Automatic Initialization of the TLD Object Tracker: Final Report

Louis Buck

May 29, 2012

Background

TLD is a long-term, real-time tracker designed to be robust to partial and complete occlusions as well as changes in perspective and scale [4]. It was developed by Zdenek Kalal and is available as a Matlab program using C libraries at his website [11]. Currently the TLD algorithm needs to be initiated by the user selecting a region of interest (ROI) in a frame of the video sequence. This prevents the algorithm from being used in completely autonomous tracking applications, or in applications where the operator cannot provide the necessary ROI input to the algorithm. For a full description of the TLD algorithm, please see Appendix A.

Figure 1: Example of TLD Object Tracking

Project Scope and Goals

The goal of this project is to design a separate object detector in Matlab that can automatically initialize the TLD tracker instead of relying on human input. The AdaBoost algorithm based on Haar features was chosen as the object detector due to its fast speed and relative simplicity. Based on the availability of training data sets and testing videos, I chose to train the detector to recognize cars from the side. The algorithm was trained on a training data base available from the University of Illinois[14], which consisted of 550 positive examples and 500 negative. The algorithm was tested on other testing images from the same UIUC database as well as two youtube videos[12][13].

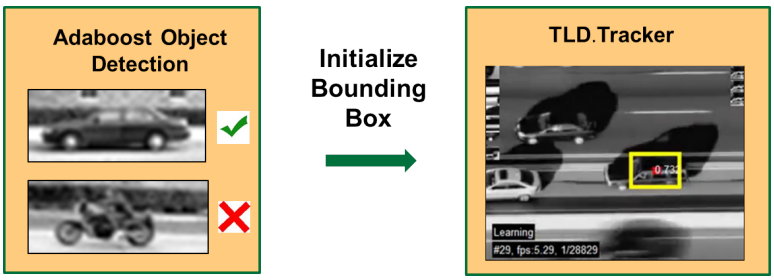

Figure 2: Project Goals

Methods

Haar-like Features

There are many different image features that can be used to characterize an object in an image so as to create an object classifier. These include local binary patterns[5], edge orientation histograms[8], and Haar-like (or just Haar) features[9]. For the purpose of fast detection of relatively simple, geometric objects (like cars), I decided Haar features were a good choice.

Haar features are based on Haar wavelets and represent intensity differences between adjacent local regions, approximating image features like derivatives, border detectors, line detectors etc. To evaluate a Haar feature one simply sums the intensities of the pixels in the black region and subtracts the sum of the pixel intensities in the white regions (see Figure 3). This can be done extremely inexpensively using integral images[5].

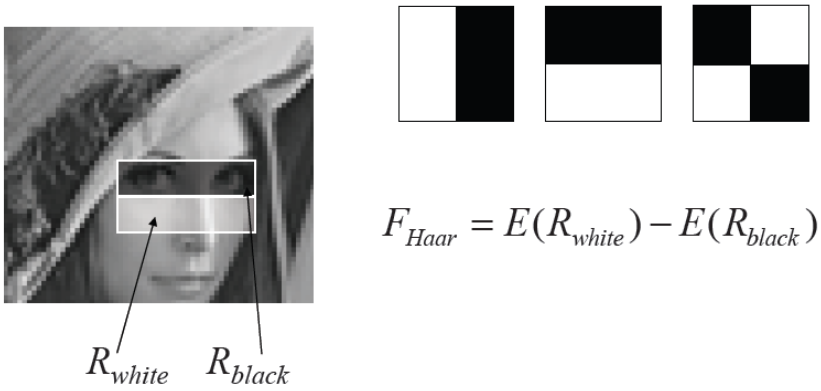

Figure 3: Calculating Haar Features

As can be seen in Figure 4, there are many different types of Haar features that can be considered. For the purposes of this project, I decided that the set of the 5 "Haar-like"

features should be sufficient, and if not the set could be expanded on in the future.

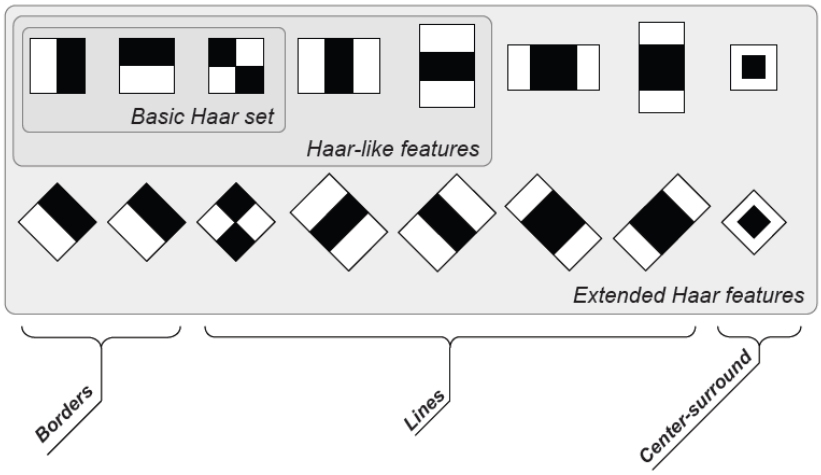

Figure 4: Haar Feature Types

The complete set of Haar features for a given image size consists of every scale, position, and aspect ratio of each feature type being used. For the 20x50 pixel images of the training set, this ends up being more than of 130000 features. Initial attempts to compute all of these features proved extremely time intensive, so I found a smaller, more sparse set of features by taking 4 pixel steps in the x and y directions reducing the number of total features to about 30,000. Example code from an on-line implementation of a Haar-based face detector was referenced for the implementation of this section of code[7].

Adaboost

Adaboost (Adaptive Boosting) is a type of boosting algorithm, a supervised machine learning technique that uses a weighted set of "weak learners", or learners that aren't very accurate, to comprise a one final, strong learner. Adaboost iteratively builds a set of T weak learners from a much larger set of learners, by choosing the learners that minimize the weighted error function.

$$\epsilon_j = \sum_i w_i |h_j(x_i) - y_i|$$

In each iteration, Adaboost considers every possible learner, evaluates their effectiveness according to this error function and chooses the learner with the smallest error to add to the set. The "adaptive" part of Adaboost is due to the fact that the weights w_i applied to each training sample are changed after each iteration so as to increase the weight given to training examples that were misclassified and decrease the weight of examples that were classified correctly.

$$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$$

where

$$\beta_t = \frac{\epsilon_t}{1 - \epsilon_t}$$

and $e_i = 0$ if data example x_i is classified correctly or $e_i = 1$ otherwise.

In this way, the algorithm is able to "adapt" and concentrate on the difficult examples so as to create a robust set of weak classifiers that can comprise an effective strong learner. After the set of T learners is found, this strong classifier is given by:

$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$

This description is largely derived from the code outlined by Viola and Jones, which can be seen in Figure 5.

- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
  1. Normalize the weights,

     $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$

     so that w_t is a probability distribution.
  2. For each feature, j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$.
  3. Choose the classifier, h_t, with the lowest error ϵ_t.
  4. Update the weights:

     $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$

     where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:

  $$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$

  where $\alpha_t = \log \frac{1}{\beta_t}$

Figure 5: Adaboost Code Outline

In the context of this project, each learner corresponds to a single Haar feature. Each learner in this case has to try and find a threshold that best differentiates positive and negative examples based on $(1 \text{x} M)$ set of values corresponding to the value of that Haar feature when

evaluated on the M training examples. To find this optimal threshold, I decided to minimize the misclassifications produced by the classifier. This was accomplished by iterating through possible threshold values (given by the set of feature values), calculating the number of misclassifications produced by threshold classification based on each value, and then finding the threshold that produced the minimum. The top 6 Haar feature-based weak classifiers chosen by AdaBoost are depicted in Figure 6.

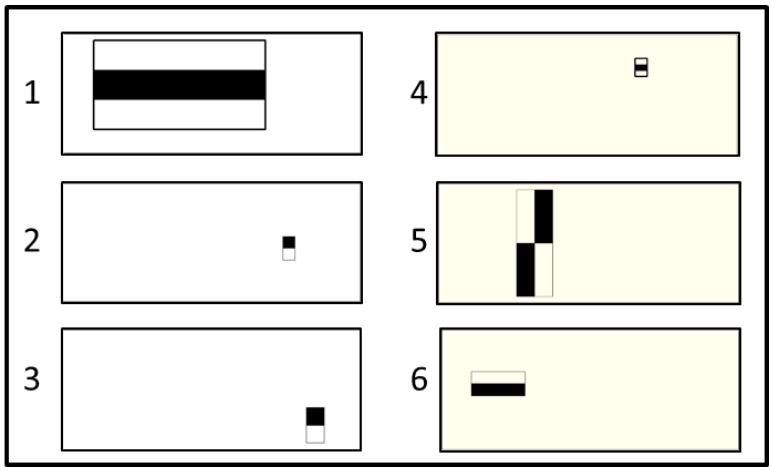

Figure 6: Top 6 Haar Features

Sliding Window

In order to find objects at multiple scales and positions within a test image, a "sliding window" implementation of AdaBoost was used. A sliding window works by "sliding" a test window across the image and testing each window for the presence of the object. This is repeated for multiple scales in order to enable detection of objects of different sizes.

Testing for the presence of an object in every possible sub-window in an image can be very time-consuming and would be difficult to do in real-time. To speed the process, my implementation skipped 20 pixels at a time in both the x and y directions. In order to test different scales, the process was repeated for images iteratively decreasing in size by a factor of 1.2 and stopping only once the image was as small as the window (20x50 pixels).

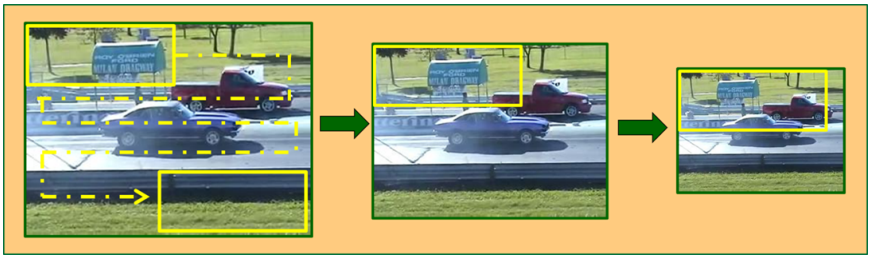

Figure 7: Sliding Window Visualization

OpenTLD

Integrating TLD first requires getting TLD working on Windows. This requires downloading and installing OpenCV and then compiling TLD's C and MEX functions. Instructions for integrating onto a 64-bit Windows 7 machine running 32-bit Matlab can be found in Appendix C.

OpenTLD expects to be pointed to a directory of sequential images. It expects that the first(lowest-numbered) image in this directory contains the object. The bounding box can either be created by the user via dragging a box on the this first image, or the coordinates of the box can be put into a textfile ("init.txt") in the image source directory. Consequently, without changing a lot of TLD code, integrating with TLD requires that the object must be present in the first image to be considered. This, in turn, prevents the AdaBoost code from initializing TLD mid-image sequence. For now, these restrictions don't prevent the proving the functionality of the automatic initialization, but as will be discussed in *Future Work* this is an issue that should be addressed in the future.

Results

Finding the Optimal T

The first phase of testing involved determining how the object detector performed on images of the same scale using different numbers (T) of weak learners. The most logical way to test this was via k-fold cross validation. 5-fold cross-validation was performed for $T = [1, 2, 3, 4, 5, 10, 20, 50, 100]$, and the resulting average errors $\left(\epsilon_T = \frac{\text{misclassifications}}{\text{training examples}}\right)$ recorded. The results of this test are plotted in Figure 8, and show the error decreasing asymptotically to a minimum with increasing T. Given the shape of the curve with an elbow at $T = 50$ and $\epsilon = .07$, a value of $T = 50$ was chosen for the AdaBoost classifier.

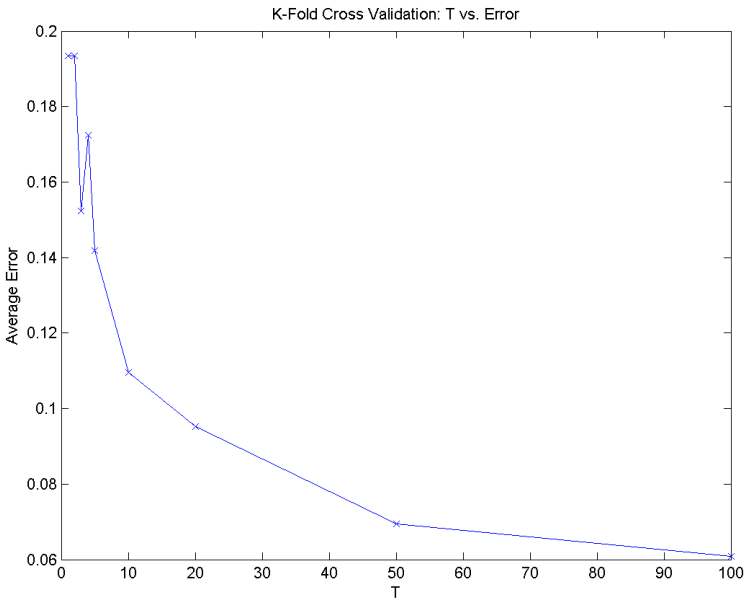

Figure 8: Finding Optimal T via Cross Validation

Final AdaBoost Classifier

Having determined the optimal value of T, the AdaBoost classifier was trained on the training set. Figure 9 provides some insight into the results and inner-workings of the final classifier. This flowchart represents three of $T = 50$ weak classifiers and their true values throughout that Adaboost process when used on one of the positive training images. This includes representations of the exact Haar-features chosen by the algorithm, their respective values, the threshold used by the greedy binary weak learner, the weights α, as well as the final output class.

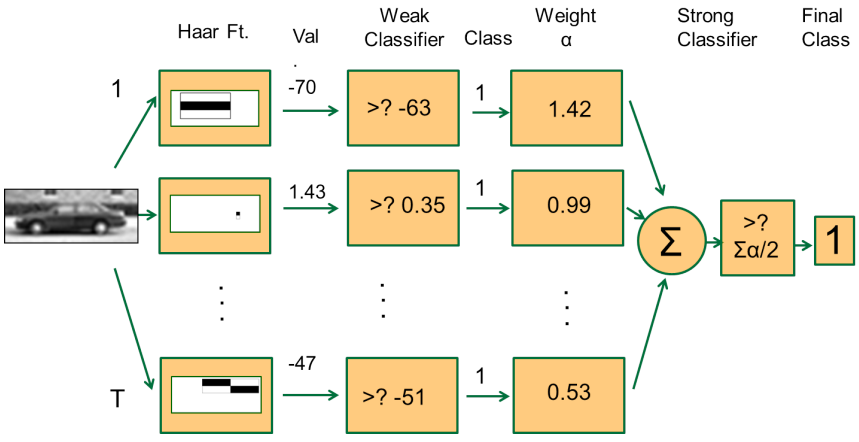

Figure 9: Boosting Pipeline

Sliding Window Results

The sliding window implementation was tested on a variety of multi-scale test images available in the same car dataset from UIUC. Due to time constraints and the inability to automate a procedure to test error rates, only qualitative testing was performed. These tests revealed three important results.

1. The detector is able to successfully detect objects at different scales in images with low background noise as can be observed in the left-hand image of Figure 10.

2. In the successfully-processed images, the detector often times finds more than one instance of the same object as can seen in the right-hand image of Figure 10.To resolve this issue, I used $\sum_{t=1}^{T} \alpha_t h_t(x)$ as a measure of confidence, and used the sub-window that returned a positive result with the highest measure of confidence as the final positively classified object bounding box. This can be seen successfully working in the right-hand image of Figure 10.

3. The detector had difficulty with images containing lots of objects or noise in the background and that false-positives were the most common type of error and the biggest concern. False detections can be seen in Figure 11The solution to this problem requires improving the object detector itself, and will be discussed in the *Future Work*

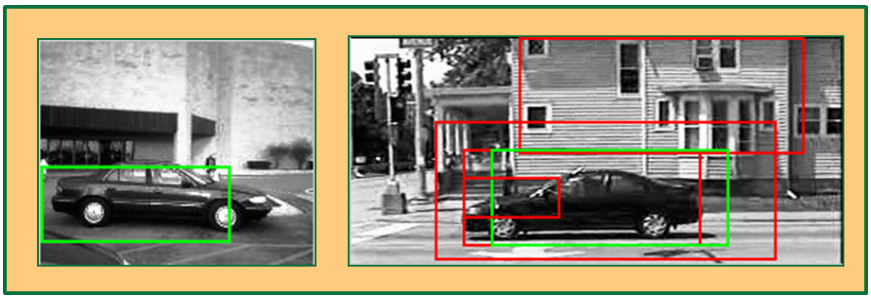

Figure 10: Sliding Window Good Detections

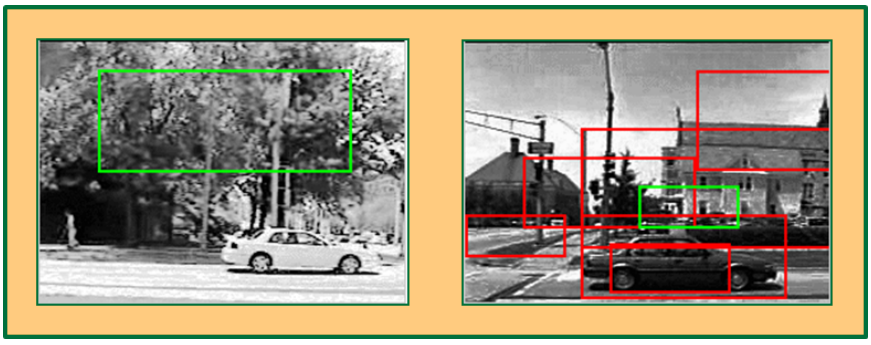

Figure 11: Sliding Window Bad Detections

TLD Initialization

The ability of AdaBoost to initialize TLD was tested on two youtube videos of cars driving in different orientations with respect to the camera, in settings with very little background noise [12][13]. In both cases, the video was cut so that the first frame contained a clear view of the car from the side. As can be seen in Figure 12, TLD was able to successfully identify and box the car in each video, and pass that bounding box to TLD, which then tracked the car for the duration of the sequence.

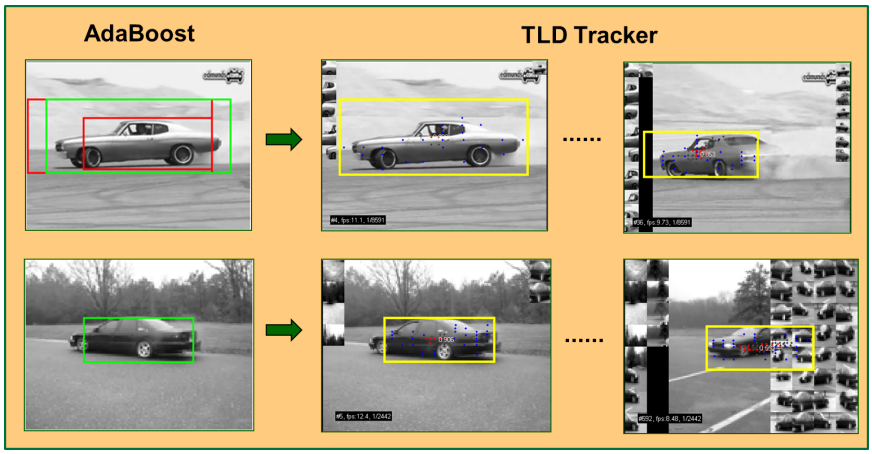

Figure 12: Successful TLD Integration

Future Work

Overall, this project was successful in initializing the TLD object tracker using a sliding window implemenation of AdaBoost based on Haar-like features. The following steps are suggested to improve upon the current project status in the future.

Cascaded Classifiers

Viola and Jones used an object classifier comprised of several cascaded AdaBoost classifiers of increasing T in their implementation[5]. Each stage can either reject the image or pass it along to the next stage. By using classifiers with an extremely low false-negative rates, this technique vastly reduces the average number of Haar-features computed per sub-window while maintaining high accuracy. Implementing a cascaded algorithm would allow for the use of a highly accurate AdaBoost classifier with a very large T, for a reasonable increase in computational cost.

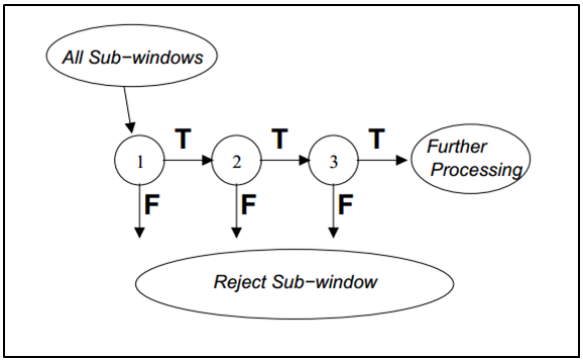

Figure 13: Cascaded Classifier Diagram

Larger Datasets

The objector in this project was trained on a relatively small training set of images (550 positive examples and 500 negative). It is likely due to this limited training set, that the detector had difficulties with false-positives. By using larger training sets in the future, we can expect improved performance.

Better (Real-time) Integration with TLD

As described in *Methods*, the current TLD initialization is limited because the current initialization of TLD is limited in that TLD requires that the first image in the directory contains the object in question. Though effort could be spent to allow TLD to initialize off of an image midway through the sequence, ultimately this algorithm would want to be used in real time, in which case no such directory of all images would exist. Instead, future work should center around incorporating the AdaBoost object detector with TLD in real-time.

References

[1] Z. Kalal, J. Matas, and K. Mikolajczyk. P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints.IEEE Conference on Computer Vision and Pattern Recognition, 2010.
[2] Z. Kalal, J. Matas, and K. Mikolajczyk. Online Learning of Robust Object Detectors During Unstable Tracking. IEEE Conference on Computer Vision Workshops, 2009.
[3] Z. Kalal, J. Matas, and K. Mikolajczyk. Forward-Backward Error: Automatic Detection of Tracking Failures. International Conference on Pattern Recognition, 2010.
[4] Z. Kalal, J. Matas, and K. Mikolajczyk. Tracking Learning Detection Demo Poster.
[5] P. Viola, and M. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. Accepted Conference on Computer Vision and Pattern Recognition, 2001.
[6] D. Geronimo. Haar-like Features and Integral Image Representation. Universitat Autonoma de Barcelona, 2009.
[7] http://www.ece301.com/ml-doc/54-face-detect-matlab-1.html
[8] J. Trefny and J. Matas. Extended Set of Local Binary Patterns for Rapid Object Detection. Computer Vision Winter Workshop, 2010.
[9] K. Levi and Y. Weiss. Learning Object Detection from a Small Number of Examples: the Importance of Good Features. The Hebrew University of Jerusalem.
[10] D. Hoiem. Object Category Detection: Sliding Windows. University of Illinois, 2011.
[11] https://github.com/zk00006/OpenTLD
[12] http://www.youtube.com/watch?v=K3W7i12a1s4&feature=related
[13] http://www.youtube.com/watch?v=vIw6wRAMGp0
[14] http://cogcomp.cs.illinois.edu/Data/Car/

APPENDIX

A TLD Tracker Description

The TLD object tracker was developed by Z. Kalal, J. Matas and K. Mikolajczyk at the University of Surrey and Czech Technical Institute. It is called TLD because it utilizes three components in parallel to accomplish the task of long-term tracking, tracking, learning and detection [4].

Tracking
The tracker used is a median-shift tracker based on Lucas-Kanade optical flow algorithm. The tracker provides an estimation of the trajectory of the object based solely on the frame-to-frame movement of key points, and independent of the system's object model.

Detection
The detector is a random forest classifier based on a collection of 2-bit binary patterns.These binary patterns are discretizations of the gradients across randomly sized and located pixel patches (called groups) within the region of interest. Each group yields its own decision tree within the random forest, the leafs of which represent different positive representations the detector has found within that pixel patch.

Learning
The learning that takes place is a semi-supervised process that fuses results from the object detector and tracker to iteratively improve the object model. False negatives close to the tracked trajectory extend the decision trees (grow the forest) by positively labelling the tracked patch and retraining the model. False positives far from the tracked trajectory prune the forest by removing leaves that led to the false identification. The tracker is initiated by the user bounding the object of interest with a box in a single frame. The initial random forest model is trained with 100 different affine transformations of this single labelled example [1][2][3][4].

B Matlab Implementation Explanation

TLD Auto-initialization

The script to run the auto-initialization is included in the main directory "\Matlab Code" and is called final_script.m. The directory of images can be changed by commenting and uncommenting the desired variables "dir_name" and "frame1". Upon running, the script will automatically run AdaBoost and initialize the chosen sequence. After the sequence has finished playing, the bounding box output of AdaBoost will be displayed (this happens after TLD runs its course due to figure conflicts).

AdaBoost

Various parts of AdaBoost can be tested by uncommenting portions of "\Matlab\AdaBoost Code\test_script". All parts of the script are commented and descriptions of the functions called can be found in the function files themselves.

TLD

TLD can be run independently of AdaBoost by running "\Matlab\TLD\openTLD\zk00006-OpenTLD-8a6934d\run_TLD". Note that to initialize the tracker by hand, one must delete "init.txt" from the image source directory.

C Installing OpenTLD in Windows

1) Install OpenCV 2.4
    a. DO NOT NEED TO RECOMPILE. Just run the executable. All lib and dll files that would be created by compiling are included in opencv \build \vc9 \lib
2) Set system path to include required dll files: \build\vc9\bin
    a. Right click my computer
    b. Properties
3) Download OpenTLD
4) Set up mex compiler in Matlab to use Visual Studio
    a. Mex -setup
5) Change compile.m
    a. Change and lib source paths as in Matlab files. Lib source path is \build\vc9\lib
6) Change some files
    a. Comment out in bb_overlap.cpp, fern.cpp and lk.cpp
        # ifdef CHAR16T
        # define CHAR16_T
        # endif
7) Changelk.cpp
    a. cvCalcOpticalFlowPyrLK() 10th argument should be status, not 0
8) Run compile.m
9) Copy some dll files into mex/ directory
    a. From Matlab Runtime
        i. libmex.dll, libmx.dll
        ii. All opencv dlls from \vc9\bin (not necessary)