AN ABSTRACT OF THE THESIS OF

Trevor Fiez for the degree of Master of Science in Computer Science presented on June 14, 2017.

Title: An Analysis of Training Methodologies for Deep Visual Tracking

Abstract approved: _____

Alan Fern Sinisa Todorovic

This thesis considers the problem of training convolutional neural networks for online visual tracking. A major challenge for single object visual tracking is that most training sets with frame-level track annotations are quite small, due to the prohibitive cost of manual annotation. Current training approaches either supplement the annotations with other data sources (e.g., object-detection training data) or generate noisy variants of the track annotations. In either case, the data generation and training methods have ignored the fact that tracking involves sequences of decisions (one per frame) that are dependent on one another. Thus, the objectives optimized by these learning algorithms are not directly tied to the end goal of tracking performance. To further study this issue, we consider the state-of-the-art imitation learning algorithm, DAGGER, for training an online tracker. We observe that the DAGGER faces difficulty when applied to tracking, because online trackers typically experience unrecoverable failures, especially early in training. To rectify this issue we introduce, analyze, and evaluate a variation of DAGGER, called DAGGER with Resets (DAGGER^2), a novel imitation learning framework which maintains the theoretical properties of DAGGER and is more appropriate for training deep trackers. Our main contribution is to compare different training methods, including DAGGER and DAGGER^2, across a variety of datasets and multiple trackers. Our experimental results show this principled training approach and methodical random augmentation is able to outperform existing training approaches across multiple visual tracking datasets.

An Analysis of Training Methodologies for Deep Visual Tracking

by

Trevor Fiez

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 14, 2017
Commencement June 2017

Master of Science thesis of Trevor Fiez presented on June 14, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Trevor Fiez, Author

ACKNOWLEDGEMENTS

.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

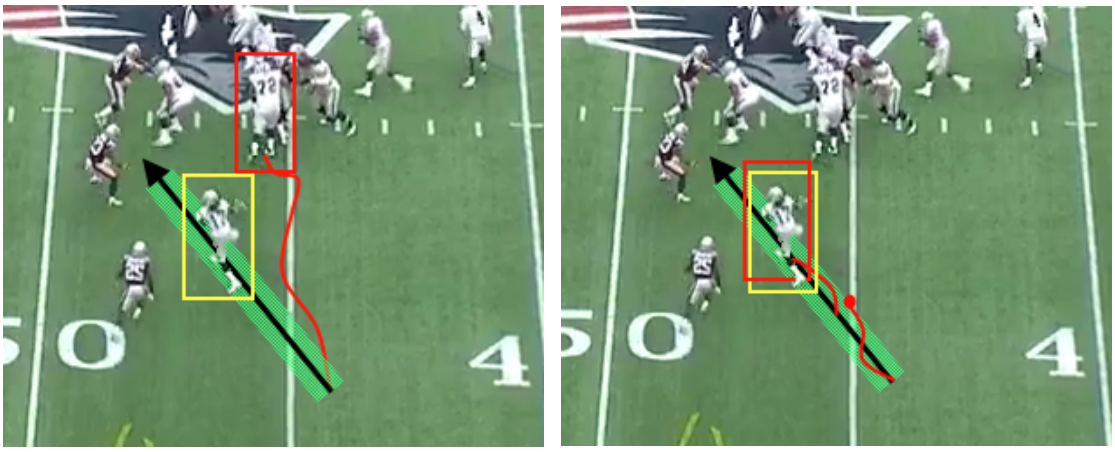
LIST OF ALGORITHMS

Chapter 1: Introduction

Deep learning methods have led to vast improvements in solutions to many problems in computer vision, including scene recognition, object detection, and activity recognition. These methods have all leveraged the availability of large offline datasets to train robust classifiers. However, deep learning methods have received much less attention for visual tracking [19, 15, 18, 26, 11]. Unlike in other vision areas, recent tracking literature continues to report that conventional algorithms are competitive with deep learning methods (e.g., [27]).

State-of-the-art conventional online trackers typically seek to learn a representation of the object being tracked [11, 19]. Many deep trackers share that objective. To learn a robust representation of objects, deep neural networks require a large and diverse set of training data. Unfortunately, most tracking datasets are quite small. In many cases, tracking datasets contain around 100 individual tracks. While this amounts to thousands of annotated images, there are still fewer than 100 objects in the entire dataset, which could cause the network to overfit. These sorts of small datasets make training deep neural networks to effectively model any object extremely difficult. One promising way to address the lack of visual tracking data for training has been to supplement the annotations with similar data sources (e.g., object-detection training data) or noisy variants of the original track annotations. In this paper, we continue this research direction by introducing, analyzing, and evaluating a new imitation learning framework for training deep trackers offline and then compare it to previous methods.

Recent deep trackers have considered various ways of using large repositories of training images available for object detection, i.e., for solving a related but not the same vision problem. For example, the CF2 tracker [16, 25] first pre-trains its deep network on an object classification dataset, as a starting point for a correlation filter based tracker. A major drawback of such a strategy is that it does not necessarily uncover the most informative features for tracking, which could hinder the tracker's performance. When training a network for object classification, all objects of the same class are treated equivalently. In a scene that contains many objects of the same type, the network



(a) DAGGER (b) DAGGER with Reset

Figure 1.1: (a) A tracker produces an erroneous track (red) compared to the ground truth (yellow). Consequently, in learning, DAGGER collects "useless" training data along the lost red track that are far from the ground truth data distribution. (b) In this case, our approach DAGGER^2 will reset the tracker when it strays too far from the ground truth. This allows DAGGER^2 to collect more meaningful training data, and consequently learn a better performing tracker.

activations of those objects are similar. This poses a problem for the tracker, because distinguishing between objects of the same class becomes challenging.

Other approaches train their deep trackers by augmenting the available tracking annotations with new object bounding boxes randomly sampled in the vicinity of the ground truth target location[19, 15, 18, 3]. These samples are treated independently of each other when training. Thus, the network loss is also based on independent decisions. However, we know that loss in visual tracking strongly depends on a sequence of previously made decisions. Trackers might go slightly off target and are still expected to perform well. Therefore, random sampling may be poorly correlated with true object trajectories. In turn, this may potentially introduce certain biases in learning, such as, e.g., finding only object poses and appearances with numerous samples relevant (e.g., trajectory parts when the object moves slowly) and ignoring other scarce samples (e.g., dramatic turns and speed-ups along the trajectory).

Thus, we find the aforementioned strategies common in training deep trackers are not directly tied to the end goal of online tracking performance. To address this issue,

we formulate online tracking as a sequential decision making problem, and consider a state-of-the-art algorithm for learning decision-making policies via imitation learning. Our key idea is to learn a policy – here, a deep tracker – that can imitate an expert performing the task of tracking, and in this way more explicitly tie our learning objective to the end goal. In our case, the "expert decisions" are available from the track annotations, which may come from either the original (small) training set or a larger set carefully augmented with new samples via probabilistic, expert-dependent sampling.

In addition to comparing traditional training methods, we start by considering a state-of-the-art imitation learning algorithm, called the dataset aggregation algorithm (DAGGER) [20], which has shown empirical success on a variety of AI control and planning problems. DAGGER iteratively learns a policy by minimizing discrepancy between decisions made by the expert and those made by a current policy estimate. As decisions have dependencies between them, DAGGER collects all sequences of decisions made by the current policy estimate after each learning iteration, and adds them to the training set. In this way, DAGGER iteratively trains the policy on all decisions induced by previously learned policies. DAGGER has strong theoretical guarantees relative to performance of the policy that best mimics the expert on training data.

We observe that DAGGER faces a significant limitation, however. In non-trivial tracking domains, even the best trackers will often get lost. In such situations, as shown in Fig. 1.1, DAGGER would continue collecting "useless" training data, far from the ground truth data distribution, as long as the tracker continues. To remedy this issue, we create an extension to DAGGER, called DAGGER with Reset (DAGGER^2). When learning, DAGGER^2 will reset the tracker at points along the trajectories when it strays too far from the ground truth. This allows DAGGER^2 to collect more meaningful training data for the next learning iteration. We show that DAGGER^2 maintains the same theoretical guarantees as DAGGER, but for a modified and intuitively attractive loss function.

We also compare whether more effective ways of supplementing the original ground truth data will also improve the tracker's performance. There are two reasons to believe a more effective sampling strategy, like that in DAGGER^2, will outperform traditional methods. First, the additional training data comes directly from decisions made by previously learned versions of the tracker. Second, important additional training data can be readily identified around the points when the tracker is declared as lost, and thus focus imitation learning on hard examples.

Our evaluation uses two deep tracking architectures, over three challenging tracking datasets, to compare the effectiveness of traditional training methodologies to those based off of DAGGER.

Our contributions include:

1. New formulation of DAGGER suitable for tracking.
2. Benchmarking of different training methods
3. New tracking dataset of NFL football videos with ground-truth annotations in every frame, complementing the existing benchmarks with new challenges prominent in the football domain.

In the following, Chapter 2 reviews closely related work, Chapter 3 specifies the trackers and datasets used for evaluations, Chapter 4 formulates our problem, reviews DAGGER, and specifies DAGGER^2, finally, Chapter 5 presents our results.

Chapter 2: Prior Work

Tracking is a long standing problem in computer vision, with extensive background literature. In this section, we focused our review on recent training strategies for deep tracking methods. Other related work on imitation learning is discussed in Cgapter 4.

Convolutional neural networks (CNN) have led to many advances in computer vision. The representational power of CNN's has led to major advances in every area of computer science. Tracking has been somewhat slower to adopt convolutional neural networks because of the difficuly training them for tracking. There have been numerous attempts to use CNNs for tracking with varying degrees of success. These difficulties arise from two main issues, the size of datasets and the time it takes to train a CNN.

There are two general ways of training a tracker, online and offline. Most previous deep trackers are trained online [15, 25, 26, 18, 2, 13]. They typically draw training samples around the detected target in every frame to train a classifier over features extracted by a CNN. However, training neural networks is typically slow, even with GPU acceleration, limiting the application of such trackers in real-world domains.

Offline models do not have to be trained while tracking so they can track objects more quickly. However, lack of data has been a persistent problem when training offline trackers. Many of these models have millions of parameters, which makes learning a robust function difficult without sufficient data.

The learning objective in most trackers are cast within the classification framework, where they are trained to classify whether two image patches are similar [16, 15, 18, 31, 13, 24, 25, 12, 26]. The approaches generally use a VGG-net [22], trained for object detection, to extract features that are then used for the comparison. Another type of approach has trackers regress to the target object's location[11]. Also, deep trackers can be trained to match patches in consecutive video frames [9, 29, 24]. However, all three types of learning objectives — namely, classification-, regression-, and matching-based — are typically formulated using loss functions that treat training data as independent samples. In addition to those approaches, we investigate using imitation learning for explicitly taking into account dependencies of sequences of ground truth annotations

along the target trajectories.

A separate class of deep trackers are aimed at particular domains, and thus trained to track a specific class of objects (e.g., pedestrians in upright poses) [5]. Our imitation learning algorithm can be used for both generic and domain-specific trackers.

Finally, when learning a tracker off-line, a number of approaches seek to first identify "corrupted" training samples (e.g., due to occlusion, or background clutter) using a combination of experts [30], another separate tracker [17], or distance comparisons [6], and then either discard or adaptively downweight them [3]. This reportedly leads to performance improvements. With a few exceptions [3], all of these approaches rely on heuristics, since they do not unify this step with training the tracker. In contrast, our DAGGER^2 manages training samples within the learning process. Importantly, not only does DAGGER^2 use *all* available training samples, but also augments this set with new noisy samples generated by the tracker in the learning iterations. According to the imitation-learning theory, this noisy augmentation of the initial training set guarantees an improved, robust learning of the tracker.

Chapter 3: Datasets and Trackers

To analyze the performance of different learning methods used to train visual trackers, we tested on a variety of different deep trackers and datasets. A tracker can be viewed as function that takes a video and an initial bounding box as input, and outputs a bounding box of the target for each frame in the video. For each frame of the video, the tracker passes some encoding of the previous sequence of decisions made by the tracker and the current frame into a CNN, whose output is then used to decide the next location of the track.

Visual tracking networks can be split into two groups: ones that learn a comparison function, and ones that try to perform some sort of regression on the tracked object's location. We will refer to the two types of tracking networks as a comparison function network and a regression network, respectively. For our tests we will train both types of networks. We will refer to the encoding of the previous detections as the track's history. Generally, the encoding used for tracking are crops of previous detections. The specific form of the previous detections used for input is unique to each tracker. In section below, we describe what is used for the input in more detail.

When tracking, comparison function networks select a set of potential candidate crops from the frame being searched and score each of them. The highest scoring candidate is then selected as the next location of the track. Regression networks, on the other hand, output coordinates of the tracked object's location. Instead of scoring many candidate for each frame, regression networks only need one search crop to detect the objects location. For each tracker, the search crop is slightly different. The search image in comparison function networks is a potential tight crop of the tracked object. In regression networks, the search image is a crop that contains the tracked object, and it does not require a tight bound. We tested our learning methods on two different deep trackers: MCDT and GOTURN. The MCDT learns a comparison function and the GOTURN tracker regresses to the location of the object.

Tracking datasets used to train these networks are composed of videos and a series of annotations for the track. Each dataset has a variety of different attributes that

distinguish them from others. Two attributes that we will use to distinguish them are the number of different object categories, and the amount of scale change the tracks goes through. Having fewer object categories makes tracking easier because the network is trained on a larger number of sample from the same object class so it can learn a better representation of that object. In our experiments, we tested our learning methods on three datasets containing different numbers of object classes. The GOTURN tracker also leverages the use of an object detection dataset when training. These datasets consist of images, the bounding box of the object, and the object's label. When training for tracking, the label is ignored.

For every given dataset we trained our set of trackers using different training methods. Overall, this gives a robust picture of how effective each training method affects a tracker's performance. In the sections below we go into more detail of both the datasets and trackers we used to analyze the training methods.

3.1 Datasets.

For evaluation, we use three datasets: ALOV300++ [23], KITTI [7], and our own new dataset of NFL football videos. Traditionally, single object tracking benchmarks have been compiled with many different object types. Benchmarks like CVPR100, ALOV300++, and the VOT2016 challenge all include many different classes. These datasets are useful when benchmarking how well the trackers generalize to objects outside the dataset. While it would be preferable to have a tracker that can track everything extremely well, in our experiments, we show how inadequate conventional single object trackers can be.

It is also reasonable to assume that many of these trackers would be used to track only a small set of specific objects. Therefore, we believe testing on a variety of datasets with different numbers of object classes is relevant to evaluating training methods. Furthermore, being able to fine tune a tracker to a small set of classes can be a strength. For example, imagine tracking a human walking whose legs are not visible for part of the video. The person is tracked correctly in the beginning of the video. However, their legs become occluded part way through the video and only the person's head and torso are now tracked. When their legs become visible again it is conceivable that if a tracker did not recognize object types while it was tracking (in this case a person), it would continue to track just the person's torso and head. If the tracker recognizes object types, then

it might be able to recover from the partial occlusion and continue to track the entire person as they move past the occluding object. In our tests we used three datasets with different numbers of object classes to explore this idea.

Scale change is the other significant difference we saw when comparing datasets. Many current state-of-the-art trackers do not adjust their object size for the entire track [16, 15]. Therefore, datasets that do not contain large scale changes are advantageous for those trackers. All three of our datasets have varying amounts of scale changes during the course of the video, as a result of the objects motion and the relative movement of the scene induced by the camera. The easiest of these datasets for scale changes is the NFL Football player dataset. The object bounding box does not change for the entire duration of a track. Videos for the dataset are filmed from a set camera with relatively little motion or zoom. On the other hand, the KITTI dataset contains significant amounts of scale changes. The videos used for the dataset are filmed from the top of a car driving down busy streets. The objects come closer, and while the camera does not zoom, the background is almost constantly moving. The ALOV300++ dataset contains many videos with varying amounts of scale change. In most of the samples, the camera is set and does not zoom often which decreases the occurrences of large scale changes.

Another measure of difficulty for trackers is occlusion. However, it is much harder to control and measure occlusion. In most datasets the object size being tracked includes parts hidden by other objects. If a person walked behind a car, the bounding box, in some datasets includes the person's legs even though they are not visible. To have some sort of measure of occlusion we would need bounding boxes explicitly showing what is visible, which most datasets do not provide. All three of our datasets include tracks that contain occlusion. However, we do not have a measure on how much occlusion occurs.

ALOV300++ consists of 314 video sequences, totaling 89364 frames, which present a host of challenges, including: low contrasts, confusion with similar objects, clutter, occlusion, camera zoom, and severe shape changes [23]. The ground truth in ALOV300++ is annotated every five frames on average, by a bounding box around the target[1], through a combination of manual annotation and linear interpolation. The ground truth bounding box in the first frame is specified to the trackers. We conduct 3-fold cross-validation on

[1]We use this data for DAGGER-style training by simply ignoring frames that did not have annotations, which amounts to training on a downsampled version of the videos. Since annotations were quite frequent, this did not seem to impact performance.

ALOV300++, where 2/3 of videos are used for training, and 1/3 for testing.

KITTI has 21 training and 29 test sequences, with each video containing many individual tracks [7]. Each video contains ground truth for 8 classes annotated in every frame. The videos in KITTI are captured from a moving car, so the dataset includes dramatic scale changes and lighting variations. While KITTI does include a test set, we are unable to use it because single object tracking requires that the first crop be given to the tracker, which is not included in the set. Therefore, in this paper, we perform 3-fold cross-validation only on the 21 original training sequences, where in each fold 2/3 of the videos are used for training, and 1/3 used for testing (ignoring their annotations in testing).

NFL Football Players consists of 18 videos, totaling 22510 annotated frames with bounding boxes around target players. We have also compiled this dataset from NFL football videos, with the goal of tracking individual football players. Our NFL dataset On average each sequence has 118 frames, which is sufficient to show long trajectories in this particular domain with extremely quick motions. The football game location and play type change with every video. Our dataset complements the above benchmarks by presenting additional challenges, including: sharp trajectory changes, confusion with similar-looking players on the same team, frequent and long-term occlusions, and players moving through a crowd. The dataset is available at: `https://github.com/trevorfiez/Football-Player-Tracking-Dataset`.

Object detection datasets have also been leveraged to help train trackers. One advantage of these datasets is that they contain many different object classes, which might be lacking in tracking datasets. However, a sample from an object detection dataset consists of a image, a bounding box of the target object, and a label. We know networks used for tracking use multiple frames as input. To rectify this, a tracker will use the same image for both the previous and current frame. In this way, we can use object detection datasets to supplement our existing training data.

3.2 Trackers

We trained two trackers in our experiments: the regression-based GOTURN [11], and our own classification-based multi-channel deep tracker (MCDT). This allows us to test different training methodologies within both regression and classification networks. We

Figure 3.1: Architecture for the multi-channel deep tracker. The history frames are concatenated with a candidate and then are fed into the network. The blue layers denote a convolution and the purple denote a fully connected layer

note that the classification-based tracker is only evaluated on the NFL dataset due to the fact that it was not designed to handle dramatic scale changes, which are present in KITTI and ALOV300.

GOTURN [11] uses a Siamese CNN to regress to a bounding box of the target object. The input consists of a crop from the previous frame and a crop of the current frame. The previous frame is cropped around the object detection, and the current frame is cropped around the previously detected location with a size and shape proportional to the previous detection. The Siamese CNN is simultaneously trained off of the ImageNet object detection dataset, and a tracking dataset. Their training augments the ground truth, in both cases, by shifting and scaling the available training samples according to a Laplacian distribution. For our tests on the ALOV300++ dataset, we randomly initialize the GOTURN network. Similar to pretraining a CNN on ImageNet for image recognition, when testing on the NFL Player and KITTI datasets we pretrain the network on the ALOV300++. We will denote the pretrained GOTURN network as GOTURN-PRE and the randomly initialized network as GOTURN.

MCDT – our multi-channel deep tracker estimates if a potential object bounding box is a good fit given a set of previous detections. Recent work [24] considers analogous formulations. However, our approach has a few important differences from recent work.

Firstly, MCDT does not use a Siamese CNN. Instead, we stack the dimensions of each crop together, as you saw in Fig. 3.1. Secondly, most recent deep trackers only use the previous frame as state encoding while MCDT looks at more [19, 11]. Our network consists of a typical CNN configuration with 7 layers. The first part of the CNN consists of 3 convolutional layers, with filter numbers and sizes of 96x5x5, 256x3x3, and 384x3x3, respectively. Each convolutional layer uses a rectified linear unit for non-linearity. Between the first and second, and the second and third layer, the network also has two 2x2 max-pooling layers. The fully connected layers have 4096 and 2 hidden units, respectively. As input, MCDT uses the bounding box detected at 20, 10, and 1 frames before the current frame.

Chapter 4: Learning Approaches

We consider a supervised learning setting, where the goal is to learn a tracker from training data $\{(x, y)\}$ drawn from a distribution $P(x, y)$, where x is a video, and y is a ground-truth sequence of bounding boxes around the target object in every frame. A tracker, π, is a function that takes a video, x, and an initial bounding box, y_1, as input, and outputs a predicted track, $\hat{y}$ (one bounding box per frame). In this paper, we focus on online trackers, which predict bounding boxes in the video in a single pass from start to finish.

The state of an online tracker up to frame t, $s_t = (x_{1:t}, \hat{y}_{1:t-1})$, encodes all video frames up to time t, $x_{1:t}$, and all previous predictions, $\hat{y}_{1:t-1}$. An online tracker is a function $\pi(s_t) = \hat{y}_t$ from states to predictions. Most online trackers do not use all information in s_t, and rather make predictions based on a short-term history of video frames and previous predictions.

When training a neural network, the designer has a variety of options including the architecture, dataset, and method to update weights. Another factor that has proven vital to certain challenges is how the input data are sampled in order to expand the dataset. In many image classification problems this has included flipping and randomly cropping images. Expanding the dataset with different forms of sampling is also important when training CNNs for visual tracking. However, since tracking is a series of sequential decisions, the sampling methods used have to take into consideration a series of annotations associated with a track. Algorithm 1 is the pseudocode used to train a neural network for visual tracking. Similar to training any network, the training algorithm samples batches from the training data. Each batch will be augmented and sampled with a method AUGMENT and will then be fed into the network. It should be noted that AUGMENT can also return the input without any changes. For our study, we will examine three different methods of training visual trackers: random sampling, naive imitation learning, and DAGGER with Reset which are described in more detail below.

Algorithm 1 LEARN

Input: TRN $= \{(x, y)\}$ – set of annotated training videos
π – network used in tracker
N – number of batches to train network
AUGMENT – random transformation to augment training data (optional)

for $i = 1$ to N **do**
  // Sample from training set
  $\{(x_i, y_i)\} = SAMPLE(\text{TRN})$
  // Augment the training batch
  $BATCH = \text{AUGMENT}(\{(x_i, y_i)\})$
  // Train the network on the batch
  TRAIN_NETWORK$(\pi, BATCH)$
end for
return π

4.1 Random Sampling Training Methods

Random sampling techniques for data augmentation have been used for almost every computer vision problem. Due to the sequential nature of tracking, random sampling must be done carefully. If a change alters the previous detection crop too much, it could hypothetically make every single candidate for the next location a bad one. For example, if AUGMENT shifted the ground truth bounding box so that it did not contain the target at all, the state encoding would not contain any relevant information.

One way training methods have ignored this dependency is by augmenting the search crop of the image and using ground truth detections for the state encoding of the network. In this way, some traditional random sampling techniques can be used to augment data. For a classification network, like MCDT, a training batch is generated by sampling locations around the ground truth. These are then labeled correct if the overlap is greater than some threshold, and incorrect if it is below. The network then learns to classify which crops have a high overlap with the ground truth target. An analagous version of random sampling in the search frame of a regression network, is to shift and scale the search image. The target bounding box still perfectly encloses the tracked object, but it looks slightly different than the original image. Held et al.(2016) proposed to randomly augment the search crop image in regression networks by shifting and scaling the search bounding box crop according to a laplacian distribution. When we are testing

the GOTURN tracker we will augment samples using the same method. For the MCDT tracker, we will uniformly shift the ground truth crop but not scale the ground truth.

Both of these methods use thresholds to keep samples in a reasonable range. When using a classification network we also require that all samples used to train the network have an overlap above a threshold of 30%. For regression networks we restrict the scale to change at most by 30% and the shift to move no greater than 50% of the object size, which is used by GOTURN[11].

While traditional random augmentation methods have effectively increased the number of samples in a dataset, they neglect the sequential nature of visual tracking. In previous methods, the encoding of the track's history is always generated from the ground truth. However, we know that even the best trackers will stray off target in non-trivial tracking domains. To further investigate how random augmentation affects the performance of a trained tracker, we will also randomly augment the state encoding by shifting the image crop of the previously detected target.

We introduce a method for random augmentation on the state encoding to further compare against DAGGER[2]. When augmenting the trackers history, we will also shift and scale the ground truth detections of the previous targets. To ensure that this does not violate the condition that the state encoding be representative of the track, we require the ratio of intersection over overlap between the augmented bounding box and the ground truth be above some threshold. In all of our tests we set that threshold to be 0.7. These should mimic the slight mistakes that trackers make, and make the data we are training our classifiers on more realistic. They are also used in tandem with the random augmentations used in the search frame.

4.2 Learning to Track via Imitation

For video x with ground truth y, an online tracker incurs the loss of a prediction $\hat{y}$, $l(x, y, \hat{y}) = \sum_{t=1}^{|x|} \Delta(y_t, \hat{y}_t)$, where Δ is a distance between two bounding boxes. For example, $\Delta(y_t, \hat{y}_t)$ may measure the distance between the centers of y_t and $\hat{y}_t$, or some other measure of their overlap. The expected loss of a tracker π is denoted by $L(\pi) = E\left[l(x, y, \hat{y}\right]$, where $(x, y) \sim P$ and $\hat{y}$ is produced by π. Our goal is to learn an online tracker using training data D that achieves a minimum expected loss.

An important aspect of online tracking is that a tracker's decision at frame t influences

its decisions in the future. In particular, the prediction $\pi(s_t) = \hat{y}_t$ for the input state s_t is conditioned on previous predictions of π. Therefore, online tracking is an instance of sequential decision making, in contrast to standard classification and regression problems where all decision are independent of one another. In general, learning for sequential decision making is more difficult than independent, identically distributed learning, since ideally learning algorithms must account for the dependence between learned decisions.

A popular learning approach for sequential decision making is imitation learning, which has been studied in many contexts including navigation [21], robotics [1], and recently for playing Atari games from video input [8]. The main idea is to attempt to learn to imitate an expert or oracle, that is available during training time to perform the desired task. In the context of tracking, obtaining an expert tracker for the training data is straightforward since we have the ground-truth track for each example. In particular, the expert tracker, π^*, always returns the ground-truth bounding box for any video frame, and hence $L(\pi^*) = 0$.

The simplest approach to imitation learning, which we will call *Naive Imitation Learning (NIL)*, collects a set of training bounding boxes along the trajectories produced by the expert tracker. For example, given annotated training video (x, y), we would produce one training example per frame of the form (s_t^*, y_t), where $s_t^* = (x_{1:t}, y_{1:t-1})$ is the state at frame t of the perfect expert tracker. The resulting training set D over all videos specifies a standard regression learning problem, with regression inputs s_t^* and target outputs y_t. NIL then proceeds to learn a tracker π based on D using a standard learning algorithm (e.g., deep learning).

While the NIL approach can sometimes produce satisfactory results, it has some well known deficiencies, which can limit its performance [20]. The fundamental problem is that the learned tracker π is trained on the state distribution of the expert π^*, but tested and evaluated on the distribution of states that π generates. Often, even small inaccuracies in π with respect to π^* can cause these distributions to be quite different. To see this, suppose that at frame t, π makes a prediction $\hat{y}_t$ that differs from the expert prediction y_t. This imperfect prediction is then used as part of the input to π at frame $t + 1$, that is $\hat{y}_t$ is added to s_{t+1}. This means that s_{t+1} will differ from the expert generated state s_{t+1}^*. Since π was trained on only expert generated states, there is a chance that it will not generalize well to s_{t+1}, and hence make a prediction for $t + 1$ that is incorrect by an even wider margin. This misprediction will then be incorporated into s_{t+2}, which may differ

even more from the expert state s^*_{t+2}. This type of *error propagation* behavior of NIL has been commonly observed in the imitation learning literature and can significantly degrade performance.

Ideally, we would like to train a tracker to perform well on its own state distribution. Indeed, it is known [20] that if a tracker π does have low prediction error with respect to the expert on its own state distribution, then π will perform nearly as well as the expert. However, this leads to an apparent chicken-and-egg problem. How can we train a tracker π on its own state distribution without first having π? In the next section, we describe a recent answer to this question for general imitation learning along with important modifications that are needed in the context of imitation learning for tracking.

4.3 DAGGER with Reset

DAGGER [20] is a state-of-the-art imitation learning algorithm that has strong theoretical guarantees in addition to a number of empirical successes [20, 8, 4, 14]. Here we describe DAGGER in the context of tracking, though it is more generally applicable to any sequential decision making problem. The main idea behind DAGGER is to start with expert generated data as in NIL, but to then iteratively generate and aggregate frame-level training data from a sequence of learned trackers. This provides training data from states that learned trackers tend to visit in addition to states that the expert visits. The iterative process converges to learning a tracker that performs well on its own state distribution.

Algorithm 2 gives pseudo-code that implements DAGGER when run with the input parameter set to $\delta = \infty$. DAGGER assumes a set of annotated training videos and a learning algorithm Learn that can learn a tracker given a set of frame-level training examples. Each of the N main iterations of DAGGER produce a learned tracker π_i based on data collected from all previous iterations. The initial tracker π_1 is set equal to the expert tracker π^* that always returns the ground truth. Each iteration then uses the current tracker π_i to generate tracks for the training video (line 6), noting that when the input parameter $\delta \neq \infty$ the tracks may be different from those produced by π_i (see below). Each such track goes through a sequence of tracking states and we collect each such state s and put it in the frame-level training data labeled by the expert prediction $\pi^*(s)$ (line 9). At the end of the iteration we learn a new policy π_{i+1} from the frame-level

Algorithm 2 DAGGER^2 (setting $\delta = \infty$ results in standard DAGGER)

Input: TRN $= \{(x, y)\}$ – set of annotated training videos
δ – reset threshold
LEARN – algorithm for learning a tracker from a frame-level training set (e.g. a deep network)

```
1: D = ∅ // Initialize frame-level training set
2: π_1 = π* // Initialize to expert tracker that returns ground truth
3: for i = 1 to N do
4:    for (x, y) ∈ TRN do
5:       // Run tracker π_i on x but reset to ground truth when error is greater than δ
6:       T = GenerateTrackWithReset(π_i, (x, y), δ)
7:       for each state s in T do
8:          // Add new frame-level example for s to data
9:          D = D ∪ {(s, π*(s))}
10:      end for
11:   end for
12:   // Learn next policy based on aggregated data
13:   π_{i+1} = Learn(D)
14: end for
15:
16: return best π_i on validation test
```

training data. After N iterations DAGGER returns the learned policy π_i (for $i > 1$) that achieves the best performance on validation data. In practice, it is common to simply return the final policy π_N, which is the policy that was trained with the largest amount of training data. In practice, significant performance improvements over NIL are observed even with just a few iterations.

DAGGER has been primarily evaluated on common AI problems (e.g., game playing), which are qualitatively quite different from tracking and tracking presents at least one critical challenge for DAGGER. The main difficulty arises at line 6, where the current learned tracker π_i is used to generate a track for the video. For challenging tracking problems, π_i will tend to "get lost", i.e., stray far away from the correct object, as illustrated in Figure 1.1a. Let us consider how this can adversely impact DAGGER. DAGGER will include such "lost" state s in the training data with the ground truth label. For example, in Figure 1.1a, DAGGER would create an example for the tracking state of the lost red bounding box with a ground truth label corresponding to the yellow bounding box. Such training examples are nearly impossible for the learning algorithm,

since the example is effectively telling the tracker that it should instantaneously jump to the location of the true target object from the current lost location. At best, such examples will be useless for learning, and at worst the examples will confuse the learner and degrade performance. For many trackers, such data are not even useable, since the trackers have a bound on how far away they will move the bounding box from frame to frame. In any of these cases, DAGGER effectively throws away a large portion of the labeled training videos. In particular, all data from frames that occur after a learned tracker are lost.

The above problem with DAGGER for tracking motivates our extension that we call DAGGER with Reset (DAGGER^2). In particular, we modify DAGGER so that when it uses π_i to generated tracks (line 6), the tracker is reset to the ground truth whenever its error becomes more than a user specified parameter δ, and then tracking continues. This is implemented via the procedure GenerateTrackWithReset. The frame-level training examples are then generated for the tracking states along the trajectory with reset. Figure 1.1b shows an example of this modified procedure where the green band indicates the region within δ of the ground truth and the red track indicates the result of running the learned tracker with reset.

DAGGER^2 spans the range of imitation learning algorithms from NIL to regular DAGGER. When $\delta = 0$ the learned trackers used to generate data at each iteration will almost always be reset at each frame, meaning that they will generate data along only the ground truth trajectories, which exactly matches the behavior of NIL. As the value of δ increases, the generated training data will be able to depart from the ground truth and be more reflective of learned policies, which is the key motivation for DAGGER. However, as δ grows beyond an acceptable error tolerance for an application, the generated training data will be along lower quality trajectories, and hence less useful or even detrimental to learning, similar to DAGGER when used for tracking. When $\delta = \infty$, or a finite value beyond the largest possible error, the tracker will never be reset and DAGGER^2 behaves exactly like DAGGER.

We now consider what loss function DAGGER^2 is optimizing. The original DAGGER algorithm provides theoretical guarantees with respect to the expected tracking loss $L(\pi)$ of the learned tracker. While the theoretical results for DAGGER are beyond the scope of this paper, at a high-level, DAGGER will learn a tracker that performs close to the performance of the best tracker that could be learned. In practice, however, for

challenging tracking domains, there will be no tracker within the class of trackers being considered that performs well in terms of $L(\pi)$. All trackers will eventually get lost during normal operation and after that accumulate significant loss.

In actual operation, when a tracker gets lost, a human must reset the tracker in order to complete a track. Thus, we specify a more meaningful notion of loss for this scenario, called the *reset loss*. Specifically, for a video x with ground truth y, the δ-reset loss of π, $l_\delta(x, y, \pi)$, is the accumulated loss in each frame when π is used to track but is reset when the error goes beyond δ, similar to how a human would reset the tracker in actual operation. Accordingly, the expected reset loss is given by $L_\delta(\pi) = E\left[l(x, y, \pi)\right]$. It is straightforward to see that the theoretical results for DAGGER carry over to DAGGER^2, but for the δ-reset loss L_δ. Thus, DAGGER^2 will converge to a tracker that approximately achieves the minimum possible L_δ.

Minimizing $L_\delta(\pi)$ is arguably a more relevant learning goal for many applications where it is expected that trackers will be reset/corrected when they get lost.

Chapter 5: Results

Our focus is how training methods affect a single tracker across the same dataset. We trained the trackers using naive imitation learning (NIL), DAGGER, DAGGER^2, and random augmentations. Since some of our trackers have tunable parameters, we also provide some assessment of how performance varies with different δ values for DAGGER^2, including evaluating the original DAGGER algorithm. Unless otherwise noted, our default threshold for DAGGER^2 is $\delta = 0.15$ for both GOTURN and MCDT. In addition, we report results for the state-of-the-art deep tracker CF2 as a reference point [16]. This tracker has been demonstrated to be among the best performers for single object tracking on the CVPR-100 dataset [28]. By design, CF2 cannot be fine-tuned to a specific dataset, which allows us to conduct comparison on a range of domains represented by the aforementioned three datasets.

5.1 Quantitative Results

We measure performance using two metrics. First, our tables will report average precision, which measures the percentage of all video frames across a dataset for which the predicted bounding box has an overlap with ground truth of at least 0.5. One issue with this single number is that it can be biased by video length. In particular, if a tracker happens to fail and get lost early in a long video, it hurts performance more than failing early in a shorter video. To counteract this bias we consider another performance measure: success versus overlap. Given an overlap threshold, the success rate is the percentage of videos in a given test set for which the average per-frame overlap was at least the threshold value. In fig. 5.1 you can see a plot of the success rate over the threshold.

Tab. 5.1 shows our precision results for the NFL videos. First considering the MCDT, we see that DAGGER^2 is able to significantly improve performance over the more conventional imitation learning approach, NIL, which demonstrates the utility of the iterative DAGGER with resets approach. We also see that random augmentation increases the performance of over traditional training as well. For GOTURN, we see that

Tracker	Tracker Precision
CF2	0.73
MCDT + NIL	0.736
MCDT + DAGGER^2	0.862
MCDT + Random	0.844
GOTURN-PRE	0.32
GOTURN-PRE + NIL	0.925
GOTURN-PRE + DAGGER	0.802
GOTURN-PRE + DAGGER^2	0.928
GOTURN-PRE + Random	0.915

Table 5.1: Tracker precision on our NFL football dataset. GOTURN-PRE denotes GOTURN [11] initialized on the ALOV300 dataset. NIL stands for naive imitation learning. Using DAGGER^2 to selectively add examples increases precision of both GOTURN and our own MCDT tracker compared to their corresponding baselines trained without DAGGER^2. Both GOTURN and our MCDT trained with DAGGER^2 outperform CF2 [16] by 4.1% and 18%, respectively.

the pre-trained network GOTURN-PRE performs quite poorly and improves significantly when trained via NIL. In this case DAGGER^2 performs similarly to NIL, which indicates that for this combination of tracker and dataset, the additional training data provided by the DAGGER-style iterations did not help improve generalization performance. The graphs for the NFL data in Figure 2a agree with the observations in the table.

When compared to augmenting the history randomly, we see slight improvements similar to our DAGGER^2 approach. However, when we train the GOTURN tracker using DAGGER, we see a large decrease in performance relative to NIL, because some of the training samples generated do not contain the ground truth track. On the other hand, DAGGER^2 fundamentally is trying add more realistic data to our training set. In the MCDT tracker, DAGGER^2 does this quite well. However, in GOTURN we only see a slight improvement. Since the naive method did not work very well on the MCDT tracker, the data added by DAGGER^2 will better fit the true state distribution. This data helps the tracker focus on more relevant targets to perform better. However as a tracker becomes better, the difference between actual realistic state encodings and ground truth state encodings becomes smaller. In the GOTURN case, during each iteration of DAGGER^2 most of the data added to the training set will be extremely similar to the

Tracker	Tracker Precision
CF2	0.126
GOTURN-PRE	0.296
GOTURN-PRE + NIL	0.355
GOTURN-PRE + DAGGER^2 ($\delta = 0.15$)	0.462
GOTURN-PRE + DAGGER^2 ($\delta = 0.5$)	0.453
GOTURN-PRE + DAGGER^2 ($\delta = 0.75$)	0.410
GOTURN-PRE + DAGGER	0.435
GOTURN-PRE + Random	0.445

Table 5.2: Tracker precision on the KITTI dataset. The acronyms are explained in the caption on Tab. 5.1. DAGGER^2 improves the performance of GOTURN-PRE and GOTURN-PRE trained with NIL.

data already within it. The GOTURN tracker worked well using just NIL. So it is to be expected that DAGGER^2 would only result in relatively minute improvements.

Table 5.2 shows our precision results for the KITTI dataset. The only tracker we test on this dataset is GOTURN, since our MCDT does not change scale which is vital for this dataset. From our tests, we see that NIL significantly improves over GOTURN-PRE and that DAGGER^2 (with the default δ=0.15) significantly improves over NIL. This again shows the utility of DAGGER-style training. However, we also see that just using random perturbations, denoted as GOTURN-PRE + Random, produces similar increases in performance to DAGGER^2. Part of the reason for the discrepancy between KITTI and the Football Player dataset is the level of difficulty between the two datasets. For one, the NFL Football Dataset only contains 1 class, whereas KITTI contains 8. The NFL football dataset is also filmed from a set camera and the bounding box of the players does not change throughout the track. Whereas, the KITTI dataset is filmed from a top of a car so it contains large variations in scale and movement. In addition to those challenges the background in the KITTI videos change dramatically throughout the duration of the track, whereas the NFL Football dataset background is always the field. Since NIL learning is insufficient to model the real distribution of the previous states of a track, DAGGER^2 and random augmentation improve the performance of GOTURN significantly. DAGGER^2 might perform slightly better than randomly augmenting the previous states because the training set generated by that method better reflects the actual performance

Tracker	Tracker Precision
CF2	0.233
GOTURN NIL	0.593
GOTURN DAGGER	0.263
GOTURN DAGGER^2	0.603
GOTURN + Random	0.623

Table 5.3: Tracker precision on the ALOV300 dataset. The acronyms are explained in the caption on Tab. 5.1. DAGGER^2 improves the performance of GOTURN trained with NIL.

of the tracker. The graphs in Figure 2b also agree with these conclusions.

On the KITTI dataset, we also evaluated how different δ values affect performance of DAGGER^2 including $\delta = 1$, which corresponds to the original DAGGER algorithm. From Tab. 5.2 we see performance degrades as δ increases from 0.15. However, interestingly pure DAGGER performs better than the δ values 0.5 and 0.75 when measuring the performance with tracker precision. When measuring the performance by average sequence, shown in Figure 2b, we see that performances degrades from $\delta = 0.15$ to $\delta = 1$. The reason for this is presently unclear.

Tab. 5.3 shows our results on the ALOV300++ dataset. Here we see that DAGGER^2 improves slightly over NIL by 2%. Still though, random perturbation on the history produces similar increases on the trackers performance. We can also see that when DAGGER is directly applied to train a tracker, it greatly decreases the trackers performance. ALOV300++ contains hundreds of different tracks, unlike KITTI or the NFL football player dataset, which contain 8 tracks or fewer. This may reduce the effectiveness of DAGGER^2 and random augmentation. If the training set does not reflect the actual targets it is tracking, it would make sense that it would not improve the tracker significantly. DAGGER^2 also might not generalize as well to unknown objects. Part of the reason random augmentation might have worked better in this dataset is because the policy learned by the tracker does not reflect how the tracker performs on the test set. In this sense, the performance is random. Therefore, by augmenting the state encoding using random augmentation we are adding more realistic state distributions to our training set, which is DAGGER^2's objective. In turn, this makes random augmentation perform

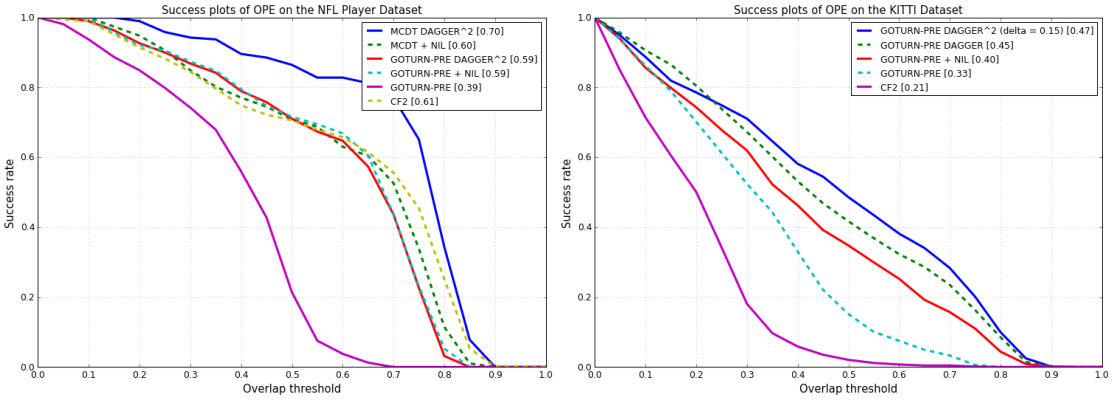
better than DAGGER^2.

The CF2 tracker performed the worst on every single benchmark we tested. While it performs well relative to other state of the art trackers on the CVPR100 dataset, it cannot be fine tuned to more specific datasets. It contains no method to adjust the targets scale, which is vital to be successful in both the KITTI and ALOV300++ dataset. This highlights the importance of being able to finetune a tracker on a specific set. Current state-of-the-art trackers are not good enough to generalize to any viewpoint or use case.

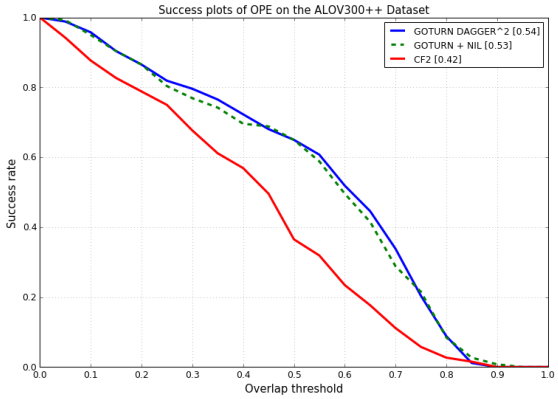
5.2 Qualitative Results

Overall, using either Dagger^2 or good forms of random augmentation will increase the trackers accuracy on a dataset. An example of a track that was improved after training with DAGGER^2 and random augmentation is shown in fig. 5.2.

However, there are still many failure cases for visual trackers which different training methods cannot overcome. There are limits to the representation power of all of the networks we tested. There are tracks that neither tracker is well suited to track well. Since GOTURN uses a state encoding of just the previous frame, all tracks that have some sort of occlusion are quite challenging to track, as you can see in figure 5.4. Furthermore, the state encoding also makes it challenging for GOTURN to track objects that look extremely similar. This failure case can be seen in figures 5.3. In addition to those failure cases, visual trackers fail on a surprising number of tracks that are fully visible throughout the video, but might get stuck on a feature in the background, or not have a good representation of the target. These sort of errors seem to be due to insufficient training data and can be seen in figure 5.5.

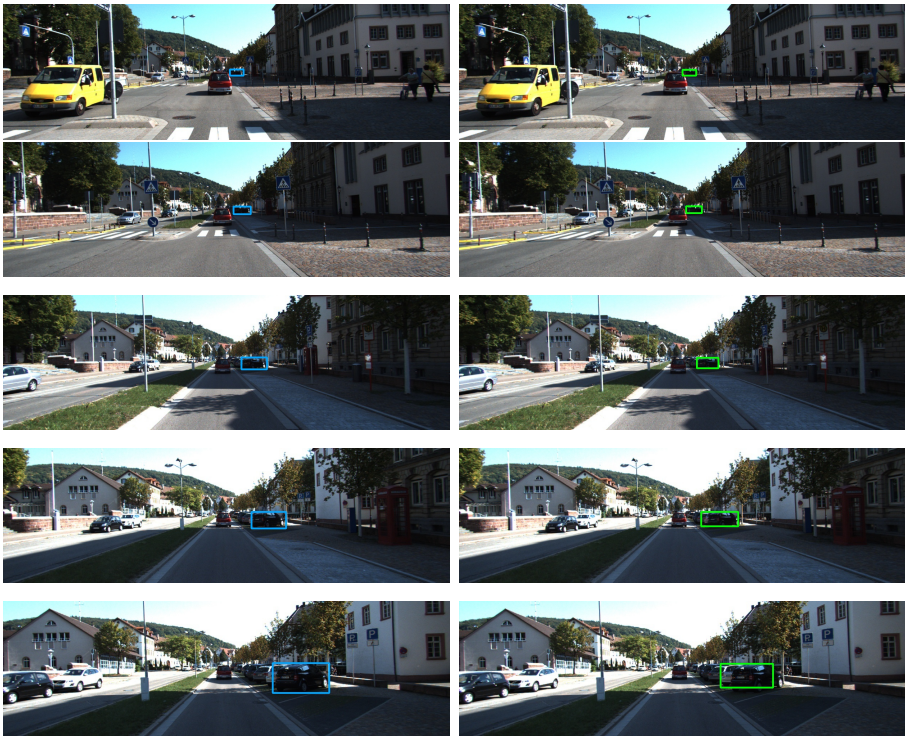


(a) NFL (b) KITTI



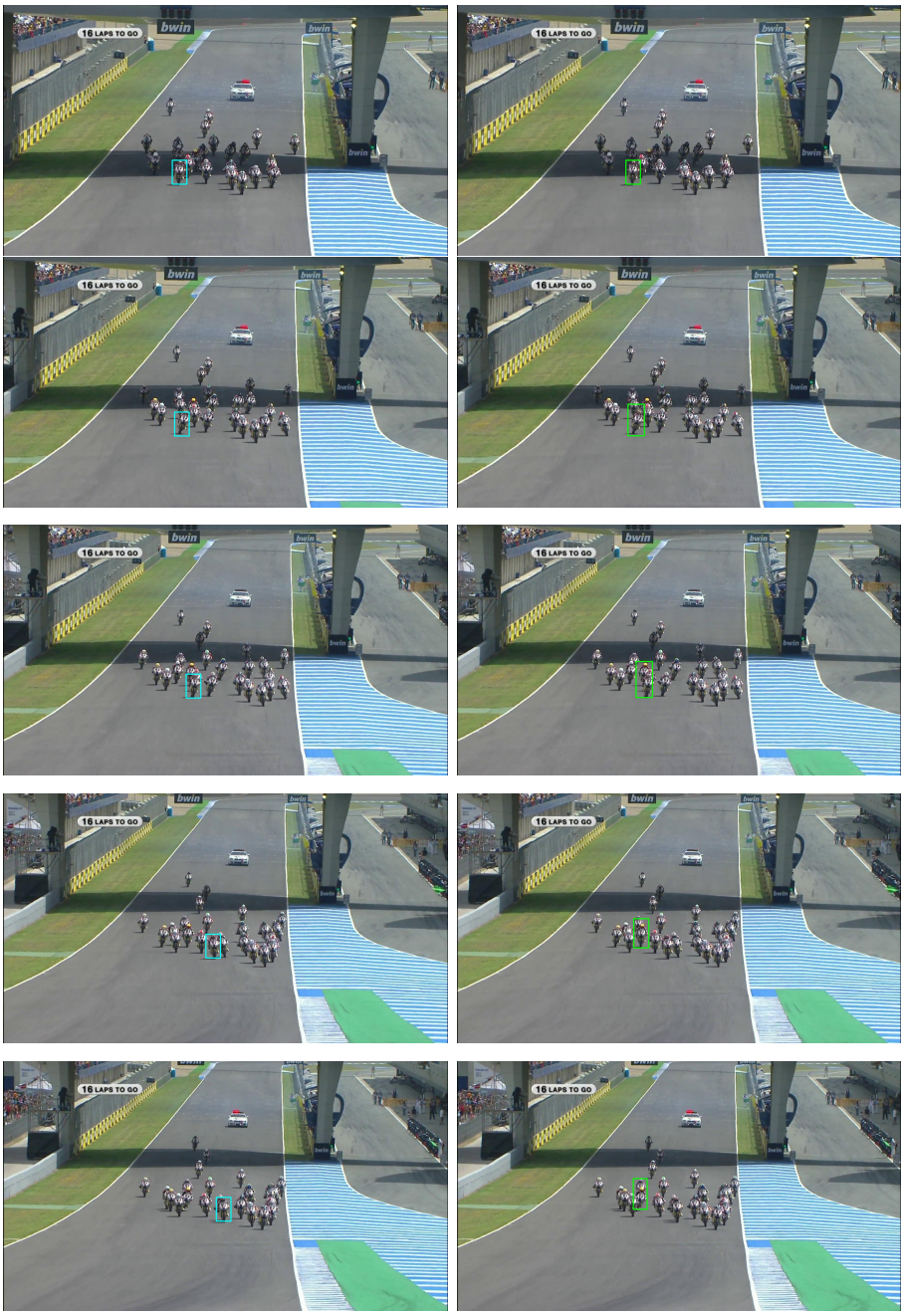
(b) ALOV300

Figure 5.1: Per video success rate versus overlap threshold. For each overlap threshold the graphs show the percentage of videos for which the average per-frame overlap was greater than the threshold.



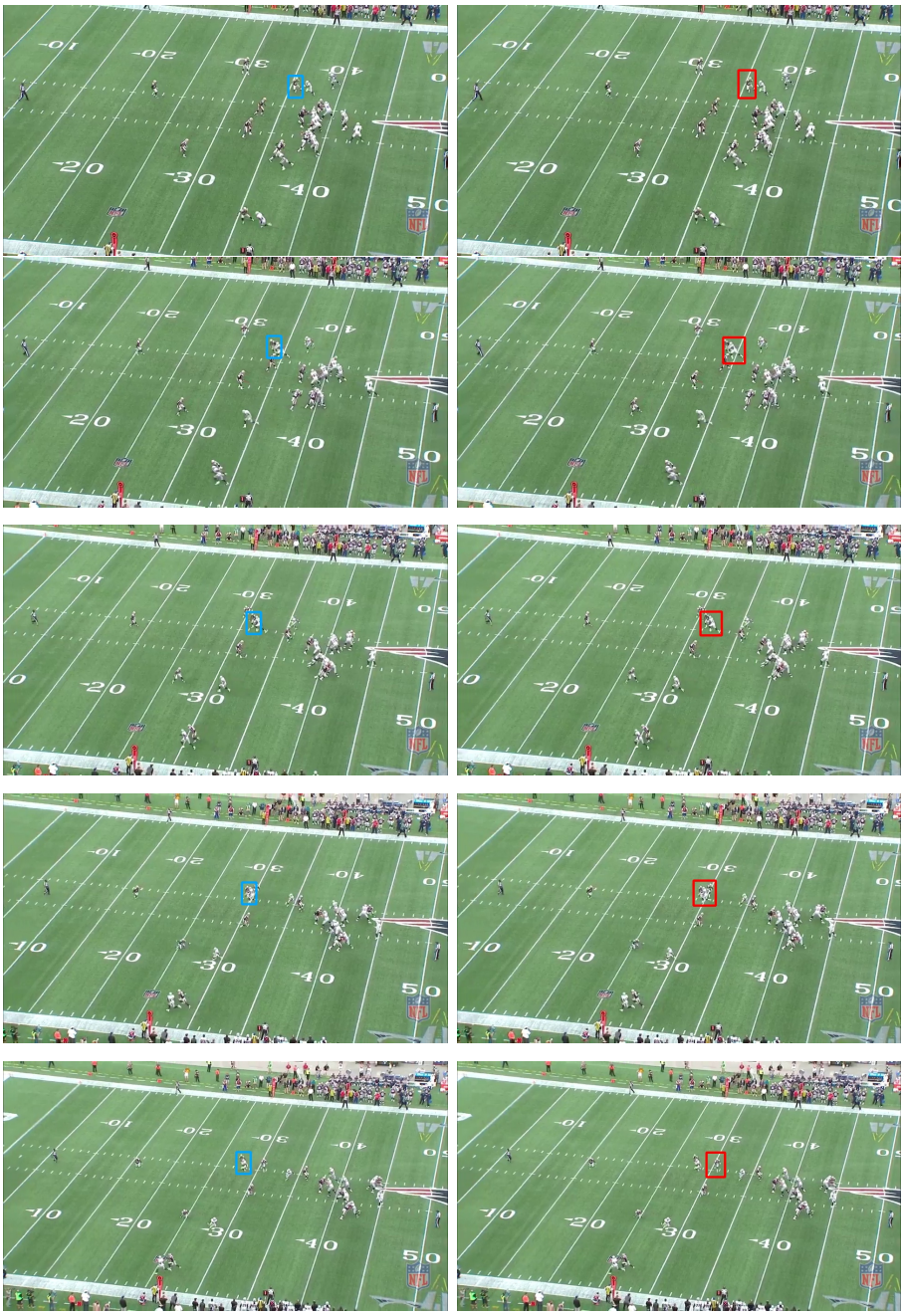
(a) Ground Truth (b) DAGGER^2

Figure 5.2: Comparison of ground truth of track, the left column, from KITTI dataset and the track generated using a a tracker trained using DAGGER^2, which is on the right column.



(a) Ground Truth (b) DAGGER[2]

Figure 5.3: An example where the GOTURN tracker's state encoding is insufficient to track an object. The tracker loses the target among many objects that appear similar.



(a) Ground Truth (b) Random Augmentation

Figure 5.4: An example where a GOTURN tracker trained with random augmentation switches the target, shown in row 4, because the target becomes ambiguous during tracking.

(a) Ground Truth (b) DAGGER[2]

Figure 5.5: An example where even in the simple case of a person walking the GOTURN tracker loses the target, and tracks another person.

Chapter 6: Conclusion

Through our work we have tested the limits of current state-of-the-art visual trackers. We have evaluated a variety of different datasets and training methods. Unfortunately we have found that visual trackers are still far from matching human performance. There have been many improvements made in deep tracking literature but as we have shown, even simple tracks with no occlusion can still be extremely difficult to track using competitive state-of-the-art trackers.

This partly comes from the difficulty inherent within visual tracking. A perfect tracker will need to encompass many different, challenging components, some of which include foreground detection, object recognition, and object permanence. Each of these components are needed to fix the error cases we saw in our tests.

In our tests, trackers would commonly get fixated on something in the background and then lose the tracked object. One of the reasons for this is the state encoding. Most trackers use a rectangular crop of the target object. While this makes processing easier, it also forces the network to include background information in the representation of the object. Therefore, the tracker needs to have some notion about the foreground to filter out the background noise. As we saw in some of the football player tracking videos, the tracker would latch onto some line and completely lose the player. The trackers we trained eventually learned to overcome most of the errors, but it still wasn't enough to correct every error case.

In addition to foreground recognition, trackers need to recognize the type of object being tracked. One error case where object recognition might help is when tracks become partial occluded. To prevent an error after the occlusion, the tracker must expand to include the newly visible part of the object. While a tracker with a robust foreground detection will be able to stay on the visible part of the track, it does not mean it would recognize the newly visible area is part of the track. Therefore, a perfect tracker would almost need to become a generic object detector to help recover from partial occlusions.

In the case a track becomes fully occluded, the tracker needs to recognize that the tracked object is not in the frame currently, and needs to have the ability to locate it

again when it reappears. Unfortunately, almost all deep learning trackers do not have the ability to recognize the disappearance of an object. Most of the trackers we tested should not be able to recover from full occlusion at all. In the GOTURN tracker for example, the only inputs are the previous crop and the search window. When the object disappears for even a single frame, the entire representation of the track is lost. Therefore, trackers will need the ability to include more information than just the previous frame when tracking.

These challenges will continue to push researchers for years. However, there are still many advances to be made in deep visual trackers. Most state-of-the-art tracking networks have been in the form of a saimese structure and this will most likely continue for the immediate future. Classification networks will continue to struggle when dealing with scale changes and the time it takes to track an object. Furthermore, their sampling methods will bound their accuracy. On the other hand, regression networks are fast but have only one chance to correctly detect the tracked object. While new architectures are being tested, researchers will also continue to add and combine larger datasets. This will allow for researchers to create more complex networks which in turn should increase the performance of visual trackers.

There are several interesting future avenues for research that we would wish to pursue. For one, there needs to be a better way of dealing with multiple images when used as the input into a CNN. Saimese networks architectures use convolutional stacks to encode the images. These encodings are then concatenated together and passed into a fully connected layer. This means that with each image added an equal number of parameters are added to the network which can cause the number of trainable parameters to increase dramatically. It also is not aligned with current trends of having fully convolutional networks, like in ResNet for example [10]. Another aspect that would be interesting is to explore how one could use recurrent layers so that the input to the network is only the frame being searched. These are avenues that have more to do with different architectures for tracking but do not generalize to all networks. A more general research avenue that tracking would benefit from is doing some form of anomaly detection to detect when trackers get lost. Many of these trackers have no mechanism to stop if they lose the track. However, in many commercial applications it is conceivable that a user would wish to know when a track left the screen or how confident it currently was.

Perfect tracking might be a long way off but visual tracking is rapidly improving. Deep learning has improved every aspect of computer vision and tracking is no exception.

Our work has sought to explore ways to improve visual tracking with different training methodologies for deep trackers and develop new architectures. We created a new architecture for tracking, named multi-channel deep tracker, to explore different ways of encoding the state of the track. We have specified a new imitation learning algorithm, called DAGGER^2, for training online deep trackers offline in a principled manner, such that our learning objective is better aligned with the end goal of tracking than those in existing formulations. This training method can be used by almost any tracker.

DAGGER^2 extends the well-understood framework of DAGGER, because the latter cannot be directly applied to tracking. DAGGER^2 iteratively fine-tunes the tracker by generating new training tracks based on the current performance of the tracker. We have tested DAGGER^2 on two benchmark datasets and another new dataset, using two different trackers, based on classification and regression networks. The results demonstrate that DAGGER^2 improves performance of both trackers relative to conventional training. We also compared DAGGER^2 to a new random history augmentation technique. Over our three tests, DAGGER^2 outperformed the random technique two out of three times. DAGGER^2 and the random augmentation technique outperformed the base case in every test. This highlights the importance of augmenting all inputs of the training data, whether using either technique. We have found that the omission of not augmenting the state encoding into the network hurt the performance of those trackers. It is our hope that other researchers use the lessons learned through our analysis to help improve the performance across all of their trackers.

Bibliography

[1] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[2] M. Danelljan, G. Hager, Shahbaz Khan, and M. F., Felsberg. Learning spatially regularized correlation filters for visual tracking. In *ICCV*, 2015.

[3] Martin Danelljan, Gustav Häger, Fahad Shahbaz Khan, and Michael Felsberg. Adaptive decontamination of the training set: A unified formulation for discriminative visual tracking. CVPR, 2016.

[4] Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. Structured prediction via output space search. *Journal of Machine Learning Research*, 15(1):1317–1350, 2014.

[5] J. Fan, W. Xu, Y. Wu, and Y. Gong. Human tracking using convolutional neural networks. *IEEE Transactions on Neural Networks*, 21(10):16101623, 2010.

[6] J. Gao, H. Ling, W. Hu, and J. Xing. Transfer learning based visual tracking with Gaussian process regression. In *ECCV*, 2014.

[7] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the KITTI vision benchmark suite. In *CVPR*, 2012.

[8] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

[9] X. Han, T. Leung, Y. Jia, R. Sukthankar, and A.C. Berg. Matchnet: Unifying feature and metric learning for patch-based matching. In *CVPR*, 2015.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[11] David Held, Sebastian Thrun, and Silvio Savarese. Learning to track at 100 fps with deep regression networks. In *ECCV*, 2016.

[12] Seunghoon Hong, Tackgeun You, Suha Kwak, and Bohyung Han. Online tracking by learning discriminative saliency map with convolutional neural network. *arXiv preprint arXiv:1502.06796*, 2015.

[13] J. Kuen, K.M. Lim, and C.P. Lee. Self-taught learning of a deep invariant representation for visual tracking via temporal slowness principle. *Pattern Recognition*, 48(10):29642982, 2015.

[14] Michael Lam, Janardhan Rao Doppa, Sinisa Todorovic, and Thomas G Dietterich. Hc-search for structured prediction in computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4923–4932, 2015.

[15] H. Li, Y. Li, and F. Porikli. Deeptrack: Learning discriminative feature representations by convolutional neural networks for visual tracking. In *BMVA*, 2014.

[16] X. Yang M. Yang M. Chao, J. Huang. Hierarchical convolutional features for visual tracking. *ICCV*, 2015.

[17] C. Ma, X. Yang, C. Zhang, and M. Yang. Long-term correlation tracking. In *CVPR*, 2015.

[18] Hyeonseob Nam and Bohyung Han. Learning multi-domain convolutional neural networks for visual tracking. *arXiv preprint arXiv:1510.07945*, 2015.

[19] Arnold W.M. Smeulders Ran Tao, Efstratios Gavves. Siamese instance search for tracking. *IEEE Confernece on Computer Vision and Pattern Recognition*, 2016.

[20] Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 1, page 6, 2011.

[21] David Silver, J Andrew Bagnell, and Anthony Stentz. Applied imitation learning for autonomous navigation in complex natural terrain. In *Field and Service Robotics*, pages 249–259. Springer, 2010.

[22] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

[23] Arnold W. M. Smeulder, Dung M. Chu, Rita Cucchiara, Simone Calderara, Afshin Deghghan, and Mubarak Shah. Visual tracking: an experimental survey. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 21(10):16101623, 2013.

[24] R. Tao, E. Gavves, and A.W.M. Smeulders. Siamese instance search for tracking. In *CVPR*, 2016.

[25] Lijun Wang, Wanli Ouyang, Xiaogang Wang, and Huchuan Lu. Visual tracking with fully convolutional networks. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[26] Naiyan Wang, Siyi Li, Abhinav Gupta, and Dit-Yan Yeung. Transferring rich feature hierarchies for robust visual tracking. *arXiv preprint arXiv:1501.04587*, 2015.

[27] Longyin Wen, Dawei Du, Zhen Lei, Stan Z. Li, and Ming-Hsuan Yang. JOTS: joint online tracking and segmentation. In *CVPR*, 2015.

[28] J. Lim Y. Wu and H.-H. Yang. Object tracking benchmark. *TPAMI*.

[29] S. Zagoruyko and N. Komodakis. Learning to compare image patches via convolutional neural networks. In *CVPR*, 2015.

[30] J. Zhang, S. Ma, and S. Sclaroff. MEEM: robust tracking via multiple experts using entropy minimization. In *ECCV*, 2014.

[31] K. Zhang, Q. Liu, Y. Wu, and M.H. Yang. Robust visual tracking via convolutional networks. In *arXiv:1501.04505*, 2015.

APPENDICES