Computer Vision
CSCI-UA.0480-002
Assignment 4.

April 19, 2014

Introduction

This assignment looks object recognition and image segmentation.

The assignment contains two questions plus a bonus question:

1. Boosted object detector – explore the object recognition system covered in lecture 20. [*40 points*].

2. Mean shift image segmentation – implement the algorithm described in lecture 24. [*40 points*].

3. Normalized cuts image segmentation – also described in lecture 24. [*Bonus question – 40 points*].

Requirements

You should perform this assignment in Matlab. If you are not familiar with Matlab, I suggest you go through some of the tutorials posted on the course web page.

This assignment is due on **Thursday May 15th**. The late policy is posted on the course webpage. Youare strongly encouraged to start the assignment early and don't be afraid to ask for help.

The grader for the class is Jiali Huang (`jh3602@nyu.edu`). If you think the assignment is not clear, or there is an error/bug in it, please contact me.

You are allowed to collaborate with other students in terms discussing ideas and possible solutions. However you code up the solution yourself, i.e. you must write your own code. Copying your friends code and just changing all the names of the variables is not allowed! You are not allowed to use solutions from similar assignments in courses from other institutions, or those found elsewhere on the web.

Your solutions should be emailed to me (`fergus@cs.nyu.edu`) and the grader as a single zip file, with the filename: `lastname_firstname_a4.zip`. This zip file should contain: (i) a PDF file `lastname_firstname_a4.pdf` with your report, showing output images for each part of the assignment and explanatory text, where appropriate; (ii) any source code that you have written, or edited.

1 Boosted Object Detector

This question asks you to walk through the boosting-based object recognition system described in lecture 20. This system is described more fully at `http://people.csail.mit.edu/torralba/shortCourseRLOC/boosting/boosting.html`; this link provides an explanation of the code and instructions on how to run it. However, the code for this system and two simple datasets are provided in the file `hw4_qu1.zip` on the course webpage. Please use this version (rather than the one from the webpage), since it contains a few changes to ensure compatibility with a wider range of platforms.

1.1 Preliminaries

Setup: Download and unzip this file into a sub-directory somewhere on your machine and set your current Matlab path to this directory. Then open up the file `initpath.m` and modify the path to reflect the location you unzipped the files to. Do likewise for `parameters.m`, editing the `BASE_DIR` string.

Simple Demo: If you've done all the above directly, then you should be able to start the simple boosting demo with the command `demoGentleBoost`. Put down some positive/negative points by clicking with left/right mouse buttons, then press any key to start the boosting procedure. The boosting weighs are indicated by the size of the circle and square markers. Initially these start off uniform, but as boosting proceeds they alter to reflect misclassifications by the strong classifier. The left subplot shows the weak learner

selected at the current round, while the right subplot shows the strong classifier which combines the weaker learners from previous rounds. After a dozen or so rounds, the strong classifier should do a reasonable job of separating the two classes (depending on the points your laid down). Run for 20 rounds and save the figure and put it in your report.

1.2 Screen detector

We will now use train an object detector using the same boosting framework. The zip file contains data for both computer monitors and cars classes. In `parameters.m` you can switch between them (see top of file), but for now we will stick with the default computer screens.

Creating the dictionary of features: We will now create the vocabulary of patches used to compute the features by running the `createDictionary` function. This takes a few images and selects at random patches within the bounding box of the object instances (see first figure). The patches are shown in the second figure. The boosting framework will select a combination of these to use to build a strong classifier, each feature being used as a correlation template as part of a weak classifier.

Computing Features: Before running the actual training procedure, we precompute the response of the patches on the training set with the `ComputeFeatures` command. It is somewhat slow, taking around 10-15 mins, so be patient.

Training the detector: The number of weak classifiers used in training is set in the `trainDetector.m` file, at line 37. By default it is set to 100. Leave it at that for the moment and train a classifier with the `trainDetector` command.

Testing the detector: Now you can evaluate the model you have just trained with the `runDetector` command. This is a bit slow, taking around 10 mins but as it runs, you will see two figures pop up. The first shows the following (from L to R): each test image with ground truth annotation of objects; output of the weak classifier; thresholded output; detector output (with correct/incorrect counts). The second figure shows a precision-recall curve. Precision measures quality (i.e. 0.8 means 80% of detections are genuine), while recall measures what fraction of true object instances have been found (i.e. 0.5 means that half of the screens have been correctly detected). Optimal performance would correspond to the top-right corner.

Having run the entire system once, you should now do the following:

1. Re-run the training and testing routines, using $\{5, 30, 100\}$ weak classifiers. Save the plots and put them in your report.

2. Set the number of weak classifiers back to 100. Now open the `gentleBoost.m` file which is the main learning routine. Alter it so that the weights, strong and weak classifier predictions at each round of boosting are each stored in (separate) `Nrounds` by `Nsamples` matrices. Now place a breakpoint (either in GUI or by inserted a `keyboard` command) after the main loop over boosting rounds. Then run the `trainDetector` command, which should now stop at the breakpoint.

3. Plot (on the same figure) the weights after 1,2,3,4 boosting rounds and include this in your report. Note how the overall weights per iteration decreases, but within each round, the points with largest weight change.

4. Use `imagesc` to display the matrix of weak classifier predictions. Include the plot in your report.

5. Boosting relies on each weak classifier having above chance level performance. If this is not the case, the combination of weak classifiers cannot be expected to be above chance itself. To show this, you should deliberately damage the algorithm by using a random choice of k and th, instead of the ones selected by the `selectBestRegressionStump` sub-routine. You can keep the other parameters a and b the same (since they will be effectively random also, given that k has changed). k should be chosen uniformly over all features, while th should be set to `0.1*randn(1)`. Now re-run `trainDetector` and `runDetector` and include the resulting precision-recall curve in your report, as well as the your modified version of `gentleBoost.m`.

6. In the previous part, if only the dimension k is randomized (i.e. a, b and th are set optimally by `selectBestRegressionStump`), the classifier does considerably better than chance, although still not as well as if k were picked optimally. Suggest (in writing) why this might be so.

7. `selectBestRegressionStump` selects the best single weak classifier, i.e. the one having optimal dimension k and stump parameters a, b, th. Explain why the strong classifier may use multiple weak classifiers that utilize the same dimension k, each having different stump parameters.

8. Explain why the final weighting w_i for each point i is equal to $e^{-y_i F(x_i)}$, where $F(x_i)$ is the output of the strong classifier on input vector x_i.

2 Mean-Shift Segmentation

Mean-shift segmentation is a simple algorithm for decomposing an image into a set of coherent regions. Lecture 22 describes its operation. In this question you should implement the algorithm, using the code template `example_mean_shift.m` provided on the course webpage. You should fill in the code sections indicated by the ... and apply it to `image1.jpg` (from the course webpage) using `windows_size=0.5`. You should turn in your source code and the resulting figure.

For bonus marks you can try to alter the parameters `window_size` and `cluster_quantization` to improve on the default segmentation of the image.

3 Normalized Cuts

Note that this is a bonus question that you can do for extra credit. It is harder than the previous questions, so only attempt if you have already completed these.

Before attempting to implement the algorithm, download and read the Shi and Malik paper: "Normalized Cuts for Image Segmentation", from the course webpage. In the guidelines that follow, we will use the same notation as the paper.

The input to the algorithm should be a grayscale image of size M by N with intensities in the range 0 to 1. You should use `tiger.jpg` and `baseball.jpg` images on the course webpage.

The first stage involves computing the affinity matrix W between all pairs of pixels which will be a large MN by MN sparse matrix. The affinities between a pixel i and j are given by equation 11 in the paper. Note that there are two terms, one using the intensity difference between the pixels and the other using the distance between their locations. However, if the distance is greater than some threshold r, then the affinity is zero (which has the effect of keeping the matrix sparse). You should use the values $r = 5$, $\sigma_I^2 = 0.05^2$ and $\sigma_X^2 = 4^2$.

Since a non-sparse matrix of size MN by MN will not fit in memory,

you should use Matlab's sparse matrix functions to represent it. Type `help sparse` to see how such a matrix can be generated from a list of row indices, column indices and sparse values. Also, using for loops to generate the matrix is likely to be slow. A better approach would be to use the `im2col` command to decompose the image into a series of patches, each of size $2r+1$ by $2r+1$ within with the two affinity terms are computed. Then you should build a list of row and column indices of the non-zero terms within each patch in terms of the original images coordinates. From this representation, it would be straightforward to generate the matrix W with the `sparse` command. To handle the edges correctly, it is probably easiest to pad the image with the `padarray` command before using the `im2col` command, and then ensure the affinities within the padded region are zero before building the sparse matrix. Some tips to debug your code: (i) use a small patch of the image to start with so that you can compute using simple methods the correct value of W; (ii) W is symmetric – check that this is the case; (iii) the diagonal should consist of ones. Please comment your code clearly so that I can figure out how you are computing W.

Once you have built the sparse matrix W, compute the diagonal matrix D consisting of the row-sum of W using the `spdiags` command.

Now, compute the matrix $A = D^{-1/2}(D-W)D^{-1/2}$ and solve for the smallest K eigenvectors using the command: $[v,d] = eigs(A,K,'SM')$;. For visualization purposes, we want to look at several of the eigenvectors, so set $K = 12$.

Plot out each eigenvector, reshaping it into an image and plotting it with the commands `subplot` and `imagesc` and using `colormap(gray)` to make the images grayscale. For each of the test images, save the figure showing these eigenvectors.

Now, the second smallest eigenvector should approximate the optimal normalized cut solution. However, we want a binary segmentation of the image. Once solution is to threshold at zero, but a better one is to try various different binarization thresholds on the 2nd eigenvector and pick the one with the lowest NCUT(A,B). You should do the latter for a dozen or so different thresholds. See the top of section 2.1 in the Shi and Malik paper to see how to compute NCUT(A,B). You should print out the optimal value and plot the resulting segmentation.

Finally, for extra credit, split the image into more segments, using the code above to recursively split each of the two test images into 16 segments. Plot the resulting image segmentation, using a different color for each seg-

ment.

You should turn in your code, along with figures showing the eigenvectors and segmentations for both test examples.