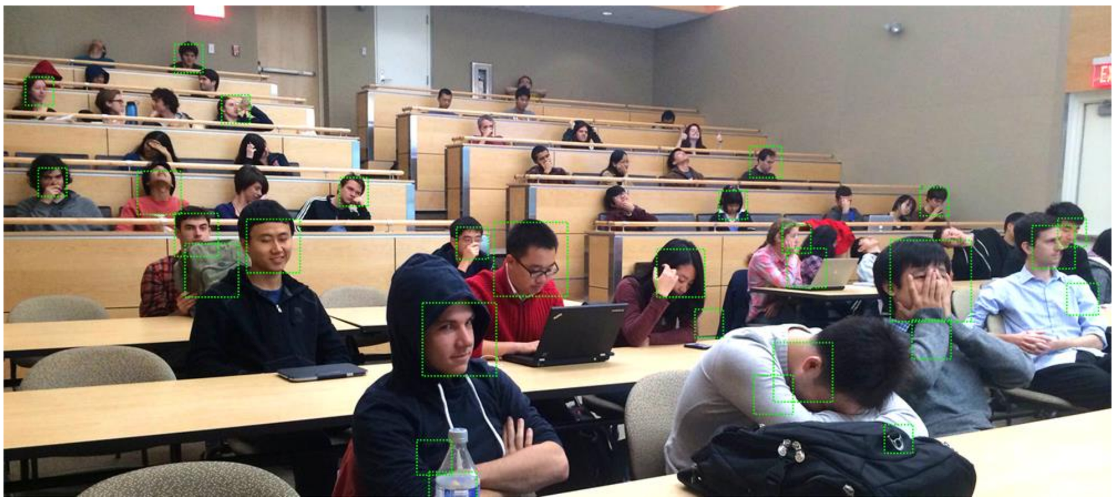
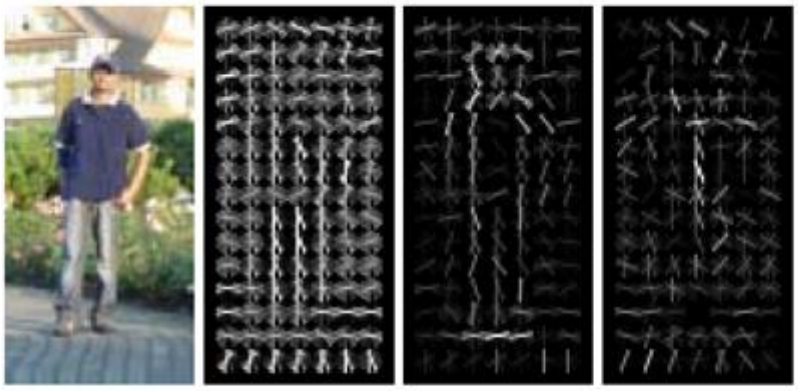

Example face detection results from this project. We're intentionally trying to foil the face detector in this photo.

Project 4: Face detection with a sliding window

Overview

The sliding window model is conceptually simple: independently classify all image patches as being object or non-object. Sliding window classification is the dominant paradigm in object detection and for one object category in particular -- faces -- it is one of the most noticeable successes of computer vision. For example, modern cameras and photo organization tools have prominent face detection capabilities. These success of face detection (and object detection in general) can be traced back to influential works such as Rowley et al. 1998 and Viola-Jones 2001. You can look at these papers for suggestions on how to implement your detector. However, for this project you will be implementing the simpler (but still very effective!) sliding window detector of Dalal and Triggs 2005. Dalal-Triggs focuses on representation more than learning and introduces the SIFT-like Histogram of Gradients (HoG) representation (pictured to the right). Because you have already implemented the SIFT descriptor, you will not be asked to implement HoG. You will be responsible for the rest of the detection pipeline, though -- handling heterogeneous training and testing data, training a linear classifier (a HoG template), and using your classifier to classify millions of sliding windows at multiple scales. Fortunately, linear classifiers are compact, fast to train, and fast to execute. A linear SVM can also be trained on large amounts of data, including mined hard negatives.



Details and Starter Code

The following is an outline of the stencil code:

- *proj4.m*. The top level script for training and testing your object detector. If you run the code unmodified, it will predict random faces in the test images. It calls the following functions, many of which are simply placeholders in the starter code:
- *get_positive_features.m* (you code this). Load cropped positive trained examples (faces) and convert them to HoG features with a call to vl_hog.
- *get_random_negative_features.m* (you code this). Sample random negative examples from scenes which contain no faces and convert them to HoG features.
- classifier training (you code this). Train a linear classifier from the positive and negative examples with a call to *vl_trainsvm.*
- *run_detector.m* (you code this). Run the classifier on the test set. For each image, run the classifier at multiple scales and then call non_max_supr_bbox to remove duplicate detections.
- *evaluate_detections.m*. Compute ROC curve, precision-recall curve, and average precision. You're not allowed to change this function.
- *visualize_detections_by_image.m*. Visualize detections in each image. You can use *visualize_detections_by_image_no_gt.m* for test cases which have no ground truth annotations (e.g. the class photos).

Creating the sliding window, multiscale detector is the most complex part of this project. It is recommended that you start with a single scaledetector which does not detect faces at multiple scales in each test image. Such a detector will not work nearly as well (perhaps 0.3 average precision) compared to the full multi-scale detector. With a well trained multi-scale detector with small step size you can expect to match the papers linked above in performance with average precision above 0.9.

Data

The choice of training data is critical for this task. While an object detection system would typically be trained and tested on a single database (as in the Pascal VOC challenge), face detection papers have traditionally trained on heterogeneous, even proprietary, datasets. As with most of the literature, we will use three databases: (1) positive training crops, (2) non-face scenes to mine for negative training data, and (3) test scenes with ground truth face locations.

You are provided with a positive training database of 6,713 cropped 36x36 faces from the Caltech Web Faces project. We arrived at this subset by filtering away faces which were not high enough resolution, upright, or front facing. There are many additional databases available For example, see Figure 3 in Huang et al. and the LFW database described in the paper. You are free to experiment with additional or alternative training data for extra credit.

Non-face scenes, the second source of your training data, are easy to collect. We provide a small database of such scenes from Wu et al. and the SUN scene database. You can add more non-face training scenes, although you are unlikely to need more negative training data unless you are doing hard negative mining for extra credit.

The most common benchmark for face detection is the CMU+MIT test set. This test set contains 130 images with 511 faces. The test set is challenging because the images are highly compressed and quantized. Some of the faces are illustrated faces, not human faces. For this project, we have converted the test set's ground truth landmark points in to Pascal VOC style bounding boxes. We have inflated these bounding boxes to cover most of the head, as the provided training data does. For this reason, you are arguably training a "head detector" not a "face detector" for this project.

Copies of these data sets are provided with your starter code and are available in *proj4/data*. You probably want to make a local copy of these to speed up training and testing, but please do not include them in your handin.

Write up

For this project, and all other projects, you must do a project report. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm. Discuss any extra credit you did, and clearly show what contribution it had on the results (e.g. performance with and without each extra credit component).
You should show how your detector performs on additional images in the data/extra_test_scenes directory.
You should include the precision-recall curve of your final classifier and any interesting variants of your algorithm.

Final Advice

- The starter code has more specific advice about the necessary structure of variables through the code. However, the design of the functions is left up to you. You may want to create some additional functions to help abstract away the complexity of sampling training data and running the detector.
- You probably don't want to run non-max suppression while mining hard-negatives (extra credit).
- While the idea of mining for hard negatives is ubiquitous in the object detection literature, it may only modestly increase your performance when compared to a similar number of random negatives.
- The parameters of the learning algorithms are important. The regularization parameter lambda is important for training your linear SVM. It controls the amount of bias in the model, and thus the degree of underfitting or overfitting to the training data. Experiment to find its best value.
- Your classifiers, especially if they are trained with large amounts of negative data, may "underdetect" because of an overly conservative threshold. You can lower the thresholds on your classifiers to improve your average precision. The precision-recall metric does not penalize a detector for producing false positives, as long as those false positives have lower confidence than true positives. For example, an otherwise accurate detector might only achieve 50% recall on the test set with 1000 detections. If you lower the threshold for a positive detection to achieve 70% recall with 5000 detections your average precision will increase, even though you are returning mostly false positives.
- When coding run_detector.m, you will need to decide on some important parameters. (1) The step size. By default, this should simply be the pixel width of your HoG cells. That is, you should step one HoG cell at a time while running your detector over a HoG image. However, you will get better performance if you use a fine step size. You can do this by computing HoG features on shifted versions of your image. This is not required, though -- you can get very good performance with sampling steps of 4 or 6 pixels. (2) The step size across scales, e.g. how much you downsample the image. A value of 0.7 (the image is downsampled to 70% of it's previous size recursively) works well enough for debugging, but finer search with a value such as 0.9 will improve performance. However, making the search finer scale will slow down your detector considerably.
- Likewise your accuracy is likely to increase as you use more of the training data, but this will slow down your training. You can debug your system with smaller amounts of training data (e.g. all positive examples and 10000 negative examples).
- You can train and test a classifier with average precision of 0.85 in about 60 seconds. It is alright if your training and testing is slower, though.
- The Viola-Jones algorithm achieves an average precision of 0.895* on the CMU+MIT test set based on the numbers in Table 3 of the paper (This number may be slightly off because Table 3 doesn't fully specify the precision-recall curve, because the overlap criteria for VJ might not match our overlap criteria, and because the test sets might be slightly different -- VJ says the test set contains 507 faces, whereas we count 511 faces). You can beat this number, although you may need to run the detector at very small step sizes and scales. We have achieved Average

Precions around .93.