Connor Thompson
04/25/2020
CSCI 4830

Final Project Report

https://github.com/RoastedPecans/CSCI4830-Final-Project

Overview and Motivation

For my final project, I attempted to implement the mPb detector found in the paper "Contour Detection and Hierarchical Image Segmentation" by Pablo Arbeláez, Michael Maire, Charless Fowlkes, and Jitendra Malik (2011). I originally found this algorithm/contour detector when I was looking at the chart found in Lecture 12 of this course and saw "gPb" was one of the best performing boundary detectors currently available. After looking at a few other more recent algorithms (OEF, SE, PMI to name a few, from this source), I chose to pursue mPb because it was one of the few that did not rely heavily on machine learning/neural net "black boxiness", the method + research paper were relatively straightforward, and it was expandable to gPb/sPb. Thus, mPb seemed like a great "starting point" for my final project that was focused, achievable, and expandable.

mPb is shorthand for "Multiscale Probability Boundary". It is a multi-scale extension of the Probability Boundary (Pb) detector which was created in "Learning to Detect Natural Image Boundaries using Local Brightness, Color, and Texture Cues" by David Martin, Charless C. Fowlkes, and Jitendra Malik (2004). The idea behind the Pb detector is you are trying to predict the posterior probability of a boundary being at a given point. An extension of mPb, sPb is short for "Spectral Probability Boundary", it uses our filter bank and some eigenvectors found within our mPb data to add global information to our output image. Combining mPb and sPb gives rise to a cutting-edge contour detector gPb or "Global Probability Boundary". I chose to initially limit my project scope to just mPb because the research paper became slightly more complicated with the introduction of sPb. In addition, I was already worried that mPb would take many computational resources due to having to make many copies of the data and run other computational intense code on these copies.

My hope for this project was to be able to get results that were similar to those found in the research paper. My hypothesis was that by following the research paper thoroughly and doing my best to do just as the authors did, I would get results that were comparable to theirs. I imagined that upon achieving comparable results I would be able to adjust some parameters (such as what features and scales we should use, what kind of smoothing to use, our choice of filter bank, etc) and ultimately obtain better results than those found in the paper for mPb.

Algorithm

The algorithm used to create our final mPb image can be broken down into 9 concrete steps. I present these formally below:

1. To begin, we first generate our filter bank. Our filter bank includes 8 angled, even and odd symmetric, gaussian derivative "bar" filters at 3 different scales. The even-symmetric filters are gaussian second derivatives and the odd-symmetric filters are just the Hilbert transform of the even. This gives us a total of 48 filters. In the research paper, they also include one difference of gaussian filter which I did not end up including since I did not notice a significant improvement when including it and it led to confusing code restructure (size difference from bar filters). I used code from the original paper to generate this filter bank which can be found on the UC Berkeley CV Group website here. The scales I ultimately ended up using were [19, 27, 39] for our texture features and 7 for our intensity feature.
2. After generating our filter bank, we then convolve each set of filters (based on scale, so 3 sets of 16 filters in this case) on our input image. This gives each pixel a 1x16 vector for each of our scales. We run K-Means clustering (K=32) on the resulting vectors for each scale to cluster our pixels into image-specific textons. An example for this step is shown in figure 2 in the "Results" section below.
3. We add our raw intensity information to the results we obtain from our K-Means clustering, giving us intensity information at one scale and texture information at three.
4. We now pre-rotate each of the 4 features we have from step 3 in 8 orientations. As mentioned in the appendix of the research paper, this pre-rotation allows us to significantly reduce our time complexity for step 5. Our 8 orientations are: [0, 22.5, 45, 67.5, 90, 112.5, 135, 157.5] all in degrees. This gives us 32 total features (4 features with 8 orientations each).
5. In the quintessential step of the algorithm, we iterate over each pixel in each of our 32 features from step 4. At each pixel, we take the scale corresponding to the feature we are currently inspecting (mentioned in step 1) and draw a rectangle around it. We then split this rectangle in half, histogram both halves, and calculate the Chi-Square distance between our two histogram bins. This distance becomes the value of that center pixel.
    a. Our pre-rotation in step 4 allows us to not have to angle each rectangle we draw 8 different ways for each pixel which saves us a lot of computation. Instead, we can just pre-rotate the image and then draw a normal, non-rotated rectangle on top of the rotated image. After step 6 we can then rotate our results back to the correct orientation without losing any information. An illustration from the original paper is shown below:

Figure 1: Illustration from original research paper showing both the pre-rotation of our images in step 4 as well as the half-rectangles in step 5.

6. With the Chi-Square distance now computed for every pixel, we apply Savitzky-Golay Filtering. This enhances local minima and smooths detection peaks.
7. With our now filtered results, we rotate them back so they are all the same orientation as the original image.
8. We now collapse our results so that our 32 results become 8 – one for each orientation. In other words, we now combine all of our features (3 textures at different scales, 1 intensity) by the angle the histogram difference was calculated at. We achieve this with a simple set of 32 weights which we learned via SSD minimization on 10 ground-truth images from the Berkeley Segmentation Data Set. This gives us mPb(x, y, theta).
9. We now collapse mPb(x, y, theta) into mPb(x, y) – our final results – by taking the max value at each pixel along the orientation axis.

Results

Below are some of the results I achieved with my final algorithm + trained weights. I also include some results from other algorithms + from the paper for comparison. For more results, take a look in the "results" folder included with my project submission, or on the GitHub linked at the top of this document.

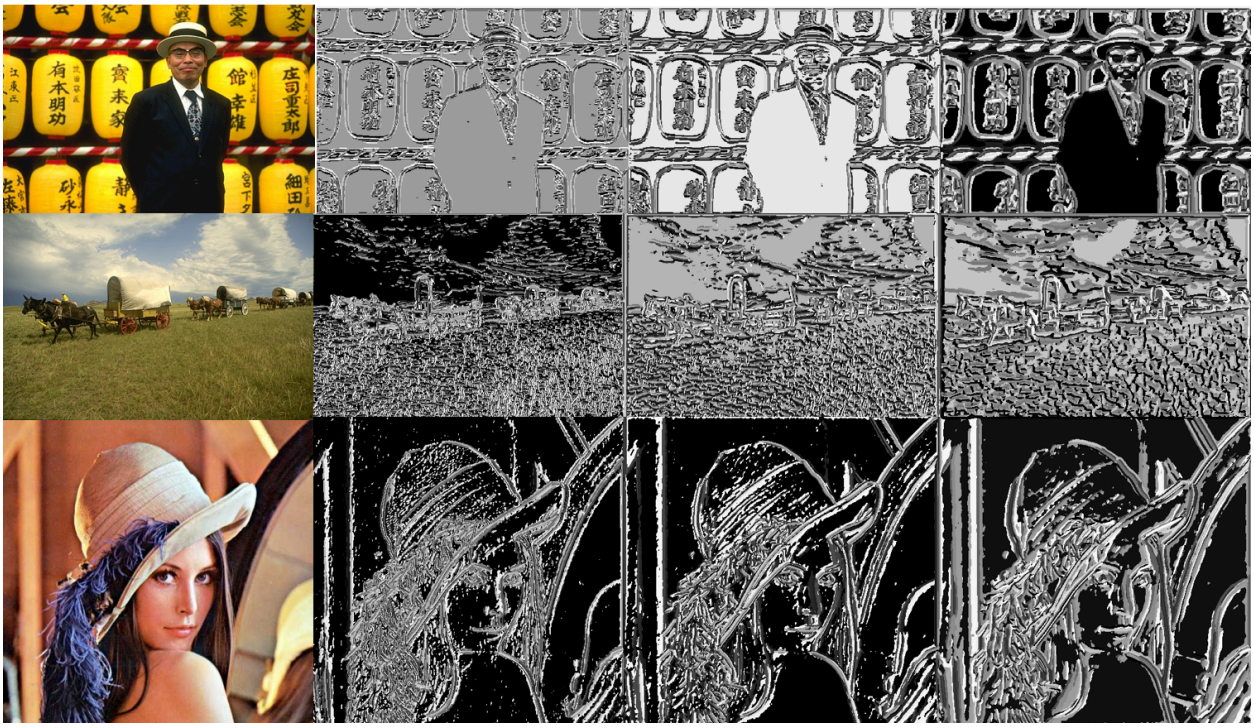

Figure 2: Texton Maps generated from my algorithm on 3 different input images. Texture scale increases as you move right.

Figure 3: Original BW Image vs mPb Results

Figure 4: Original BW Image vs mPb Results

Figure 5: In order: original BW Image, my mPb Results, Canny Edge Detection Results, mPb Results from paper

Problems

Over the duration of this project, I encountered a few problems which if given more time could've likely been resolved. The problems I encountered and the impact they had are listed below:

- First and foremost, my plan was to start with mPb straight from the paper and then add on additional features to see if I could get better results. However, after actually finishing my implementation I realized my results while good, weren't as good as I had hoped. As shown in figure 5 above, my results appear "noisier" than the results from the paper. As a result of this, I decided that adding global information (instead of just local) could be valuable and thus tried to pivot from adding additional features to working on getting sPb/gPb working.
    - However, I ran into many issues getting sPb to work as well. Initially I tried to just use the code from the original paper (since it's advanced math and I was worried I wouldn't implement it correctly). Unfortunately, I kept receiving errors about missing libraries and files from MatLab when trying to run the original code so after ~4 hours I gave up on this approach. I then started trying to implement sPb by myself, but realized that by the time I finished it I wouldn't have any time to

actually train the weights used enough for either mPb or gPb… Thus, I decided to pivot to just focusing on mPb and trying to get my results closer to the paper, but I unfortunately wasted a lot of time in the process.

- Second, after noticing my results were noisy as shown in Figure 4, I tried to retrain my weights on results I obtained using larger filters. However, this approach did not generalize as well and my larger filters became more "disproportionate" as they were scaled up due to an elongation factor used in the filter bank code. Ultimately, this gave me some decent results, but I didn't have enough time to fully re-train my weights. This is definitely something that I would've liked to investigate further.
    - I also would've liked to investigate more into filtering/non-max suppression to further eliminate some of the noise in my results. However, I was too busy focusing on getting sPb to work because I thought that would prove better overall.
    - It's also unclear from the paper which scales were used. In the paper, they mention using 3 scales for each feature, but in the code I found for the paper, I see 4 scales used so I'm assuming they added in an additional scale to the code post-publication? Or it's also possible I'm misreading the code. However, I would've liked to further investigate if using more scales would've been useful.
    - In addition, I also only used one scale for the intensity feature while the authors used 3 just like for texture.
- Another problem I ran into was with the Savitzky-Golay Filter (SGF) mentioned in the paper. MatLab includes a 1D SGF in the Signal Processing Toolbox and I was able to find a 2D SGF online (not used in final project). However, I also noticed that the authors had their own 2D SGF implementation which I had difficulties running with my code. However, my comparisons between 1D SGF and 2D SGF revealed very little difference so ultimately I went with the built-in MatLab function for safety, speed, and robustness. In the future, I would be curious if the authors use some "custom implementation" of SGF which leads to their less noisy results.
- Finally, my last and most annoying problem was having to train my algorithm. While this algorithm doesn't require any "real" ML, it does require training a small set of 32 weights. This ended up being a bigger problem for me than I anticipated for multiple reasons:
    - Before even beginning to train weights, I had to run my algorithm on each image I wanted to train on to generate some pre-requisite data for the training routine. My algorithm takes 10/15 minutes to run per image and thus every time I changed any of my parameters, I would have to spend ~2 hours regenerating this training data before I could begin training again. The training itself however was reasonably fast.
        - As a result of this, I had to severely limit my training set, I ended up using just 10 images out of the 500 available. I imagine if I was able to use the entire dataset my results would be much improved.
        - This would also be less of a problem if I wasn't working solely off a 13" Macbook Pro. If I had access to a computer with more thermal

capacity/more GPU power this likely wouldn't have been a problem. I blame Coronavirus…

- This also was the only real "complaint" I had about MatLab for the duration of this course. I ended up having to write my training algorithms entirely from scratch and there's not even a built-in way to set-up something like a grid search. It just seems like MatLab should have some "standard" way to run a very simple training routine. If I had chosen to do this project in Python instead, I likely would've been able to use a more advanced training algorithm from some third-party library. In addition, I used SSD to train instead of F-Score like the authors used which is a worse metric for generalization. While it's nice to implement things on your own, I would've liked to see how far I could've pushed my results with a more robust training program.

Conclusion

Throughout the duration of this project, I became a lot more confident in my ability to develop complex applications, particularly with respect to Computer Vision. This gave me hands-on experience finding a research subject, dissecting a particular paper within that subject, and then trying to replicate the results on my own which I think is invaluable. In addition, I got to try working with texture information that is encoded in images for the first time which was good practice and much easier than I thought it would be. However, I think the biggest take-away from this project was my project management growth. Having to manage this project entirely on my own helped me develop my project management skills since I had to constantly be thinking ahead and trying to figure out what I needed to do for my project and when. In a few cases, I drove myself into a corner and focused too much on one aspect of the project while ignoring others that could've proven useful to pursue. One example of this was not implementing non-max suppression because I was too focused on trying to get sPb to work with my version of MatLab. However, learning from these mistakes is just as valuable as committing them.

For future Computer Vision students, I would recommend a few things:

1. You really actually should start the homework early for this class. I know that gets said about every class but starting early in this class will save you a lot of stress. The homeworks are *long* (but fun!). (For what it's worth, I think the homework amount was about perfect actually, but I procrastinated on the first 2 and doing it all at once is not fun…)
2. Only take this class if you're passionate about Computer Vision. I know a lot of my friends just sign up for electives without looking too much into them (or they just choose the easiest available…). This class is not one of those. It is challenging, technical, and at times tedious, but the fruits of your labor are well worth it if it's something you actually care about doing. I never felt like I was doing "homework" in this class because

it was more of just “building a really cool program that happens to be for school”, but I’ve always been very interested in computer vision.

3. It is not computer vision via machine learning :) I’m glad this was highlighted early in the semester, I also noticed the class seemed to get much smaller after this + as we continued the semester… It is very interesting to learn the “traditional” way however and part of the reason I looked into gPb for my final project was because I wanted to explore some of the current “limits” of traditional CV.

Bibliography

This project would not have been possible without the following outside sources:

UC Berkeley Computer Vision Group

I utilized two research papers heavily throughout the course of this project. Citations for these are given below.

Contour Detection and Hierarchical Image Segmentation
P. Arbelaez, M. Maire, C. Fowlkes and J. Malik.
IEEE TPAMI, Vol. 33, No. 5, pp. 898-916, May 2011.

D. R. Martin, C. C. Fowlkes and J. Malik, "Learning to detect natural image boundaries using local brightness, color, and texture cues," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 5, pp. 530-549, May 2004.

Beyond the papers, I also made extensive use of the "Berkeley Segmentation Data Set and Benchmarks 500" (BSDS500) which includes human-drawn ground truth segmentations for 500 images, as well as utilities to benchmark your algorithm. A link to the dataset is given below, of note, a link to the PDF version of the above research paper, and the code used for the research paper, is hosted on this site as well.

https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html

Others

The code I used to experiment with 2D Savitzky-Golay Filtering can be found via the citation below:

Shao Ying Huang (2020). Savitzky-Golay smoothing filter for 2D data (https://www.mathworks.com/matlabcentral/fileexchange/37147-savitzky-golay-smoothing-filter-for-2d-data), MATLAB Central File Exchange. Retrieved April 25, 2020.

In my code, I also utilized and slightly modified the pdist2 function from Piotr Dollar's Toolbox. This is included in my code as pdistNew since Matlab's pdist2 doesn't include Chi-Square distance. A link to this function, which includes a link to the original toolbox, is found below:

https://web.archive.org/web/20190131024241/http://www.cs.columbia.edu:80/~mmerler/project/code/pdist2.m

I also used this curated list of computer vision research papers while looking for project inspiration:

https://github.com/MarkMoHR/Awesome-Edge-Detection-Papers