CMSC 25400 Homework 4

Yun Dai

February 11 2019

1 Data, Environment, Packages and How to Run

1.1 Data

All of the 4000 examples, i.e. 2000 faces and 2000 backgrounds are included in my implementation. The training data should be stored in `/data/faces/` and `/data/background/` respectively by default. The test data I use is the standard testing image provided by the instructor, and should be saved as `/data/test_img.jpg`

1.2 Environment and Packages

This project is built upon python 3.6. External packages used include `numpy`, `skimage` (load the image data as matrix), `matplotlib`, `tqdm` (for display of process bar – long waiting is not funny), and `click` (for interactive running via command line). So please be sure to install these packages before running the program.

1.3 How to Run

Be sure the directory of the program contains following files:

```
.
├── adaboost.py
├── 20190211_15/51.cascade
├── data
│   ├── background
│   │   ├──0.jpg
│   │   ├──1.jpg
│   │   ├──...
│   ├── faces
│   │   ├──0.jpg
│   │   ├──1.jpg
│   │   ├──...
│   └── test_img.jpg
├── haar.py
├── mytype.py
├── predict.json
├── train.json
└── util.py
```

First, to train the model, specify the number of cores for multiprocessing in `train.json` (default value is 4). Then run the following command in terminal (you can also add argument `-v` for verbose log information):

```
python3 adaboost.py -t
```

This should produce a `.cascade` file that stores the final model with file name specifying the time when the model is built. Then, to predict on the test image using the trained model, you should firstly modify the `model_path` config in `predict.json` to the model file just produced. I also include the final model

as `20190210_14/59.cascade`. The path of the image being predicted should be clarified in the `test_file` config. Finally, run the following command to predict on the test image:

```
python3 adaboost.py -p
```

2 Parameters

The important parameters in the algorithm are assigned in the `train.json` and the `predict.json` files.

2.1 Training

For the parameters in training process, r_stride and c_stride specify the stride by which we collect haar-like features and I set them to 2. Similarly, $zoom = 2$ specifies the pixel by which the width and height of rectangles are enlarged.

The min_size and max_size denote the minimum and maximum length of the edge of a Haar-like feature rectangle, and the algorithm produces the best testing result when $min\_size = 2$ and $max\_size = 24$. The purpose of limiting the size of Haar-like feature is that, if without limitation, there will be an extremely large number of features, i.e. a high model complexity while the data set is quite small (only 4000 examples in total) and hence the model will be very likely to overfit. Reducing the number of available features will greatly reduce the chance of overfitting. Since Haar-like features that are too large are not very informative, it will be better to discard large features instead of small ones.

During the process of training a single layer of cascade, I set the threshold of false positive rate to 0.3, which means the algorithm will stop adding features (i.e. weak learners) to a layer when the integrated false positive rate of the learners in this layer reaches below 0.3 and the false negative rate reaches below 0.01, or error rate is 0. Since in some cases the FPR will be stuck in a value larger than 0.3 for a long time, I restrict the maximum number of iterations per layer, the $niter$, to be 40.

The whole cascade will be finished when either 1) the number of layer reaches 10, or 2) the chained FPR and FNR converge (change less than 0.01). After testing on different values of Θ, I found the minimum value of Θ that can ensure a zero FNR in training set is approximately 0.4, and set it to be the final value of $thres$. An alternaive of setting the threshold Θ, is setting it adaptively based on the ratio between positive and negative labels during training process on each layer. The $adaptive\_thres$ parameter specifies whether or not switch to this alternative setting.

Another important parameter during training process is the penalty on negative examples, $penalty\_on\_neg$, is set to be 0.5, which means that for each round of boosting, the weight of negative examples will be increased by the following formula

$$w_- \leftarrow (\max\{0, \frac{N_+}{N_-} - 1\} \times penalty\_on\_neg + 1) \times w_-$$

in which w_- denotes the weight of a negative example before normalization, and N_+, N_- indicate the number of positive and negative examples respectively. The purpose of increasing the weight of negative examples is that, after each round of discarding negative examples, the samples will be unbalanced as the total number of negative examples decreases, resulting in an unsatisfactory FPR and extremely high recall rate, which can be easily solved by increasing the weight of negative examples.

2.2 Prediction

The major parameters during prediction process are about merging similar results. The $merge\_thres = 20$ parameter indicates that predicted boxes with relative distance less than 20px will be considered as a potential cluster, and the $cluster\_thres = 3$ means that only clusters with 3 or more boxes will be finally merged.

3 Modules and Functions

In this section, I will introduce my detailed implementation of the Viola Jones algorithm by modules.

3.1 Adaboost

`adaboost.py` is the main module of the program which includes functions in the process of training and predicting.

3.1.1 Adaboost Class

The Adaboost class contains functions for training a single layer of the cascade classifier:

opt_p_theta(): compute the optimal p and θ values for for the weak learner corresponding to a fixed feature index following the method provided at slide p.28. In addition to the standard algorithm that needs to compute the S^+, S^-, T^+, T^- for each sorted example, I come up with an alternative implementation that can compute the p and θ based on the same rules but with much higher speed: first calculate the accumulative sum of signed sorted weights, then the feature values of examples that reach the minimum ($\mathbf{x}_{\sigma(n)}$) and the maximum ($\mathbf{x}_{\sigma(m)}$) will just be the optimal θ^+ and θ^-. If the absolute value of accumulative sum at $\sigma(n)$ is larger than that at $\sigma(m)$, then the optimal $\theta = \theta^+$ and $p = 1$, else $\theta = \theta^-$ and $p = -1$. I found it hard to mathematically prove the validity of this alternative implementation, but it turns out to be valid after 10,000 random testing. I also implemented the standard algorithm just to be safe and sure.

_opt_weaklearner() and **_opt_weaklearner_worker()**: train a weak learner for each Haar-like feature, calculate the corresponding weighted error rate, and find the optimal feature in one round of boosting with the lowest error rate. The worker function is the function that does the exact computation, and the `_opt_weaklearner()` function executes it in multiprocessing way.

_update_data_weights(): update the weights of existing examples based on the given update rule, and add penalty on negative examples based on the proportion of number of positive and negative examples (see 2.1).

_weak_predict() and **predict()**: `_weak_predict()` function implements the prediction on training set based on one weak learner, and the `predict()` function predicts on the training set based on the model constructed so far.

compute_metrics(): compute the major metrics of the model, including false positive rate, false negative rate, recall rate and accuracy rate.

train(): the main function that runs the training process integrating the functions above. Utilizes shared memory and multiprocessing to leverage speed.

3.1.2 CascadeClassifier Class

The CascadeClassifier class contains functions for training the whole cascade classifier:

train(): the main function that trains the whole cascade classifier. It updates the negative examples and train a single layer that contains a number of weak learners using `Adaboost.train()` iteratively until the chained false positive rate reaches 0.01 or the number of layers reaches 10.

_predict(): uses the cascade classifier to make prediction on a single 64×64 subwindow of image.

pred_whole_img() and **_pred_whole_worker()**: Itereate through the whole test image via a 64×64 sliding window and predict if the image within the window contains a face. The worker function is the function that does the exact prediction, and the `pred_whole_img()` function executes it in multiprocessing way.

3.2 Haar

`haar.py` contains the functions for calculating integral image, creating Haar-like feature list and calculating the values of Haar-like features. I implemented two-, three- and four-rectangle features. For faster computation, I store the positions to add and the positions to substract in each feature list instead of following the homework guide.

3.3 Mytype

`mytype.py` is a helper module for better development and display purpose. It encapsulates weak learners, figure coordinates, models, model metrics and Haar-like features into classes.

3.4 Util

`util.py` contains several utility functions about loading data, merging predicted boxes and drawing boxes on the test image.

4 Results

Figure 1. Prediction Result

Table 1. Model Result

Layer	# of Weak Learners	FPR	FNR	Runtime (seconds)
1	2	0.219	0.0075	405
2	8	0.284	0.0095	888
3	13	0.242	0.0085	1201
4	12	0.267	0.001	874

In total 259245 Haar-like features are considered during the training process. As we can see from Table 1, the training process of each stage of the cascade is generally finished within a dozen of boostings. Since I used multiprocessing and optimize the speed of the algorithm via vectorization, the time it took to train the classifier is not very long. Though the model exhibits a good performance on training data, it turns out to overfit on the test image as can be seen from Figure 1: it detects the faces well, but labels other things as faces as well. One of the most interesting things we can discover is that the detector makes many predictions on the sweater of the person at the bottom-right corner. This is probably because the weak learners on Haar-like features with relatively small size (e.g. 2×4 pixels) occupy large weights in the whole classifier.

Though I have tried various methods to optimize the algorithm and parameters, the result above has been the best one I have got. I think the key to a better result is to add more examples, especially negative examples, to the training set. Since the complexity of our hypothesis class is very high (there are hundreds of thousands of features) whereas small size of training data, the model will inevitably overfit to some degree. Also, since the number of available negative examples will decrease after training on one stage of the cascade, we generally need more negative examples than positive ones to make sure the training samples are relatively balanced. The author of the Viola Jones algorithm used 4916 face images and 9544 background images (Viola and Jones, 2001, pp. 6), which might further prove the feasibility of adding more examples especially background examples.

References

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on (Vol. 1, pp. I-I). IEEE.