Low-Latency Detection and Tracking of Aircraft in Very High-Resolution Video Feeds

Låglatent detektion och spårning av flygplan i högupplösta videokällor

Jarle Mathiesen

Supervisor : Magnus Bång
Examiner : Erik Berglund
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Jarle Mathiesen
Abstract

Applying machine learning techniques for real-time detection and tracking of objects in very high-resolution video is a problem that has not been extensively studied. In this thesis, the practical uses of object detection for airport remote towers are explored. We present a Kalman filter-based tracking framework for low-latency aircraft tracking in very high-resolution video streams. The object detector is trained and tested on a dataset containing 3000 labelled images of aircraft taken at Swedish airports, reaching an mAP of 90.91% with an average IoU of 89.05% on the test set. The tracker is benchmarked on remote tower video footage from Örnsköldsvik and Sundsvall using slightly modified variants of the MOT-CLEAR and ID metrics for multiple object trackers, obtaining an IDF$_1$ score of 91.9%, and a MOTA score of 83.3%. The prototype runs the tracking pipeline on seven high resolution cameras simultaneously at 10 Hz on a single thread, suggesting large potential speed gains being attainable through parallelization.
Acknowledgments

I want to give a special thanks to Magnus Bång for making this thesis possible. A lot of pieces had to come together during the course of my work for the end result to be achieved, and your help has been instrumental in making that happen.

I also want to thank the Swedish air navigation service provider LFV for providing the remote tower video footage. I want to thank Lothar Meyer from LFV in particular for our meetings and correspondence.

I want to thank Natanael Log for providing detailed and valuable feedback on my thesis. Finally I want to thank my family for being supportive of me moving to Sweden to study, as well as all the wonderful people I have gotten to know during my stay here.
Contents

Abstract

Acknowledgments

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

1.2 Aim

1.3 Research questions

1.4 Delimitations

2 Theory

2.1 The Current State of Digital Air Traffic Control Environments

2.2 Basic Principles of Machine Learning

2.3 Deep Neural Networks

2.4 Convolutional Neural Networks

2.5 Visual Object Tracking

3 Method

3.1 Frameworks, Platforms and Hardware

3.2 Preparing the Videos

3.3 Detecting Aircraft

3.4 Tracking Objects of Interest

3.5 Evaluating the Application

4 Results

4.1 Tracking metrics

4.2 Object Detection Performance

4.3 Application Latency

4.4 Overall

5 Discussion

5.1 Results

5.2 Method

5.3 Source Criticism

5.4 The work in a wider context

6 Conclusion

v
List of Figures

2.1 Detailed view of a neuron ............................................. 5
2.2 Fully-connected neural network ..................................... 6
2.3 Visualization of a CNN architecture .............................. 7
2.4 Convolution with a $3 \times 3$ filter ................................ 8
2.5 Visualization of CNN filter activations .......................... 8
2.6 Max Pooling .......................................................... 9
2.7 Illustration of SGD step size ....................................... 11
2.8 Background subtraction using Gaussian mixture models .... 15
2.9 Binary image dilation ................................................. 16
2.10 Binary image erosion ................................................. 16

3.1 Flowchart for the proposed system ................................. 22
3.2 Background subtraction mask result comparison on SE-MJJ ... 23
3.3 Resulting mask after morphological operations ................ 23
3.4 YOLOv2 Bounding Box .............................................. 26
3.5 Dataset labelling strategies ......................................... 28
3.6 Logarithmic Loss for ArlandaNet .................................. 30
3.7 IoU comparison between ground truth and detection ......... 36

4.1 Detection snapshots for ArlandaNet ................................. 38
4.2 Detection snapshots for YOLOv2 ................................... 38
4.3 Average time distribution for processing a frame ............... 39
4.4 NTJ3102 turning on the runway after a landing ............... 40
4.5 SE-MJJ being occluded during landing ......................... 41
4.6 SE-MJJ detail .......................................................... 41
4.7 NTJ3102 detail ........................................................ 42
4.8 SE-MKA detail ........................................................ 42
4.9 CCIXT detail .......................................................... 43
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Kalman filter process model matrices</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Source video information</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Darknet-19</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Final three layers of YOLOv2</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Configuration values used for data augmentation</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Final three layers of ArlandaNet</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>A selection of trackers from the OpenCV contrib module</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Kalman filter state variables</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Detection result from the SE-MKA video</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>MOT Metrics</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>MOT Metrics explanation</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Time until classification</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Average latency for frame processing</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Top five trackers from MOT16</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>PASCAL VOC2012 test detection results for the class aeroplane</td>
<td>45</td>
</tr>
</tbody>
</table>
Introduction

Air traffic controllers (ATCOs) oversee numerous landings and take-offs daily, and play a large role in maintaining a safe flow of air traffic in real-time. This tremendous responsibility is the reason that the profession is widely regarded as one of the most mentally demanding professions in the world.

A large number of factors influence the decisions taken by an air traffic controller, including equipment, weather, traffic volume, and human factors. While air traffic control (ATC) environments are constantly being modernized, with innovations such as remote and virtual tower (RVT), the decision making of the controllers are central to a functioning air traffic service. One aspect of the profession that has not changed despite modernization, is that a significant amount of decision making done by ATCOs is based on visual information that is obtained by observing through the tower window [1]. Few automated tools exists to aid the controller in decision making, and no automated alert systems exist that can detect potential dangerous situations on airfields [1].

Previous studies have shown that air traffic controllers tends to be sceptical to autonomous ATC solutions where decision-making shifts away from the controllers themselves [3], which could slow down the adoption of these types of solutions. This scepticism is not unfounded, as there are important challenges to partially automating the air traffic controller workload, such as the potential loss of situational awareness, deskilling, and automation surprises [4]. Furthermore, introducing new systems risk further splitting the focus of the ATCO between the different areas of interest [1].

Object classification- and detection efforts using Deep Learning models has been improving steadily since the major breakthrough of AlexNet in 2012, as both large labelled datasets and highly accurate models have become widely available in recent years. Most efforts are made in the area of image classification for still images, but results from these efforts can be directly adapted for object recognition in live video as well.

A framework with robust object detection- and tracking capabilities could be the basis of new systems for use in Remote Tower Centres (RTC). The types of systems that could benefit from such tracking capabilities are - among others - runway incursion alert systems and systems for automated ATCO visual attention analysis.
1.1 Motivation

Remote Tower Centres are supplied with a video feed from an airport that is based on up to 15 high-resolution cameras. Detecting and tracking aircraft in such a large amount of raw data requires very efficient and computationally inexpensive algorithms in order to be usable in real-time decision making by the ATCO. Varying weather and illumination further complicates the tracking problem, as most state of the art trackers generate appearance models that are sensitive to variation in colour and lightning. Furthermore, the spatial input size for modern object detectors are much too small to keep enough spatial information from all cameras, or even from one camera at a time. These are all obstacles that must be tackled in order to apply any form of automated object detection and visual object tracking on a remote tower video feed, while keeping computation time per frame low.

1.2 Aim

The purpose of the thesis project is to explore how computer vision and deep learning can be used for tracking aircraft in remote tower video feeds. The result of the thesis will be a framework realised as a prototype system for detecting and tracking aircraft in remote tower video feeds.

1.3 Research questions

The following research questions will be answered in this thesis:

1. How can a prototype system for tracking objects in very high-resolution video feeds be implemented so that the system functions in near real-time while having high precision and recall?

2. How can object tracking be used in combination with object detection in order to only track airplanes in the prototype?

3. How well can a custom trained object detection convolutional neural network for detecting airplanes perform versus pre-trained networks?

1.4 Delimitations

The system will be built for remote air traffic control towers video feeds with static cameras.
The relevant theory for the thesis is presented in this chapter. The theory will give a high-level view of certain topics that are required for understanding how the system is implemented and evaluated in later chapters.

2.1 The Current State of Digital Air Traffic Control Environments

The air traffic tower work environment has shown to be very similar worldwide, with a traditional set of tools such as flight progress strips, outside view, radar, and possibly ground radar for larger airports [1]. The digital innovations in these work environments have been limited to electronic strips with flight information, replacing traditional paper strips. The e-strips only reflect instructions entered manually into the system by the air traffic controllers, and thus provide no proactive safety measures for the controller. This means that there is currently very little automation in the tower environment.

Situations where unauthorized vehicles or persons is on a runway, otherwise known as runway incursions, continue to be a persisting problem at airports. One such situation is when there is an aircraft present on the arrival runway; ideally there would be a system in place making the ATCO aware of the unauthorized aircraft. There are no widespread systems in place for detecting common situations where runway incursions can occur [1]. Detecting incursions is currently done by having the air traffic controller spend a significant time observing the runway through the tower window. While the air traffic controllers spend a significant time observing through the ATC tower windows, it is not obvious whether the controller is actively focusing while doing so, or simply resting. Analyzing eye movement has shown promise for recognizing task complexity and mental workload [6]. Through analyzing simple and complex situations at an airport, previous studies have shown that air traffic controllers follow highly trained procedures, and that scan patterns are broadly similar between the controllers [2].

Several factors may be the cause of the lack of automation in the air traffic controller environment. Some challenges when automating are: loss of situational awareness, deskilling, and automation surprises. There is also the necessity for the automated systems to provide useful support to the user. Failing to provide useful support to the air traffic controller will cause them to revert to traditional tools and patterns [1].
2.2 Basic Principles of Machine Learning

At its most basic, a machine learning algorithm is an algorithm that is able to learn from data. Said learning can be defined by the description that “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.” This definition opens for a wide array of machine learning problems, but only the most relevant principles used to construct machine learning algorithms will be presented in this section. It is important to distinguish between the "task" and the learning itself: learning is the means used to attain the ability to perform the task.

2.2.1 Classification

Classification is a common type of machine learning task (T) where the program is asked to specify which of k categories some input belongs to. For example, object recognition is a classification task where the input is an image, and the output is a numeric code identifying the object in the image. The learning algorithm is usually asked to produce a function:

\[ f : \mathbb{R}^n \rightarrow \{1, \ldots, k\} \]

When \( y = f(x) \), the model maps an input described by vector \( x \) to a category that is identified by the numeric code \( y \).

Mapping directly to a class is not the only way to define a classification task - other types of classification tasks involves training the learning algorithm to produce a function \( f \) that outputs a probability distribution over classes.

2.2.2 Performance Measure

Evaluating a machine learning algorithm is done by performing a quantitative measure of its performance, specific to the task being performed by the system. For classification tasks, the accuracy of the model is measured. The accuracy is the proportion of examples for which the model predicts the correct output.

The accuracy is usually measured on a test set of data that is separate from the data set that the machine learning algorithm was trained on (training set). This is done in order to test the algorithm on data it has not seen before, which will give an indicator of how well the system will function once deployed.

2.2.3 Supervised Learning

Broadly speaking, supervised (as opposed to unsupervised) learning algorithms, experience (E) a dataset containing features as well as labels associated with each example. For example, a car dataset can contain images of cars together with labels of the car model associated with each example. An unsupervised learning algorithm does not have the labels associated with each example, and must therefore learn to make sense of the dataset itself without this guidance.

2.3 Deep Neural Networks

The goal of a deep neural network (DNN) is to approximate some function \( f^* \). This is done by defining a mapping \( y = f(x; \theta) \) and learning the value of the parameters \( \theta \) that results in the best approximation of the function \( f^* \).

DNNs are called feedforward networks because they are composed of many functions in an acyclic graph where information only flows in one direction. If a network contains three functions \( f^{(1)}, f^{(2)}, \) and \( f^{(3)} \) that are connected in a chain to form \( f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))) \),
2.3. Deep Neural Networks

Figure 2.1: Each input $x_i$ to a neuron is weighted with $w_{ij}$, added to a sum, and passed through an activation function $\phi$ in order to calculate an activation value $o_j$. $\phi$ is the activation function, and $w_{ij}$ is the weight associated with input $x_i$ for the neuron. An illustration of a neuron is shown in Figure 2.1.

Long chains of functions lead to a "deep" network, hence the name "deep neural networks". DNNs are trained by driving $f(x)$ to match the given function $f^*(x)$. The training data should consist of approximate examples of $f^*(x)$, with each sample $x$ having an associated label $y = f^*(x)$. These examples specify what the output layer - the final layer in the network - must do at each point $x$, which is to output a value that is close to $y$.

There are layers that the training data does not specify the desired output for; these layers are called hidden layers, and can be seen as the internals of a DNN. The learning algorithm must decide how to use the hidden layers of the network for the best approximation of $f^*$ by tweaking the parameters $\theta$ during training.

The name neural in "deep neural network" comes from the fact that DNNs are vaguely inspired by biological neural networks that makes up animal brains. Typically, each hidden layer in the DNN is vector valued, with the dimension of the hidden layers determining the width of the model. Thus, instead of interpreting the layer as representing a single vector-to-vector function, we can think of the layer as many elements that act in parallel where each element represents a vector-to-scalar function. In this representation the learned parameters for the model are called weights and biases. Each element receives many inputs from other units, and calculates its own activation value $o_j$ according to the formula:

$$o_j = \phi\left(\sum_{i=1}^n w_{ij}x_i\right)$$

where $\phi$ is the activation function, and $w_{ij}$ is the weight associated with input $x_i$ for the neuron. An illustration of a neuron is shown in Figure 2.1.

Layers where all the activation outputs of one layer is connected to each input of the next layer, are called fully-connected layers. The last fully-connected layer in a deep neural network is called the output layer. A simple, fully-connected neural network is shown in Figure 2.2.

If the network is used for classification, we want to produce a vector $\hat{y}$ where:

$$\hat{y}_i = P(y = i \mid x)$$

This is known as a categorical distribution, where each element $\hat{y}_i$ is between 0 and 1, and the entire vector $\hat{y}$ sums up to 1.

We assume that the final linear layer (sans activation function) predict unnormalized log probabilities on the form:

$$z = W^T h + b$$ (2.1)

where $W$ is the weight matrix, $h$ is the input vector, $b$ is the bias matrix, and $z_i = \log P(y = i \mid x)$.

By exponentiating and normalizing $z$ with an activation function called the softmax function, we can obtain an output $\hat{y}$ where all the values in $\hat{y}$ sums up to 1, and are in the range

$^1$Diagram of an artificial neuron by Chrislb licensed under CC BY-SA 3.0
2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a specialized kind of neural network that has shown good results in a wide range of tasks such as visual object detection. The input to these kinds of CNNs is an image encoded as a 3-dimensional matrix, with each dimension corresponding to the colour channels red, blue and green. The neurons of convolutional neural networks are arranged in three dimensions: width, height, and depth (depth here refers to the third dimension in the layer - not to the depth of a full neural network).

Every layer has a simple mode of operation: a layer transform a 3D volume of activations to another 3D volume through a differentiable function that may or may not have parameters. By assuming that the input is an image, CNN architectures can be very efficient compared to the densely connected general neural nets by vastly reducing the amount of parameters in the network. Instead of fully connected layers, the neurons in a layer is only connected to a small region in the preceding layer.

Example: Neural network by Kjell Magne Fauske licensed under CC BY 2.5

Figure 2.2: A fully-connected neural network. Each blue circle represent a neuron stacked in a vertical layer.

(0,1). This will in effect represent a probability distribution over \( n \) different classes, also called categorical distribution. The softmax function is given by the following equation:

\[
\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} = \hat{y}_i
\]

(2.2)

Which calculates the probability for the \( i \)th class given the vector \( z \).

When using the softmax function we want to maximize the log-likelihood given by:

\[
\log P(y = i; z) = \log \text{softmax}(z)_i
\]

The softmax function works well for training the softmax layer because we can relate the contribution from the input \( z_i \) directly to the log-likelihood cost function:

\[
\log \text{softmax}(z)_i = z_i - \log \sum_j e^{z_j}
\]

(2.3)

In order to maximize the log-likelihood function, the first term \( z_i \) must be pushed up, and the second term must become very small. The intuition is that the second term, \( \log \sum_j e^{z_j} \) can be roughly interpreted by \( \max_j z_j \). In turn, this means that the negative log-likelihood cost function will strongly penalize the most active incorrect prediction: if the correct answer is the largest input to the softmax, the \( -z_i \) term and \( \log \sum_j e^{z_j} \approx \max_j z_j = z_i \) term will roughly cancel. By having these terms cancelling, the training cost will be dominated by other training examples that are not correctly classified.

2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a specialized kind of neural network that has shown good results in a wide range of tasks such as visual object detection. The input to these kinds of CNNs is an image encoded as a 3-dimensional matrix, with each dimension corresponding to the colour channels red, blue and green. The neurons of convolutional neural networks are arranged in three dimensions: width, height, and depth (depth here refers to the third dimension in the layer - not to the depth of a full neural network).

Every layer has a simple mode of operation: a layer transform a 3D volume of activations to another 3D volume through a differentiable function that may or may not have parameters. By assuming that the input is an image, CNN architectures can be very efficient compared to the densely connected general neural nets by vastly reducing the amount of parameters in the network. Instead of fully connected layers, the neurons in a layer is only connected to a small region in the preceding layer.

\[\text{Example: Neural network by Kjell Magne Fauske licensed under CC BY 2.5}\]
CNNs are built using three types of layers: the convolutional layer, the pooling layer, and the fully-connected layer (identical to the fully-connected layers used in DNNs). Whereas the fully-connected layers make up the most of DNNs, they are only seen in the final layers for CNNs. An example CNN architecture with the basic layers can be seen in Figure 2.3.

2.4.1 Convolutional Layers

The parameters of a convolutional layer is a set of learnable filters, which are typically small in the spatial dimension. A typical filter on the first layer of a CNN for RGB images will have the size $5 \times 5 \times 3$, corresponding to 5 pixels in width and height, and a depth of 3. The depth of the filters in the first layer is 3 because the input image has three channels: red, green, and blue. The filter might be much smaller in the spatial dimension than the input image, but the filter will be applied to the entire image by "sliding" (convolving) the filter over the input volume and compute the dot product between the filter and the input at all positions. The output of the convolution operation on a 2D matrix is a new 2D-matrix, illustrated in Figure 2.4.

Note that the "sliding" is only performed spatially; since the depth of the filter matches the depth of the input volume, no information is lost. By sliding one filter across the input volume, we will produce a 2-dimensional activation map that contains the response from the filter at any given position. The response from the filter will intuitively be visual features such as edges, colours, or basic shapes on early layers, as can be seen in Figure 2.5.

Each convolutional layer will have a set of multiple filters, with the output from each filter being stacked in the depth dimension. This means that the depth of the output volume for any given convolutional layer is equal to the number of filters the layers consists of. A typical progression of CNN architectures is to shrink the spatial dimension while increasing the depth of the convolutional layers.

Convolutional layers calculates their output using a mathematical operation named convolution: the input to a layer is convolved with a filter, which creates an output that is fed as input to the next layer in the network. The convolution operation for a two-dimensional
input with a two-dimensional filter is given by the formula

\[ S(i, j) = (I \ast K)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n) \]

where \( S \) is the resulting feature map, \( I \) is the two-dimensional input, and \( K \) is the filter.

Convolutional layers have three hyperparameters that control the size of the output volume: filter spacial size \( F \), depth \( K \), stride \( S \), and amount of zero-padding \( P \). The depth of the output volume corresponds to the number of filters belonging to a convolutional layer. The stride dictates how the filter is slid across the input volume: if the stride is 1, the filter will be moved one pixel at a time; with a stride of 2, the filter will jump 2 pixels at a time, and so on. A larger stride will produce a smaller output volume spatially. Finally, zero-padding is a hyperparameter for controlling the spatial size of output volumes by padding the input volume with zeros around the border. With these hyperparameters, the output volume size of a convolutional layer that accepts a volume of size \( W_1 \times H_1 \times D_1 \) can be computed with the
2.4. Convolutional Neural Networks

Figure 2.6: Max pooling a single depth slice with a $2 \times 2$ filter size and a stride of 2.

Using $1 \times 1$ convolutions as proposed by [11] has become common, and can be interpreted as a coordinate-dependent, cross-channel parametric pooling layer: the filter is coordinate-dependent because the filter performs the transformation on a single coordinate point per operation, and it is cross-channel because the convolution is performed on each depth of that coordinate point. In practice, the $1 \times 1$ convolution is often used for preventing an explosive depth growth at deeper layers, while retaining cross channel information. Additionally, convolutional layers based on $1 \times 1$ filters can replace fully-connected layers, as we will see in section 2.4.3.

Filters that activated on visual features were typically hand-engineered up until 2012 when the CNN AlexNet competed in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) and achieved a top-5 error of more than 10 percentage points better than the runner-up [5]. Modern deep CNNs can be composed of as many as 152 layers [12], where filters in the deeper layers activate on more complex features such as for example "eyes" and "flowers" [9].

2.4.2 Max Pooling Layer

Max pooling is a function used to modify the output of a layer by outputting the maximum input value within a rectangular region [7]. In effect, the pooling layer downsamples the input volume spatially (width and height), as can be seen in Figure 2.6.

The pooling operation provides several benefits, such as invariance to translation. Invariance to translation means that small translations in the input does not lead to many pooled outputs changing. Max pooling is a fixed function, and therefore does not require any parameters to learn.

2.4.3 Fully-Connected Layer

Just as in regular neural networks, fully-connected layers in CNNs consists of neurons with full connections to all activations in the previous layer. The activations is computed with matrix multiplication followed by a bias offset.

Since both fully-connected layers and convolutional layers compute dot products, it is possible to convert between a fully-connected layer and a convolutional layer using filters of the same spatial size as the final activation volume. This is possible because of the spatial dimension reduction that has occurred by the final layers in the CNN: for example the AlexNet

---

\[ \begin{align*}
W_2 &= (W_1 - F + 2P)/S + 1 \\
H_2 &= (H_1 - F + 2P)/S + 1 \\
D_2 &= K
\end{align*} \] (2.4)

---

Max pooling by Andrej Karpathy licensed under The MIT License
architecture downsamples the input spatiality by a factor of two for each layer, starting with an image of size $224 \times 224 \times 3$ with the final activation volume size of $7 \times 7 \times 512$. AlexNet uses two fully-connected layers of size 4096 and a final fully-connected layer with 1000 neurons for classification. The first fully-connected layer can be replaced with a convolutional layer with 4096 $7 \times 7$ filters (recall that the depth of the filter is equal to the depth of the input volume, in this case 512). This will result in a $1 \times 1 \times 4096$ input volume to the next layer, which is replaced with a convolutional layer again with 4096 $1 \times 1$ filters. The final layer is then converted to a convolutional layer containing 1000 $1 \times 1$ filters, giving a final output of $1 \times 1 \times 1000$, thereby expressing the class probability in filter space.

### 2.4.4 ReLU Layer

Most convolutional layers are followed directly by a rectified linear unit (ReLU) layer in modern architectures. The output of a convolutional layer will highlight a feature in the input data by activating the filter, which is achieved by passing the result of the convolution through the activation function. The activation function - the rectifier - is defined as the positive part of its argument:

$$f(x) = \max(0, x)$$  \hfill (2.5)

The activation is applied element-wise, meaning that the size of the output volume will be equal to the size of the input volume. Nonlinearity is an important property of activation functions, as linear activation functions will make the network output a linear function of the input, regardless of how many layers the network is composed of.

### 2.4.5 Backpropagation

Filters in a CNN are randomly initialized, and completely constructed during training through a process named backwards error propagation (backpropagation), where the weights of each filter is iteratively adjusted based on the contribution of the neuron to the total loss. One method of performing backpropagation is using stochastic gradient descent (SGD) which uses the chain rule from calculus. The updated filter weights (denoted $w^{t+1}$) after an iteration of SGD, where $w^t$ are the current weights at iteration $t$, $\eta$ is the step size, and $\epsilon$ is the total loss, is given by the formula:

$$w^{t+1} = w^t - \eta \frac{\partial \epsilon}{\partial w}$$  \hfill (2.6)

Intuitively, an iteration of SGD will drive the loss towards a local minimum for the loss curve with step size $\eta$ (also called learning rate), since the gradient of the error ($\frac{\partial \epsilon}{\partial w}$) points in the direction in which the error has steepest increase. By increasing the size of $\eta$, the correction might overshoot and increase the error, but keeping $\eta$ too small will require many more iterations to reach the local minimum. The effect of the step size is illustrated in Figure 2.7.

### 2.4.6 Transfer Learning

When it comes to CNNs trained for image classification, it has been shown that broadly speaking the first layers are general, while the final layers are more specialized on the training dataset. If a CNN is trained to recognize cars, we can assume that the earlier layers of the network activates on primitive features such as lines, circles and triangles; the final layers will activate on complex features such as headlight and rear-view mirror. If we then wanted to train a new CNN for recognizing boats, we would see the same trend: the first layers would activate on primitive features, and the final layers would activate on more complex features such as hull and bow. This observation can be used to speed up training of new CNNs: if we have access to the trained weights for a CNN, we can replace the final layers and resume training on our own dataset. This process is called transfer learning, and is primarily used to shorten the time required to fully train a CNN.
2.4. Convolutional Neural Networks

Figure 2.7: Illustration of the SGD step size for a loss function where the white vector is the negative of the gradient. By choosing a too large step size, we overshoot the local minimum. By choosing a too small step size, we will have to perform many iterations to reach it.

2.4.7 Data Augmentation

Data augmentation for image classification is a way of increasing available training data by the usage of simple image manipulation techniques such as cropping, rotating, and flipping the input images [13].

2.4.8 Evaluation

Mean average precision (mAP) is the primary metric for measuring the accuracy of object detectors. In order to calculate the mAP of a set of predictions, it is necessary to calculate the precision and recall. A bounding box prediction is said to be a true positive (TP) if it has predicted the correct class and has an intersection over union (IoU) ratio of over 0.5 with a ground truth bounding box, corresponding to a 50% overlap or more. A prediction with the wrong class or a smaller overlap than 50% (or both) with a ground truth bounding box is said to be a false positive (FP). A ground truth bounding box that does not have a corresponding prediction with an IoU over 0.5 counts as a false negative (FN). The definition of IoU is given by the equation:

\[
\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}} \tag{2.7}
\]

where area of overlap is the area of the overlap between the two bounding boxes, and the area of union is their total combined area.

The equations for precision and recall are thus:

\[
\text{Precision} = \frac{TP}{TP + FP} \quad \tag{2.8}
\]

\[
\text{Recall} = \frac{TP}{TP + FN} \quad \tag{2.9}
\]

\[
F_1 = 2 \frac{PR}{P + R} \quad \tag{2.10}
\]

where precision is the percentage of correct detections, and recall is the percentage of ground-truth detections that are correctly identified.
The average precision (AP) for a class summarizes the precision/recall curve for that class. It is defined as the numerical integral of the precision/recall curve for a set of eleven recall levels, starting at 0 with a step size of 0.1.

A common benchmark for object detectors is the Visual Object Classes Challenge (VOC) that was held yearly between 2005 and 2012. From 2007 and onwards, the VOC challenge contained a test dataset of 20 different classes. The last VOC challenge was held in 2012, but a different challenge called the Common Objects in Context (COCO) challenge is still held yearly. The COCO dataset contains 80 different object classes. Both challenges are useful benchmarks for comparing object detectors, as they provide a common dataset that makes direct comparison of detectors possible based on quantifiable metrics such as mAP.

2.5 Visual Object Tracking

Visual object tracking is the problem of estimating the trajectory and transformation of an object in a sequence of frames when only the initial location of a target is known. Among the factors affecting object tracking is variations in scale and appearance, occlusions, and motion blur. There are two common tracking approaches that is used in order to learn an appearance model of the target: discriminative and generative methods. The learned appearance model is used for estimating the state of the target in a new frame. This estimated state consists of the target location and size, which are properties that can be influenced by both the motion along the camera axis and the changes in target appearance. Using brute-force for estimating the change in scale is the most straightforward approach to estimating the target scale, but this approach can be computationally expensive, making it unsuitable for real-time systems. Visual trackers based on discriminative correlation filters (DCF) are both computationally efficient, and has shown to have a significant performance advantage over the brute-force approach. In fact, all top three trackers in the Visual Object Tracking 2014 challenge were based on correlation filters. The speed achieved by the correlation filters comes from the fact that the Fast Fourier Transform (FFT) can be applied both at the tracker learning and detection stages. DCF-based trackers typically focus on translation estimation, but novel frameworks such as the discriminative scale space tracker (DSST) has gone even further, employing separate DCFs for explicit translation and scale estimation. From these efforts, DCF-based visual object trackers are robust and accurate while providing real-time performance operating at over 100 frames per second (FPS) on the OTB dataset.

2.5.1 The Hungarian Method

The assignment problem is a fundamental combinatorial optimization problem that at its most general can be explained as assigning a set amount of tasks to a set amount of agents. Each agent-task assignment incurs a cost specific to that combination, and the assignment problem is solved when the assignments are done so that the total cost of the assignments is minimized.

The Hungarian Method is an optimization algorithm that solves the assignment problem. The Hungarian Method can be explained by representing the assignment problem as a non-negative \( n \times n \) matrix, where the element \( a_{ij} \) is the cost of assigning the \( j \)-th task to the \( i \)-th agent.

The first step of the algorithm is performing row operations on the matrix: the lowest value of each row is subtracted from each element in the row. After this operation we can attempt to assign tasks to agents such that each agent is only assigned to one task, and the penalty for the assignment is zero (achieved if the assignment cell has value 0).

If we fail to assign the tasks after the first step, the same operation is performed column-wise: the lowest value in each column is subtracted from each element in the column. We can now try to assign the tasks again, and if we fail move on to the final step.

In the final step, all cells containing zeros should be covered by marking as few rows and columns as possible. Finally, the lowest value of the unmarked cells should be subtracted from
all unmarked cells, and added to all cells that are marked twice. This step should be repeated until it is possible to solve the initial assignment problem. The problem is solved once the minimum number of lines used to cover all zeros is equal to $n$.

2.5.2 The Kalman Filter for Linear Filtering and Prediction

The Kalman Filter is an optimal state estimator, particularly useful for systems where measurements (for example from sensors) are distorted by measurement noise [21]. The filter tries to estimate a system’s true state by combining information from the system’s dynamic model, control inputs to the system, and sequential measurements. The estimate provided by the Kalman filter is better than the estimate that is obtained by only measurements, as the filters deals with the uncertainty that results from noisy sensor data. The filter produces an estimate by predicting the system’s current state (based on the previous estimated state), and combining that information with new measurements.

Take for example a Kalman filter that should track a moving object that can be represented by its position $x$ and velocity $\dot{x}$. The state for the system is then represented by the vector $\vec{x}$:

$$\vec{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \quad (2.11)$$

We cannot know exactly what the true value of these state variables are, but the Kalman filter assumes that they are random and Gaussian distributed. That means that the variables can be represented by the centre of the random distribution $\mu$ (the mean) and the variance of the distribution $\sigma^2$, which is the uncertainty of the variable. The true state of a variable is most likely to be centred at the mean, with the probability decreasing depending on the variance of the distribution. This is then our best estimate for the true state of the system at a given time $k$, denoted as $\hat{x}_k$.

To what degree two variables correlate is called covariance. The covariance of state variables at time $k$ can be represented as a covariance matrix $P_k$. Each element of the covariance matrix is the degree of correlation between the $i$th state variable and the $j$th state variable, making $P_k$ symmetric. This means that we can represent the state and covariance matrix at time $k$ as:

$$\hat{x}_k = \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} \quad (2.12)$$

$$P_k = \begin{bmatrix} \sigma^2_{xx} & \sigma^2_{x\dot{x}} \\ \sigma^2_{\dot{x}x} & \sigma^2_{\dot{x}\dot{x}} \end{bmatrix} \quad (2.13)$$

Using the state vector and a given covariance matrix at a time $k-1$ we can predict the next state at time $k$, $\hat{x}'_k$. This works even if we have expressed out state as a Gaussian distribution, because the new state estimate will also be a Gaussian distribution. In order to predict the new state, we need a matrix prediction matrix $F_k$ that models how the system changes over a discrete time step. The new state at time $k$ should be calculated by multiplying the prediction matrix $F_k$ with the state variable $\hat{x}_{k-1}$. In order to achieve this, our problem is well suited for the basic kinematic equations:

$$\begin{align*}
x_k &= x_{k-1} + \Delta t \hat{x}_{k-1} \\
\dot{x}_k &= \dot{x}_{k-1}
\end{align*} \quad (2.14)$$

We assume that the velocity at time $k$ equals the velocity at time $k-1$, modelling a system with constant velocity. Equation $2.14$ can be expressed as a matrix and used to predict the next state at time $k$:

$$\hat{x}'_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{x}_{k-1} \quad (2.15)$$
The prediction matrix is also used to calculate the updated covariance matrix by projecting the covariance at time $k-1$ with the prediction matrix:

$$P'_k = F_k P_{k-1} F_k^T$$  \hspace{1cm} (2.16)

Of course our predictions will not be perfect in a real world scenario, so we need a way to express the uncertainty of our prediction. After each prediction we therefore add the covariance matrix $Q_k$, called the \textit{process noise covariance matrix}, to the overall covariance. The \textit{prediction step} of the Kalman filter can thus be expressed as:

$$
\begin{align*}
\hat{x}'_k &= F_k \hat{x}_{k-1} \\
P'_k &= F_k P_{k-1} F_k^T + Q_k
\end{align*}
$$  \hspace{1cm} (2.17)

The Kalman filter also allows for optional control input to be added to the prediction step, which is omitted in this section. As we use the Kalman filter for passive tracking, we do not contribute any control input to the system, and can therefore leave out the control input term in the prediction step.

So far, the Kalman filter has used a motion model for predicting the state of the system at time $k$. This assumption can hold true for some time interval in a real life scenario, but our model of the real world will always meet its limits when external forces are acting on our system. In order to deal with this problem we can feed our Kalman filter observations of our object, which will be used to improve our estimate. For example we might have a sensor that measures the current speed of the object, with a given uncertainty for each reading. The sensors providing readings to the Kalman filter are modelled with the observation matrix $H_k$, which is a transformation of the predicted state $\hat{x}'_k$ to the expected sensor reading, expressed as the expected sensor reading mean $\mu_{\text{expected}}$ with variance $\sigma_{\text{expected}}^2$, given by the equations:

$$
\begin{align*}
\mu_{\text{expected}} &= H_k \hat{x}'_k \\
\sigma_{\text{expected}}^2 &= H_k P'_k H_k^\top
\end{align*}
$$  \hspace{1cm} (2.18)

The actual observation of one or more sensor values is a Gaussian distribution with a mean $\tilde{z}_k$ equal to the observed reading, and a sensor noise $v_k$ with the covariance matrix $R_k$. We now have both an expected sensor reading based on the current state estimation, and an actual sensor reading that is accurate to a certain degree. Since both our estimation and observation are Gaussian distributions, we can multiply them to obtain a new \textit{best guess} that is also a Gaussian distribution. The result of the multiplication is the overlap of the two distributions, which is more precise than either the sensor reading or the prediction alone. The multiplication of two distributions with means along each axis $\mu_0$, $\mu_1$ and covariance $\sigma_0^2$, $\sigma_1^2$ can be expressed in matrix form:

$$
\begin{align*}
K &= \sigma_0^2 (\sigma_0^2 + \sigma_1^2)^{-1} \\
\tilde{\mu}' &= \mu_0 + K (\mu_1 - \mu_0) \\
\sigma'^2 &= \sigma_0^2 - K \sigma_1^2
\end{align*}
$$  \hspace{1cm} (2.19)

where $K$ is defined as the \textit{Kalman gain}. Expressing the equations in terms of our predicted sensor reading and our actual sensor reading gives the equations:

$$
\begin{align*}
K &= P'_k H_k^\top (H_k P'_k H_k^\top + R_k)^{-1} \\
\tilde{\mu}' &= \hat{x}'_k + K (\tilde{z}_k - H_k \hat{x}'_k) \\
P_k &= P'_k - KH_k P'_k
\end{align*}
$$  \hspace{1cm} (2.20)

The equations in equation 2.20 forms the \textit{update step} performed by the Kalman filter. For each time step $k$ the Kalman filter will first perform the \textit{predict step} (equation 2.17) followed by the \textit{update step} (equation 2.20). Note that the update step only makes sense if there are new sensor readings at time $k$, which might not be the case. The update step can therefore be skipped when there are no new sensor readings, and the new estimated step will be completely based on the prediction step.

The matrices that are required for the Kalman filter framework are listed in Table 2.3.
### 2.5. Visual Object Tracking

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_k$</td>
<td>State-transition model</td>
</tr>
<tr>
<td>$H_k$</td>
<td>Observation model</td>
</tr>
<tr>
<td>$w_k$</td>
<td>Process noise</td>
</tr>
<tr>
<td>$v_k$</td>
<td>Observation noise</td>
</tr>
<tr>
<td>$Q_k$</td>
<td>Process noise covariance</td>
</tr>
<tr>
<td>$R_k$</td>
<td>Observation noise covariance</td>
</tr>
</tbody>
</table>

Table 2.1: The matrices required for modelling a process in accordance with the Kalman filter framework.

Figure 2.8: The resulting mask created by MOG \[24\] (centre) and the improved model with shadow detection \[25\] (right).

#### 2.5.3 Background Subtraction

Background subtraction is a fundamental computer vision technique for extracting the foreground of an image for further processing. The technique is especially suitable for detecting moving objects in video from static cameras, as the foreground objects can be detected through changes in pixel values from a reference frame (often called the background image) \[22\]. Problems that background subtraction algorithms must deal with are: changing weather, illumination changes, high-frequency repetitive motion such as tree leaves and flags etcetera, and long-term changes in the scene \[22\].

The most common methods of modelling the background image is by averaging the pixel values of a series of consecutive frames with or without a Gaussian average \[23\], and modelling the intensity value of every pixel as a Gaussian mixture model \[24\]. The Gaussian mixture model (MOG, Mixture-of-Gaussians) approach is the most used approach \[22\]. Newer models based on MOG are even capable of differentiating between foreground objects and their shadows \[25\]. An illustration of the binary image mask created by both the original MOG and one of the improved implementations is shown in Figure 2.8.

#### 2.5.4 Morphological Image Operations

Morphological algorithms play a large part in the field of filtering noise, boundary detection, and shape detection \[26\]. The most common approaches to removing noise from an input image is by using one pass of erosion followed by a pass of dilation (called opening) and dilation followed by erosion (closing).

For a binary image, the dilation operation is performed by scanning a kernel $K$ over an image, computing the maximal pixel value overlapped by $K$, and replacing the given pixel with that maximal value. For a binary image, this means that white shapes will grow, as illustrated in Figure 2.9.

---

7 Background subtraction result by Open Source Computer Vision Library licensed under BSD-3-Clause
8 Dilation by Open Source Computer Vision Library licensed under BSD-3-Clause
2.5. Visual Object Tracking

![Figure 2.9: Before (left) and after (right) applying the dilation operation to a binary image.](image)

![Figure 2.10: Before (left) and after (right) applying the erosion operation to a binary image.](image)

The erosion operation is very similar to dilation, with the difference being that the pixel value is replaced by the minimal pixel value overlapped by $K$. For a binary image, this means that white shapes will shrink, as illustrated in Figure 2.10.

2.5.5 Tracker Evaluation

While the quest for a general evaluation metric for multiple object trackers is ongoing, the CLEAR-MOT metrics has emerged as the standard measure $[27]$. The problem is finding a metric that can summarize the performance into one single number, so that it is easier to compare different trackers. By condensing the information into one number, we might lose some information about errors made by algorithms. Therefore, the trend has been to employ two sets of measures that are established in the literature: the CLEAR metrics proposed by Stiefelhagen, Bernardin, Bowers, Garofolo, Mostefa, and Soundararajan (2006) and a collection of quality measures introduced by Wu and Nevatia (2006), collectively referred to as the CLEAR-MOT metrics. The CLEAR-MOT metrics are used to in the MOT challenge that has been held yearly since 2015 $[27]$. The purpose of the MOT challenge is to benchmark multiple object trackers on the same data in order to help advance state-of-the-art in the tracking field. In this thesis, the data format used in the 2016 MOT challenge is used in order to calculate metrics.

For quantifying the performance we look at the output of the tracker, and determine whether the output accurately describes a target. The detection might be a true positive (TP) that describes an actual target, or it could be a false positive (FP) if it outputs a false alarm. The detection is classified as TP or FP by thresholding some measurement of distance $d$ between the ground truth and the hypothesis. If a target is missed by any hypothesis in the output, it is a false negative (FN). A good tracker will have few FPs and FNs, so these absolute numbers are included in the evaluation.

The optimal matching for a tracker is solved using the Hungarian algorithm. The matching is performed on multiple frames in order to achieve a temporal correspondence between the ground truth and the hypothesis. The definition is given by Milan, Leal-Taixé, Reid, Roth, and Schindler (2016): “if a ground truth object $i$ is matched to hypothesis $j$ at time $t - 1$ and the distance (or dissimilarity) between $i$ and $j$ in frame $t$ is below $t_d$, then the correspondence between $i$ and $j$ is carried over to frame $t$ even if there exists another hypothesis that is closer to the actual target. A mismatch error (or equivalently an identity switch, IDSW) is counted if a ground truth target $i$ is matched to track $j$ and the last known assignment was $k \neq j$. ” It is desirable to keep the number of ID switches low, however the evaluation procedure does

---

$[9]$ Erosion by Open Source Computer Vision Library licensed under BSD-3-Clause
not handle re-identification scenarios. When a target disappears from the field-of-view and then re-appears, it is treated as an unseen target and given a new ID. The guidelines for the MOT-metrics recommends interpreting the number of ID switches in relation to the number of recovered targets, because a tracker that detects more trajectories will naturally produce more identity switches. For this reason, the relative number of ID switches is included in the metric suite, computed as $\frac{\text{IDSW}}{\text{Recall}}$.

The distance measure $d$ used for evaluating tracking metrics is the intersect over union, which gives a measure of the overlap between the two bounding boxes for the ground truth and the hypothesis. The threshold value $t_d$ for the 2016 MOT benchmark is set to 0.5, which corresponds to a 50% overlap between bounding boxes.

The Multiple Object Tracking Accuracy (MOTA) is one of the most used metrics to evaluate the performance of a tracker [27]. By combining three sources of error, it captures multiple characteristics of a given tracker in a single number. The MOTA score is given by the formula:

$$
\text{MOTA} = 1 - \frac{\sum_t (\text{FN}_t + \text{FP}_t + \text{IDSW}_t)}{\sum_t \text{GT}_t} \tag{2.21}
$$

where $t$ is the frame index, and $\text{GT}$ is the number of ground truth objects. The MOTA can be negative in cases where the number of errors made by the tracker is greater than the number of all objects in the scene.

The Multiple Object Tracking Precision (MOTP) is a metric for the average dissimilarity between all the true positives and their corresponding ground truth targets. The MOTP metric is given by the formula:

$$
\text{MOTP} = \frac{\sum_{t,i} d_{t,i}}{\sum_t c_t} \tag{2.22}
$$

where $c_t$ is the number of matches in frame $t$, and $d_{t,i}$ is the bounding box overlap between target $i$ and its assigned ground truth object. Since MOTP is the average bounding box overlap for all correctly matched hypotheses, the score will be between $t_d = 50\%$ and $100\%$ for the MOT benchmark suite. The score gives an indicator for the localization accuracy for the tracker, but it provides very little information about the actual performance of the tracker [27].

There are also metrics for how well the ground truth trajectory is followed by the tracker. More precisely, a ground truth trajectory can be classified as mostly tracked (MT), partially tracked (PT), and mostly lost (ML). In order for a ground truth to be mostly tracked (MT), it must be successfully tracked for at least 80% of its life span. If the trajectory is followed for less than 20% of its life span, it is mostly lost (ML). Any track that falls between these two limits are said to be partially tracked (PT). The track quality measure does not regard whether the ID for the object is kept throughout its life span in order to classify how well the ground truth trajectory is followed.

Finally, the track quality can be partly quantified by the number of track fragmentations (FM) that occurs. Track fragmentations are the number of times a ground truth trajectory is interrupted and then resumed again at a later point. This occurs when the trajectory is first marked as tracked, then untracked, and then tracked again.

The CLEAR-MOT metrics that have been presented will provide a range of measurements that can be compared between trackers, but they do not tell us anything about a tracker’s ability to correctly identify an object when it is lost and reacquired. While the CLEAR-MOT metrics do track the number of ID switches, they do not reward switching back to the original “correct” ID. Arguably, this is an important quality to end-users: it is valuable to know if an object that appears is the same object that was seen a few seconds earlier in a wide array of use-cases. ID metrics were designed with this problem in mind: instead of solving the assignment problem on a per-frame basis, the ID metrics finds the minimum cost of objects and predictions over all frames [30]. This will generate different values for TP, FP, and FN, which
are used in the following equations that make up the ID metrics (identical to the equations in subsection 2.4.8, but with different definitions of TP, FP, and FN):

\[
P = \frac{TP}{TP + FP} \tag{2.23}
\]

\[
R = \frac{TP}{TP + FN} \tag{2.24}
\]

\[
F_1 = 2 \frac{PR}{P + R} \tag{2.25}
\]

where \( P \) is the definition of ID precision (IDP), \( R \) the definition of ID recall (IDR), and \( F_1 \) the \( F_1 \) score for a given tracker (IDF1). The three metrics can be interpreted in the following ways [30]:

- IDP: Fraction of computed detections that are correctly identified
- IDR: Fraction of ground-truth detections that are correctly identified
- IDF1: Harmonic average of ID precision and ID recall

Employing both the CLEAR-MOT together with the ID metrics in the evaluation of a multi object tracker will highlight different aspects of the tracker [30].
In this chapter, the approach for answering the research questions raised in chapter 1 is described. The approach towards building the system can roughly be divided into solving three problems: detecting potential airplanes in the video feed, tracking those objects, and attempting to classify whether a tracked object is an airplane or not during the track. The final pipeline of the system is presented in Figure 3.1.

Chosen methods are justified and described in detail, with a focus on how they fit into the framework as a whole and what purpose they serve. The selected object detector is also presented in this chapter, as our modified detector has an almost identical architecture - only the final layer differs between the two.

3.1 Frameworks, Platforms and Hardware

The prototype was written entirely in Python 3.5. Python lack support for multi-threaded applications using native threads, and might not seem suitable for a system where real-time performance is an important property. However, many libraries written in highly efficient C and C++ provides Python bindings. The broad range of supported libraries, together with the dynamic nature of Python, enables fast and iterative development of prototypes that can later be realized in robust applications for use in industry.

The CNNs were implemented in the Darknet framework, which utilizes GPU computation through the NVIDIA computing platform CUDA. The neural network was trained on a machine with a 1080 Ti graphics card with 11GB of memory. The development machine used for benchmarks also has a 7th generation Intel Core i7 Processor.

Most of the image processing was performed using the contrib version of the Open Source Computer Vision Library (OpenCV). This version of the library contains new modules that are not present in the official OpenCV library because they do not have stable APIs and are not as well-tested.

1Darknet framework available at pjreddie.com/darknet/
3.2 Preparing the Videos

Table 3.1: Source video information. The minimum number of cameras needed to capture the full trajectory of the airplane is included, leading to different video resolutions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Airport</th>
<th>Weather</th>
<th>Notes</th>
<th>Cameras</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE-MJJ</td>
<td>OER</td>
<td>Sunny</td>
<td>Turns on the runway</td>
<td>2</td>
<td>2160 × 1920</td>
</tr>
<tr>
<td>NTJ3102</td>
<td>OER</td>
<td>Sunny</td>
<td>Turns on the runway</td>
<td>4</td>
<td>4320 × 1920</td>
</tr>
<tr>
<td>CCIXT</td>
<td>SDL</td>
<td>Overcast</td>
<td>Turns on the runway</td>
<td>7</td>
<td>7560 × 1920</td>
</tr>
<tr>
<td>SE-MKA</td>
<td>SDL</td>
<td>Overcast</td>
<td>Turns on the runway</td>
<td>7</td>
<td>7560 × 1920</td>
</tr>
</tbody>
</table>

3.2 Preparing the Videos

The video dataset consists of multiple videos recorded from the camera feed for the Remote towers at Örnsköldsvik and Sundsvall–Timrå airport (airport codes OER and SDL respectively). The videos were provided by the Swedish air traffic service provider LFV. Combined, there were five recorded landings and departures; two landings from SDL, and two landings and one departure from OER. The recording for each event were originally divided into multiple files: one from each camera at the airport with a resolution of 1920 × 1080 per camera. Logistically, it would be challenging to keep the recordings in separate video files that were analysed in parallel, so the videos were ultimately merged into one using the horizontal stacking filter available in the open source FFmpeg software suite. Only the cameras that featured the airplane at some time during the event were included in the merged video; this differed between videos because smaller aircraft did not require the entire runway for landing. The video from each camera had to be rotated 90 degrees before being stacked, resulting in the rearranging of the resolution to 1080 × 1920 for each camera.

The process resulted in five videos - one for each event - of which the one video of a departure was excluded. The final videos were named after the aircraft that was featured in them, and screen captures of the different aircraft can be seen in chapter 4. A description of each video along with how many camera feeds were horizontally stacked is given in Table 3.1. The horizontal resolution for the merged videos can be calculated by multiplying the horizontal resolution for each camera with the number of cameras: Resolution\textsubscript{h} = 1080 · N, where N is the number of cameras.

The videos featured landings and departures made under different weather, with the videos from OER being recorded on sunny days, and the SDL videos being cloudy. All videos were recorded during winter with a snow-covered background environment, with most of the airport tarmac cleared of snow.

3.3 Detecting Aircraft

Detecting all objects of interest (in our case aircraft) is a non-trivial task considering the sheer size of the video feed that is to be processed. A naive solution would be to simply capture a frame from the video feed and send it through an object detection algorithm that in theory should return an array of all detected objects. This solution will not work in practice because of the fixed-size input dimensions of the object detection CNN: if we used YOLOv2, the frame would be scaled down to 416 × 416 pixels before being sent through the network, meaning that incoming aircraft might end up not even be represented by a single pixel in the input matrix [11]. Ideally we would pass all the pixels belonging to an approaching airplane through the CNN, increasing the likelihood of a detection. Down-sampling to 416 × 416 should ideally only occur if the bounding box of the airplane has greater dimensions than what fits in the CNN input.

Another solution would be using the sliding-window technique, where the video feed is divided into patches that are sequentially passed to the CNN, and detections aggregated once the entire frame has been passed through. This approach would undoubtedly work better.
since no spatial information is lost; several object detection algorithms have previously been based on this approach \[32\]. However, since this operation is performed sequentially, the application would not be able to function in real-time as one forward pass through a CNN requires approximately 6ms \[31\]. The frame can at most be divided into 5 or 6 patches before the latency would cause the application to not operate in real-time for a video running at 30 frames per second, excluding any other operations that would also require computing time.

A third solution is possible because of the characteristics of the remote tower video feed: since the video feed comes from stationary cameras, it is possible to perform basic background subtraction and thus in theory detect all foreground objects. The foreground objects will then become candidates, meaning that they might be airplanes that we want to track. In order to detect if a candidate is an airplane, the system can then crop out of the candidate from the frame and pass only the cropped image through the object detection network in order to detect its class. Very little, if any, spatial information is lost using this approach, increasing the probability of the CNN successfully predicting the right class. By tracking the objects, it would also enable us to increase our certainty once an airplane has been detected by cumulating successive detections on the same object. This is the solution that was implemented in the application.

The flowchart of the solution is shown in Figure 3.1. The foreground objects is the set of objects that were detected in a given frame, while the candidates is the set of objects that are being tracked. The foreground objects makes up the measurements that are fed to the Kalman filters of the candidates. Object classification is performed for each candidate on an individual basis depending on how long it has been since the last classification attempt.

3.3.1 Detecting Tracking Candidates

In order to detect foreground objects we use a background subtraction algorithm. While MOG is the most widely used algorithm for this problem \[22\], early tests showed that the computations required by the MOG implementations did not scale well with the large video resolution of our source videos. Furthermore, the shadow detection for the improved MOG algorithm interfered when the airplane was far away in the video feed.

Another, more rudimentary background subtraction algorithm was chosen for its simplicity together with performance comparable to that of the MOG solution: The BackgroundSubtractorCNT (CNT stands for count) algorithm, which is a background subtraction algorithm based on computationally inexpensive operations. The algorithm implements frame differencing between two consecutive frames by applying a threshold to each pixel of the differential image. Pixels that fall below the given threshold multiple frames in a row are eventually marked as background, and all other pixels are marked as foreground. Pixels that are marked as background multiple times will increment a counter up until a maximum value, finally marking the pixel as stable. Changes in a stable pixel will reset the counter, and mark it as foreground.

The algorithm also offers a “history” functionality for each pixel, where the colour value of pixels that are marked as stable for a long time will be stored. Whenever the algorithm is called, the current pixel will first be compared to the historically stable pixel colour instead of the previous pixel colour: if the difference is below a certain threshold, the pixel will eventually be marked as background. If the pixel does not match the historical colour value the normal algorithm is applied, with the slight difference that a historically stable pixel value is selected.

Before applying the subtraction algorithm to the frame, a slight blur using a normalized box filter with filter size 3 x 3 is applied using convolution in order to smooth out the image. The resulting masks from both the BackgroundSubtractorCNT and the improved MOG2 algorithm are showed in Figure 3.2.

After the background subtraction algorithm has been applied to the entire frame, we denoise the resulting mask using erosion. The erosion step is then followed by 20 iterations of dilation, with a typical shape for the airplane in the resulting mask seen in Figure 3.3. The resulting shapes are then extracted from the binary image using the findContours method in the OpenCV.
3.3. Detecting Aircraft

Figure 3.1: Flowchart for the proposed system. Foreground objects are extracted from a frame, and fed as measurements to the Kalman filter of so-called candidates. If no measurement is available, a candidate will attempt to predict its own current state based on prior measurements and a motion model. Finally, periodic classification attempts are performed individually for each candidate.
Figure 3.2: Background subtraction mask comparison on SE-MJJ, with BackgroundSubtractionCNT shown left and the improved MOG algorithm showed right. The improved MOG algorithm is able to differentiate the reflection in the ice (masked with grey colour, bottom right) from the actual airplane. In the top right frame, the MOG shadow detection method is interfering with the actual shape of the object.

Figure 3.3: Detail of a resulting shape after one iteration of erosion, and 20 iterations of dilation on the background subtraction mask.
library, which extracts the contours of foreground objects using the algorithm presented in \[33\]. For each extracted contour, we extract a bounding box using `boundingRect`, and finally store all the bounding boxes in an array.

With the foreground extraction we have performed, we do not yet differentiate between airplanes and other objects, hence calling the foreground objects “candidates”. In order to classify whether a foreground object is an airplane or not, we will track the object and attempt to classify it at regular intervals using a convolutional neural network trained to recognize airplanes.

### 3.3.2 You Only Look Once 2

You Only Look Once version 2 (YOLOv2) is an open source real-time object detection method that obtains similar accuracy to other state of the art detectors on the VOC 2007 benchmark, while running significantly faster at 67 FPS \[31\]. YOLOv2 is unique in that it was created by training an object classification model named Darknet-19 on the ImageNet dataset containing 1000 classes, and then "repurposing" the neural network as an object detector. The Darknet-19 architecture can be seen in Table 3.2. Features are repeatedly extracted using 3×3 convolutional layers increasing the depth dimension, before compressing those features in the same dimension with 1×1 convolutional layers, and then downsampling the spatial dimensions by a factor of two in pooling layers.

Finally, a convolutional layer with 1000 filters of size 1×1 are fed into a softmax layer that provides class probabilities over the 1000 classes. Older architectures would employ fully-connected layers with 1000 neurons before the softmax activations, but in the Darknet-19 architecture we can see in practice the technique for replacing fully-connected layers that was described in subsection 2.4.3.

After training the classification model, the authors removed the final convolutional layer while keeping the network parameters. They then replaced the final layer with three 3×3 convolutional layers each containing 1024 filters, and a final 1×1 convolutional layer with the number of filters that was needed for detection. The final four layers for YOLOv2 are shown in Table 3.3.

What makes YOLOv2 so fast is the fact that it uses a custom network based on the Googlenet architecture \[31\], which utilizes 1×1 filters to compress the depth dimension which in turn reduces the number of operations required for a forward-pass \[34\].

Other network architectures uses fully-connected final layers to predict bounding boxes, which means that spatial information is lost. YOLOv2 takes another approach by predicting bounding boxes with hand-picked priors, so called anchor boxes. The actual prediction performed by the network is the coordinates, size, and confidence of a bounding box based on these anchor boxes. The intuition is that common objects will have similar bounding box dimensions; for example will pedestrians have a thin and tall bounding box, while a car will have a wide and short bounding box. By selecting good priors, it will be easier for the network to learn to predict good detections. This approach simplifies the problem and makes it easier to train the detector since the class prediction mechanism and the spatial location are decoupled: class probabilities (for every class) and objectness (how likely the box contains an object) are predicted for every anchor box. The objectiveness score is a score of how confident the model is that the box contains an object, and also how accurate it thinks the box is that it predicts. Formally, the confidence is defined as $P_r(\text{Object}) \times \text{IOU(}\text{truth, prediction)}$ \[35\].

The predictions are performed on a 13×13 feature map, which might not be large enough in the spatial dimension for predicting small objects. Therefore, a pass-through layer is added which brings features from an earlier 26×26 layer, and concatenates the higher resolution features with the low resolution features in the depth dimension. This is done by stacking the adjacent features of the 26×26 layer into different channels, instead of spatial locations. In effect, this transform the 26×26×512 feature map into a 13×13×2048 feature map that can be concatenated with the 13×13 feature map in the final layer.
### 3.3. Detecting Aircraft

<table>
<thead>
<tr>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional</td>
<td>32</td>
<td>3 × 3</td>
<td>224 × 224</td>
</tr>
<tr>
<td>Maxpool</td>
<td>2 × 2/2</td>
<td></td>
<td>112 × 112</td>
</tr>
<tr>
<td>Convolutional</td>
<td>64</td>
<td>3 × 3</td>
<td>112 × 112</td>
</tr>
<tr>
<td>Maxpool</td>
<td>2 × 2/2</td>
<td></td>
<td>56 × 56</td>
</tr>
<tr>
<td>Convolutional</td>
<td>128</td>
<td>3 × 3</td>
<td>56 × 56</td>
</tr>
<tr>
<td>Convolutional</td>
<td>64</td>
<td>1 × 1</td>
<td>56 × 56</td>
</tr>
<tr>
<td>Convolutional</td>
<td>128</td>
<td>3 × 3</td>
<td>56 × 56</td>
</tr>
<tr>
<td>Maxpool</td>
<td>2 × 2/2</td>
<td></td>
<td>28 × 28</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>3 × 3</td>
<td>28 × 28</td>
</tr>
<tr>
<td>Convolutional</td>
<td>128</td>
<td>1 × 1</td>
<td>28 × 28</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>3 × 3</td>
<td>28 × 28</td>
</tr>
<tr>
<td>Maxpool</td>
<td>2 × 2/2</td>
<td></td>
<td>14 × 14</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>3 × 3</td>
<td>14 × 14</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>1 × 1</td>
<td>14 × 14</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>3 × 3</td>
<td>14 × 14</td>
</tr>
<tr>
<td>Convolutional</td>
<td>256</td>
<td>1 × 1</td>
<td>14 × 14</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>3 × 3</td>
<td>14 × 14</td>
</tr>
<tr>
<td>Maxpool</td>
<td>2 × 2/2</td>
<td></td>
<td>7 × 7</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>7 × 7</td>
</tr>
<tr>
<td>Convolutional</td>
<td>512</td>
<td>1 × 1</td>
<td>7 × 7</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>7 × 7</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>7 × 7</td>
</tr>
<tr>
<td>Avgpool</td>
<td>1000</td>
<td>1 × 1</td>
<td>7 × 7</td>
</tr>
<tr>
<td>Softmax</td>
<td></td>
<td>Global</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 3.2: Darknet-19: The object classification model that forms the basis for YOLOv2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>13 × 13</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>13 × 13</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>13 × 13</td>
</tr>
<tr>
<td>Convolutional</td>
<td>425</td>
<td>1 × 1</td>
<td>13 × 13</td>
</tr>
</tbody>
</table>

Table 3.3: The final four layers that were added to the modified Darknet-19 architecture to create YOLOv2. Note that the Darknet-19 architecture operates with an input size of 224 × 224, while YOLOv2 uses 416 × 416, resulting in a spatial size of 13 × 13 for the final layers instead of 7 × 7.
3.3. Detecting Aircraft

Each of the spatial coordinates in the final $13 \times 13$ feature map is called a "cell". If the centre of an object falls into a cell, that cell is responsible for predicting that object. Each cell uses five anchor boxes each in order to predict the object, which can all have separate scaling and coordinates. The centre of the bounding box is therefore restricted to the cell, but the bounding box size is not constrained to the spatial dimensions of the cell: it can overlap with other cells.

The coordinates for each anchor box are given as follows: $t_x, t_y, t_w, t_h,$ and $t_o$, where $t_x$ and $t_y$ are the centre coordinates of the bounding box relative to the cell, $t_w$ is the anchor box width scaling exponent, $t_h$ is the anchor box height scaling exponent, and finally $t_o$ is the objectiveness score of the bounding box. A visualization of how the relative coordinates are finally converted to bounding box coordinates is shown in Figure 3.4. The final coordinates $b_x$ and $b_y$ are calculated by passing the $t_x$ and $t_y$ coordinate through the sigmoid function $S(x) = \frac{e^x}{1 + e^x}$ which outputs values between 0 and 1. This ensures that a bounding box will be constrained to a cell, as previously mentioned giving the cell "ownership" over that bounding box.

In addition to the anchor boxes, each cell calculates $C$ conditional class probabilities $P_i(Class_i | Object)$; this number is independent of the number of boxes belonging to a cell. At test time these conditional class probabilities are multiplied by the individual box confidence predictions, giving class-specific confidence scores for each box. These scores provide the probability of a class appearing in the box, and how well the predicted box fits the object. Non-max suppression is then applied to the predictions, finding overlapping bounding boxes belonging to different cells with an IOU score over 0.5 and ignoring (suppressing) the bounding box with the lowest score.

If we want to be able to detect any of the 20 classes in the VOC benchmark, each grid cell needs $5 \times 5$ outputs for the bounding box coordinates (five boxes with five coordinates each), and $5 \times 20$ outputs for the class probabilities, totalling 125 outputs. Since we have $13 \times 13$ grid cells in total, the final output from the network must have the dimensions $13 \times 13 \times 125$. This means that the final convolutional layer we added with size $1 \times 1$ must contain 125 filters.

During training, the YOLOv2 network changes the input image size in order to make it robust on different image sizes, varying it between $320 \times 320$ and $608 \times 608$ in multiples of 32. The ability to change the input resolution on the fly is a result of the network only consisting of convolutional and pooling layers.
In order to effectively train the network, an effective loss function was proposed in [35]. During training, the loss for a true positive requires that only one bounding box is provided per object. Therefore, the bounding box with the highest IOU with the ground-truth is chosen for calculating the loss. The loss function is the sum-squared error (SSE) of multiple properties between the prediction and the ground-truth; these properties are: the classification loss, the localization loss, and the confidence loss. The classification loss is the SSE of the class conditional probabilities of each class. The localization loss is the SSE of the predicted boundary box location and size. The confidence loss is the SSE of the objectness of the box. Since it is not necessary to modify the loss function unless performing major changes on the YOLOv2 architecture, the reader is referred to [35] for a more thorough description of the loss function.

All code relating to the YOLOv2 detection system has been released as open source code in the neural network framework Darknet, and is therefore a natural choice for incorporating the system in the application. The Darknet framework is written in C, and there exists bindings for Python with a very simple interface in the ppyolo wrapper. Since ppyolo is a simple wrapper for Darknet, it allows us to completely replace the underlying detection model for our application by providing trained model weights and network architecture configuration files. This allows us to train custom detectors on the Darknet framework, and access the detector from our Python application without any major code changes.

3.3.3 Arlanda Dataset

The initial dataset consisted of 3000 photographs of aircraft from jetphotos.com, where users have submitted their own pictures of aircraft. Only photos of aircraft taken at Arlanda airport were used in order to ensure that the dataset was specialized in the kinds of aircraft that are currently in service in Northern Europe. All 3000 photographs of airplanes were manually labelled, employing a strategy where only the fuselage of the aircraft was marked by a bounding box. This labelling strategy differs from more conventional labelling strategies of aircraft that is employed in the datasets for the COCO [15] and VOC [14] challenges. These datasets feature labels of the entire aircraft, including wings and vertical stabilizer. While these entities are a factor in making aircraft visually distinct from other objects, including them in the bounding box means that a large part of the background will be included, as illustrated in Figure 3.5.

Even slight yawing, rolling, and pitching of an aircraft will significantly transform the appearance of the aircraft when these parts are included in the detection. If the CNN is only trained to detect the fuselage, the hypothesis is thus that the CNN will be more robust to airplane maneuvers and require fewer training samples for detecting aircraft from different angles.

During early training of the CNN, it was detected that only training the classifier on photographs of airplanes led to many false positives in practice, as detailed later in 3.3.5. This was an accuracy paradox [36] caused by an imbalanced training dataset: since the training set only consisted of images of airplanes, the CNN could easily guess that an area in each image contained an aircraft and obtain a high accuracy score. In order to remedy this, the dataset was extended with 3000 randomly chosen images from the 2012 VOC dataset that does not feature any airplanes, which balanced out the dataset. This gives a total of 6000 images, which was divided into a training set and a validation set with 1-10 split: 600 images in the validation set, and 5400 images in the training set. By balancing out the training dataset, we would reduce the number of false positives during detection.

The Darknet framework enables automating common data augmentation tactics such as random image crops, rotations and hue, saturation, and exposure shifts. The dataset is augmented by copying each image and randomly distorting the copy using these attributes, which

---

2 ppyolo available at github.com/digitalbrain79/ppyolo
3 Norwegian Boeing 787 G-CIXO by MercerMJ licensed under CC BY-SA 2.0
3.3. Detecting Aircraft

Figure 3.5: The labelling strategy employed for the Arlanda dataset (top) versus the labelling strategies employed in other datasets (bottom). Only the airplane fuselage is covered by the bounding box for the Arlanda dataset, resulting in less of the background being included.

<table>
<thead>
<tr>
<th>Angle</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturation</td>
<td>1.5</td>
</tr>
<tr>
<td>Exposure</td>
<td>1.5</td>
</tr>
<tr>
<td>Hue</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 3.4: Configuration values used for data augmentation.

can be specified in a configuration file. The values used in the application were kept identical to the values used for training the original YOLOv2 detector, as shown in Table 3.4.

The distortion is performed in the hue-saturation-value (HSV) colour space. With these values the hue value is increased or decreased by a number chosen randomly from a uniform distribution between $-0.1$ and $0.1$, while the exposure and saturation are scaling numbers between $\frac{1}{1.5}$ and $1.5$. Since the data augmentation step in Darknet creates a copy of each image in the original dataset, the final dataset used to train the detector consists of 10,800 images. The validation set does not undergo any augmentation, and therefore contains 600 images.

3.3.4 The Custom Object Detector

The following detectors will be benchmarked with the prototype:

1. A generic detector (YOLOv2)
2. A custom detector trained on aircraft from the Arlanda dataset

The object custom object detector is a variant of the YOLOv2 CNN, trained on the Arlanda dataset. While the new detector does not differ noticeably from the YOLOv2 architecture, we name it ArlandaNet in order to distinguish it from the pre-trained YOLOv2 detector. Just as the YOLOv2 network, the custom object detector is based on the Darknet-19 classification model that is available for the Darknet framework. The layout of the Darknet-19 classifier
### 3.4 Tracking Objects of Interest

<table>
<thead>
<tr>
<th>Type</th>
<th>Filters</th>
<th>Size/Stride</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>13 × 13</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>13 × 13</td>
</tr>
<tr>
<td>Convolutional</td>
<td>1024</td>
<td>3 × 3</td>
<td>13 × 13</td>
</tr>
<tr>
<td>Convolutional</td>
<td>30</td>
<td>1 × 1</td>
<td>13 × 13</td>
</tr>
</tbody>
</table>

Table 3.5: The final three layers of ArlandaNet

can be seen in Table 3.2. A pre-trained model is available for the Darknet-19 classifier, so transfer learning was utilized in order to shorten the time needed for training the two custom detectors.

The transfer learning was performed through the same process as detailed in subsection 3.3.2. Darknet-19 was modified in order to use the network for detection, by removing its final convolutional layer and replacing it with three 3 × 3 convolutional layers (1024 filters each), and a final 1 × 1 convolutional layer with the number of outputs required for detection. The number of outputs is dependent on how many classes are to be detected (C), since we predict 5 boxes with 5 coordinates each, and C classes per box. The number of outputs can therefore be calculated by the formula \( n = (C + 5) \times 5 \) where C is the number of classes to be detected:

- ArlandaNet, (1 class): 30 outputs
- YOLOv2 (80 classes): 425 outputs

The final three layers of ArlandaNet are shown in Table 3.5.

#### 3.3.5 Training the Detector

The custom aircraft detector was trained for 78750 batches of 64 images each, corresponding to 466 epochs. The CNN reached an average loss of 8% on the final 100 batches. Each batch required around 2 seconds, giving a total estimated training time of 44 hours on one GPU. After training approximately 10,000 batches on the dataset containing only images of aircraft, it became apparent from that the detector was frequently misclassifying objects as aircraft (false positives) in practice. In order to mitigate this problem, the size of the dataset was doubled with 3000 additional images randomly selected from the VOC 2012 dataset, as described in subsection 3.3.3. The training was continued from the same point, and the impact on the average loss can clearly be seen around batch 10,000 in Figure 3.6.

The reasoning for not completely restarting the training at when this occurred was time considerations, and the fact that the CNN was successfully detecting aircraft in practical testing at this point; it was only the false positives that were problematic. The average loss during training is just an indicator of how the training is going; the final accuracy of the detector is later quantified by evaluating it on the test dataset.

#### 3.4 Tracking Objects of Interest

Given a video feed from a remote virtual tower, the task that the prototype should achieve can be summarized with the following algorithm:

1. Detect all objects of interest
2. Track each object until they go out of frame or become stationary

*Available for download at https://pjreddie.com/darknet/imagenet/#pretrained*
3.4. Tracking Objects of Interest

Figure 3.6: Logarithmic graph of the average loss for ArlandaNet. Around batch number 10,000 the dataset was augmented with images containing no aircraft, causing a spike in the average error.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOSTING [37]</td>
<td>Did not handle occlusions or turns on the runway.</td>
</tr>
<tr>
<td>MIL [38]</td>
<td>Very low FPS. Did not handle occlusions well.</td>
</tr>
<tr>
<td>KCF [39]</td>
<td>Lost track during occlusions and when planes performed turns on the runway.</td>
</tr>
<tr>
<td>TLD [40]</td>
<td>Very low FPS. Had major difficulties tracking the airplane at most points.</td>
</tr>
<tr>
<td>MEDIANFLOW [41]</td>
<td>Failed to track airplane at any point during landing.</td>
</tr>
<tr>
<td>MOSSE [42]</td>
<td>Did not handle occlusions or turns. Bounding box had a slight drift</td>
</tr>
</tbody>
</table>

Table 3.6: A selection of trackers from the OpenCV contrib module.

We have detailed how foreground objects (candidates) are detected, and now the task is to track each candidate.

3.4.1 Surveying Existing Trackers

In order to get an initial indicator of how well existing trackers would work on the videos, a small application was created for simple testing on the source videos. Most trackers follow an API with one method for initializing the tracker by providing a bounding box, and one method for updating the tracker (init and update respectively). The entire frame is passed to update, which in turn returns the estimated bounding box for that frame.

The application enabled manually selecting a bounding box that would be provided to the trackers for initialization, and passing the current video frame to the update method as the video played back. Whenever the tracker under test lost track of the airplane, the application could be paused in order to manually re-select it. The trackers that were tested are presented in Table 3.6.

It is important to note that these tests were performed with the default parameters set in the OpenCV contrib library. The results might not reflect the best possible performance possible by these trackers, but they do indicate that there are challenges with using trackers based on appearance models for the video footage from remote tower feeds. The trackers are designed to be general trackers, and they all learn some appearance model of the aircraft, as described in subsection 2.5. The videos proved challenging for the trackers, because of
3.4. Tracking Objects of Interest

3.4.2 Selecting a Tracker

After the survey detailed in subsection 3.4.1 it became apparent that none of the generalized trackers performed well enough for the problem: the airplanes coming in for landing would start very far away, making it difficult for the trackers to create a solid appearance model. Furthermore, most trackers struggled with occlusion and following turns performed by aircraft. Because of the shortcomings of the surveyed trackers, it was decided to construct a tracker inspired by the SOR T tracking algorithm \[43\]. The major differentiating factor of the approach used by the SOR T algorithm is that it does not generate any appearance model.

It was not possible to properly test the SOR T algorithm due to its reliance on frequent detections, but the ideas behind the tracker are very clear. SOR T uses two classical, yet highly efficient methods: the Kalman filter and the Hungarian method. The Kalman filter is used for handling motion prediction, while the Hungarian method is used for assigning detected objects to already tracked objects (tracklets): a batch of foreground object predictions is fed to the tracking manager, which uses the Hungarian method to assign detections to existing Kalman filters based on the IoU between the detection and the Kalman filter state variables. The IoU needs to be at least 0.3 in order for the detection to be assigned to a Kalman filter. Any unmatched detections will result in a new tracker based on the detection bounding box. The update method on each Kalman filter that was associated with a detection will be invoked in order to update the state variables with the new measurements. Thus, the tracker differs from the other state of the art trackers in that it ignores appearance features other than the predicted bounding box for an object.

The consequence of ignoring appearance features when tracking is that either the motion model needs to be highly accurate, or the detections need to be accurate and provided frequently, or ideally both. The simplicity of the framework enables extremely fast tracking, with the requirement that it is frequently supplied with accurate detections. Furthermore, the tracker does not handle long-term occlusion or re-entry: handling of these cases are purposefully excluded from the framework in order to avoid adding overhead that limits its usage in real-time applications. With these properties, the architecture of the SOR T algorithm is suitable as a starting point for the prototype.

3.4.3 Designing the Kalman Filter

While the SOR T algorithm is based on a Kalman filter that assumes constant velocity, such a model does not work very well for approximating the position of rapidly decelerating aircraft that we observe at our two airports. Therefore, the tracker used in the application used a completely redesigned underlying Kalman filter to better approximate the characteristics of the motion of airplanes in the video feed.

The sensor readings provided to the Kalman filter in regular time intervals is a bounding box in 2D space extracted from the video feed, with the values \(x, y, \) box scale, and box ratio. Since the video feed runs at 30 frames per second, the discrete time step between each state transition becomes \(\Delta t = \frac{1}{30} s \approx 0.0333 s\).

State Variables

We have chosen nine state variables in order to keep track of the aircraft, of which the foreground extraction approach provides direct measurements for four. The state variables for the filter are described in Table 3.7. The state vector is ordered by dimension, and is shown in equation 3.1:

\[
x = [x \ \dot{x} \ \ddot{x} \ y \ \dot{y} \ \ddot{y} \ s \ \dot{s} \ r]^\top.
\]
3.4. Tracking Objects of Interest

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>The centre x-coordinate of the bounding box</td>
</tr>
<tr>
<td>$\dot{x}$</td>
<td>The horizontal speed of the bounding box</td>
</tr>
<tr>
<td>$\ddot{x}$</td>
<td>The horizontal acceleration of the bounding box</td>
</tr>
<tr>
<td>$y$</td>
<td>The centre y-coordinate of the bounding box</td>
</tr>
<tr>
<td>$\dot{y}$</td>
<td>The vertical speed of the bounding box</td>
</tr>
<tr>
<td>$\ddot{y}$</td>
<td>The vertical acceleration of the bounding box</td>
</tr>
<tr>
<td>$s$</td>
<td>The bounding box scale (width $\cdot$ height)</td>
</tr>
<tr>
<td>$\dot{s}$</td>
<td>The bounding box scale rate of change</td>
</tr>
<tr>
<td>$r$</td>
<td>The bounding box ratio (width $/$ height)</td>
</tr>
</tbody>
</table>

Table 3.7: State variables for the Kalman filter. All variables are in relation to the camera axis of the horizontally stacked video feeds.

Since the velocities and accelerations are unobservable, the initial velocities and accelerations are very uncertain. This is reflected in the initial covariance matrix $P$, where all unobserved variables have been given a very high initial variance. The covariance matrix will be continuously updated as sensor readings are provided during the lifetime of the Kalman filter.

$$
P = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 10000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 10000 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 10000 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 10000 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 100000 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 
\end{bmatrix}
$$

**State Transition Matrix**

The state transition matrix is based on Newton’s equations of motion. A motion model will always be a simplification of the real-world behaviour of a system. It is of great importance to our system that we accurately predict the trajectory for an airplane, but there is no major benefit in modelling the size and ratio of the bounding box. In order to simplify the overall architecture of the Kalman filter, we do not make any attempts at modelling the bounding box characteristics in great detail. In practice this means that we will rely heavily on measurements for the bounding box size and ratio. The following simplifications have been made in order to model the physical system:

- The bounding box has constant acceleration in the vertical and horizontal directions
- The bounding box scale rate of change is constant
- The bounding box ratio is constant
The recursive state transition equations reflecting these simplifications over a time step $\Delta t$ are presented in equation 3.3:

\[
\begin{align*}
    x_k &= x_{k-1} + \dot{x}_{k-1}\Delta t + \frac{1}{2}\ddot{x}_{k-1}\Delta t^2 \\
    y_k &= y_{k-1} + \dot{y}_{k-1}\Delta t + \frac{1}{2}\ddot{y}_{k-1}\Delta t^2 \\
    s_k &= s_{k-1} + \dot{s}_{k-1}\Delta t \\
    r_k &= r_{k-1} \\
    \ddot{x}_k &= \ddot{x}_{k-1} + \dddot{x}_{k-1}\Delta t \\
    \dot{y}_k &= \dot{y}_{k-1} + \ddot{y}_{k-1}\Delta t \\
    \ddot{s}_k &= \ddot{s}_{k-1} \\
    \dot{r}_k &= \dot{r}_{k-1} \\
    x_k &= x_{k-1} \\
    y_k &= y_{k-1} \\
\end{align*}
\] (3.3)

With the state vector $x$ presented in equation 3.1, the resulting linearised state transformation matrix is presented in equation 3.4:

\[
F = \begin{bmatrix}
1 & \Delta t & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & \Delta t & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\] (3.4)

**Observation Noise**

The uncertainty in the sensor measurement can be estimated by utilizing the fact approximately 95% of all values in a Gaussian distribution falls within two standard deviations ($\pm 2\sigma$) of the mean \[44\]. We can then express the numbers in terms of how far from the measurement mean we believe 95% of the measurement values will range. For example, if the measurements of the horizontal position $x$ of an object has the Gaussian $\mu = 500$ pixels and a standard deviation of 10, 95% of the measurements will range from 480 to 520 pixels.

For our Kalman filter we assume that 95% of the horizontal position measurements falls within $\pm 20$ pixels, 95% of the vertical position measurements falls within $\pm 10$ pixels, and that the size $s$ measurement falls within $\pm 44$ pixels. The assumptions are shown in equation 3.5:

\[
\begin{align*}
    2\sigma_x &= 20 \\
    2\sigma_y &= 10 \\
    2\sigma_s &= 44 \\
\end{align*}
\] (3.5)

The observation noise matrix for the application was created with the assumption that the noise for the observed variables $x$, $y$, and $s$ are independent, meaning that they have no covariance. This gives the final diagonal observation noise covariance matrix:

\[
R = \begin{bmatrix}
100 & 0 & 0 & 0 \\
0 & 25 & 0 & 0 \\
0 & 0 & 484 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\] (3.6)

**The Process Noise Covariance Matrix**

Designing the $Q$ matrix is not as straightforward as the $P$ matrix. We have defined a process model that we know is incomplete, and therefore the $Q$ matrix will also be incomplete. By
having a too small $Q$, the Kalman filter will be overconfident in the prediction model, while a too large $Q$ will cause the Kalman filter to be heavily influenced by measurement noise. A core part of designing the Kalman filter is therefore to design a good $Q$ matrix. We do so by keeping all other provided matrices such as the $R$ matrix fixed, and tweaking the $Q$ matrix fields to work well with our typical measurement data.

The process noise for $x$ and $y$ is represented using a continuous white noise model, which assumes that the acceleration is constant over each time period $\Delta t$. The process noise for $s$ is represented using a discrete white noise model, which holds the assumption that the acceleration is uncorrelated between time periods. These noise models have shown to perform well, and they give the advantage of expressing the white noise using a single number. The final $Q$ matrix will be constructed from the individual noise covariance matrices, which are generated using the filterpy library using the functions $Q_{discrete\_white\_noise}$ and $Q_{continuous\_white\_noise}$. The functions have in common that they require the number of dimensions and the delta between each time step. They differ in that the continuous white noise is calculated using spectral density, while the discrete white noise is generated from the noise variance. This gives us three parameters that need to be defined in order to create a process noise matrix: $\sigma_s$, $\Phi_x$, and $\Phi_y$, where $\sigma_s$ is the noise variance of scale measurements, and $\Phi_x$ and $\Phi_y$ are the spectral densities for the horizontal and vertical position. Each of these parameters had to be experimentally tuned in order to fit the characteristics of the application; this was done by running the tracker on the video footage with the described configuration and iteratively adjusting the process noise matrix. The final parameters for the process noise matrix are presented in equation (3.7).

\[
\Phi_x = 0.35 \\
\Phi_y = 0.05 \\
\sigma_s = 850 \\
\text{(3.7)}
\]

While somewhat unintuitive, the spectral densities $\Phi_x$ and $\Phi_y$ can be interpreted in relation to each other: the airplanes hold a much higher horizontal speed than vertical speed, and therefore the absolute horizontal acceleration will also be greater when landing. This means that the Kalman filter will most likely have an easier time predicting $y$, $\dot{y}$, and $\ddot{y}$ than any of the corresponding horizontal state variables. This characteristic is reflected in $\Phi_x$ being 7 times greater than $\Phi_y$ in the chosen parameter values.

$\sigma_s$ was chosen by estimating that approximately 95% of the predicted values would be within an area of 1700 pixels (corresponding to an area of approximately 41 × 41 pixels with ratio 1) of the true value for $s$. In effect, this means that the Kalman filter will favour the observed value for $s$ over the predicted value. An example where this is balance is especially desirable is when an airplane is turning on the runway, and the bounding box in a short amount of time must adjust to fit the entire wingspan of the aircraft.

The final $Q$ matrix is the result of concatenating each of the individual noise matrices:

\[
Q = \begin{bmatrix}
\Phi_x \frac{\Delta t^5}{5!} & \Phi_x \frac{\Delta t^4}{4!} & \Phi_x \frac{\Delta t^3}{3!} & 0 & 0 & 0 & 0 & 0 & 0 \\
\Phi_x \frac{\Delta t^4}{4!} & \Phi_x \frac{\Delta t^3}{3!} & \Phi_x \frac{\Delta t^2}{2!} & 0 & 0 & 0 & 0 & 0 & 0 \\
\Phi_x \frac{\Delta t^3}{3!} & \Phi_x \frac{\Delta t^2}{2!} & \Phi_x \frac{\Delta t^1}{1!} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \Phi_y \frac{\Delta t^5}{5!} & \Phi_y \frac{\Delta t^4}{4!} & \Phi_y \frac{\Delta t^3}{3!} & 0 & 0 & 0 \\
0 & 0 & 0 & \Phi_y \frac{\Delta t^4}{4!} & \Phi_y \frac{\Delta t^3}{3!} & \Phi_y \frac{\Delta t^2}{2!} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \Phi_y \frac{\Delta t^3}{3!} & \Phi_y \frac{\Delta t^2}{2!} & \Phi_y \Delta t & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \sigma^2_s \frac{\Delta t^4}{4!} & \sigma^2_s \frac{\Delta t^3}{3!} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \sigma^2_s \frac{\Delta t^3}{3!} & \sigma^2_s \Delta t^2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \\
\text{(3.8)}
\]
3.4.4 Handling Occlusions

Airplanes might become occluded during tracking - that is, something obstructs the view of the camera so that the airplane is no longer visible while still in frame. By design, the remote tower cameras are mounted in such a way that the amount of occlusion is minimal. However, some occlusion will inevitably occur that poses a challenge for the tracker. The main causes of occlusion are masts present at the runway, along with difficult lightning conditions.

While the original SORT tracker does not handle occlusions at all \[43\], we require this feature for our application to function properly. In order to handle occlusion, we allow the Kalman filter for each tracker to exclusively use the predict up to five seconds of video time without being provided any new measurements. This means that the motion model is trusted fully for those five seconds. If any measurement bounding box that has a IoU with the predicted box which is higher than 0.3, it is assigned to the Kalman filter and tracking proceeds as normal. If the tracker fails to assign a foreground object detection to a tracklet’s Kalman filter during those 5 seconds, the tracklet is deleted.

3.5 Evaluating the Application

In order to evaluate the tracking ability of the application, ground truths were generated in the format described in \[27\]. The format of the ground truth is very similar to the format of the detection shown in Table 3.8, as they both convey the same type of information about bounding box location and size. Ground truths are simply bounding boxes for each frame that describes the true position of one or more objects, while the detection bounding box is our tracker’s best guess at the true position of a given object. The ground truth represents the “truth” that the tracker will be compared against when calculating the metrics; ideally we want the tracker to predict the exact same bounding box as the ground truth. While the MOT challenge is primarily directed at tracking pedestrians, the benchmark metrics can be used to quantify the performance of our aircraft tracker.

There were no existing solution for manually generating bounding boxes that handled the very large resolution of the videos in a satisfactory manner, so a custom application based on the OpenCV contrib implementation of the Kernel Correlation Filter (KCF) tracker \[39\] was created. Using the KCF tracker for aiding in creating ground truth detections was very time-efficient compared to a completely manual process, but the process also highlighted the problems that the application would have if it was based on that tracker. Very frequent corrections were required for the labelling process, as the KCF tracker had issues with differentiating the aircraft from the background.

While some of the videos had significant activity such as trucks, traffic in the background, and employees working on the ground, ground truth bounding boxes were only generated for the airplane coming in for landing in each respective clip. The bounding boxes were tightly fitted around the airplane to the extent it was possible to discern which pixels belonged to the aircraft.

Once ground truths were generated for each video, actual tracking predictions were extracted from the main application. While the application keeps track of many potential candidates, only the candidates marked as belonging to the class “aeroplane” by the object detection algorithm was included in the prediction file. As an example of the actual format, one row of the prediction file for the SE-MKA video for the object with ID 20 at frame 843 is shown in Table 3.8. The airplane is displaced 7487 pixels to the left, and 1748 pixels from the top of the camera axis. Notice how the application gives the object a bounding box of only 79 × 45 pixels,
Figure 3.7: The detection bounding box (orange) has an IoU ≈ 0.37 with the ground truth (blue). This example would not be considered a valid detection with an IoU threshold of 0.5.

showing the large difference in scale between the total area of the video feed and the size of the aircraft that is being tracked. Confidence, class, and visibility are factors that are specific to the ground truths for the MOT challenge, and are therefore ignored in the evaluation.

Once there is both a ground truth file and a prediction file for each video clip, the metrics can be calculated using one of the multiple tools available for the MOT benchmark. The official MOT developer kit is written in Matlab code [27], but alternatives such as py-motmetrics that is written in Python exists. The latter was chosen for calculating the metrics for the application. During the evaluation process, it became apparent that the IoU threshold of 0.5 was punishing the far-away detections that were bounded by as few as 20 × 10 pixels. The threshold for regarding a match as a true positive was lowered to 0.35. A situation where this is relevant is shown in Figure 3.7.

py-motmetrics available at github.com/cheind/py-motmetrics
The results from the tests are presented in this chapter. The chapter is divided into separate sections for the results for the tracker, detector, and application. Finally, screen captures from the application in various situations are presented.

4.1 Tracking metrics

The calculated metrics for the tracker is presented in Table 4.1. A short description of the different metrics can be found in Table 4.2.

4.2 Object Detection Performance

When running the custom object detection on the test dataset containing 600 images, the detector achieved an mAP of 90.91%, and an average IoU of 89.05%.

The custom object detector was also benchmarked against the YOLOv2 object detector on each of the video files, with the metric being the number of frames before the tracked object was accurately classified as an airplane (time until-detection). 30 frames corresponds to one second of video playback time. Frames were used as a unit of benchmarking instead of seconds in order to make the results independent of processing time. The results have been presented in Table 4.3, where a lower score is favourable because it implicates earlier classification of aircraft.

A snapshot was taken of the cropped images for each of the first detections done by both ArlandaNet and YOLOv2, presented in Figure 4.1 and Figure 4.2.

<table>
<thead>
<tr>
<th>ID1</th>
<th>ID2</th>
<th>ID3</th>
<th>Recall</th>
<th>Percentage</th>
<th>GT</th>
<th>MT</th>
<th>PT</th>
<th>ML</th>
<th>FP</th>
<th>FN</th>
<th>IDs</th>
<th>FM</th>
<th>MOTA</th>
<th>MOTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE-ML</td>
<td>93.0%</td>
<td>93.4%</td>
<td>94.4%</td>
<td>94.4%</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>17</td>
<td>97.5%</td>
<td>92.4%</td>
<td>0.324</td>
</tr>
<tr>
<td>SE-MKA</td>
<td>93.9%</td>
<td>92.4%</td>
<td>94.4%</td>
<td>94.4%</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>503</td>
<td>203</td>
<td>37</td>
<td>97.5%</td>
<td>92.4%</td>
<td>0.324</td>
</tr>
<tr>
<td>OTHER</td>
<td>91.9%</td>
<td>90.5%</td>
<td>91.4%</td>
<td>90.8%</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2047</td>
<td>1134</td>
<td>24</td>
<td>84.3%</td>
<td>91.3%</td>
<td>0.324</td>
</tr>
</tbody>
</table>

Table 4.1: MOT metrics for the tracker with IOU threshold $t_d = 0.35$
4.2. Object Detection Performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDF1</td>
<td>ID F1 score</td>
</tr>
<tr>
<td>IDP</td>
<td>ID precision</td>
</tr>
<tr>
<td>IDR</td>
<td>ID recall</td>
</tr>
<tr>
<td>Recall</td>
<td>Number of detections over number of objects</td>
</tr>
<tr>
<td>Percentage</td>
<td>Number of detected objects over sum of detected and false positives</td>
</tr>
<tr>
<td>GT</td>
<td>Number of ground truth tracks</td>
</tr>
<tr>
<td>MT</td>
<td>Number of mostly tracked objects</td>
</tr>
<tr>
<td>PT</td>
<td>Number of partially tracked objects</td>
</tr>
<tr>
<td>ML</td>
<td>Number of mostly lost tracks</td>
</tr>
<tr>
<td>FP</td>
<td>Number of false positives</td>
</tr>
<tr>
<td>FN</td>
<td>Number of false negatives</td>
</tr>
<tr>
<td>IDs</td>
<td>Number of ID switches</td>
</tr>
<tr>
<td>FM</td>
<td>Number of track fragmentations</td>
</tr>
<tr>
<td>MOTA</td>
<td>Multiple object tracker accuracy</td>
</tr>
<tr>
<td>MOTP</td>
<td>Multiple object tracker precision</td>
</tr>
</tbody>
</table>

Table 4.2: Short descriptions for each tracking metric.

<table>
<thead>
<tr>
<th>Video</th>
<th>Algorithm</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE-MJJ</td>
<td>Arlanda</td>
<td>981</td>
</tr>
<tr>
<td>SE-MJJ</td>
<td>YOLOv2</td>
<td>1161</td>
</tr>
<tr>
<td>NTJ3102</td>
<td>Arlanda</td>
<td>281</td>
</tr>
<tr>
<td>NTJ3102</td>
<td>YOLOv2</td>
<td>281</td>
</tr>
<tr>
<td>CCIXT</td>
<td>Arlanda</td>
<td>747</td>
</tr>
<tr>
<td>CCIXT</td>
<td>YOLOv2</td>
<td>849</td>
</tr>
<tr>
<td>SE-MKA</td>
<td>Arlanda</td>
<td>1520</td>
</tr>
<tr>
<td>SE-MKA</td>
<td>YOLOv2</td>
<td>1406</td>
</tr>
</tbody>
</table>

Table 4.3: Time until classification with 15 frames between each classification attempt. The best score for each video is highlighted where applicable.

(a) SE-MJJ (70 × 55)  (b) NTJ3102 (112×72)  (c) CCIXT (66 × 55)  (d) SE-MKA (120 × 70)

Figure 4.1: Detection snapshots for ArlandaNet

(a) SE-MJJ (98 × 61)  (b) NTJ3102 (112×72)  (c) CCIXT (85 × 53)  (d) SE-MKA (81 × 52)

Figure 4.2: Detection snapshots for YOLOv2.
4.3 Application Latency

The average time to process a frame was calculated for each video. Two video crops are compared: one where the runway and airport apron are observed by the application (Full), and one where only the runway is observed (Runway). In Table 4.4 the results are presented, with an indication of whether the processing time is above or below the threshold for processing real-time (approximately 33 ms). Lower time is better.

The average distribution of time spent per frame is shown in Figure 4.3. The computation for each frame is divided into four different areas:

1. **Read**: Reading frame from video file
2. **BG Subtraction**: Performing background subtraction and contour extraction
3. **Track update**: Associating and updating all Kalman-based trackers
4. **Detect**: Running object detection on tracked areas

Detection is not performed for every frame, so the total time for the average time distributions in Figure 4.3 will exceed the average latency shown in Table 4.4.

### Table 4.4: Average latency for frame processing with different video cropping

<table>
<thead>
<tr>
<th>Video</th>
<th>Crop</th>
<th>Avg time</th>
<th>Real-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE-MJJ</td>
<td>Full</td>
<td>27.0 ms</td>
<td>Yes</td>
</tr>
<tr>
<td>SE-MJJ</td>
<td>Runway</td>
<td>11.7 ms</td>
<td>Yes</td>
</tr>
<tr>
<td>NTJ3102</td>
<td>Full</td>
<td>51.0 ms</td>
<td>No</td>
</tr>
<tr>
<td>NTJ3102</td>
<td>Runway</td>
<td>22.3 ms</td>
<td>Yes</td>
</tr>
<tr>
<td>CCIXT</td>
<td>Full</td>
<td>96.5 ms</td>
<td>No</td>
</tr>
<tr>
<td>CCIXT</td>
<td>Runway</td>
<td>63.5 ms</td>
<td>No</td>
</tr>
<tr>
<td>SE-MKA</td>
<td>Full</td>
<td>97.5 ms</td>
<td>No</td>
</tr>
<tr>
<td>SE-MKA</td>
<td>Runway</td>
<td>63.3 ms</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 4.3: Average time distributions in milliseconds for processing a frame in the SE-MKA landing clip with 7 horizontally stacked cameras. "Large crop" refers to the application observing both the runway and airport apron, while "small crop" refers to observing the runway only.

4.3 Application Latency

The average time to process a frame was calculated for each video. Two video crops are compared: one where the runway and airport apron are observed by the application (Full), and one where only the runway is observed (Runway). In Table 4.4 the results are presented, with an indication of whether the processing time is above or below the threshold for processing real-time (approximately 33 ms). Lower time is better.

The average distribution of time spent per frame is shown in Figure 4.3. The computation for each frame is divided into four different areas:

1. **Read**: Reading frame from video file
2. **BG Subtraction**: Performing background subtraction and contour extraction
3. **Track update**: Associating and updating all Kalman-based trackers
4. **Detect**: Running object detection on tracked areas

Detection is not performed for every frame, so the total time for the average time distributions in Figure 4.3 will exceed the average latency shown in Table 4.4.
4.4 Overall

Screen captures were taken for each video to highlight different situations. Figure 4.4 illustrates an airplane rotating on the runway after a landing, in order to taxi to the airport apron. Figure 4.5 shows the tracker following SE-MJJ through occlusion during the landing of SE-MJJ.
Figure 4.5: SE-MJJ being occluded during landing. It has not yet been classified as an airplane by the application and is therefore considered a candidate.

Figure 4.6: SE-MJJ detail.
Figure 4.7: NTJ3102 detail. The bounding box picks up both the full shadow of the airplane and the reflection in the nearby ice.

Figure 4.8: SE-MKA detail.
Figure 4.9: CCIXT detail.
In this chapter, the results and other aspects of the thesis work are discussed. The results from chapter 4 are compared with state-of-the-art. The method is also discussed, with focus on validity and reliability of the approach. Finally, some source criticism is presented followed by discussion about the work in a wider societal context.

5.1 Results

The results from the tracker evaluation will be discussed both quantitatively and qualitatively. The different modules of the system have to a certain degree been benchmarked with separate metrics, which in turns reflects the characteristics of the final application.

5.1.1 Tracking

The top five trackers from the MOT16 challenge is presented in Figure 5.1, with the highest MOT A score achieved being 71.3, and the highest ID $F_1$ score being 70.1. The tracking metrics presented in Table 4.1 shows that the application does a good job of tracking the airplanes by achieving “Mostly T racked” for the airplane in each video. Furthermore, all ID $F_1$ scores are in the range 90% to 94%, indicating that the tracker is highly accurate over the entire lifetime of a track. The MOTA metric for accuracy varies between 79% and 87%, which also a good score compared to the state of the art trackers from the MOT16 challenge.

These difference in scores can be explained by the fact that the trackers in the MOT benchmark are tested on several types of objects with varying motion. Since it is important

<table>
<thead>
<tr>
<th>Rank</th>
<th>Tracker Name</th>
<th>MOTA</th>
<th>IDF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HT_SJTUZTE</td>
<td><strong>71.3</strong></td>
<td>67.6</td>
</tr>
<tr>
<td>2</td>
<td>LMP_p</td>
<td>71.0</td>
<td><strong>70.1</strong></td>
</tr>
<tr>
<td>3</td>
<td>LM_NN</td>
<td>69.0</td>
<td>61.9</td>
</tr>
<tr>
<td>4</td>
<td>KDNT</td>
<td>68.2</td>
<td>60.0</td>
</tr>
<tr>
<td>5</td>
<td>LM_CNN</td>
<td>67.4</td>
<td>61.2</td>
</tr>
</tbody>
</table>

Table 5.1: The top five trackers from the MOT16 benchmark.
5.1. Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Data</th>
<th>Aero mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast R-CNN</td>
<td>07++12</td>
<td>82.3%</td>
</tr>
<tr>
<td>Faster R-CNN</td>
<td>07++12</td>
<td>84.9%</td>
</tr>
<tr>
<td>YOLO</td>
<td>07++12</td>
<td>77.0%</td>
</tr>
<tr>
<td>SSD300</td>
<td>07++12</td>
<td>85.6%</td>
</tr>
<tr>
<td>SSD512</td>
<td>07++12</td>
<td>87.4%</td>
</tr>
<tr>
<td>ResNet</td>
<td>07++12</td>
<td>86.5%</td>
</tr>
<tr>
<td>YOLOv2</td>
<td>07++12</td>
<td>86.3%</td>
</tr>
</tbody>
</table>

Table 5.2: PASCAL VOC2012 test detection results for the class aeroplane for a selected group of object detectors.

that the airplane is tracked throughout the video, the ID metrics should be emphasized, as they punish lost ID tracks more than the MOT metrics.

The tracker scores significantly higher on the ID $F_1$ metric for the two videos from Sundsvall-Timrå airport. This can be explained by the fact that the two videos from SDL are filmed during overcast weather, which result in very few reflections and shadows that can interfere with the tracking. The videos from Örnsköldsvik airport are filmed during very sunny days, which impacts the visuals of the airplanes and leads to lens artefacts because of reflections from surfaces in the video.

As can be seen in Figure 4.7, the bounding box for the tracker has a tendency to include the shadows and reflection of the airplane in the bounding box. A strategy for mitigating this would be to use the improved MOG algorithm, which has shown the ability to differentiate shadows and reflections. Switching out the background subtraction algorithm would however increase the computational complexity of the application, in turn increasing the overall latency. The BackgroundSubtractionCNT algorithm can be seen as a compromise between latency and precision, with the precision being very comparable to that of the MOG algorithm in all situations except those where reflections and shadows are interfering in the tracking.

5.1.2 Comparing the Object Detector with State-of-the-Art

With the CNN detector achieving a mAP of 90.9%, it surpasses the mAP of several detectors on the class aeroplane, including the original YOLOv2 detector by 4.6% as can be seen in Table 5.2.

A direct comparison is difficult because of the differing labelling strategy detailed in subsection 3.3.3. While the training set for the VOC2012 features labels enclosing the entire airplane, our custom dataset from Arlanda airport features labels enclosing the fuselage only. The resulting increase in mAP for the aeroplane class appears to be in agreement with the anticipated effect of employing this labelling strategy.

5.1.3 Latency

The latency of the application has shown to increase as the video resolution increases, which is caused by the time required to read a frame from disk as seen in Figure 4.4. This led to the videos with the lowest number of cameras also having the lowest overall latency, at around 27ms per frame. The time to read the frame from disk is roughly constant regardless of what area of the video is being actively observed by the application, as seen in Figure 4.3.

The time required to perform background subtraction and contour extraction increases greatly as a larger area within the same video is being observed, as seen in Figure 4.3. The effect of this could potentially be heavily mitigated by utilizing more processor cores for computing the background subtraction mask in parallel. This could be achieved by dividing the video feed into different sections, run the background subtraction algorithm on each section in parallel, and assemble it back into one mask once all computation is done.
5.2. Method

The time required for detection stays approximately constant, but at around 30 ms it impacts the performance of the application quite a lot. This delay could be almost completely eliminated by performing the detections asynchronously in a separate thread. Since each tracked candidate is given a unique ID, detectors could be queued in a separate thread, and then update the tracker for that ID once finished predicting the object class.

Updating the tracker takes very little time at around 5 ms, and therefore contributes very little to the overall latency. This can mostly be attributed to the efficiency of the Kalman filter and Hungarian assignment algorithm.

The videos with the most cameras took up to almost 100 ms in average to process one frame, with almost half of that time being used to read a frame from the video. This could in most cases be largely resolved by reading the next frame while processing the current one, in order to prevent the blocking of the main thread by file IO. Adjusting the application to only observe the runway improved latency significantly, hinting at the possibility of achieving real-time performance in part by only observing areas where airplane activity is expected.

In summary, the prototype does not achieve real-time performance mainly because a lack of a multi-threaded architecture. This is partly caused by the application having been written in an interpreted language where only one thread of execution was used. By spreading out the computation on several cores for the background subtraction step, latency would most likely be drastically reduced. The same would be true for the object detection step, as have been discussed. Implementing these optimizations were however not a goal of building the prototype, as a clean architecture favouring rapid iterations was prioritized over achieving performance through parallelization.

5.2 Method

During the training of the CNN, there was an interruption where the dataset had to be expanded. Having an unbalanced dataset was an oversight caused by an unwarranted belief that the loss function would properly punish false positives in the limited range of images that were originally used. Luckily, the fix to the problem was trivial, as the VOC2012 dataset contains meta-information that enabled the extraction of images where aircraft where explicitly not featured. In hindsight, the training should have been restarted in order for the log error graph in Figure 3.6 to be more descriptive of the training that was performed.

It could be argued that using an object detector instead of an object classifier was not necessary, since the predicted bounding boxes were not fed into the Kalman filters. The object detector could be replaced with a classifier that judged whether the candidate image crop passed to it contained an airplane or not, and it would serve the same functionality. The YOLOv2 object detector was chosen because originally the foreground objects were intended to be detected using the object detection method, and then tracked. It quickly turned out that there were some challenges associated with using a CNN to detect foreground objects, detailed in section 3.3. From the overview of the YOLOv2 architecture in subsection 3.3.2 we saw that the detector is basically a re-purposed object classifier, which means that we can look at the YOLOv2 object detection system as a sort of image classifier with the extra feature of outputting bounding boxes for objects.

A consequence of the survey of the trackers in subsection 2.5 was that no trackers using appearance features were deemed usable for the application. This can be seen as a bit problematic since these are the most popular type of trackers in literature, and therefore the trackers where most research is done. Visual trackers have recently incorporated techniques from the field of Deep Learning, such as using deep neural networks to train appearance features [50]. These developments are very recent, and are therefore not ported to other languages than what they were written in, with Matlab and C++ being the most popular choices for trackers. This is a major obstacle for implementing very recent tracking algorithms, as Python wrappers have to be written for the trackers, or the trackers have to implemented from scratch in Python.
Many trackers also feature complicated architecture that makes them challenging to adapt for custom problem areas.

For the MOT metric evaluation, the required bounding box IoU threshold with the ground truth for a detection to count as a true positive was changed to 0.35. This was done after a qualitative assessment where the tracker was punished for detections on objects far away, where achieving an IoU over 0.5 for a tightly bounded ground truth was not feasible using any technique. This somewhat hurts the validity of the comparison to state-of-the-art trackers, since they operate with an IoU threshold of 0.5.

There are more differences between the MOT benchmark and the use-case for our application. The major downside of comparing to the MOT benchmark is that it the MOT trackers are only expected to track pedestrians: the entire MOT test and train datasets contain labelled videos of pedestrians. There is another challenge for tracking single objects, named the Visual Object Tracking challenge (VOT) \[51\]. Using the same metrics as the tracker participating in the VOT challenge would probably make the comparison of metrics more valid than comparing with the scores from the MOT challenge. However, the implemented tracker in the application is a tracker capable of tracking multiple objects: we just do not have any video clip containing multiple airplanes moving at the same time. The application picks up the personnel, vehicles, etcetera that moves on the tarmac as candidates for tracking; if the application was modified to also track people and vehicles, the MOT metrics would definitely be better suited.

One challenge was that the dataset contains many similar images of airplanes taken from similar angles. Ideally, the dataset would contain images from many different angles, but there is some bias by using images taken by enthusiasts. These images were fairly high-resolution compared to the detections that were done in the application: when an airplane arrived, its bounding box would be very small, resulting in very few pixels being passed to the CNN. Having a CNN specialized in far-away objects could potentially improve the time-until-classification metric that was documented in Table 4.3.

One of the strengths of the video dataset is that we used videos from two different airports, which ensures that the tracker is somewhat generalizable. The footage also contained different types of airplanes, which affects both the motion model and the object detection. The larger airplanes required more of the runway for landing than the smaller ones, which meant that they had to turn 180 degrees on the tarmac in order to taxi to the airport apron. One drawback is that all the videos are recordings of landings, meaning that the algorithm was not benchmarked on any take-offs. This is significant because the motion model might not work as well for departures as it does for arrivals.

5.3 Source Criticism

For the Deep Learning theory I have almost exclusively referred to the textbook *Deep learning* written by Goodfellow, Bengio, Courville, and Bengio. The book is a well-cited book that is often referenced in other works, and it has currently been cited over 2600 times according to Google Scholar. The textbook provides a solid foundation and reference for applying Deep Learning, but since the field is rapidly evolving, a textbook will often lag behind the most recent developments. This is especially true of the selection of hyperparameters for training a convolutional neural network, as well as the network architecture itself.

For both the Hungarian method and the Kalman filter, the theory is based on the original papers that were released in 1955 and 1960 respectively. The age of these algorithms stand in contrast to the very recent developments done in the Deep Learning field. They are both frequently cited papers, with the paper presenting the Kalman filter having been cited over 27,800 times, and the paper presenting the Hungarian method being cited over 7,000 times. These are tried and true algorithms that are used in different fields.

\[1\] As of 2018-05-21
5.4 The work in a wider context

Surveillance laws in Sweden are changing in 2018, with the goal of maintaining personal integrity without hindering technological advancement in a modern society where surveillance cameras are widespread [52]. The Swedish Data Protection Authority is - starting in 2018 - responsible for overseeing all forms of camera surveillance in Sweden [53]. Implementing a system for remote tower video feeds based on neural networks poses questions about personal integrity at the workplace: a consequence of the usage of object detection in traffic, and tracking algorithms being benchmarked on videos of pedestrians, is that these systems are becoming increasingly adept at detecting and tracking people. While our proposed application tracks airplanes exclusively, the application could be extended to track people, vehicles, and other objects. Any system to be implemented at a remote tower location in Sweden would naturally have to be developed with strict focus on adhering to the new laws for camera surveillance, as well as any other regulation that would apply.

It could be hypothesized that tracking system could be implemented in a larger framework for complete automation of common ATC operations. There has been shown to be a reluctance among air traffic controllers towards accepting automation of ATC solutions [3]. This scepticism is not unfounded, as automation has shown to have some side-effects that are unwanted in a safety-critical environments such as the air traffic control environment. These side-effects include deskilling, loss of situational awareness, and automation surprises [4]. Complete automation also causes some concerns for passenger of airplanes, including the crew: would they be positive towards participating in a flight if they knew that parts of the airspace would be observed by automated systems? Studies regarding autonomous cars have shown that even if participants are positive towards autonomous cars that might sacrifice passengers to save others, they would not themselves ride with such an autonomous car [54].

It is not obvious how to judge whether it is safe to rely on a system based on neural networks. Since trained neural networks can be considered a form of black box, it can be difficult to "dissect" the application and follow its line of reasoning as is normally possible with program code. Efforts have been made to inferring how the neural networks operate by measuring what inputs activates what neurons [55] [56], opening the possibility of developing tools that practitioners could use to troubleshoot and debug their neural networks similarly to debugging ordinary code. The development of such tools would go far in legitimizing applications based on neural networks in safety-critical systems.

For the tracking algorithm evaluations I have used the MOT challenge [27] from 2016. MOT is the industry standard for benchmarking trackers capable of tracking multiple objects.
The purpose of this thesis was to develop a prototype for detecting and tracking airplanes in very high resolution videos, while processing the video in near real-time. By using standard tracking metrics such as the CLEAR-MOT \cite{28} and ID \cite{30} metrics, we have shown how well the tracker follows airplanes in remote tower video footage with different weather conditions and airplane models.

By measuring the average time required to process frames, and where in the program that processing time is allocated, several areas with potential for improvement in the prototype have been identified. It was detected that the major bottleneck of the application mainly has to do with a lack of multi-threading for the file IO and computations. Multi-threading was avoided partly because of a lack of support for native threading in Python, as well as the goal to keep the prototype architecture clean for a potential future implementation in a compiled language. Most of the libraries used in the prototype are originally written in C++, a compiled language that is often used for multi-threaded performance-critical software systems \cite{57}, which makes C++ a strong candidate for a future implementation of the prototype.

Another goal of the application was to explore how object detection and object tracking could be combined for the tracking problem in very high-quality video feeds. Common object detector networks have a fixed input size in the range $416 \times 416$ pixels, which means that very high-resolution video frames can not be analysed in a single forward-pass without significantly down-sampling the frame. The application architecture solves the problem of down-sampling by detecting tracking candidates through relatively inexpensive computational operations, and passing cropped snapshots of the candidates through the neural net. This way, the application is continuously attempting to judge whether the candidate is of the class that is to be tracked or not, while retaining as much spatial information from the original frame as possible. One more result of the architecture design is that it opens up for object detection to be performed asynchronously, which in the future could lessen the constraint that the object detector should be low-latency.

A goal of the thesis was also to compare the performance of a custom convolutional network versus a CNN completely trained on one of the large image datasets used for challenges \cite{14} \cite{15}. The motivation for this was the hypothesis that maintaining a custom network trained on images from the Scandinavian airspace would be desirable if a tracking system were to be put in use at remote towers located in Sweden. Our tests showed that it was possible to get very comparable results to state of the art detectors with a moderate-sized dataset containing
labelled images of airplanes taken at Arlanda airport. One key factor to achieving these results was the use of transfer learning when training the object detector.

Future work that could further improve the prototype, includes the improvement of time-until-detection of airplanes, which is the measurement used for how early the airplanes were detected by the object detector during landing. This could be achieved by experimenting with different CNNs for detection/classification by having a specific focus on images with very low resolution, thereby detecting far-away airplanes as early as possible. Investigating the potential integration of trackers based on appearance models could also be beneficial, as the recent developments in this field has shown impressive results.


[57] C++ Applications. URL: http://www.stroustrup.com/applications.html