CSSE 463

Puzzler Final Report

Shunhao (Scott) Cai

Netta Gneezy

Eric Richardson

Ethan Russell

Table of Contents

Abstract

This paper describes a color and shape based approach to determine jigsaw puzzle pieces fitting. The algorithm starts by creating a black and white mask of each piece. For each piece, it finds the corners and classifies the sides. Then for every possible combination of sides of each piece, it computes features based on the shape of both pieces and the color distance of the edge pixels. Finally, we run these features through a trained SVM to determine how well the sides match. Using the score of each possible match, we compute the best way to solve the puzzle by maximizing the score. The trained SVM has a true positive rate of finding matches of 84.15% and a false positive rate of 20.45%. While this is poor accuracy, this paper presents a powerful approach to puzzle solving with many areas for further improvement.

Introduction

Solving puzzles is an interesting problem because it is typically seen as an intelligent activity, but advances in AI have made the task much easier. It is also interesting to think about how a computer would go about solving a puzzle as, in contrast to other games such as chess, puzzle-solving does not have a particular set of rules or instructions that the computer can follow. Furthermore, puzzle-solving can be extended beyond jigsaw puzzles to similar real world problems like broken object reconstruction (i.e., broken artifact or shredded document reconstruction). Puzzle solving becomes particularly challenging as the solution space grows exponentially with larger puzzle sizes. To put this in terms of numbers, in a puzzle with n pieces the number of possible combinations for one side of a puzzle piece is approximately $4 * (n - 1)$, without taking into account certain factors such as whether the side in question is part of the puzzle's border or not. Our proposed solution will be able to extract pieces from a scanned image and solve a large puzzle set.

Figure 1: RGB mask output of mated pieces

Literature Review

Our first step in this project was to review past work on the topic of solving jigsaw puzzles in order to gain a better understanding of the problem, in addition to providing us with a good starting point. One of the first sources that we came across provided some ideas and algorithms for solving puzzles that either had fixed directions and random locations, or that had unknown orientation and locations. The method implemented by this source was to loop over four dimensions of each piece as a basic unit for building up the puzzle and involved looked at possible solutions in both LAB and RGB color space [3]. In another source, the authors demonstrated a method for solving jigsaw type puzzles, that uses the geometry and color features of each piece to find pieces that go together [5]. For a full list of the resources that we reviewed, please refer to the References section. While reviewing past work done on solving jigsaw puzzles provided us with a clearer understanding of what the problem entails, we did not find much use for them when it came to the actual implementation of our puzzle solver. This was likely due to the fact that the works we reviewed had much more detailed solutions and also took place over a longer period of time and therefore had more room to experiment with different methods for puzzle solving. In the future, we would like to expand our feature set to include some of the features discussed in these papers.

Process

Before we began work on the actual jigsaw puzzle solver, we first established our baseline goals for the solver and set some basic guidelines to follow.

Goals and Desired Features

We started by establishing that our baseline solver should be able to match two puzzle pieces together with the assumptions that the background for the image of the puzzle pieces will be a solid color. We also set the following constraints/assumptions for our baseline solver:

- At this stage the solver will only be tested with photos or scans of physical puzzle pieces, not computer-generated puzzles (or puzzle pieces)
- All puzzle pieces have 4 sides and have four distinct corners.
- Each puzzle piece has at least two sides with a concave and/or convex region.
    - Can have one concave and one convex
- Puzzle pieces can be at any orientation
- The border is rectangular (follows from above constraint)
- This solver should be able to at least solve a puzzle of 4 pieces

This was the first goal that we set for ourselves and proved to be more difficult to meet than we had initially anticipated (see Working on the Solver sub-section for more details). The next few goals that we set for ourselves largely involved making improvements to the baseline solver and testing the solver more thoroughly. This included: trying computer-generated puzzles, solving puzzles of up to 100 pieces, and making general improvements to the matching and probability algorithm(s). Our ideal goal would be to have the a jigsaw puzzle solver capable of solving puzzles, both scanned and computer-generated, of up to 1000 pieces.

Obtaining the Data Set

Once we established our baseline goals and assumptions, our next step was to get our data set, which consisted of different images of puzzle pieces. At the start of this project our data set consisted of photos of the puzzle pieces that were carefully taken (see Figure 2 for an example). We later replaced these images due to the variability in perspective, puzzle piece size, orientation, background, and other factors that made it difficult for us to obtain an accurate mask of the puzzle pieces from the images. Our final data set was obtained by scanning the puzzle pieces with

a color scanner, which eliminated many of the variable factors and worries such as background and brightness (see Figure 3 for an example).

Figure 2: Image of puzzle pieces with first method

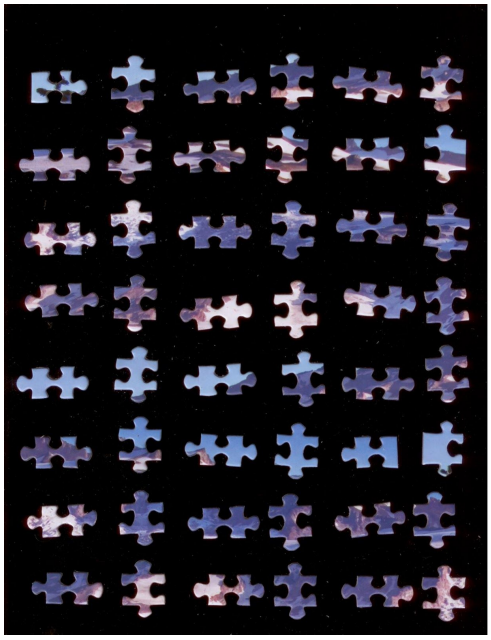

Figure 3: Image of puzzle pieces using color scanner

Working on the Solver

Generating the Mask

In order to improve the accuracy of our solver, we first needed to preprocess the data set and generate a mask for each puzzle piece in a given image. The first part of this involved finding smooth edges and a complete shape, with no holes or "broken" edges, for each puzzle piece. This was somewhat challenging since it was vital not to lose any side attributes, such as whether a side is concave or convex, that allowed us to differentiate between different sides later. Once the puzzle pieces were scanned in we used MATLAB to read

in the image in uint8 format, and generated a wide edge filter that we implemented using a Gaussian Filter. This filter (see Figure 4 and Figure 5) allowed us to search for the gradient of puzzle's image more easily, while also ignoring the noise pixels in the background.

-0.0008	-0.0018	-0.0034	-0.0050	-0.0056	-0.0050	-0.0034	-0.0018	-0.0008
-0.0018	-0.0044	-0.0082	-0.0119	-0.0135	-0.0119	-0.0082	-0.0044	-0.0018
-0.0034	-0.0082	-0.0153	-0.0223	-0.0253	-0.0223	-0.0153	-0.0082	-0.0034
-0.0050	-0.0119	-0.0223	-0.0325	-0.0368	-0.0325	-0.0223	-0.0119	-0.0050
0	0	0	0	0	0	0	0	0
0.0050	0.0119	0.0223	0.0325	0.0368	0.0325	0.0223	0.0119	0.0050
0.0034	0.0082	0.0153	0.0223	0.0253	0.0223	0.0153	0.0082	0.0034
0.0018	0.0044	0.0082	0.0119	0.0135	0.0119	0.0082	0.0044	0.0018
0.0008	0.0018	0.0034	0.0050	0.0056	0.0050	0.0034	0.0018	0.0008

Figure 4: This contains the 9x9 horizontal Gaussian Filter we generated. Notice the lower half of the weights stays positive, while the other upper half is flipped negative.

-0.0008	-0.0018	-0.0034	-0.0050	0	0.0050	0.0034	0.0018	0.0008
-0.0018	-0.0044	-0.0082	-0.0119	0	0.0119	0.0082	0.0044	0.0018
-0.0034	-0.0082	-0.0153	-0.0223	0	0.0223	0.0153	0.0082	0.0034
-0.0050	-0.0119	-0.0223	-0.0325	0	0.0325	0.0223	0.0119	0.0050
-0.0056	-0.0135	-0.0253	-0.0368	0	0.0368	0.0253	0.0135	0.0056
-0.0050	-0.0119	-0.0223	-0.0325	0	0.0325	0.0223	0.0119	0.0050
-0.0034	-0.0082	-0.0153	-0.0223	0	0.0223	0.0153	0.0082	0.0034
-0.0018	-0.0044	-0.0082	-0.0119	0	0.0119	0.0082	0.0044	0.0018
-0.0008	-0.0018	-0.0034	-0.0050	0	0.0050	0.0034	0.0018	0.0008

Figure 5: This contains the 9x9 vertical Gaussian Filter we generated. Notice the right half of the weights stays positive, while the other left half is flipped negative.

After we found the gradient of the image, we got a neater mask by thresholding the value of the gradient. Once that was done, we removed all of the noise that was smaller than 500 pixels, filled in the holes according to the closed edges, and cleaned up the mask by using 4-connected erosion twice to reduce the redundant pixels on the edges.

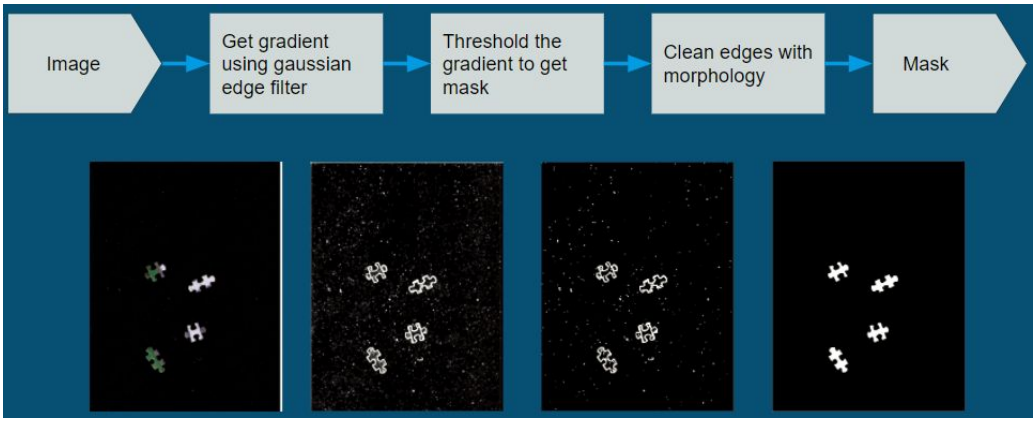

Figure 6: This vividly contains the overview of mask generation procedure, turning a raw image input from the scanner into a clean mask output.

Characterization of Pieces

In order to have a uniform way to refer to puzzle pieces and keep track of the various properties that each puzzle piece has, we created a PuzzlePiece object in MATLAB. This object saved us a lot of work and duplicate code and was used throughout our solver to represent the properties of a puzzle piece.

For each piece, several important properties were found, including centroid, perimeter, corner location, and side shape. These properties would be used in the future for extracting features for combining edges, so we stored them inside the puzzle piece object easy access. Centroid and perimeter were computed using MATLAB's built-in *regionprops* function.

To find each corner, we computed the linear distance from each pixel along the perimeter to the centroid. We rotated the perimeter array so that the minimum distance is located at the start and end of the perimeter array, so that the peaks would not get cut off by the end of the perimeter. The raw distance array was quite noisy, so we ran it through a smoothing vector. The smoothed distance can be seen in the top plot in Figure 7. The corners along the smoothed distance plot are circled. The corner peaks are distinctly more pointed than the peaks caused by convex regions, such as that near index 150 of the same figure. To capture this, we convolved by an ideal normalized corner, which was approximately a normalized upside down absolute value. We then convolved that with the perimeter distance to find the corner correlation. The corner correlation is shown in the bottom plot of Figure 7. The corner correlation has a higher peak at the corner peaks, so we found the 4 highest peaks in the corner correlation for the locations of the corner points. Finally, to ensure that the corner location was the best corner pixel, we found the max of the raw perimeter distance near the predicted corner pixel.

To find the side types, we used the corner locations to our advantage. If there was a large peak in the smoothed distance between the two corner locations, we called that side convex. For example, the side between the first two corners in Figure 7 contains a large peak in perimeter distance, which corresponds with the convex region pictured on the right of Figure 7. If a given side is not convex, we then measured the distance between two corners and compared that with the distance along the perimeter of that side.

If the distance along the perimeter is much greater than the distance between the corners, that side must be concave. If the distance along the perimeter is similar to the distance between the corners, the side is a straight side.

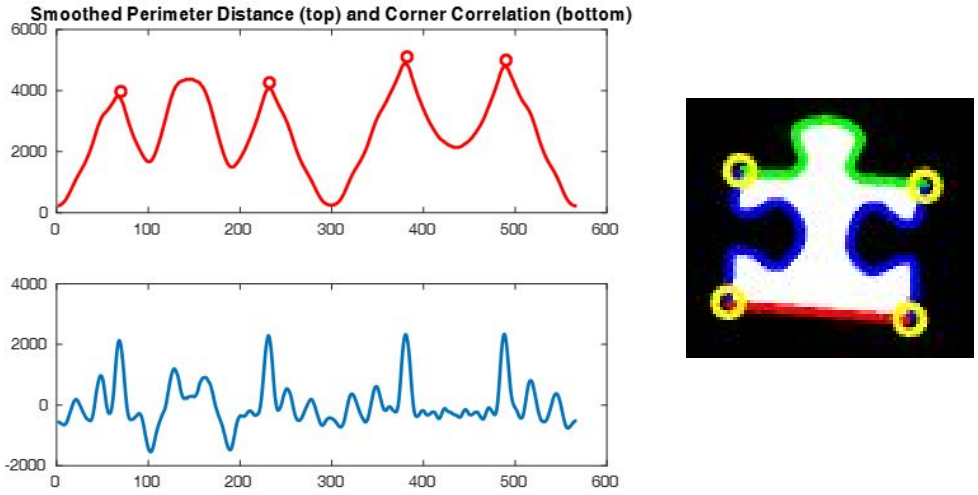

Figure 7: This contains a plot of smoothed perimeter distance and corner correlation (left) and piece being examines (right)

The corner and edge characteristics were computed and verified for 348 pieces. Some of these verified pieces can be found in Appendix A.

Extracting Features for Edge Combinations

A puzzle piece contains two main types of information that we can use to extract features, color and shape. We extracted three features using color information and two features using shape information.

Color features were found by calculating the euclidean color distance in the RGB color space. First the edge pixels for the chosen side of each piece were found and stored into arrays. These arrays were then divided into three corresponding section. This division was because a side of a puzzle piece has three main sections. The smooth flat edges near each corner and the middle that contains the convex/concave portion of the side. However, a smooth side does not have three sections, but a smooth side does not mate with any other pieces. The mean for each segment was then computed and the distance between corresponding segments were was computed using the equation,

$$D = \sqrt{(\Delta R)^2 + (\Delta G)^2 + (\Delta B)^2}. \quad (1)$$

The distances were then normalized with respect to the maximum RGB color distance. The normalized distance for each segment was then used as a feature for the SVM.

The final features were founding using the shape information from two pieces. The first was the difference between the lengths of each sides. In theory, a perfect match will have corners that are evenly spaced, so this distance would be very small. The second shape feature we calculated was overlap. Non-matching puzzle pieces all have slightly different sizes for their heads and holes. As such, we can find the ratio of non-overlapping area compared to quarter area of the smallest piece. First we used rotation and translation to mate the two pieces together at the corners. We then created a mask fused from the masks of the two pieces. This fused mask highlighted the overlapping pixels between the two masks as white. We could then use this mask to find the area of the smallest piece and the area of overlap. The mask used to find these areas is shown in the figure below.

Figure 8: Fused mask of two separate puzzle piece masks that highlights overlap pixels in white.

Finding Score Using an SVM

We chose to use a Support Vector Machine to determine if two pieces matched well using the features described in the previous section. To do this, we trained an SVM using a training set of puzzle pieces. We tuned hyperparameters using a validation set of puzzle pieces for creating the SVM by looping through the bounding box and kernel width parameters for a

gaussian SVM. Finally we ran a set of test pieces through the SVM to test performance.

Getting the Optimal Solution

The first step in finding the optimal solution for the puzzle was to generate a matrix for all of the possible matches (of two pieces) and their scores. This matrix had the following format for each row (which represented a possible match of two pieces): [piece1, piece2, side1, side2, score]. Both piece1 and piece2 were the number representations of the actual puzzle pieces (from PuzzlePiece.Number) and indicated which pieces the were being matched. Similarly, side1 and side2 refer to the sides of the respective pieces that the match was occuring on. For example, a row with values [1, 2, 4, 1, 0] would translate to mean that matching puzzle piece number 1 on side 4 to puzzle piece number 2 on side 1 matches with a score of 0. The scores used in this matrix were all extracted from the output of the SVM.

After generating the matrix for all of the possible edge combinations with their scores, our next step involved actually going through the combinations to get the best possible results. Our original idea was to use a matching algorithm to solve this problem, but we realized that this wouldn't be a good approach due to the fact that most matching algorithms don't allow for the consideration of having more than one match, which is necessary in this case since each puzzle piece has anywhere from two (if it is a corner piece) to four (for non-border pieces) matches that need to be found. Instead, the approach that we took was to simply compare the scores for the possible edge combinations of each puzzle piece.

While we do have a working solution for putting two pieces together and also have a function for getting the best score, we have not yet fully tested the latter and it is likely that it does not return the actual best solution as we took more of a brute-force approach. The main reason that we did this was because the brute-force approach should always provide a solution, even if it is not the best one, while a "greedy" approach could give us the best solution, but would also run the risk of not getting a correct, or even any, solution to the puzzle.

Putting Together the Final Solution

For the final puzzle image, we needed to move and join the puzzle pieces that we determined to fit. For the two-piece solver, we only moved one piece, so we calculated the angle that piece needed to rotate by examining the orientation of the line formed by joining the corners of the side that needed to be joined. We then performed a rotation using *imrotate*, then translated the piece so the corners would match the corners of the other piece. At this point, we are left with two color puzzle piece images with the correct rotation and location. To join the images, we added them together then divided by two in areas of overlap.

Setup & Results

For each of our training, validation, and test sets of pieces, we used an image containing 48 pieces with known matches. We created a matrix containing the true class of each of the piece matches. We created a features matrix with each row of the matrix being all of the features of each possible matching piece. For each of the data sets, there were over 8,000 possible matches, and 82 true matches. The large number of false matches made optimizing the SVM an interesting challenge.

When optimizing the bounding box and kernel width hyperparameters, the scores output were overwhelmingly negative, meaning that the algorithm will tend towards classifying non-matches. Because of this, as we iterated through the hyperparameters, we also varied the cutoff threshold and found the hyperparameters with the minimum cartesian distance from (1,0) to (TPR, FPR). We found this minimum distance occurred at a kernel scale of 1 and a bounding box of 32. This yielded a true positive rate of 84.15% and a false positive rate of 20.45%. This performance is quite disappointing, especially considering that for any puzzle, the number of false matches greatly outnumbers the number of matches. See Appendix B for the hyperparameter tuning data.

To put these numbers into perspective, when solving a 2 piece puzzle, with each piece having two concave and two convex regions, there are 8 possible ways they could fit together (assuming concave must be paired with convex). That means there are 7 incorrect matches and 1 correct match. The probability of our algorithm detecting the one correct match and the 7 incorrect matches is (0.8415)*((1-.2045)^7) = 16.96%. The probability of randomly selecting a pairing and having that be correct is 12.5%. Our algorithm performs better than random chance, but not by much.

We wanted to determine which of the features were most indicative of the pieces matching, so we ran the SVM tuning on the color features and shape features separately. The color features performed very badly, with the optimal trained SVM yielding 60.98% TPR and 57.31% FPR. The color features barely identified matches more strongly than non-matches. The shape features accounted for much of the distinction between matches and non-matches, with an optimal trained SVM yielding 76.83% TPR and 21.63% FPR.

Future Work

Currently, our solver is built to classify an input of two puzzles pieces, but we have laid the framework for being able to solve much larger puzzles through feature extraction and classification. However, the classification is not as accurate as we would like. To improve this, we would like to give the SVM more features to help it classify matches better. For color features, we can experiment with taking color moments and breaking each side up into more than three segments. It is also possible that the color features perform badly due to the sole use of the perimeter pixels, which makes the color features depend on the precision of the mask. For example, if the mask cuts off right at the start of the color portion on one piece, that would yield very different color means than if the mask cuts off at the edge of the background. We would like to improve by finding a solution that is not dependent on just the perimeter pixels. We would also like to optimize the feature extraction process so that it takes less time by performing calculations without having to rotate and translate pieces, which proved to be computationally expensive. Once we are able to classify matches more precisely, our next step will be to write a solver that can handle multiple pieces. First, we will get a complete solution for a 2x2 puzzle. From there we can scale the algorithm so that it can handle much larger puzzles.

Key Challenges

Background Detection

The foremost major problem we met is background detection of the puzzle piece photos. Due to the exposure of camera, and reflection on background, we experienced various troubles, such as darkness, glare, etc., on the photos we took for the puzzle pieces. Since we could not predict the texture of the background, we tried to integrate one algorithm for all the situations at first— to increase brightness and contrast, to adjust saturation, and to remove unexpected glares. We started to generate our background erasing algorithm with averaging background features from the four corners of an image, and going through the whole image to remove the similar pixels. However, to afford for background variability, we greatly sacrificed the sharpness of our mask. We also tried k-mean clusters, combined with boundary

detectors with different structure elements, but they are still not ideal enough for our further processing. To remove the noises in the background, we downsampled the size of the images before background detection. Furthermore, we converted the images from RGB model to other color spaces, such as LAB[3], YCbCr[4]. With the above methods, we made a few processes, but we were not able to achieve breakthroughs due to the various backgrounds of the photos. We eventually decided to opt for color scans of puzzle pieces instead of photographs, since these yielded much more consistent lighting and background. From this we learned that it is better to start from a very controlled data collection process, then later increase the variability of the data collection process if time allows.

Feature Extraction

Feature extraction turned out to be a major shortcoming of our algorithm. We had no real way to test the accuracy of the feature extraction process until we built, trained and optimized the SVM. By the time we had the SVM up and running, we didn't have time to go back and improve the features being extracted. This is the area where the biggest accuracy and time improvements can be seen. We learned that we should have a framework with which to test the effectiveness of an algorithm at an earlier stage in development, so that deficiencies can be spotted addressed more quickly.

Appendix A: Corner and Edge Testing

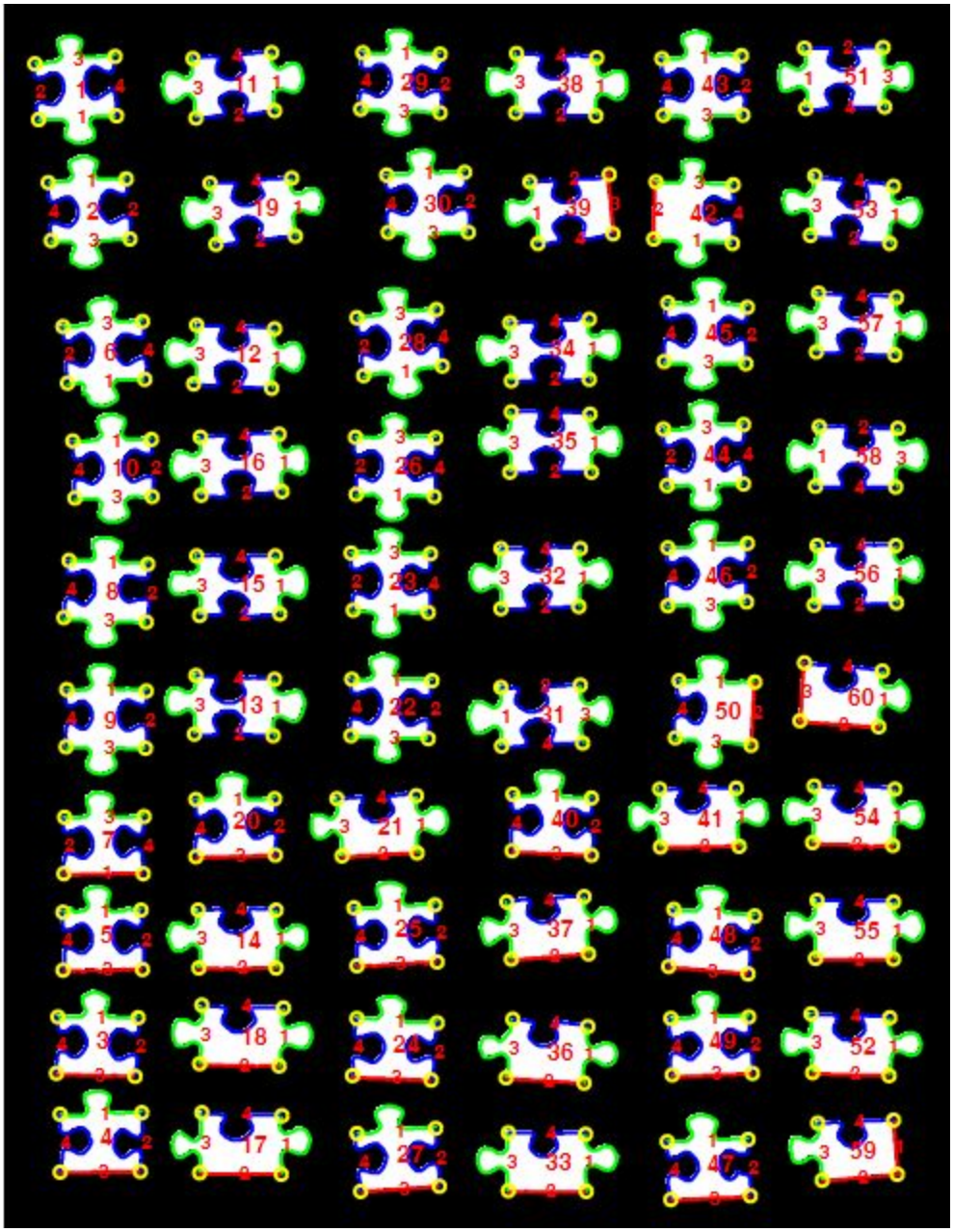

Figure A1: Puzzle pieces with corners circled in yellow and sides colored: red = edge (border), green = convex, blue = concave

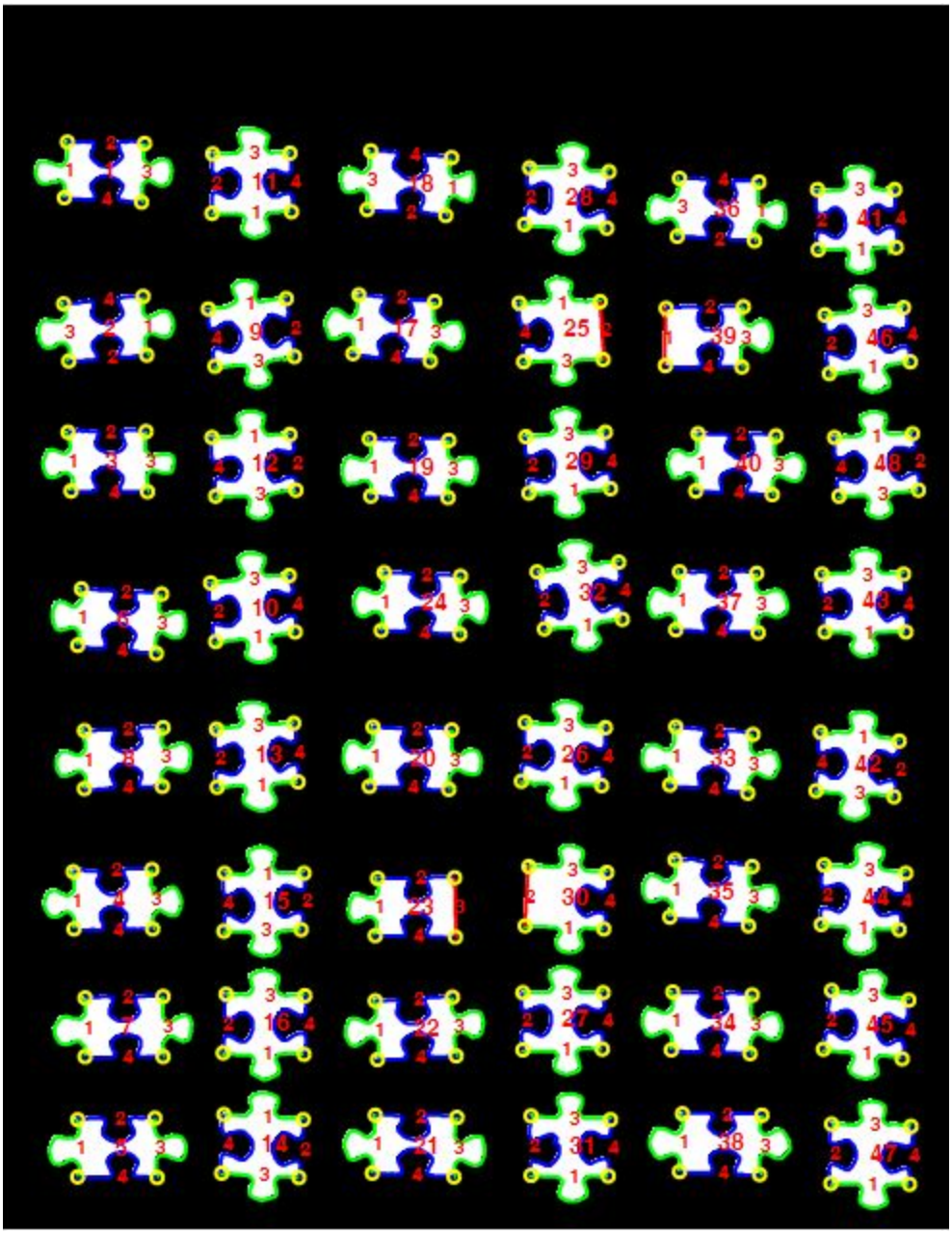

Figure A2: Puzzle pieces with corners circled in yellow and sides colored: red = edge (border), green = convex, blue = concave

Appendix B: Hyperparameter Tuning

Kernel Scale	Box Constraint	TPR	FPR	Dist
0.03125	0.03125	1	1	1
0.03125	0.0625	1	1	1
0.03125	0.125	1	1	1
0.03125	0.25	1	1	1
0.03125	0.5	1	1	1
0.03125	1	1	1	1
0.03125	2	1	1	1
0.03125	4	1	1	1
0.03125	8	1	1	1
0.03125	16	1	1	1
0.03125	32	1	1	1
0.0625	0.03125	1	1	1
0.0625	0.0625	1	1	1
0.0625	0.125	1	1	1
0.0625	0.25	1	1	1
0.0625	0.5	1	1	1
0.0625	1	1	1	1
0.0625	2	1	1	1
0.0625	4	1	1	1
0.0625	8	1	1	1
0.0625	16	1	1	1
0.0625	32	1	1	1
0.125	0.03125	1	1	1
0.125	0.0625	1	0.99988	0.99976
0.125	0.125	1	0.99916	0.998321
0.125	0.25	1	0.99952	0.999041
0.125	0.5	0.02439	0.002279	0.95182
0.125	1	0.036585	0.003839	0.928182
0.125	2	0.085366	0.014635	0.83677
0.125	4	0.097561	0.015475	0.814636
0.125	8	0.097561	0.015475	0.814636
0.125	16	0.097561	0.015475	0.814636
0.125	32	0.097561	0.015475	0.814636
0.25	0.03125	0.865854	0.959333	0.938315
0.25	0.0625	0.804878	0.932342	0.907334
0.25	0.125	0.085366	0.006958	0.836604
0.25	0.25	0.134146	0.016915	0.749989
0.25	0.5	0.256098	0.027711	0.554159
0.25	1	0.341463	0.041867	0.435423
0.25	2	0.426829	0.06238	0.332416
0.25	4	0.439024	0.065259	0.318952
0.25	8	0.439024	0.064419	0.318843

0.25	16	0.439024	0.064299	0.318828
0.25	32	0.439024	0.064299	0.318828
0.5	0.03125	0.341463	0.053863	0.436572
0.5	0.0625	0.353659	0.053263	0.420594
0.5	0.125	0.353659	0.052783	0.420543
0.5	0.25	0.341463	0.052543	0.436431
0.5	0.5	0.341463	0.052663	0.436444
0.5	1	0.414634	0.078215	0.348771
0.5	2	0.414634	0.078095	0.348752
0.5	4	0.463415	0.086852	0.295467
0.5	8	0.512195	0.108685	0.249766
0.5	16	0.52439	0.111204	0.238571
0.5	32	0.573171	0.117202	0.19592
1	0.03125	0.390244	0.044986	0.373826
1	0.0625	0.317073	0.036348	0.46771
1	0.125	0.304878	0.035869	0.484481
1	0.25	0.329268	0.034309	0.451058
1	0.5	0.926829	0.31274	0.10316
1	1	0.743902	0.177663	0.09715
1	2	0.829268	0.247121	0.090218
1	4	0.804878	0.215211	0.084388
1	8	0.817073	0.21797	0.080973
1	16	0.817073	0.216291	0.080244
1	32	0.817073	0.205494	0.07569
2	0.03125	0.487805	0.134717	0.280492
2	0.0625	0.402439	0.066339	0.36148
2	0.125	0.158537	0.041507	0.709783
2	0.25	0.256098	0.03239	0.55444
2	0.5	0.268293	0.034669	0.536598
2	1	0.268293	0.038628	0.536888
2	2	1	0.631358	0.398613
2	4	0.987805	0.476967	0.227647
2	8	0.95122	0.384477	0.150202
2	16	0.853659	0.268954	0.093752
2	32	0.853659	0.271113	0.094918
4	0.03125	0.292683	0.075696	0.506027
4	0.0625	0.182927	0.034549	0.668802
4	0.125	0.121951	0.020034	0.771371
4	0.25	0.231707	0.041147	0.591967
4	0.5	0.134146	0.028311	0.750504
4	1	0.304878	0.033109	0.484291
4	2	0.097561	0.033109	0.815492
4	4	0.231707	0.033829	0.591418
4	8	0.256098	0.039467	0.554949
4	16	0.243902	0.027711	0.572451

4	32	0.243902	0.038988	0.573204
8	0.03125	0.170732	0.06298	0.691652
8	0.0625	0.231707	0.074976	0.595895
8	0.125	0.109756	0.055302	0.795593
8	0.25	0.243902	0.105206	0.582752
8	0.5	0.573171	0.165067	0.20943
8	1	0.353659	0.06154	0.421544
8	2	0.04878	0.016315	0.905085
8	4	0.268293	0.02975	0.536281
8	8	0.158537	0.021113	0.708506
8	16	0.317073	0.049424	0.468832
8	32	0.231707	0.025312	0.590914
16	0.03125	0.52439	0.294266	0.312797
16	0.0625	0.841463	0.347169	0.14566
16	0.125	0.841463	0.347169	0.14566
16	0.25	0.341463	0.073776	0.439113
16	0.5	0.170732	0.053623	0.690561
16	1	0.170732	0.026751	0.688402
16	2	0.243902	0.042826	0.573518
16	4	0.036585	0.020513	0.928589
16	8	0.219512	0.02999	0.610061
16	16	0.109756	0.017874	0.792854
16	32	0.280488	0.042586	0.519511
32	0.03125	0.634146	0.239683	0.191297
32	0.0625	0.012195	0.003119	0.975768
32	0.125	0.012195	0.003119	0.975768
32	0.25	1	1	1
32	0.5	0.414634	0.101607	0.352977
32	1	0.512195	0.132917	0.255621
32	2	0.329268	0.066339	0.454282
32	4	0.45122	0.096569	0.310486
32	8	0.512195	0.228407	0.290123
32	16	0.182927	0.026751	0.668324
32	32	0.195122	0.026152	0.648513

References

[1] Feng-Hui Yao, Gui-Feng Shao, "A shape and image merging technique to solve jigsaw puzzles", Elsevier Science, Japan, 2002. General Information Processing Center, Shimane University. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.462.7781&rep=rep1&type=pdf [Accessed 1 February, 2018]

[2] Gallagher, Andrew C, "Jigsaw Puzzles with Pieces of Unknown Orientation", Eastman Kodak Research Laboratories, University of Rochester, Rochester, NY, 2012. http://chenlab.ece.cornell.edu/people/Andy/Andy_files/Gallagher_cvpr2012_puzzleAssembly.pdf [Accessed 18 Januray 2018]

[3] Kilho Son, James Hays and David B. Cooper, "Solving Square Jigsaw Puzzles with Loop Constraints", Brown University, Providence, RI, 2014. https://www.cc.gatech.edu/~hays/papers/puzzle_eccv14.pdf [Accessed 18 January 2018]

[4] Liang Liang, Zhongkai Liu, "A Jigsaw Puzzle Solving Guide on Mobile Devices", Department of Applied Physics, Stanford University, Stanford, CA. https://pdfs.semanticscholar.org/a10b/0e0750d65dcd65aad52c4b8ed4cff7cb462d.pdf [Accessed 8 Feburary, 2018]

[5] Makridis M., Papamarkos N., Chamzas C., "An Innovative Algorithm for Solving Jigsaw Puzzles Using Geometrical and Color Features", in Sanfeliu A., Cortés M.L.: *Progress in Pattern Recognition, Image Analysis and Applications*, vol.3773, Springer, Berlin, Heidelberg, 2005. https://link.springer.com/chapter/10.1007/11578079_99#citeas, https://link.springer.com/chapter/10.1007/11578079_99#citeas [Accessed 18 January 2018]

[6] Nielsen, T., Drewsen, P. and Hansen, K., "Solving jigsaw puzzles using image features", *Pattern Recognition Letters*, vol.29, no.14, pp.1924-1933, 15 October 2008. http://www.sciencedirect.com/science/article/pii/S0167865508001931, https://doi.org/10.1016/j.patrec.2008.05.027 [Accessed 8 Feburary, 2018]