Struck: Structured Output Tracking with Kernels

Sam Hare*, Stuart Golodetz*, Amir Saffari*, Vibhav Vineet,
Ming-Ming Cheng, Stephen L. Hicks and Philip H. S. Torr

Abstract—Adaptive tracking-by-detection methods are widely used in computer vision for tracking arbitrary objects. Current approaches treat the tracking problem as a classification task and use online learning techniques to update the object model. However, for these updates to happen one needs to convert the estimated object position into a set of labelled training examples, and it is not clear how best to perform this intermediate step. Furthermore, the objective for the classifier (label prediction) is not explicitly coupled to the objective for the tracker (estimation of object position). In this paper, we present a framework for adaptive visual object tracking based on structured output prediction. By explicitly allowing the output space to express the needs of the tracker, we avoid the need for an intermediate classification step. Our method uses a kernelised structured output support vector machine (SVM), which is learned online to provide adaptive tracking. To allow our tracker to run at high frame rates, we (a) introduce a budgeting mechanism that prevents the unbounded growth in the number of support vectors that would otherwise occur during tracking, and (b) show how to implement tracking on the GPU. Experimentally, we show that our algorithm is able to outperform state-of-the-art trackers on various benchmark videos. Additionally, we show that we can easily incorporate additional features and kernels into our framework, which results in increased tracking performance.

Index Terms—tracking-by-detection, structured output SVMs, budget maintenance, GPU-based tracking

1 Introduction

Visual object tracking is one of the core problems of computer vision, with a wide range of applications that include human-computer interaction, surveillance and augmented reality. It also forms an essential part of higher-level vision tasks such as scene understanding and action recognition.

In some scenarios, the object to be tracked is known in advance, and it is possible to incorporate prior knowledge when designing the tracker; in others, however, the target is arbitrary and only specified at runtime, in which case the tracker must be able to model the object's appearance on the fly and adapt the model during tracking to account for object motion, lighting conditions and occlusion. Even when the object is known, having a flexible tracker that can adapt its model at runtime is attractive and, in real-world scenarios, often essential for successful tracking.

The tracking-by-detection approach [1], which treats the tracking problem as one of repeated detection over time, has become particularly popular recently, partly due to significant recent progress in object detection (with many of the ideas being directly transferrable to tracking) and partly due to the development of methods that allow the classifiers used by these approaches to be trained online, providing a natural mechanism for adaptive tracking [2]–[4].

- *(*) S. Hare, S. Golodetz and A. Saffari assert joint first authorship.*
- *S. Hare was with Oxford Brookes University, and is now a co-founder of Obvious Engineering.*
- *S. Golodetz, S.L. Hicks and P.H.S. Torr are with Oxford University.*
- *A. Saffari was with Sony Europe Ltd., and is now with Affectv.*
- *V. Vineet was with Oxford University and is now with Stanford.*
- *M.-M. Cheng was with Oxford University and is now with Nankai.*

Traditional approaches to *adaptive* tracking-by-detection have tended to train a binary classifier online to distinguish the target object from the background. This classifier is used to estimate the object's location in each new input frame by searching for the maximum classification score in a local region around the previous frame's estimate, typically using a sliding-window approach. After estimating the new object location, traditional approaches have tended to generate a set of binary-labelled training samples with which to update the classifier. Such approaches thus separate the learning of the tracker into two distinct parts: (i) the generation and labelling of samples, and (ii) the classifier update.

Whilst common, this separation is problematic. Firstly, it is unclear how to generate and label the samples in a principled manner (the usual approaches rely on predefined rules such as labelling a sample positive/negative based on its distance from the estimated object location). Secondly, the learning process does not explicitly couple the classifier's goal (predicting binary sample labels) to the tracker's goal (estimating object location), and so the maximum classifier confidence may not correspond to the best location estimate (Williams *et al.* [5] raised a similar point). State-of-the-art adaptive tracking-by-detection methods have mainly focused on improving tracking performance by increasing the robustness of the classifier to poorly-labelled samples resulting from this approach, e.g. using robust loss functions [6], [7], semi-supervised learning [8], [9], or multiple-instance learning [3], [10].

In this paper, we take a different approach and frame the overall tracking problem as one of *structured* output prediction, in which the task is to directly predict the change in object configuration between frames. We present a novel

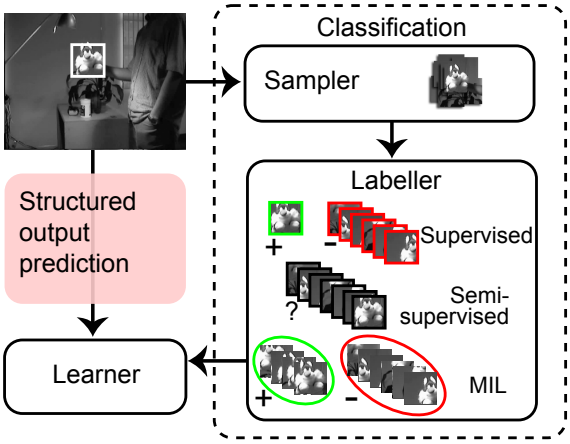


Fig. 1: Different adaptive tracking-by-detection paradigms: given the current estimated object location, traditional approaches (shown on the right-hand side) generate a set of samples and, depending on the type of learner, produce training labels. Our approach (left-hand side) avoids these steps and operates directly on the tracking output.

and principled adaptive tracking-by-detection framework called *Struck* that integrates learning and tracking, avoiding the need for ad-hoc update strategies (see Figure 1).

Many recent tracking-by-detection approaches, e.g. [2]–[4] have been inspired by the successes of boosting-based approaches in object detection. Indeed, elements of such methods, such as the Haar features used in the very successful Viola-Jones face detection approach [11], have become almost standard in tracking-by-detection. However, a significant amount of successful research in object detection itself has tended to use SVMs rather than boosting, since these generalise well, are robust to label noise and can represent objects flexibly using kernels [12]–[14]. The approach we present here thus makes use of SVMs rather than boosting. Since we are interested in *structured* output prediction, we make use of the structured output SVM framework of Tsochantaridis *et al.* [15]. In particular, we extend LaRank, the online structured output SVM learning method proposed by Bordes *et al.* [16], [17], and adapt it to the task of adaptive object tracking.

Whilst structured output SVMs have been applied in computer vision before, most notably for object detection by Blaschko and Lampert [12], our work differs from such approaches in having no offline labelled data available for training (except the object's location in the first frame) and relying instead on online learning. This makes a significant difference, because online learning with kernels suffers from the so-called *curse of kernelisation*, whereby the number of support vectors in the SVM grows unboundedly as more and more training data is supplied. Since evaluation of a kernelised SVM is linear in the number of support vectors in its representation, this growth inevitably inhibits real-time evaluation of the SVM if steps are not taken to control the number of support vectors maintained. We therefore introduce a novel way of limiting the support vectors we maintain that builds on recent proposals for the online learning of classification SVMs on a fixed budget [18], [19]. This approach allows us to achieve large gains in computational efficiency without significantly degrading the performance of our tracker.

An earlier version of this paper appeared in [20]. We extend it here in the following ways:

1) We extend Struck to deal with scale (§4.7.1).
2) We show how Struck can be implemented on the GPU so as to achieve high frame-rates (§5 and supplementary material).
3) We evaluate Struck extensively on the recent benchmark of Wu *et al.* [21] (§6), performing experiments that quantify how tracking performance is affected by (i) multi-kernel learning, (ii) structured learning, (iii) parameter changes, and (iv) scale. *In particular, we show that one of our multi-kernel variants of Struck can in many cases achieve comparable or even superior tracking performance to the state-of-the-art KCF tracker of Henriques* et al. [22].
4) We include detailed derivations of our SVM formulation to make it easier for others to build on our approach (supplementary material).

This paper is organised as follows: in §2, we briefly review related work; in §3, we provide an overview of traditional approaches to tracking-by-detection; in §4, we describe the Struck tracker; in §5, we describe our GPU implementation of Struck; in §6, we evaluate numerous variants of our CPU and GPU implementations of Struck on the Wu *et al.* [21] benchmark and compare them to state-of-the-art trackers, and in §7 we conclude.

Code for our CPU implementation of Struck can be found at http://www.samhare.net/research. Our GPU implementation can be found at https://bitbucket.org/sgolodetz/thunderstruck.

2 Related Work

Due to the importance of the tracking problem, a wide variety of different approaches have been proposed to solve it over the years. Whilst a comprehensive review of tracking techniques is beyond the scope of this paper, we direct the reader to [23] for a survey, and also to [21], [24], [25] for some benchmarks that compare a significant number of trackers on large datasets. We focus here on a representative selection of recent trackers.

Dictionary-based trackers maintain dictionaries of object templates and aim to represent candidate object regions in a new frame using combinations of these templates. A popular idea is to try and represent the candidates sparsely using ℓ_1-norm minimization [26]–[28]. Prediction from one frame to the next is often done using particle filtering [29], with a sensor model that assigns higher confidence to candidates that are more easily represented by the templates. For example, Xing *et al.* [28] describe an approach that combines short-term, medium-term and long-term dictionaries to achieve a compromise between adaptivity (the short-term dictionary will adapt more quickly to new data) and robustness (the long-term dictionary will

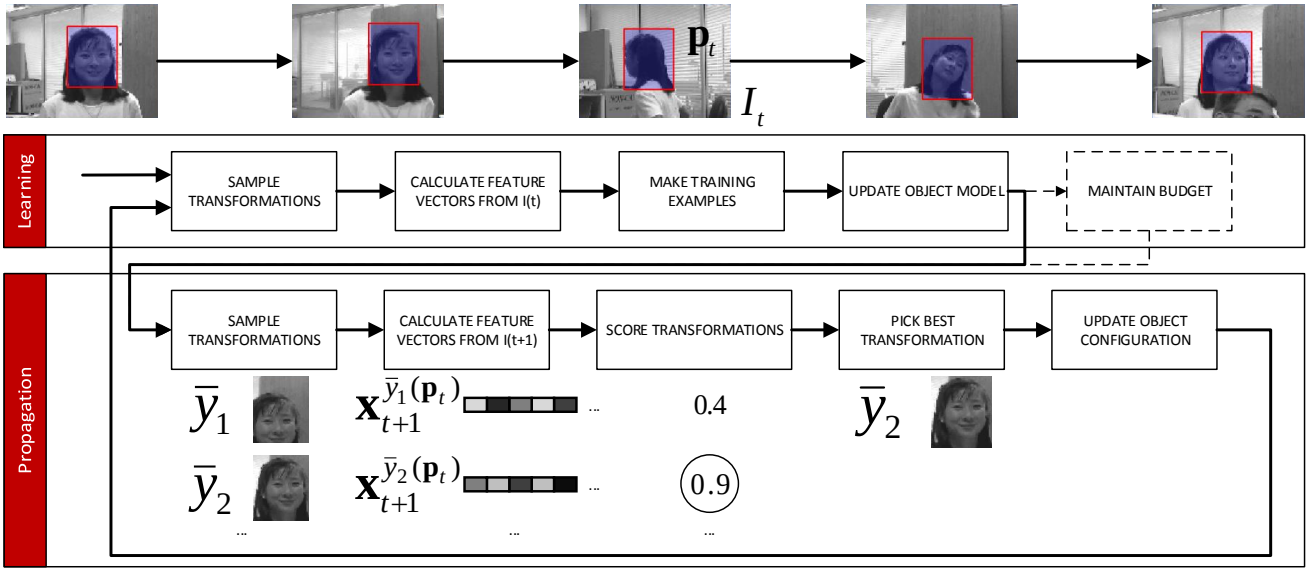


Fig. 2: The typical structure of a tracking-by-detection tracker (see §3). The dashed box is specific to Struck and indicates the place in which we add a budget maintenance step to the pipeline (see §4).

remember more of what the object originally looked like). Wang *et al.* [27] describe another interesting dictionary-based approach that aims to learn templates that capture distinct aspects of the object.

Ensemble-based trackers combine the results of a set of individual 'weak' classifiers to form a strong classifier that can be used to predict an object's bounding box in a new frame. For example, Cao and Xue [30] describe an adaptive random forest method that maintains a collection of candidate decision trees and picks half of them each frame to form an ensemble. (Other techniques that incorporate random forests can be found in [31], [32].) Bai *et al.* [33] combine weak classifiers trained on 8×8 patches within the object's bounding box using weight vectors sampled from a Dirichlet distribution that is updated over time. Wang *et al.* [34] show how to combine an ensemble of different trackers (including Struck) using a conditional particle filter approach to try and meld the best features of the trackers.

Segmentation-based trackers [35]–[37] actually segment (possibly coarsely) the object being tracked in each frame, so as to try and avoid the problem of drift that occurs when trackers inadvertently incorporate parts of the background in their object representation. For example, Duffner and Garcia [35] describe PixelTrack, an approach that co-trains a probabilistic segmentation model alongside a pixel-based Hough model so as to better handle non-rigid deformations of the tracked object between frames.

Circulant trackers are an interesting recent type of tracker that exploit the circulant structure of adjacent sub-windows in an image to achieve extremely fast tracking. The original such tracker, CSK [38], works by evaluating a classifier trained using kernel regularised least squares (KRLS) quickly at all sub-windows around the estimated target location and maximising the response. Danelljan *et al.* [39] build on this by introducing colour attributes to achieve superior performance on colour sequences. The current state-of-the-art approach is KCF [22], a kernelised correlation filter tracker that achieves its best results using a Gaussian kernel and histogram-of-oriented-gradients (HOG) features, and runs at 172 FPS.

Since [20], various approaches have used *structured learning* in a tracking context. For example, Yao *et al.* [40] describe a part-based tracker based on online latent structured SVMs, which they solve in the primal by extending the online Pegasos solver to perform structured prediction. Shen *et al.* [41] describe a generic boosting framework for combining weak structured learners, and show how it can be used to develop a visual tracker that achieves competitive results. Separately, structured SVMs have proved useful for solving data association problems in the context of multi-target tracking [42].

A number of trackers we survey do not fall into any of the above categories. For example, Pernici and Del Bimbo [43] describe a tracker called ALIEN based on Nearest Neighbour classifiers that tracks using an oriented rather than axis-aligned bounding box, handles occlusions well and is designed for long-term tracking. Lu *et al.* [44] describe an interesting approach based on And-Or graphs that achieves good tracking performance at the cost of some speed. Finally, Zhang and van der Maaten [45] describe an appealing multi-object tracker based on structured SVMs that can be co-opted for single-object, part-based tracking.

3 Tracking-by-Detection

In this section, we provide an overview of traditional adaptive tracking-by-detection trackers, which attempt to learn a binary classifier to distinguish the target object from the background.

The typical structure of such a tracker is shown in Figure 2. Given a sequence of images $(I_1, I_2, \ldots)$, in which I_t is the image at time $t \in \mathcal{T}$, the tracker produces a sequence $(\mathbf{p}_1, \mathbf{p}_2, \ldots)$ of parameter vectors such that each $\mathbf{p}_t \in \mathcal{P}$ represents the estimated configuration of the target object in the corresponding image I_t. In the simplest case,

a parameter vector might contain only the position of a 2D bounding box of fixed size, but additional components can be added, e.g. to account for scale, rotation or shape. The initial parameter vector $\mathbf{p}_1$ is usually given, in which case the tracker's goal is to estimate $\mathbf{p}_{t+1}$ from $\mathbf{p}_t$. To help with this, it maintains a classifier with a scoring function $g : \mathcal{X} \to \mathbb{R}$ that can be used to score feature vectors in a feature space $\mathcal{X}$ based on how well they correspond to the target object. The classifier is trained with binary-labelled examples of the form $(\mathbf{x}_t^{\mathbf{P}}, \ell)$, in which $\mathbf{x}_t^{\mathbf{P}}$ denotes the vector of features extracted from the patch of image I_t denoted by $\mathbf{p} \in \mathcal{P}$, and label $\ell \in \{+1, -1\}$ specifies whether the example is a positive or negative one. The predicted label $f(\mathbf{x})$ for a feature vector $\mathbf{x}$ can be computed as $\text{sign}(g(\mathbf{x}))$.

3.1 Learning/Adaptation

Given an estimated object configuration $\mathbf{p}_t$ in image I_t, a traditional tracker will use a three-stage process to generate a set of training examples from I_t and update the classifier. First, a *sampler* will generate a set of n different transformations $\{\bar{y}_i\}$ in a *search space* $\mathcal{Y}$ of $\mathcal{P} \to \mathcal{P}$ transformations around $\mathbf{p}_t$, and calculate the corresponding feature vectors $\{\mathbf{x}_t^{\bar{y}_i(\mathbf{p}_t)}\}$. Then, a *labeller* will choose labels $\{\ell_i\}$ for the samples to produce a set of labelled training examples $\{(\mathbf{x}_t^{\bar{y}_i(\mathbf{p}_t)}, \ell_i)\}$. Finally, these examples will be used to update the classifier.

3.2 Propagation

In order to propagate the estimated object configuration from one image to the next, it is assumed that the estimate $\mathbf{p}_{t+1}$ for the object's configuration in image I_{t+1} can be determined by maximising g in a local region around $\mathbf{p}_t$. The tracker will generally iterate over the search space of transformations[1] $\mathcal{Y}$, and pick the transformation $y_t \in \mathcal{Y}$ that maximises the response of the classifier:

$$y_t = \operatorname*{argmax}_{\bar{y} \in \mathcal{Y}} g\left(\mathbf{x}_{t+1}^{\bar{y}(\mathbf{p}_t)}\right) \tag{1}$$

It will then set $\mathbf{p}_{t+1} = y_t(\mathbf{p}_t)$.

The form of $\mathcal{Y}$ depends on the type of motion being tracked. For most existing tracking-by-detection approaches, this is 2D translation, in which case one option is for it to be of the form $\{\mathbf{p} \mapsto \mathbf{p} + \Delta\mathbf{p} : \|\Delta\mathbf{p}\| < r\}$, where r is a search radius. However, we will see in §4 that this approach can be straightforwardly extended to deal with other types of motion, e.g. scale can be incorporated by using parameter vectors of the form (x, y, s), where $s \in \mathbb{R}^+$ denotes a factor that can be used to scale the bounding box around the object, and choosing $\mathcal{Y}$ appropriately.

3.3 Issues

This approach to tracking raises a number of issues. Firstly, the assumption made in (1) that maximising g provides an accurate estimate of object configuration is not explicitly incorporated into the learning algorithm, since the classifier is trained only with binary examples and has no information about transformations. Secondly, all the training examples are equally weighted, meaning that a negative example that overlaps significantly with the current object configuration is treated the same as one that overlaps very little. One implication of this is that slight inaccuracy during tracking can lead to poor labelling of examples, which is likely to reduce the accuracy of the classifier and lead to further tracking inaccuracy. Thirdly, the labeller is usually designed using heuristics, rather than being tightly-coupled to the classifier. Mistakes made by the labeller manifest themselves as *label noise*, and many state-of-the-art approaches try to mitigate this problem by using robust loss functions [6], [7], semi-supervised learning [8], [9], or multiple-instance learning [3], [10]. We argue that all of these techniques, though justified in increasing the robustness of the classifier to label noise, are not addressing the real problem, which stems from separating the labeller from the learner. Our algorithm does not depend on a labeller and tries to overcome all of these problems within a coherent framework by directly linking learning to tracking and avoiding an artificial binarisation step. Sample selection is fully controlled by the learner itself, and relationships between samples such as their relative similarity are taken into account during learning.

To conclude this section, we describe how a conventional labeller works, as this provides further insight into our algorithm. Traditional labellers use a *transformation similarity function* to determine the label of a sample. Such a function has the form $s_{\mathbf{p}} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ and can be used to assign a similarity score to the samples denoted by a pair of transformations expressed relative to a reference configuration $\mathbf{p}$. For example, the function defined by

$$s_{\mathbf{p}}^{o}(y_i, y_j) = \frac{y_i(\mathbf{p}) \cap y_j(\mathbf{p})}{y_i(\mathbf{p}) \cup y_j(\mathbf{p})} \tag{2}$$

measures the degree of overlap between the two bounding boxes $y_i(\mathbf{p})$ and $y_j(\mathbf{p})$. For 2D translation, an alternative function could be defined based on the difference between the translations.

Let y_0 denote the identity (or null) transformation, *i.e.* $y_0(\mathbf{p}) = \mathbf{p}$. Given a transformation similarity function $s_{\mathbf{p}}$, the labeller determines the label ℓ of the sample generated by transformation y by applying a *labelling function* $\ell = L(s_{\mathbf{p}}(y_0, y))$. Most commonly, this can be expressed as

$$L(s_{\mathbf{p}}(y_0, y)) = \begin{cases} +1 & \text{if } s_{\mathbf{p}}(y_0, y) \geq \theta_u \\ -1 & \text{if } s_{\mathbf{p}}(y_0, y) < \theta_l \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

in which θ_u and θ_l are upper and lower thresholds, respectively. Binary classifiers generally ignore the unlabelled examples (those with a label of 0) [2], whilst classifiers based on semi-supervised learning use them in their update phase [8], [9]. In approaches based on multiple-instance learning [3], [10], the labeller groups all of the positive examples into a *bag* and assigns a positive label to the bag

1. In practice, it is often profitable to use different search spaces for learning and propagation, but we do not distinguish between the two search spaces here to simplify the presentation.

instead. Most, if not all, variants of adaptive tracking-by-detection algorithms use a labeller which can be expressed in this fashion. However, it is not clear how the labelling parameters (*e.g.* the thresholds θ_u and θ_l) should be estimated in an online learning framework. Additionally, such heuristic approaches are often prone to noise and it is not clear why such a function is in fact suitable for tracking. In the next section, we derived our structured output algorithm that fundamentally addresses these issues and can be thought of as generalising these heuristic methods.

4 The Struck Tracker

4.1 Overview

Our Struck tracker broadly follows the general structure of a tracking-by-detection approach outlined in the previous section, but with a couple of key differences.

The first difference is that instead of learning a binary classifier, we learn a prediction function $f : \mathcal{T} \to \mathcal{Y}$ that directly estimates the object transformation between consecutive images. That is, $f(t)$ is the required transformation from $\mathbf{p}_t$ to $\mathbf{p}_{t+1}$. We learn f in a structured output SVM framework [12], [15], and make use of a scoring function $g : \mathcal{T} \times \mathcal{Y} \to \mathbb{R}$ that can be used for prediction via

$$f(t) = \operatorname*{argmax}_{y \in \mathcal{Y}} g(t, y). \tag{4}$$

The output space of f is now the search space $\mathcal{Y}$, rather than $\{+1, -1\}$. Note the similarity between (4) and (1): we are still performing an argmax in order to predict the object transformation, but the scoring function g now has direct access to the transformation y, allowing it to be incorporated into the learning algorithm.

The second difference is that we introduce a budgeting mechanism at the end of the adaptation stage to limit the number of support vectors we maintain. As we will see shortly, our approach uses a *kernelised* SVM, which must explicitly maintain a set of support vectors (as opposed to a linear SVM, for which it is sufficient to maintain only a weight vector). Since kernelised SVM evaluation is linear in the size of the support vector set, it is crucial that we limit this size in order to achieve efficient evaluation and make our tracker run in real time.

4.2 Primal SVM Formulation

To formulate our SVM, we first restrict g to be a linear function $g(t, y) = \langle \mathbf{w}, \mathbf{\Phi}(t, y) \rangle$, in which $\mathbf{w}$ is the SVM's weight vector and $\mathbf{\Phi}$ is a function (known as the *joint kernel map*) that is used to generalise the SVM to non-linear kernels in a way that will be described later. When using a linear kernel, $\mathbf{\Phi}(t, y) = \mathbf{x}_{t+1}^{y(\mathbf{p}_t)}$, and $g(t, y) = \langle \mathbf{w}, \mathbf{x}_{t+1}^{y(\mathbf{p}_t)} \rangle$.

We can learn g in a large-margin framework from a set of examples $\{(t_1, y_1), ..., (t_n, y_n)\}$ by solving the quadratic program

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & \forall i : \ \xi_i \geq 0 \\ & \forall i, \forall y \neq y_i : \ \langle \mathbf{w}, \delta\mathbf{\Phi}_i(y) \rangle \geq \Delta(y_i, y) - \xi_i, \end{aligned} \tag{5}$$

where $\delta\mathbf{\Phi}_i(y) = \mathbf{\Phi}(t_i, y_i) - \mathbf{\Phi}(t_i, y)$ and we set $C = 100$. Intuitively, an example (t_i, y_i) specifies that the 'correct' transformation of the object from $\mathbf{p}_{t_i}$ to $\mathbf{p}_{t_i+1}$ is y_i.

The optimisation aims to ensure that the value of $g(t_i, y_i)$ for the i^{th} training example is greater than $g(t_i, y)$ for $y \neq y_i$, by a margin that depends on a (symmetric) loss function Δ. This loss funcion should satisfy $\Delta(y, \bar{y}) = 0$ iff $y = \bar{y}$, and increase as y and $\bar{y}$ become more dissimilar. The loss function plays an important role in our approach, as it allows us to address the issue raised previously of all samples being treated equally. This can be achieved by making use of the transformation similarity function introduced in §3.3. For example, as suggested by Blaschko and Lampert [12], we can base our loss function on bounding box overlap according to

$$\Delta(y, \bar{y}) = 1 - s^o_{\mathbf{p}}(y, \bar{y}), \tag{6}$$

in which $s^o_{\mathbf{p}}$ is the overlap function (2).

The attentive reader will observe that the formulation above involves a large (indeed, potentially infinite) number of constraints, and it is by no means clear at this stage how the SVM can be solved in a computationally-feasible way. In practice, as we will see in §4.5, we address this issue by making use of the LaRank solver of Bordes *et al.* [16], [17], which optimises the SVM using a sequence of minimal steps rather than trying to solve it monolithically. It also bears mentioning that at an implementation level, we do not actually consider an infinite set of constraints: in practice, our search space $\mathcal{Y}$ of possible transformations is finite, and we only retain constraints relating to images in the tracking sequence that are providing one or more current support vectors to the SVM.

4.3 Dual SVM Formulation

Using standard Lagrangian duality techniques [46], (5) can be converted into its equivalent dual form

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i, y \neq y_i} \Delta(y, y_i)\alpha_i^y - \frac{1}{2} \sum_{\substack{i, y \neq y_i \\ j, \bar{y} \neq y_j}} \alpha_i^y \alpha_j^{\bar{y}} \langle \delta\mathbf{\Phi}_i(y), \delta\mathbf{\Phi}_j(\bar{y}) \rangle \\ \text{s.t.} \quad & \forall i, \forall y \neq y_i : \ \alpha_i^y \geq 0 \\ & \forall i : \ \sum_{y \neq y_i} \alpha_i^y \leq C \end{aligned} \tag{7}$$

and the scoring function expressed as

$$g(t, y) = \sum_{i, \bar{y} \neq y_i} \alpha_i^{\bar{y}} \langle \delta\mathbf{\Phi}_i(\bar{y}), \mathbf{\Phi}(t, y) \rangle. \tag{8}$$

A derivation for this can be found in the supplementary material. As in the case of classification SVMs, a benefit of this dual representation is that because the joint kernel map $\mathbf{\Phi}$ only ever occurs inside scalar products, it can be defined implicitly in terms of an appropriate joint kernel function $k(t, y, \bar{t}, \bar{y}) = \langle \mathbf{\Phi}(t, y), \mathbf{\Phi}(\bar{t}, \bar{y}) \rangle$. (This is the so-called *kernel trick*, which allows us to define non-linear kernels k for which we do not have an explicit representation for $\mathbf{\Phi}$.) The kernel functions we use during tracking are discussed in Section 4.7.2.

4.4 Reparameterising the Dual SVM

By reparameterising (7) [16] according to

$$\beta_i^y = \begin{cases} \sum_{\bar{y} \neq y_i} \alpha_i^{\bar{y}} & \text{if } y = y_i \\ -\alpha_i^y & \text{otherwise,} \end{cases} \tag{9}$$

the dual can be considerably simplified to

$$\begin{aligned} \max_{\boldsymbol{\beta}} \quad & -\sum_{i,y} \Delta(y, y_i)\beta_i^y - \frac{1}{2}\sum_{i,y,j,\bar{y}} \beta_i^y \beta_j^{\bar{y}}\, k(t_i, y, t_j, \bar{y}) \\ \text{s.t.} \quad & \forall i, \forall y:\ \beta_i^y \leq \delta(y, y_i)C \\ & \forall i:\ \sum_y \beta_i^y = 0, \end{aligned} \tag{10}$$

where $\delta(y, \bar{y}) = 1$ if $y = \bar{y}$ and 0 otherwise. (A detailed derivation can be found in the supplementary material.) This also simplifies the scoring function to

$$g(t, y) = \sum_{i,\bar{y}} \beta_i^{\bar{y}}\, k(t_i, \bar{y}, t, y). \tag{11}$$

In this form, we refer to those pairs $(t_i, \bar{y})$ for which $\beta_i^{\bar{y}} \neq 0$ as *support vectors* and those t_i included in at least one support vector as *support patterns* (note that a support pattern corresponds to one of the images in the tracking sequence, and support vectors correspond to samples taken from those images at particular locations). Note that for a given support pattern t_i, only the support vector (t_i, y_i) will have $\beta_i^{y_i} > 0$, while any other support vectors $(t_i, \bar{y})$, $\bar{y} \neq y_i$, will have $\beta_i^{\bar{y}} < 0$. We refer to these as positive and negative support vectors respectively.

4.5 Learning/Adaptation

To update the SVM, we perform online optimisation of (10) using the LaRank approach of Bordes *et al.* [16], [17]. LaRank is an SVM solver based on the sequential minimal optimisation (SMO) approach introduced by Platt [47]. The key idea is that a large quadratic program can be decomposed into a series of small sub-programs (each involving only two Lagrange multipliers) that can be solved analytically.

LaRank adopts such an approach, allowing us to solve our SVM by performing a sequence of SMO steps (see Algorithm 1), each of which monotonically improves (10) with respect to a pair of coefficients $\beta_m^{y_+}$ and $\beta_m^{y_-}$. Due to the constraint $\sum_y \beta_m^y = 0$, these coefficients must be modified by opposite amounts, $\beta_m^{y_+} \leftarrow \beta_m^{y_+} + \lambda$, $\beta_m^{y_-} \leftarrow \beta_m^{y_-} - \lambda$. The value of λ is chosen so as to maximally increase the value of (10): see the supplementary material for a derivation.

The key to LaRank is in how it chooses the coefficients to be optimised by each SMO step. For a given m, y_+ and y_- are chosen to define the feasible search direction with the highest gradient, where the gradient of (10) with respect to a single coefficient β_i^y is given by

$$\begin{aligned} \nabla_i^y &= -\Delta(y, y_i) - \sum_{j,\bar{y}} \beta_j^{\bar{y}} \langle \boldsymbol{\Phi}(t_i, y), \boldsymbol{\Phi}(t_j, \bar{y}) \rangle \\ &= -\Delta(y, y_i) - g(t_i, y). \end{aligned} \tag{12}$$

Algorithm 1 SMOSTEP

Require: m, y_+, y_-, $\mathcal{S}$, β, ∇, C

$k_{(++)} = k(t_m, y_+, t_m, y_+)$
$k_{(--)} = k(t_m, y_-, t_m, y_-)$
$k_{(+-)} = k(t_m, y_+, t_m, y_-)$
$\lambda^u = \frac{\nabla_m^{y_+} - \nabla_m^{y_-}}{k_{(++)} + k_{(--)} - 2k_{(+-)}}$
$\lambda = \max(0, \min(\lambda^u, C\delta(y_+, y_m) - \beta_m^{y_+}))$
Update coefficients
$\beta_m^{y_+} \leftarrow \beta_m^{y_+} + \lambda$
$\beta_m^{y_-} \leftarrow \beta_m^{y_-} - \lambda$
Update gradients
for $(t_i, y) \in \mathcal{S}$ **do**
  $k_{(+)} = k(t_i, y, t_m, y_+)$
  $k_{(-)} = k(t_i, y, t_m, y_-)$
  $\nabla_i^y \leftarrow \nabla_i^y + \lambda\left(k_{(-)} - k_{(+)}\right)$
end for

Three different update steps are considered, which map very naturally onto a tracking framework:

- PROCESSNEW Processes a new example (t_m, y_m). Since all the β_m^ys are initially 0, and only $\beta_m^{y_m}$ may be > 0, $y_+ = y_m$. We choose $y_- = \operatorname{argmin}_{y \in \mathcal{Y}} \nabla_m^y$. During tracking, this corresponds to adding the true label y_m as a positive support vector and searching for the most important sample to become a negative support vector according to the current state of the learner, taking into account the loss function. Note, however, that this step does not necessarily add new support vectors, since the SMO step may not need to adjust the β_m^ys away from 0.
- PROCESSOLD Processes a random existing support pattern t_m. We choose $y_+ = \operatorname{argmax}_{y \in \mathcal{Y}} \nabla_m^y$, but since a feasible search direction requires $\beta_m^{y_+} < \delta(y_+, y_m)C$ (since we need to able to increase $\beta_m^{y_+}$ without violating the relevant constraint), this maximisation will only involve existing support vectors (since if $y_+ \neq y_m$ then $\beta_m^{y_+} < 0$, and if $y_+ = y_m$ then $\beta_m^{y_+}$ must be > 0 because t_m is a support pattern). As for PROCESSNEW, $y_- = \operatorname{argmin}_{y \in \mathcal{Y}} \nabla_m^y$. During tracking, this corresponds to revisiting a frame for which we have retained some support vectors and potentially adding another sample as a negative support vector, as well as adjusting the associated coefficients. Again, this new sample is chosen to take into account the current learner state and loss function.
- OPTIMIZE Processes a random existing support pattern t_m, but only modifies coefficients of existing support vectors. We choose y_+ as for PROCESSOLD, and set $y_- = \operatorname{argmin}_{y \in \mathcal{Y}_m} \nabla_m^y$, where $\mathcal{Y}_m = \{y \in \mathcal{Y} \mid \beta_m^y \neq 0\}$.

PROCESSNEW and PROCESSOLD steps are both able to add new support vectors, which gives the learner the ability to perform sample selection during tracking and discover important negative samples. This selection involves searching over $\mathcal{Y}$ to minimise ∇_m^y, which may be a relatively expensive operation. In practice, we found that for the 2D translation case, it was sufficient to sample from $\mathcal{Y}$ on a

polar grid, rather than considering every pixel offset. The OPTIMIZE case only considers existing support vectors, so is a much less expensive operation.

As suggested by Bordes *et al.* [17], we schedule these update steps as follows. A REPROCESS step is defined as a single PROCESSOLD step followed by n_O OPTIMIZE steps. Given a new training example (t_i, y_i), we call a single PROCESSNEW step followed by n_R REPROCESS steps. In practice, we typically set $n_O = n_R = 10$, as recommended in [17], but the effects of changing them can be seen in the experiments we perform in the supplementary material.

During tracking, we maintain a set of support vectors $\mathcal{S} \subset \mathcal{T} \times \mathcal{Y}$. For each $(t_i, y) \in \mathcal{S}$, we store the coefficient β_i^y and gradient ∇_i^y, which are both incrementally updated during an SMO step (see the supplementary material for the derivation of the gradient update). If the SMO step results in a β_i^y becoming 0, the corresponding support vector is removed from $\mathcal{S}$.

4.6 Incorporating a Budget

The kernelised SVM we have described thus far suffers from the so-called *curse of kernelisation* (the number of support vectors it maintains can grow unboundedly over time). This is problematic for two reasons. Firstly, storage costs increase linearly with the number of support vectors. Secondly, evaluating the SVM is linear in the size of the support vector set: indeed, we can see from (11) that evaluating $g(t, y)$ involves evaluating scalar products (or kernel functions) between (t, y) and each support vector. Moreover, since (12) involves evaluating g, both the PROCESSNEW and PROCESSOLD update steps will become more expensive as the number of support vectors increases. This issue is particularly important in the case of tracking, as in principle we could be presented with an infinite number of training examples. It is thus crucial to limit the number of support vectors maintained in order to achieve efficient tracking using a reasonable amount of memory.

Recently, a number of approaches have been proposed for online learning of *classification* SVMs on a fixed budget [18], [19], meaning the number of support vectors cannot exceed a specified limit. If the limit has already been reached and a new support vector needs to be added, these approaches identify a suitable support vector to remove and adjust the coefficients of the remaining support vectors as necessary to compensate for the removal.

We build on these approaches to devise a budgeting strategy for our scenario. Like Wang *et al.* [19], we choose to remove the support vector that results in the smallest change to the weight vector $\mathbf{w}$, as measured by $\|\Delta\mathbf{w}\|^2$. However, as with the SMO step used during optimisation, we must also ensure that the $\sum_y \beta_i^y = 0$ constraints remain satisfied: this means that when we remove a support vector, we must modify the coefficient of one of the other support vectors for the same support pattern in order to compensate for the change. In practice, there is only one positive support vector for each support pattern, so we restrict our choice of which support vector to remove to negative support vectors, and modify the coefficient of the positive support vector for that pattern to compensate for the removal. A special case occurs when there are only two support vectors being maintained for a pattern (i.e. the positive one and a single negative one): in that case, after removing the negative support vector, we also remove the positive support vector (whose coefficient is now 0) and the corresponding pattern. In general, removing the negative support vector (t_r, y) results in the following change to $\mathbf{w}$:

$$\Delta\mathbf{w} = -\beta_r^y \mathbf{\Phi}(t_r, y) + \beta_r^y \mathbf{\Phi}(t_r, y_r). \tag{13}$$

However, since we only have an explicit expression for $\mathbf{\Phi}$ if we are using a linear kernel, we are not in general able to calculate $\Delta\mathbf{w}$ explicitly. Instead, we use $\|\Delta\mathbf{w}\|^2$, which can be calculated in terms of the joint kernel map k:

$$\begin{aligned}\|\Delta\mathbf{w}\|^2 = \quad & (\beta_r^y)^2 \{k(t_r, y, t_r, y) + k(t_r, y_r, t_r, y_r) \\ & - 2k(t_r, y, t_r, y_r)\}\end{aligned} \tag{14}$$

Each time the budget is exceeded, we remove the negative support vector resulting in the minimum $\|\Delta\mathbf{w}\|^2$. We show in the experimental section that this does not impact significantly on tracking performance, even with modest budget sizes.

We note in passing that because we do not directly consider positive support vectors for removal during budgeting, the only way in which a negative support vector can be removed is if it causes the minimum change in $\mathbf{w}$. Thus, the $\sum_y \beta_i^y = 0$ constraints do not cause us to discard potentially useful information about the scene background. However, they can force us to maintain less useful positive support vectors to balance the useful negatives, which can in theory take up valuable space in our support vector budget. We did not observe this to be a problem in practice.

Algorithm 2 Struck tracking loop.

Require: f, t, $\mathbf{p}_t$, $\mathcal{S}_t$
Propagate the estimated object configuration
$y_t = f(t)$
$\mathbf{p}_{t+1} = y_t(\mathbf{p}_t)$
Update the SVM
PROCESSNEW(t, y_t)
MAINTAINBUDGET()
for $i = 1$ to n_R **do**
  PROCESSOLD()
  MAINTAINBUDGET()
  for $j = 1$ to n_O **do**
    OPTIMIZE()
  end for
end for
return $\mathbf{p}_{t+1}$, $\mathcal{S}_{t+1}$

4.7 Practical Considerations

The overall tracking loop for Struck is shown in Algorithm 2. Having learnt/adapted the SVM as just discussed, we can use it to propagate the estimated object configuration from one image in the tracking sequence to the next as described in §4.1. However, the formulation discussed

thus far has been entirely general, and we have not yet committed ourselves to specific choices for our parameter vectors, search spaces, kernel functions and image features. At an implementation level, these are obviously crucial, so we discuss them now.

4.7.1 Parameter Vectors and Search Spaces

We consider two different types of parameter vector, one designed to handle only 2D translation, and the other designed to handle both 2D translation and scale. For 2D translation, we use parameter vectors of the form $(x, y) \in \mathbb{N}^2$, denoting the top-left corner of a fixed-size, rectangular bounding box around the target object (the size is obtained from the initial bounding box in the first image). For our search space, we consider all integer-valued offsets within a finite search radius r (e.g. $r = 30$):

$$\mathcal{Y} = \{\mathbf{p} \mapsto \mathbf{p} + \Delta\mathbf{p} : \|\Delta\mathbf{p}\| < r\} \tag{15}$$

For 2D translation and scale, we use parameter vectors of the form $(x, y, s) \in \mathbb{N}^2 \times \mathbb{R}^+$, in which the scale factor s denotes the amount by which to scale the initial fixed-size bounding box relative to its centre. For our search space, we consider the same offsets as before, but also relative scaling factors that scale s from one image to the next. For example, we might choose possible scaling factors of $\{0.95, 0.96, ..., 1.05\}$, allowing the bounding box to become at most 5% smaller or larger between consecutive frames.

4.7.2 Kernel Functions

We consider three different types of kernel:

- The *linear* kernel sets $\mathbf{\Phi}(t, y) = \mathbf{x}_{t+1}^{y(\mathbf{p}_t)}$, whence $k(t, y, \bar{t}, \bar{y}) = \langle \mathbf{x}_{t+1}^{y(\mathbf{p}_t)}, \mathbf{x}_{\bar{t}+1}^{\bar{y}(\mathbf{p}_{\bar{t}})} \rangle$.
- The *Gaussian* kernel defines $k(t, y, \bar{t}, \bar{y})$ to be $\exp(-\sigma \|\mathbf{x}_{t+1}^{y(\mathbf{p}_t)} - \mathbf{x}_{\bar{t}+1}^{\bar{y}(\mathbf{p}_{\bar{t}})}\|^2)$, where σ is the standard deviation of the Gaussian. (We used $\sigma = 0.2$ in our experiments.)
- The *intersection* kernel defines $k(t, y, \bar{t}, \bar{y})$ to be $\frac{1}{D} \sum_{j=1}^{D} \min(\mathbf{x}_{t+1}^{y(\mathbf{p}_t)}[j], \mathbf{x}_{\bar{t}+1}^{\bar{y}(\mathbf{p}_{\bar{t}})}[j])$, where D is the feature vector size.

Of these, the linear kernel is especially computationally attractive because it uses an explicit representation for $\mathbf{\Phi}$, whence the SVM can be evaluated efficiently on a set of feature vectors by first calculating its weight vector $\mathbf{w}$, and then computing a dot product of $\mathbf{w}$ with each of the feature vectors. Specifically, we can derive:

$$\begin{aligned} g(t, y) &= \sum_{i, \bar{y}} \beta_i^{\bar{y}} \langle \mathbf{x}_{t_i+1}^{\bar{y}(\mathbf{p}_{t_i})}, \mathbf{x}_{t+1}^{y(\mathbf{p}_t)} \rangle \\ &= \sum_{i, \bar{y}} \beta_i^{\bar{y}} \sum_j \mathbf{x}_{t_i+1}^{\bar{y}(\mathbf{p}_{t_i})}[j] \times \mathbf{x}_{t+1}^{y(\mathbf{p}_t)}[j] \\ &= \sum_j \mathbf{x}_{t+1}^{y(\mathbf{p}_t)}[j] \underbrace{\left(\sum_{i, \bar{y}} \beta_i^{\bar{y}} \mathbf{x}_{t_i+1}^{\bar{y}(\mathbf{p}_{t_i})} \right)}_{\mathbf{w}}[j] \\ &= \langle \mathbf{w}, \mathbf{x}_{t+1}^{y(\mathbf{p}_t)} \rangle \end{aligned} \tag{16}$$

However, a disadvantage of this kernel is that it can cope poorly with data that is not linearly separable in feature space, so it is not always a good choice when trying to learn a classifier that incorporates different views of the same object. To address this problem, non-linear kernels such as the Gaussian or intersection kernels map feature vectors to a high-dimensional space in which linear separability may be easier to achieve. The downside is that we no longer have an explicit representation for $\mathbf{\Phi}$ and must evaluate the SVM in terms of its support vectors (11).

4.7.3 Image Features

We consider three different types of image feature:

- *Raw* features, obtained by scaling an image patch to 16×16 pixels, normalising the greyscale value of each pixel into the range $[0, 1]$ and converting the result into a 256D feature vector by reading the normalised values in raster order.
- *Haar* features, inspired by the features used in the Viola-Jones face detector [11], and calculated as a weighted combination of box sums over the pixels of the current image. By default, we use 6 different types of Haar feature (see Figure 3), arranged at 2 scales on a 4×4 grid, resulting in a 192D feature vector with each feature normalised to the range $[-1, 1]$.
- *Histogram* features, obtained by concatenating 16-bin intensity histograms from a spatial pyramid of 4 levels. At each level L, the patch is divided into $L \times L$ cells, resulting in a $16 \sum_{L=1}^{4} L^2 = 480$D feature vector.

Some kernel/feature combinations tend to work better than others. In particular, our raw features worked best with a linear kernel, our Haar features worked best with a Gaussian kernel and our histogram features worked best with an intersection kernel. Moreover, the number of features used mediates a trade-off between tracking performance and speed: in general, it is possible to achieve better tracking performance using larger feature vectors in exchange for a decrease in the speed of the tracker. In our experiments, we found that our Gaussian/Haar combination achieved a reasonable compromise between tracking performance and speed; however, we obtained better performance using the multi-kernel approach described in the following section (which uses larger feature vectors), and better speed by combining a linear kernel (which is a natural fit to the GPU architecture) with raw features (which are cheap to compute). To quantify the difference that larger feature vectors can make, our supplementary material contains an additional experiment in which we investigate the effects of using more Haar features than described here.

4.7.4 Multiple-Kernel Learning (MKL)

In addition to experimenting with individual kernels, we investigated combining kernels to implement a basic form of multiple-kernel learning (MKL). The idea is to average the results of a number of kernels $k^{(i)}$ on potentially different sets of features:

$$k(t, y, \bar{t}, \bar{y}) = \frac{1}{N_k} \sum_{i=1}^{N_k} k^{(i)}(t, y, \bar{t}, \bar{y})$$

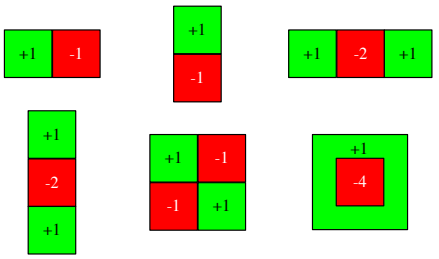


Fig. 3: The different types of Haar feature used by Struck. The numbers in the boxes are the (unnormalised) weights used when calculating the features. Note that no feature requires more than four boxes, which makes for efficient evaluation on the GPU (see §5.4).

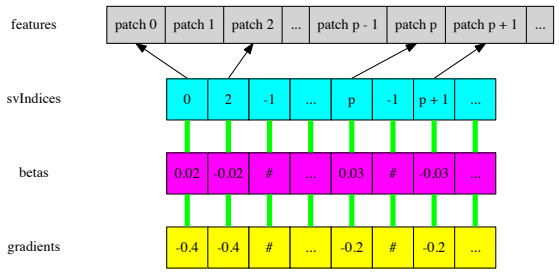


Fig. 4: The representation of the SVM in ThunderStruck.

It has been shown [48] that in terms of performance, full MKL (in which the relative weighting of the different kernels is learned from training data) does not greatly improve upon this simple approach.

We found experimentally (see §6.3) that MKL was able to yield substantial improvements in tracking performance, enough for our mklHGHI tracking variant (which combines a Gaussian kernel with an intersection one on Haar and histogram features) to outperform the state-of-the-art KCF tracker [22] *on the Wu* et al. [21] *benchmark*. However, since the larger feature vectors involved have a significant impact on the speed of our tracker, simpler variants may be preferable for tasks for which a reasonable compromise between performance and speed must be achieved.

5 ThunderStruck

To investigate the speed potential of an optimised implementation of Struck, we implemented a CUDA version of it called *ThunderStruck*. In this section, we briefly describe some of the key features of this implementation (additional details can be found in the supplementary material). Its speed is compared to the original CPU version in §6.5.

5.1 SVM Representation

A Struck SVM maintains two separate sets of data: the features computed for patches associated with the support patterns (some of the previously-seen frames), and a record of which patches are in the current set of support vectors, together with their corresponding β coefficients and gradient values. Since GPU code is easier to optimise when using fixed-size arrays and it is possible to use a finite budget of support vectors without degrading tracking performance, we chose to use a fixed-size representation for our SVM data in ThunderStruck (see Figure 4). We use a single large GPU-based array to store all of the features for every patch within every support pattern. We use a smaller array of indices to specify the current support vectors: each element of this array either refers to a patch or is -1 to indicate the absence of a support vector. The corresponding β coefficients and gradient values are stored in the similarly-sized arrays $betas$ and $gradients$, such that the β value for support vector k is stored in $betas[k]$ and the gradient value is stored in $gradients[k]$. Since access to the three support vector arrays is required on both the CPU and GPU, we store mirrored copies of them in both places to minimise costly memory transfers over the CPU-GPU bus (note that the storage cost involved is low due to the small size of the arrays).

Addition and removal of support vectors can be implemented via a simple ID allocator that maintains a set of used IDs and a set of free ones. The IDs index into the $svIndices$, $betas$ and $gradients$ arrays. To add a support vector, we allocate a free ID and use the corresponding elements in the arrays; to remove one, we deallocate the ID, causing it to be returned to the free set, and set the corresponding element of $svIndices$ to -1.

5.2 SVM Evaluation

We focus on the key part of SVM evaluation using the linear kernel here, and defer both how to calculate the SVM's weight vector $\mathbf{w}$ and how to implement other kernels to the supplementary material. For the linear kernel, we first calculate $\mathbf{w}$ and then compute a straightforward dot product of $\mathbf{w}$ with each of the samples. Each thread block in our implementation evaluates the SVM for a single sample (see Figure 5). Individual threads multiply corresponding elements of the SVM's weight vector and the sample's feature vector and store the results in shared memory (accessing shared memory is hundreds of times faster than accessing global memory). The result of the dot product is then computed in shared memory using a reduction.

5.3 Budget Maintenance

As a result of the fixed-size SVM representation we use, the way in which we maintain our support vector budget for ThunderStruck has to differ slightly from that described in §4.6. In particular, instead of removing a support vector when the budget is exceeded, we now remove a support vector at the point at which we need to add a new one but have no available space in the arrays. We found this to be an equivalent scheme that made little difference to the results; however, it would nevertheless be possible to implement a version of the original scheme for fixed-size arrays by adding additional space at the ends of the arrays and maintaining the budget after adding a support vector.

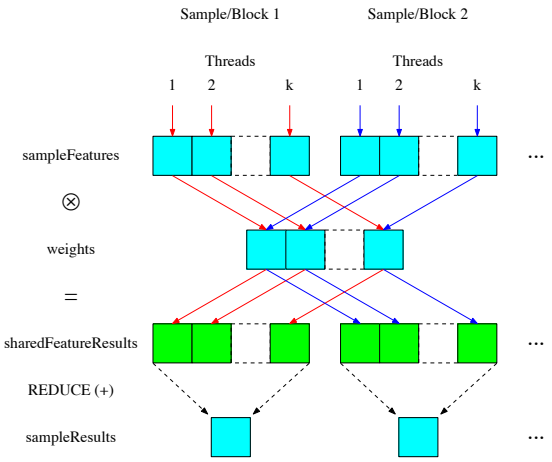


Fig. 5: To evaluate an SVM with a linear kernel efficiently using CUDA, we use a thread block for each sample and compute a dot product between the sample's features and the SVM's weight vector. Each dot product is computed using a pointwise multiplication followed by a reduction in shared memory. The coloured boxes indicate where the data is stored (cyan = global memory, green = shared memory). The coloured arrows distinguish between thread blocks.

5.4 Feature Calculation

Prior to calculating features, we transfer the current frame and any derived images that are best computed on the CPU (e.g. an integral image, for computing Haar features) across to the GPU as CUDA textures. To calculate raw features, we use a variant of the approach in the original Struck that is a better fit for the GPU architecture. Rather than scaling a patch to 16×16 pixels and then densely sampling the result, we instead sample from a 16×16 uniform grid placed over the unscaled patch: this allows us to avoid resizing patches on the GPU, whilst producing equivalent results. We compute each raw feature on a separate CUDA thread and calculate the features for all patches in parallel.

To calculate Haar features, we observe that each Haar feature we use can be calculated as the weighted combination of at most four box sums over the pixels of the current frame (see Figure 3). The sum of each box can be calculated efficiently using the integral image for the frame [11]. We can thus compute all of the Haar features without needing to branch on feature type on the GPU by making each CUDA thread calculate a single feature and assigning zero weights to boxes that are unnecessary for features of particular types.

5.5 Multi-Threading on the CPU

Whilst it does not represent an improvement to the speed of the tracker itself, it is worth observing that we were able to obtain a further improvement in the speed of the ThunderStruck system as a whole by running the tracker on one CPU thread whilst rendering the output on another. This allows the tracker to process the next frame whilst the current one is still being rendered. This improvement could clearly also have been made to the CPU version of Struck.

6 Experiments

We evaluate our CPU and GPU implementations of Struck on the full 50-video benchmark of Wu *et al.* [21]. In the main paper, we perform experiments at a single scale, first using only individual kernels (§6.2), and then combining multiple kernels (§6.3) as described in §4.7.4. We also perform experiments showing the difference structured learning makes in improving tracking performance relative to a baseline classification SVM (§6.4), and evaluate our implementations for speed (§6.5). Further experiments investigating multi-scale tracking and the effects of changing various parameters of our approach can be found in the supplementary material, together with a study comparing our approach to the state-of-the-art KCF tracker [22] and more details on the effects of structured learning.

6.1 Benchmark

The Wu dataset consists of numerous test videos from the recent literature. In their paper, the authors performed a large-scale evaluation that compared 29 trackers (including the CPU version of Struck, SCM [49], TLD [50] and ASLA [51]). Their key ideas were (a) to perturb the initialisation of the trackers in both time and space to improve the robustness of the evaluation, and (b) to evaluate the trackers on sequences that had been annotated to highlight tracking challenges, e.g. fast motion, occlusion, or changes in scale or illumination. Trackers were evaluated using a variety of different tests, in each case using *location error* (the percentage of frames whose predicted bounding boxes were within a fixed pixel threshold of the ground truth bounding boxes) as a precision measure, and *overlap* (the percentage of frames whose predicted bounding boxes overlapped the ground truth bounding boxes by more than a fixed threshold) as a success measure. These measures were calculated for a range of thresholds in each case. The trackers were ranked in terms of precision using a fixed location error threshold of 20 pixels, and in terms of success using the area under the curve (AuC) approach.

Three different types of test were performed to compare the trackers' performance: (a) one-pass evaluation (OPE), which initialises the trackers with the ground truth bounding box from the first frame of each sequence, (b) temporal robustness evaluation (TRE), which initialises the trackers at another starting frame in the sequence, and (c) spatial robustness evaluation (SRE), which shifts or scales the ground truth bounding box in the first frame. Overall performance tests of all three types were performed for each tracker on all of the available sequences; further tests were also performed to compare the trackers' performance when restricted to specific types of sequence, e.g. those with a significant amount of fast motion or occlusion.

6.2 Single-scale, single-kernel tracking

For our initial experiments, we tested single-scale (i.e. 2D translation only), single-kernel CPU and CUDA variants of Struck on the entire dataset using the TRE and SRE tests and various different feature/kernel/budget combinations.

Tracker	Variant	Features	Kernel	Budget	Success		Precision	
					TRE	SRE	TRE	SRE
Struck	fkbRL100	Raw	Linear	100	0.471	0.400	0.651	0.569
Struck	fkbRL25	Raw	Linear	25	0.446	0.377	0.611	0.529
Struck	fkbHG100	Haar	Gaussian	100	*0.504*	0.434	0.706	0.628
Struck	fkbHG25	Haar	Gaussian	25	0.479	0.406	0.665	0.579
ThunderStruck	fkbRL100	Raw	Linear	100	0.459	0.384	0.633	0.546
ThunderStruck	fkbRL25	Raw	Linear	25	0.367	0.308	0.494	0.421
ThunderStruck	fkbHG100	Haar	Gaussian	100	0.490	0.417	*0.681*	*0.602*
ThunderStruck	fkbHG25	Haar	Gaussian	25	0.410	0.350	0.562	0.479
Baseline	–	Haar	Gaussian	100	0.473	0.401	0.656	0.567
ASLA	–	–	–	–	0.485	*0.421*	0.620	0.577
SCM	–	–	–	–	0.513	0.420	0.652	0.575
TLD	–	–	–	–	0.448	0.402	0.624	0.573
KCF	–	–	–	–	**0.556**	**0.463**	**0.774**	**0.683**

TABLE 1: The tracking performance of single-scale, single-kernel Struck and ThunderStruck variants on the Wu *et al.* [21] benchmark using various feature/kernel/budget combinations. We used search radii of 30 pixels for propagation and 60 pixels for learning, and set n_R and n_O, the numbers of reprocessing and optimisation steps used for LaRank, to 10.

For these experiments, we searched for the target object in each new frame using a fixed search radius of 30 pixels around the previous object location. For learning the SVM, we used a larger search radius of 60 pixels to help ensure stability, and sampled potential transformations from a polar grid with 5 radial and 16 angular divisions, giving 81 potential transformations in total (including the identity transformation).

Table 1 shows the results obtained by our two implementations for various combinations of parameters, along with published results from the best-performing methods in the original Wu benchmark, and the results we obtained for the state-of-the-art KCF tracker [22]. It can be seen from these results that Struck fkbHG100, which uses Haar features, a Gaussian kernel and a budget of 100 support vectors, outperforms methods like ASLA, SCM and TLD on all but the TRE success tests, in some cases by a considerable margin. The corresponding CUDA variant of our tracker (ThunderStruck fkbHG100) achieves slightly lower tracking performance than this due to differences in the way in which the budget maintenance step is implemented (see §5.3), but compensates for this by a significant increase in speed (see §6.5). The Struck fkbHG25 results demonstrate that it is possible to achieve reasonable tracking performance using a relatively small number of support vectors.

Although Struck fkbHG100 achieves excellent results in comparison to trackers such as ASLA, SCM and TLD, the KCF tracker substantially outperforms it. We believe a key advantage KCF has is its relative computational efficiency in comparison to Struck, which gives it the freedom both to use better features than Struck fkbHG100 (it uses HOG rather than Haar features) and to sample densely around the object without encountering speed or memory issues. Of these, we believe that the features make the more significant difference: when we tried densely sampling around the object, we found that it made little difference to the overall results. By contrast, as our experiments in the next section demonstrate, when we used more effective features as part of a multi-kernel learning (MKL) approach, we achieved significant improvements in tracking performance, allowing the mklHGHI variant of our tracker to outperform KCF on all but the TRE success tests of the Wu benchmark.

6.3 Single-scale, multi-kernel tracking

In order to examine the effects of the multi-kernel learning (MKL) approach we articulated in §4.7.4, we compared a number of multi-kernel variants of our Struck tracker (using various combinations of features and kernels) with the corresponding single-kernel variants by running them on the entire Wu benchmark. The results are shown in Table 2. For all variants, we set the unrelated parameters to default values to ensure a fair comparison (specifically, we used a support vector budget of 100, a learning radius of 60 pixels and a propagation radius of 30 pixels, and set n_R and n_O, respectively the numbers of reprocessing and optimisation steps used for LaRank, to 10).

Our results demonstrate that by combining the right features and kernels, it is possible to achieve significant improvements in tracking performance. In particular, the mklHGHI variant of our tracker, which combines a Gaussian kernel with an intersection kernel on Haar and histogram features, achieves state-of-the-art results on the Wu benchmark, outperforming even KCF. This improvement in tracking performance is bought at the expense of using larger feature vectors, which leads to a roughly eight-fold decrease in the speed of the tracker when compared to Struck fkbHG100 (see §6.5). However, given the increases in speed obtained by ThunderStruck, we believe that a careful implementation of Struck mklHGHI on the GPU could run at a decent frame rate.

6.4 Effects of structured learning

To investigate the effects of structured learning on the performance of our tracker, we compared the results of Struck fkbHG100 against a baseline classification SVM with the same parameters (Haar features, Gaussian kernel, budget of 100) on the Wu benchmark. To achieve this, we modified our tracking framework to train the learner using binary rather than structured examples. In each frame, a single positive example was generated from the estimated object

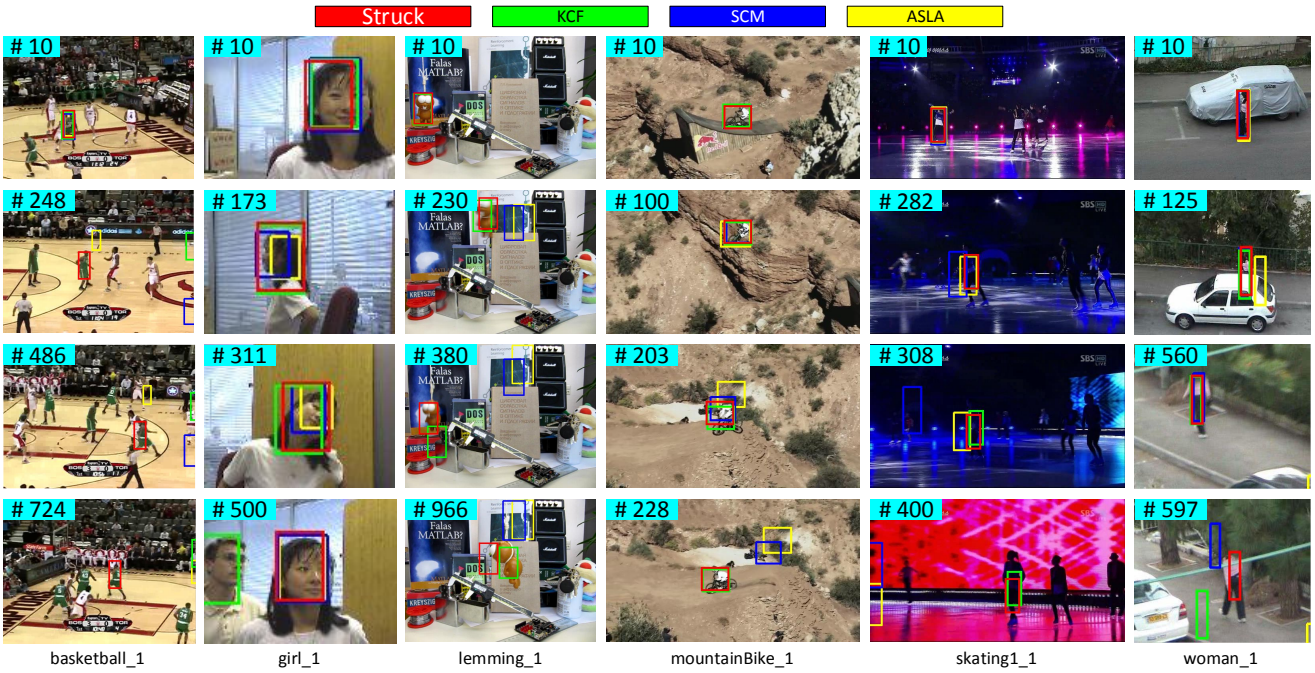


Fig. 6: Example frames from benchmark sequences, comparing the results of Struck (variant mklHGHI) with KCF [22], SCM [49] and ASLA [51]. Videos of these results can be found at https://goo.gl/cJ1Dg7.

Tracker	Variant	Features/Kernels	Feature Count	Success		Precision	
				TRE	SRE	TRE	SRE
Struck	mklHGRL	Haar/Gaussian + Raw/Linear	448	0.476	0.401	0.660	0.575
Struck	mklHGHI	Haar/Gaussian + Histogram/Intersection	672	0.545	**0.469**	**0.785**	**0.707**
Struck	mklHIRL	Histogram/Intersection + Raw/Linear	736	0.494	0.418	0.690	0.606
Struck	mklHGHIRL	Haar/Gaussian + Histogram/Intersection + Raw/Linear	928	0.495	0.422	0.692	0.610
Struck	fkbHG100	Haar/Gaussian	192	0.504	0.434	0.706	0.628
Struck	fkbHI100	Histogram/Intersection	480	*0.517*	*0.455*	*0.734*	*0.679*
Struck	fkbRL100	Raw/Linear	256	0.471	0.400	0.651	0.569
KCF	–	–	–	**0.556**	0.463	0.774	0.683

TABLE 2: Comparing the tracking performance of some single-kernel variants of Struck with variants that use multiple kernel learning (MKL). For all variants, we use a learning search radius r_L of 60 pixels, a propagation search radius r_P of 30 pixels and a support vector budget of 100, and set n_R and n_O, respectively the numbers of reprocessing and optimisation steps used for LaRank, to 10. We show the results of the KCF tracker for comparison purposes.

configuration, and negative examples were generated by sampling from $\mathcal{Y}$ and keeping examples that overlapped the estimated object configuration by less than 0.5 (i.e. $\theta_u = 1$ and $\theta_l = 0.5$ in the labelling function described in §3.3).

The results of the baseline classifier are shown in Table 1. In comparison with those for Struck fkbHG100, they demonstrate that for the Wu benchmark as a whole, our structured learning framework achieves meaningful improvements over a traditional classification-based approach. Indeed, we show in the supplementary material that our structured learner improves upon the baseline for each individual attribute-based subset of the benchmark. However, the improvements achieved did vary with the attributes of the sequences tested. In particular, for sequences exhibiting fast motion or in which some part of the target leaves the view, we found that the structured learner did not significantly improve upon the baseline. We investigate this further in the supplementary material.

6.5 Timings

To investigate the speed improvements yielded by using the GPU, and to quantify the effects on tracking speed of changing various parameters of our trackers, we timed a number of variants of both Struck and ThunderStruck on the entire Wu benchmark (see Table 3). The machine we used had a 12-core Intel i7-4960X CPU, running at 3.6 GHz, and an NVIDIA GeForce GTX Titan Black GPU, and we ran our experiments under Ubuntu 14.04. In all cases, we suppressed rendering and text output so as to time only the speed of the actual tracking process.

Our results demonstrate that in a head-to-head comparison between similar variants of Struck and ThunderStruck, our GPU variants achieve significantly higher average frame rates. In particular, our default fkbHG100 variant of ThunderStruck (Haar features, a Gaussian kernel with $\sigma = 0.2$ and a budget of 100 support vectors) runs at an average of 93.9 FPS, whilst our ro5_5 variant, which is

Tracker	Variant	Average FPS
Struck	fkbRL100	20.9
Struck	fkbHG100	20.8
Struck	mklHGHI	2.4
ThunderStruck	fkbRL100	**146.3**
ThunderStruck	fkbHG100	*93.9*
ThunderStruck	ro5_5	125.1
ThunderStruck	sHG95_105_1	19.9

TABLE 3: Comparing the average speed (in frames per second) of a number of variants of Struck and ThunderStruck over the entire Wu benchmark. For details of the parameters used and the tracking performance obtained for each variant, see the corresponding experiments sections.

similar to fkbHG100 but uses fewer LaRank optimisation steps, is even faster, running at an average of 125.1 FPS. Even faster tracking can be obtained using ThunderStruck fkbRL100 (raw features, a linear kernel and a budget of 100 support vectors), but with a sizeable decrease in tracking performance (see Table 1).

By contrast, our multi-kernel mklHGHI variant of Struck achieves high performance values (see §6.3), but is comparatively slow. Our current multi-kernel implementation is CPU-only, and given the speed improvements demonstrated by ThunderStruck, we believe it could be speeded up significantly using the GPU; however, the larger feature vectors and more complicated kernels used by mklHGHI mean that it will always be slower than simpler variants.

The effect that incorporating scale has on speed can be seen by comparing ThunderStruck sHG95_105_1 (a multi-scale variant that uses 11 possible scaling factors) with ThunderStruck fkbHG100. Although the multi-scale variant is nearly five times slower, owing to the much greater number of object configurations it must consider during the propagation step, it remains real-time at nearly 20 FPS.

7 Conclusion

In this paper, we have presented Struck, an adaptive tracking-by-detection framework based on structured output prediction. From a learning point of view, we take advantage of the well-studied large margin theory of SVMs, which brings benefits in terms of generalisation and robustness to noise (both in the input and output spaces). Unlike prior methods based on classification, our algorithm does not rely on a heuristic intermediate step for producing labelled binary samples with which to update the classifier, which is often a source of error during tracking. Our approach uses an online structured output SVM learning framework, making it easy to incorporate image features and kernels. To prevent unbounded growth in the number of support vectors, and allow real-time performance, we introduced a budget maintenance mechanism for online structured output SVMs. We have experimentally shown the benefits of structured output prediction by comparing our approach with a baseline classification SVM.

In comparison with the state-of-the-art KCF tracker, the conventional single-kernel variants of Struck achieve lower tracking performance. However, by incorporating larger feature vectors into a multi-kernel tracking approach, we have shown that we are able to outperform KCF and achieve state-of-the-art results on the popular Wu *et al.* benchmark.

From a speed perspective, we have shown how Struck can be implemented on the GPU using CUDA. This improves upon the already real-time performance of the original unoptimised CPU implementation of Struck to achieve high frame-rates.

Acknowledgements

Financial support was provided by ERC grant ERC-2012-AdG 321162-HELIOS. Stuart Golodetz was funded via a Royal Society Brian Mercer Award for Innovation awarded to Stephen L. Hicks. Philip H. S. Torr is in receipt of a Royal Society Wolfson Research Merit Award and acknowledges support from the Leverhulme Trust and EPSRC.

References

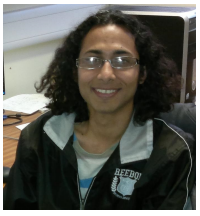
[1] S. Avidan, "Support vector tracking," *IEEE TPAMI*, vol. 26, no. 8, pp. 1064–72, 2004.
[2] H. Grabner, M. Grabner, and H. Bischof, "Real-Time Tracking via On-line Boosting," in *BMVC*, 2006.
[3] B. Babenko, M.-H. Yang, and S. Belongie, "Robust Object Tracking with Online Multiple Instance Learning," *IEEE TPAMI*, 2011.
[4] A. Saffari, M. Godec, T. Pock, C. Leistner, and H. Bischof, "Online Multi-Class LPBoost," in *CVPR*, 2010.
[5] O. Williams, A. Blake, and R. Cipolla, "A Sparse Probabilistic Learning Algorithm for Real-Time Tracking," in *ICCV*, 2003.
[6] C. Leistner, A. Saffari, P. M. Roth, and H. Bischof, "On Robustness of On-line Boosting - A Competitive Study," in *ICCVW*, 2009.
[7] H. Masnadi-Shirazi, V. Mahadevan, and N. Vasconcelos, "On the design of robust classifiers for computer vision," in *CVPR*, 2010.
[8] H. Grabner, C. Leistner, and H. Bischof, "Semi-Supervised On-Line Boosting for Robust Tracking," in *ECCV*, 2008.
[9] A. Saffari, C. Leistner, M. Godec, and H. Bischof, "Robust multi-view boosting with priors," in *ECCV*, 2010.
[10] B. Zeisl, C. Leistner, A. Saffari, and H. Bischof, "Online Semi-Supervised Multiple-Instance Boosting," in *CVPR*, 2010.
[11] P. Viola and M. J. Jones, "Robust Real-Time Face Detection," *IJCV*, vol. 57, no. 2, pp. 137–154, 2004.
[12] M. B. Blaschko and C. H. Lampert, "Learning to Localize Objects with Structured Output Regression," in *ECCV*, 2008.
[13] A. Vedaldi, V. Gulshan, M. Varma, and A. Zisserman, "Multiple kernels for object detection," in *ICCV*, 2009.
[14] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object Detection with Discriminatively Trained Part-Based Models," *IEEE TPAMI*, vol. 32, no. 9, pp. 1627–1645, 2010.
[15] I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun, "Large Margin Methods for Structured and Interdependent Output Variables," *JMLR*, vol. 6, pp. 1453–1484, 2005.
[16] A. Bordes, L. Bottou, P. Gallinari, and J. Weston, "Solving multiclass support vector machines with LaRank," in *ICML*, 2007.
[17] A. Bordes, N. Usunier, and L. Bottou, "Sequence Labelling SVMs Trained in One Pass," in *Proc. ECML-PKDD*, 2008.
[18] K. Crammer, J. Kandola, R. Holloway, and Y. Singer, "Online Classification on a Budget," in *NIPS*, 2003.
[19] Z. Wang, K. Crammer, and S. Vucetic, "Multi-Class Pegasos on a Budget," in *ICML*, 2010.
[20] S. Hare, A. Saffari, and P. H. S. Torr, "Struck: Structured Output Tracking with Kernels," in *ICCV*, 2011.
[21] Y. Wu, J. Lim, and M.-H. Yang, "Online Object Tracking: A Benchmark," in *CVPR*, 2013, pp. 2411–2418.
[22] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-Speed Tracking with Kernelized Correlation Filters," *IEEE TPAMI*, 2015.
[23] A. W. M. Smeulders, D. M. Chu, R. Cucchiara, S. Calderara, A. Dehghan, and M. Shah, "Visual Tracking: An Experimental Survey," *IEEE TPAMI*, vol. 36, no. 7, pp. 1442–1468, 2014.
[24] Y. Pang and H. Ling, "Finding the Best from the Second Bests – Inhibiting Subjective Bias in Evaluation of Visual Tracking Algorithms," in *CVPR*, 2013.

[25] M. Kristan *et al.*, "The Visual Object Tracking VOT2014 challenge results," in *ECCVW*, 2014.
[26] X. Mei, H. Ling, Y. Wu, E. Blasch, and L. Bai, "Minimum Error Bounded Efficient ℓ_1 Tracker with Occlusion Detection," in *CVPR*, 2011, pp. 1257–1264.
[27] N. Wang, J. Wang, and D.-Y. Yeung, "Online Robust Non-negative Dictionary Learning for Visual Tracking," in *ICCV*, 2013.
[28] J. Xing, J. Gao, B. Li, W. Hu, and S. Yan, "Robust Object Tracking with Online Multi-lifespan Dictionary Learning," in *ICCV*, 2013.
[29] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking," *IEEE TSP*, vol. 50, no. 2, pp. 174–188, February 2002.
[30] S. Cao and W. Xue, "Robust Visual Tracking via Adaptive Forest," in *ICICIP*, June 2013, pp. 30–34.
[31] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof, "On-line Random Forests," in *Proc. ICCV-OLCV*, 2009.
[32] J. Santner, C. Leistner, A. Saffari, T. Pock, and H. Bischof, "PROST: Parallel Robust Online Simple Tracking," in *CVPR*, 2010, pp. 723–730.
[33] Q. Bai, Z. Wu, S. Sclaroff, M. Betke, and C. Monnier, "Randomized Ensemble Tracking," in *ICCV*, 2013, pp. 2040–2047.
[34] N. Wang and D.-Y. Yeung, "Ensemble-Based Tracking: Aggregating Crowdsourced Structured Time Series Data," in *ICML*, 2014.
[35] S. Duffner and C. Garcia, "PixelTrack: a fast adaptive algorithm for tracking non-rigid objects," in *ICCV*, 2013, pp. 2480–2487.
[36] X. Ren and J. Malik, "Tracking as Repeated Figure/Ground Segmentation," in *CVPR*, 2007.
[37] S. Wang, H. Lu, F. Yang, and M.-H. Yang, "Superpixel Tracking," in *ICCV*, 2011, pp. 1323–1330.
[38] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "Exploiting the Circulant Structure of Tracking-by-detection with Kernels," in *ECCV*. Springer Berlin Heidelberg, 2012, pp. 702–715.
[39] M. Danelljan, F. S. Khan, M. Felsberg, and J. van de Weijer, "Adaptive Color Attributes for Real-Time Visual Tracking," in *CVPR*, 2014.
[40] R. Yao, Q. Shi, C. Shen, Y. Zhang, and A. van den Hengel, "Part-based Visual Tracking with Online Latent Structural Learning," in *CVPR*, 2013, pp. 2363–2370.
[41] C. Shen, G. Lin, and A. van den Hengel, "StructBoost: Boosting Methods for Predicting Structured Output Variables," arXiv:1302.3283v3, February 2014.
[42] S. Kim, S. Kwak, J. Feyereisl, and B. Han, "Online Multi-Target Tracking by Large Margin Structured Learning," in *ACCV*, 2013, pp. 98–111.
[43] F. Pernici and A. D. Bimbo, "Object Tracking by Oversampling Local Features," *IEEE TPAMI*, vol. PP, no. 99, 2013.
[44] Y. Lu, T. Wu, and S.-C. Zhu, "Online Object Tracking, Learning and Parsing with And-Or Graphs," in *CVPR*, 2014.
[45] L. Zhang and L. van der Maaten, "Preserving Structure in Model-Free Tracking," *IEEE TPAMI*, vol. 36, no. 4, pp. 756–769, 2014.
[46] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2007, pp. 707–710.
[47] J. C. Platt, *Fast Training of Support Vector Machines Using Sequential Minimal Optimization*. MIT Press, 1999, pp. 185–208.
[48] P. Gehler and S. Nowozin, "On Feature Combination for Multiclass Object Classification," in *ICCV*, 2009.
[49] W. Zhong, H. Lu, and M.-H. Yang, "Robust Object Tracking via Sparsity-based Collaborative Model," in *CVPR*, 2012.
[50] Z. Kalal, J. Matas, and K. Mikolajczyk, "P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints," in *CVPR*, 2010.
[51] X. Jia, H. Lu, and M.-H. Yang, "Visual Tracking via Adaptive Structural Local Sparse Appearance Model," in *CVPR*, 2012.

Sam Hare received his PhD from Oxford Brookes University in 2012 as a member of the Brookes Vision Group. His PhD research focused on online structured learning for real-time computer vision applications, and was carried out in collaboration with Sony Computer Entertainment Europe. He is now co-founder and CTO of Obvious Engineering, developing 3D scene reconstruction technology for mobile devices.

Stuart Golodetz obtained his PhD in Computer Science at the University of Oxford in 2011, working on 3D image segmentation and feature identification. After working in industry for both SunGard and Semmle, he is currently a stipendiary lecturer at the University of Oxford. His areas of interest include image processing, C++ and computer games development. He reviews papers for IJCV and SOCP.

Amir Saffari obtained his PhD on Online and Semi-supervised Learning from Graz University of Technology in 2010, and in the same year joined Sony Computer Entertainment Europe's London Studio to work on computer vision-based technology for video games – notably Wonderbook, an interactive augmented reality book for PS3. In 2013, he moved to Affectv as the Director of Research and leads the data science team working on big data machine learning algorithms.



Vibhav Vineet is a postdoc at Stanford University and closely collaborates with the I3D group at Microsoft Research Cambridge. His research interests are in computer vision and machine learning.

Ming-Ming Cheng received his PhD degree from Tsinghua University in 2012. He is currently an associate professor at Nankai University. His research interests include computer graphics, computer vision, image processing and image retrieval. He has received a Google PhD fellowship award, an IBM PhD fellowship award, and a "New PhD Researcher Award" from the Chinese Ministry of Education. He reviews papers regularly for IEEE TPAMI, ACM SIGGRAPH, etc.

Stephen L. Hicks received his PhD from the University of Sydney and is a Research Fellow in neuroscience and visual prosthetics at the University of Oxford. He leads a program of research to develop and validate non-invasive sight enhancement techniques based on computer vision for legally blind individuals. He won the 2013 Royal Society Brian Mercer Award for Innovation and the 2014 SET for Britain prize for Engineering. Stephen is funded by the NIHR i4i program.

Philip H. S. Torr received the PhD degree from Oxford University. After working for another three years at Oxford, he worked for six years as a research scientist for Microsoft Research, first in Redmond, then in Cambridge, founding the vision side of the Machine Learning and Perception Group. He is now a professor at Oxford University. He has won awards from several top vision conferences, including ICCV, CVPR, ECCV, NIPS and BMVC. He is a senior member of the IEEE, Royal Society Wolfson Research Merit Award holder, and program co-chair of ICCV 2013.