RGB-D tracking attempt with DS-KCF

CMPUT 428

Wenzhang Qian

1459776

April 27, 2018

Table of Contents

Abstract

An RGB-D camera, unlike a conventional camera, provides per-pixel depth information in addition to an RGB image. Moreover, it is more and more popular in recent years because of the significant cost reduction in RGB-D camera, which makes it a better and cheaper device for depth information than stereo cameras.

The purpose of this project is to explore the tracking system based on both color image and depth image. The main focus of this project is to reproduce DS-KCF [1], which is a modified version of KCF tracker by integrating depth information in tracking.

Depth information can be integrated into RGB tracker to overcome problems when the target rotates, deform or is under occlusion. KCF is a really good tracker and a starting point. DS-KCF purposed a compute-efficient way to make use of the depth information dealing with occlusion. The implementation is hard and the result does not pay for the work. However, it shows the possibility of integrating depth information with traditional RGB trackers to overcome some problems hard to solve with only color information.

Introduction

An RGB-D camera combines best of the active and passive sensor worlds, in that it consists of a passive RGB camera along with an active depth sensor. An RGB-D camera, unlike a conventional camera, provides per-pixel depth information in addition to an RGB image.

Traditionally, the active depth sensor is an infrared (IR) projector and receiver. Much like a Continuous Wave Time of Flight sensor, an RGB-D camera calculates depth by emitting a light signal on the scene and analyzing the reflected light, but the incident wave modulation is performed spatially instead of temporally.

Kernelized Correlation Filter (KCF) combines both high accuracy and fast processing speed. It is in the eighth position in the top ten [2] while being the fastest.

Dr. Hannuna and his team proposed an augmented version of KCF [1] with point cloud called DS-KCF. It improves on the performance of the KCF tracker by 1)integrating depth and colour features in the KCF framework, and efficiently handling 2)scale changes 3)occlusions 4)aspect ratio changes of the target model

Procedure

The first step of this project is to find a decent RGB tracker as the basis to work on. The first idea is KCF tracker because it is both accurate and fast and its performance has been tested in previous projects. Other algorithms (MIL, BOOSTING, MEDIANFLOW, TLD) have been considered. The biggest advantage of KCF is that an augmented version of KCF with point clouds called DS-KCF has been purposed. Moreover, it ranks at the 4th place in Princeton Tracking Benchmark, which proves it is a good tracking algorithm.

A general work flow of KCF tracker is attached below.

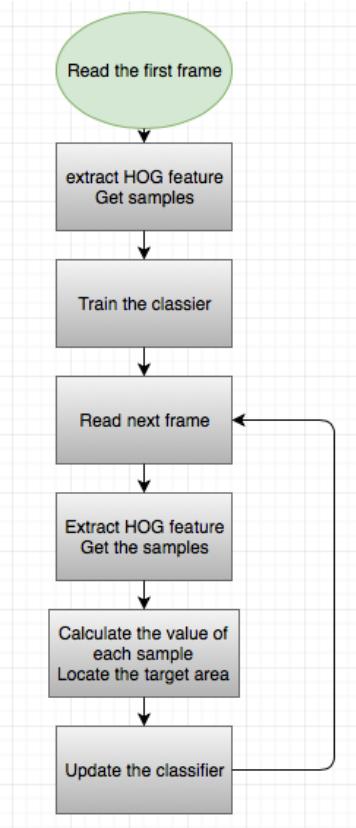

Figure 1: Work flow of KCF tracker

The most important feature used in KCF tracker is Histogram of Oriented Gradient (HOG). It is a common feature in both detection and tracking.

The gradient in x,y direction is calculated by

$$G_x = I(x+1, y) - I(x, y)$$

$$G_y = I(x, y+1) - I(x, y)$$

The gradient in magnitude is calculated by

$$G(x, y) = \sqrt{G_x^2 + G_y^2}$$

The gradient in direction is calculated by

$$\theta = \arctan\left(\frac{Gy}{Gx}\right)$$

The author of the paper purposing the KCF tracker [3] uses FHOG in his implementation. Felzenszwalb's HOG (FHOG) features is a fast implementation of the HOG variant used by Felzenszwalb et al. in their work on discriminatively trained deformable part models.

[http://www.cs.berkeley.edu/~rbg/latent/index.html] Gives nearly identical results to features.cc in code release version 5 but runs 4x faster (over 125 fps on VGA color images).

The result of FHOG feature extraction is a tensor of the size of Wxhx31, where

$$W = \frac{Width}{cellsize}$$

$$h = \frac{Height}{cellsize}$$

The best visualization of FHOG is attached. Unfortunately, Chinese is used in the visualization. However, the idea of FHOG extraction is given.

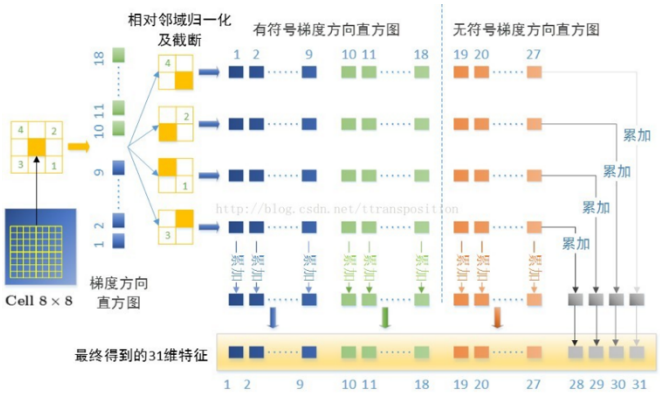

Figure 2: FHOG Feature

One of the most important creations purposed in this paper is Circulant Toeplitz Matrices.

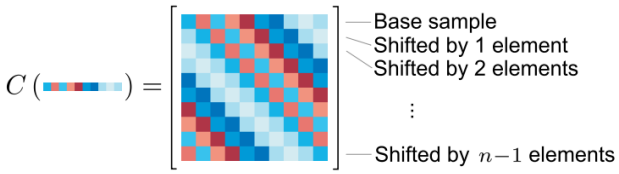

Figure 3: Circulant Toeplitz Matrices

By using Circulant Toeplitz Matrices, dense sampling is obtained at a fast speed. Two other sampling algorithms are random sampling and sliding windows. They are worse than Circulant Toeplitz Matrices because random sampling will miss some datapoints while sliding windows is too slow.

Gaussian projection is also done to separate datapoints as well. Datapoints in tracking problem cannot be separated linearly. However, they can be separated linearly after they are projected to a higher dimension. An example of 2D -> 3D project is attached below.

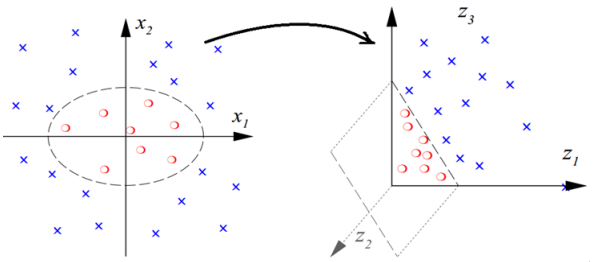

Figure 4: Gaussian Projection

During the projection, a Gaussian kernel is used.

$$K(\mathbf{x},\mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|^2/2\sigma^2}$$

It is difficult to find the explicit expression of this projection in most cases. However, the inner product of datapoints (in higher dimension) can be used for classification.

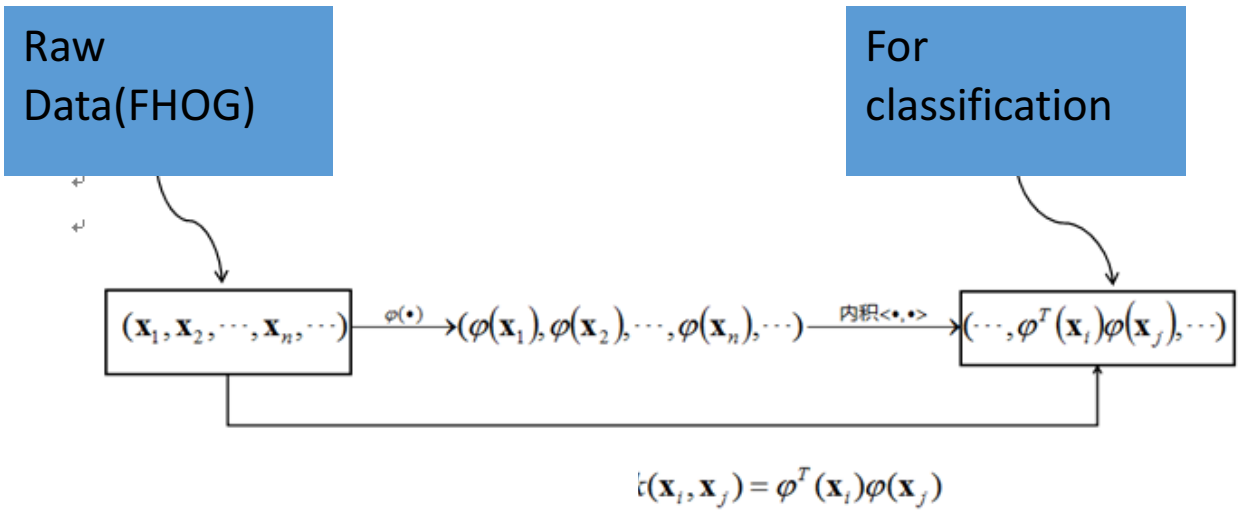

Figure 5: Workflow of Gaussian Projection

After understanding KCF tracker, two ways of improving the performance is purposed.

The first is the combination of parameters used in KCF. There are three major tunable parameters I care about.

Parameters	Padding (search area)	Interploation-factor	Lambda
function	Provide the search region	Decide the refresh speed of the classifier	Prevent overfitting of the classifier
how	Higher the padding, better the resistance for fast moving objects	Higher the factor, better for fast motion at the cost of stability	Too small → over fitting Too larger → under fitting

Table 1: Tunable Parameters

The second is the performance measurement. My performance measurement is: FPS > Location difference > Scaling difference. I care about the speed the most because KCF is the fastest among the most accurate trackers. Then the location difference is more important than the scaling difference.

Then DS-KCF comes into the play. The general workflow of DS-KCF is attached below.

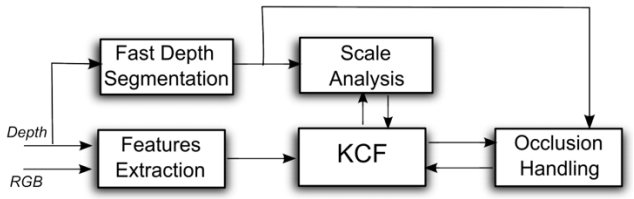

Figure 6: Workflow of DS-KCF

Integrating depth information, the tracker can deal with scale difference and occlusion. There are three major parts of DS-KCF.

The first part is fast depth segmentation. There are two steps in this part.

1. K-mean algorithm to evaluate the initial cluster or ROI. And in K-mean algorithm, there are three major steps.
    a. In HOG, every local maximum is used as the initial cluster center.
    b. Every datapoint is distributed to its closest cluster.
    c. Keep updating the centers until convergence.
2. Accurately determine the ROI by analyzing the feature of the neighboring region. After this step, regions with same depth are separated and the small regions are removed.

The second part is detecting and handling scale changes. There are two types of scaling factor used in DS-KCF.

Continuous scaling factor: $S^r = d_{obj}/d_{obj}^{t_0}$

And measurement scaling factor: $S^q = \{s_j, (\forall j = 1 \dots J)\}$

A factor in S^q which is closest to S^r is selected.

The assumption made in this part is that the depth distribution in the bounding box is a Gaussian distribution, which is definitely wrong in most cases. However, it can be a good starting point to develop the algorithm.

The condition of occlusion is when

$$\left(\Phi\left(\Omega_{obj}\right) > \lambda_{occ}\right) \wedge \left(f(z)_{max} < \lambda_{r1}\right)$$

In my implementation, I use $\lambda_{occ} = 35\%$ and $\lambda_{r1} = 0.4$.

When the condition of occlusion is triggered, the occlusion object is tracked. During occlusion, the search region in current frame(i) is

$$\Omega^{i}_{T_{search}} = \Omega^{i-1}_{T_{occ}} \cup \Omega^{i-1}_{T_{bc}} \cup \Omega^{i}_{T_{occ}}$$

The recovery condition of occlusions is

$$\left(\text{Overlap}\left(\Omega_{T_{occ}}, \Omega_{T_{bc}}\right) < \lambda_{occ}\right) \vee \left(f(z)_{max} > \lambda_{r2}\right)$$

In my implementation, $\lambda_{occ} = 35\%$ and $\lambda_{r2} = 0.2$ are used.

All my work is built upon the KCF tracker I found on https://github.com/joaofaro/KCFcpp.git. The three parts I modified are in getDepth() in kcftracker.cpp (Fast Depth Segmentation) and update() in kcftracker.cpp (Scale Changes and Occlusion Handling). The main function is written as well.

The dataset I use is Princeton Tracking Benchmark from http://tracking.cs.princeton.edu. There are 100 RGBD tracking datasets and 5 of them have the ground truth. The most important factor to choose it is that there are a lot of tracking algorithm performance tested on these datasets so that I can compare my results with those well-known algorithms. Unfortunately, I put too much time on improving my implementation. As a result, I had no time to test my tracker on the test sets.

The images are named r-timestamp-frameID (e.g. r-0-1, r-1233-2), which makes further processing harder. A python script is used to rename these images as frameNumber.png before tracker starts.

Results and Discussions

Unfortunately, a quantitative performance measurement is not developed but only a subjective judgement. The speed is nearly as fast as KCF tracker itself at the resolution of 640x480. However, the accuracy is terrible. The problems of KCF when occlusions happen are not solved. The tracker does not perform as expected to shrink to visible parts at all even if the occlusion is detected.

The other major problem is that the tracker could accidentally start tracking the background and never recover back. And my parameters are set manually rather than set automatically according to tracking task.

There are a lot of future work could be done. First, occlusion detection and handling are implemented already and the detection works well. The only thing needs improvements is handling. Second, diving into the background tracking is necessary while I tried and failed. One of the obvious indicators of this failure is the sudden change of distance. Removing changes too fast might be a solution. Third, adaptive parameters can help tracker work in different tasks. The adaptive parameters in HOG extraction are done but could be improved. Last but not least, multithread processing is used in the original DS-KCF implementation. I was unable to get intel TBB working on my laptop. But it worth trying as it will increase the processing speed dramatically.

The last suggestion is beyond the scope of DS-KCF but more general RGB-D tracking. DS-KCF focus on occlusion but there are two more aspects can be improved using depth information, rotations and deformation.

Conclusion

Depth information can be integrated into RGB tracker to overcome problems when the target rotates, deform or is under occlusion. KCF is a really good tracker and a starting point. DS-KCF purposed a compute-efficient way to make use of the depth information dealing with occlusion. The implementation is hard and the result does not pay for the work. However, it shows the possibility of integrating depth information with the traditional RGB tracker to overcome some problems hard to solve with only color information.

References

Hannuna, S., Camplani, M., Hall, J. et al. J Real-Time Image Proc (2016). https://doi.org/10.1007/s11554-016-0654-3

Wu, Y., Lim, J., Yang, M.: Online object tracking: a benchmark. In: IEEE CVPR, pp. 2411–2418 (2013)

Joao F. H., Rui C., Pedro M, et al. High-Sped Tracking with Kernelized Correlation Filters https://arxiv.org/abs/1404.7584