Automatic Initialization of the TLD Object Tracker: Milestone Update

Louis Buck

May 08, 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]. The algorithm is of interest to my research in object tracking using machine vision on a quadrotor micro air vehicle (MAV). 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 scope of this project has changed significantly since the proposal. I originally proposed to implement a Matlab version of the TLD tracker and then implement a separate object detector in Matlab to initialize the TLD. Following the recommendation of Professor Torresani, I have since scaled back my project to concentrate on the object detector implementation in Matlab and integrating it into the open-source, C++-based TLD code available on-line. Consequently, the formal goal of my project is now to write a Matlab implementation of the Adaboost learning algorithm based on Haar-like features to automatically recognize a car from the side in a video sequence and initialize the bounding box required by TLD so that it may then track the car.

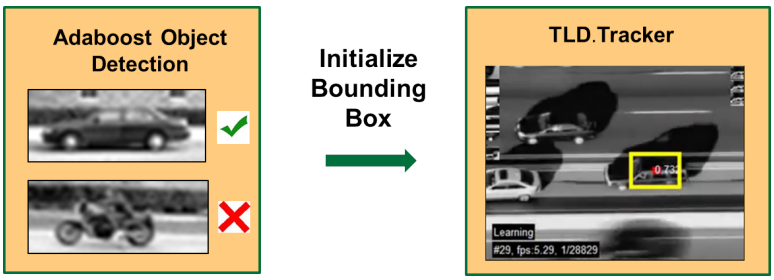

Figure 2: Project Goals

Algorithm Implementation

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].

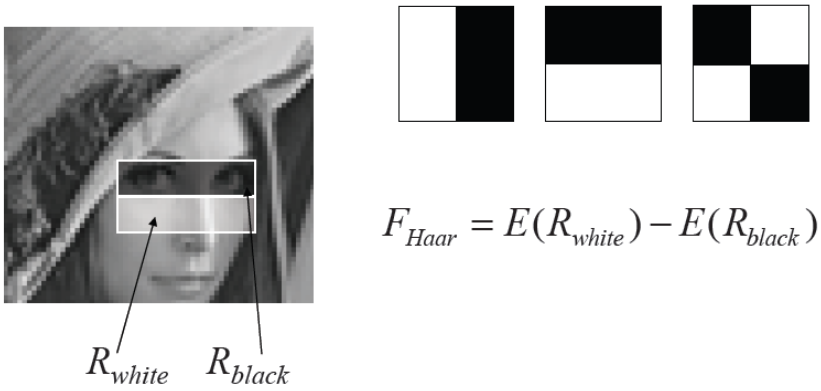

Figure 3: Calculating Haar Features

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.

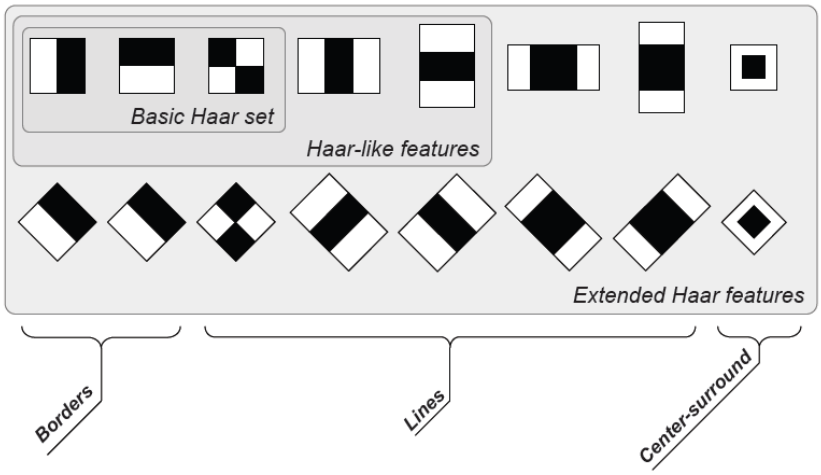

Figure 4: Haar Feature Types

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. Depending on the size of the image or subimage under consideration, this ends up being on the order of 100000 features. 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, 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. Thus, our initial pool of learners is on the order of 100,000. Each learner in this case has to try and find a threshold that best differentiates positive and negative examples based on (1xM) 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 maximize the decrease in entropy

given by the equation.

$$i(\tau) = -\sum_{c \in C} P_c^{(\tau)} \log P_c^{(\tau)}$$

This was accomplished by iterating through possible threshold values (given by the set of feature values), calculating the decrease in entropy for each case, and then finding the threshold value that yielded the maximum decrease.

Results

Preliminary results testing the algorithm on the training set indicate that the classifier is working. The training set was used simply as an fast, easy preliminary test (true testing will require k-fold cross-validation and then sliding window object detection). The algorithm is able to correctly identify images from the training set with an accuracy ranging between 80% for $T = 1$ and 95% for $T = 20$. Again, these numbers are not representative of the true accuracy of the classifier as they were found by testing the classifier on the same image set that it was trained on, but they do indicate that the algorithm is working. A plot showing the decrease in misclassifications as the the number of features used increases can be seen in Figure 6.

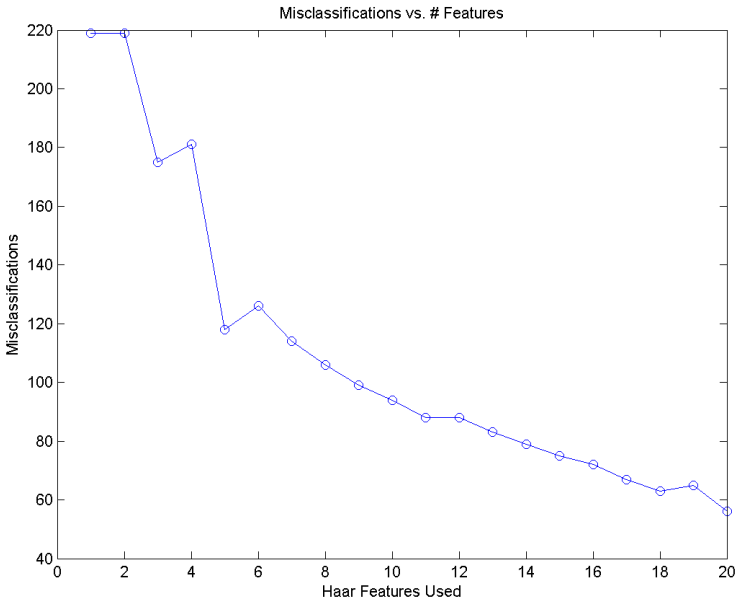

Figure 6: Different Haar Features

Figure 7 provides some insight into the inner-workings of the final classifier. This flowchart represents three of $T = 20$ 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 entropy-reducing binary weak learner, the weights α, as well as the final output class.

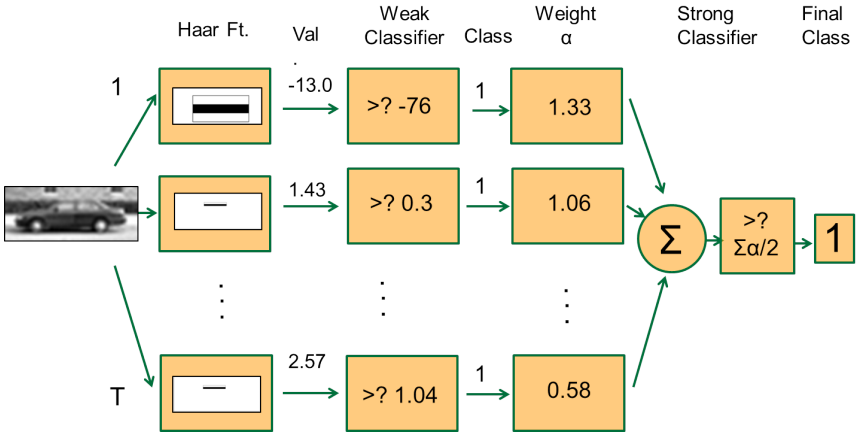

Figure 7: Boosting Pipeline

Future Work

There are several steps that still need to be taken for the successful completion of this project:

1. Test the single-scale classifier effectiveness and find optimal number of Haar-features using k-fold cross-validation.

2. Implement the sliding-window searching algorithm to enable object detection at different scales[10].

3. Compile and run the TLD object tracker using open source code[11]

4. Integrate the Adaboost output into TLD bounding box input.

5. Test the integrated algorithm on several youtube videos containing sideviews of cars moving in the video sequence[12][13].

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=QKUT2WUMTfk&feature=related
[13] http://www.youtube.com/watch?v=-LHHq31xNTU

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].