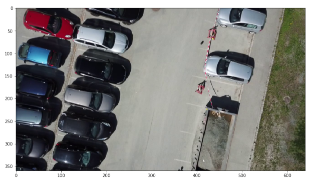
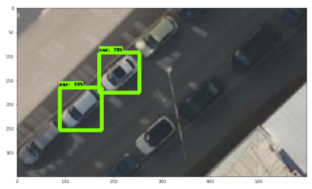
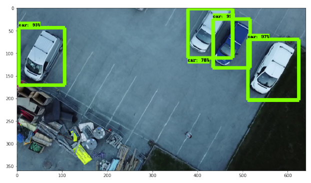

(c) Iteration 2: sample 3

Figure 4.5: Sample predictions from the object detector produced in the 2nd iteration


(a) Iteration 3: sample 1


(b) Iteration 3: sample 2

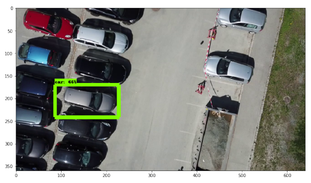
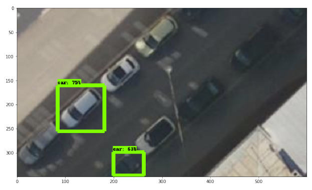
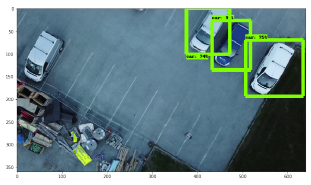

(c) Iteration 3: sample 3

Figure 4.6: Sample predictions from the object detector produced in the 3rd iteration


(a) Iteration 4: sample 1


(b) Iteration 4: sample 2

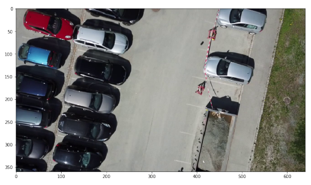
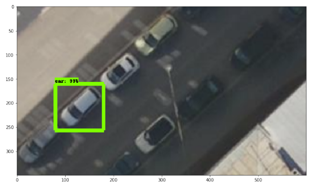
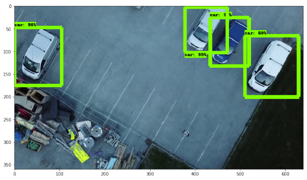

(c) Iteration 4: sample 3

Figure 4.7: Sample predictions from the object detector produced in the 4th iteration


(a) Iteration 5: sample 1


(b) Iteration 5: sample 2

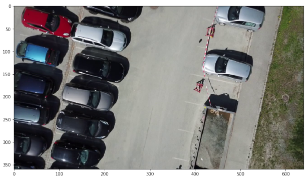
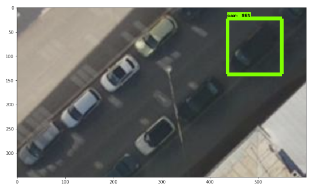
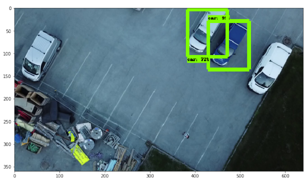

(c) Iteration 5: sample 3

Figure 4.8: Sample predictions from the object detector produced in the 5th iteration


(a) Iteration 6: sample 1


(b) Iteration 6: sample 2

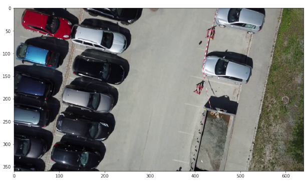
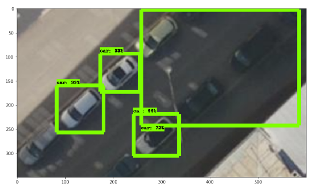
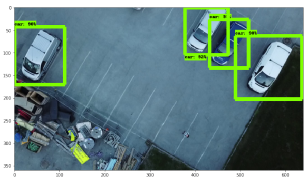

(c) Iteration 6: sample 3

Figure 4.9: Sample predictions from the object detector produced in the 6th iteration


(a) Iteration 7: sample 1


(b) Iteration 7: sample 2

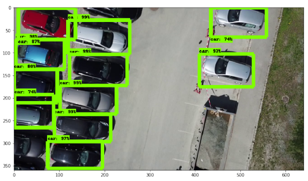
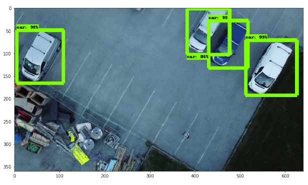

(c) Iteration 7: sample 3

Figure 4.10: Sample predictions from the object detector produced in the 7th iteration

(a) Iteration 8: sample 1


(b) Iteration 8: sample 2

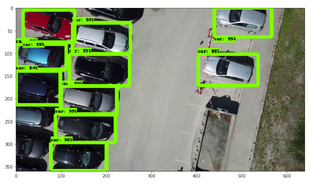

(c) Iteration 8: sample 3

Figure 4.11: Sample predictions from the object detector produced in the 8th iteration

4.3 Image-based geospatial data generation

In this section, we will present the obtained findings after performing experiments regarding geospatial data generation. To make it easier to assess the experiments we projected the bird's eye view image of the quadcopter down on the map view as seen in figure 4.12. The technique takes the drones altitude, heading, geolocation, and camera field of view into account to construct an overlay. The overlay changes size depending upon the altitude of the drone, and it changes rotation depending upon the drones heading. It takes much computational time for an old smartphone to perform these projections, so they only update at about 5-10 fps in our application. It does, however, make it very easy to verify and understand why the drone predicts geolocations of objects. The overlays are displayed as soon as the quadcopter starts flying.

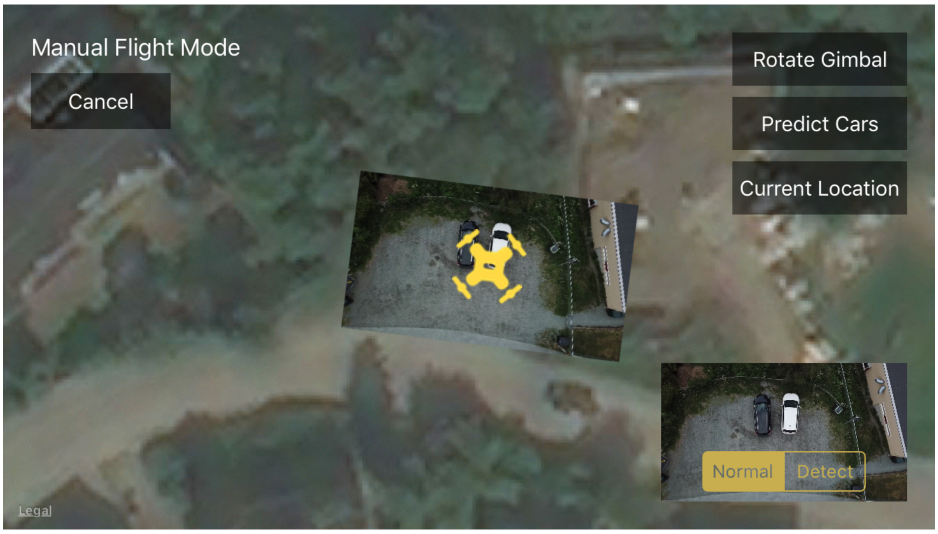


Figure 4.12: Example of overlay projection used to indicate what the drone sees.

4.3.1 Geolocation prediction - experiment 1

The first experiment we conducted to test the accuracy of the geospatial predictions aimed at detecting one car at a parking lot. The results of the experiment can be seen in table 4.2.

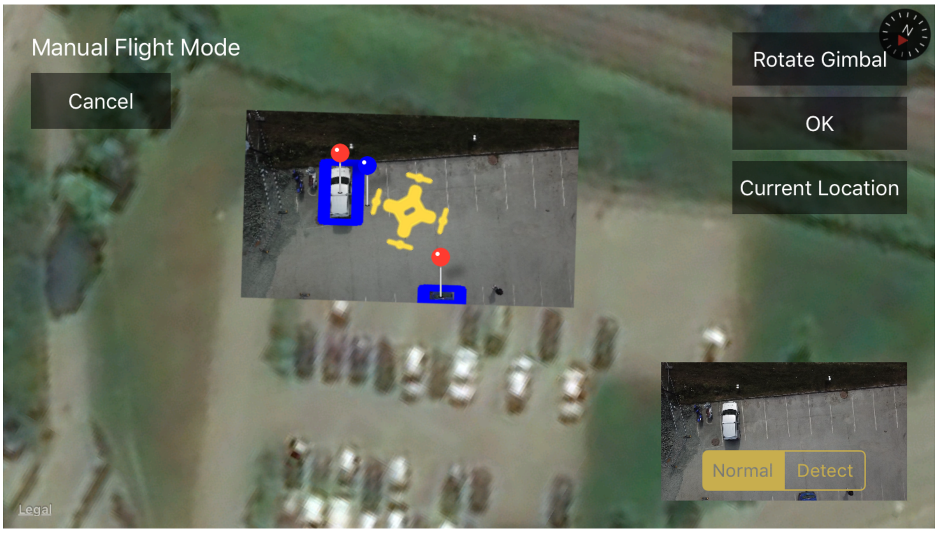


Figure 4.13: Predicted car locations shown as red pins next to annotations shown as blue pins.

Table 4.2: Results from experiment 1

	True Location	**Predicted Location**	**Difference**	**Distance**
Car 1	lat: 63.414284 lng: 10.410909	lat: 63.414275 lng: 10.410897	lat: 0.000009 lng: 0.000012	1.169 m
Car 2	Unknown	lat: 63.414308 lng: 10.410629		

4.3.2 Geolocation prediction - experiment 2

The second experiment we conducted aimed at detecting four cars at a parking lot. The results of this experiment can be seen in table 4.3.

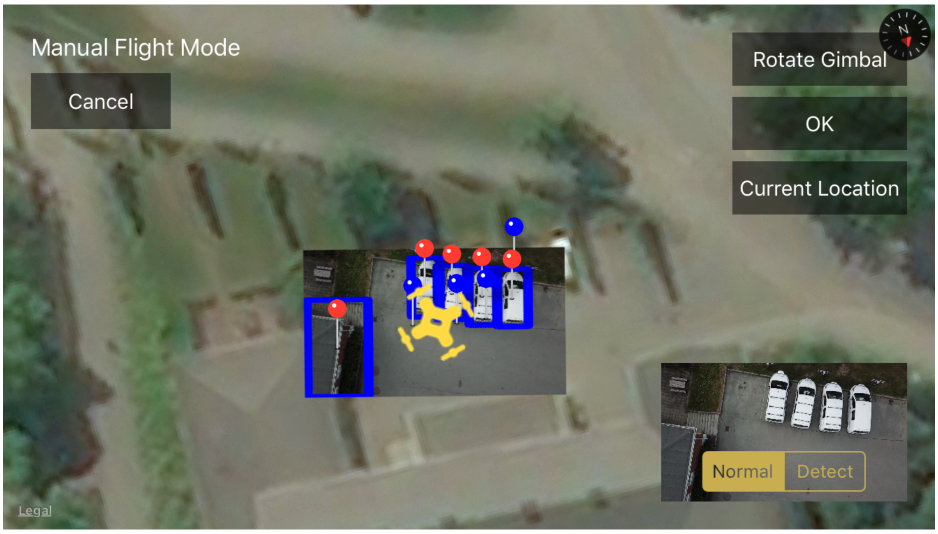


Figure 4.14: Predicted car locations shown as red pins next to annotations shown as blue pins.

Table 4.3: Results from experiment 2

	True Locations	**Predicted location**	**Difference**	**Distance**
Car 1	lat: 63.414866 lng: 10.410482	lat: 63.414869 lng: 10.410456	lat: -0.000003 lng: 0.000026	1.341 m
Car 2	lat: 63.414864 lng: 10.410443	lat: 63.414880 lng: 10.410409	lat: -0.000016 lng: 0.000034	2.463 m
Car 3	lat: 63.414864 lng: 10.410394	lat: 63.414890 lng: 10.410355	lat: -0.000026 lng: 0.000039	3.492 m
Car 4	lat: 63.414882 lng: 10.410370	lat: 63.414899 lng: 10.410301	lat: -0,000017 lng: 0.000069	3.933 m
Car 5	NO CAR	lat: 63.414897 lng: 10.410650		

4.4 Inference delay experiments

In this section we will present the round-trip delay for the inferences sent from the drone client.

A sample of round-trip delay times when the client is connected to Wi-Fi is illustrated in figure 4.16. As one can see the total delay is around 40 milliseconds. A sample of the round-trip delay time when the client is connected to 4G is shown in figure 4.15. The delay is around 400 milliseconds in this case, which is a significant change. The statistics in table 4.4 reveal that the difference between using Wi-Fi and 4G is an order of magnitude.

The bandwidth usage is quite reliable. Every image that is sent from the client to the server is of size 300 × 300 with an element size of 3 bytes per pixel. The image dimensions cause every image to be 270,000 bytes. There are some control parameters sent as well, which makes every prediction call to the server take 270,016 bytes in size. When the server is sending the predictions back to the client, analysis of the server logs reveal that the prediction information takes up sizes in the range 8 - 82 bytes depending upon how many bounding-boxes it produces.

Table 4.4: Statistics from the round-trip delay time

	4G	**Wi-Fi**
Mean round-trip time	0.27002785 s	0.0387898 s
Median round-trip time	0.2678265s	0.0381515 s
Max round-trip time	0.351848 s	0.054187 s
Minimum round-trip time	0.191009 s	0.0032968 s

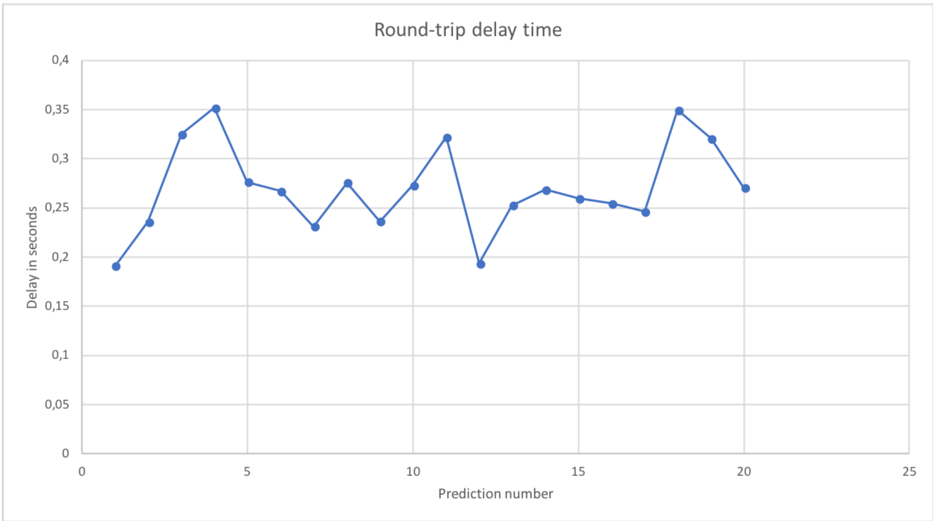


Figure 4.15: Round-trip delay when the system is connected to 4G.

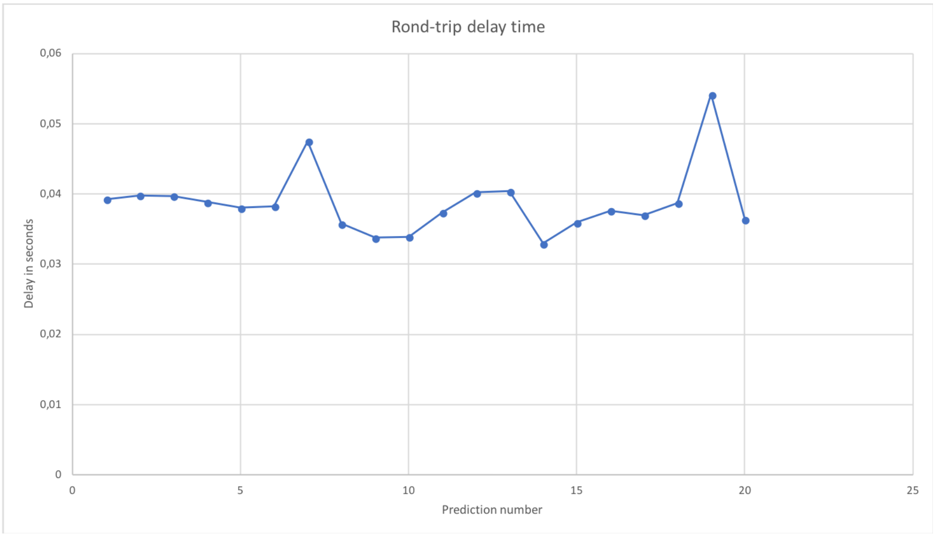


Figure 4.16: Round-trip delay when the system is connected to Wi-Fi.

4.5 Autonomous surveillance and inspection

The goal of the following experiments are meant to assess the drones surveillance and inspection abilities. We will first present the results from the waypoint-based inspection mission, after which we present the area cover surveillance mission. The following experiments utilizes all the methods developed in this thesis.

4.5.1 Waypoint-based inspection mission

The inspection mission was executed on a parking lot near Kristiansten Festning in Trondheim. We sat sat two waypoints in the application as shown in figure 4.18. Then we initiated the mission. When the missions were finished, we retrieved the predictions seen in figure 4.18. As can be seen in figure 4.19, the object detector missed two of the cars present. The results from the experiment can be seen in table 4.5.

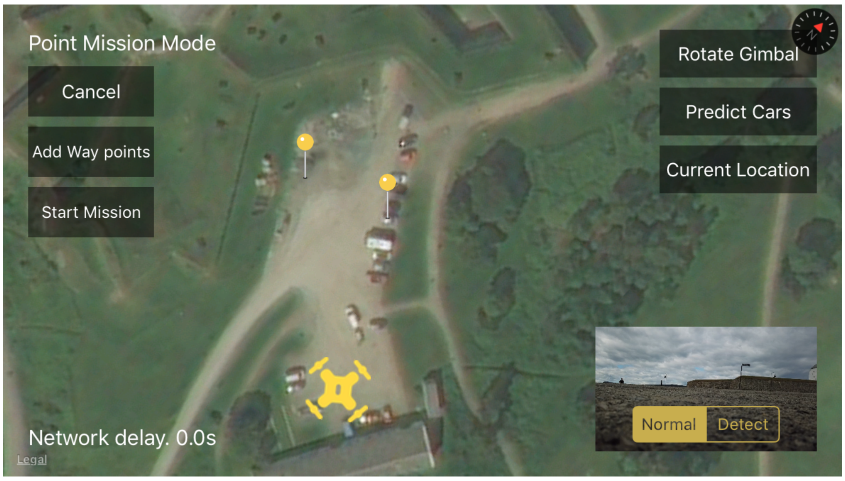


Figure 4.17: The way points the drone was visiting during its mission.

Table 4.5: The results from the geolocation predictions

	True Locations	Predicted Locations	Difference	Distance
Car 1	lat: 63.426663 lng: 10.411817	lat: 63.426727 lng: 10.411780	lat: -0.000064 lng: 0.000037	7.3 m
Car 2	lat: 63.426682 lng: 10.412416	lat: 63.426659 lng: 10.412407	lat: 0.000023 lng:0.000009	2.6 m

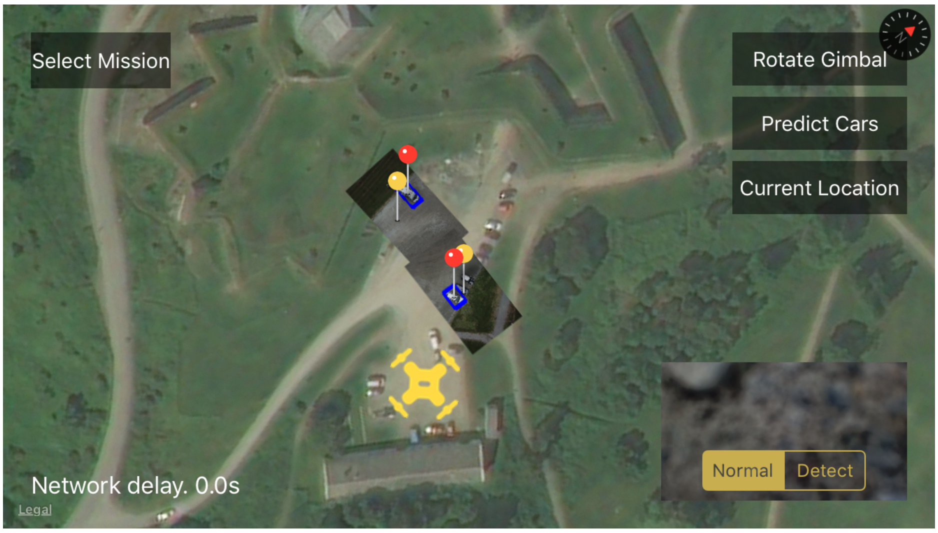


Figure 4.18: The predicted locations of the cars on top of the projected overlay. The predicted locations are given with red pins.

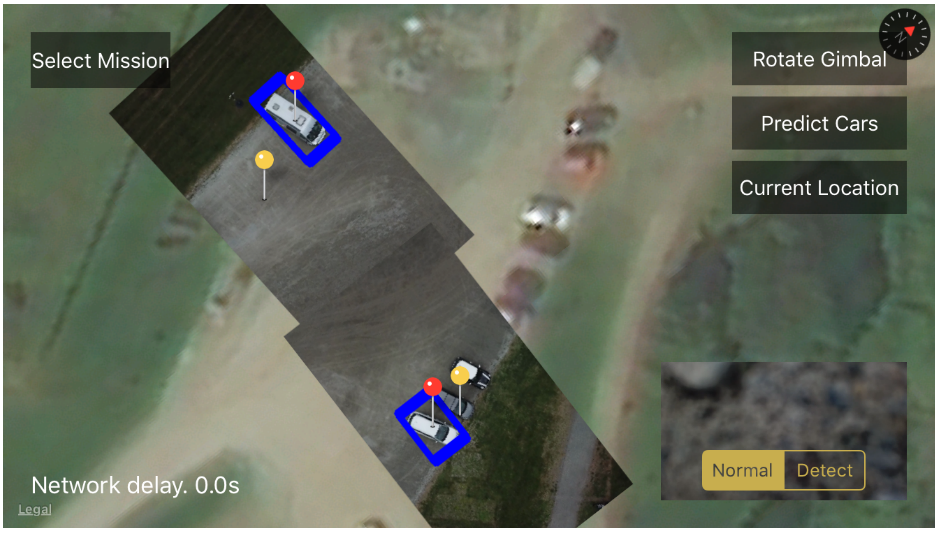


Figure 4.19: Zoomed in on the predictions of the waypoint-mission.

4.5.2 Area cover geolocation prediction

The area cover mission was executed at a parking lot near Realfagsbygget at NTNU in Trondheim. The area marked in figure 4.20 denotes the area we used for the area cover mission. When the mission was finished, we retrieved the predictions seen in figure 4.21. The drone visually covered the same area more than once in some cases which then ended up producing several predictions for the same car. Table 4.6 denotes the results of this experiment.

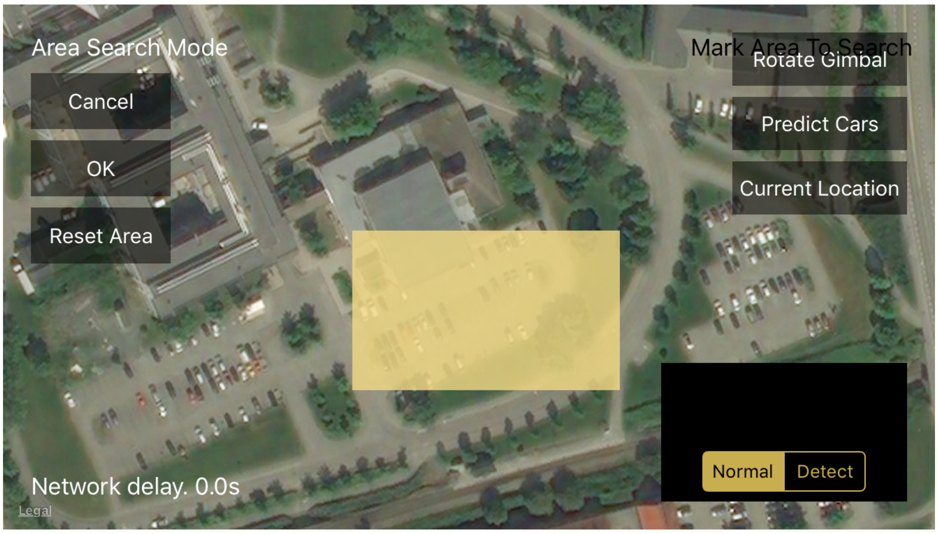


Figure 4.20: The area marked for the mission.

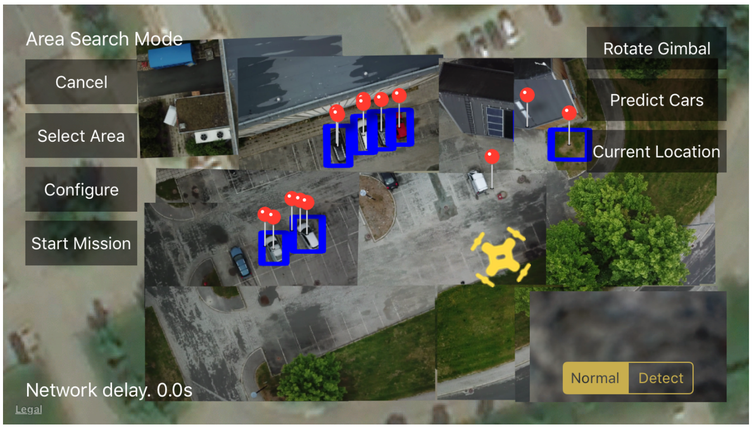


Figure 4.21: The predictions from the area cover mission.

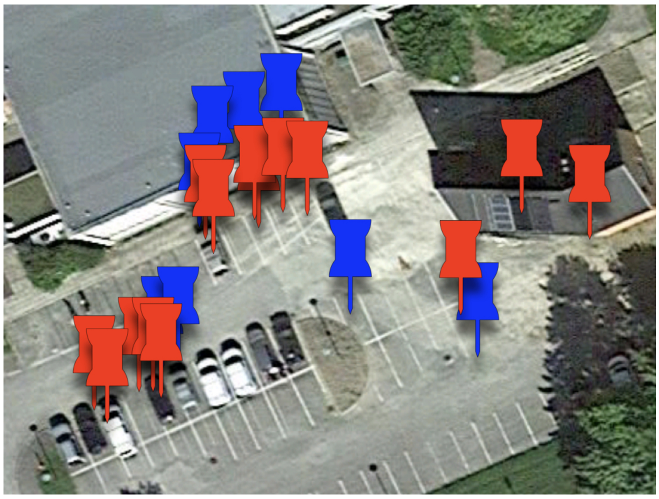


Figure 4.22: Predicted car locations showed as red pins, while the ones annotated using the iPhone is displayed in blue.

Table 4.6: Results from the area cover surveillance experiment

	True Locations	**Predicted Locations**	**Difference**	**Distance**
Car 1	lat: 63.414441 lng: 10.408678	lat: 63.414383 lng: 10.408753	lat: 0.000058 lng: -0.000075	7.4 m
Car 2	lat: 63.414426 lng: 10.408628	lat: 63.414378 lng: 10.408697	lat: 0.000048 lng: -0.000069	6.3 m
Car 3	lat: 63.414422 lng: 10.408585	lat: 63.414378 lng: 10.408641	lat: 0.000044 lng: -0.000056	5.6 m
		lat: 63.414372 lng:10.408641	lat: 0.00005 lng: -0.000056	6.2 m
Car 4	lat: 63.414383 lng: 10.408540	lat: 63.414361 lng: 10.408558	lat: 0.000022 lng: -0.000018	2.6 m
		lat: 63.414357 lng: 10.408561	lat: 0.000026 lng: -0.000021	3 m
Car 5	lat: 63.414249 lng: 10.408487	lat: 63.414240 lng: 10.408418	lat: 0.000009 lng :0.000069	2.6 m
		lat: 63.414237 lng: 10.408444	lat: 0.000012 lng: 0.000043	3.5 m
		lat: 63.414234 lng: 10.408466	lat: 0.000015 lng:0.000021	1.9 m
Car 6	lat: 63.414256 lng: 10.408458	lat: 63.414218 lng: 10.408335	lat: 0.000038 lng: 0.000123	7.4 m
		lat: 63.414214 lng: 10.408361	lat: 0.000042 lng:0.000097	6.7 m
Car 7	lat: 63.414296 lng: 10.408808	No prediction		
Car 8	lat: 63.414261 lng: 10.409066	lat: 63.414298 lng: 10.409042	lat: -0.000037 lng: 0.000024	4.3 m

Chapter 5

Discussion

In this chapter, we will go through results from the previous chapter and examine if the different methods applied were successful or not with regards to the research questions posed in the introduction of the thesis.

5.1 Commercial quadcopter drones for autonomous flight missions

In this section, we will handle the methods from chapter 4 that helps answer the first research question, **RQ1**: is it possible to use an affordable, commercial drone for autonomous flight?

The experiments performed with regards to autonomous flight control were centered around waypoint-missions and area cover missions. The missions were tested in both simulated and real-world environments. One of the interesting observations made through work on the thesis was how big the difference could be between simulated experiments and real-world experiments. The simulator provides all the necessary parameters such as altitude, location and heading for the drone through the SDK when they are needed. That was not always the case when we performed experiments in the field. For instance, during windy or cloudy conditions we experienced that the images streamed from the drone were highly unreliable and often would be received as a mixture of pixels in green and pink in the client video view. The GPS signal of the drone is not always perfect either, and if the drone is located behind tall objects such as buildings or trees, the GPS may be highly undependable, and in some cases even unreachable.

Although there are few objective metrics to measure the experimental results regarding the flight control, we can see from the empirical analysis that the drone performs desired actions for the autonomous missions and as such that the first research question is affirmed. By monitoring the map view during mission execution, we could see that the drone visits the desired locations in the desired order. The display on the remote controller of the drone indicated the current state of the aircraft such as altitude. The drone easily executes both the area coverage missions and the waypoint-missions under the right conditions. Drone operation is not always going to be possible, however since the drone applied in this thesis had some limitations. The DJI Mavick Pro requires temperatures above 0^o Celcius, winds has to be below 10 $\frac{m}{s}$, and there can be no fog, rain or snow. The conditions mentioned are however common throughout the Northern hemisphere for large parts of the year. The battery furthermore puts a hard restriction on the distance that can be covered by the drone during mission execution. The flight manual of the DJI Mavick Pro says that the maximum battery time is 27 minutes, yet practical usage has revealed that it often lower depending upon temperature.

5.2 Infrastructure for mobile real-time analysis utilizing neural networks

This section will cover techniques applied to answer **RQ2**: what infrastructure is needed to enable real-time autonomy?

The platform we used to control, monitor and communicate with the drone was focused on the mobile operating system iOS. Through the iOS-platform, we created a client application with a graphical user interface used to interact with the quadcopter, DJI Mavick Pro. Using a mobile platform with a requirement for real-time object detection capabilities is a tough challenge. The smartphones running iOS can run neural networks through SDKs provided by apple such as CoreML. However, the platform falls short when the task demands object detection with high frame rates and good accuracy. As such, we handed the heavy lifting involved in neural network calculations over to a powerful workstation containing a high-performance GPU. The new challenge was communication between the iOS-client and the server. Through the experiments we conducted we measured a round-trip delay between 0.2-0.5 seconds when connected to a 4G network. The 4G network is the main network used when experiments are performed in the field as there in most cases are no Wi-Fi available. When the client is connected to a Wi-Fi network, the frame rate improves somewhat.

Using an interplay between C++ and Python on the server for object detection analysis gave great results regarding speed. By using C++, we developed fast socket communication utilizing memory buffers and shared the networking code between the client and the server. Furthermore, Python is the language of data science and machine learning. By making sure that our system could utilize it, we took advantage of the very popular object detection framework contained in Tensorflow. To make the interaction between the two different components of the server communicate as quickly as possible, we utilized shared memory. Communication via shared memory for two processes written in two quite different languages is difficult, but it gave a great performance. Since we modeled the interplay between the two components using a state diagram, we created a highly reliable communication paradigm.

Several techniques would be interesting to test out in further detail with regards to the performance of the networking. The network protocol best suited for real-time communication is not TCP, it is UDP. UDP is, however, a lot more challenging to work with as it provides few of the reliability guarantees that TCP gives. Moreover, it would have been interesting to test out the usage of multiple TCP-connections and to divide the image data over each TCP-connection running on separate threads. Developing systems to take advantage of the networking capabilities does increase the complexity of the development by quite a bit. If it can be avoided, then experience from this project indicates that it is smart to avert from that direction with regards to time.

Using mobile 4G data is expensive. In case of the drones communication pipeline, if the system sends seven images per second with an average bandwidth usage of 270,100 bytes per prediction for a sustained period that quickly adds up. 1 minute will make the client output 113,442,000 bytes or 108 megabytes. 10 minutes of streaming predictions at the before mentioned rate will become 1.06 gigabyte of data, which is not sustainable in the long run.

The development of mobile device performance has increased drastically in the last years. The first smart-phones are orders of magnitude weaker in CPU cycles as compared to modern devices. The need for separate servers used for neural network inference may decrease as a consequence of the development, and we may see smart-phones with capabilities sufficient for real-time object detection analysis in the future. The maturation of mobile computers could alleviate future developers from having to think about distributed computing when handling challenges as the one in this thesis.

5.3 Deep learning for inspection and surveillance

This part of the discussion will regard the methods that assisted in answering **RQ3**: How can one use deep learning for inspection and surveillance?

There is a wide range of high performing object detectors as posed in chapter 2. They have different capabilities and drawbacks, the two most important parameters to consider in this thesis are the accuracy and speed at which inference can be performed. The object detection architecture that was chosen for this project is the single shot box detector - SSD. SSD provides a great compromise between accuracy and speed, and it has been shown to handle high frame rates when it is running at sufficiently powerful hardware. We trained multiple versions of SSD through several iterations with different datasets. Both the strengths and weaknesses became apparent during this project. When objects are tiny, the detector has a big problem with detecting them, as was shown in the first iterations when we trained the system on airplane photographs. It is important to have a balanced and nuanced dataset to train the object detector to provide adequate predictions. As we progressively modified and augmented the training process, the detector became gradually more robust. That was one of several takeaways from different barriers experienced during the project. We trained object detectors with high recall. However, the precision was not perfect. We could have probably relieved the precision problem by increasing the size of the dataset and gathering more images of more cars and at a higher variation of altitudes and conditions. The data gathering process is, however, time-consuming, and we had to prioritize our resources towards the end of the project to finish all aspects of the system.

The reason that the object detector from iteration 1 failed was that after careful inspection, we saw in the system monitor that the training process attempted to allocate 50 gigabytes of memory. We adjusted some of the settings in the configuration file such as setting the batch size to 1 in an attempt to fix it, but it failed. The images we gave to the network were simply too big, even though the training automatically scales the input to 300×300.

During the second iteration of training, the training got the loss down around 2. However, the loss had extreme variance during the training, and we did not see the convergence we had hoped for. When we performed a spot test with images from the test-set we saw no bounding boxes nor did we get any results when we ran the predictions on the footage from the drone. These results made us realize that the cars had gotten too small in the images we used for training, so small that SSD was not able to pick them up. The cars in the down-scaled images probably did not manage to cascade any information to the last layers of the neural network, and this would explain why the loss during training had such great variance.

During iteration 3, the training converged fast with the tiled airplane training set and stabilized around a value of 2. When we ran an evaluation on both images from the test set as well as drone footage we could see that the system had learned very well how the cars from the airplane photos looked, but it had some struggle with the drone footage. Introducing augmentations for iteration 4 did help to generalize the network, and the performance improved somewhat. The extra scale augmentation for iteration 5 made the performance worse than the previous two, we do however suspect that more data or longer training sessions would make it perform better in the validation test. Iteration six did not provide notable improvements.

The real improvements happened after we changed the dataset to the drone-based data. The mAP score more than doubled. Even though the cars in the airplane images were similar to the cars in the drone footage, they were not similar enough. Zooming in on the airplane images reveal a lot of pixelated noise, which is not present in the drone footage. The final two iterations produced object detectors with satisfactory mAP scores, however, they still gave quite a lot of false positives during field tests. We suspect that playing around with the hard mining ratio could improve the number of the false positives. We used the ratio recommended in the SSD paper , 3:1, the detector would generate 3 false examples per positive during training. If we made the detector generate 4 or more negatives per positive, we could maybe se an improved precision.

5.4 Image-based geospatial data generation

In the last piece of the discussion we will evaluate the techniques used to answer **RQ4**: is it possible to accurately map perceived objects to global coordinates using commercial drones?

For surveillance drones to be truly useful, they should be able to relate detected objects to geographical locations. The approximation of geolocations was an interesting challenge we set for ourselves in this project. To showcase this ability, we trained an object detector with the ability to detect cars. We ran the object detector on a server with a communication pipeline connected to the drone. As the system was set up, the drone essentially became an eye in the sky. By using the neural network object detections and parameters provided with the drone we could approximate global coordinates from the bounding boxes. One can ask the question, how robust are the predictions though? There are many uncertainties related to the calculations, among them is the drones global positioning system parameters. It is hard to ascertain just how big that uncertainty is, but the DJI Mavick Pro states that it utilizes a combination of signals from the Russian navigations system GLONASS in addition to the American system GPS. GPS is known to have an accuracy in the range around 3 meters, yet experiments seem to indicate a higher accuracy, this could be related to the fact that most of the experiments performed in this thesis are operated 20 meters or higher above the ground. Albeit a bit ironic, the way we annotate the true locations of the cars for the geolocation experiments may be less precise than the drones predictions, and this is because the phone is used at ground level where interference with the GPS signal from surrounding objects may cause lower accuracy.

There are additional variables present in the geolocation approximations. The measured altitude contains measurements from the barometric sensor in the drone. The altitude is measured relative to the starting point, but as with the positioning system used in the DJI Mavick Pro, the user manual contains no information regarding measurement accuracies. Our results did, however, show small discrepancies between the location annotated using the phone and the drones prediction. The distance errors were within a range of 7 meters to 1.2 meters, which given the premises are intriguing results. An interesting observation we made when we analyzed the results from the geolocation missions is that the predictions that had the largest errors were usually also the ones closest to tall objects, walls and buildings. When we inspected the observation closer, we could see that the point annotations corresponding to objects close to tall structures usually seemed more displaced in comparison to annotations produced in open areas. We think this is because the phone gets worse GSP signals due to interference from the surrounding objects. We believe the predictions that got the largest discrepancies are not much worse than the other ones, we rather suspect the annotation markings here to be inaccurate.

A simpler way to estimate geolocation from bounding boxes could be to use the drones current location when a bounding box is detected. That would provide a crude approximation. At low altitudes that method should provide decent results, assuming that the GPS signal is good. For instance, if the drone is at an altitude of 10 meters, its field of view is 12×7 meters. That means that the error is in the range of 0 to 7 meters, which is comparable to the range of errors we saw in the experiments performed. Nevertheless, when the drone rises to altitudes above 30 meters, the field of view is 39×21 meter, which means the distance error is in the range from 0 to 22 meters. The geolocation approximation presented is thus a lot more accurate. The errors we saw were mostly in the range between 1 to 7 meters.

We used the geolocation calculations to create a projection of the drones camera view down on the satellite map used in the client application. Figure 4.12 is a fitting example of how good the visual results are. One can only empirically evaluate how well the projected overlay fits with the surrounding satellite image, but in many cases, the results seemed accurate. We lowered the frame rate when computing the projected overlay to save computational power. As a consequence, the drone should be stationary on a location around one second before projecting an overlay for optimal results.

5.5 Reflections on the research process

The system developed in this project consists of several parts. We chose to create a server for neural network inference, which in itself consists of two programs, as well as a GUI-based client controlling the drone. The server was developed in Python and C++ , and Objective-C and C++ on the client. Although the final system met our performance requirements, working with such a large number of different programming languages introduces many moving parts, and getting the inter-process communication correct is hard. In hindsight, it would be wise to spend more time upfront researching potential system architectures to ease the rest of the engineering. We furthermore spent a lot of time extracting image data from the drone SDK, the system was dependent upon this to work, however we felt that an unnecessarily large amount of time was spent getting this right. Despite the engineering headaches that we ran into during the project, we are excited about the final results. We created an entire autonomy platform for commercial drones that can perform missions in real-time.

Chapter 6

Conclusion and future work

6.1 Conclusion

This thesis has proved that commercial drones are capable of performing autonomous surveillance and inspection missions. We created a system comprised of commercial drone hardware paired with a custom made network communications platform. On the server side of the system, we deployed an SSD object detector to execute analysis of drone footage in real-time. The inference round-trip delay was observed in the range between 3 milliseconds to 350 milliseconds, depending upon network connectivity. Through a combination of neural network inference and geodetic methods we were able to approximate the geolocation of visible objects. The discrepancies between ground annotated geolocations and predictions were in the range between 1 to 7 meters. Furthermore, our best object detector achieved a mean average precision of 0.84 on the validation set for detecting cars from a birds eye view.

The final system is able to cover designated areas autonomously while executing complex analysis in real-time. As it stands, the platform can serve as foundation for a wide range of autonomous tasks.

6.2 Future work

6.2.1 Coastal garbage detection

Coastal garbage is a world-wide problem. The system developed in this thesis can be used to help detect garbage autonomously along the coast and give its geolocation, such that it can be collected at a later time. The steps necessary to realize a coastal garbage detection system as the one proposed is mainly to re-train the object detector. It has to be finetuned to detect garbage instead of cars. The other parts of the system are ready to use as is. One could use the autonomous area cover mission to make the drone cover coastal areas of interest. By utilizing the confidence scores of the detected objects along with their geolocation, one can produce a heat-map that shows where large concentrations of trash are located. All this is possible in real-time.

With the help of gamification, one could further imagine crowd-sourcing the garbage collection process through some form of internet application. By deploying multiple drones for the detection task, one can quickly cover large areas and make a true impact.

6.2.2 Communication pipeline

To attain better performance for surveillance and inspection missions out in the field one could experiment with other socket methods for real-time streaming. UDP networking does not utilize persistent connections, and the headers of the UDP datagram is minimized which means the communication has potential for going much faster.

6.2.3 Crowd analysis

By using a system like the one we created in this thesis, one can perform a wide variety of tasks. One can replace the object detection system with other kinds of neural networks such as neural network architectures for crowd analysis [6]. With the ever growing urbanization of the western hemisphere, large masses of people are continuously growing larger. One could enable cheap analysis of crowds to help governmental institutions in circumstances demanding for instance evacuation.

6.2.4 Image stitching

Image stitching is the process of combining multiple images with an overlapping field of view into one image. An interesting application is to use the geospatial mapping methods devised in this project to create a low-level, high-resolution aerial map. The method can take advantage of the area cover missions to autonomously map out an area marked by a user on the screen. By doing this, we hypothesize that one can create super high-resolution image maps autonomously. Such high-resolution maps could prove useful for applications such as agricultural analysis.

Bibliography

[1] (2018). Camera Calibration with OpenCV. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=calib. [Online; accessed 04-June-2018].

[2] (2018). FFmpeg. https://ffmpeg.org/about.html. [Online; accessed 02-June-2018].

[3] (2018). LabelImg: Graphical Annotation Tool for Python. https://github.com/tzutalin/labelImg. [Online; accessed 01-June-2018].

[4] (2018). Open Source Computer Vision Library. https://opencv.org/about.html. [Online; accessed 02-June-2018].

[5] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

[6] Cao, L., Zhang, X., Ren, W., and Huang, K. (2015). Large scale crowd analysis based on convolutional neural network. *Pattern Recognition*, 48(10):3016–3024.

[Farhadi] Farhadi, J. R. A. Yolov3: An incremental improvement.

[8] Girshick, R. (2015). Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448.

[9] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

[10] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[11] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[12] Kurose, J. F. and Ross, K. W. (2013). Computer networking: a top-down approach. *Addison Wesley Computing.*

[13] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer.

[14] Patterson, J. and Gibson, A. (2017). *Deep Learning.* O'Reilly Media.

[15] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788.

[16] Redmon, J. and Farhadi, A. (2017). Yolo9000: better, faster, stronger. *arXiv preprint.*

[17] Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99.

[18] Sau, T. (2015). Breath acetone-based non-invasive detection of blood glucose levels.

[19] Schmalstieg, D. and Hollerer, T. (2016). *Augmented reality: principles and practice.* Addison-Wesley Professional.

[20] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556.*

[21] Singh, A., Patil, D., and Omkar, S. (2018). Eye in the sky: Real-time drone surveillance system (dss) for violent individuals identification using scatternet hybrid deep learning network. *arXiv preprint arXiv:1806.00746.*

[22] Storn, R. (2005). Thinking About Thinking: The Discovery of the LMS Algorithm. `http://www-isl.stanford.edu/~widrow/papers/j2005thinkingabout.pdf`. [Online; accessed 31-May-2018].

[23] Tsai, R. Y. (1986). An efficient and accurate camera calibration technique for 3d machine vision. *Proc. of Comp. Vis. Patt. Recog.*, pages 364–374.

[24] Veness, C. (2018). Vincenty's Direct Formula. `https://www.movable-type.co.uk/scripts/latlong-vincenty.html`. [Online; accessed 02-June-2018].

[25] Vincenty, T. (1975). Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, 23(176):88–93.

[26] Zhang, Z. (2000). A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334.