Middle Report of Pedestrain Counting

Wang Zhengrong, Hsienyu Meng, Liuyang Zhan

May 3, 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 [1], 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 [2] 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. **Current Progress**

- 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:

```
1  // Normal integral image.
2  virtual unsigned int GetSum(const Rect &roi) const;
3
4  // Used in HoG integral image.
5  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 two 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.

– FeatureExtractor

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

`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.

– 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:

```
1  virtual bool Update(const IntegralImage *intImage,
2          const Rect &roi, int target);
3  virtual float Evaluate(const IntegralImage *intImage,
4          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.

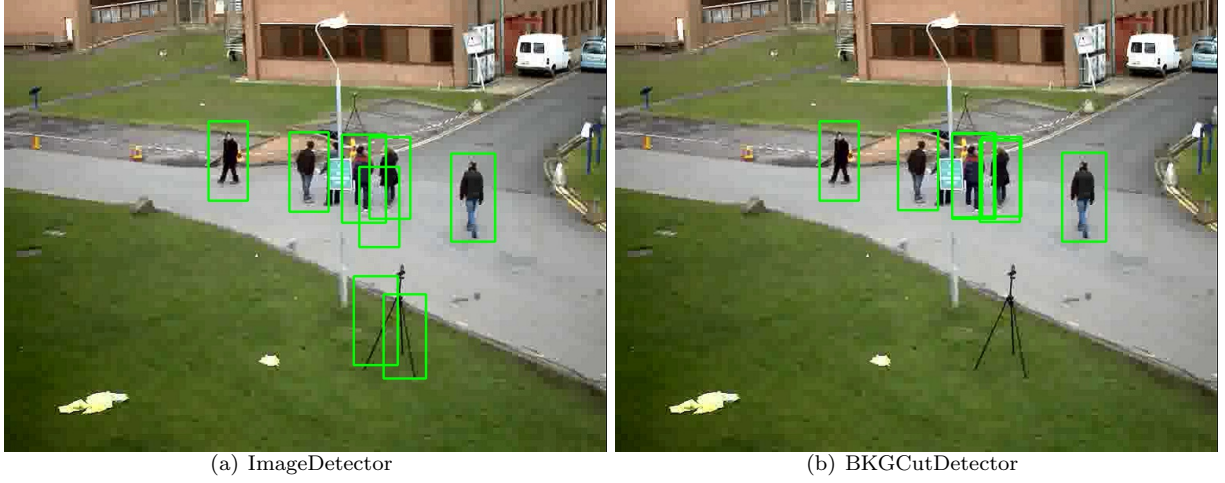(a) ImageDetector   (b) BKGCutDetector

Figure 1: Detection Results

- 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[2]. This Gaussian distribution is estimated with Kalman filter[4]. We use the following update equations for adaptive estimation from [2]:

$$K_t = \frac{P_{t-1}}{P_{t-1} + R} \tag{1a}$$

$$\mu_t = K_t f(\mathbf{x}) + (1 - K_t)\mu_{t-1} \tag{1b}$$

$$\sigma_t^2 = K_t(f(\mathbf{x}) - \mu_t)^2 + (1 - K_t)\sigma_{t-1}^2 \tag{1c}$$

$$P_t = (1 - K_t)P_{t-1} \tag{1d}$$

  - 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.

  - WeakClassifierHaar

    It uses the Haar-like 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.

  - 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 [3] to draw a random variable $k \sim Poisson(\lambda)$ and this feature is trained for k times.

    * Selecting

3

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 ith selector, and then the voting weight is:

$$\alpha_i = \frac{1}{2} \cdot \ln(\frac{1 - err_i}{err_i}) \tag{6}$$

And the importance of this sample is updated with:

$$\lambda_{i+1} = \lambda_i \cdot \frac{1}{2 \cdot (1 - err_i)}, \qquad\qquad if\ h_i^{selector}\ correct \tag{7a}$$

$$\lambda_{i+1} = \lambda_i \cdot \frac{1}{2 \cdot err_i}, \qquad\qquad if\ h_i^{selector}\ wrong \tag{7b}$$

- 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.

   – `ParticleFilter`
   This is a basic particle filter. The state space is just the position $[upper, left]$. The motion model is also very simple:

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

where $N(0, \sigma)$ is a Gaussian random variable with variance proportional to the size of the target.
As for observation, it just use the scores given by strong classifier as the weight of the particles and resample it. We may improve the motion model later.

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

4. **Further Plan**
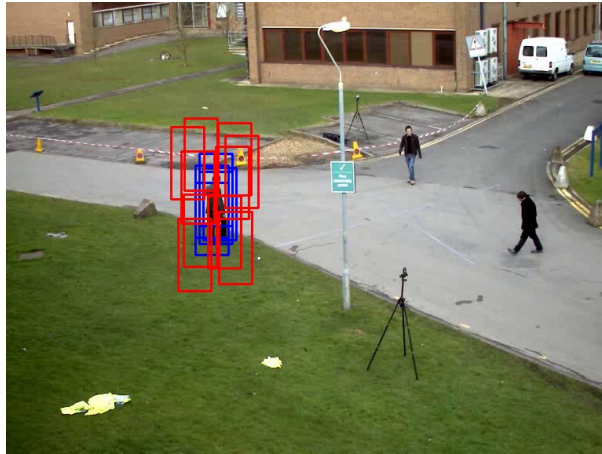
(a) SingleSampler

Figure 2: Detection Results

References

[1] 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.

[2] 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.

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

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