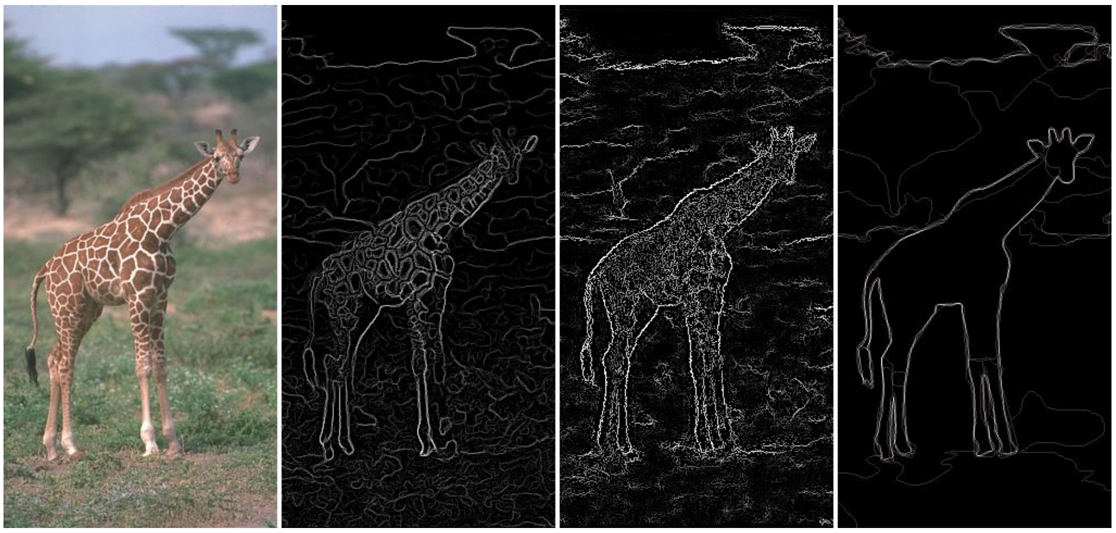


From left to right: (1) input image (2) boundary strength based on the average of many Canny edge detections (3) results from this project (4) average human boundary annotations. The Canny detector responds strongly to the interior texture of the giraffe instead of its boundary against the background. The Sketch Token result agrees more with human boundary annotation. The multitude of weaker boundary detections are not a problem as long as they are less confident than the correct boundaries.

Project 5: Boundary Detection with Sketch Tokens

Overview

Boundary detection is an important, well-studied computer vision problem. Clearly it would be nice to have algorithms which know where one object stops and another starts. But boundary detection from a single image is fundamentally difficult. Determining boundaries could require object-specific reasoning (E.g. I know this is a boundary because I recognize this as an elephant) making boundary detection "vision hard".

Classical edge detection algorithms, including the Canny and Sobel baselines we will compare against, look for intensity discontinuities. The more recent Pb boundary detectors (e.g. Martin, Fowlkes, and Malik 2004) significantly outperform these classical methods by considering texture and color gradients in addition to intensity. Much of this performance jump comes from the ability of the Pb algorithm to suppress false positives that the classical methods produce in textured regions. The more recent "global" Pb (gPb) method of Arbelaez, Maire, Fowlkes, and Malik 2011 (pdf) improves boundary detection further by reasoning about longer range relationships between contours. This work is still very near state of the art performance.

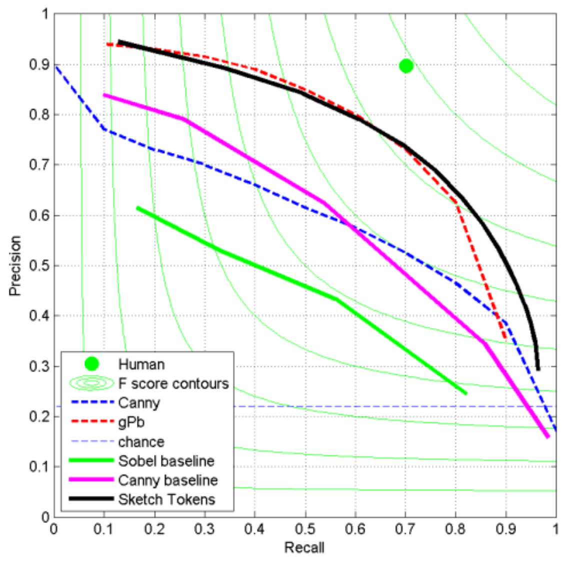
This year Lim, Zitnick, and Dollar introduced the Sketch Tokens boundary detector. The Sketch Tokens approach differs from the gPb algorithm in several ways. (1) Sketch Tokens is an entirely local algorithm. Each boundary decision is based on local patch statistics without any sort of global or long range reasoning. (2) Sketch Tokens uses relatively simple gradient and color features instead of the relatively complex, hand designed texture half-disc descriptor of Pb. (3) Sketch Tokens relies heavily on machine learning to determine the mapping from image structure to boundary scores, while Pb does not. The Sketch Tokens algorithm far exceeds the performance of the local Pb algorithm and matches the performance of the more complex "global" Pb

In this project, you will implement a slightly simplified version of Sketch Tokens.

Berkeley Segmentation Data Set

Before we go into more details about the algorithm you will implement, we will discuss the data set you will use for learning and evaluation. The BSDS 500 data set is included with your starter code. It contains 500 images -- 200 training images, 100 validation images, and 200 test images. You don't need to use the validation images for this project.

Each of the 500 images has 4 or more "ground truth" human segmentations. We don't actually care about the segments for this project but instead the boundaries between them. These boundary maps are stored in Matlab readable .mat files. These are the "Sketches" from which "Sketch Tokens" are derived. You will use some or all of the 200 training images to train a classifier. You will run this classifier on every pixel of the test set images and create boundary images. These boundary images will be passed to code from the Berkeley Segmentation Data Set to compute the precision-recall and F measure of your algorithm. Unfortunately, this evaluation method is quite slow because it uses a complex correspondence algorithm to relate your detected edges to the human drawn boundaries. This is preferable to a naive metric which simply checks per-pixel overlap.



Dotted lines are copied from figure 17 in Arbelaez et al. 2011 (pdf).Solid lines come from our implementations.

Light green lines are iso-F measure lines

By default, the starter code only measures performance on 10 of the 200 test images to speed up testing and evaluation. However, if you want a more trustworthy evaluation you should run your algorithm and the evaluation on the full test set. If you want to "win" the performance competition

for this project you must report a score on the full test set.
A sample precision-recall curve is shown to the right.

The BSDS evaluation computes a precision-recall curve just like we used in project 4. While in project 4 we used average precision to summarize this curve, in the boundary detection community one typically reports the highest "F-measure" of any point on the curve, where F-measure is the harmonic mean of precision and recall. F-measure may be preferred because certain boundary detection algorithms have trouble producing a full precision recall curve, and those algorithms would be heavily penalized by average precision. On the other hand, optimizing only for F-measure seems to have led to algorithms that don't perform especially well in high precision or high recall regimes, because those parts of the curve would never have high F-score. The Sketch Tokens paper reports average precision and it is quite possible that you beat most of the start of the art methods in boundary detection according to average precision.

The Sketch Token Algorithm

This project is a fairly faithful implementation of the Sketch Tokens algorithm. Therefore it is important that you read the paper through Section 3!

Your Implementation

We suggest implementing the Sketch Tokens algorithm in a particular order to make debugging easier. In particular, we recommend skipping the entire concept of Sketch Tokens with your first implementation! Instead you will first train a binary classifier which recognizes boundary versus non-boundary.

Part one: Image representation in get_channels.m. First you need to be able to represent image patches as the Sketch Tokens algorithm does. This should feel pretty familiar to you as it uses some of the same techniques as the SIFT implementation which you've already done. As described Section 2.2.1 in the paper, you need a function to convert an RGB image to 14 particular channels: 3 LUV color channels, 3 overall gradient magnitude channels, and 8 oriented gradient magnitude channels. You can use the function rgbConvert(I,'luv') in Piotr's Toolbox to create the color channels. For the gradient magnitude images you want to blur the image by the appropriate amount (Gaussians with sigmas 1.5, and 5 and no blur [sigma=0]) and then compute the gradient magnitude at each pixel (i.e. the square root of the squared x and y derivatives). For the blurs of sigma 0 and 1.5 you want to compute oriented gradient magnitude, as well, for orientations of 0, pi/4, pi/2, and 3pi/4. We are interested in gradient magnitude not the signed gradient, thus computing the gradient magnitude at pi/2 and 3pi/2 would be redundant. You can create the the gradient channels with imfilter and appropriate oriented filters (e.g. Sobel-like filters at particular orientations). You should use imfilter's 'symmetric'option to minimize boundary artifacts, although some are unavoidable. You could, alternatively, compute the gradient magnitude at particular orientations analytically by projecting the x and y gradients onto vectors at the appropriate orientation. You are not required to implement the "self similarity" features described in Sketch Tokens, although you may do so for extra credit. You are not allowed to use Piotr's toolbox channel functions other than imPad and rgbConvert.

Debugging tips: Make sure that the channels you compute aren't somehow degenerate (e.g. certain channels are all zero). Check that rgbConvert is returning valid images. Make sure you are taking the absolute value of the oriented gradient responses because we want gradient magnitude.

Part two: Creating training data based on BSDS annotations in *get_sketch_tokens.m*. The starter code provides an example of loading a BSDS annotation file for a particular image. The annotations you will use are simply binary images with pixels=1 for boundaries and pixels=0 for non-boundaries. There are 4 or more annotations for every training and test image. You need a function which loops through the training images and pulls out positive and negative training examples. A positive training example is a NxN patch of the 14 channels you computed in part one which is centered on a pixel that has been annotated as a boundary. Negative training examples are the other patches. You want to use a single call to get_channels.m for each training image. If you call get_channels.m for every patch then the resulting representation could be dominated by image filtering boundary effects. You want to be able to specify the ratio of negative and positive training examples -- you'll get a more accurate classifier if your training data is close to a 50/50 split than if it follows the natural distribution of edges and non-edges. For initial experiments, you should keep the patches small because otherwise they use a lot of memory (e.g. a 35x35 patch of 14 channels, as used in the paper, has 17150 dimensions). 15x15 patches should be fine for initial testing. While more training data is always better, 30 thousand training samples is enough for initial testing.

Debugging tips: Check that you're cutting out the correct patches. Many students make mistakes in the indexing and think they are cutting out positive examples but instead are cutting out arbitrary patches because they didn't take into account padding they added to the training image. You don't necessarily need to pad the training image here, you could instead cut out training examples only from the valid central region of the training images. It is important that your training examples be diverse. Using 100,000 training examples can lead to poor performance if they all come from two training images. Ideally you would ranodmly sample all 200 training examples. For debugging, though, you might get away with sampling from 20 or 40 images. You also don't want to simply sample the boundary and non-boundary examples in scanline order. This will also lead to training examples that are overly redundant. Sampling random training examples from each image works well. Yuo should be able to eyeball the training features returned by get_sketch_tokens (using a visualization such as imagesc) and see the difference between boundary and non-boundary training examples. Non-boundary training examples should have slightly lower gradient magnitude on average. Also make sure that your labels are '1' (non-boundary) and '2' (boundary). There will be more labels in part five once we try to recognize sketch tokens.

Part three: Training a random forest classifier. As argued in the paper, a random forest classifier is well suited to our boundary detection task. Random forest classifiers are powerful (they are non-linear, unlike the linear SVMs we've used for projects 3 and 4) yet they are still very fast to evaluate. That is important when you need to make decisions about tens of millions of pixels! However, training a random forest can be quite slow. This will depend a great deal on the particular parameters you choose. You will use the forestTrain function to train a classifier to distinguish boundary and non-boundary features. See the documentation for forestTrain function for more details. Although there are many parameters, the defaults work fairly well. You will want to use more than 1 tree, though. 20 is a reasonable number. Experiment with the other parameters once you have things working. You do not need to make multiple passes through the training data to train and combine multiple decision trees as is done in the Sketch Tokens paper. You can get reasonable performance with however many training samples you can fit into memory at once.

Debugging tips: The starter code by default reports the training error of your random forest classifier. Because a random forest classifier is non-linear with an arbitrary number of model parameters, it tends to have very low training error. If your training error is not high (.9 average precision or more) then there may be something degenerate in your features (e.g. they are all near zero or all near duplicate). If your training error is low your test error could still be high. Unlike the linear SVMs used in previous projects, the random forest is able to overfit to whichever training data you provide it (even random noise). As you sample a bigger, more diverse set of training examples you might see your training error increase while your testing error decreases. That's fine. Don't try to optimize training error. The starter code only evaluates training error as a sanity check. To better assess how accurate your random forest is without going through the entire boundary detection pipeline you can create a validation set by taking a new random sample from the training images or even better a sample from the BSDS validation set. The starter code includes some placeholder code for this purpose.

Part four: Boundary classification in detect_sketch_tokens.m. Now you need to create a function which takes as input test image and returns a real-valued boundary likelihood image. This is fairly easy, actually, because our random forest classifier natively returns likelihoods. Simply invert the probability of the background class as returned from forestApply or equivalently sum the likelihoods of the non background classes (this is equation 1 in the Sketch Tokens paper). Your boundary image needs to be the same height and width as each test image, so you need to pad the boundaries of the test image in order to build features centered around each pixel. You can use imPad for this purpose if you want. You can call forestApply once per pixel to find the boundary likelihood, but that is somewhat slow. To make your code faster you can convert the image to the channel features and stack the patches into rows of a large matrix and make one call (memory permitting) to forestApply. If you don't have enough memory to do this, you can call applyForest on groups of patches (e.g. all of the patch features from a single row of the test image) and this will still be fast. You can use reshape() to convert a 1d array of likelihoods to a 2d image. The boundary image generated by applying your classifier to each pixel will look a bit blurrier than you might expect. You should apply non-maximum suppression to it using the provided function stToEdges.m

Debugging tips: Make sure that you are computing the 'channels' exactly the same was at test time as you did at training time. If you divide the input image by 255 at training time but not at test time (or vice versa) your performance will be terrible because the image features will be drastically different. Make sure that you flatten the patches from the channel images in the same way at training and testing time.

At this point you have a complete boundary detection method. With some parameter tuning, it can work extremely well! Don't move on to the next parts until you're beating the Canny baseline (F score above 0.58). Now, let's define and detect "Sketch Tokens".

Part five: cluster annotations into Sketch Tokens and use them during classifier learning. Instead of just sampling image features from the training set, you now need to sample the human annotations as well. You should convert human annotation images into the DAISY representation with a call to compute_daisy.m. compute_daisy.m returns a structure containing a DAISY descriptor centered on each pixel in the image. To get a particular DAISY descriptor use the function get_descriptor.m. You can implement this as a modification to get_sketch_tokens.mor as a separate function. It is probably easiest to sample the image features and the sketch features simultaneously, but this strategy may not generalize as well to more advanced training methods. E.g. if you want to sample a particular number of image features for each sketch token then you may want to have a vocabulary of sketch tokens decided on already. Once you have enough image and sketch samples, e.g. 30 thousand of each from the same locations in the same images, you will run k-means on the DAISY descriptors and then the cluster membership of those sketches implies the Sketch Token categories of the corresponding image features. These labels can simply be passed to the forestTrain function without further modification because the forestTrain natively supports multiway classification. This is one of the advantages of using a random decision forest rather than the SVMs we have used for prior projects (even though we did, in fact, use SVMs for multiway classification). Instead of having two labels (e.g. 1 for background, 2 for boundary) you will have K labels (e.g. 1 for background, 2 through 17 for 16 sketch token classes). Your detection function may need to change, as well, to sum up the probabilities of the individual sketch tokens. Alternatively you can simply use 1 minus the background probability returned from forestApply and thus your detector code doesn't even change.

Starter Code

To make this project easier for you we've included four different packages. Unfortunately, each of these relies on mex'd C or C++ code which means it is not easily cross platform. We've tried to precompile everything for Windows, Linux, and MacOS. If you get an error saying that a particular function isn't found, e.g. "Undefined function 'mex_compute_all_descriptors' for input arguments of type 'single'" then you may need to compile the function according to the instructions below.

- The Berkeley Segmentation Data Set evaluation method in folder bench. This relies on a mexed function, correspondPixels. Compile it with build.m in bench/source if you need to. We went through considerable effort to create a version that compiles in Windows.
- Piotr Dollar's Image and Video Matlab Toolbox in folder <codepiotr_toolbox< code="">will be used to train random forest classifiers (specifically the function forestTrain). The toolbox includes binaries for Windows and Linux. The function toolboxCompile can be used for other operating systems.</codepiotr_toolbox<>
- The DAISY descriptor is a SIFT-like descriptor with a ring structure that is used to cluster human annotations into Sketch Tokens. We provide a modified version of the code in folder daisy. The code is compiled for Windows 64 and Linux 64, but you can use the build.mfunction

for other operating systems.

- VLFeat is once again provided. You can use it for k-means or simply use Matlab's built in version.

The starter code provides an outline of the pipeline and provides examples of image loading and visualization. If you run the starter code unchanged, it will use Sobel and Canny based methods to detect boundaries in a small test set and then evaluate those results with respect to the BSDS ground truth. The average of Canny edge detections with different parameter settings is a non-trivial baseline, but you can beat it!

Performance Tips

This section contains suggestions for achieving higher accuracy. Everything proposed here is optional. By considering the issues below, we were able to achieve a best performance of F=0.73, matching the Sketch Tokens paper and gPb. However, as long as you are beating the Canny baseline then you can get full credit for this project.

While it may seem as if boundaries should be relatively simple structures (at least compared to previous visual phenomena that we've tried to recognize such as scenes and objects), it turns out that the Sketch Token algorithm depends heavily on having rich image features and large amounts of training data. This makes it computational difficult to achieve the highest performance. For example, our reference implementation achieved an F-score of 0.70 only by using 150 thousand training samples of 10 thousand dimensional features. That's 6GB of training data! The Sketch Token paper used even more training data by training decision trees independently with different random samples of the training data and then combining them into a final decision forest. Multiple trees (or forests of trees) learned with forestTrain can simply be concatenated together (e.g. forest = [forest1; forest2]). If you're going to adopt this stategy it is recommended that you save the intermediate forests to disk using save() and then read them and combine them later. You're not required to independently learn and combine classifiers. You should debug your system with smaller amounts of training data and lower dimensional features. For example, 11x11 patches of the various channels can achieve an F score of 0.67 with enough training data.

Even though a key concept of the Sketch Tokens paper is, obviously, the Sketch Tokens, you may not see much performance gain by using them. The simpler alternative of not clustering the annotations and simply training a boundary versus background classifier can work well. For different choices of features, training data, and learning parameters we saw the use of Sketch Tokens decrease or increase performance by about 0.02. The paper reports an increase in F-score of 0.05 by using sketch tokens but we could not match that. Maybe you can!

To make development easier, make use of code cell structures and consider saving the learned decision forests for future reuse. The learned forests are relatively compact. You don't want to try and debug your detector while having to resample training data and relearn the classifier each time.

Many parts of the code can be parallelized in Matlab (see help parfor). This can provide a significant speedup on a multicore system but it can make debugging more difficult.

You might benefit from some careful treatment of the training data (part two, above). For example, what do you do when one human annotator labels a pixel as a boundary but another (or perhaps, all four other) annotator does not? Also, should the pixel directly adjacent to a human-labeled boundary be treated as negative training data? Maybe it is actually a boundary because the humans weren't pixel-perfect accurate. You can try different strategies for deciding which annotations to trust as negative and positive training data. Luckily the naive approach will not break things catastrophically.

Blurring the probability of boundary estimate that your decision forest produces before running non-maximum suppression can improve performance and make the results look far better qualitatively.

Writeup

For this project 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. 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). For this project you should include the precision recall plot of your best performing variant as well as the reported F-score (maximum harmonic mean of precision and recall). You should also include some input / output pairs. You might (optionally) want to include examples of learned sketch tokens (as in Figure 1 of the Sketch Tokens paper). To create such a visualization store the image patch representation of each boundary sample and then average them according to the kmeans Sketch Token cluster assignments. To visualize the learned classifiers you can show the detections of individual sketch tokens in a test image.