Computer Vision Fundamentals - Assignment 5

Due: Fri 21 July, 2017 at 11:55 pm Grade Scale: 100 points (Twice weight of Assignment 1)

This assignment is to be implemented using MATLAB. It is recommended that you use the VLFEAT library, which has a lot of useful functions built-in. Your deliverable should be in the form of a single report document in which you describe what you did in each problem and why. Include all your relevant code segments as well as your explanations and results figures.

Zip all the code for each problem, named by the problem number, and upload them along with your report.

You can gain up to 20% additional points for implementing extra credit suggestions.

--

This assignment is taken from Project 4 for James Hays Computer Vision Course, available at http://www.cc.gatech.edu/~hays/compvision/proj4/. The text below is mostly taken from this webpage with acknowledgement.

The goal of this assignment is to introduce you to image recognition. We will explore a scene categorization problem, initially using very simple methods, and then building on towards Bag of Words classification. Bag of Words approach is discussed in the lecture slides, and in Szeliski 14.3.2 and 14.4.1.

We will be using a 15-category dataset introduced by Lazebnik et al. (http://www.di.ens.fr/willow/pdfs/cvpr06b.pdf). The dataset is organized into 15 scene categories – Bedroom, Coast, Forest, Highway, Industrial, InsideCity, Kitchen, LivingRoom, Mountain, Office, OpenCountry, Store, Street, Suburb, and TallBuilding – with a 100 training images for each category and a variable number of test images.

Project material, including image dataset, starter code in MATLAB and a report template in HTML is available at the give link. You must also download VLFeat 0.9.20 binary package from http://www.vlfeat.org/download.html. VLFeat is an open source computer vision library with MATLAB API, though many underlying functions are actually written in C for efficiency. Documentation of VLFeat MATLAB API is available at http://www.vlfeat.org/matlab/matlab.html.

The starter code, starting from `proj4.m` contains more concrete guidance on the inputs, outputs, and suggested strategies for the five functions you will implement in this assignment: `get_tiny_images.m`, `nearest_neighbor_classify.m`, `build_vocabulary.m`, `get_bags_of_sifts.m`, `andsvm_classify.m`. The starter code also contains `create_results_webpage.m` which you are not required to modify (but you can, if you want to show more images or change the size of the thumbnails).

Some Useful Functions in VLFeat Library

The starter code contains more complete discussions of useful functions from VLFeat. One thing to keep in mind is that while MATLAB and the starter code represent points as row vectors, VLFeat uses column vectors. Thus you might need to transpose your matrices / vectors frequently.

`vl_dsift()`. This function returns SIFT descriptors sampled at a regular step size from an image.

`vl_kmeans()`. This function performs kmeans clustering and you can use it when building the bag of SIFT vocabulary. MATLAB also has a built in kmeans function, but it is slow.

`vl_svmtrain()`. This function returns the parameters of a linear decision boundary (a hyperplane in your feature space). You will use the distance from this hyperplane as a measure of confidence in your 1-vs-all classifier.

`vl_alldist2()`. This function returns all pairs of distances between the columns of two matrices. This is useful in your nearest neighbor classifier and when assigning SIFT features to the nearest cluster center.

Problem 0. [0 points] Setup the starter code and the VLFeat Library. Run the starter code. This will randomly assign each image to a category. The accuracy of this random assign should be roughly about 7%, because on average 1 in 15 guesses should be correct. Go through the code to understand this functionality, as we will be using it again for every experiment.

Note that the code creates a *confusion matrix*, which provides the accuracy of every class being assigned to every other class. The mean of the diagonal of the confusion matrix can be considered as an accuracy measure. Check the code of creating the confusion matrix within the `create_results_webpage` function.

The code also creates a report of this experiment in HTML. You can use material from these automatically generated HTML reports to compose your final report.

Problem 1. [20 points] TINY IMAGES + NEAREST NEIGHBOR CLASSIFIER

You will start by implementing the tiny image representation and the nearest neighbor classifier. They are easy to understand, easy to implement, and run very quickly for our experimental setup.

The "tiny image" feature, inspired by the work of the same name by Torralba, Fergus, and Freeman (http://groups.csail.mit.edu/vision/TinyImages/), is one of the simplest possible image representations. One simply resizes each image to a small, fixed resolution (we recommend 16x16). It works slightly better if the tiny image is made to have zero mean and unit length. This is not a particularly good representation, because it discards all of the high frequency image content and is not especially invariant to spatial or brightness shifts. Torralba, Fergus, and Freeman propose several alignment methods to alleviate the latter drawback, but we will not worry about alignment for this project. We are using tiny images simply as a baseline. See `get_tiny_images.m` for more details.

The nearest neighbor classifier is equally simple to understand. When tasked with classifying a test feature into a particular category, one simply finds the "nearest" training example (L_2 distance is a sufficient metric) and assigns the test case the label of that nearest training example. The nearest neighbor classifier has many desirable features -- it requires no training, it can learn arbitrarily complex decision boundaries, and it trivially supports multiclass problems. It is quite vulnerable to training noise, though, which can be alleviated by voting based on the K nearest neighbors (but you are not required to do so). Nearest neighbor classifiers also suffer as the feature dimensionality increases, because the classifier has no mechanism to learn which dimensions are irrelevant for the decision. See `nearest_neighbor_classify.m` for more details.

Together, the tiny image representation and nearest neighbor classifier will get about 15% to 25% accuracy on the 15 scene database. For comparison, chance performance is ~7%.

Problem 2. [40 points] BAG OF SIFT FEATURES + NN-CLASSIFIER

After you have implemented a baseline scene recognition pipeline it is time to move on to a more sophisticated image representation -- bags of quantized SIFT features. Before we can represent our training and testing images as bag of feature histograms, we first need to establish a vocabulary of visual words. We will form this vocabulary by sampling many local features from our training set (10's or 100's of thousands) and then clustering them with kmeans. The number of kmeans clusters is the size of our vocabulary and the size of our features. For example, you might start by clustering many SIFT descriptors into k=50 clusters. This partitions the continuous, 128 dimensional SIFT feature space into 50 regions. For any new SIFT feature we observe, we can figure out which region it belongs to as long as we save the centroids of our original clusters. Those centroids are our visual word vocabulary. Because it can be slow to sample and cluster many local features, the starter code saves the cluster centroids and avoids recomputing them on future runs. See `build_vocabulary.m` for more details.

Now we are ready to represent our training and testing images as histograms of visual words. For each image we will densely sample many SIFT descriptors. Instead of storing hundreds of SIFT descriptors, we simply count how many SIFT descriptors fall into each cluster in our visual word vocabulary. This is done by finding the nearest neighbor kmeans centroid for every SIFT feature. Thus, if we have a vocabulary of 50 visual words, and we detect 220 SIFT features in an image, our bag of SIFT representation will be a histogram of 50 dimensions where each bin counts how many times a SIFT descriptor was assigned to that cluster and sums to 220. The histogram should be normalized so that image size does not dramatically change the bag of feature magnitude. See `get_bags_of_sifts.m` for more details.

You should now measure how well your bag of SIFT representation works when paired with a nearest neighbor classifier. There are many design decisions and free parameters for the bag of SIFT representation (number of clusters, sampling density, sampling scales, SIFT parameters, etc.) so performance might vary from 50% to 60% accuracy.

Problem 3. [40 points] BAG OF SIFT FEATURES + 1-VS-ALL SVM

The last task is to train 1-vs-all linear SVMS to operate in the bag of SIFT feature space. Linear classifiers are one of the simplest possible learning models. The feature space is partitioned by a learned hyperplane and test cases are categorized based on which side of that hyperplane they fall on. Despite this model being far less expressive than the nearest neighbor classifier, it will often perform better. For example, maybe in our bag of SIFT representation 40 of the 50 visual words are uninformative. They simply don't help us make a decision about whether an image is a 'forest' or a 'bedroom'. Perhaps they represent smooth patches, gradients, or step edges which occur in all types of scenes. The prediction from a nearest neighbor classifier will still be heavily influenced by these frequent visual words, whereas a linear classifier can learn that those dimensions of the feature vector are less relevant and thus downweight them when making a decision. There are numerous methods to learn linear classifiers but we will find linear decision boundaries with a support vector machine. You do not have to implement the support vector machine. However, linear classifiers are inherently binary and we have a 15-way classification problem. To decide which of 15 categories a test case belongs to, you will train 15 binary, 1-vs-all SVMs. 1-vs-all means that each classifier will be trained to recognize 'forest' vs 'non-forest', 'kitchen' vs 'non-kitchen', etc. All 15 classifiers will be evaluated on each test case and the classifier which is most confidently positive "wins". E.g. if the 'kitchen' classifier returns a score of -0.2 (where 0 is on the decision boundary), and the 'forest' classifier returns a score of -0.3, and all of the other classifiers are even more negative, the test case would be classified as a kitchen even though none of the classifiers put the test case on the positive side of the decision boundary. When learning an SVM, you have a free parameter 'lambda' which controls how strongly regularized the model is. Your accuracy will be very sensitive to lambda, so be sure to test many values.
See `svm_classify.m` for more details.

Now you can evaluate the bag of SIFT representation paired with 1-vs-all linear SVMs. Accuracy should be from 60% to 70% depending on the parameters.

There are many excellent extra credit suggestions on the project webpage of James Hays course. In particular, you can try the “Experimental Design Extra Credit”.