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

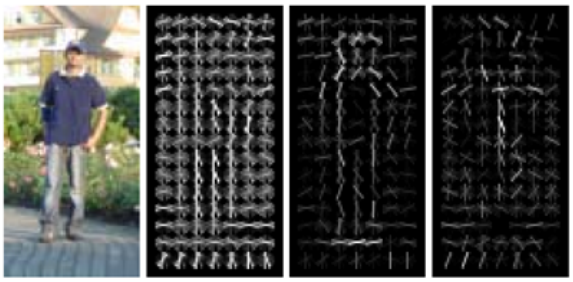
Project 5: Face detection with a sliding window

CS 6476: Computer Vision

Brief

- Due date: Wednesday, November 14th, 11:55pm
- Project materials including the html writeup template: **proj5.zip (77KB)**.
- Training and testing data: **data.zip (93MB)**.
- Handin: through **Canvas**
- Required files: README, code/, html/, html/index.html

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.

Setup

Please reinstall your conda environment as we have made changes to the list of packages installed.

1. Install **Miniconda**. It doesn't matter whether you use 2.7 or 3.6 because we will create our own environment anyways.
2. Create a conda environment using the appropriate command. On Windows, open the installed "Conda prompt" to run this command. On MacOS or Linux, you can just use a terminal window to run the command. Modify the command based on your OS (`linux`, `mac`, or `win`).

`conda env create -f environment_<OS>.yml`

3. This should create an environment named `cs6476p5`. Activate it using the following Windows command:
   `activate cs6476p5`
   or the following MacOS/Linux command:
   `source activate cs6476p5`
4. Run the notebook using:
   `jupyter notebook ./code/proj5.ipynb`
5. Generate the submission once you've finished the project using:
   `python zip_submission.py`

Details

For this project, you need to implement the following parts:

- Extracting features: `get_positive_features()`, `get_random_negative_features()`
- Mining hard negatives: `mine_hard_negs()`
- Train a linear classifier: `train_classifier()`
- Detect faces on the test set: `run_detector()`

Get features from positive samples

You will implement `get_positive_features()` to load cropped positive trained examples (faces) and convert them to HoG features. We provide the default feature parameters in `proj5.ipynb`. (You are free to try different parameters, but it doesn't guarantee better performance.) For improved performance, You can try mirroring or warping the positive training examples to augment your training data. Please refer to the documentation for more details.

Get features from negative samples

You will implement `get_random_negative_features()` to sample random negative examples from scenes which contain no faces and convert them to HoG features. The output feature dimension from `get_random_negative_features()` and `get_positive_features()` should be the same. For best performance, you should sample random negative examples at multiple scales. Please refer to the documentation for more details.

Mine hard negatives

You will implement `mine_hard_negs()` as discussed in **Dalal and Triggs**, and demonstrate the effect on performance. The main idea of hard negative mining is that you use the trained classifier to find false-positive examples, and then include them in your negative training data, so you can train the classifier again to improve the performance. This might not be very effective for frontal faces unless you use a more complex feature or classifier. You might notice a bigger difference if you artificially limit the amount of negative training data (e.g. a total budget of only 5000 negatives).

Train a linear classifer

You will implement `train_classifier()` to train a linear classifier from the positive and negative features by using **scikit-learn LinearSVC**. The regularization constant C is an important parameter that affects the classification performance. Small values seem to work better e.g. 1e-4, but you can try other values.

Detect faces on the test set

You will implement `run_detector()` to detect faces on the testing images. For each image, You will run the classifier with a sliding window at multiple scales and then call the provided function `non_max_suppression_bbox()` to remove duplicate detections.

Your code should convert each test image to HoG feature space for each scale. Then you step over the HoG cells, taking groups of cells that are the same size as your learned template, and classifying them. If the classification is above some confidence, keep the detection and then pass all the detections for an image to `non_max_suppression_bbox()`. The ouputs are the coordinates ([xl, yl, xh, yh]) of all the detected faces and their corresponding confidences and testing image indices. Please See `run_detector()` documentation for more details.

`run_detector()` will have (at least) two parameters which can heavily influence performance:

- how much to rescale each step of your multiscale detector: More scales usually help.
- threshold for a detection: If your recall rate is low and your detector still has high precision at its highest recall point, you can improve your average precision by reducing the threshold for a positive detection.

Using the starter code (`proj5.ipynb`)

The top-level starter code `proj5.ipynb` provides data loading, evaluation and visualization functions and calls the functions in `student_code.py`. If you run the code unmodified, it will return random bounding boxes in each test image. It will even do non-maximum suppression on the random bounding boxes to give you an example of how to call the function. detect random faces in the test images. We provide the following functions to help you examine the performance:

- `report_accuracy()`: Show some statistics about your trained classifier. The training accuracy should be really low. Good classification performance doesn't guarantee good detection performance, but it's a good sanity check. We also provide the visualization that you can see

how well separated the positive and negative examples are at training time.

- `visualize_hog()`: Visualize the HOG feature template to examine if the detector has learned a meaningful representation.
- `evaluate_detections()`: Compute ROC curve, precision-recall curve, and average precision. You're not allowed to change this function.

  Performance to aim for:
  - random (stater code): 0.001 AP
  - single scale: ~ 0.3 to 0.4 AP
  - multiscale: ~ 0.75 to 0.9 AP

- `visualize_detections_by_image()`: Visualize detections in each image (with ground truth). You also can use `visualize_detections_by_image_no_gt()` for test cases which have no ground truth annotations (e.g. the class photos).

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 were previously 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.

Please do *not* include the data sets in your handin.

Useful Functions

`vlfeat.hog.hog()`. The main function to extract the HOG feature. For detailed information, please refer to the link or the comments in `student_code.py` and `proj5.ipynb`.

`sklearn.svm.LinearSVC()`. Linear support vector classification. The regularization constant C is the most critical parameter affecting the classification performance.

Banned Functions

You should not use any third-party library that do multi-scale object detection, such as `cv2.HOGDescriptor().detectMultiScale()`. You need to implement the detection function by yourself. You may also not use anyone else's code. The main principle is that you should not use any third-party library that can directly perform one of these functions: `get_positive_features()`, `get_random_negative_features()`, `mine_hard_negs()`, `train_classifier()` or `run_detector()`. If you still have quetions, feel free to ask on Piazza.

Write up

For this project, and all other projects, you must do a project report in HTML. 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 (e.g. if you want to discuss one parameter that you have experiments, we expect you show how this parameter affects the performance with the support of your experiment results, and some descriptions about how and why). Particularly, we want to see the effect of the 'C' parameter in SVM, multi-scale detection and hard negative mining. 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).

We suggest you show your face detections on the class photos in the `data/extra_test_scenes` directory.

You should include the precision-recall curve of your final classifier and the visualization of the learned hog template, and any interesting variants of your algorithm. **Do not include code or your GTID in your report.**

Extra Credit

For all extra credit, be sure to analyze on your web page whether your extra credit has improved classification accuracy. Each item is "up to" some amount of points because trivial implementations may not be worthy of full extra credit. You will **not** get the full extra credit if you only implement without enough analyses in your report. Some ideas:

- up to 10 pts: Implement a HoG descriptor yourself.
- up to 10 pts: Implement a cascade architecture as in Viola-Jones. Show the effect that this has on accuracy and run speed. Describe your cascade building process in detail in your handout. Unfortunately, with the current starter code this is unlikely to improve run speed because the run time is dominated by image and feature manipulations, not the already fast linear classifier. Your cascade probably needs to start with simpler (faster) features.
- up to 10 pts: Detect additional object categories. You'll need to get your own training and testing data. One suggestion is to train and run your detector on the **Pascal VOC** data sets, possibly with the help of their support code. The bounding boxes returned by the stencil code are already in VOC format.
- up to 10 pts: Interesting features and combinations of features. Be creative!
- up to 10 pts: Improve the training data. There are several ways to do that, such as finding and utilizing alternative positive training data, augmenting the provided training data, or extracting negative examples in multi-scale, etc.
- up to 10 pts: Use additional classification schemes (e.g. full decision trees, neural nets, deep convolutional networks, or nearest neighbor methods).
- up to 10 pts: Add contextual reasoning to your classifier. For example, one might learn likely locations of faces given scene statistics, in the spirit of **Contextual priming for object detection, Torralba**. You could try and use typical arrangements of groups of faces as in **Understanding Images of Groups of People** and **Finding Rows of People in Group Images** by Gallagher and Chen.
- up to 10 pts: Use deformable models instead of fixed templates as in the work of **Felzenszwalb et al.**
- up to 10 pts: Implement the **tiny face detector** from CMU.

Finally, there will be extra credit and recognition for the students who achieve the highest average precision. You aren't allowed to modify `evaluate_detections()` which measures your accuracy.

Web-Publishing Results

All the results for each project will be put on the course website so that the students can see each other's results. In class we will highlight the best projects as determined by the professor and TAs. If you do not want your results published to the web, you can choose to opt out.

Handing in

This is very important as you will lose points if you do not follow instructions. Every time after the first that you do not follow instructions, you will lose 5 points. The folder you hand in must contain the following:

- README - text file containing anything about the project that you want to tell the TAs
- code/ - directory containing all your code for this assignment
- html/ - directory containing all your html report for this assignment, including images (any images not under this directory won't be published)
- html/index.html - home page for your results

Hand in your project as a zip file through **Canvas**. This zip must be less than 5MB. If images are taking up too much space, you can use things like imagemagick to shrink them. **Please Note that when grading, we will not be using your notebook, so make sure extra credit is thoroughly documented in the README, and the function signatures and return values in student_code.py are unchanged.**

Rubric

- +20 pts: Use the training images to create positive and and negative training HoG features.
- +10 pts: Mine hard negatives.
- +5 pts: Train linear classifier.
- +45 pts: Create a multi-scale, sliding window object detector.
- +20 pts: Writeup with design decisions and evaluation.
- +10 pts: Extra credit (up to ten points)
- -5*n pts: Lose 5 points for every time (after the first) you do not follow the instructions for the hand in format

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.
- Creating the sliding window, multiscale detector is the most complex part of this project. It is recommended that you start with a *single scale* detector 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.

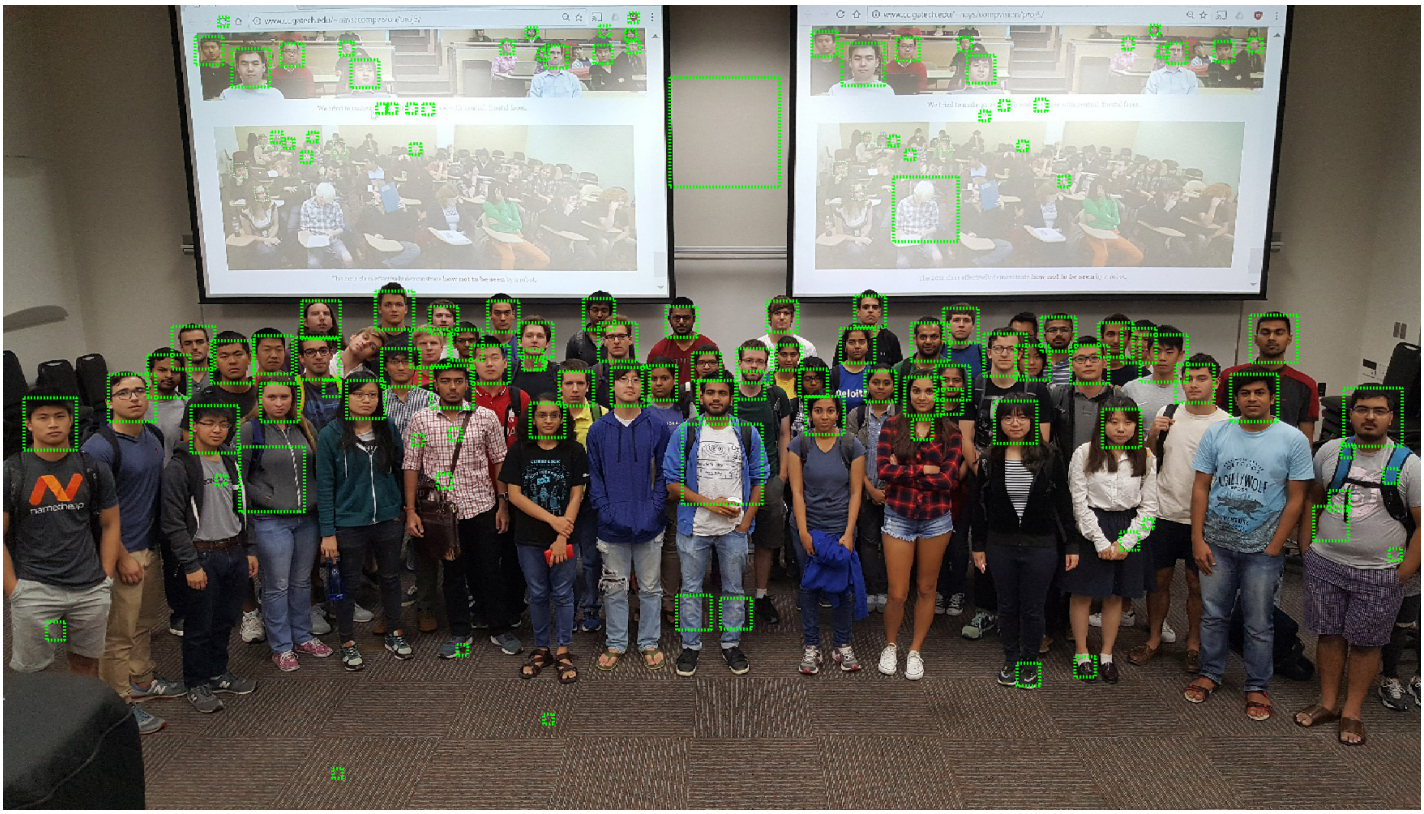
- You probably don't want to run non-max suppression while mining hard-negatives.
- While the idea of mining for hard negatives is ubiquitous in the object detection literature, it may only modestly increase your performance when compared to a similar number of random negatives.
- The parameters of the learning algorithms are important. The regularization parameter C 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()`, 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, as long as the total runtime does not exceed 10 minutes. If you are doing extra credit that is slower that is fine, but turn in code that runs with sufficient accuracy in less than 10 minutes.
- 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.

Credits

Project description and code by James Hays, and has been expanded and edited by Samarth Brahmbhatt, Amit Raj and Min-Hung (Steve) Chen. Figures in this handout are from **Dalal and Triggs**. Thanks to Jianxin Wu and Jim Rehg for suggestions in developing this project.

We tried to make an especially easy test case with neutral, frontal faces.

The 2011 class effectively demonstrate **how not to be seen** by a robot.



Fall 2016 faces detected by Wenqi Xian.