Tracking-Learning-Detection Documentation

We started this project in order to gain a better understanding of the concepts behind the Tracking-Learning-Detection algorithm (TLD). It was observed that while the relevant papers provide an overview, they lack the precise details required to implement the system. The open source implementation available online is also difficult to glean insight from.

To answer our questions we chose to implement the algorithm ourselves. Our initial approach was to code in Matlab; however, incorporating C++ provided necessary and substantial speed improvements. The majority of the source was thus written in C++ for this reason. Converting the entire project into C++ may be beneficial, though use of Matlab was maintained due to better webcam support. A main unexplored area of interest for this project is improving tracking performance at smaller patch sizes where the classifier currently breaks down.

Contents

Compilation

Matlab is required to run the provided TLD implementation, although the source code is mainly written in C++ for efficiency. This implementation must be compiled before it is executed. Each operating system requires a different compilation. However, the file *compile.m* should compile the whole project successfully on Linux, Windows or Mac. OpenCV2.2 is also required. If your OpenCV2.2 installation is not in the standard location you must alter the file paths in *compile.m*. If your OpenCV2.2 file paths are correct, the following steps should compile the project:

1. Open Matlab
2. Use the 'cd' command to change directory to the TLD Implementation directory
3. Compile the program simply by running the *compile.m* file with the instruction 'compile'

If this does not work ensure you are using OpenCV version 2.2, that you have indicated the correct location of it, and that your mex compiler is set-up correctly. Use the command 'mex -setup' to change the C++ compiler used.

Running the Program

A webcam is required to run the program. Start the program in Matlab by calling the *tld.m* file with the instruction 'tld'. Click and drag to draw a bounding-box around the object you wish to track; if the bounding-box is too small you will be required to draw it again. The bounding-box can be resized and repositioned after being drawn. Double-click inside the bounding-box to begin tracking. A yellow box will appear around the object while it is tracked. This box changes scale with the tracked object and will jump to a new location when it becomes more confident about another position. The box will disappear if the tracker is exceedingly uncertain about the object's position, or the object was lost and has not been redetected with significant confidence. Close the program by force quitting with Ctrl + C (Command + C in Mac), errors will be reported as a result of the forced exit but this is expected.

Algorithmic Overview

TLD is motivated by the shortfalls in 'short-term' trackers when applied to longer tracking tasks. Short-term trackers 'drift' away from their targets over time and are further limited by complete object occlusion or disappearance.

This section provides an algorithmic description of TLD, also known by the names Predator and Tracking-Modelling-Detection (TMD). The process is to use a tracker as the supervisor for training a detector, which is used to reinitialise the tracker when its own detections are more likely to be the object. We describe the classifier, detector, tracker, and learning process, respectively.

The Classifier

The TLD classifier is a random forest classifier consisting of random ferns. We have a slightly special case in that there are only two classes (positive and negative).

Random Ferns

Random ferns are similar to random trees except that all patches are evaluated at every node. Instead of using a tree we have an ordered set of nodes. Each node contains a test on the patch: a feature. We describe the result of the test as a number represented in binary. Accumulating, in order, the results of all the nodes computed on a single patch, defines a binary code that specifies a specific leaf node.

For the two-class case, each leaf node records the number of positive p and negative n examples training examples that fall into it. The posterior then is computed by maximum likelihood estimator $P(c = 1) = p / (p + n)$ or 0 if $p = 0$ where c is the class of the patch and is 1 if positive. In the case where we have more than 2 classes, the posterior is computed as $P(c = X) = x / n$ or 0 if $x = 0$, where X is the class we are computing the posterior for, x is the number of class X training examples that have fallen into this leaf, and n is the total number of training samples that have fallen into this patch. Note that while an initial training phase is required, this classifier can be updated with new samples at any time.

Random Forest of Ferns

Each fern has a different set of features and all training examples are passed through each fern. The initial training is performed in the first frame. Positive samples are added to the training set by affine warping the selected object patch (using more general homographies is untested but should also work). Negative patches are created by passing a sliding window over areas far from the selected object patch. Training is repeated with each iteration of the learning process; in practice, this only needs consist of updating the posteriors with several new examples.

In order to classify a new patch, it is put through each fern until it reaches a leaf node. The positive class posteriors $P(c = 1)$ are averaged over the ferns and the classifier outputs a positive response if the average is greater than 0.5. This can be generalised to more classes.

Features (Tests)

Two different features have been used: the papers describe the *2-bit Binary Pattern* while the open source implementation uses a 1-bit binary pattern (we assume these are the same as simple Haar-like features). If there are N nodes in a fern then it has 2^N leaf nodes for 1-bit results or 4^N leaf nodes for 2-bit results. For a more detailed description of 2-bit Binary Patterns see the paper '*Online learning of robust object detectors during unstable tracking*'.

The Detector

We use a sliding window approach, scanning the entire frame with overlapping bounding-boxes at several scales and classifying each patch with the classifier. The papers indicate that positive examples are returned, however results were noticeably improved by returning boundary cases as well. These cases are patches classified as positive that were spatially distant from the tracked patch this frame and patches classified as negative that were spatially close to the tracked patch this frame. Spatial distance is described as the spatial overlap of bounding-boxes: if a detected patch overlaps the tracked patch by more than some threshold it is considered that the detected patch must be positive if (and only if) we are confident that the tracked patch tightly contains the object we are tracking.

The Tracker

The tracker described in the most recent paper is the *Median Flow* tracker. Our implementation does not contain steps 3 and 4 of the algorithm below as tracking was deemed exceedingly suitable for purpose without these. However, these stages are important for slower frame rates than were achieved in our implementation. For a more detailed view see the paper '*Forward-Backward Error: Automatic Detection of Tracking Failures*'.

Algorithm

Input: Frames at times t and t + 1, and a bounding-box (over the object) at time t.
Output: A bounding-box (over the object) at time t + 1.

1. Initialise a uniform set of points within the bounding-box at time t.
2. Track these points with Lucas-Kanade.
3. Estimate the tracking error of each point (using both Forward-Backward error and normalised cross-correlation independently).
4. Filter out the 50% of points with largest error.
5. Use the remaining points to estimate the position of the bounding box at time t + 1.

Forward-Backward error: using an arbitrary tracker, from time t, track point forwards n frames, then track the point found at time t + n backward n frames. The error is the Euclidian distance between the original point and the back-tracked point at time t.

Normalised cross-correlation: normalised cross-correlation between the two patches at times t and t + 1.

The Learning Process

We describe the process of called *P-N Learning*. For a more detailed view see the paper '*P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints*'.

Initialisation

Add to the training set positive and negative examples extracted as aforementioned. Train the classifier on this set.

Iteration per Frame

1. The tracker returns the tracked patch. The detector returns detected patches. All these patches are added to our unlabelled dataset. Samples in this dataset are then classified by the classifier (not removed; each frame this set increases in size).
2. If the confidence (Normalised Cross-Correlation between a patch and the first frame patch) of the previous trajectory patch is greater than 80% and the tracked patch has higher confidence than all detected patches then we apply the P-N constraints, otherwise we discard all tracked and detected patches from our unlabelled dataset. If a detected patch had higher confidence, we reinitialise the tracker at this patch.
    a. Application of P constraints: samples labelled in 1. as negative that are spatially close to the tracked patch must be positive, and are added with a corrected label to the training set.
    b. Application of N constraints: samples labelled in 1. as positive that are spatially distant from the tracked patch must be negative, and are added with a corrected label to the training set.
    c. Closeness to the tracked patch is defined as having a spatial patch overlap that is greater than 60% of the area both bounding boxes cover. Overlap = intersection area / (area of bounding-box 1 + area of bounding-box 2 – intersection area).
3. The classifier is retrained on the training set. In practice, we only require updating the p and n variables in the leaf nodes the new patches fall into in order to update the posteriors.

Our implementation differs from this algorithm in several ways. Importantly, there is the aforementioned difference in the return values of our detector. We also realise that keeping such a training dataset requires storing every frame that contains a training example. This makes the system unscalable to very long video sequences and we also notice that performance (in terms of frames-per-second) decreases slightly over time in the TLD Demo application. Therefore, we do not store a training dataset and thus application of the P-N constraints now means retraining on each example from our detector once only in the frame in which they occur. Despite this difference, we do not notice, by eye, a difference in the system's ability to learn.

Another crucial difference is that we define the confidence of a patch as the value returned from the classifier, the higher the value the more confident we are. This provides an evaluation method which changes over time, unlike the original approach. Further, we do not limit the number of training examples evaluated each frame as in the TLD Demo. It is also worth noting that using different constants (such as patch overlap threshold) can noticeably alter the behaviour of the system and may be worth experimenting with.

Implementation Overview

Files

The source code provided comes in two distinct sections: the Matlab source in .m files and the C++ source in .h and .cpp files.

Matlab Source

- **tld.m:** program entry point; runs TLD using your webcam.
- **init.m:** program initialisation; opens a figure displaying a raw camera stream and allows the user to define a bounding-box around the object.
- **updateDisplay.m:** updates the information displayed in the figure; called after each frame has been processed. Uncomment the for loop to display the detector bounding-boxes – these are drawn in green if the bounding-box was a positive training example, and in black for negative examples.
- **compile.m:** compiles the C++ elements of the program.

C++ Source

- **TLD.cpp:** entry point for Matlab. Calling *[left, hand, side, variables] = TLD(arguments);* will call the method *mexFunction*. We designed this method to be called in two ways, one for initialisation and one to process a frame. These calls are described in the comments above the method.
- **Detector.h, Detector.cpp:** the object detector class. While successfully tracking, searches patches ranging in size around the tracked patch size; when the object is lost, searches patches ranging in size around the initial first frame patch.
- **Tracker.h, Tracker.cpp:** the Median Flow tracker class.
- **Classifier.h, Classifier.cpp:** the random forest classifier class.
- **Fern.h, Fern.cpp:** the random fern class.
- **Feature.h, Feature.cpp:** the superclass for patch features (tests). Features are initialised as a random rectangle defined as a percentage x, y, width, and height on the patch space. Features are then calculated on these randomised rectangles. This class was added to provide an easy way to switch between several features.
- **HaarTest.h, HaarTest.cpp:** subclass of *Feature*, a Haar-like feature on a patch. Currently unused – replaced by more effective TwoBitBPTest.
- **TwoBitBPTest.h, TwoBitBPTest.cpp:** subclass of *Feature*, a 2bitBP feature on a patch.
- **IntegralImage.h, IntegralImage.cpp:** integral image class. Integral images are used for efficiency when calculating features.

Coding Style

A few important points will be mentioned. All classes are described, often in some detail, above their declarations. All methods are commented above their first declarations, usually in the header file. These comments contain a description of what the method does, what it returns, and what each parameter is. Each file defines a maximum of one class. All class fields are described above their declaration in the header file. Values declared with the *#define* directive appear in header files and also in *TLD.cpp*, and are commented thoroughly.

Observations

General

Observations in this section apply to both the TLD Demo application and our own implementation.

Positive

- Tracks successfully from frame 1
- Learns weak transformations (such as small rotations)
- Not affected by object disappearance or complete occlusion
- Not affected by object scale
- Robust to some partial occlusion

Negative

- Sensitive to the object's spatial surroundings
- Slow to learn appearances, appears to require training time
- Sensitive to transformations (such as rotation)
- Sensitive to things that look similar
- Sensitive to reflections on the object
- Sensitive to lighting conditions
- TLD Demo gets slower over time
- Detection fails with smaller patch sizes (must be due to a lack of detail causing the features to lose discriminability)

Neutral

- Better at tracking slow motion than fast motion
- Tracks some objects much better than others (those with more detail are probably tracked better as the features should be more selective on these)
- Better at tracking than re-detecting

Our Implementation

- *MIN_REINIT_CONF* (defined in *TLD.cpp*) is exceedingly important in determining the 'jumpiness' of the trajectory. If this value is too low, we reinitialise the tracker more often and track by detection from the beginning.
- The ratio of positive to negative training examples for the classifier is very important during initialisation. Too many positive examples means we over-fit our classifier and make many incorrect detections, and too many negative examples means we under-fit and make few detections even when the object is seemingly clear.
- Similarly, creating too much variation in the warped positive patches used to initialise the classifier tended to decrease classification performance in that patches less like the object than expected were returned as positive matches.

- Over time more patches that bare less apparent similarity to the object are classified as positive (and thus used as negative training examples). This can sometimes be seen when tracking for a longer time while displaying the patches returned from the detector.
- Due to the lack of success of patches at small sizes we do not classify patches smaller than a certain size, which means that objects that were close but become more distant sometimes switch to tracking with Lucas-Kanade only.
- Changing the number of ferns in the classifier and also the number of nodes per fern (with the *TOTAL_FERNS*, and *TOTAL_NODES* values defined in *TLD.cpp* respectively) has a bearing on the success of the detection. This appears to change whether we over-fit or under-fit our classifier.
- In the method *trainNegative* in *TLD.cpp*, which trains the classifier on negative training patches, we experimented with also training the classifier on positive patches when they were found, however this proved to decrease classification success.
- Just using the Lucas-Kanade tracker without any detection is very successful for a while at high frame rates (greater than 30) before drifting. At slower frame rates (less than 20) this method drifts much more quickly.
- 2-bit Binary Patterns worked much better as features than Haar-like features.

References

The first 3 papers are provided with the implementation and provided the starting point for our own research. The remaining papers proved useful.

Z. Kalal, K. Mikolajczyk, J. Matas. *Online learning of robust object detectors during unstable tracking* (2009).

Z. Kalal, K. Mikolajczyk, J. Matas. *P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints* (2010).

Z. Kalal, K. Mikolajczyk, J. Matas. *Forward-Backward Error: Automatic Detection of Tracking Failures* (2010).

M. Özuysal, P. Fua, V. Lepetit. *Fast Keypoint Recognition in Ten Lines of Code* (2007).

P. Fua, V. Lepetit. *Towards Recognizing Feature Points using Classification Trees* (2004).

M. Özuysal, M. Calonder, P. Fua, V. Lepetit. *Fast Keypoint Recognition using Random Ferns* (2010).

T. Vojír. *Demo application for Tracking-Modeling-Detection* (2010).