# Development of a Test Suite for Single Object Tracking Algorithms in Video

**Kieran Donnelly**

Supervised by

**Etienne Pienaar**



A minor dissertation presented for the degree of
Master of Science in Data Science

UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

Department of Statistical Sciences
2019

# Acknowledgements

This report and the underlying dissertation research was carried out at the Sony office in Lund, Sweden. Without the help of certain individuals and entities, this project would not have been possible. In no particular order, thanks are due to:

My supervisor, Etienne Pienaar, who gracefully took on an aimless student. Supervising a masters project is surely an arduous task, and I am grateful for Etienne's thoughtful, friendly and inspired feedback throughout - particularly during the closing stages.

The UCT Postgraduate Centre & Funding Office, for the award of the generous Max & Lillie Sonnenberg Scholarship, without which it would not have been financially viable to undertake what has been a life- and career-defining trip.

Lijo George, AI Business Lead at Sony AI, who took a chance in inviting a non-EU citizen to travel 10,000 km for an internship. Of the 16 master's thesis positions in Europe that I applied to, just one chose not to reject me. Lijo was the man that made it all happen.

Johan Helgertz, Lead Business Developer and Team Leader at Flying Camera Solutions, for welcoming me onto the FlyCam team in June of 2019 and providing the use-case for an interesting and challenging dissertation topic.

# Abstract

Flying Camera Solutions (FlyCam), within Sony Lund's startup accelerator, intends to provide drone videography to paying customers in ski resorts: a customer should be able to go about their activity as usual while a drone films them. Visual object tracking, enabling the drone to track the customer throughout the activity, is a primary obstacle in creating a viable autonomous videography service. FlyCam needs an object tracking algorithm which is accurate, robust, real-time, and requiring minimal computational overhead. We propose two innovations to aid in the selection of an appropriate tracking algorithm. Firstly, a video annotation algorithm, making use of an object detector to record the position and type of object in each frame of a video clip. Secondly, an algorithm designed to evaluate the performance of any given object tracker based on a set of performance metrics. These metrics include, among others, measures of positional accuracy, frame rate, and false positive rate. For the video annotation algorithm we implemented the state-of-the-art Mask R-CNN object detector, which achieved an average frame rate of 1.5 fps annotating video clips in up to 4K resolution. Another algorithm then played back the annotated clips to the user such that incorrect object detections could be rooted out or rectified. With little relevant annotated video available, the annotation algorithm proved useful in preparing a suite of 18 clips to be evaluated. Ten performance metrics were adapted from multi-object to single-object tracking. Nine tracking algorithms were then run on each of the 18 test video clips at varying resolutions to produce 375 tracking observations for analysis. The evaluation results revealed the optimal tracking algorithm to be Re3: a recurrent-convolutional neural network tracker which runs at respectable speeds on a consumer laptop. This is a promising result; with enough annotated data, neural networks can be retrained to improve performance. Within just a few months of operation, FlyCam could amass enough specific video data to significantly improve the neural network-based tracker.

A note to the reader: a hosted video presentation of this report is available at the following URL and provides a useful introduction and context: https://youtu.be/LSqSnstxgE0.

# Contents

# List of Figures



# List of Tables

# 1 | Introduction

FlyCam is a project within and receiving support from Sony in Lund, Sweden. FlyCam has commercial intent and its proposed business case is to provide paying customers with novel, memorable video of themselves performing some specified sporting activity. This will be achieved by using an unmanned aerial vehicle (or UAV), commonly known as a drone, to autonomously track and record the customer while they engage in these activities.

The key differentiator of this service from other video production services available is the entirely automated nature of the shooting process, which will be enabled by tracking the moving subject and in turn generating commands to control the motion of the drone accordingly.

The task is suitable for automation as it is a highly repeatable one: the sports FlyCam is considering (snow skiing, cycling, wakeboarding, ziplining) are typically of a point A to point B type nature, following a rough path or at the very least are conducted in a defined 3-dimensional space. A drone is a sensible device for such a system: drones require very little infrastructure (no tripods or cabling setups) and what little infrastructure is required is unobtrusive; they are able to provide novel, dynamic video clips for engaging content; and they are very easy to navigate.

The FlyCam project itself faces a multitude of challenges: chiefly, enabling autonomous flight of a drone; automation of battery-swapping, or charging; offloading/dumping of footage after filming; and automation or streamlining of the video production process.

Of the aforementioned items, visual object tracking is the primary obstacle in the path of enabling autonomous flight as agreed by the team - without being able to track an object the drone would need to be piloted manually, defeating the primary purpose of FlyCam. The challenges specific to visual object tracking are numerous. For on-aircraft tracking, physical size and mass of the computational device should be minimised, and the power drawn by this device should not adversely affect battery life. Tracking results will be used to determine the immediate future state of the drone, thus tracking should run in real-time. Further, the video product is of no commercial value if the paying customer doesn't feature in it, thus the tracking algorithm should track reliably and robustly throughout the activity.

The purpose of this research is twofold: firstly, to develop a reliable means of annotating video data for testing tracking algorithms and, secondly, to develop a test suite which enables thorough evaluation of a tracking algorithm across a variety of metrics.

The rest of this report will be structured as follows: we begin in Chapter 2 by discussing object detection in detail, including the underlying machinery and its role in object tracking. Chapter 3 is a detailed look at object tracking algorithms and the introduction

of a set of evaluation metrics that we will use in the testing stage. Chapter 4 discusses concepts related to the video data to be used in the testing stage, including the acquisition and annotation thereof. Chapter 5 describes the method by which we will evaluate the various tracking algorithms using the test data. Chapter 6 is a detailed summary of the results from the testing stage. Finally, Chapter 7 concludes the report highlighting the results obtained during the project and the possible areas to focus on for any future work.

# 2 | Object Detection

Video tracking is enabled through the combination of various fields of computer vision, namely: object detection, object tracking, and visual tracking. This section discusses **object detection** as it pertains to the area of visual tracking.

Humans are able to glance at an image or scene and immediately discern the objects within it. As noted by Redmon et al. (2016: 779), the human visual system is fast and accurate, which underpins our ability to do all manner of complex tasks in realtime and with little cognitive load.

In the context of computer vision, whereas image classification refers to simply listing the objects in an image, object detection is the task of both classifying and locating objects within a digital image. According to Redmon et al. (2016: 779), most methods make use of repurposed classification models which analyse the image at varying locations and sizes, and are thus able to apply the general image classification problem to different subregions of the overall image, knowing the location and size of each subregion. The typical machinery with which object detection models are built is the convolutional neural network (CNN), a type of artificial neural network.

An **artificial neural network, or ANN**, is a highly parameterised mathematical model used for supervised learning. It attempts to derive features from linear combinations of its inputs, in turn producing an output which is a nonlinear function of these features (Hastie, Tibshirani, and Friedman 2001: 389). Neurons, the basic units of the ANN, exist in layers of three types: input, hidden and output. The outputs of the neurons in each layer are fed in turn to the neurons making up the next layer. At each neuron the input values are multiplied by a weight and summed with a bias term before passing through a nonlinear activation function. The nonlinearity introduced by the activation function helps to transform the inputs in a way that can be used to partition the feature space. Finally, the partitioned feature space is mapped to the associated response values such that they can be predicted.

**Feedforward neural networks** are a subset of neural networks designed to approximate non-linear relationships between a set of features and an observed response. Notionally, if we let $f$ denote a function which gives the relationship between an input image and a response, say $y$, then we may use this model class to approximate $f$, i.e., $\hat{f}$. Feedforward networks are so called because information only flows in a singular, forward direction through the network. The direction of the flow says nothing about the route it takes, however. The route is determined by the architecture, which defines how the neurons and layers are connected to one another. Layers are stacked sequentially and together form a chain, the last layer of which is the output layer. Higher layers are those nearer the input layer, and lower layers those nearer the output layer. Thus, as you progress through the ANN, you pass through lower and lower layers. The length of this chain refers to the depth of the network, which is incidentally where the term deep learning arises. Hidden

layers are those in-between the input and output layers.

A **convolutional neural network, or CNN**, is a type of feedforward neural network, the primary feature of which is the convolutional layer. In these layers, the network uses filters, or kernels, to detect features in an image. The filter is a matrix, normally small - on the order of $3 \times 3$ pixels - whose values are trained to detect specific features.

The filter is moved, or convolved, over the input image pixels in steps, subject to a specified step size parameter known as the stride. At each position, the purpose of the filter is to check for the presence of the feature it has been trained to detect.

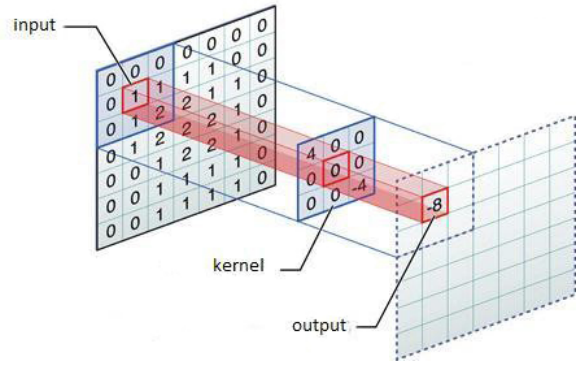



Figure 2.1: A visual representation of the convolution operation applied to a $7 \times 7$ matrix using a kernel of size $3 \times 3$ (Bluche 2017).

The output at each position is a value indicating the confidence of the presence of the specified feature. This value is the result of what is known as the convolution operation: the element-wise product and sum of two matrices. The matrices in question are the filter, and the set of pixels it is covering. Should the feature be present, the output of the operation is a high real number, and vice versa, as indicated in Figure 2.1. Passing the filter over the whole image results in an output matrix that stores the convolutions of this filter. The filter must have the same number of channels as the input (three for RGB images) for matrix multiplication to be possible. This way, convolutional layers preserve the spatial representation of the input data, important when processing images. The stride value determines the output dimensions, and in practice a value of one works best (Karpathy et al. 2016).

At this stage it is important to understand the operation of a simple neuron within a neural network. First, multiply each input by a weight. Then, sum these products with some bias term. Finally, pass the result through a non-linearity function.

For the network to learn the values that a filter must contain in order to detect features, the filter is passed through a non-linear mapping. The output of the convolution operation is summed with a bias term, and then passed through some non-linear activation function. The activation function introduces non-linearity into the network, so that we have a way of learning the non-linear nature of how pixels combine to make features. Some common activation functions: rectified linear unit, or ReLU, which sets all negative values to zero and leaves positive values as they are; softmax, which produces a set

of probabilities which sum to one for $C$ classes; sigmoid, used for binary classification and produces a single output for the probability of belonging to a particular class.

Another concept central to CNNs and indeed ANNs in general is called downsampling. It is intuitive that there should be fewer outputs from the model than there are inputs (pixels in the image, in this case); an image will likely never contain as many features or objects as there are pixels. We use downsampling to arrive at the desired number of outputs, be they probabilities, coordinates, or class labels. There are a number of mechanical advantages to downsampling, too: it reduces redundancy present in input features; it helps to reduce memory consumption; and in turn it speeds up training and inference.

One common method of downsampling in CNNs is known as pooling, specifically max pooling, which takes place in a layer of the same name. Just like the convolution operation, in a max pooling layer a window, or filter, of specified dimensions is passed over an image. In most cases, this 'image' is the output matrix from a convolutional layer and is referred to as a feature map. The window moves with some predefined stride value, and in each position the maximum value within the window is pooled into an output matrix. This helps to reduce some set of pixels (say, $2 \times 2$) to a single value. The max pooling operation is portrayed in Figure 2.2.

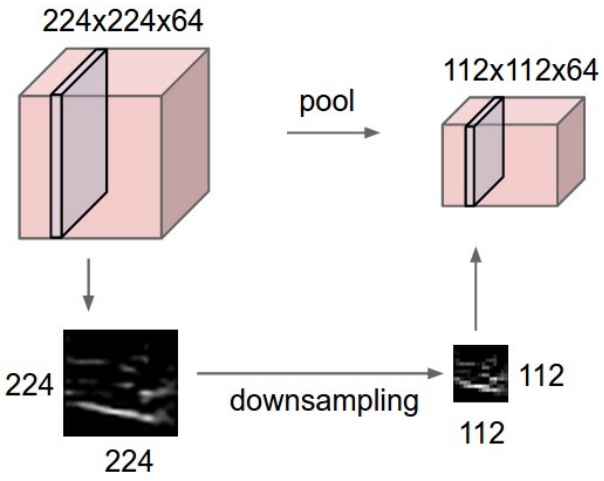



Figure 2.2: Max pooling operation using a window size of $2 \times 2$ and a stride of 2, applied to a set of 64 feature maps of shape $224 \times 224$ (Karpathy et al. 2016).

Lastly, we come to the fully-connected, or FC, layer. The distinction between this and a convolutional layer is that the latter contains neurons which are only connected to local areas of an image. The inputs of the FC layer connect to all the activations of the previous layer. This creates a feature vector, or a 1-dimensional list, of all the image features that the previous layers have teased out of the image.

The information passes through at least one FC layer, the last of which contains the output. The output could be a list of probabilities for each class (in a classification task), but in object detection could also be a set of bounding box or polygon coordinates, making it

a regression task. Combining the convolutional, ReLU, max pooling and fully-connected layers in some fashion produces the convolutional neural network.

CNNs are well-suited to image processing tasks because they are able to handle inputs of high and varying dimensionality. For example: a frame from a Full HD resolution colour video sequence could be represented as a matrix of size $1080 \times 1920 \times 3$, which we could feed into a CNN: height, $h = 1080$ *pixels*; width, $w = 1920$ *pixels*; depth, $d = 3$ (red, green and blue) - indicated in Figure 2.3.

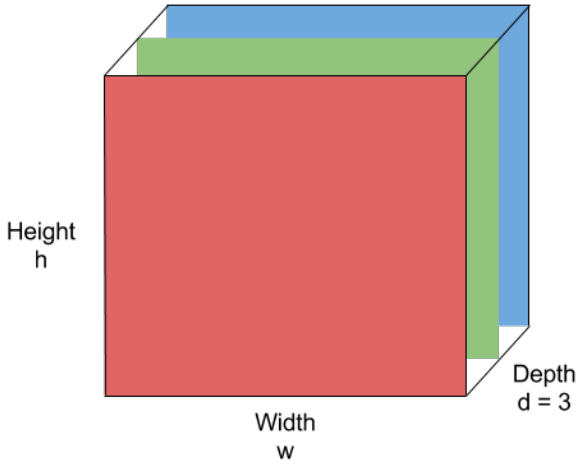



Figure 2.3: 3-Dimensional nature of digital colour image (Punyawiwat and Wattanapenpaiboon 2018).

# Relevant Literature

In the field of object detection there have been several landmark advances in the last five to ten years. In their breakthrough paper, Simonyan and Zisserman (2014) reveal a number of findings about deep convolutional neural networks for object detection, on which many subsequent works are based. The paper covers experiments with different levels of depths in convolutional neural networks (CNNs), and the work itself helped the authors to place both first and second in the 2014 ImageNet competition - a popular image recognition benchmark. The authors noted that improvements in error rate - that is, a lower mean average precision (mAP) for image classification - could be achieved by increasing the depth of a CNN (Simonyan and Zisserman 2014: 6). In their implementations they found that by using three rectified linear units (ReLUs) rather than one as was standard in prior-art, the decision function of the model could be made more discriminative. The authors do not go on to explain what increased discriminability might mean, but we assume it to refer to an improvement in the model's ability to differentiate between object classes. They note that they were able to increase the depth of the model without increasing the number of parameters relative to previous shallow networks.

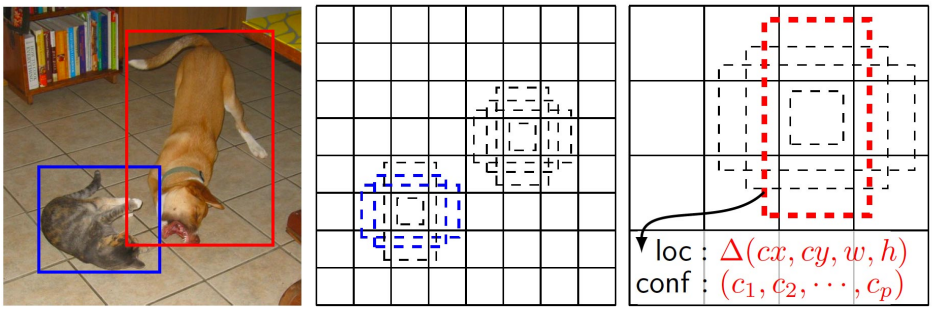
The architecture itself, which has come to be known as VGGNet, is now a popular model architecture. It specifies a fixed input of a $224 \times 224$ RGB image. The convolutional

layers, of which there are either 13 or 16 depending on the implementation, are of stride and padding value one, and of kernel size $3 \times 3$. The convolutional layers are interspersed with max pooling layers which do not count towards the depth of the network. The pooling layers are of window size $2 \times 2$, stride value two, and there are five of them. The most popular implementation is VGG-16, where 16 refers to the 13 convolutional layers and the three fully-connected layers.

The first algorithm we detail which makes use of VGGNet is Single Shot MultiBox Detector, or SSD: an approach to object detection using convolutional neural networks by Liu et al. (2016: 1–17). The premise of SSD is that it predicts both category scores and box offsets for a fixed set of default bounding boxes, using small convolutional filters applied to feature maps. Some degree of robustness arises from the fact that the model produces predictions of different scales from feature maps of different scales, and these predictions are separated by aspect ratio (Liu et al. 2016: 8). Robust in this case means a higher degree of invariance to the shape of an object. For instance, a narrow object like a skyscraper and a round object like a ball are equally well-detected despite the difference in their aspect ratios. The different scales used to produce feature maps refer to differences in granularity of feature maps in the network. This is illustrated in Figure 2.4: in (a) we note that the cat appears smaller than the dog. In (b) we see an $8 \times 8$ feature map effectively used to detect the smaller cat, but in (c) we can see that a coarser $4 \times 4$ feature map is sufficient to detect the slightly larger dog. Some aspect ratios are more suitable than others for a given object, so multiple predictions are made and centered in the same location, and later ranked by confidence. The different aspect ratios are indicated by the dotted lines in Figure 2.4, frames (b) and (c). These features lead to a simple end-to-end trainable model with high accuracy, even on low resolution images. For comparison, a region proposal-type network is double shot - requiring one pass to generate region proposals, and another to detect objects from each proposal (more on region-based networks below). *Single shot* means that an image need pass through the entire network just once for training or inference - whichever is required.

The multibox detector part of the model is described as follows: an image is passed through some number of convolutional layers for feature extraction - the authors state they used VGG-16 (discussed above), but that other networks should also be useful for purposes of feature extraction. The result is a feature layer of size $m \times n$ (the number of object locations), with $p$ channels. $m$ and $n$ are the fixed width and height values, respectively, to which input images are resized, and the number of channels, $p$, is commonly three for RGB (red, green, and blue) images. A $3 \times 3$ convolutional layer is applied to the $m \times n \times p$ feature map. At each of the $m \times n$ locations, $k$ bounding boxes are produced, each with different sizes and aspect ratios (different objects have shapes which might be suited by different rectangles). $c$ class scores and four box offsets relative to the original bounding box shape are computed for each of these bounding boxes, resulting in a total of $(c+4)kmn$ outputs. Producing bounding boxes for the feature maps by passing them through convolutional layers of different sizes produces a total of 8,732 outputs (YOLO, discussed below, produces just 98). Figure 2.4 illustrates the SSD framework.

You Only Look Once, or YOLO, is another popular object detection algorithm detailed by Redmon et al. (2016). YOLO treats object detection as a regression, rather than a classification problem, producing predictions for the coordinates of a bounding box




(a) Image with GT boxes (b) $8 \times 8$ feature map (c) $4 \times 4$ feature map

Figure 2.4: SSD framework for training and inference. (a) presents two objects of different size and aspect ratio, (b) shows a feature map scale of $8 \times 8$ is used to detect the smaller cat, and (c) shows a coarser feature map of $4 \times 4$ is used to detect the larger dog (Liu et al. 2016: 3).

(*width*, *height*, *and location*), along with the class probabilities for the object. It is fast to train as there is only one component to optimise: a single neural network to both predict bounding box coordinates and calculate class probabilities (Redmon et al. 2016: 779).

One of the drawbacks of the procedure is that YOLO tends to make localisation errors compared with other state of the art models, though it still achieves appreciable accuracy. Localisation errors are inaccuracies in the estimate of an object's precise location, for instance when the model's prediction of the edge of a bounding box isn't quite where the edge of the object is. Whereas other algorithms might only consider a local area or patch within an image (for instance using sliding windows, or region proposal networks), YOLO considers the entire image at both train and test time. This provides contextual information about classes (what common backgrounds might look like, for instance), enabling a global perspective of the image. Compared to Fast R-CNN (a now dated model having been superseded by Mask R-CNN), YOLO makes as few as half the number of background errors (misjudgments in regard to whether pixels are part of the background or foreground of the image) because it is always considering the entire image. The authors note too that the model learns generalisable representations of objects, which means the model generalises well to other datasets and tends to understand the *concept* of an object (emphasis Saha (2017)). We understand concept here to mean any general patterns, or common shapes and colours, which similar objects tend to exhibit. We note here, however, that it is not clear how the authors presume to establish the ability of a mathematical model to internalise or understand abstract phenomena - at least in the way humans do. The word 'learning' here is a metaphor, not the process of learning as it pertains to human cognition.

The algorithm is summarised as follows: the image is divided into an $S \times S$ grid (typically $13 \times 13$). For each cell in the grid, produce $B$ bounding boxes, typically five. For each bounding box, determine if *any* object exists within it, and produce a corresponding confidence score if so. Confidence is a measure of the IoU (intersection over union), a measure of rectangular overlap, between any prediction and the ground truth object. In this context, ground truth object refers to the objective presence of the object in the image. In other words, someone has previously confirmed the presence and location of

the object - its existence in the image is provable. Then, compute a class probability for each bounding box object along with the centroid coordinates $x, y$ and box dimensions $w, h$. For the cell, a class probability is produced for each of the C classes. The goal is to predict if the cell is part of an object and if so, to determine which class the object belongs to. The architecture of the CNN consists of 24 convolutional layers followed by two fully-connected layers. In CNNs capped off by fully-connected layers, the number of units in the fully-connected layers is a hyperparameter chosen by the designer of the network. In their implementation, the authors trained the first 20 convolutional layers on the ImageNet dataset for a week, before converting it to be trained for detection. To do so, they appended the four remaining convolutional layers - followed by the two fully-connected layers - and initialised them with random weights. Training again on the ImageNet data, the final layer predicts both the object bounding box and class, with outputs normalised to lie between $[0, 1]$ using *leaky ReLU* as an activation function.

Finally, we investigate an algorithm entitled Mask R-CNN: the fourth and latest algorithm in a family known as R-CNNs, detailed by He et al. (2017: 1–12) from Facebook AI Research. The three approaches preceding it are known as R-CNN, Fast R-CNN, and Faster R-CNN, in that order. R-CNNs, where *R* stands for *region proposal*, are a means of region-based object detection which since 2014 have yielded impressive accuracy but have not quite been able to compete with the likes of SSD and YOLO in terms of inference speed. Typical object detection models are able to predict the location (using a bounding box) and class of an object. Mask R-CNN goes one further and is able to generate pixel-level masks in what is called *instance segmentation* - indicating exactly which pixels in an image belong to which object, the class of that object, and additionally is able to distinguish between objects of the same class.

The primary distinction between these models and other state of the art methods is the presence of the region proposal network, or RPN. The input image, which is subject to no size restrictions, is passed through the RPN. For a given image, the job of the RPN is to propose good object candidates - that is, areas of the image likely to contain objects. The RPN divides the image into cells, and for each cell considers several anchor boxes with different aspect ratios. For each anchor box, an object confidence score is also computed. In earlier versions of R-CNN, these regions of interest (RoIs) were then passed through an RoI-pooling layer. RoIPool layers presented an alignment problem: the region of the feature map corresponding with an RoI in the original image was quantized, which meant that the region in the feature map might be slightly larger or slightly smaller than the object it was intended to represent. Figure 2.5 shows how a $15 \times 15$ RoI requires a corresponding region of $2.93 \times 2.93$ in the feature map. Fractions of a pixel could not be considered, so information was lost.

The alignment issue was rectified through Mask R-CNN's introduction of RoIAlign, where no quantization is carried out, and bilinear interpolation is used to compute values where RoIs require partial pixel information. The RoIs proposed by the RPN are then passed through to the head of the network where three branches are each responsible for a different task. The first branch is responsible for object classification (using a support vector classifier followed by a softmax activation function). The second branch is responsible for linear bounding box regression. The third branch is responsible for instance segmentation (using a small, fully convolutional network followed by a sigmoid activation

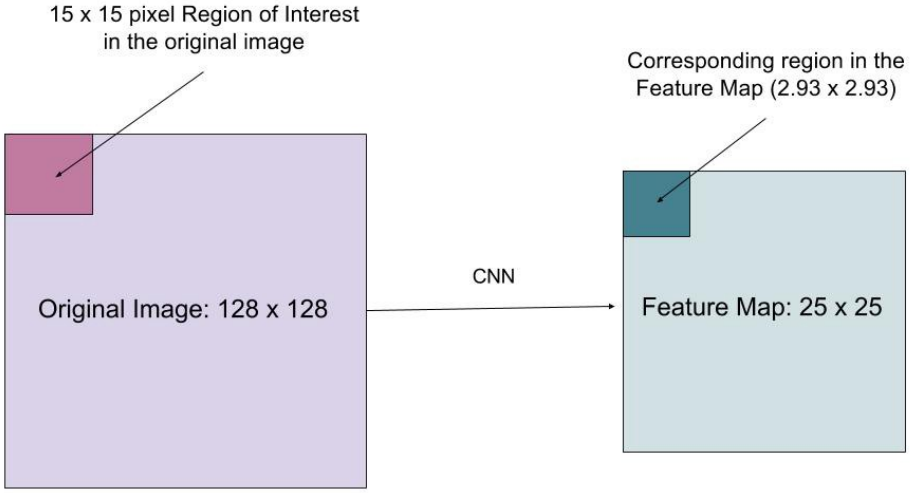



Figure 2.5: Understanding the alignment issue caused by RoIPool in early R-CNN models (Parthasarathy 2017).

function). The instance segmentation branch is known as the mask branch, and uses a sigmoid activation function to ensure that a) classes don't compete with one another and b) outputs are either zero or one on a pixel basis (producing binary masks).

Of course, considering multiple RoIs per cell in the image grid produces multiple overlapping bounding boxes of the same object. To ensure just one bounding box per instance, a technique called non-max suppression, or NMS is used. During NMS, an intersection over union (or IoU - detailed in Equation 3.6) measure is computed pairwise between the bounding box with the highest confidence, and all the other bounding boxes. Those with the highest IoU values are considered to refer to the same object, and are thus ignored.

# 3 | Video Tracking

**Object tracking** refers to using sensor measurements to determine the state of an object, where the state might refer to position, attitude, or velocity. There are multiple sensor types which can provide measurements to enable tracking: GPS, radio transmitters, radar, bluetooth beacons, aural sensors (SONAR), visual sensors like LIDAR and cameras. This project will focus on tracking with visual sensor data, specifically the use of single-object tracking in a video feed using computer vision techniques. This field is more commonly known as visual object tracking or video tracking.

All object tracking techniques refer to the object being tracked as the target (Challa et al. 2011). Most target tracking systems use filtering and/or prediction techniques which invoke sensor measurements to obtain best estimates of the target state for the current time or until some time in the future (Challa et al. 2011). *State* here refers to the velocity and attitude of the target.

**Video tracking** systems are enabled by combining both object detection and object tracking. As above, CNNs make excellent platforms for object detectors, and indeed many of the state of the art models we will look at make use of some CNN architecture. Broadly defined, video tracking is the process of locating a moving object over time in a video. The output of the system is the object track: the sequence of object locations in each frame of a video. With few exceptions, object trackers themselves are model-free. That is, they don't know anything about the object except its position in the first frame (Konushin and Artemov 2018).

For purposes of this exposition we detail the segmentation criteria. In their comprehensive review of multiple object tracking in video, Luo, Zhao, and Kim (2014: 3) note that there is no obvious universal criterion for categorising different tracking methods. They suggest a segmentation of the field into three separate criteria: *initialisation method*, *processing mode*, and *output type*. Their work and comments refer to multiple object tracking (MOT). However, as single object tracking (SOT) is the focus of this project, their MOT methods will be adapted to apply to SOT where possible.

The so-called **initialisation method** refers to one of two modes of tracking: *detection-based tracking (DBT)*, or *detection-free tracking (DFT)*. In *DBT*, given a sequence of frames, type-specific object detection is applied to each frame. "Then, sequential or batch tracking is conducted to link these detection hypotheses" (the objects detected in each frame), "into trajectories", or tracklets (Luo, Zhao, and Kim 2014: 3). Luo, Zhao, and Kim (2014: 3) note two issues with DBT. Firstly, the object detector needs to be trained in advance, limiting the tracker to some defined set of object classes. Secondly, the "performance of the tracker largely depends on the performance of the employed object detector" (Luo, Zhao, and Kim 2014: 3). In *DFT*, a manual initialisation of some fixed number of objects in the first frame is required. The output of the manual initialisation step is typically a bounding box around each object, either drawn manually by

a human, or as the result of inference from an object detector. In subsequent frames, each of the objects is localised. The authors note of this method that it cannot deal with new objects entering the sequence and that it does not require a pre-trained object detector (necessarily). We note that even when an object detector is used for manual initialisation of the first frame, but no detection is performed on subsequent frames, the tracking method is still considered to be detection-free.

The **processing mode** of the tracking procedure can be either *online* or *offline*, the distinction being "whether or not observations from future frames are utilised when handling the current frame". *Online* "tracking methods only rely on the past information available up to the current frame" (Luo, Zhao, and Kim 2014: 3). "The image sequence is handled in a step-wise manner, thus online tracking is also named as sequential tracking. Results are represented by an object's location and ID. Based on the up-to-time observations, trajectories are produced on the fly." *Offline* tracking, also known as batch tracking, employs information from both past and future frames to process data. Observations are "obtained in advance and are analysed jointly to estimate the final output" (Luo, Zhao, and Kim 2014: 4). Such processes are highly memory- and computationally-intensive, so it is not always possible to handle all of the frames at once. One way to overcome this is to split the video into shorter sequences or batches, inferring results sequentially for each batch.

**Output type** is either *deterministic* or *probabilistic*, where *deterministic*-type trackers will always produce the same results for the same sequence of frames, when the tracking operation is carried out multiple times. Probabilistic-, or stochastic-type trackers may produce various trajectories, each with a corresponding likelihood, each time the operation is run on the same sequence of frames. In the probabilistic case, we measure the total difference across all observed outputs, and attempt to minimise it to obtain a more consistent, though not necessarily more accurate tracker.

Having defined the criteria for different types of trackers, the authors go on to detail the tracking models employed by different algorithms. That is, the means by which the tracker recognises and keeps track of objects from one frame to the next. Luo, Zhao, and Kim (2014) refer to these as the different *components* of MOT, each of which are discussed below.

The **appearance** model is a common method for SOT, though it is seldom used in MOT applications (Luo, Zhao, and Kim 2014). In SOT, a sophisticated appearance model is constructed in order to discriminate the object from the background. To do this, the appearance model typically comprises two functions: *visual representation* and *statistical measuring*.

*Visual representation* describes visual characteristics of the object using features, categories of which are detailed here. *Local features* are those that exist in the pixels directly representing the object. The Kanade–Lucas–Tomasi (KLT) feature tracker is a popular example which searches for good local features, and has seen success in both MOT and SOT. Optical flow can also be regarded as making use of local features.

According to Luo, Zhao, and Kim (2014: 5), "*region features* are extracted from a wider

range", for example, a bounding box around the object. Luo, Zhao, and Kim (2014: 5) segment region features into three groups: zero-order, first-order, and up-to-second-order. Here, order refers to the degree of discrepancy when computing representation, for example: in zero-order, pixel values are not compared; in first-order, pixel values are compared once, etc. Zero-order features are most widely used for MOT, the most common types being the colour histogram and raw pixel template. Common first-order features are gradient-based representations like histogram of gradients (HOG), and level-set formulation. The region covariance matrix computes up to second-order discrepancy.

*Other features* are those neither local nor regional. The probabilistic occupancy map (POM) is used to estimate the likelihood of an object occurring in a specific grid cell, while depth is another feature often used to refine detection hypotheses.

In conclusion, the authors note the following on features: the colour histogram is common, but "ignores the spatial layout of the object region. Local features are efficient, but sensitive to issues like occlusion and out-of-plane rotation" of the object. Occlusion is the temporary disappearance of the target object behind some other object in the scene. Deformation is the change in physical shape of the target object throughout the scene, due either to its own change in orientation, or to a change in the perspective of the camera relative to the object. For example, consider the difference in appearance of a human filmed from the front at eye-level to that when filmed from directly above. Gradient-based features effectively describe shape "and are robust to certain transformations such as illumination changes, but they cannot handle occlusion and deformation well. Region covariance matrix features are more robust as they take more information in [*sic*] account, but this benefit is obtained at the cost of more computation." Lastly, "depth features make the computation of affinity more accurate, but they require multiple views of the same scenery and/or additional algorithm [*sic*] to obtain depth measurements" (Luo, Zhao, and Kim 2014). In the context of computing, the words *similarity* and *affinity* are used interchangeably.

*Statistical measuring*, based on the visual representation features, computes the similarity between two observations, which in this instance are the objects in each of two frames which need to be compared. The similarity between objects **i** and **j** can be written:

$$S_{ij} = F(o_i, o_j)$$

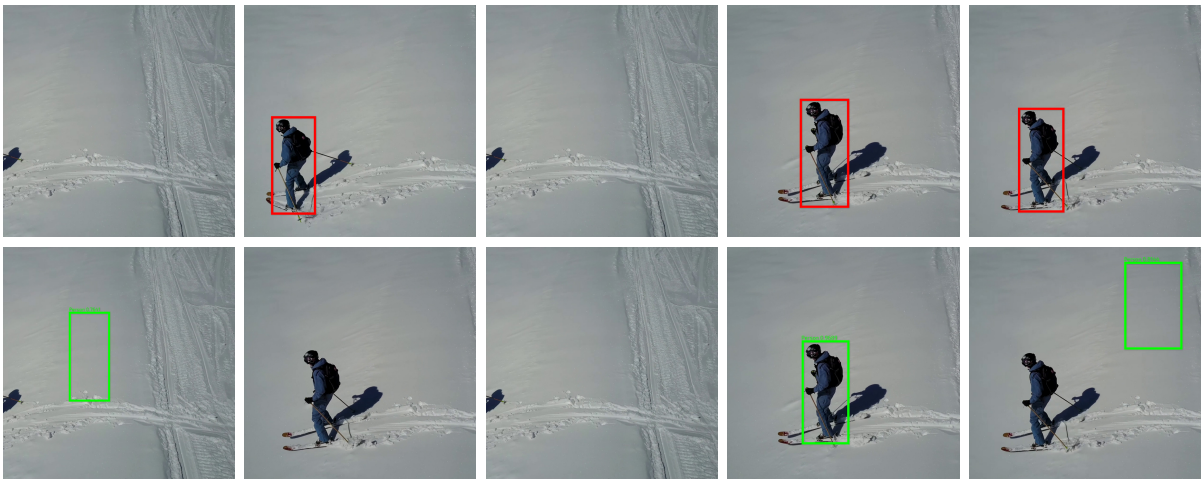
where $o_i$ and $o_j$ are visual representations of different objects and $F$ is a function measuring similarity. These measures are divided into two categories: *single cue* and *multiple cue*.

Modeling appearance using *single cue* is either transforming distance into similarity or directly calculating the affinity. For example, the Normalized Cross Correlation (NCC) is usually adopted to calculate the affinity between two counterparts based on the representation of raw pixel template" (Luo, Zhao, and Kim 2014: 5). Another measure is the Bhattacharyya distance, $B(.,.)$, used to calculate the distance between two colour histograms $\mathbf{c}_i$ and $\mathbf{c}_j$. The result is transformed into a similarity $S(\mathbf{T}_i, \mathbf{T}_j) = exp(-B(\mathbf{c}_i, \mathbf{c}_j))$. $\mathbf{T}_i$ and $\mathbf{T}_j$ are tracklets in a sequence, where tracklets are trajectories of objects through some subset of frames.

*Multiple cue* measures combine cues of different types, as they typically complement each other to result in a more robust appearance model. *Boosting* is a common method in statistical modeling, where feature subset selection is performed sequentially from the feature pool. *Concatenation* sees different kinds of features linked or chained together for computation. *Summation* methods take the "affinity values from different features and balance these values with weights", as in a linear combination. *Product* methods see feature values multiplied to produce integrated affinity. This assumes independence among features. Lastly, *cascading* methods use "various types of visual representation, either to narrow the search space or model appearance in a coarse-to-fine way." (Luo, Zhao, and Kim 2014: 6)

**Evaluation metrics** are discussed in some detail by Luo, Zhao, and Kim (2014: 10), with further clarity given by Stiefelhagen et al. (2006: 11–9). Their work applies to MOT so, where possible, metrics have been adapted for SOT. The metrics are summarised in Table 3.1. Unless otherwise stated, a metric measures over an entire sequence of frames, where a ground truth object exists in a subset of the frames or in all of them.

For each frame $t$ in a video sequence, there is either a ground truth object in it, or there is not. Further, when performing object tracking on that video clip, for each frame the tracker will either produce an output, or it will not. For the purposes of this report, we will refer to a ground truth object simply as an object $o$, and to a tracking system output as a hypothesis $h$. For each frame there is a binary state for the object and a binary state for the hypothesis, which means we can approach SOT as a binary classification problem. With the help of the sequence of images in Figure 3.1, we discuss the possible combinations of object and hypothesis for a single frame.



a. Failure (false positive) | b. Failure (false negative: miss) | c. Success (true negative) | d. Success (true positive) | e. Failure (false positive: exceeds threshold)

Figure 3.1: Example ground truth frames and corresponding hypotheses. *Top row*: ground truth objects. *Bottom row*: tracker system outputs (hypotheses).

Starting with column a. in the figure above, we can see that when there is no object but a hypothesis is made, the result is a *false positive* failure. In column b., where we have an object but no hypothesis, the result is a *false negative* failure, or a miss. Column c. indicates the case where neither an object nor a hypothesis exists, and the result is a *true negative* success. Column d. indicates the presence of an object and a hypothesis correctly made at the location of the object, the result of which is a *true positive* success. Column e. is similar to column d., in that an object is present and a hypothesis is made, but the hypothesis is clearly incorrect. This is a special case of the *false positive* failure, where the tracker has made an attempt but failed to produce an output anywhere near the object. In order to differentiate between the results of columns d. and e., we define a maximum threshold distance between object and hypothesis - if the hypothesis is too far from the object, we judge the result to be a false positive. We can therefore define a number of metrics to evaluate the performance of an individual tracking algorithm, beginning with two intuitive ones.

**Tracking precision.** Single Object Tracking Precision (SOTP) is adapted from Multiple Object Tracking Precision (MOTP) proposed by Stiefelhagen et al. (2006), and indicates the tracker's ability to estimate precise object positions. SOTP is effectively an Euclidean distance measure between the centroids of the object and hypothesis in a matched frame, normalised over all the matches, and is given by:

$$\text{SOTP} = \frac{\sum_{t=1}^{N_{frames}} \sqrt{(x_{o^t} - x_{h^t})^2 + (y_{o^t} - y_{h^t})^2}}{N_{match}} \tag{3.1}$$

where $(x_{o^t}, y_{o^t})$ and $(x_{h^t}, y_{h^t})$ denote the coordinates of the centroids of the object and hypothesis for frame $t$, respectively, $N_{frames}$ is the total number of frames in the sequence, and $N_{match}$ is the number of frames in which both an object and hypothesis exist (the sum of columns d. and e. in Figure 3.1).

**Tracking accuracy.** In SOT, we consider one primary metric which measures a tracker's ability to keep consistent trajectories, which is adapted from the Multiple Object Tracking Accuracy (MOTA), in research by Kasturi et al. (2008: 327). They compute

$$\text{MOTA} = 1 - \frac{\sum_{t=1}^{N_{frames}} (c_m(m_t) + c_f(fp_t) + c_s(ID\_SWITCHES_t))}{\sum_{t=1}^{N_{frames}} N_{objects}^t}$$

where, for a frame $t$, $m_t$ is the number of misses, $fp_t$ is the number of false positives, $ID\_SWITCHES_t$ is the number of ID mismatches based on the object mapping from frame $(t-1)$, $c_s$ is a weighting function for ID switches, $N_{frames}$ is the length of the sequence, and $N_{objects}^t$ is the number of objects in frame $t$ - either zero or one. Kasturi et al. (2008: 327) set $c_m = c_f = 1$, declaring missed detections as critical as false positives. They set $c_s = log_{10}$, and begin counting ID switches from 1 accordingly. The weights can be adjusted: if misses are more critical than false positives, $c_m$ can be increased or $c_f$ reduced, or both. However, in SOT, there is no possibility of an ID switch as there

is only one subject being tracked, thus the third term in the numerator of MOTA falls away. We compute the adapted SOTA metric

$$\text{SOTA} = 1 - \frac{\sum_{t=1}^{N_{frames}}(c_m(m_t) + c_f(fp_t))}{\sum_{t=1}^{N_{frames}} N_{objects}^t}. \tag{3.2}$$

Knowing the total quantities of successful and failed hypotheses over a given video sequence, we are able to define three further metrics. Recall is the ratio of the correctly-matched detections to the total number of ground truth objects in the sequence, and is computed

$$\text{Recall} = \frac{\sum_{t=1}^{N_{frames}} N_{tp}^t}{\sum_{t=1}^{N_{frames}} N_{objects}^t} \tag{3.3}$$

where, for a frame $t$, $N_{tp}^t$ is the number of true positives. Precision is the ratio of correctly-matched detections to the total detections actually made in the sequence, computed

$$\text{Precision} = \frac{\sum_{t=1}^{N_{frames}} N_{tp}^t}{\sum_{t=1}^{N_{frames}} (N_{fp}^t + N_{tp}^t)} \tag{3.4}$$

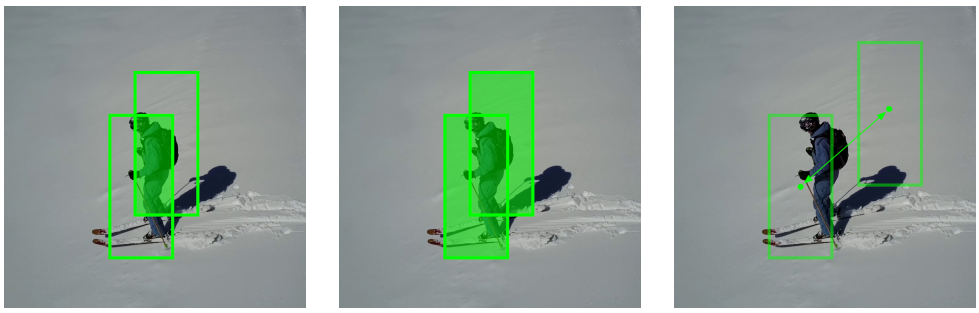
where $N_{fp}^t$ is the number of false positives. False Positives per Frame (FPF), discussed by Yang, Huang, and Nevatia (2011: 1238), is the number of false positive detections per frame, averaged over the sequence. FPF is given by

$$\text{FPF} = \frac{\sum_{t=1}^{N_{frames}} N_{fp}^t}{N_{frames}}. \tag{3.5}$$

**Detection precision.** Here we do not use precision in the classification sense, but rather to quantify how the size and position of the hypothesis bounding box compares to the ground truth bounding box. We need to introduce first the idea of Intersection over Union (IoU), also known as Single Object Detection Precision (SODP), which measures the affinity of two bounding boxes using overlap. Put simply, it is the ratio of the overlapping area of two rectangles to the total area of the rectangles. Intersection and union are illustrated in Figure 3.2. SODP is adapted in this report from a measure called Multiple Object Detection Precision (MODP), detailed by Kasturi et al. (2008: 327), which is applicable to MOT. In particular, for a frame $t$:

$$\text{SODP}(t) = \text{IoU}(t) = \frac{|A_o^t \cap A_h^t|}{|A_o^t \cup A_h^t|} \tag{3.6}$$

where $A_o^t$ denotes the area of the ground truth object rectangle in the $t$th frame, and $A_h^t$ denotes the area of the hypothesis object rectangle.



a) Intersection b) Union c) Distance

Figure 3.2: Examples of the IoU and distance measures used in the tracking metrics.

In the case that there is no mapped object pair in the given frame (either because the detection is missed, or because no ground truth object exists in the frame), SODP is forced to a value of zero for that frame. SODP can be thought of as the average spatial similarity of detected bounding boxes to ground truth bounding boxes in a sequence. The normalised SODP gives the precision for an entire sequence and is computed

$$\text{SODP} = \frac{\sum_{t=1}^{N_{frames}} \text{SODP}(t)}{N_{match}}. \tag{3.7}$$

We note here a problem encountered during the development of these evaluation metrics. Many of the metrics discussed in this section consist of terms including, but not limited to: *misses*, *false positives*, and *positives* - quantities that are computed based on whether a hypothesis made by the tracking system is valid or not. As shown in Figure 3.1, when a hypothesis is made and an object exists in the frame, the validity of the hypothesis is determined by distance. If a minimum threshold distance is exceeded, the hypothesis is invalid. Many of the other metrics are then computed on the basis of this validity, which means distance is the primary measure on which those metrics depend. However, it would indeed be possible to declare validity on something other than distance, for instance using overlap, as defined in Equation 3.6. All of the ensuing metrics would then be based on overlap of the hypothesis and object bounding boxes, instead of on distance. We were faced with two options: use both distance and overlap to determine validity, and calculate the full set of metrics for each; or, select just one of overlap and distance to determine validity, and use the other as part of the set of independent evaluation metrics.

We elected to go with the latter option, and use just distance to determine validity. Then, as shown in Equation 3.7, average overlap would be computed as an independent metric. Our rationale is that if the overlap between two bounding boxes is above some threshold value, they are assumed to be close enough that the direction the drone camera

is pointed in will encompass the target within the frame, but not necessarily be able to centre it. Thus we decided that maximising overlap would not optimise tracking as much as minimising distance would.

**Tracking completeness**. As noted by Luo, Zhao, and Kim (2014: 11), "metrics for completeness indicate how completely the ground truth trajectories are tracked." Mostly tracked (MT), mostly lost (ML), and partially tracked (PT), refer to the proportion of the ground truth trajectory's length during which it is covered by the output of the tracker. The definition of 'covered' was not found in any documentation available to this author. Thus it is assumed to mean there is some overlap between the tracker output and the ground truth object, and we define a minimum threshold overlap value, $T_{iou}$, which must be exceeded in order for the hypothesis to count. This implies that neither false positives nor misses are considered when measuring completeness performance. MT means the trajectory was covered for more than 80% of its length, ML for less than 20% of its length, and PT for between 20 and 80% of its length.

The literature defining these metrics makes no reference to the mathematics behind those measuring completeness. In order to calculate the completeness metrics, we need first to derive a function with binary output denoting whether or not a frame $t$ can be considered 'covered'. This function is given by

$$\text{Covered}(t) = \begin{cases} 1 & \text{if } IoU(t) \geq T_{iou}, \\ 0 & \text{otherwise.} \end{cases}$$

Summing this result for each matched frame in the sequence and dividing by the total number of matches produces the average proportion of covered frames for a sequence, defined as Completeness:

$$\text{Completeness} = \frac{\sum_{t=1}^{N_{frames}} \text{Covered}(t)}{N_{match}}.$$

We can now segment the range of outputs of Completeness into three categories, each one corresponding to one of the three completeness metrics. These are binary metrics, and refer to an entire sequence $s$. Mostly tracked (MT), given by

$$\text{MT} = \begin{cases} 1 & \text{if Completeness} \geq 0.8, \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

partially tracked (PT), which is given by

$$\text{PT} = \begin{cases} 1 & \text{if } 0.2 < \text{Completeness} < 0.8, \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

and mostly lost (ML), which is computed

$$\text{ML} = \begin{cases} 1 & \text{if Completeness} \leq 0.2 \\ 0 & \text{otherwise} \end{cases}. \tag{3.10}$$

One last metric here, fragmentation (FM), measures the "number of times a ground truth trajectory is interrupted in the tracking result" (Luo, Zhao, and Kim 2014: 10). Here, interrupted means that there is no tracker output for at least one frame (a miss), or the IoU between the ground truth object and the tracking output is below $T_{iou}$ for at least one frame (a false positive). The FM metric is normalised over the number of ground truth objects in the sequence. We note that the FM result is the complement of *SOTA*, introduced in equation 3.2, and is computed

$$\text{FM} = \frac{\sum_{t=1}^{N_{frames}} (c_m(m_t) + c_f(fp_t))}{N_{objects}}. \tag{3.11}$$

Finally, a generic performance metric important for all time-critical applications is speed. In this case, we are interested in the number of frames that the tracking algorithm is able to process per unit time. We denote this metric as FPS, or frames per second. In particular, for a tracking algorithm $a$, running on a video sequence $s$ of resolution $r$:

$$\text{FPS} = \frac{N^s_{frames}}{\sum_{t=1}^{N^s_{frames}} \triangle T^t_{track}} \tag{3.12}$$

where $\triangle T^t_{track}$ is the time taken to run tracking on frame $t$, and $N^s_{frames}$ is the number of frames in sequence $s$.

# Relevant Literature

Bewley et al. (2016: 1–5) explore an approach to multiple object tracking (MOT) which focuses on associating detected objects to existing tracks. Their model is based on tracking-by-detection - a type of visual tracking where detection is carried out on each frame, as opposed to running detection once on the first frame and then handing over to the tracker. Their system performs online tracking, where only past and current frames can be used.

Table 3.1: Evaluation metrics applicable to SOT (Luo, Zhao, and Kim 2014: 10). An up arrow indicates increasing performance with increasing quantity, and vice versa. The letter $B$ indicates a binary metric; a $+$ sign indicates desired performance corresponding with the positive binary result, and vice versa.

| **Metric** | **Description** | **Note** |
|---|---|---|
| SOTP | Average distance between centroids of ground truth and system output for successful hypotheses | $\downarrow$ |
| SOTA | Combines false negatives and false positives without weighting factors | $\uparrow$ |
| Recall | Correctly matched detections as proportion of total ground truth objects (of a sequence) | $\uparrow$ |
| Precision | Correctly matched detections as a proportion of total detections (of a sequence) | $\uparrow$ |
| FPF | Number of false positives per frame averaged over number of objects | $\downarrow$ |
| SODP | Average overlap between ground truth and system output for successful hypotheses | $\uparrow$ |
| MT | The ground-truth trajectory is covered by the tracker output for more than 80% of its length | $B+$ |
| ML | The ground-truth trajectory is covered by the tracker output for less than 20% of its length | $B-$ |
| PT | The ground-truth trajectory is covered by the tracker output for between 20% and 80% of its length | $B-$ |
| FM | Number of times that a ground-truth trajectory is interrupted in the tracking result, normalised over sequence | $\downarrow$ |
| FPS | Frame rate at which tracking algorithm runs | $\uparrow$ |

For MOT in general and for tracking-by-detection algorithms in particular, the problem is defined as one of data association. That is, the association of detections across frames. The authors note that they attempted to keep their approach as simple as possible: it would use only the position and size of the bounding box for both motion estimation and data association, and no additional appearance features. Further, they argue that re-identification is too complex, and that using it introduces too much overhead into the tracker. Importantly, they note that rather than trying to design the model to be robust to detection errors, they focus on making only high quality detections, thereby solving the detection problem directly. For clarification: whereas a robust model contains "components to handle various edge cases and detection errors", Bewley et al. (2016: 2) designed the algorithm to make only high quality detections in the first place. For this, they use a modern CNN. They combine the CNN-based detector with a Kalman filter for predicting motion of the target, and use the Hungarian algorithm to solve the data association problem between detected objects across frames.

Their methodology, beginning with the detection component first, is as follows: The CNN-based detector used is known as Faster R-CNN, detailed in Chapter 2. They comment that the end-to-end nature of this CNN and the fact that the detector architecture can be swapped out for a more effective one made it desirable. For their task, they were only concerned with the detection of pedestrians, and hence ignored or suppressed the detections of all other classes. They introduced a threshold confidence value of 50% to ensure that only the more likely detections were passed to the tracking framework. They note "that the detection quality is highly influential on the tracking performance".

Bewley et al. (2016: 3) use a representation and motion model to predict a target's identity into a new frame. The model approximates inter-frame displacements with a linear constant velocity model. The state of the target is given by

$$\mathbf{x} = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}]^T$$

where $u$ and $v$ are the horizontal and vertical coordinates of the target centre, respectively, $s$ is the target scale and $r$ is the target aspect ratio, which is assumed to be constant. Bewley et al. (2016: 3) write:

> When a detection is associated to a target, the detected bounding box is used to update the target state where the velocity components are solved optimally via a Kalman filter framework. If no detection is associated to the target, its state is simply predicted without correction using the linear velocity model.

Finally, the data association model's task is to assign a detection to an existing target. To do this, the target's bounding box geometry is estimated by its new location in the current frame. The assignment cost matrix, which maps out the confidence levels of each detection-target combination, is computed as the intersection-over-union (IOU) distance between each detection and all the predicted bounding boxes from existing targets. The assignment is solved via the Hungarian algorithm, and a minimum IOU value $IOU_{min}$ is imposed to reject poor detection-to-target assignments. They further note that using IOU distance implicitly handles target occlusion (Bewley et al. 2016: 3).

In Coursera's Deep Learning for Computer Vision course, Konushin and Artemov (2018) take the viewer through a variety of examples of visual object tracking, starting by defining two evaluation metrics for object tracking. The first, accuracy, measures the average overlap over a series of successfully tracked frames (presumably, overlap of the target in each frame). The second, robustness, measures the number of times the tracker drifts off target in some number of frames. They then mention that there are typically three ways of representing an object within a frame. The first is with a template: a set of pixels is defined as belonging to the object either with a bounding box, polygon or some sort of mask. The second is as a set of parts or fragments. This is a method used often in the film industry, whereby small locators are placed at designated positions on a human's body and systems of multiple cameras are set up to track them. Lastly, we can use appearance features to represent an object, such as a colour histogram of the pixels belonging to the object.

The authors then begin with some examples of tracking methods falling into the above three object representations. The first is known as template matching. In a frame, we record an image of the object in question and refer to it as template $\mathbf{t}$. For each new frame, we search for the new location of $\mathbf{t}$, typically using a sliding window approach: for each window, do a per-pixel comparison between the template and the window using a distance metric. Two such metrics are normalised cross-correlation (NCC) and sum of squared distances. This creates a score map.

The next example is called colour-based tracking, which is effective when the contrast between the target object and the background is high. This method computes the likelihood of a colour sample of belonging to the target or the background. We first compute the colour histogram for the target region inside a bounding box. Second, we compute a colour histogram for the neighbourhood around the target. Then, for each colour value in a new frame we compute the likelihood that it belongs to the target and the background.

A key problem in this setting is similarities of target colours to other objects in the same category as the target. For instance, a human target might be well contrasted with a white, snowy background, but poorly contrasted with another human in the frame.

However, the authors comment on how we can detect these 'distractors'. First we need to detect them, so we begin by applying the object colour model. Then we compute the colour histograms at each of the distractor regions, which makes the correct target feature more prominently. Finally, combine both the colour histograms of all the objects in the target's class with that of the target in a linear combination. This produces a high likelihood for discriminative object pixels and decreases the impact of distracting regions.

Moving on, the authors note that colour distributions rely on pixel values to calculate target vs. background and as such have no concept of 'locality,' or where in the image the target is. This makes colour matching robust to changes in shape, but sensitive to blur and changes in illumination. Template models, on the other hand, rely on spatial data. This makes them robust to blur and illumination changes, but sensitive to changes in shape. STAPLE, or Sum of Template And Pixel-Wise Learners, is an example of these two methods combined, exploiting the advantages of each.

Lastly, they refer to the use of CNNs for tracking. One method, of which the MDNet tracker is an example, is to train an object vs. background classifier and apply it to sampled candidate regions within an image, selecting the region with the highest score as the target. Another, of which GOTURN is an example (see description below), is to train a CNN regressor to predict the position of the target in the next frame. It is trained on pairs of sequential frames, and has been proven to run at 100 fps on a consumer GPU.

In the research for their master's thesis, Börjesson and Hillborg (2018: 13–21) attempt to modify an off-the-shelf DJI drone for the purposes of object tracking and autonomous flight. Their use case was for search and rescue - a drone would be deployed to locate, track and approach a liferaft (the target) in the open sea, thereafter dropping a line down to the liferaft which would in turn be used to reel the liferaft in towards a rescue vessel.

The tracking system would use as its input the video feed from the drone camera. The drone connects via radio signal to its proprietary remote controller. A video feed is output from the controller and into a standalone computer system (the PC). The PC analyses each frame and using a convolutional neural network (CNN) determines whether or not an instance of the target exists in a particular frame. Should one exist in the camera's field of view, the location of the target within the X-Y frame is recorded in order to determine its offset from the centre of the frame.

Knowing the offset enables the camera to be rotated about its transverse (pitch) axis or the drone itself to be rotated around its vertical (yaw) axis such that the target is centered in the frame. Once it is centered horizontally, the compass direction of the target will correspond to the current heading of the drone. Once it is centered vertically, the known tilt (pitch) position of the camera and the known altitude of the drone can be used to determine the distance to the target. Should it be within range, the drone then proceeds towards the target for rescue.

The focus of the work, the imaging system, has the following properties. It is an image classifier in the form of a convolutional neural network. The model architecture is based on VGG-16 (detailed above). It consists of 13 convolutional layers and three fully-connected hidden layers. All the convolutions are $3 \times 3$. The output layer is of size $1 \times 1 \times 1000$, as the ILSVRC competition has 1000 classes.

For this project, the researchers used the existing trained weights of the first 14 hidden layers in VGG-16, but retrained the final two layers using image data of their own, as liferaft was not one of the 1000 classes in the original dataset. The output layer was modified to contain just three classes: liferaft, water, and other.

The image dataset created by the researchers contained images captured by a drone of a liferaft placed on land. Later, augmented images were created by replacing the backgrounds in some of the images with water to simulate an ocean environment. The performance of the CNN to locate a liferaft in an image was satisfactory. This result, however, should be taken with a pinch of salt - the test data contained only images captured and modified by the researchers themselves and is not representative of real-world conditions.

In another relevant work, L. Chen et al. (2018) combine a variety of computer vision and object detection frameworks to develop a robust and efficient system for tracking multiple people.

They write that "online multi-object tracking is a fundamental problem in time-critical video analysis applications. A major challenge in the popular tracking-by-detection framework is how to associate unreliable detection results with existing tracks" (P. Chen et al. 2017: 131). The authors propose a sort of ensemble model by considering the outputs of both the tracker and the detector for each and every frame: "detection results of high confidence prevent tracking drifts in the long term, and predictions of tracks can handle noisy detection caused by occlusion" (P. Chen et al. 2017: 131).

"The objective of multi-object tracking is to estimate trajectories of objects in a specific category" (P. Chen et al. 2017: 131). In the commonly-used method of tracking-by-detection, an object detector is applied to each frame, and detections across frames are associated to generate object trajectories. There are two problems with this approach: it relies on consistently good detections, and it often struggles with intra-category occlusion (in other words, when objects of the same type pass in front of one another).

They write that some studies have proposed what is known as a batch-mode, whereby frames from a specified temporal window are grouped together and use future frames to smooth out one-off detections or noise. While this can be effective, it is simply not viable for time-critical applications, like where the results of tracking are required for a realtime feedback loop. Thus, this proposed system focusses only on the current and previous frames.

The novel method proposed aims to mitigate unreliable detections made in online mode, by optimally selecting candidates from outputs of both the tracker and the detector in each frame. The authors comment on a similar work by Yan et al. (2012), which proposed treating the detector and tracker as independent entities. The results from each of them would be kept as candidates selected using techniques like colour histogram, optical flow and motion features. P. Chen et al. (2017: 132) write:

> The intuition behind generating redundant candidates is that detection and tracks can complement each other in different scenarios. On the one hand, reliable predictions from the tracker can be used for short-term association in case of missing detection or non-accurate bounding. On the other hand, confident detection results are essential to prevent tracks drifting to backgrounds in the long term. How to score outputs of both detection and tracks in an [*sic*] unified way is still an open question.

In his web article about implementing OpenCV's in-built tracking algorithms, Mallick (2017) briefly describes the underlying machinery for each of the eight trackers. Here we expound upon each of them.

**Boosting** is the first of the eight tracking algorithms inheriting from OpenCV's `Tracker` class. It uses an online version of the *AdaBoost* algorithm, which combines the output of multiple weak classifiers into a weighted sum to represent the final output of the boosted classifier, with performance exceeding any of the individual underlying classifiers. Here,

online is not interpreted as we explained earlier (referring to whether the algorithm uses only past information, or future *and* past information), but rather that the classifier is trained at runtime with positive and negative examples of the object to be tracked. The bounding box supplied to initialise the tracker is treated as the positive example, and everything in the image outside of the bounding box is treated as background, or negative examples. For each new frame, the classifier is run on each pixel in the proximity of the previous object location and its score recorded. The new location of the object is where this score is at its maximum. This new location is treated as the newest positive example for the classifier, which continues to train with each new positive example in an online manner.

**MIL**, or Multiple Instance Learning, is a tracker which builds on the Boosting algorithm described above. Whereas Boosting considers only the current location of the object as a positive example, MIL looks at the neighbourhood of the current positive example to generate other potential positive examples. Positive and negative examples are not specified as such, but rather placed into groups known as *bags*. The examples from the neighbourhood of the known positive example are placed into a positive bag, increasing the likelihood that the positive bag contains the image with the updated location of the object.

The next tracker, **KCF**, standing for Kernelised Correlation Filters, builds upon both Boosting and MIL. In the words of Mallick (2017), it goes a step further than MIL to exploit the large overlapping regions between the potential positive examples to improve the speed of tracking.

**TLD**, or Tracking-Learning-Detection, is an algorithm detailed by Kalal, Mikolajczyk, and Matas (2011: 1–14) in their paper on its development. As the name suggests, TLD decomposes the long-term tracking task into three minor tasks: track (short-term), learn and detect. The "tracker follows the object from frame to frame. The detector localises all appearances observed so far" and, if necessary, "corrects the tracker." The learner estimates the detector's errors and updates the detector to ensure it avoids the same mistakes in the future.

Varfolomieiev and Lysenko (2016: 527–34) discuss another tracker, **Median Flow**, in their paper on an improved version of the algorithm of the same name. In their words, the essence of the Median Flow algorithm is to estimate in sequential frames the location of an object using sparse optical flow. This tracker assumes the object consists of small, rigidly connected patches moving synchronously together with the motion of the whole object. A so-called forward-backward error is computed by tracking the object in both forward and backward directions and measuring the discrepancies between the two trajectories. Minimising this forward-backward error allows the tracker to both reliably detect failures, and select reliable trajectories.

The only algorithm included in OpenCV to make use of a neural network, **GOTURN**, or Generic Object Tracking Using Regression Networks, is detailed by Held, Thrun, and Savarese (2016: 1–26) in their original paper. It uses a convolutional neural network regressor trained on pairs of consecutive images to predict the position of the object in the next frame. The tracker learns a generic relationship between an object's motion and

appearance and as such can be used to track novel objects that do not necessarily appear in the training set. Because all model training is performed offline (i.e., not at runtime), this tracker is able to run at a high frame rate.

Bolme et al. (2010) discuss the tracker known as **MOSSE**, or the Minimum Output Sum of Squared Error. It uses adaptive correlation for object tracking which produces stable correlation filters when initialised using a single frame. It is also able to detect occlusion based on a measure known as the peak-to-lobe ratio, allowing the tracker to pause and resume where it left off after the object reappears.

The final tracker of the OpenCV eight is **CSRT**, or Channel and Spatial Reliability Tracker. This one uses only two standard features: colour names and histogram of gradients. The spatial reliability map is used to adjust the filter support to the part of the selected region from the frame for tracking (Mallick 2017).

Moving away from OpenCV, Gordon, Farhadi, and Fox (2018: 1–8) detail their development of a neural network-based tracker named $Re^3$: Real-time Recurrent Regression Networks for Visual Tracking of Generic Objects. They note that robust tracking requires considerable information about the object being tracked, including its motion, its appearance, and how that appearance changes over time. They note further that in real-time applications, there is no opportunity to specify ahead of time the types of objects to be tracked, necessitating a tracker's ability to track generic ob.jects. $Re^3$ (to be referred to as Re3 throughout this work), is an end-to-end trainable recurrent neural network which utilises LSTM (long short term memory) units to keep track of historic aspects of the object's appearance. The network is trained once, offline, on a large image and video dataset containing a high number of object classes. Because it requires no training at runtime, inference (which in this case is the tracking task), is extremely fast and the authors report achieving frame rates as high as 150 fps using an Intel Xeon CPU/Nvidia Titan X GPU combination.

Similar to the GOTURN tracker, during training Re3 is passed pairs of consecutive image crops: the first is cropped to be centered on the location of the object in the previous frame, and the second is cropped to the same location but on the *current* frame. The idea is that by comparing the two images, the model learns how motion affects the pixels. They acknowledge that given a sufficiently fast-moving object, it is possible that it may have moved outside the frame of the second image crop. However, given the 2x padding afforded to each crop, the object would have to have moved 1.5× its own width/height in a single frame, which is unlikely. With skip connections, the object's appearance can be represented by high-, mid-, and low-level features by using the outputs from different convolutional layers in the model, providing a richer appearance model. The authors trained Re3 on two large datasets: Imagenet Video, and the Amsterdam Library of Ordinary Videos (ALOV), which combined contain over 4,000 videos annotated with bounding boxes on single objects. Their results, after testing on the Online Object Tracking benchmark (OTB), as well as the Visual Object Tracking (VOT) challenges from 2014 and 2016, are promising and indicate class-leading performance at very high frame rates. They also note that Re3 proves comparatively robust to occlusion relative to other trackers.

# 4 | Data

The **test data** is in the form of video clips filmed from drones, obtained from three sources: the first filmed over time by this author, the second filmed by members of FlyCam, and finally video obtained from a contemporary of this author (Rogers 2016). The content of the videos is varied, but remains relevant to the intended purposes of FlyCam and, as such, consists mostly of individual athletes taking part in such sports as skiing, snowboarding, mountain biking, and wakeboarding.

For the remainder of the report, when the reader is referred to a listing, it can be found in Appendix A, which contains listings for various blocks of Python code used throughout the project.

Figure 4.1: Screenshots of eight of the test videos, referred to as images 1 through 8, counting clockwise from top-left (Rogers 2017).

All of the content was originally filmed in 4K resolution ($3840 \times 2160$ pixels). 4K footage contains four times as many pixels as does 1080p footage ($1920 \times 1080$ pixels), and as such is typically more computationally intensive to analyse. The de-facto method for accessing live drone video is via the feed that the drone sends to the receiver. This feed is transmitted via radio (typically 2.4/5.8 GHz), at a resolution of either 720p ($1280 \times 720$ pixels), or 1080p. Though the live drone video feed is unlikely to be greater than 1080p, it might still be informative to retain the higher resolution clips to assess their impact on performance, if any.

Figure 4.1 provides an idea of the composition of the test video clips. The videos range from 2 to 37 seconds in length, and the scenes are selected to pose varying degrees of complexity or difficulty for the tracking algorithms against which the data will be tested. Generally, snow skiing (images 1 and 4 in Figure 4.1) is considered an easier environment for tracking, given the high contrast between snow and target - however, the intricacy of the trails in the snow left by the skier readily confuse tracking algorithms, as does the movement of the athlete into and out of shadows. Surfing clips, like that indicated by image 8 in Figure 4.1, are also considered difficult - the wave behind the target is constantly changing shape and spray often partially occludes it, and at times it is completely

Figure 4.2: Animation showing a sample of five of the unannotated test video clips. The video is also available at the following URL: https://youtu.be/JTao6qT87bE.

occluded inside the barrel. There is a clip (not pictured) of a car moving steadily across a green landscape which might be considered an easier clip on which to perform tracking: the car moves slowly, steadily, and barely changes scale throughout. The few mountain biking clips are tricky: they normally portray the subject from unusual angles (bird's eye view), and the environment surrounding and behind the athlete changes rapidly as he rides berms and straddles features.

The first working prototype explored was detailed in a guide called OpenCV Object Tracking (Rosebrock 2019). The author takes the viewer through the steps required to set up an object tracker in Python, using the OpenCV library for Python. The tracking code is simple to set up and fairly straightforward. It is worth noting a few things. Firstly, the code does not provide for any sort of automated object detection. The user is required to specify the object to be tracked by drawing a bounding box around it. As written in the article, the code does not automatically pause a video at the first frame and wait for the user to provide a bounding box - it will play until a particular key is pressed. The code was modified in such a way that the video feed would pause on the first frame to allow the user to define the tracking area. On pressing the `Space` or `Enter` keys, the video would proceed and each frame tracked according to the bounding box specified.

OpenCV 4.1., the version used for this prototype, includes eight different tracking algorithms, most of which have been discussed in this report previously: Boosting, MIL, KCF, CSRT, MedianFlow, TLD, Mosse, and GOTURN. All eight methods are online trackers in that they only use past frame data, not future *and* past, in determining object trajectory. These algorithms are discussed in Chapter 3. Having successfully implemented this variety of tracking algorithms, we needed a means by which to theoretically evaluate the performance of each of them - as well as any other algorithms that might be implemented. The video data on which the performance of the algorithms would be evaluated has been detailed earlier in this chapter, and consists of different sporting activities believed to be representative of the FlyCam use case. However, raw, non-annotated video data is only useful for empirical performance evaluation; that is, observing by eye the output of each

tracking algorithm. The frame rate of the live video feed from the drone, and thus that of the test video data, is 30 frames per second (fps); fast enough that a human observer might miss a phenomenon or failure occurring over a sufficiently short period (one frame occupies just $1/30th$ of a second). Theoretical evaluation, then, using the evaluation metrics described in Chapter 3, necessitated annotated ground truth video data, against which to compare the output from the tracking algorithms.

The output of a tracking algorithm for each frame of a video sequence is simply a rectangular bounding box around the target. In order to effectively evaluate the output of the tracker, any ground truth data would thus need to be similarly annotated, with bounding boxes placed around the target intended to be tracked. The video sequence that the tracker receives as input is not visibly annotated: the known bounding box attributes are held back from the tracker in the same way class labels are held back during the validation stage of a classification task. Comparing the tracker output to the ground truth object data enables computation of the elements required by the evaluation metrics previously discussed, including presence, misses, position, distance, overlap, among others. In any event, annotated video data is difficult to come across, particularly so for drone video. The task, then, is to effectively generate ground truth bounding box annotations for each frame of each video sequence in the test data.

The preliminary test dataset detailed above comprises 18 video clips, at an average clip duration of 12 seconds and frame rate of 32 fps. If a human could somehow effectively annotate a single frame, that is, draw an accurate bounding box around the target, in a respectable time of five seconds, the entire annotation task would take

$$5\tfrac{seconds}{frame} \times 32\tfrac{frames}{second} \times 12\tfrac{seconds}{clip} \times 18\ clips = 34\,560\ seconds = 9\ hrs\ 36\ mins.$$

which is clearly not a practical use of anyone's time. The proposed solution repurposes an object detector to carry out the annotation. The object detector would iterate through each frame of each video sequence, detecting all objects and storing the bounding box information in a tabular format.

Table 4.1: Header row and example observation from the raw annotated video data.

| clip | frame | class_id | conf | x1 | y1 | w | h |
|---|---|---|---|---|---|---|---|
| Bike05.mp4 | 1 | 1 | 0.9994385 | 0.563 | 0.517 | 0.042 | 0.168 |

Table 4.1 indicates the information stored during annotation, where each row is a detected object. *clip* is the name of the video file, $frame$ is the position in the clip of the frame in which the detection has been made, *class_id* is the category of detection (1 for human, 2 for bicycle, etc), $conf$ is the confidence of the detector that the object is present, $(x_1, y_1)$ are the coordinates of the upper-left corner of the bounding box (measured from the upper-left corner of the frame), and $w$ and $h$ are the width and height of the bounding box, respectively. $x_1, y_1, w$ and $h$ are all relative values between zero and one. Before they are used, they are multiplied by the appropriate dimension of the frame: $x_1$ and $w$ are multiplied by width, $y_1$ and $h$ by height. Other ways of denoting

bounding box positions and dimensions include the coordinate method, which gives the upper-left and bottom-right corners of the bounding box: $(x_1, y_1), (x_2, y_2)$. The selected format matches that used by the tracking algorithms built into OpenCV. Table 4.2 gives a summary of the video test data, including the key specifications for each clip.

Table 4.2: Video data summary. The format of the duration value is first the number of seconds, followed by the number of frames remaining which don't make up a full second (based on the frame rate).

| **Clip** | **Duration(ss:ff)** | **Frame Rate** | **Resolution** |
|---|---|---|---|
| Bike02 | 03:08 | 29.97 | $1504 \times 846$ |
| Bike03 | 03:04 | 28.43 | $1504 \times 846$ |
| Bike04 | 03:04 | 29.97 | $1504 \times 846$ |
| Bike05 | 05:07 | 29.97 | $1504 \times 846$ |
| Bike07 | 02:14 | 29.97 | $1504 \times 846$ |
| Bike08 | 18:17 | 29.99 | $1920 \times 1080$ |
| Bike09 | 14:15 | 29.99 | $1920 \times 1080$ |
| Car01 | 05:06 | 29.97 | $1504 \times 846$ |
| Human01 | 02:04 | 28.25 | $1504 \times 846$ |
| Ski01 | 22:17 | 59.86 | $3840 \times 2160$ |
| Ski02 | 20:02 | 29.97 | $1920 \times 1080$ |
| Ski03 | 37:03 | 29.97 | $1920 \times 1080$ |
| Ski04 | 04:13 | 60 | $1920 \times 1080$ |
| Slalom01 | 03:24 | 25 | $1504 \times 846$ |
| Snowboard01 | 04:16 | 24.78 | $1504 \times 846$ |
| Sup01 | 14:22 | 25 | $1504 \times 846$ |
| Surf01 | 33:10 | 23.98 | $1504 \times 846$ |
| Wake01 | 25:05 | 29.97 | $1920 \times 1080$ |

As well as being faster than a human, it is assumed that the object detector used is very accurate: it should approximate both the size and location of the object well. Of course, there will be frames where it is not able to produce an accurate detection. For this event, the annotation algorithm will, in addition to storing the bounding box information for each detection, store the visual representation of each frame with any detected object bounding boxes overlaid. Once annotation has been carried out for all frames in all sequences, an observer can quickly flip through the frames to uncover any mistakes. A misplaced or missing bounding box can then be manually corrected in or inserted into the tabular data.

Many object detection algorithms benefit from running on hardware with access to a graphical processing unit (GPU) and high amounts of random access memory (RAM). Some machine learning libraries like Google's Tensorflow (TF), have been designed to break down tasks like training and inference of ANNs into highly parallelizable subroutines. GPUs, which comprise hundreds or even thousands of cores designed to execute such subroutines, often help to speed up processes which on a typical consumer CPU might be slow. Inference, which in our case refers to feeding an input frame to the object detector for the purposes of object detection, is one such task which sees performance efficiencies on GPU-based hardware. Google Colaboratory (or Colab) is a free, cloud-based integrated development environment (IDE) which offers GPU support via an Nvidia Tesla

K80 with access to around 12GB of RAM.

Colab is a notebook environment, where a notebook is a document in which Python code can be interspersed with markdown, and code blocks run individually, simulating an interactive Python shell. Notebooks grew in popularity through the use of Jupyter, a similar development environment which runs in the browser, but typically runs on the user's local machine as opposed to a remote server. Modifying a Python script which implements the Mask R-CNN object detection algorithm for inference, we developed a Colab notebook to carry out the test video data annotation procedure.

Algorithm 1 summarises this procedure, and the code which implements it is provided in Listing A.5. One input is required by this algorithm: `clip`, the video clip on which we would like to make detections. We begin by declaring an empty list, `frame_list`, in which each detection will be stored. We proceed to iterate over each frame, executing a forward pass through the object detector with each one for inference. If objects are detected in the frame, we iterate over them, appending each of them to `frame_list`. The variables stored for each detection are indicated in Table 4.1. When no frames remain in `clip`, we convert `frame_list` to tabular format, filter it to include only those detections we are interested in (a single detection of type *person* with the highest *confidence* per frame), and return the tabulated data as output. The code which implements this filtering procedure is provided in Listing A.2.

**Algorithm 1** Video Data Annotation

**Require:** `clip`
1: `frame_list` ← empty list
2: **while** `next_frame` in `clip` **do**
3: `results` ← forward pass through object detector
4: **if** `results` is not None **then**
5: **for** `object` in `results` **do**
6: Append bounding box data to `frame_list`
7: **end for**
8: **end if**
9: **end while**
10: Tabulate `frame_list`
11: Filter for relevant detections and return `frame_list`

Table 4.3 shows an example row from the test video data after Algorithm 1 has been carried out: we have removed all detections which were not of type *human*, and where there were multiple *human* detections, we retained the one with the highest confidence.

Table 4.3: Header row and example observation from the annotated video data after filtering.

| frame | x1 | y1 | w | h |
|---|---|---|---|---|
| Bike05.mp4 | 0.563 | 0.517 | 0.042 | 0.168 |

Object detectors are not perfect, and indeed the video annotation procedure sees some

proportion of frames with objects in them missed by the detector. This can be due to one of a number of reasons. Among others: the frames are highly compressed and the object is not clear, or simply that the detector does not detect with 100% reliability. Thus before moving on to the manual process of filling in detections for objects in frames which the detector used for Algorithm 1 may have missed, it is pertinent to check for and root out any obvious mis-detections. This way, detections which passed the filtering process by virtue of being of type *human* and/or of having the highest confidence but which were indeed not actually human objects could be removed. A rudimentary yet effective method was conceived to carry out this process. The CSV files of detections would be opened in Microsoft Excel, and data bars overlaid on each of the $x_1, y_1, w$ and $h$ values. A data bar is a horizontal single-axis measure of the numeric value in a cell, relative to the minimum and maximum values in the column. Having observed the video data in realtime, we know the motion of the objects to be relatively smooth; no object jumps any appreciable distance between frames. We can expect then that the data bars themselves would produce a smooth plot, and that any outliers would be easily detectable, as shown in Figure 4.3. Obvious outlier rows, encircled in red, are simply deleted from the table.

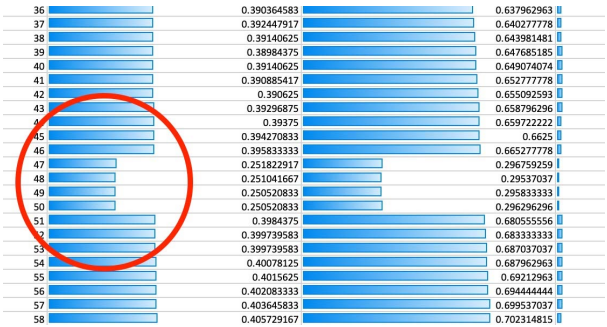



Figure 4.3: Excel's data bars feature used to root out obvious detection outliers from frame data.

In Algorithm 2 we summarise the process of filling in bounding boxes for objects in frames which were skipped or otherwise missed by the object detector in Algorithm 1. The relevant code for this algorithm is provided in Listing A.3. Two inputs are required here: `frames`, an incomplete table containing the bounding box data for each detected frame, and `clip`, the video clip pertaining to `frames`. We first set a counter variable `frame_count` equal to one, to check each frame in the video starting from the beginning. We also declare an empty list, `new_frames`, in which any new bounding box data will be stored. From Line 3 we begin iterating over frames in `clip`. The first step is to check whether the frame we are currently processing exists in the table of frame data. If it does, the object detector has previously found something in the frame and recorded it, so we draw the rectangular bounding box on it and show it to the user. If the current frame cannot be found in the frame data, the object detector did not record any objects, though we are not sure if it neglected to do so correctly or incorrectly. In any case, the frame is displayed to the user and the program awaits input (i.e., it pauses on and displays the unannotated frame). If the user's input is in the form of a drawn rectangle on the displayed window, the program saves the dimensions of the drawn rectangle after the user hits a confirmation key. The user can also choose to skip the frame without providing drawn input, for the case where an object might not exist in the frame. Before moving on to the next frame we increment `frame_count` by one. When no frames remain, `new_frames`

is converted from list to tabular format, inserted into `frames` and finally sorted by frame number, ascending.

**Algorithm 2** Fill Missing Frame Data for Single Video Clip

**Require:** `frames`, `clip`

```
frame_count ← 1
new_frames ← empty list
while next_frame in clip do
    if frame_count exists in frames then
        Draw BB on frame
    end if
    Show frame to user
    if frame_count not in frames then
        user_input ← prompt user for input
        if user_input is drawn then
            Append rectangle dimensions to new_frames
        else if user_input is keystroke then
            Continue to next frame
        end if
    end if
    Increment frame_count
end while
Tabulate new_frames, append to frames, sort frames
```

# 5 | Tracking and Evaluation Methodology

In Algorithm 3 we break down the process of evaluating different tracking algorithms using the metrics defined in Chapter 3, and the code which implements it is provided in Listing A.6. To be sure, this algorithm does not define how a tracker proposes object trajectories from one frame to the next; rather recording values for how well each different tracker is able to do that. Two inputs are needed by this algorithm: `clips`, a list of video clips to perform tracking on, and `trackers`, a list of names of the trackers to be evaluated. We begin by declaring an empty list object which will contain the data about each detected object. An example of the type of entry to be stored in this list is shown in Table 4.1. We then proceed to iterate over each video clip in the dataset, initialising a frame counter variable, `frame_count`, to the position in the clip in which an object first appears. In other words, the frame counter will not necessarily start at one, for an object might only appear for the first time later in the clip. A nested loop then iterates over the different trackers we wish to test. At the beginning of each iteration we declare and initialise two counter variables to zero: `miss_count` to count tracker misses and `fp_count` to count tracker false positives.

A further nested loop then iterates over each frame in the clip, starting from the one `frame_count` has been initialised with. For each frame, we first check to see if a tracker object, `tracker`, exists. If it does not, we initialise it by passing it the known bounding box of the object in the current frame: `bb_gt`. This should only happen once per clip, and always for the frame where the object first appears. If a `tracker` does exist we know that tracking has already begun and we have passed the first frame in which an object appears. We thus retrieve two artefacts from the tracker's last iteration: `bb_trk` and `state_trk`. The former is the proposed bounding box of the object in the current (newest) frame, the latter is a state flag indicating whether the tracker has failed or succeeded. When it has failed, `bb_trk` is Null: a miss has occurred and we increment `miss_count` by one. When it has succeeded, we compute the distance and IoU between `bb_trk` and `bb_gt`. If the distance exceeds our threshold, a false positive exceeds threshold has occurred and we increment `fp_thresh_count`. We append the distance to `dist_list`, and if IoU is non-zero, we append the IoU value to `iou_list`. We have now performed all necessary actions for a single frame and begin processing the next frame. Once each frame in the clip has been processed, the metrics for the current combination of tracker and video clip are computed and recorded by appending a row to `results_list`. We move on to the next tracker and repeat until no trackers remain, then move on to the next clip and repeat until no clips remain. Finally, we tabulate `results_list`, example rows from which are shown in Table 5.1. In Table 5.1, *Res* is the width of the clip in pixels, *Track* is the shortened name of the tracking algorithm, *IOU* is the minimum overlap threshold, *Dist* is the maximum distance threshold, and the remainder of the columns refer to the metrics as described in Chapter 3.

**Algorithm 3** Evaluate Tracking Performance

```
Require: clips, trackers, frames
results_list ← empty list
for clip in clips do
    frames ← Frame bounding box data for clip
    for tracker in trackers do
        frame_count ← first frame containing object
        miss_count, fp_count ← 0
        iou_list ← empty list
        while next_frame in clip do
            if Tracker not initialised then
                Create tracker object
                Initialise tracker with bb_gt
            else
                bb_trk, state_trk ← Updated bounding box, state from tracker
                if state_trk is False then
                    Increment miss_count
                else if state_trk is True then
                    Compare bb_trk to bb_gt; compute iou, dist
                    if dist exceeds dist_thresh then
                        Increment fp_thresh_count
                    end if
                    Append iou, dist to iou_list, dist_list
                end if
            end if
        end while
        Compute metrics
        Append metrics to results_list
    end for
end for
Tabulate results_list and document results
```

Up to this point we have not discussed the role that object detection algorithms play in the object tracking process. In reality, when a tracking algorithm is implemented, it is paired with an effective object detection algorithm. To initialise a track, the detector is run first to locate any relevant objects in the image. The location information of these objects is then passed to the tracker, which proceeds to generate the tracklet from frame to frame. Tracking continues until either of the following events occurs: the video stream comes to an end, or the tracking fails. Determining whether or not tracking has failed typically comes down to whether or not the tracking algorithm in use has the ability to self-report failure. Indeed, many of the algorithms explored for this project do not have the ability to self-report failure, and so even when the target object is objectively lost, the tracker still believes it is tracking successfully. This introduces unreliability both in testing and in implementation. In implementation, we desire to know exactly when tracking has failed, so that the tracker can be reset or reinitialised in order for tracking to continue. Indeed in the FlyCam use case, tracking failure for even only a few frames could mean the target athlete disappears out of frame, rendering the entire recording

Table 5.1: Header row and example observations from the tracker performance data.

| Clip | Res | Track | IOU | Dist | SOTP | SOTA | Rec | Prec | FPF | SODP | MT | PT | ML | Comp | FM | FPS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bike05 | 1280 | csrt | 0.4 | 0.1 | 0.0034 | 1.0 | 1.0 | 1.0 | 0.0 | 0.7998 | 1 | 0 | 0 | 1.0 | 0.0 | 14.74 |
| Ski01 | 1920 | mil | 0.4 | 0.1 | 0.0023 | 1.0 | 1.0 | 1.0 | 0.0 | 0.8078 | 1 | 0 | 0 | 1.0 | 0.0 | 17.92 |

useless, both for FlyCam and for the customer. During testing, it is pertinent to know about tracking failure when it happens, because many of the metrics defined in Chapter 3 are measured using the tracking hypothesis for each frame. For example, if the tracker is unable to self-report failure, it will continue to make a tracking hypothesis for each frame, even if it has long since lost track of the object. Referring again to Figure 3.1, this means that a tracker which cannot report failure will never record a false negative or a true negative, both of which require that a hypothesis is not made for a particular frame.

For those tracking algorithms that do have the ability to self-report failure, we ideally would like to immediately invoke the object detector again when tracking failure occurs. Running detection again in this event is akin to an in-situ reinitialisation of the tracker. However, in reality it is far from ideal, because object detectors are not capable of running at the same high frame rates as object trackers. For instance, assume a tracking algorithm running at 30 frames per second suddenly fails to continue tracking. We immediately invoke detection to find the object within the last known frame with the intention of handing off the detection information to the tracker such that it continues to do its job. A reliable, robust detector might be able to complete inference in 100-200 ms, which means that in the time taken to re-locate the target object, 5-10 frames have been missed, during which time the target object has likely moved on and could well be outside of the frame. Finding the target object again at this stage is nigh on impossible, as it relies on the camera continuing to move in the exact direction of the physical object and framing the object by chance. Possible reasons that tracking might stop are as follows: a) the tracker itself reports failure; b) the object is occluded (temporarily or permanently); or c) the maximum distance threshold is exceeded. Tracking failure as a result of exceeding the maximum distance threshold is not equivalent to whatever failure a tracker might report of its own accord. For this reason, we elected for the present report *not* to reinitialise the tracker with a new detection or ground-truth bounding box in the event of failure. Rather, we initialise the tracker only once - when the object first appears - and leave it to track for as long as possible.

Ideally, in a real-world implementation, the following sequence of events would occur: we initialise the tracker with the detected target object. Tracking commences, and continues until one of the three stopping cases listed above occurs. Immediately we run object detection on the last known frame, and reinitialise the tracker with the result of that detection. Theoretically, tracking continues, having missed a small number of frames due to the inferior speed of the detector. Therefore, in real-world implementations, the inference time *would* be a factor, and an important one to consider when choosing a detector.

We acknowledge that because tracking might cease for any one of a multitude of reasons, the results we record are perhaps not as comparable as we might like. For example, one tracker might accurately track the target throughout an entire clip; another might do so for the first 40% of the clip and then report that it has failed; and yet another might

complete 40% of the clip and not report failure, generating hypotheses for the remaining 60% of frames but with no success. In the first two instances, results would appear excellent for both, despite the obvious discrepancy between the quantity of frames tracked in each case. In the third instance, recorded results would be terrible as a result of 60% of hypotheses being incorrect, but in reality the results are no different from the second instance, in which exactly the same number of frames were successfully tracked. For this reason, during the computation of tracking performance, we include one additional measure known as TR, or *tracked ratio* - a simple ratio of matched hypotheses to total frames in the clip.

# 6 | Results

The primary means by which the evaluation metrics are generated is by running each tracking algorithm on each video clip in the test data suite. Slight tweaks to this strategy can be employed, such as varying the resolution of the video, or by simulating the presence of an object detector by reinitialising the tracker in the event of failure. We also note the presence of probabilistic algorithms, discussed above, which might not produce identical results each time they are run on the same data. In these cases, it would be pertinent to run the algorithm perhaps 10s or 100s of times on the same clip, and analyse performance by observing a distribution of the results (as opposed to observing the results from just a single run). However, under the present hardware constraints, it is not computationally feasible to run each tracker on each clip at varying resolutions 10s or 100s of times. As such, we propose to first identify those trackers with the most promising results, and generate results distributions on perhaps a small subset of the data at the specific resolutions only.

To begin the collection of results, we proceeded to implement Algorithm 3 in an Anaconda virtual environment, using Python 3 and a Jupyter Notebook. Having defined a list of video clips to iterate over, and a list of trackers with which to do the same, we encountered an issue: the OpenCV trackers would not allow the tracker instance to be reused within the programming environment without first interrupting and restarting the kernel (i.e., clearing the environment of variables). Though it was possible to iterate over the different tracker instances, it was not possible to use the same tracker for multiple video clips. OpenCV is a C++/C-based library and, as such, its Python framework is not thoroughly documented. Extensive investigation revealed an opinion that OpenCV will not let you reuse a tracker instance within an environment. The workaround was to run each of the trackers on a single video clip at a time, restarting the Python kernel for each new clip. Fortunately, the Re3 tracker is not OpenCV-based and was not subject to the same limitations - we were able to have Re3 run through all 18 video clips uninterrupted.

Having attempted to run each of the nine trackers on some subset of the clips, we found that: boosting, the first of the eight OpenCV trackers described above, was crippling the automation of the evaluation process. It would freeze partway through the tracking process unpredictably and inconsistently. Those clips that it was able to complete were so poorly tracked (as determined by observation and by the results produced) that we decided to simply leave it out of the evaluation process completely. With the remaining eight trackers (*MIL, KCF, TLD, Median Flow, GOTURN, MOSSE, CSRT* and *Re3*) having been run successfully on the video clips at their native resolutions, we noticed an interesting trend: frame rates for an individual tracker would vary with the resolution of the video clip in question, even among visually-similar clips. As such, we decided to resize each video clip to two smaller resolutions and run each tracker on three different resolutions of each clip. Three native resolutions were present in the video data: $1504 \times 846$, $1920 \times 1080$, and $3840 \times 2160$. We elected to group resolutions into three categories: High, Mid and Low, and refer to each as a *Res Group*. High would contain all

the native resolutions as listed above, Mid those resolutions resized to $1280 \times 720$, and Low those resolutions resized to $720 \times 404$ or $640 \times 360$. With tracker speed being such an important aspect of the performance question, we deemed it necessary to determine what effect resizing the video stream before performing tracking on it would have.

The hardware on which the testing was carried out is inferior to what it is assumed would be deployed for the FlyCam implementation. We used a top-end MacBook Pro with a 4th-generation 2.5GHz quad-core Intel i7, 16GB RAM and without GPU access. The CPU featured Intel Hyperthreading technology, creating eight virtual cores from the four physical ones. As discussed previously in this report, convolutional neural networks typically see impressive performance upticks when running on GPUs, especially when using Tensorflow-GPU. As such, tracking speeds listed in the results (particularly for the CNN-based trackers GOTURN and Re3) can be assumed to be lower than they would be in reality where tracking and detection would be carried out with, and programmed specifically for, a GPU.

The names by which the metrics are referred to in this chapter are largely as they were presented in Chapter 3, with a few exceptions. *Prec* refers to Precision, *Comp* to Completeness, and *TR* to Tracked Ratio. TR is the ratio of $N_{match}$ to $N_{objects}$, as described in Equations 3.1 and 3.2. Clearly, a higher TR value is preferred. With all the results generated for a single run of each tracker, on each video clip at each of the three resolutions, another phenomenon was then exposed: the MOSSE tracker results were full of zeroes. Fortunately, the evaluation algorithm had been implemented to save a video of the tracking output for each run. We examined each clip that MOSSE had attempted tracking on and found that for most clips (85%+), the tracker had been passed an initialisation bounding box, and then simply failed to begin tracking. The occurrence was again unpredictable, taking place on some clips and not others and on repeated runs would only occur some of the time, even on the same clip. An internet investigation revealed no relevant anecdotes and we elected to drop the MOSSE tracker from the analysis.

Two further irregularities were revealed after the results had been generated: the first was that the GOTURN tracker simply refused to run to completion on the clip entitled *Car01*. Even after fetching the original Car01 video file and confirming GOTURN's compatibility with each other clip, the tracking always froze around 10% of the way into the video. Given that there are 18 clips and that Car01 does not represent the type of content we anticipate for the FlyCam use-case, we were happy to proceed with the use of GOTURN on the other clips. The second was that it became apparent the SOTA and Recall metrics were taking on identical values for each run. After ensuring that the Python functions for each were indeed computing independent results, we returned to the definitions as presented in Chapter 3. Examination of each of the formulae reveal an intuitive mistake. Firstly, the metrics share the same denominator (the total number of ground truth objects in the clip). Secondly, recall is a measurement of the total number of true positives, while SOTA is a measurement of the *complement* of the total number of misses and false positives, which is equal to the number of true positives. One of them could thus be left out of the results, and recall remains. After all the single-run tests were completed, it also became apparent that the FM metric is the complement of Recall (and of SOTA by association).

We are now able to consolidate the results from all the single runs. That is, we have run each of the seven remaining tracking algorithms on each of the 18 video clips at three different resolutions (with the exception of GOTURN, which was run on only 17 clips), providing us with 375 observations from which to draw initial conclusions about the tracking performance. We note here that two of the *hyperparameters*, as it were, of the tracker evaluation suite are the thresholds for distance and IoU, discussed in Chapter 4. For initial testing, $T_{iou}$ and $T_{dist}$ were set to 0.4 and 0.1, respectively. $T_{iou}$ is a lower limit, defining a minimum overlap threshold, and $T_{dist}$ is an upper limit, defining a maximum distance between bounding box centroids. Both are unit-less and exist in a range $[0, 1]$. Table 6.1 summarises these results: each cell represents the mean value of a particular metric for a particular resolution group, averaged over 18 video clips. Each metric has its own column and is accompanied by an up- or down-arrow to indicate whether a higher or lower value, respectively, is preferred. Additionally, each column is colour-scaled such that darker cells represent more optimal intra-column values. The colour palettes used in the data visualisation in this chapter are effective both in grayscale and in colour, and vary enough in hue and saturation to be effective at all brightness levels. The shading means the viewer needn't consider each cell value individually (tedious), and can rather at a glance observe the performance of a tracker across different resolution groups.

Table 6.1: Mean single-run tracking results for each evaluation metric, grouped first by tracker, then by resolution group. Direction of arrow (up/down) indicates whether higher or lower values preferred, respectively. Cells colour-scaled according to column values: darker cells represent more optimal values.

| Tracker | Res Group | SOTP ↓ | Recall ↑ | Prec ↑ | FPF ↓ | SODP ↑ | Comp ↑ | MT ↑ | PT ↓ | ML ↓ | FM ↓ | FPS ↑ | TR ↑ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CSRT | High | 0.017 | 0.884 | 0.932 | 0.059 | 0.582 | 0.833 | 0.667 | 0.333 | **0.000** | 0.116 | 19.026 | 0.916 |
| | Mid | 0.041 | 0.821 | 0.880 | 0.106 | 0.514 | 0.727 | 0.611 | 0.389 | **0.000** | 0.179 | 22.004 | 0.900 |
| | Low | 0.013 | 0.888 | 0.945 | 0.037 | 0.563 | 0.781 | 0.611 | 0.389 | **0.000** | 0.112 | 32.305 | 0.895 |
| GOTURN | High | 0.234 | 0.582 | 0.555 | 0.442 | 0.195 | 0.231 | 0.118 | 0.235 | 0.647 | 0.418 | 10.927 | **1.000** |
| | Mid | 0.306 | 0.575 | 0.548 | 0.450 | 0.197 | 0.231 | 0.118 | 0.176 | 0.706 | 0.425 | 11.744 | **1.000** |
| | Low | 0.258 | 0.490 | 0.463 | 0.533 | 0.184 | 0.230 | 0.118 | 0.176 | 0.706 | 0.509 | 14.480 | **1.000** |
| KCF | High | 0.010 | 0.407 | 0.991 | 0.007 | 0.678 | 0.846 | 0.667 | 0.278 | 0.056 | 0.593 | 45.600 | 0.413 |
| | Mid | 0.009 | 0.314 | **0.992** | **0.006** | 0.703 | 0.899 | 0.833 | 0.111 | 0.056 | 0.686 | 52.313 | 0.320 |
| | Low | **0.008** | 0.131 | 0.991 | **0.006** | **0.828** | **0.928** | **0.889** | **0.056** | 0.056 | 0.869 | **89.135** | 0.138 |
| Median Flow | High | 0.136 | 0.577 | 0.671 | 0.264 | 0.415 | 0.576 | 0.333 | 0.389 | 0.278 | 0.423 | 42.565 | 0.824 |
| | Mid | 0.167 | 0.579 | 0.642 | 0.298 | 0.395 | 0.548 | 0.389 | 0.278 | 0.333 | 0.421 | 51.716 | 0.859 |
| | Low | 0.207 | 0.433 | 0.510 | 0.411 | 0.266 | 0.360 | 0.222 | 0.222 | 0.556 | 0.566 | 87.311 | 0.840 |
| MIL | High | 0.113 | 0.742 | 0.705 | 0.294 | 0.383 | 0.529 | 0.278 | 0.389 | 0.333 | 0.259 | 12.990 | **1.000** |
| | Mid | 0.110 | 0.731 | 0.712 | 0.287 | 0.377 | 0.502 | 0.167 | 0.611 | 0.222 | 0.268 | 14.614 | **1.000** |
| | Low | 0.090 | 0.773 | 0.744 | 0.254 | 0.349 | 0.477 | 0.167 | 0.611 | 0.222 | 0.228 | 17.343 | **1.000** |
| TLD | High | 0.209 | 0.545 | 0.543 | 0.454 | 0.275 | 0.417 | 0.167 | 0.444 | 0.389 | 0.455 | 7.872 | 0.992 |
| | Mid | 0.184 | 0.577 | 0.568 | 0.428 | 0.263 | 0.389 | 0.222 | 0.333 | 0.444 | 0.423 | 11.163 | 0.995 |
| | Low | 0.242 | 0.407 | 0.391 | 0.587 | 0.173 | 0.257 | 0.056 | 0.333 | 0.611 | 0.592 | 20.837 | 0.982 |
| Re3 | High | 0.051 | 0.885 | 0.850 | 0.149 | 0.590 | 0.806 | 0.722 | 0.222 | 0.056 | 0.116 | 10.716 | **1.000** |
| | Mid | 0.044 | **0.907** | 0.870 | 0.129 | 0.601 | 0.813 | 0.667 | 0.333 | **0.000** | **0.094** | 12.770 | **1.000** |
| | Low | 0.048 | 0.889 | 0.857 | 0.143 | 0.602 | 0.822 | 0.722 | 0.222 | 0.056 | 0.111 | 13.931 | **1.000** |
| *Mean* | | *0.119* | *0.626* | *0.731* | *0.254* | *0.435* | *0.581* | *0.416* | *0.311* | *0.273* | *0.375* | *28.636* | *0.861* |

In Figure 6.2 we visualise the single-run tracking results in a different way: each plot represents the results for a single metric (for instance, heights on the top left plot are

SOTP values). Each group of four columns are the tracking results from a single tracker (for instance CSRT, and each of the four columns is a resolution group (High, Mid, Low and Tracker Mean, the last being the mean value of the first three). The title of each plot includes parentheses containing the same indicator arrow (up/down) for whether a higher or lower value is preferred. With the exception of FPS, all the metrics are unit-less and map to the range zero to one, as they are all proportions. FPS is an absolute value with the unit *frames per second*.

With the aid of the data representations in Table 6.1 and Figure 6.2, we are able to draw a number of conclusions. The code which produces the visualisations in Figure 6.2 is provided in Listing A.4. Firstly, we note the dark horizontal band in the row of the KCF tracker. In eight of the 12 metrics, KCF achieves the optimal value (indicated in bold typeface). It manages the highest range of recall values of all the trackers, indicating a sharp drop in performance with decreasing resolution. However, the crucial number to do with KCF is TR: it has far and above performed the worst of all seven trackers in proportion of frames tracked. It seems peculiar that such good results for other metrics might accompany such poor track ratios, but it is somewhat simply explained. KCF is excellent at self-reporting failure, and it is also excellent at failing: observing the accompanying tracking output videos reveals that it seldom gets more than a few frames into tracking before giving up completely. Indeed, this means it is not at all robust - but if it were to be paired with a fast detector which could reinitialise it frequently, it might be improved. However, detectors, as discussed, are not very fast and we would prefer not to have to invoke them regularly. KCF's effectiveness at reporting failure is why its results for other metrics are so flattering: many of them only measure over those frames on which tracking is 'attempted', and if no hypothesis is made on a particular frame then it is not considered. This is, for example, why Precision (ratio of correct matches to all matches made) is high, but Recall (ratio of correct matches to all objects in sequence) is low. One promising metric for KCF is its high frame rate performance, producing a range of 45 fps for the High Res Group to 90 fps for the Low Res Group. Recall that we anticipate an incoming video stream frame rate of 30 fps for the FlyCam use case.

We note another dark horizontal band across the row of the CSRT tracker. While it achieves only one best-in-class result in its low ML value, it does score consistently well across the other metrics. At this point it is worth mentioning TR again: those trackers that achieved a 1.000 score for TR are in fact those which are not able to self-report failure, thus a tracking attempt is made on every single frame and all frames are considered tracked, leading to a perfect score. It is a good indication then, when a tracker scores a high-but-not-perfect result for TR. It means that it is able to report failure, but still attempted tracking on a high proportion of available frames. CSRT is one of those, with a narrow range of TR values between 0.895 and 0.916 from Low to High resolutions. Indeed, reviewing the tracking output videos reveals promising results, with CSRT managing tight bounding boxes throughout the duration of most clips, not obviously affected by changes in resolution. The very low SOTP values confirm the observation of tight bounding boxes, while the relatively high Completeness score paired with high TR values mean that a) tracking was attempted on most frames and b) objects were well-tracked on most of those attempts. Another encouraging result is the high frame rates achieved - presumably not shaded very darkly because of linear shading and KCF's results in that category being far greater. However, a result of 19 fps for tracking in high resolution

Figure 6.1: Animation showing three tracking failures: TLD running on Bike04, GOTURN running on Human01, and MedianFlow running on Wake01. The video is also available at the following URL: https://youtu.be/1TUR10kzy-Y.

video on a consumer CPU is a positive sign. Of course, a mean ML value of zero for all resolution groups is excellent, proving that CSRT was consistently able to achieve at least mediocre completeness on the entire video suite.

Looking down the TR column we see two more darkly-shaded cells with high-but-not-perfect scores: Median Flow and TLD. Scanning first the Median Flow row, we notice many results which are perfectly average. Importantly, we see a range of colour shades between Res Groups within the Median Flow row, indicating a drop in performance with decreasing resolutions, and in the cases of FPF and SODP, decreases as much as 60% between high and low resolutions. This is perhaps not too much of an issue given the high frame rates Median Flow appears to achieve (as high as 42 fps at high resolution), meaning the user could feed high resolution video into this tracker to obtain adequate performance without speed concerns. Moving onto TLD, we see good rates of completion in the TR column, but many cells within the row shaded white, indicating worst-in-class performance in many of the metrics at low resolution, and unremarkable results at high resolution. Of particular concern are the low SODP values, which indicate poor bounding box overlap when normalised over the matched frames. Indeed, when we review the tracking output video for TLD, we notice bounding boxes jumping around all over the frame and mostly in incorrect locations. Figure 6.1 gives an indication of TLD failing to track on the clip of a cyclist completing a jump.

Moving onto the MIL row, we are faced with largely unremarkable colours and values, many of which are explained by reviewing the tracking output video. Consultation with Figure 6.2 confirms the middle-of the-road performance of MIL (3rd column group from the right in each plot). SOTP is one of the few well-performing metrics for MIL, placing 4th overall out of the seven trackers in that category. In the output video we can see fairly consistent trajectories estimated by the tracker, seldom veering too far from the target object. This explains the solid SOTP (aka average distance) values. What we do notice, however, is that the bounding boxes tend to 'vibrate' around the target object, rather than moving smoothly along with it between frames. This explains the poor SODP (aka

average overlap) values. We also notice a tendency of the tracker to jump violently to new and incorrect positions within the frame, which might suggest why the Completeness values are so low (Completeness is a measure of the number of bounding boxes meeting the minimum overlap threshold). Frame rates for MIL form a range of 13 to 17 fps, which presumably would increase given the appropriate hardware.

Initial testing with GOTURN seemed promising: it appeared, albeit empirically, to maintain smooth trajectories and produce tight bounding boxes, when paired with an object detector or after initialisation with a user-defined bounding box. However, thorough testing on the video data across different resolutions has produced contrasting and conclusive results. GOTURN performs worst in class for almost every metric. It scored a perfect 1.000 for TR, which as discussed above means that the tracker is typically unable to self-report failure - thus considering every frame an attempt. The only other noteworthy result is for the PT metric, which was fairly impressive at a range of 0.235 - 0.176. However, this comes at the expense of both a high MT score, where GOTURN has achieved the worst result, and a low ML score, also achieving the worst results of all seven trackers. This is summarised well by GOTURN's Completeness results, the worst in the entire test - implying poor overlap values and reinforced by the lowest SODP results across the board. Clearly, the promising initial results were due to chance and perhaps because we were not testing GOTURN on a wide enough range of clips. In reviewing the output video for GOTURN, two results become clear. First, the tracker is highly sensitive to disturbances in the image very near to the object - messy or similar-looking patches near to the target object are easily able to 'hijack' the tracker from the desired trajectory. Second, when the tracker does not get distracted, we see a strange phenomenon of the bounding box slowly growing larger until it encompasses most of the frame. Given that this is a neural network-based tracker, we can say with considerable confidence that this is due to a difference between the data the model was trained on and the data we tested it on. That is, the dissimilarity in appearance of the objects we are tracking to the objects used to train the GOTURN regressor lead to small prediction errors carrying forward with each frame and slowly snowballing to produce giant bounding boxes. An example of this is given in the clip of a human jumping off a cliff in Figure 6.1.

The last of the seven trackers we need to critique is Re3. Recall that Re3 is the only tracker in the test arsenal that is not part of OpenCV's tracker module. Scanning the Re3 row in Table 6.1 reveals an almost-solid shade of navy blue, indicating positive results across all metrics, with the exception of FPS which is significantly less than KCF and Median Flow. Re3 achieves best-in-class results for four metrics: Recall, ML, FM, and TR. Its perfect TR score can be justified as follows. Unfortunately, Re3's authors did not build in any ability for the tracker to self-report failure, and in order to implement it in our test suite, we needed to force the *tracking* flag to be permanently set to True. However, both GOTURN and MIL suffer from the same drawback, so Re3 is not disadvantaged any more than those two are. A few encouraging signs are visible in Re3's results. The first is that there is very little degradation in results with changing resolution, which means we could implement Re3 to improve tracking speed without sacrificing tracking quality and robustness. Extremely low SOTP values mean trajectories were followed accurately and not erratically. Both Recall and Precision are high, and when considered alongside the high Completeness values indicate that very few objects are missed. Perhaps the only area of concern is the SODP values - high relative to the other trackers but not

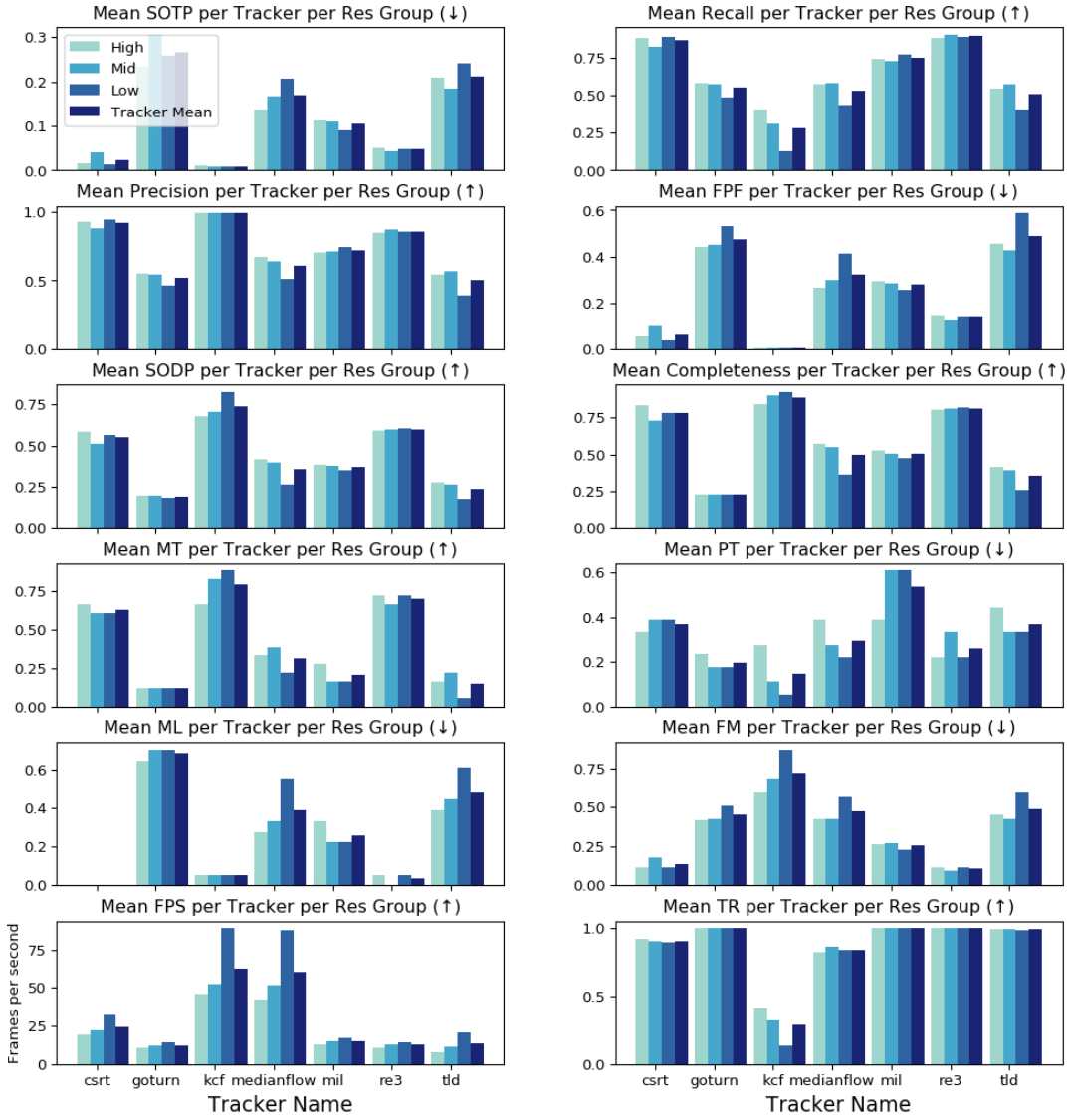



Figure 6.2: Graphical representations of the results after running each tracker on each clip. Each plot represents a separate metric, while each bar represents the mean value for that metric for the resolution group in question. The direction of the arrow (up or down) in parentheses in each plot title indicates whether a higher or lower result is preferred, respectively.

Figure 6.3: Animation showing first 100 frames of output of Re3 tracker running on Snowboard01 at high resolution. Animation is running at 60% of original speed of clip (15/25 fps). The video is also available at the following URL: https://youtu.be/oQ8Bzkn4agM.

exceptional. It is worth noting that SODP relies on accurate data annotation. Even if a tracking trajectory was perfectly smooth and the object tightly contained in bounding boxes throughout, errors in the ground truth bounding boxes would manifest in inferior SODP values. Ground truth errors such as these are indeed possible, given the imperfect nature of the annotation process: we first rely on an object detector, and then a human to fill in any incorrectly classified frames. Indeed, even the highest SODP result (0.828 for KCF) is not perfect. We recall that KCF tracked the fewest number of frames overall and thus would have been exposed to the fewest volume of possible annotation errors. All in all, Re3's results are extremely impressive, which a review of the tracking output videos confirm. The animation in Figure 6.3 reveals the impressive performance of the Re3 tracker running on one of the more challenging clips in the test data (*Snowboard01*). Table 6.2 provides an indication of how challenging *Snowboard01* is - it produced some of the worst tracking results among all of the 18 clips.

Table 6.2 offers another take on the single-run test results: segmentation by clip. Each value in the table is the mean value for that column's metric taken for that row's clip across all resolution groups. We note a few interesting outcomes.

Bike02 gets consistent and impressive results. It is an easy clip for the task: the object is centrally located and stable, not changing too much in appearance and shape throughout. Sup01 shares these attributes, and we notice similar performance. It should be noted here that the detector struggled to locate any human object in Bike02 in almost all the frames during the annotation process, presumably due to the complex appearance of the background. Bike03 produced some worst-in-class scores but with a high TR value, indicating that there must have been a short portion of the clip which all of the trackers struggled with. Indeed, the rider is completely occluded in last 20% of the clip.

Clips Bike04 through Bike08 are similar in that the target is constantly in the frame and visible. The results on these clips are impressive given the messy composition, high speed and changing size of target throughout. Bike09 produces poor results. It is a long clip,

which means there is more room for error, and there is some definite occlusion during the central portion of the clip. However, frames in which occlusion occurs are left out of the ground truth frame data, which means the poor results on clips containing occlusion are likely because most of the trackers are not robust in their ability to self-report failure (most of them *can*, but few of them do it well). Ski01 is an interesting case. The results are not terrible, given the high speed and diminishing size of the target object, and the rapidly tilting camera. Our inference here is that the metrics reflect imperfections in the ground truth annotations, which became increasingly difficult to label as the object became smaller. The tiny boxes drawn during both annotation and tracking mean the SOTP distances would have been calculated over tiny distances, which could vary wildly. SODP is also below average, which confirms this hypothesis somewhat. Ski03's mediocre-good results are easily explained. Most metrics are good with the exception of SODP and Comp, which are intrinsically linked. The problem here is that the rough snow behind the skier gets attributed to the skier, resizing the bounding box incorrectly, and in some cases the tracker becomes stuck on it. In most cases the bounding boxes are huge and scale inversely with the size of the target as it changes; only CSRT and Re3 resize along with it, which leads to the poor SODP values. In Ski04, the skier moves very smoothly, but incredibly fast, moving significant distances between frames and with noticeable blur. This makes most trackers include the shadow of the skier in their hypotheses, increasing the size of the bounding box and reducing SODP.

Slalom01 is just a very tricky clip. The object shape is changing throughout as well as moving very quickly. The background is dynamic and complex. Snowboard01 is similar, but there are no sharp turns (rider follows a smooth arc), which might suggest why the results are slightly better than Slalom01. Surf01 is another difficult clip to track - both object and background change constantly in size, appearance, speed and position. There is also a lot of occlusion as the rider moves in and out of the barrel. Most of the trackers cannot keep up with the rider and erratically resize their hypotheses near and around him - though Re3 does a good job at keeping tight boxes and smooth trajectories. Wake01 suffers from compression artefacts which makes the athlete difficult to distinguish at times. Splashing, reflection and waves on the water are often attributed to the athlete incorrectly. Many trackers simply lose the object to the background when the rider jumps off the kicker obstacle. Interestingly, the range of frame rates is not very wide, with the exception of Ski01 which is a deadly combination of both 4K resolution and 60 fps frame rate. However, the other 60 fps clip (Ski04) had FPS results which were indeed higher than the mean, thus it seems resolution is the primary hindrance to tracking frame rates.

Based on both empirical and analytical review, it would seem Re3 is the most effective tracker that was tested. Despite the low frame rates it achieves (albeit on consumer hardware), the tracking trajectories and bounding boxes are smooth and tight, respectively, and the output videos make for appealing viewing. In most cases it would be pertinent to conduct tracking with the best tracker on the same data 10s or 100s of times and review a distribution of the results for each metric with the intention of ruling out the role of chance in the outcome. However, Re3 is a convolutional neural network-based tracker, and CNNs are finely-tuned approximations of complex non-linear functions. That is, we can expect with certainty that running inference on a CNN will produce the same results each time given the same input.

Table 6.2: Mean tracking results for each evaluation metric, grouped by clip, for all resolutions.

| Clip | SOTP ↓ | Recall ↑ | Prec ↑ | FPF ↓ | SODP ↑ | Comp ↑ | MT ↑ | PT ↓ | ML ↓ | FM ↓ | FPS ↑ | TR ↑ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bike02 | 0.021 | 0.877 | 0.925 | 0.074 | 0.630 | 0.808 | 0.810 | **0.000** | 0.190 | 0.123 | 26.635 | 0.952 |
| Bike03 | 0.157 | 0.554 | 0.565 | 0.429 | 0.420 | 0.603 | 0.571 | 0.048 | 0.381 | 0.445 | 29.199 | 0.874 |
| Bike04 | 0.088 | 0.612 | 0.745 | 0.253 | 0.417 | 0.573 | 0.524 | 0.143 | 0.333 | 0.388 | 26.977 | 0.867 |
| Bike05 | 0.136 | 0.780 | 0.861 | 0.135 | 0.525 | 0.712 | 0.381 | 0.524 | 0.095 | 0.219 | 27.503 | 0.917 |
| Bike07 | 0.105 | 0.604 | 0.727 | 0.269 | 0.470 | 0.628 | 0.571 | 0.143 | 0.286 | 0.397 | 29.390 | 0.860 |
| Bike08 | 0.130 | 0.667 | 0.742 | 0.257 | 0.383 | 0.527 | 0.333 | 0.381 | 0.286 | 0.333 | 26.364 | 0.922 |
| Bike09 | 0.188 | 0.387 | 0.534 | 0.441 | 0.353 | 0.525 | 0.238 | 0.571 | 0.190 | 0.613 | 25.916 | 0.761 |
| Car01 | **0.011** | **1.000** | **1.000** | **0.000** | **0.750** | **0.975** | **0.944** | 0.056 | **0.000** | **0.000** | 33.219 | **1.000** |
| Human01 | 0.112 | 0.569 | 0.591 | 0.404 | 0.407 | 0.519 | 0.429 | 0.143 | 0.429 | 0.430 | 28.931 | 0.864 |
| Ski01 | 0.280 | 0.522 | 0.683 | 0.305 | 0.392 | 0.548 | 0.381 | 0.238 | 0.381 | 0.479 | 14.239 | 0.826 |
| Ski02 | 0.022 | 0.919 | 0.967 | 0.033 | 0.577 | 0.800 | 0.619 | 0.333 | 0.048 | 0.081 | 27.727 | 0.952 |
| Ski03 | 0.117 | 0.684 | 0.734 | 0.256 | 0.336 | 0.431 | 0.286 | 0.381 | 0.333 | 0.316 | 26.329 | 0.940 |
| Ski04 | 0.018 | 0.923 | 0.994 | 0.006 | 0.469 | 0.609 | 0.143 | 0.857 | 0.000 | 0.077 | 29.760 | 0.929 |
| Slalom01 | 0.120 | 0.305 | 0.526 | 0.468 | 0.277 | 0.359 | 0.238 | 0.333 | 0.429 | 0.697 | 31.849 | 0.778 |
| Snowboard01 | 0.235 | 0.459 | 0.644 | 0.309 | 0.417 | 0.547 | 0.333 | 0.381 | 0.286 | 0.541 | 33.464 | 0.770 |
| Sup01 | 0.041 | 0.759 | 0.899 | 0.086 | 0.514 | 0.672 | 0.429 | 0.476 | 0.095 | 0.240 | 35.804 | 0.837 |
| Surf01 | 0.133 | 0.383 | 0.572 | 0.393 | 0.310 | 0.377 | 0.190 | 0.333 | 0.476 | 0.617 | **37.748** | 0.726 |
| Wake01 | 0.190 | 0.319 | 0.523 | 0.394 | 0.263 | 0.353 | 0.190 | 0.238 | 0.571 | 0.680 | 27.377 | 0.713 |
| *Mean* | *0.117* | *0.629* | *0.735* | *0.251* | *0.439* | *0.587* | *0.423* | *0.310* | *0.267* | *0.371* | *28.802* | *0.861* |

# 7 | Conclusion

The purpose of this project was twofold: firstly, to develop a means of efficiently and accurately annotating test video data for object tracking and, secondly, to develop a means of accurately evaluating single object tracking algorithms using test video data. We would need to keep in mind the application during the interpretation of any results: object tracking would need to be conducted in real-time in a lightweight and robust manner. The tracking would also need to be repeatable and reliable.

We took an in-depth look at ANNs, and specifically CNNs which see their primary use in object detection algorithms. Understanding how image data passes through these networks is key to understanding where bottlenecks exist and to interpreting results in a useful way. We learned that the VGGNet architecture is the backbone to many modern CNNs, most of which use at least a portion of VGG in their own network architecture, combining it with all manner of combinations of convolutional and fully-connected layers, and in the case of Re3, with a recurrent network architecture to enable the model to have 'memory'.

When pairing object detectors and object trackers, we know that the detector is the slowest element in the process. Speeding up inference of the object detector would therefore be a logical place to focus efforts in an attempt to improve the overall speed of tracking. This is particularly important knowing that when a tracker fails it needs to be reinitialised with a ground truth object, which would be fed to it from the paired detector. Three common modern algorithms in the world of object detection are SSD, YOLO, and R-CNN. The most recent implementation of R-CNN, known as Mask R-CNN, is capable of instance segmentation: identifying individual objects and distinguishing between objects of the same type using masks. R-CNN is typically the slowest of those mentioned, but consistently the most accurate. We ought to keep in mind that the data on which these models are trained is largely effectual in the detection ability we observe, as well as the nuances of the model architecture itself (e.g., the region proposal network within R-CNN).

We learned that a logical way of grouping tracking algorithms is to refer to the initialisation method they use. All of the trackers tested for this work are detection-free trackers. Presumably, the reason for the lack of popularity of detection-based trackers is the poor speed of detectors, which preclude those trackers from applications where real-time processing is required. Those detectors that are faster, like OpenCV's HAAS face detector, typically are not as accurate. Detection-free tracking appears to be the way forward, ideally with faster and faster detection algorithms.

In developing the evaluation metrics, we built on the work of existing multiple object tracking researchers, adapting the equations to be applicable for single object tracking. In all cases, the SOT metrics are more simple than the MOT metrics, because we don't have to worry about assigning IDs to different objects and carrying that data through the duration of the clip. With the exception of SODP, all the metrics make use of values

derived from SOTP, referring to the average distance between the tracking hypothesis and ground truth object in a particular frame. We validate the hypothesis in each frame using the distance metric rather than an overlap metric, justifying this decision as follows: many of the frames were manually annotated, and as such there is likely considerable error in the sizing and positioning of the bounding boxes. However, we assume the centre of the box is generally somewhere on the target object. The assumption is that optimising for distance (or rather, penalising for greater distances), should make for smoother tracking. Optimising for overlap would make bounding boxes tighter, which wouldn't help to guide the drone as accurately as knowing where the centre of the object is. Thus, we have a metric which measures distance, and based on how good the distance is, we measure how many frames were dropped. Admittedly, the effectiveness of the metrics are somewhat limited by the degree to which a tracker is able to report failure. Some trackers did it stringently, others poorly, others not at all. However, using the same criteria to judge all of the candidate algorithms does level the playing field a little.

Referring back to Mask R-CNN, our intuition is that effective visual model tracking algorithms could be improved by initialising them with masks rather than the contents of bounding boxes. When a bounding box is placed around a non-rectangular object, the visual data supplied to the tracking algorithm includes some portion of the background (typically in each of the four corners). A mask explicitly defines an object, so if we could initialise a tracker with a mask rather than a bounding box, only pixels pertaining to the object would be included. This would theoretically improve tracking when the object is visible but not help to improve tracking during temporary occlusion.

During interpretation of the results, we saw that all the trackers suffered during object occlusion. Knowing that all the trackers we tested use the visual model for tracking (including the CNN-based trackers, by learning features), it is perhaps understandable that trackers struggle when objects disappear. Given the flexibility of CNNs, it would make sense to focus on re-training or using transfer learning to improve robustness to occlusion for something like Re3 - the tracker we believe to be the best based on our results. An intuitive approach might be to amass training data that included a sufficient number of examples of occlusion. We are not entirely sure how annotated occlusion manifests in reality - perhaps just the visible portion of the object is annotated, and if it is completely invisible, perhaps the annotation is a box indicating the position of the object even though we cannot see it, with an accompanying flag variable indicating occlusion. An immediate problem with this approach is that because the visual model learns features, we cannot expect performance to improve after training on examples where bounding boxes contain invisible objects. One solution might be to have different classes of visibility: *occluded, visible, not present*. Annotated, occluded clips would theoretically enable the network to learn how objects move when they disappear temporarily, though I suspect the implementation would be less simple.

We were pleasantly surprised to find that a method as complex as a recurrent convolutional neural network produced the most satisfactory results, especially considering how slow convolutional neural networks have been historically with image inference. Upon closer inspection, however, it makes sense that a visual model that tediously learns features by example (the supervised learning process), would be able to most accurately locate and track a given object. We note here a significant drawback of the current state

of object tracking, particularly as it pertains to the FlyCam use case. We have already established that the object detection component is the bottleneck in the tracking process because of the time taken to perform inference on an image. In our time-sensitive application, the drone would be required to detect and locate the athlete within the frame, which might take as long as one second, by which time the athlete and/or the drone may well have moved. This means that the object information passed to the tracking algorithm is out of date, and the data with which the tracker is initialised is not the most recent and therefore not the most accurate representation of the target object, something which would undeniably affect tracking performance. One possible solution would be to gather a 360 degree representation (a template) of the athlete by having the drone first circle around them. The tracking algorithm would be modified to continuously (on a frame-by-frame basis) compare the tracking hypothesis to the template, creating an additional measure of validity for tracking performance. This would negate the need for the object detector to run first, enabling use in time-sensitive scenarios.

# Appendices

# A | Python Code

The listings in this appendix each contain the Python code developed to carry out specific functions as detailed earlier in the report.

Listing A.1: Python script which converts the bounding box attributes in a Pandas DataFrame from absolute to relative such that they are invariant to the resolution of the video file.

```
import pandas as pd
import argparse
import cv2

# argument parser
ap = argparse.ArgumentParser()
ap.add_argument("-f", "--inputFile", required=True, help="Choose CSV
    file to filter")
args = vars(ap.parse_args())

# find video clip in question
# Bike05, Surf01, etc
title = args['inputFile'].split('/')[-1].split('.')[0]
video_file = title+'.mp4' # Bike05.mp4
frames = pd.read_csv(str('./frame_data/'+title+'.csv'))

# open video stream
vs = cv2.VideoCapture('./raw_clips/' + video_file)

# find resolution of clip
grabbed, frame = vs.read()
orig_width = frame.shape[1]
orig_height = frame.shape[0]

print('[INFO] Resolution is ' + str(orig_width) + ' x ' + str(
    orig_height))

# scale
frames.x1 /= orig_width
frames.w /=  orig_width
frames.y1 /= orig_height
frames.h /=  orig_height

frames.to_csv('./frame_data/' + title + '.csv', index=False)
```

Listing A.2: Python script which carries out the filtering of a Pandas DataFrame to ensure only one detection per frame remains.

```
import pandas as pd
import argparse

# argument parser
ap = argparse.ArgumentParser()
ap.add_argument("-f", "--inputFile", required=True,
```

```
  help="choose which file to filter")
ap.add_argument("-c", "--class", type=int, default=1,
  help="choose the class_id to filter for")
args = vars(ap.parse_args())

input_file = args['inputFile']
csv_name = input_file.split('/')[-1]
class_id_input = args['class']

# read in unfiltered csv as dataframe
unfiltered_df = pd.read_csv(input_file)

# function to filter
def filter_frames(data, class_id):
    # we include reset_index to ensure the indexes from
    # original df are not used
    # drop=True ensures index column is not included in result
    result = data.loc[data['class_id'] == \
      class_id].loc[data.groupby(['frame'] \
      ['conf'].idxmax()].reset_index(drop=True)
    result = result.drop(['class_id', 'conf'], axis=1) \
      .dropna(how='all')
    return result

filtered_df = filter_frames(unfiltered_df, class_id_input)

# save to new csv
filtered_df.to_csv('./frame_data/' + csv_name + '_new', index=False)
```

Listing A.3: Python script which iterates through a Pandas DataFrame of frame detections and displays them to the user in order to observe the quality of detections.

```
import pandas as pd
import numpy as np
import argparse
import cv2
import os.path

# argument parser
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--inputVideo",
  help="choose which file to filter")
args = vars(ap.parse_args())

# frame data
filename = args['inputVideo'].split('/')[-1].split('.')[0]
frames = pd.read_csv(str('./frame_data/'+filename+'.csv'), sep=';')

# find video and open stream
clip = args['inputVideo']
print("INFO - clip name is " +str(clip))
vs = cv2.VideoCapture(clip)

# initialise variables
writer = None
frame_count = 1
new_frames = []
RESIZE_WIDTH, RESIZE_HEIGHT = 1280, 720
```

```
27
28 # iterate over frames in video
29 while True:
30     grabbed, frame = vs.read()
31     # if a frame was not grabbed, the stream has ended
32     if not grabbed:
33         break
34
35     # video native resolution
36     orig_width = frame.shape[1]
37     orig_height = frame.shape[0]
38
39     print("[INFO] Processing frame number " + str(frame_count))
40     print('[INFO] Resolution is '
41       + str(orig_width) + ' x '
42       + str(orig_height))
43
44     # initialize video writer
45     if writer is None:
46         fourcc = cv2.VideoWriter_fourcc(*'mp4v')
47         fps = vs.get(5)
48         print("[INFO] FPS = " + str(fps))
49         writer = cv2.VideoWriter('./ground_truth_clips/' + filename\
50           + '_gt' + '.mp4', fourcc, fps,\
51           (frame.shape[1], frame.shape[0]), True)
52
53     # find which row the current frame is stored in
54     # it might not exist, in which case row = None
55     row = next(iter(frames[frames['frame'] ==\
56       frame_count].index), None)
57
58     if row is not None: # some bb exists
59         print('[INFO] Found BB for frame ' + str(frame_count))
60         # bb info exists for this frame
61         # define corner coordinates
62         startX, startY, endX, endY =\
63           (frames.iloc[row]['x1'])*orig_width,\
64           (frames.iloc[row]['y1'])*orig_height,\
65           (frames.iloc[row]['x1']+frames.iloc[row]['w'])*orig_width,\
66           (frames.iloc[row]['y1']+frames.iloc[row]['h'])*orig_height
67
68         # then draw it
69         cv2.rectangle(frame, (int(startX), int(startY)),\
70           (int(endX), int(endY)), (0,255,0), 2)
71
72     # show the output frame
73     display_frame = cv2.resize(frame, (RESIZE_WIDTH,RESIZE_HEIGHT))
74     cv2.imshow(str(filename), display_frame)
75     key = cv2.waitKey(60) & 0xFF
76
77     if key == ord('q'): # manual quit
78         break
79     if row is None: # ie no bb found, either needs bb, or no objects
80         key = cv2.waitKey(1)
81         # dims of bb drawn by observer
82         drawn_bb = cv2.selectROI(str(filename), display_frame,\
83           fromCenter=False, showCrosshair=True)
84
```

```
85
86
87          temp_dict = {
88                              'frame': frame_count,
89                              'x1': drawn_bb[0]/RESIZE_WIDTH,
90                              'y1': drawn_bb[1]/RESIZE_HEIGHT,
91                              'w': drawn_bb[2]/RESIZE_WIDTH,
92                              'h': drawn_bb[3]/RESIZE_HEIGHT
93                          }
94          # add dict to list of dicts
95          if temp_dict['x1'] != 0 or temp_dict['y1'] != 0:
96              new_frames.append(temp_dict)
97
98      # write frame to file
99      if writer is not None:
100         print('[WRITER] - saving frame ' + str(frame_count))
101         writer.write(frame)
102
103     # increment frame counter
104     frame_count += 1
105
106 # DataFrame of new frames from list
107 new_frames = pd.DataFrame(new_frames,\
108   columns=['frame', 'x1', 'y1', 'w', 'h'])
109
110 # add new frames to dataframe, sort by frame
111 frames = frames.append(new_frames, ignore_index = True,\
112   sort=False).sort_values('frame')
113
114 # save new dataframe as csv w/ same name
115 # if list doesn't exist, save it. if does, leave it as is
116 if ~os.path.isfile('./frame_data/' + filename + '.csv'):
117     frames.to_csv('./frame_data/' + filename + '.csv',\
118       index=False, sep=';')
119
120 # if list doesn't exist, save it. if does, save as different
121 if ~os.path.isfile('./frame_data/human_labelled/' +\
122   filename + '_new_frames.csv'):
123     new_frames.to_csv('./frame_data/human_labelled/' +\
124       filename + '_new_frames.csv', index=False, sep=';')
125
126 elif os.path.isfile('./frame_data/human_labelled/' +\
127   filename + '_new_frames.csv'):
128     new_frames.to_csv('./frame_data/human_labelled/' +\
129       filename + '_new_frames_new.csv', index=False, sep=';')
130
131 # release the file pointers and do some cleanup
132 print("[INFO] cleaning up...")
133 if writer is not None:
134     writer.release()
135 vs.release()
136 cv2.destroyAllWindows()
```

Listing A.4: Python code which produces 12 subplot bar graphs for a visual representation of the tracking results when grouped by resolution.

```
1 # set up figure and subplot layout
2 fig, axs = plt.subplots(6,2, figsize=(12.3, 13), sharex=True)
```

```
fig.subplots_adjust(hspace = .25, wspace=.25)

groups = ('High', 'Mid', 'Low', 'Tracker Mean')

metrics = ('SOTP','Recall','Precision','FPF','SODP',\
  'Completeness','MT','PT','ML','FM','FPS','TR')
# figure parameters initialisation
count = 1
width = 0.22
N = 7
ind = np.arange(N)

for ax, m in zip(axs.ravel(), metrics):

    # means for each res group for each tracker
    high_means = df.groupby(['Tracker', 'Res Group'])\
      [m].mean().unstack().loc[:,('High')]
    mid_means = df.groupby(['Tracker', 'Res Group'])\
      [m].mean().unstack().loc[:,('Mid')]
    low_means = df.groupby(['Tracker', 'Res Group'])\
      [m].mean().unstack().loc[:,('Low')]
    # means for each tracker
    mean_means = df.groupby(['Tracker'])[m].mean()

    colours = ['#9ed5cd', '#44a7cb', '#2e62a1', '#192574']

    p1 = ax.bar(ind, high_means, width, bottom=0, color=colours[0])
    p2 = ax.bar(ind+width, mid_means, width, bottom=0,\
      color=colours[1])
    p3 = ax.bar(ind+2*width, low_means, width, bottom=0,\
      color=colours[2])
    p4 = ax.bar(ind+3*width, mean_means, width, bottom=0,\
      color=colours[3])

    def better(x):
        return {
            'SOTP': ['Lower', '\u2193'],
            'Recall': ['Higher', '\u2191'],
            'Precision': ['Higher', '\u2191'],
            'FPF': ['Lower', '\u2193'],
            'SODP': ['Higher', '\u2191'],
            'Completeness': ['Higher', '\u2191'],
            'MT': ['Higher', '\u2191'],
            'PT': ['Lower', '\u2193'],
            'ML': ['Lower', '\u2193'],
            'FM': ['Lower', '\u2193'],
            'FPS': ['Higher', '\u2191'],
            'TR': ['Higher', '\u2191']
        }[x]

  # setup axes, labels, title, and legend
    ax.set_title('Mean ' + str(m) +\
      ' per Tracker per Res Group (' + better(m)[1] + ')')
    ax.set_xticks(ind + width)
    ax.set_xticklabels(high_means.index)
    ax.set_xlabel('Tracker Name', fontsize=14) if count == 11\
      or count == 12 else None
    ax.set_ylabel('Frames per second') if count == 11 else None
```

```
    ax.legend((p1[0], p2[0], p3[0], p4[0]), groups, loc='upper left')
   if count == 1 else None
    ax.autoscale_view()
    count += 1
# save figure to file
fig.savefig('/trk-results-res-graphs.png', bbox_inches='tight', dpi=96)
```

Listing A.5: Python function which for a given video file runs object detection on each frame and returns a Pandas DataFrame of all of the detections.

```
def annotate_fn(video_clip, resize=0):
  frame_list = []

  # Initialize the video stream and pointer to output video file
  vs = cv2.VideoCapture(video_clip)

  confidence = 0.3
  i = 0

  # repeat until no frames left
  while True:
    # read the next frame from the file
    grabbed, frame = vs.read()

    # increment frame counter
    i += 1

    # if the frame was not grabbed, then we have reached end of clip
    if not grabbed:
      print ("Not grabbed!")
      break

    # retrieve resolution from frame
    orig_width = frame.shape[1]
    orig_height = frame.shape[0]

    # will resize the frame for faster processing in our NN
    # if applicable, and save the resolution
    if resize is not 0:
        frame = imutils.resize(frame, int(resize))

    # start timer for fps metric
    detectorTime = time.time()

    # run detection
    results = model.detect([frame], verbose=0)

    # ensure at least one detection is made
    if len(results) > 0:

    # if none, fill frame info with blanks and go to next frame
    else:
        dict1 = {'clip': 'Bike05.mp4' ,
                 'frame': i,
                 'class_id': np.nan,
                 'conf':np.nan,
                 'x1': np.nan,
                 'y1': np.nan,
```

```
                    'w': np.nan,
                    'h': np.nan}; continue

    # store results
    r = results[0]

    # made detection(s), loop through and record objects
    for k, id in enumerate(list(r['class_ids'])):
        # record all detection info for each frame, append to list
        dict1 = {'clip': video_clip.split('/')[-1],
                    'frame': i,
                    'class_id': r['class_ids'][k],
                    'conf':r['scores'][k],
                    # divide coords by width/height for absolute values
                    'x1': (r['rois'][k][1])/orig_width,
                    'y1': (r['rois'][k][0])/orig_height,
                    'w': (r['rois'][k][3] - r['rois'][k][1])/orig_width,
                    'h': (r['rois'][k][2] - r['rois'][k][0])/orig_height}
        # append object to overall list
        frame_list.append(dict1)

  return pd.DataFrame(frame_list, columns=\
    ['frame', 'class_id', 'conf', 'x1', 'y1', 'w', 'h'])
```

Listing A.6: Python script which runs a specified tracking algorithm on a specified video file and calls a function to calculate evaluation metrics.

```
def run_track(tracker_name, tracker_inst, video_name, video_file,
   frames_df, resize=0):
    # CONSTANTS
    THRESHOLD_IOU, THRESHOLD_DIST = 0.4, 0.1

    # initialise variables
    frame_count = frames_df['frame'].min()
    fn_count, fp_count, fp_thresh_count, tp_count,\
      tn_count, covered_count = 0, 0, 0, 0, 0, 0
    iter_count = 0
    tracking_fps = 0
    iou_list, dist_list = [], []
    tracker = None
    writer = None
    re3 = False

    # initialize the video stream and pointer to output video file
    vs = cv2.VideoCapture(video_file)

    # count of frames in clip for progress bar
    total_frames = int(vs.get(cv2.CAP_PROP_FRAME_COUNT))

    # start clock for FPS
    tracker_time = start_timer()

    for i in tqdm(range(total_frames),desc=tracker_name + ' progress'):
        # flag indicating whether bb has been drawn set to false
        drawn = False

        # read the next frame from the file
        grabbed, frame = vs.read()
```

```
31
32          if not grabbed:
33          # reached end of clip
34              break
35
36          # flag indicating if ground truth object in frame
37          object_exists = (frames_df['frame'] == frame_count).any()
38
39      # resize frame if applicable
40          if resize is not 0:
41              frame = cv2.resize(frame, (resize, int(resize*(9/16))))
42          new_width = frame.shape[1]
43          new_height = frame.shape[0]
44
45          # number of rows in frames is value of N_objects
46          N_objects = len(frames_df.index)
47
48          # bb for ground truth obj (relative values)
49          gt_bb = (frames_df.loc[frames_df['frame'] == frame_count].\
50            squeeze()[1:])*(new_width, new_height,\
51            new_width, new_height) if object_exists else 0
52
53          # initialize the writer
54          if writer is None:
55              print()
56              fourcc = cv2.VideoWriter_fourcc(*'mp4v')
57              fps = vs.get(5)
58              writer = cv2.VideoWriter('./' + video_name + '.mp4',\
59                fourcc, fps, (frame.shape[1], frame.shape[0]), True)
60
61          if tracker is None:
62              # create tracker object
63              tracker = tracker_inst
64
65              # initialise with bounding box
66              # special case for re3
67              if tracker_name == 're3':
68                  # NB: format is (x1, y1, x2, y2) for re3
69                  re3 = True
70                  tracker.track('video', frame[:,:,::-1],\
71                    (gt_bb[0], gt_bb[1],\
72                    gt_bb[0] + gt_bb[2],\
73                    gt_bb[1] + gt_bb[3]))
74
75              # normal case for opencv trackers
76              else:
77                  tracker.init(frame, tuple(gt_bb))
78
79              # first frame is provided to tracker, so always draw it
80              drawn = draw_rect(frame, gt_bb, frame_count, drawn, False)
81
82          if tracker is not None:
83              # update the tracker and grab the tracked object
84              if re3 == True:
85                  tracking, trk_bb =\
86                    tracker.track('video', frame[:,:,::-1])
87                  # convert to (x1, y1, x2, y2)
88                  trk_bb = (trk_bb[0], trk_bb[1],\
```

```
                    trk_bb[2] - trk_bb[0],\
                    trk_bb[3] - trk_bb[1])
            else:
                tracking, trk_bb = tracker.update(frame)
            # tracker has self-reported failure
            if tracking is False:
                if ~object_exists:
                    # true negative
                    tn_count += 1

                elif object_exists:
                    # false negative
                    fn_count += 1

            # tracker has self-reported success
            elif tracking:
                if object_exists:
                # calculate iou and dist
                # NB: (x1, y1, w, h) format
                    iou_t, dist_rel_t = compute_iou_dist(gt_bb,\
                      trk_bb, new_width, new_height)

                    # completeness metrics
                    if iou_t >= THRESHOLD_IOU:
                        covered_count += 1

                    if (dist_rel_t < THRESHOLD_DIST):
                        # true positive
                        tp_count += 1

                    elif (dist_rel_t >= THRESHOLD_DIST):
                        # false positive
                        fp_thresh_count += 1

                    # record iou and dist values
                    iou_list.append(iou_t)
                    dist_list.append(dist_rel_t)

                elif ~object_exists:
                    # false positive
                    fp_count += 1

                # tracking taken place, so draw frame
                drawn = draw_rect(frame, trk_bb, frame_count,\
                  drawn, re3)

        # increment counters
        frame_count += 1
        iter_count += 1

        # write frame to disk
        if writer is not None:
            writer.write(frame)

    # record FPS
    tracking_fps = stop_and_report(frame_count, tracker_time)

    if writer is not None:
```

```
        writer.release()

    # call metrics function - returns dictionary of metrics
    metrics = compute_metrics(fp_thresh_count, fp_count, tp_count,\
      tn_count, fn_count, covered_count, dist_list, iou_list,\
      N_objects, frame_count, tracking_fps, clip_name,\
      str(new_width), tracker_name, THRESHOLD_IOU, THRESHOLD_DIST)

    return metrics
```

# References


Bewley, A., Ge, Z., Ott, L., Ramos, F., and Upcroft, B. (2016), "Simple online and real-time tracking", in *2016 IEEE International Conference on Image Processing (ICIP)*, IEEE, 3464–8.

Bluche, T. (2017), "Convolution operation", http://technodocbox.com/3D_Graphics/70716176-Deep-neural-networks-applications-in-handwriting-recognition.html.

Bolme, D., Beveridge, J., Draper, B., and Lui, M. (2010), "Visual object tracking using adaptive correlation filters", in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE, 2544–50.

Börjesson, A. and Hillborg, N. (2018), "Autonomous drone for liferaft detection", MA thesis (Chalmers University of Technology).

Challa, S., Morelande, M. R., Mušicki, D., and Evans, R. J. (2011), *Fundamentals of object tracking* (Cambridge University Press). doi: 10.1017/CBO9780511975837.

Chen, L., Ai, H., Zhuang, Z., and Shang, C. (2018), "Real-time multiple people tracking with deeply learned candidate selection and Person Re-Identification", *CoRR* abs/1809.04427, arXiv: 1809.04427, http://arxiv.org/abs/1809.04427.

Chen, P., Dang, Y., Liang, R., Zhu, W., and He, X. (2017), "Real-time object tracking on a drone with multi-inertial sensing data", *IEEE Transactions on Intelligent Transportation Systems*, 19/1: 131–9.

Gordon, D., Farhadi, A., and Fox, D. (2018), "Re3: Real-time recurrent regression networks for visual tracking of generic objects", *IEEE Robotics and Automation Letters*, 3/2: 788–95.

Hastie, T., Tibshirani, R., and Friedman, J. (2001), *The Elements of Statistical Learning* (Springer Series in Statistics; New York, NY, USA: Springer New York Inc.).

He, K., Gkioxari, G., Dollár, P., and Girshick, R. (2017), "Mask R-CNN", in *Proceedings of the IEEE international conference on computer vision*, 2961–9.

Held, D., Thrun, S., and Savarese, S. (2016), "Learning to track at 100 FPS with deep regression networks", in *European Conference Computer Vision (ECCV)*.

Kalal, Z., Mikolajczyk, K., and Matas, J. (2011), "Tracking-learning-detection", *IEEE transactions on pattern analysis and machine intelligence*, 34/7: 1409–22.

Karpathy, A. et al. (2016), "CS231n Convolutional neural networks for visual recognition", *Neural Networks*, [Online; accessed 20-June-2019], https://cs231n.github.io/convolutional-networks/.

Kasturi, R., Goldgof, D., Soundararajan, P., Manohar, V., Garofolo, J., Bowers, R., Boonstra, M., Korzhova, V., and Zhang, J. (2008), "Framework for performance evaluation of face, text, and vehicle detection and tracking in video: Data, metrics, and protocol", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31/2: 319–36.

Konushin, A. and Artemov, A. (2018), *Deep learning in computer vision*, https://www.coursera.org/learn/deep-learning-in-computer-vision/home/welcome.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C., and Berg, A. (2016), "SSD: Single shot multibox detector", in *European conference on computer vision*, Springer, 21–37.

Luo, W., Zhao, X., and Kim, T. (2014), "Multiple object tracking: A review", *CoRR* abs/1409.7618, arXiv: 1409.7618, http://arxiv.org/abs/1409.7618.

Mallick, S. (2017), "Object tracking using OpenCV (C++/Python)", Learn OpenCV, https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/.

Parthasarathy, D. (2017), "A brief history of CNNs in image segmentation: From R-CNN to Mask R-CNN", Medium, https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn-to-mask-r-cnn-34ea83205de4.

Punyawiwat, P. and Wattanapenpaiboon, N. (2018), "Interns explain CNN", https://blog.datawow.io/interns-explain-cnn-8a669d053f8b.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016), "You only look once: Unified, real-time object detection", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 779–88.

Rogers, C. (2016), "GoPro: Surfing The World's Best Wave - Skeleton Bay 2016", YouTube, https://www.youtube.com/watch?v=0uRhy3eUHtM.

Rogers, C. (2017), "Drone Around The World | Chris Rogers Drone Showreel 4k", YouTube, https://www.youtube.com/watch?v=U_jOf0_eE0k&t=3s.

Rosebrock, A. (2019), *OpenCV object tracking*, https://www.pyimagesearch.com/2018/07/30/opencv-object-tracking/.

Saha, A. (2017), "You only look once, v1", GitHub, https://github.com/anandsaha/paper.summaries/blob/master/summaries/Yolo.md.

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

Stiefelhagen, R., Bernardin, K., Bowers, R., Garofolo, J., Mostefa, D., and Soundararajan, P. (2006), "The CLEAR 2006 evaluation", in *International evaluation workshop on classification of events, activities and relationships*, Springer, 1–44.

Varfolomieiev, A. and Lysenko, O. (2016), "An improved algorithm of median flow for visual object tracking and its implementation on ARM platform", *Journal of Real-Time Image Processing*, 11/3: 527–34.

Yan, X., Wu, X., Kakadiaris, I. A., and Shah, S. K. (2012), "To track or to detect? An ensemble framework for optimal selection", in *European Conference on Computer Vision*, Springer, 594–607.

Yang, B., Huang, C., and Nevatia, R. (2011), "Learning affinities and dependencies for multi-target tracking using a CRF model", in *CVPR 2011*, IEEE, 1233–40.