Final Report of Pedestrain Counting

Wang Zhengrong, Hsienyu Meng, Liuyang Zhan

June 20, 2015

1. **Introduction**

   Pedestrain detection and counting has been heavily researched in the last few years and people have made great improvement. Dalal's HoG (Histogram of Gradients) operator has been widely used in pedestrain detection with SVM, AdaBoost or other machine learning algorithms, which we use as the basic detector in our project.

   However, detection is not enough to count the pedestrains in the ROI, therefore we have to track each person. Our basic idea is to use particle filter to track each specific person.

   Particle filter is an approximation of Bayes inference and is widely used in tracking. Compared with Karman filter, it can simulate any probability distribution. However it's main drawback is the high complexity of computation. Which we will try to optimize with multiple threads.

   This project is hosted as a private project on GitHub. You will find the project page and the documents on it.

2. **Basic Plan**

   Here is our basic plan for this project.

   - Code Reconstruction

     The code offered by the teacher is not object-oriented, and is very difficult to modify and extend. Hence our first goal is to reconstruct the program so that we can easily build our particle filter on it.

   - Merge Paritcle Filter

     The main idea is from [2], in which there are mainly two new ideas. The first one is that instead of using one offline trained general classifier, they train one online classifier for each detected pedestrain and the classifier is only updated on non-overlapping detections. Secondly, the detections are used to guide the particles' propagation which is implemented to estimates the conditional likelihood of the new observation .

     – Data Association Problem Use the greedy algorithm to find the $pair(t_r, d)$ with maximum score in the matching score matrix and delete the columns and rows belonging to tracker t_r^* and d

     – Online Boosting
       The online boosting classifier for each pedestrain is similar to that in [3] and we will select some features to train it.

   - Optimization

     With multiple threads or even GPU programming, we may archieve the real time interactive result.

3. **Code Reconstruction**

   First we reconstruct the code. We left kmeas and meanshift algorithm unchanged cause they are not important in our project. And we divide the whole project into these 5 parts.

   - Utility

     We implement some utility classes here. Mainly some geometry classes such as `Size`, `Rect`, `Point2D`. These are very similar to those in OpenCV library. However we still implement them as sometimes we need overload some operators. We also implement a container called `Pool`, which is basically just a vector that never shrinks, in order to improve performance.

And we also reconstruct the `ConnectedComponents` here. It basically does the same thing as before.

- IntegralImage

  As most of the features will be extracted using integral image to speed up, we implement an `IntegralImage` interface. This is an abstract class containing some virtual functions. The most important method is:

```
// Normal integral image.
virtual unsigned int GetSum(const Rect &roi) const;

// Used in HoG integral image.
virtual void GetSum(const Rect &roi, float *result) const;
```

  Other integral image classes should overload these two functions according to their purpose. Here we mainly implement 3 integral images.

  `GrayScaleIntegralImage` calculates the integral image for a grayscale image. It overloads the first `GetSum` function.

  `HoGIntegralImage` calculates the 9 bins HoG for a grayscale image. Of course this is used to extract the HoG feature.

  `RGIIntegralImage` calculates the integral image for 3 channel: RED, BLUE, INTENSITY.

- FeatureExtractor

  In this part we implement three classes: `Feature`, `HaarFeature`, `HoGFeature`, `RGIFeature`.

  `Feature` is bascially just a container for the feature we extraced using the other two classes.

  `HaarFeature` extracts a haar-like feature given an integral image and roi. When being constructed, it randomly chooses from the five haar-like features.

  `HoGFeature` extracts a HoG feature given an `HoGIntegralImage` and roi.

  `RGIFeature` extracts a RGI feature given an `RGIIntegralImage` and roi.

- Classifier

  Here we reconstruct the original AdaBoost classifier with the following classes.

  First we build an `WeakClassifier` interface and it has two main virtual methods:

```
virtual bool Update(const IntegralImage *intImage,
        const Rect &roi, int target);
virtual float Evaluate(const IntegralImage *intImage,
        const Rect &roi);
```

  `Evaluate` evaluates the roi with the feature inside this weakclassifier, while `Update` is used in training.

  Then we implement a class called `WeakClassifierHoG`. It doesn't overload `Update` method therefore it can't be trained. It's only used in the offline AdaBoost classifier.

  We construct the AdaBoost classifier using `WeakClassifierHoG`.

- Detector

  With the AdaBoost classifier above we are able to build the detectors now.

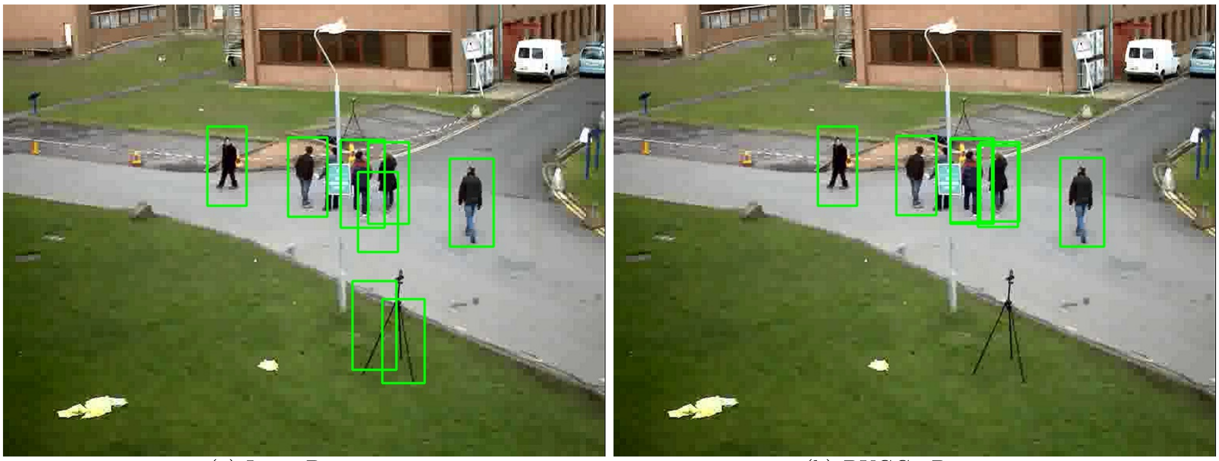
  `ImageDetector` uses the AdaBoost classifier and slide windows to detect pedestrain in the whole image.

  `BKGCutDetector` inherits from `ImageDetector`. It cuts the background and uses the `ConnectedComponents` to speed up the detection. When it is not sure whether a connected component is a pedestrain or not, it calls `ImageDetector` to judge.

  `VideoDetector` receives a pointer of `ImageDetector` and use it to detect pedestrain in every two frames. Notice that with virtual function we can use `BKGCutDetector` here as well.

Besides, while reconstructing the program, we rewrite some parts of the program in a more memory friendly way, which leads to quite tremendous improvement. The original video detector on the first training video takes 212s, while our reconstructed program takes 66s with one main thread. After optimizing some parameters it reduces to 27s without deteriorating its precision.

Here are some results from our reconstruction: Figure 1. We can see that with background cut we have less false positive.



(a) ImageDetector (b) BKGCutDetector

Figure 1: Detection Results

4. **Online Boosting**

After the reconstruction, we start to work on the online boosting algorithm to track a single target. The main work is focus on Classifier part. We implement the following new classes.

- `EstimatedGaussianDistribution`
  Given a feature $f(\mathbf{x})$, the probability of $P(1|f(\mathbf{x}))$ and $P(-1|f(\mathbf{x}))$ is estimated as Gaussian distribution[3]. This Gaussian distribution is estimated with Kalman filter[7]. We use the following update equations for adaptive estimation from [3]:

$$K_t = \frac{P_{t-1}}{P_{t-1}+R} \tag{1a}$$
$$\mu_t = K_t f(\mathbf{x}) + (1-K_t)\mu_{t-1} \tag{1b}$$
$$\sigma_t^2 = K_t(f(\mathbf{x})-\mu_t)^2 + (1-K_t)\sigma_{t-1}^2 \tag{1c}$$
$$P_t = (1-K_t)P_{t-1} \tag{1d}$$

- `ClassifierThreshold`
  It estimates the Gaussian distribution for both positive features $N(\mu_+, \sigma_+)$ and negative features $N(\mu_-, \sigma_-)$. Then it uses a simple distance threshold to a new feature to whether positive or negative: $h(\mathbf{x})$ for "hypothesis"

$$h(\mathbf{x}) = \min_{+,-}(D(f(\mathbf{x}), \mu_+), D(f(\mathbf{x}), \mu_-)) \tag{2}$$

  where $D(f(\mathbf{x}), \mu)$ is just the Euclidean distance in feature space.

- `WeakClassifier`
  It uses some feature above and the `ClassifierThreshold` to build a weak classifier. For classify, it uses `HaarFeature` to extract the feature and sends it to `ClassifierThreshold` to classify. For training, it uses the Kalman filter in `EstimatedGaussianDistribution`.
  Here we will test Haar-like feature and RGI feature for online boosting.

- `ClassifierSelector`
  Given a pool of weak classifiers, the `ClassifierSelector` selects the best one with lowest error rate.
  - Training
    Each training feature $f(\mathbf{x})$ has an importance λ, and we use the idea from [6] to draw a random variable $k \sim Poisson(\lambda)$ and this feature is trained for k times.

– Selecting
For each weak classifier, we maintain two vaiables $\lambda_{correct}$ and λ_{wrong}:

$$\lambda_{correct} = \sum_{i_{correct}} \lambda_i \tag{3a}$$

$$\lambda_{wrong} = \sum_{i_{wrong}} \lambda_i \tag{3b}$$

And the error rate is estimated by:

$$err = \frac{\lambda_{wrong}}{\lambda_{correct} + \lambda_{wrong}} \tag{4}$$

Then we choose the best weak classifier with lowest error rate.

– Replacing
To improve the performance, each time we not only choose the best weak classifier but also replace the worst one with a randomly generated new weak classifier.

- `StrongClassifier`
The `StrongClassifier` has N `ClassifierSelector`s, each with a voting weight α_i. The final hypothesis is:

$$h^{strong}(\mathbf{x}) = \text{sign}(\sum_{i=1}^{N} \alpha_i \cdot h_i^{selector}(\mathbf{x})) \tag{5}$$

Suppose err_i is the error rate of the i^{th} selector, and then the voting weight is:

$$\alpha_i = \ln(\frac{1 - err_i}{err_i}) \tag{6}$$

And the importance of this sample is updated with:

$$\lambda_{i+1} = \lambda_i \cdot \sqrt{\frac{err_i}{1 - err_i}}, \qquad if\ h_i^{selector}\ correct \tag{7a}$$

$$\lambda_{i+1} = \lambda_i \cdot \sqrt{\frac{1 - err_i}{err_i}}, \qquad if\ h_i^{selector}\ wrong \tag{7b}$$

5. **Particle Filter**

After we have the online boosting strong classifier. We try to combine it with particle filter.

- `SingleSampler`
Given a target, it samples around it the positive and negative samples using Gaussian noise. Here is the samples: Figure 2, red for negative samples and blue for positive ones.
This is used in training the classifier.

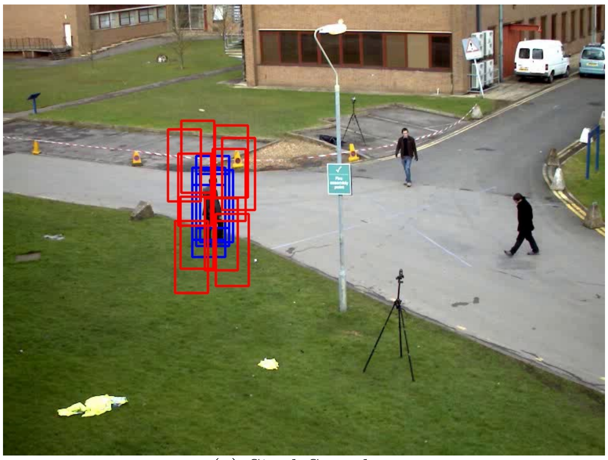
- `ParticleFilterConstVelocity`
This is a basic particle filter with constant velocity. The state space is just the position $[upper, left]$ and the velocity. The motion model is also very simple:

$$p_t = p_{t-1} + v_{t-1} + N(0, \sigma_p) \tag{8}$$

$$v_t = v_{t-1} + N(0, \sigma_v) \tag{9}$$

where $N(0, \sigma_p)$ is a Gaussian random variable with variance proportional to the size of the target, and $N(0, \sigma_v)$ is a Gaussian random vairiable with variance proportional to how many frames in the past has been successfully detected.

- `ParticleFilterTracker`
This class just combines everything together, use particle filter and strong classifier to track a target.



(a) SingleSampler

Figure 2: Detection Results

6. **Match Matrix**

   Since we are trying to track mulitple targets, we will have multiple detections and targets in one frame. In order to solve this data-association problem, we define a match score[2]:

$$s(tr,d) = g(tr,d) \cdot (c_{tr}(d) + \alpha \cdot \sum_{p \in tr}^{N} p_N(d-p)) \tag{10}$$

   The exact meaning of this equation can be found in [2]. We won't talk much about it here.

   A match score matrix will be calculated for each pair of (target, detection). Then a greedy algorithm will be applied to find the match.

   (a) find the maximum match score and set this as a matched pair
   (b) eliminate this target and this detection
   (c) go back to (a) if there are still unmatched detections and targets
   (d) finally only pairs with match score higher than a threshold will be taken

7. **Continous Energy Minimize Method**

   Since the result from HOG detector and particle filter is not very good, therefore we also tried another method from [4][5][1]. It use a Kalman filter or an extended Kalman filter to get a initial tracking result, then minimize an energy function within a temporal window. However this method is not causal and uses information from the future, therefore it can not be realtime.

8. **Experiment**

   All the experiment is executed on Dell Inspiron 14R SE with the following configuration. All the experiment result can be found here.

   - Intel Core i7-3612QM 2.10GHz
   - 8.00GB RAM
   - System: Windows 8.1 x64
   - OpenCV: 3.00

   For video we have three clips: 1.avi, 2.avi, 3.avi.

   For single image detection we only use test.jpg, which is from 1.avi.

   The original baseline program is tested with all default settings.

   With the following configuration we tested our single tracker on a small video clip. The result can be found here.

- The online boosting classifier is built with RGI subpatch feature.
- The particle filter resamples with confidence of each particle.
- 500 particles.
- User initializes the target bounding box.

  We can see that the classifier works quite good in Figure 3. After observation, the particles are drawn with their confidence. The brighter the particle is, the more confidence it has. Notice that the particle is at the upper-left corner of the target.

(a)

Figure 3: Particles Confidence

- Drift problem.

  From Figure 4 we can see that the current grayscale Haar-like classifier can still not distinguish different pedestrains. We think this is due to the grayscale haar-like feature doesn't contain enough information. [2] reports that with RGI (Red, Green, Intensity) histogram feature, 3 bins for each channel the results are quite good.

  Hence we use RGI feature to construct the weak classifier. The resulting video can be found here. From this video and Figure 5 we can see that the classifier is more robustic and can handle some situation as two people crossing each other. However in Figure 5(b) the classifier still goes for wrong pedestrain! Well they wear the same jeans.

(a) Original target (b) Drift to another target

Figure 4: With grayscale haar-like feature, the tracker lost its target.(Black for original target)

- HoG Detector.

  We first use the simple HoG detector to detect a single picture.

(a) Original target (b) Drift to another target

Figure 5: With RGI feature, the tracker is more robustic, but still lost its target finally. (Black for original target)

	Time	Accuracy	False Positive
Original	3.43s	5 / 6	3
Reconstructed	3.20s	5 / 6	2

- Background Cut Detector.

  We use the simple Background cut detector to detect pedestrain in a single picture.

	Time	Accuracy	False Positive
Original	1.76s	6 / 6	0
Reconstructed	1.27s	5 / 6	0

- HoG Detector for Video.

  We use the simple HoG detector to detect pedestrain in a video.

	Time	Accuracy	False Positive(per frame)
Original	1051s	48.9%	1.47
Reconstructed	900s	48.7%	1.41

- Background Cut Detector for Video.

  We use the simple Background Cut detector to detect pedestrain in a video.

	Time	Accuracy	False Positive(per frame)
Original	207.36s	52.7%	0.13
Reconstructed	159.14s	52.4%	0.15

- Multiple Pedestrains Tracking

  This is the finally object of this project. However after two months' hard-working, we still can't get satisfactory result. The result can be found here. We will analyse the results.

  First we can see in Figure 6(a) that when there are no occlution, the detector works fine, and the multiple tracker successfully initializes the three targets and the count them.

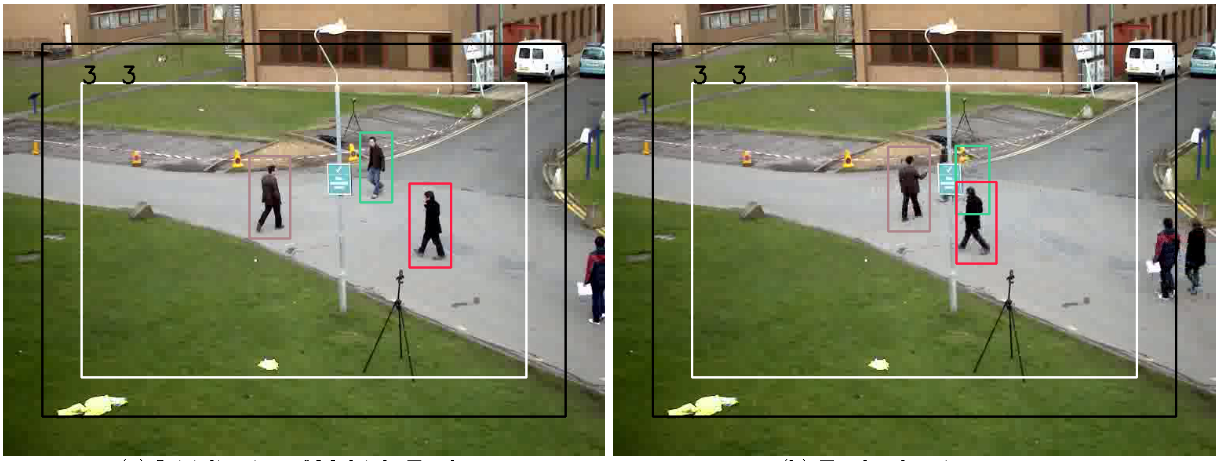
  However when the pedestrain is occluded, the detector failed to detect him. And since we didn't have any occlusion reasoning for this situation, the particle filter soon lost its target, which can been seen in Figure 6(b).

- Energy Minimization

  Figure 7(b) shows the result of energy minimization method. We can see that it's very good. However as discussed above, this method is not causal.

- Pedestrain Counting

  We also test our particle filter method and the energy minimization method on pedestrain counting on PETS 2009 S3MF1. Figure 7 shows the result.



(a) Initialization of Multiple Tracker (b) Tracker lost its target.

Figure 6: Result of Multiple Tracker

(a) MultipleTracker (b) Energy Minimization

Figure 7: CountResult

	Particle Filter	Energy Minimization	Ground Truth
Number of Pedestrain	6	7	7

9. **Summary**

- Accomplishment

  We reconstructed the original baseline Adaboost classifier and improved the performance.

  An online-boosting tracking system has been successfully implemented. It is quite good in tracking for single target. We also tried HoG, Haar-like and RGI features.

- Problems

  Since this is a tracking system based on detection, it requires a state of art detector. However the detector we used here is not so good. If the detector fails to detect the target for consecutive frames, the particle filter will also loses its target.

  What's more, the match matrix is not working very fine. If there are multiple targets nearby, it sometimes gets confused.

  Also since all the detection and tracking are done without depth information, this makes occlution reasoning impossible.

  All of these problems limit the performance of our system.

- Solutions

  The solutions to the above problem is straight forward. Use a state-of-art detector will significantly improve the performance. And also depth information will help a lot.

References

[1] Anton Andriyenko and Konrad Schindler. Multi-target tracking by continuous energy minimization. In *CVPR*, 2011.

[2] Michael D. Breitenstein, Fabian Reichlin, Bastian Leibe, Esther Koller-Meier, and Luc Van Gool. Robust tracking-by-detection using a detector confidence particle filter. In *IEEE International Conference on Computer Vision*, October 2009.

[3] Helmut Grabner and Horst Bischof. On-line boosting and vision. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, CVPR '06, pages 260–267, Washington, DC, USA, 2006. IEEE Computer Society.

[4] A. Milan, S. Roth, and K. Schindler. Continuous energy minimization for multitarget tracking. *IEEE TPAMI*, 36(1):58–72, 2014.

[5] Anton Milan, Stefan Roth, and Konrad Schindler. An analytical formulation of global occlusion reasoning for multi-target tracking. In *Proc. of the 11th International IEEE Workshop on Visual Surveillance*, 2011.

[6] Nikunj C. Oza and Stuart Russell. Online bagging and boosting. In *In Artificial Intelligence and Statistics 2001*, pages 105–112. Morgan Kaufmann, 2001.

[7] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995.