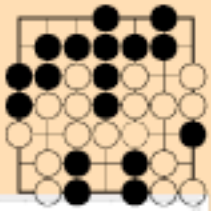
Symphony of Vision

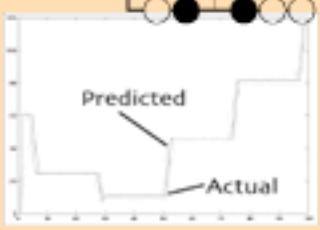
Student Projects in Computer Vision at Carnegie Mellon

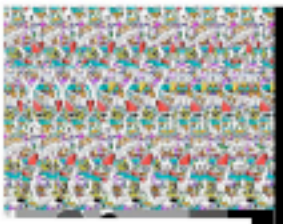
Volume 1

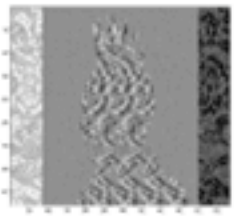


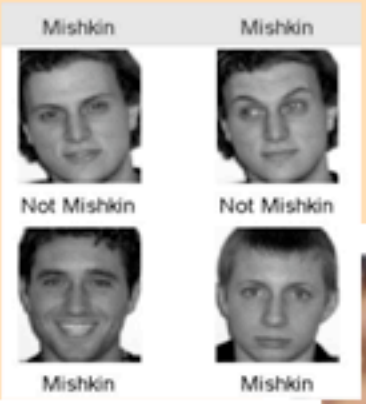



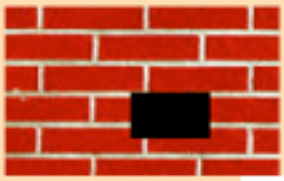















Symphony of Vision

Student projects in computer vision at Carnegie Mellon

Volume 1

May 4, 2004

Contents

Introduction

This volume is a collection of the student term projects in 15-385 – the undergraduate course in computer vision at Carnegie Mellon this year. The students had approximately five weeks to do the term projects, either individually or in teams, starting after Spring break until the last day of class. Given that they were taking at least three other courses at the same time and had other term projects to tackle with, each student had about 30 to 40 hours to propose a project, carry out the basic background research, develop a strategy and idea, implement it in Matlab, test it, write a technical paper, and give a 20 minute presentation. The fact that they can do all these in such a short period of time is nothing short of spectacular. The reports of all their projects, unedited, are now included in this volume to celebrate their accomplishments. The papers speak volumes about their creativity, imagination, ingenuity, technical fluency and expertise, as well as their great stamina and raw talents.

What is most remarkable is that these are not graduate students, but rather undergraduates. In fact, over half of them are sophomore students, and from diverse backgrounds: computer science, electrical and computer engineering, biomedical engineering, mechanical engineering, cognitive science, psychology and physics. The only course prerequisites are a course in programming and a course in calculus and matrix algebra, although signals and systems are strongly recommended. They must have a good mastery of basic programming and mathematical skills, and above all, they must have the passion to learn. Otherwise, they could not survive the rigor of the course, as they would have to, in the span of one semester, learn the fundamentals of signals and systems, pattern analysis, statistical inference, machine learning, Matlab programming, most of the basic concepts as well as the newest and the most advanced techniques in computer vision. For many of them, this was the first time they have ever encountered linear system theory, convolution and correlation, pyramid and wavelets, Fourier transform and frequency analysis, covariance and principle component analysis, Bayesian classification and inference, independent component analysis and blind source separation, spectral graph cut methods and Markov network methods, expectation maximization and clustering, boosting and discriminant analysis, Bayesian belief network and particle filtering, matching and recognition, active contour and dynamic active vision, not to mention the psychology and the biology of human and animal vision. Despite the vast intellectual territory that they had traversed in the short span of four months, they did not learn and understand those materials superficially or merely in concept, but, through six demanding prob-

lem sets, as well as a term project of their choosing, mastered the concepts and techniques well enough that they are able to apply them to solve real and challenging problems.

This is almost an impossible feat. When you read these papers, despite their shortcomings, you should recognize and admire what amazing achievements they represent. Each paper in this collection is a piece of remarkable music on its own right, and together they form a beautiful and powerful symphony, spanning almost the entire dynamic range in computer vision, and representing the blossoming of the creative processes in these powerful young minds.

Conductor and Instructor:

Tai Sing Lee
Associate Professor
Computer Science Department
and Center for Neural Basis of Cognition
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA.
May 4, 2004.

Teaching Assistants:

Jiayu Pan and Hua Zhong
Ph.D. students in Computer Science
Computer Science Department
School of Computer Science
Carnegie Mellon University

An Automated Jigsaw Puzzle Solver

Ka Wing Ho
kwho@andrew.cmu.edu
Carnegie Mellon University

Kermin E. Fleming
kfleming@andrew.cmu.edu
Carnegie Mellon University

Abstract

We propose an automated method of solving jigsaw puzzles. The classic jigsaw puzzle involves assembling an assortment shaped pieces into a single contiguous picture. Rather than focusing on solving the puzzle using the visual information contained on each piece to realize a completed picture, the naïve approach used by most human puzzlers, our approach utilizes the shapes of the pieces to determine the solution of jigsaw puzzle. To this end we view the puzzle as a graph of edge connections. We then apply various physical heuristics to remove edges from the edge graph. Then, we rank the edges of the graph using a curve-matching algorithm. Finally, we recursively construct the puzzle by applying additional physical constraints to the edge graph. Our algorithm has demonstrated the ability to solve a twenty-four piece puzzle.

Introduction

The jigsaw puzzle debuted in the eighteenth century, and has captured the fancy of both children and adults across the generations. The jigsaw puzzle consists of a collection of shaped pieces that, when assembled, form a picture, with typical puzzles containing between a few dozen and a few thousand pieces. With some intuition and a little practice, it is possible for human puzzle solvers to learn techniques that can greatly reduce the time needed to correctly assemble a puzzle. Typical human strategies (at least the ones that authors used to solve the puzzles in question) involve a combination of pictorial and physical matching, e.g. assembling uniquely colored pieces and assembling the puzzle boarder first. However, when using a computer to piece together a jigsaw puzzle, many intuitive human strategies become difficult and, in some cases, impossible. For example, physically placing and comparing the edges of a puzzle via filtering may require rather intensive computation, thus necessitating the development of other, faster edge matching strategies.

In 1964, Freeman and Garder, [1] made the first attempt to solve a jigsaw puzzle using shape information, and since that time several improvements have been made to improve both the robustness and to increase the maximum number of pieces that can be solved. Most recently, in 2002, Goldberg, Malon, and Bern [2] produced an automated solver capable of completing a 204-piece puzzle in 20 minutes.

Our approach to solving puzzles most closely resembles that of Wolfson [3], whose solver was capable of assembling a 100-piece jigsaw puzzle. We make use of a set of simplifying assumption suggested by Wolfson [3]. Namely, we assume (1) that the puzzle has a well-defined rectangular boarder, (2) that each interior piece has four neighbors (one each on the piece's left, right, top, and bottom), (3) that pieces join with their neighbors by the concave portion of one piece locking into the convex portion of the other piece, (4) that four neighboring pieces for a corner '+' at their mutual border. However, we differ from Wolfson in that we do not construct the boarder of the puzzle first. Rather our algorithm constructs the puzzle by attempting to limit the number of edge matchings that it must 'guess'. For example, if somehow the algorithm knows that a piece must be located at a certain position, it will place the piece at that location. Edge and corner pieces (with the exception of the initial start location) do not receive any sort of priority from our algorithm. Additionally, our algorithm uses backtracking, an element not present in previous some previous algorithms [1], [2].

Our approach relies heavily on the concept of curve matching. Essentially we seek

describe each edge in a manner that facilitates a comparison with other edges so that a best-fit matching can be obtained. To accomplish this goal we used a modified version of the 'chain codes' as described in [4]. Once chain codes have been determined we use a simplistic version of the string matching proposed by Wolfson [5].

Our algorithm has demonstrated the ability to solve a twenty-four piece jigsaw puzzle in roughly fifteen minutes (this figure includes substantial preprocessing time, the actual solving algorithm requires on the order of thirty seconds). We have not tested our solver on other puzzles, and we used the tested puzzle to assist in the development of our algorithm. Therefore, some additional testing would better confirm the generality and robustness of our algorithm.

Methods

Our algorithm can be partitioned into four steps:

1) Data Collection: Although not technically part of our algorithm, out method of data collection merits some discussion as it helped to motivate the development of some critical parts of our algorithm. Here, we produce a digital representation of a physical jigsaw puzzle which lends itself to our solving techniques.

2) Preprocessing: In this step, our algorithm discerns important information about the nature of the pieces of the puzzle. Specifically, we classify the edge types of the puzzle, determine the locations of the corners of the puzzle, encode the edges of each puzzle piece as a chain code, and calculate the corner-to-corner distance and arc length of each edge.

3) Affinity Matrix Construction: We use the results of step B to create an affinity matrix of edge matches. Using the data, we eliminate a large number of choices, first by using several forms of physical discrimination and second, by using applying some logical constraints.

4) Puzzle Assembly: In this stage we use a recursive mechanism to assemble the puzzle starting with one of the corner pieces. To avoid incorrectly assembling the puzzle and to allow some error tolerance, we use backtracking to correct any mistakes made in putting together the puzzle.

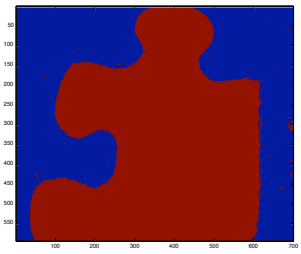
A. Data Collection

Representing a physical object in a digital format can be a difficult problem. Most objects found in nature are continuous, that is natural objects exhibit continuous and smooth transitions. However, despite the high levels of precision and attempts to induce smoothness and continuity, digital objects are inherently discrete, and thus a certain amount of error, be it noise or otherwise, is introduced into a digital representation of a natural object.

To create a reasonable digital representation of a puzzle, we used a high resolution scanner to scan in each individual piece. Initially, we attempted to scan the side of the pieces which contained the picture because we anticipated using the pictorial information as a means of assembling the puzzle. However, this method resulted in somewhat ambiguous piece boundaries. Additionally, the scanner had some difficulties in faithfully representing the pictorial data (Fig.1). Upon encountering these obstacles, we revised our general approach and focused on curve matching rather than texture and picture matching. As we only use the shapes of the puzzle pieces, we were able to make some adjustments in our scanning procedure. We decided to scan the backsides of the pieces, to provide more uniform image on which our algorithm could operate more robustly. To increase the edge contrast of the pieces, we applied a black background to the scanner to reduce noise around the piece edges. This resulted sharper piece boundaries.

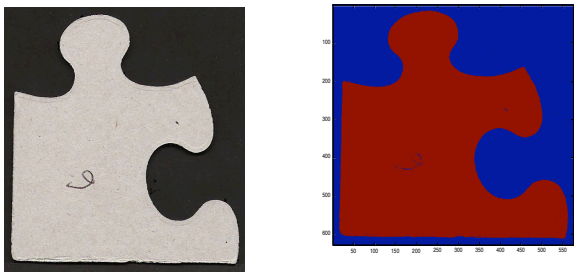
Next, we used a threshold to transform each of the puzzle images into a binary matrix representation (Fig. 2). Our algorithm actually received these binary puzzle piece representations.

(a)



(b)

Fig. 1. (a) A representative image of the first method. (b) The resulting binary image. Note the noise around the edges.



(a) *(b)*

Fig. 2. (a) A representative image obtained using the second data collection method. (b) The resulting binary image. Note the noise reduction from figure (1).

B. Preprocessing

Preprocessing begins with the classification of the edges of each puzzle piece. There are three broad categories of edge classification: flat edge, convex edge, concave edge. The edge type is calculated by threshholding the sum of the rows and the sum of the columns of the binary image. Each edge type produces a particular sum spectrum, for example a flat edge produces a somewhat broad region of nearly constant sums at the edge of the piece. We determined these thresholds empirically, based on the size of the puzzle piece, and they seem to be reasonably robust as the thresholds had a margin of error of roughly 30%. As an example of how the edges are classified, consider figure 3.

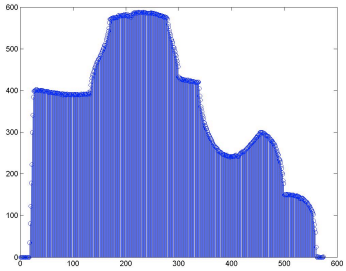
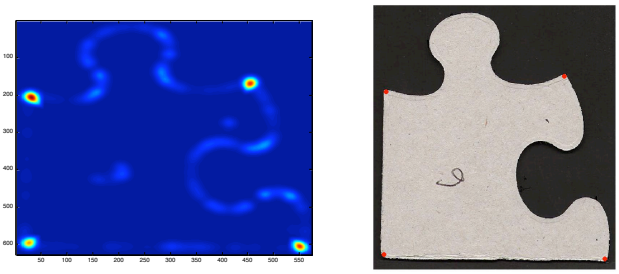


Fig. 3. This sum spectrum corresponds to the columns of Fig. 2.b. Note that the flat region on the left corresponds to the flat left edge in Fig. 2b. and the peaky region on the right corresponds to the concave right edge in Fig. 2b.

After each edge is classified, our algorithm next determines the location of the corners of each piece. To detect the corners, we employ a version of the Harris corner detector. Due to the sheer size of the puzzle images (typically on the order of 700 x 700 pixels) the solver utilizes a large Harris filter to adequately capture the corner images. However, the use of the large Harris filter results in somewhat broad regions of peak filter response. Thus, to ensure that four different corners are selected (rather than selecting the same corner four times), when our algorithm discovers a corner, it suppresses the filter response of the area surrounding the corner. Thus, the algorithm is guaranteed to discover four unique corners. Due to noise and other non-idealities in the binary images, the corner detector does not necessarily detect the exact location of the image corner, but does identify a point within five pixels of the actual corner.



(a) *(b)*

Fig. 4. (a) The Harris filter response for a puzzle piece. Note peaky corner response. (b)Detected corners plotted on the original piece image.

Next, the solver determines the edge length of each edge of the puzzle. The edge length is simply the Cartesian distance between the two corners of each edge. Our algorithm uses edge length to help construct the affinity matrix.

The next step of the preprocessing phase is the characterization of the curves of the edges. This part is mainly accomplished through contour construction and curve matching. In order to get an accurate encoding of all the edges, we have to first obtain a good contour map, which has one along the contours and zero in all other places in the puzzle. The algorithm to construct the contour map is as follows:

--First, look for the left most pixels with one in each row and set that pixel to one in the output image.

--Set a flag to 1 and keep reading from left to right along the row until you see the first zero. Set the last pixel with value 1 the program saw to 1. Set the flag back to zero.

--If we see another one before we reached the end of the row, go back to step one. Keep repeating the two previous procedures until you have scanned all the rows.

--After scanning all the rows, repeat the whole procedures for all the columns.

Essentially, the above algorithm is just setting the boundary of pixels with value 1 in each row.

The next step is encoding the edges of each piece into chain code. A chain code is a string of numbers each of which represents offset angle along the contour. To simplify string comparison and to compensate for small errors in chain code assembly, we quantize the angles into 16 levels, as shown in Fig. 5. Each of them has a tolerance of +/-(pi/16). Using the corner location information from the corner detector, we locate a particular corner in a puzzle. First, we start from the bottom left corner and gradually climb up the left edge of the puzzle till we reach the top left corner. In each step up the edge, the program scans all the surrounding pixels in the contour map by sweeping in 360 degrees with a fixed radius. Then, among all the points it picks up, the algorithm computes the deviation of each point from the current point as expressed by an angle measured in clockwise direction. The algorithm then picks the point whose angle deviates the least from the previous angle of deviation. The angle of deviation is the key which is then encoded into a corresponding number.

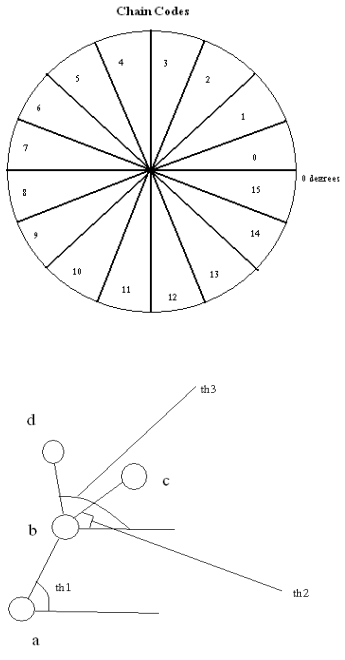


Fig.5. This figure illustrates the idea of picking the point that deviates the least from its previous angle

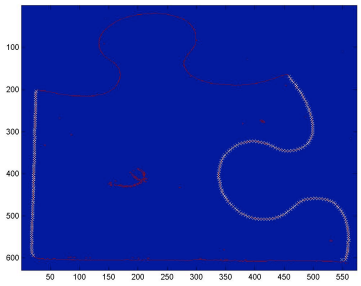
Assuming that in the previous step, we swept at a and chose point b. At b, we swept again and picked up both point c and point d .Here, we would choose point c as the next point because the difference between the angle of deviation between a and b is closer to the angle of deviation between b and c than that between point b and d

Having computed the chain code of each edge on all the pieces, the program will further compute the difference between each consecutive entry in the chain code. This string of new number is called the 'first difference of the chain code' as described in [4]. The first difference of the chain code is the final description that we will use in the curve matching procedure. One thing special about the 'first difference' is that it is orientation invariant. As we scanned in the pieces, they are not likely to be perfectly aligned. It may be tilted for a small degree. This small error in alignment could pose a serious problem when chain code is used directly for comparison. By instead measuring the rate at which the slopes of the edges change, this problem can be overcome because of its special property of orientation invariant.

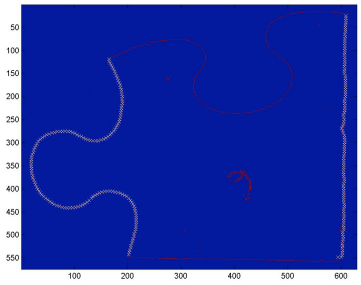
For every puzzle piece, we encode the piece in four orientations by rotating it by 90 degrees in clockwise direction three times. For each orientation, we encode both the left edge and the right edge of the rotated piece. Therefore, for each piece of puzzle, we would have a total of eight different strings of numbers describing the four edges of a puzzle piece. On rotating the puzzle pieces three times by 90 degrees, we are actually encoding each edge twice. One representation is encoding as if the edge is a right edge and the one is for the left edge. These two seemingly redundant representations can account for the fact that a puzzle piece may be connecting to another piece in a different orientation from which it is scanned in. So, the right edge on a scanned-in puzzle piece may end up connecting with the right edge of another piece after rotation of 90 degrees twice. Through encoding the edges in these two different orientations, we can tell the orientation of each piece with respect to others. The details in computing the orientations will be discussed in details in the section of assembly.

One important factor determining the accuracy of the chain code is the radius of the circle that the program sweeps in. Different radii have been tried when computing the chain code.

For a radius that is too small, say 2 pixels, the sweeping may not be able to look far away enough to pick up a point. In these cases, the program may end up picking its own previous point. In overcoming this issue, we can increase the radius. As a secondary measure to ensure that we are not actually picking the previous point, we have also set a threshold to the maximum size of the angle of deviation. If for some steps, the smallest possible deviation is larger than the threshold, we would go one step backward and pick the second best point in the previous step. Another issue when making the measurement is that the coordinate fed back from the corner detector may not exactly be on the contour. So, the program should look around its surrounding 10-20 pixels on the same row and pick the most suitable one that is on the contour. For instance, if the program is in the process of scanning its left edge, it should try to pick the leftmost point on the contour that it can see and pick the point in a similar manner for the right edges. That is the way to make sure that the detection actually started on the contour.



(a)



(b)

Fig. 6. The 'climbing' along the left and right edges of a puzzle piece. (a) the original scanned-in image and (b) the piece after rotations.

In the final stage of preprocessing, our algorithm determines the arc length of each edge. Edge arc length measures the physical length of the curve between the two corners of and edge. In calculating the chain codes, we took care to maintain a nearly constant sweep radius. Thus, we take the length of the chain code as a rough estimation of the arc length of an edge.

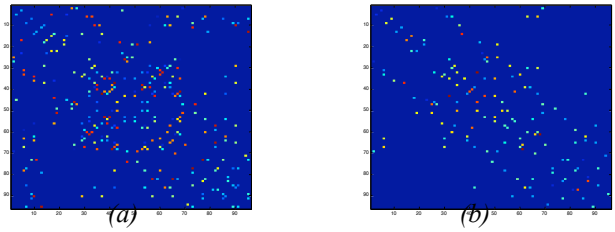
C. Affinity Matrix Construction

During the preprocessing step, our solver calculates a volume of physical data about each edge of each puzzle piece. We exploit the knowledge gained during the preprocessing phase to eliminate edge matchings from consideration. We use a matrix to represent possible edge matchings, with a non-zero value representing an edge matching that we consider to still be a possibility. Initially, all edge matchings are considered. First, we eliminate entries that, for physical reasons, obviously cannot match. In this phase, we remove flat edges from consideration, since flat edges must be boarder pieces. Additionally, we remove matchings between edges on the same piece, since clearly we cannot bend the puzzle pieces during assembly. With this reduced matrix, we next remove matchings whose edge classifications do not allow a fitting. From assumption (3), we can eliminate edge matchings that pair two convex edges or two concave edges. At this point, we have removed from the matrix all illegal edge matchings.

We use two cycles of thresholding to eliminate edges that are physically unlikely to fit together. From assumption (4), we consider that pieces must come together to form a '+' shaped junction. Thus, if two edges match, the distances between their respective corners should exactly match. Similarly, since the two pieces should fit together snugly, their arcs should not only be congruent, but should also have similar lengths. Ideally, if two edges match, the differences of their edge and arc lengths should be exactly zero. However, since we cannot assume that our data is perfectly accurate, for reasons noted above, we must allow some tolerance in our thresholding. With the assistance of some empirical evidence, we determined that tolerances of fifteen pixels

and four chain lengths (this roughly corresponds to fifteen pixels, as chain lengths are somewhat variable) would be suitable for our discrimination. First, we sieve our matrix of edge matchings using edge length. All matchings whose edges' lengths differ by more than fifteen pixels are eliminated from consideration. Then, we filter the matrix by considering arc length. As before, any matchings whose edges' arc lengths differ by more than four chain lengths are removed from the matrix. At this point, all remaining edge matchings are assigned chain code scores as their entry in the affinity matrix. Chain code scores are computed by calculating the sum of squares error of the first difference of the matching edges chain codes. In the case that the edges have different length chain codes (and most of them do), the solver calculates the difference of the chain code lengths; the smaller chain is then 'slid' over the larger chain one length at a time and the chain code score is calculated at each step. The minimum of these chain code scores is selected as the overall score for the edge matching.

At this point, the affinity matrix the affinity matrix has been greatly reduced. Although, we can no longer remove edge matchings for physical reasons, we can still remove edges for logical reasons. Our reductions to the affinity matrix have resulted in forcing certain edge matchings to occur, that is, some edges can only be mated with one other edge. Thus, we deduce that the two edges must match together in the final puzzle, and we can remove any other edges matchings involving the pair. In this manner, we can further simplify the affinity matrix.



(a) (b)

Fig. 7. (a) The affinity matrix after thresholding out edge length matching impossibilities. (b) The final affinity matrix.

D. Puzzle Assembly

The affinity matrix saves us a lot of work for assembly. The matrix has been reduced to the point that almost only one mate exists for a given edge. Thus, we can look up edge matchings directly in the matrix rather than doing any computational matching. We represent the puzzle solution in a 47X47 array structure, which we will call it the 'big picture' henceforth. The reason for this structure is that we do not assume the dimensions of the whole puzzle in general though we can make the assumption that we are solving a 24 pieces puzzle. Therefore, if we manage to pick a corner piece and put it in the middle of the 47X47 array structure, we can span in any directions by as many as 24 steps. In general, we can represent the solution of a jigsaw puzzle by using a (2*w-1) by (2*w-1) array where w is the number of jigsaw puzzle pieces. For each cell in the 47X47 array, there will be 3 numbers that gives the details of each piece after assembly. The first one corresponds to the piece index, the second one is the number of 90-degree rotations in clockwise directions and the third one is the number of neighbors of that particular piece.

We have incorporated an algorithm that is similar to the A* algorithm in the assembling of the pieces. A* allows backtracking in cases where the program makes mistakes in the middle of the assembly, which allows us to more robustly solve the puzzle. Essentially, the algorithm just recursively picks the best possible guess in each step and it keeps growing from that guess till it runs into an error, or until it runs out of pieces (i.e. a solution is obtained). Here is the algorithm:

--First, get a corner piece and use that as the seed to build the whole puzzle. Input the seed as a parameter into the recursive method.

In the recursive method:

--Look at the rows that correspond to the edges of the chosen piece. Pick those rows that contain at least one non-zero entry. If there isn't any row that has non-zero entries, return with an error.

--Sort the list of non-zero entries in ascending order. Dequeue the element with the lowest score and propose that it is the next piece to be implanted into the big picture. Compute the appropriate orientation and coordinate of the proposed piece in the big picture.

--Check to see if there is any piece that is already placed in the surroundings of the proposed coordinate. If there is any, verify that the piece to be implanted will actually match those pieces.

--If the proposed piece to be implanted matches with all of its existing surrounding pieces, update the information in the big picture to reflect the changes. Input this updated copy of the big picture into the next level of recursion.

--If there is a conflict with its surrounding piece, go back to step 3 and pick the next available entries. If there isn't any, return with an error.

--Recurse until there are no more pieces left.

Orientation is another major issue in solving the jigsaw puzzle. Not only does it allow us to see how the computer actually assembles the pieces, it also helps in the verification step to ensure that two neighboring pieces connect. By enumerating all the possible connections between edges of different orientations, we manage to compute the number of 90 degree clockwise-rotations needed in order to fit two pieces together. The formula is as follows:

(6-p+a+aori) mod 4

where p = the edge number to which the newly implanted piece will connect to the current piece

a = the edge number of the current piece to which the newly implanted piece is supposed to connect to

aori = the number of 90 degrees clockwise-orientation that the current piece has gone through

Another problem that the program ran into during assembly is running into a 'sandbox'. Imagine that the program is building the puzzle in an inward spiraling manner. It starts off at the lower left corner, go two rows up, then two columns to the right, two rows downward and finally one column to the left. The next piece that the program will pick is likely to be the center piece of this 3X3 squares. Afterwards, the program will break because all four edges of the center piece are already found. Thus, the program itself runs into a box from which it cannot escape. Whenever we come into such situation, we would look in the affinity matrix for any row that contains non-zero entries, substitute that edge in as the current piece and searches for the next available edges again. This technique solves this issue of 'sandboxing' during assembly.

Results

```
>> final

final =

     1     7    13    19
     2     8    14    20
     3     9    15    21
     4    10    16    22
     5    11    17    23
     6    12    18    24

>>
```

Fig. 8. This is the final output of our solver. Note that each number corresponds to a puzzle piece. This is a correct solution

The program managed to give a correct solution starting from any of the four corners. Orientations can also be included in the solution since it is directly available from the 47X47 'big Picture'. For each run, it took about 20 minutes on a typical cluster machine. Most of the time is spent on computing the chain code and detecting the corners. The 'big picture' mentioned in the section of assembly is fed into another function to determine the appropriate locations of the pieces in the end.

Discussion and Conclusion

We chose this project because we thought it would both relatively simple and somewhat rewarding. We were wrong on one count: developing a robust automatic jigsaw puzzle solver is exceptionally difficult. Jigsaw puzzle solving is a rich problem in computational geometry and pattern recognition, and clever, insightful techniques are required to solve even the smallest puzzles. These techniques were, of course, not immediately obvious, and it took us roughly a week of

thinking and testing to come up with a rudimentary algorithm. After a week of coding, we developed a working solution and spent a few days generalizing it as best we could. In terms of work division, both members were present for ninety percent of the time spent on the project.

Although we learned a good deal about integrating code and group development, curve matching stands as our greatest learning experience. Curve matching was the most difficult technique employed in this project and it took the largest amount of time to implement. Curve matching by itself is a pretty neat idea. However, obtaining the representation of a curve is the pre-requisite for a successful curve-matching. This has prompted us to learn the idea of chain code. In the process of finding the chain code, we were also pushed to come up with an effective way to represent the contour of the puzzle pieces. Through this process, we have learned to formulize a problem into a step by step solution.

Although our solver manages to assemble a 24 piece jigsaw puzzle, it has several major weaknesses. The most obvious and debilitating weakness is a lack of generality. The solver uses several hand tuned parameters to complete its task. Although these parameters make some sense (especially those involved in the calculation of the affinity matrix), attempts to solve other jigsaw puzzles would likely end in failure. A particular point of weakness is the function that determines the classification of an edge. This function heavily depends on empirically calculated parameters to operate correctly. A possible solution for this problem is to use machine learning. Possibly, a large library of puzzles could be examined, and the solver could learn how to choose correct edge recognition parameters based on the size of the puzzle piece.

Another weakness of our solver is its runtime. The solver takes roughly thirty minutes to solve the twenty-four piece puzzle. Considering that Malon, Goldberg, and Bern [2] solve an one-hundred piece puzzle in three minutes, our code runs quite slowly. Although some of the slowdown is caused by Matlab, the main reason that our code runs slowly is our over use of filtering. To ensure that we obtain accurate corner locations, we use the Harris filter several times. However, filtering is extremely computationally expensive, and thus if we could cut down on it we could drastically improve the speed of our solver. A possible solution is to develop a means of translating the corner locations when we rotate the pieces.

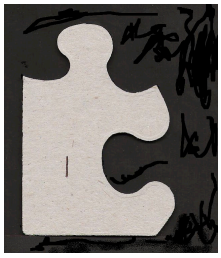
Our curve matching algorithm, perhaps our greatest triumph, was not perfect. Occasionally, sometimes without warning, the chain codes produced would have extremely high scores. Indeed, our implementation of the curve matching in this project is pretty crude. Curve matching is a useful technique applied extensively in archeology to assemble broken pieces. In some other works that were shown in [2],[3], curve matching is applied in a more general manner to objects which don't necessarily have angles in it. Therefore, instead of computing the chain code separately for four edges separately, we can generalize the problem by computing just a set of chain code for the whole contour. Afterwards, we can do string matching to obtain the portions to which two contours match. In this way, we wouldn't have to go into rotating the image and finding the corner every time. This would essentially save a lot of computation.

While our solver succeeds at a single puzzle, there are several future improvements that we could make. First, we should test more puzzles to ensure that our solver produces correct results for other puzzles, including those of different sizes. Additionally, we could develop a mechanism by which our solver could support randomly oriented pieces. Currently, our solver outputs a solution matrix. While simply achieving a correct solution is a great success, we could upgrade the user-friendliness of our solver by having it actually outputting an image of the completed puzzle.

In conclusion, we developed an automated jigsaw puzzle solver capable of solving a twenty-four piece jigsaw puzzle solver. Using a combination of heuristic techniques and curve-matching, our solver is capable of outputting a correct puzzle solution in approximately twenty to thirty minutes.

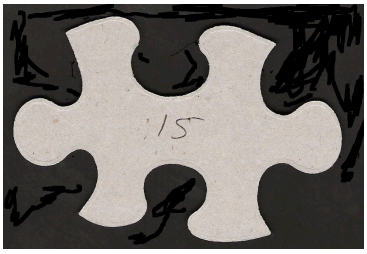
Some Pictures:

Fig .9 The Actual Jigsaw Puzzle that we solved



(a1)

(a2)



(b1)

(b2)

Fig.10. Some Example Piece: (a1): backside of piece 1 (a2) Front side of piece 1; (b1) backside of piece 15, (b2) front side of piece 15

Reference

[1] H. Freeman and L. Gardner. Apictorial jigsaw puzzles: The computer solution of a problem in pattern recognition. *IEEE Trans on Electronic Computers* **13** (1964) 118-127.

[2] M. Bern, D. Goldberg, and C. Malon. A global approach to Automatic Solution of Jigsaw Puzzles. *Proc. Symposium on Computational Geometry*, 2002, 82-87.

[3] H. Wolfson, E. Schonberg, A. Kalvin, and Y. Lamdan. Solving jigsaw puzzles by computer. *Annals of Operations Research* **12** (1988), 51-64.

[4] R. Jain, R. Kasturi, and B. Schnuck. *Machine Vision*. 1995, 423-434.

[5] H. Wolfson. On Curve Matching. *Annals of Operations Research* **12** (1990), 483-489.

Biography:

Ka Wing Ho is from Hong Kong and is a sophomore in the bachelor program of Electrical and Computer Engineering. His major interest is in machine learning and computer security.

Kermin Fleming is a sophomore in the Electrical and Computer Engineering Department at CMU. His interests included distributed architecture and hardware verification. Kermin hails from Lexington, Kentucky.

Height Detection in Clothing Store Using Video Camera

Author Vidit Nagory
Carnegie Mellon University
Email: vnagory@andrew.cmu.edu

Abstract

In this paper I develop a method to find out the heights of customers entering a clothing store. For a clothing store like GAP and Banana Republic it would be very helpful if they knew the size of the customers who are entering their stores. For example, a GAP store could be doing very well having huge sales, but now we get the information that 80% of the customers entering that GAP store wear clothes of Large Size whereas 90% of the sales in the store have been in the medium category. This information could prove to the management that there is a problem with clothes in their large category because people are not buying them. Using this information the management could then take steps in finding out what the problem is and how it could be solved.

1. Introduction

Our goal is to develop a system that finds the heights of all the people entering a store. This data then could be then used to generate a database for the store that gives the store statistics of people belonging to a size range. For example the following data might be generated that shows the number of people belonging to a particular size that entered the store on that specific day:

Dates	Small	Medium	Large	Xlarge
1/1/2005	10	60	15	5
1/2/2005	15	40	4	10
1/3/2005	20	25	10	20
.				
.				
1/31/2005	11	30	5	9

The solution to this problem involves tracking people as they enter the store and calculate their heights. The tracking of people ensures that a single person is not counted twice. The solution also needs to have some sort of fast background modeling technique, so that the location of people can be detected by simple background subtraction.

Another constraint of this problem is that the solution has to be general enough to work in all the stores. The store should be easily able to install the camera system without much and if possible any calibration. Preferably a cheap camera should be used to solve the problem so that the stores do not incur expenses in buying expensive camera.

To solve this problem we can either store all the images recorded in a day and then analyze them later. This would require additional hard disk costs. So a solution that works in real time is preferable.

The following paragraphs outline the techniques I tried to solve this problem and then the final technique that I chose. All the merits and problems of all the techniques are also mentioned.

2. Background Modeling

The easiest and fastest way to find a person moving in front of a camera is background subtraction. So we have to always have a picture of the background stored in the memory.

For background subtraction to work in a changing environment we have to constantly update the background. For example consider the scenario that we have a fixed picture of the background taken in the morning, now if we are extracting the foreground objects by background subtraction, this method might work till the evening. In the evening the whole image will get darker and the entire image might be detected as foreground if we subtract the image of the background taken in the morning from it. So there is a need to constantly update the background.

There are many papers with techniques for adaptive background modeling available on the internet. Some of the techniques I tried are given as follows:

1. Median Value Pixel: This technique involves storing all the images of the day. Then we take the median value of each pixel taking values from all the images. For example if three images are [1 1 1], [1 2 3], [2 4 6] Then the image with median values would be [1 2 3]. This technique assumes that most of the images seen in a day were of the background, so that the median value of a pixel would always correspond to the background. The problems with this technique are:
    - It is really slow
    - It calculates a single background picture, thus will cause problems with images in which the illumination has changed. However with proper thresh holding it seems to give reasonable results with even such images. In addition we could calculate a background image for a set of images recorded in every 20 minutes, to reduce such illumination change problems.
    - It assumes that most pictures are of the background, which in the real case might be an invalid assumption
    - It does not work in real time and needs all the images to be fed into the program to calculate the background.

2. Mode Value Pixel: This technique is the same as the Mean Value Pixel except that the pixel value in the background image is the mode of all those pixels. This technique has the same problems as mentioned with the Mean Value Pixel.

3. Deviation Thresh hold: In this technique we start with one initial picture of the background. Now when the next frame is read, we compare the new frame with the stored background pixel by pixel. If the new value of the pixel – old value of the pixel is within a certain thresh hold we replace the old value of this pixel with the new value, considering that there is a minor illumination change in the background. However if the new value is outside the threshold, we assume that this pixel belongs to the some object in the foreground, so we take the old value of the pixel which we believe is the background.

This method gives really good results. The only problem is that it is slow as it has to operate on all the pixels in the image.

4. Constant Update: This method also starts with one initial image of the background. Now when we get a new frame, we compare the new value of a particular pixel with an old value. Then we calculate the deviation of the new value from the old value. Based on the deviation we calculate x which ranges between 0.5 and 1. If the deviation is large x is large and vica versa. Now we calculate the new value of the pixel as : x*(old value of pixel)+(x-1)*(pixel value new frame).

The problem with this method is that lots of 1 pixel noise can come into the picture. This considerably has bad effects on the results of this method. Another problem is that it is slow.

5. Final Method Chosen: At this point I realized that the speed of the algorithm would always be slow if I operated on each pixel, like I have done in all the methods mentioned before. I figured out that to make it faster I would need to operate on a group of pixels at the same time. Now the only change that could be happening from one frame to the next would be the subject appearing in the new frame or the subject (the person whose height we have to find out moving in the image). Now if by background subtraction and using some basic techniques I can get a boxes containing all foreground objects, then I can make my new background image= the new frame read in, and then replace all the pixels in these boxes with the pixels in the old background.

This method worked really well. The algorithm was really fast, as I did not have to compare all the pixels. The results of the background were not as clear as with the methods mentioned earlier, but were good enough to be used as background for the height detection problem.

3. Boxing

This is the method used to detect objects in the background. After subtracting the background from the new frame we get an image result. Then using a good thresh hold I create a binary image. Then I run a blocker program over the binary image that takes out noise. The blocker program sums up 5X5 blocks in the image. If the sum is below a threshold then the entire block is made 0 else its made 1. This method removes

the noise. Removing all noise is very important for the box detection method that I am going to mention now.

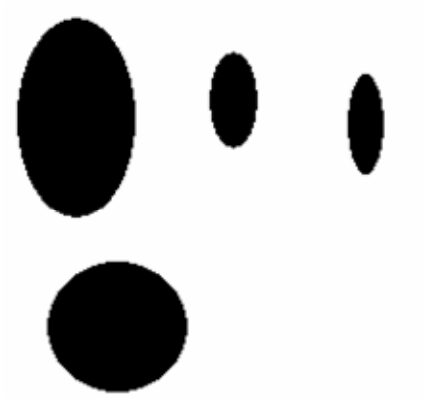
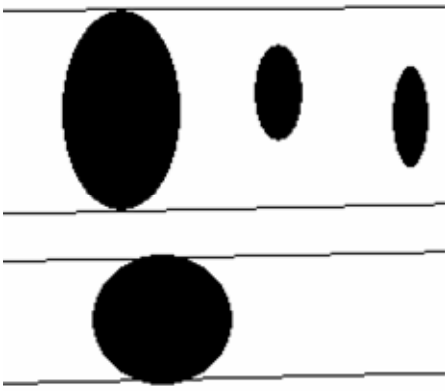


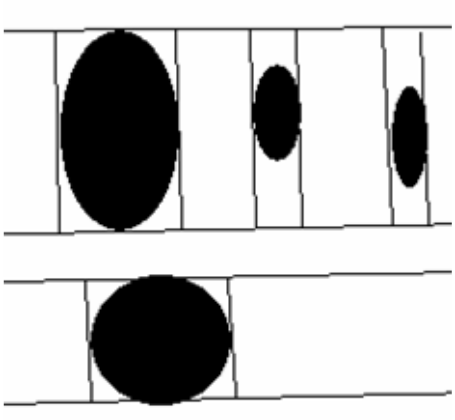
*Image to illustrate the boxing concept

Now we take sum of rows starting from the top row. If the sum changes from 0 to some other value or some value to 0 we know that this is a line of the box, and store these values at which the change occurs. We know that these values correspond to the top and bottom edges of the boxes. What we don't know is the number of boxes that are between these lines.



*After the row sum mentioned above

Now we find the sum of vertical colums just in between these lines. If the sum again changes from 0 to some other value, or vica versa, we know that these are the vertical lines of the boxes that contain our foreground objects.



*After the column sum

Now in between these vertical lines we again search for the horizontal row changes, by summing up the rows between these column numbers. This way we find the box that contains the foreground object

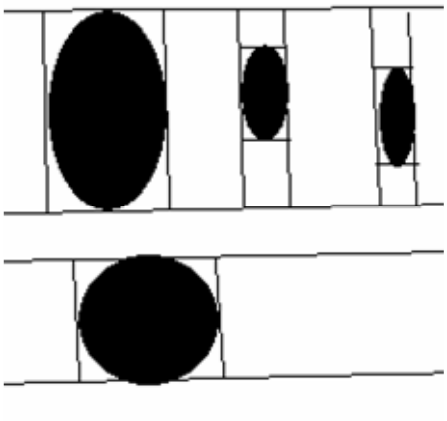


*Image showing all the objects found

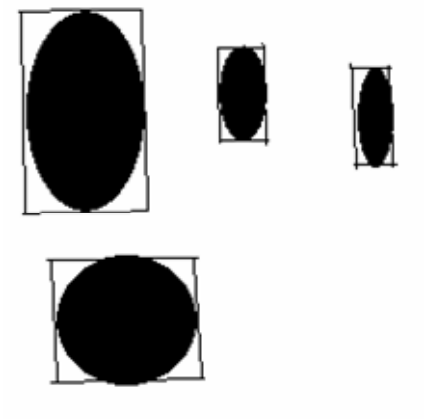


*Image showing the final boxes around all foreground objects

The boxing technique works really fast as it operates on many pixels at one time. To begin with it operates on (sums) one whole row and then a large part of a column and then a part of a row. This speed allows us to work in real time.

4. Tracking

Tracking is an essential part of the project. You have to ensure that you enter a person's height in the database only once. That is while the person is entering the store you don't count him twice. So you have to track the person until he crosses the camera.

Tracking a person seemed to be a hard problem when many people were walking together. So I just went on to solve the problem of one person walking a considerable distance in front of the next person, so that they both do not walk side by side ever.

We know that when a person moves towards the camera, he starts moving down the image. The following picture illustrates this concept:

The small dot is a person far from the camera, and the big oval is a person closer to the camera, as you can see the big oval is closer to the bottom of the image. This shows that as a person moves towards the camera he also moves closer to the bottom of the image. I used this information to set a threshold when I want to track the person for his height detection. I said that if the bottom edge of the box holding this person has crosses row 350 start tracking the person. I assume that no other person will cross this 350 thresh hold until this current person being tracked crosses the camera and hence is not present in the image anymore.I wait for the current person's bottom edge to equal the bottom most row of the image and then I stop tracking this person and wait for the next person to come in the image.

This high thresh hold was set because the results of boxing improved considerable after the persons bottom edge had crossed 350. Below this number at times there would be several boxes covering the person and will give us bad results as I will mention later.

4. Height Detection

Height detection is the basic goal of the project and now we mention how this is done. Our problem is that our system has to work in all the stores, so we cannot rely on calculating the height based on the geometry of the camera as the camera could be placed at different heights and angles. Calibrating the camera could be a tedious job, and a clothing store man might not be able to do it.

So to solve this problem, what we do is, we make our primary subject walk across the camera to begin with, and then we feed the system with this person's height. The program then calculates all the relations based on the height of this person. Now the system is ready for any person to walk through and it will be able to detect his/her height with a high level of accuracy. The following paragraphs explain the details and mathematics behind how it is done.

When we track the first person, we get a set of points (x,y) where x is the height of the box holding the person's image and y is the row number that corresponds to the bottom edge of the box. If we plot these points for a single subject we see that we get almost a linear line showing that the relationship between the x and y is linear. To find the exact linear relationship we find a best fit line for all these points as shown below:

If the line is y=ax+b

Then

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum_i x_i^2 & \sum_i x_i \\ \sum_i x_i & \sum_i 1 \end{bmatrix}^{-1} \begin{bmatrix} \sum_i x_i y_i \\ \sum_i y_i \end{bmatrix}$$

This way we form a linear relation between the height of the box and the row in which bottom edge of the box lies. Please note that we even know the height of our first subject. Therefore now we know that a certain box height with its bottom edge at a certain row corresponds to a certain real height. We know that as the bottom edge of the box moves down the image bigger box heights correspond to smaller real heights.

Now to calculate the height of the next person who walks in we do the following. When the next person walks in, we start tracking him. In each frame

we get the box height of the box that holds this person (hnew) inside it and we also get the row corresponding to the bottom edge of this box (bottomnew). Now we use this bottomnew to calculate the height of box of the first person had he been at this bottomnew using the equation that we derived earlier. This box height, call it old box height can be derived as follows:

oldboxheight=(bottomnew-b)/a

We also know our first subject's height (heightold), now the height of this person (heightnew) can be calculated as follows:

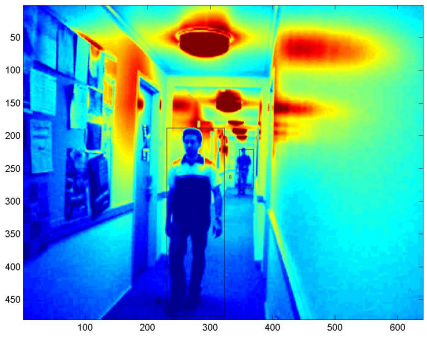
heightnew/heightold=hnew/oldboxheight
heightnew=(hnew*heightold)/oldboxheight

To find the final height of this person, we take the average of all the frames in which his height was detected when he was being tracked.

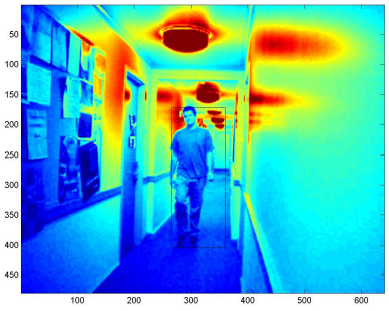
5. Results

The results have been very good. The height of people has been detected with an error of less than an inch. The following pictures show a few results.

In the following picture the subject's height is 5 feet 3 imches, and he is used to calculate all the relations.



In the next picture the subject's height is 6 feet 3 inches.



The subject's height is detected to be 6 feet 2 and a half inches, with an error of half an inch.

The system was tried on many other locations to make sure that it works in all environments and the results were always good with error less than 1 inch.

The system takes a frame a second, and can calculate the height based on that. A frame a second can be supplied by a really cheap camera, so the system meets that constraint.

Also no calibration is required by the store, as the system calibrates itself based on the height of the first person. The store could just have one of their employees walk in whose height is known at the starting of the day.

The system works in real time also.

6. References

[1] "Real Time Tracking of Multiple People Using Continuous Detection" by David Beymer

Vidit Nagory is a sophomore in Computer Science at Carnegie Mellon University

Computer Vision Based Sign Language Recognition for Numbers

Kenny Teng, Jeremy Ng, Shirlene Lim
kth@ece.cmu.edu, jwng@andrew.cmu.edu, shirlene@cmu.edu

Abstract

Sign language was created to enable communication between people with hearing and speech disabilities. However, this created a communication barrier between the people who uses sign language as their ONLY means of communication and others who doesn't know sign language. The main cause of this communication breakdown is the different grammar and rules of sign language that is not intuitive to those who lack knowledge in sign language.

In this paper, we propose a system that automatically recognizes and classifies an image of a hand signing a number. The unique grammar and rules of sign language makes number representation in sign language different than what is normally used by people without knowledge of sign language.

Two methods for recognition and classification of the number representation in American Sign Language (ASL) is presented – boundary contour & medial axis (skeleton & thinning).

1. Introduction

Sign language is one form of communication for the hearing and speech impaired. Similar to spoken language, there is no universal sign language. Sign language is itself a separate language with its own grammar and rules. Some signs are expressed as static gestures while others incorporate some dynamic hand movements. For static gestures, the prominent sign is captured within a specific time frame. For dynamic gestures, a sequence of finger and hand positions needs to be identified and analyzed in order to be recognized.

The focus of our project is on static gestures with a single hand. We strive to detect a hand signing the sign language representation of the numbers from 0 to 9.

Section 2 explains the initial challenge of locating the hand in the image. Section 3 explains how we determine the general orientation of the hand. Section 4 gives a brief overview of the two recognition methods that we implemented. Detailed explanation of the two methods used will be presented in Sections 5, 6, 7and 8.

2. Locating the hand

There were many ways to solve the problem of locating the hand in the images. The hand detection method that we implemented was fast and simple.

The hand detection method that we implemented was fast and simple. Initially, a video stream of the static background is obtained and a mean background is computed from the different frames. This background template is saved and is going to be used every time with every single incoming frame from the video stream to obtain the foreground. An arbitrary threshold is then used on the resulting foreground image to eliminate noise on the picture, and at the same time creating a binary image of the foreground.

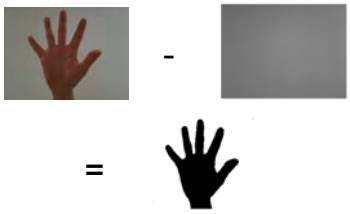


Figure 1: Foreground extraction by subtracting background from image

We used two hand location methods. For the boundary contour implementation, we scan the image from top to bottom (starting first from left to right). The first black pixel that we encounter is set as the "left" side of the hand. We then scan the image from

top to bottom again, this time from right to left. The first black pixel that we encounter is then set as the "right" side of the hand. Next, we do a horizontal scan from left to right starting from the top of the image, within the vertical boundaries previously defined. The first black pixel that we encounter is then set as the "top" of the hand. We assume the hand extends from the bottommost part of the image, thus there's no cropping of the image to locate the "bottom" part of the hand.

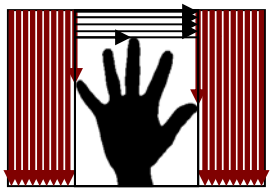


Figure 2:
Locating the hand from the binary image.

For both our medial axis implementation – skeleton and thinning, we start with a horizontal scan from bottom to top. First black pixels that are encountered are then stored into an array. Thus, we would have an array with the left most black pixel of the hand (from bottom to top) in the array. The same is applied from right to left starting from bottom to top. Once done, the minimum of the array storing the "leftmost" pixels of the hand is set as the "left" side of the hand. The maximum of the "rightmost" pixels is set as the "right" side of the hand. The "topmost" pixel in both the arrays is then set as the "top" of the hand.

3. Orientation Detection

Another issue we encountered, particularly for our boundary contour algorithm, was the orientation of the hand. Different orientations of the hands may distort the results obtained and may distort the classification of that method. Thus, we implemented a self-correcting orientation algorithm.

We start by a rough approximate of the center of mass (COM), using a simple heuristic based on the geometric shape of the cropped hand. Then, three scan lines are shot from the left and another three from the right. The two pairs of pixels that are located on the left and right edge of the hand are recorded. From there, the position vectors of those points are computed and the general direction vector of each set of three is determined (please refer to figure 3). We then average the orientation vectors to obtain one final orientation vector. Then, the cut-off line is set as the orthogonal vector of the orientation vector, and passing through the COM. The COM is then recomputed and adjusted to be the middle point of the cut-off line. From there, we can determine the starting and the ending points (left most black pixel intersecting the cut-off line and the rightmost black pixel intersecting the cut-off line) for our boundary contour generation.

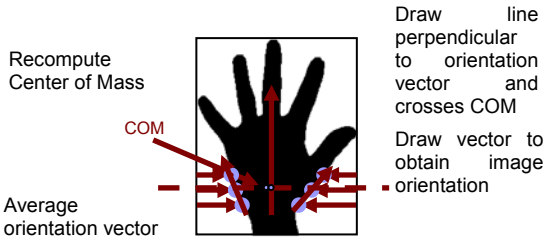


Figure 3: Determining the orientation of the hand using vectors

4. Method Overview

Different techniques have been used to analyze and classify the hand gestures, namely boundary contour, skeletonization and thinning.

4.1 Boundary Contour

Boundary contour (as we call it) is the process of determining the Euclidean distance of any point on the edge of an image to the center of mass (COM). We used this method to differentiate fingers and non-fingers as fingers have a distinctive length which will enable us to easily determine whether a certain point is a finger, thumb or neither.

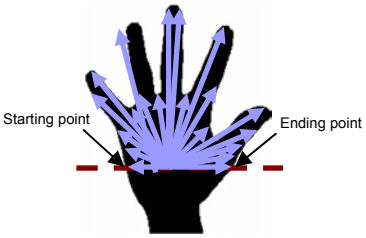


Figure 4: Determine the distance from edges to COM

4.2 Skeletonization

Skeletonization is the process for reducing foreground regions in a binary image to a skeletal remnant that largely preserves the extent and connectivity of the original region while throwing away most of the original foreground pixels.

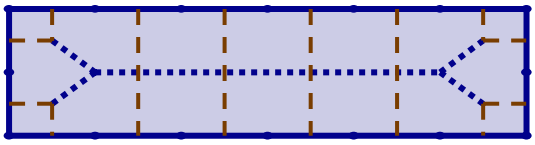


Figure 5: The skeleton is a line of points that are no nearer to one point than another

To see how this works, imagine that the foreground regions in the input binary image are made of some uniform slow-burning material. Light fires simultaneously at all points along the boundary of this region and watch the fire move into the interior. At points where the fire traveling from two different boundaries meets itself, the fire will extinguish itself and the points at which this happens form the so called `quench line'. This line is the skeleton. Under this definition it is clear that thinning also produces a sort of skeleton.

Figure 6: Equidistance skeletons of basic shapes

4.3 Thinning

Thinning is a morphological operation that is used to remove selected foreground pixels from binary images. It can be used for several applications, but is particularly useful for skeletonization. It is commonly used to tidy up the output of edge detectors by reducing all lines to single pixel thickness. Thinning is normally only applied to binary images, and produces another binary image as output.

The behavior of thinning is determined by the structuring elements used for the specific points being “thinned”.

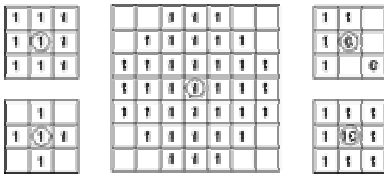


Figure 7:
Structuring elements used for thinning

5. Boundary Contour

The cut-off points obtained from the orientation determination process are used as start point and end point of an edge detection heuristic. We cycle the points along the edge of the binary image, while saving them in that sequence, and at the same time computing the Euclidean distance between that point and the COM.

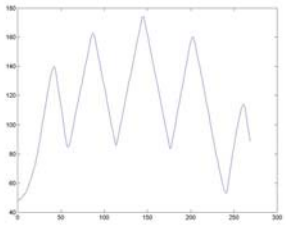
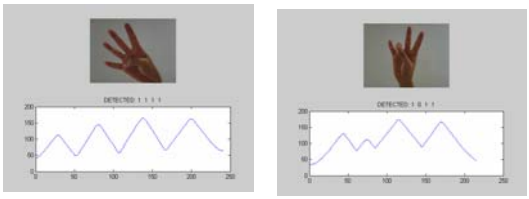


Figure 9:
The distance from each pixel to the COM is then plotted.

The peaks (maximum) are the furthest point from the COM, that is, they represent the positions of the tip of the fingers.

5.1. Outputs from Boundary Contour



Here are some outputs from the boundary contour algorithm, showing an inclined four, and a seven.
Next steps would be the classification of those outputs.

5.2. Classification for Boundary Contour

From the graphs, we determine the height of the maximum peak. The minimum difference between the maximum and the closest minimum is computed. If that value is above 20% of the highest finger, it is encoded as ‘1’ whilst peaks below that threshold are classified and encoded as ‘0’.

For example, [1 1 1 1 1] will be categorized as five fingertips, thus representing the number ‘5’.

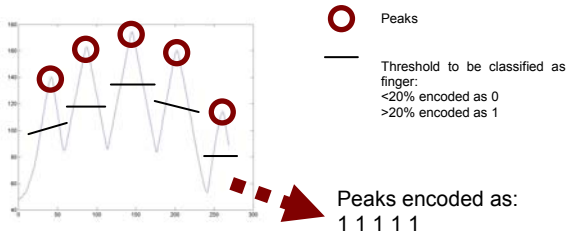


Figure 10: Peaks above threshold encoded as '1' and peaks below threshold encoded as '0'

5.3. Results from Boundary Contour

For thinning, we ran tests for 5 sets of hands of different color, orientation and size. A video clip of a hand signing an average of 10 numbers over a period of time was also used. Correct matching for thinning were dramatically better than skeleton but performed a little under if compared to boundary contour. Skeleton had a matching rate of ~**80%**.

Boundary contour has a fairly high accuracy of classifying the numbers is because the shape of the hand for the different numbers is really well defined with sharp changes. That is why, when the hand is orientated different in the z-plane, we lose some of those sharp changes and boundary contour starts failing.

6. Equidistance Skeleton

There are two ways of skeletonization the hand structure. We used both the equidistance skeleton and the thinning method for our project.

To implement the equidistance skeleton method, we used MATLAB's morphing function. MATLAB's morphing function requires us to invert the colors of the binary image. Morphing the binary image of the hand, we obtain the skeletal representation of the hand.

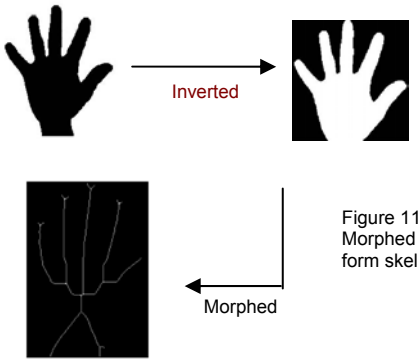


Figure 11:
Morphed image to form skeleton

To implement the thinning method, we also used MATLAB's morphing function.

6.1. Output from Equidistance Skeleton

Those are outputs for skeletonization showing number four and eight respectively.

6.2. Classification for Equidistance Skeleton

For the equidistance skeleton implementation, we needed to trace through the skeleton and store the values in a data structure in order to identify/classify the hand. To move along the skeleton, we use a window to determine the next valid pixel to move to.

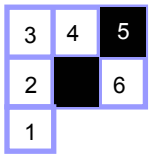
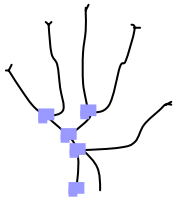


Figure 12:
The window used to determine the next "valid" neighbor.

Figure 13:
The window is then placed at the starting point (bottom most) and scans through the skeleton. Each branch point is recorded and marked.



We classify each pixel as we move along the skeleton using identifiers such as "endpoints", "branch" and "normal points". "Endpoints" is classified as the pixel at which there are no other valid neighboring pixels. "Branches" are classified as the pixel at which it has more than one valid neighboring pixel. This pixel is then marked so that after reaching an endpoint of one of the branching pathways, our window tracking would return to the branch point and move along the next valid pathway from that particular pixel.

Values of importance are the coordinates of the pixels, the distance from starting point, the distance from the nearest branch point and branch point coordinates. Storing values inside arrays, we then

analyze the arrays by comparing the distance of each pathway we traced.

Using the maximum distance as reference, distances that are ¼ of the maximum distance stored will be categorized as a thumb, thus encoded as a '1'. Distances that are more than ½ of the maximum distance is categorized as a finger, thus encoded as a '2'. Other invalid distances are then classified as a '0'. Based on the encoding, we determine the number the hand represents. For example, [2 2 2 2 1] will be classified as detecting the number '5'.

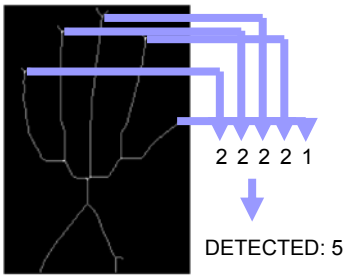


Figure 14: Endpoints a certain distance away from starting point is encoded as '2', '1' and '0'.

6.3. Results for Equidistance Skeleton

We ran our program with 5 sets of hands of different color, orientation and size. We also ran a video clip of a hand signing an average of 10 numbers over a period of time with our program. Correct matching between the original number and the detected number was less than 50%. It was an approximate average of ~**40%** correctness.

We attribute the low performance of this algorithm to the many extra branching paths in the skeleton. In the future, we believe the performance of this algorithm can be improved by applying an extra skeleton "cleaning" step to remove all the small branching that were extraneous. Due to time constraints, we did not have time to implement this extra feature into the program to enhance its performance.

7. Thinning

To implement thinning, first, translate the origin of the structuring element (middle) to each possible pixel position in the image. If foreground and background pixels in the structuring element exactly match foreground and background pixels in the image, the image pixel underneath the origin of the structuring element is set to background. Otherwise it is left unchanged.

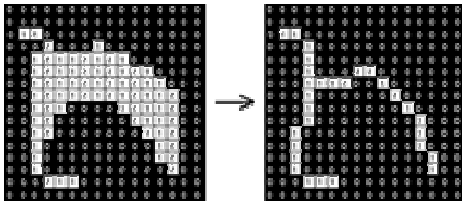


Figure 8: Thinning also produces a skeleton

7.1. Outputs for Thinning

Those are outputs for the thinning algorithm showing number four and eight respectively.

7.2. Classification for Thinning

For the thinning implementation, the classification stage is almost similar to that of the equidistance skeleton. The same process of tracing through the skeleton is also needed. However the algorithm is less complex since we do not need to take care of split ends as perceived from the skeleton image.

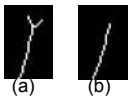


Figure 15: (a) Split end at the endpoint of the branch of a skeleton, (b) endpoint of a branch of from thinning

Basically, we calculate the length of each branch, getting rid of insignificant branches which have length shorter than a given threshold. Based on the longest branch, we calculate how many short branches and how many long branches. Long branches would represent stretched fingers while short branches represent folded fingers. I would then code them in terms of binary values as 1 for stretched fingers and 0 for folded fingers.

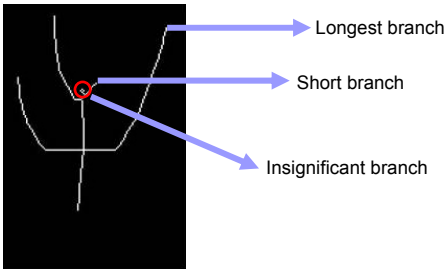


Figure 16: An example of how the branches are being classified. This particular image represents the number eight and is coded as [1101].

Based on the binary values, we would be able to classify what number does the hand represents. There are some special cases which need to be taken care of. For an example, a short branch could exist at the right most part of the image. This branch could represent a thumb and thus should be coded as 1 instead of a 0.

7.3. Results for Thinning

For thinning, we ran tests for 5 sets of hands of different color, orientation and size. A video clip of a hand signing an average of 10 numbers over a period of time was also used. Correct matching for thinning were dramatically better than skeleton but performed a little under if compared to boundary contour. Skeleton had a matching rate of ~**70%**.

Although devoid of branching problems like equidistance skeleton did, it still had problems due to the "front" and "back" tilted hand positions. The result of forward and backward tilted hand would distort the binary shape of the hand, thus producing a shorter than normal length, which sometimes is wrong interpreted by the program.

Another problem is the lack of branching. For folded fingers, the lack of branching made the program fail to recognize the folded fingers among the outstretched fingers thus wrongly interpreting them. Unlike equidistance skeleton where slight bumps and extensions of the hand would produce branching, thinning sometimes failed to detect the small curves and bumps of folded fingers.

8. Conclusion & Discussion

Once we were able to classify the data obtained from images, we moved onto parsing a streaming video. We weren't able to classify the signs in real-time because the morphing, calculation and classifications took time to complete for each frame we captured.

We parsed a video clip of a hand signing the different numbers and for each frame we would test it against our program. Each frame and result is then stored and compiled into an AVI file. Once completed, the AVI file was converted to MPEG format. Altogether we have AVI files containing the results of the video clip against our 3 algorithms.

Among the three algorithms, boundary contour was the most reliable and stable. The equidistance skeleton performed worst among the three algorithms. We attribute the performance level of the equidistance skeleton to the fact that the skeletons had a lot of "noise". There was a lot of branching which were confused as a key branch point representing branching of fingers. Thus, results weren't as stable and reliable. Thinning performed reasonably well due to the fact that it had more "test" cases in the program. It was more carefully tested against more possible cases and classified accordingly. It also did not have the problem of "noise" in the form of "branching" as that of the equidistance skeleton method.

Overall it was an interesting project and all three of us certainly enjoyed ourselves exploring ways of implementing the algorithms and classifying the data we obtained. Our project of classifying a hand signing the sign language representation of numbers using the three algorithms: Boundary contour, Equidistance skeleton and Thinning was successfully implemented.

9. Acknowledgements

We'd like to extend our thanks and appreciation for all the help we received during the course of the semester. Particularly, we'd like to thank Prof. Lee and for providing good feedbacks to us and helped us along the way. We'd also like to thank Ellen Lai for contributing images and video clips of her hand to aid us in diversifying our database.

10. References

[1] Koichi Ogawara, Soshi Iba, Tomikazu Tanuki, Yoshihiro Sato, Akira Saegusa, Hiroshi Kimura and Katsushi Ikeuchi, "*Recognition of Human Behavior using Stereo Vision and Data Gloves*", 3rd Institute of Industrial Science (Univ. of Tokyo), Robotics Institute (Carnegie

Mellon University), Research Division (Komatsu Ltd.) and Univ. of Electro-Communications.

[2] Nobuhiko Tanibata, Nobutaka Shimada, Yoshiaki Shirai, "*Extraction of Hand Features for Recognition of Sign Language Words*", Osaka University.

[3] James MacLean, Rainer Herpers, Caroline Pantofaru, Laura Wood, Konstantinos Derpanis, Doug Topalovic, John Tsotsos, "*Fast Hand Gesture Recognition for Rea-Time teleconferencing Applications*", University of Toronto, University of Applied Science.

[4] "*Morphology – Skeletonization/Medial Axis Transform*", http://homepages.inf.ed.ac.uk/rbf/HIPR2/skeleton.htm

[5] "*The Gesture Recognition Homepage*", http://www.cybernet.com/~ccohen/

Kenny Teng is a Junior in Electrical and Computer Engineering at Carnegie Mellon University. He is from a tiny island in the middle of Indian Ocean, called Mauritius. Kenny is very interested in Computer Vision and Image Processing, and would like to pursue a career in that direction.

Jeremy Ng is a Junior in Electrical and Computer Engineering at Carnegie Mellon University. He is very interested in computer vision, image processing and computer graphics. He is an international student from Malaysia.

Shirlene Lim is a junior in Electrical & Computer Engineering, double majoring in Biomedical Engineering at Carnegie Mellon University. Shirlene is passionate about technology that assists and improves people's lives. She is originally from Kuala Terengganu, Malaysia.

Creating Computer Generated Caricatures

Dylan Goings
Physics
Carnegie Mellon University

Dylan@cmu.edu

Jean Sun
Computer Science
Carnegie Mellon University

Jsun@andrew.cmu.edu

Abstract

We present a program using a training set of frontal face images to create a caricature of a new face by exaggerating the differences in spatial relations of the new face to the training set (the "average face"). This was accomplished using Principal Component Analysis (PCA) and a morphing algorithm based on the orientation of line "landmarks." Our program was successful in warping an image according to non-average features, but did not create a good caricature in the well-known sense of the word.

1. Introduction

A caricature is commonly thought of as a severely exaggerated, "cartoonish" portrayal of a human face. The exaggerated features are those that differ from our perception of the average features of a human face, specifically the spatial dimensions and orientations of the main features like eyes, nose, mouth, forehead, cheeks, teeth, ears, etc., and they're relative relations to each other.

Our goal was to use image analysis techniques to be able to take a picture of a human face and create a caricature of that face. In accomplishing this objective we feel that we would somewhat be teaching computers to create art, much as caricatures drawn by humans are often used in artistic applications such as comic strips and for sentimental value at events such as country fairs. The method we chose to accomplish this task was through the use of a face database and Principal Component Analysis (or PCA).

In order to exaggerate facial features that differ from the norm or average features, we have to have a definition of the average. To create a description of the average human face, we used a database of face images gathered from several different sources. For our later PCA analysis, it was important to have all of the faces be fully frontal (i.e. the person is staring straight at the camera). We then decided on a set of points, or landmarks, representing key characteristics of a human face, such as the location of the eyes, lips, jaw line, etc.

PCA is an analysis technique for computing the variances, or correlations, between samples in a data set. For example, in a sample of people comparing height to weight, the data is likely to have a strong correlation between these two variables. This correlation is given by the Covariance matrix of the data, defined as:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}(y_i - m_y)(y_i - m_y)^T \qquad (1)$$

where n is the number of samples, or data points, y_i is the i[th] sample, T means the transpose, and m_y is:

$$m_y = \frac{1}{n}\sum_{i=1}^{n} y_i \qquad (2)$$

The covariance matrix is symmetric and can be decomposed using Single Value Decomposition into a matrix whose columns are the eigenvectors of Σ, and a square diagonal matrix whose nonzero elements are the eigenvalues of Σ. These eigenvectors represent the direction of correlation, and the eigenvalues are representative of how correlated the variables are (larger eigenvalues means stronger correlation). Figure 1[1] shows an equivalent sample of data with the first principal eigenvector and its orthogonal axis projected onto the data.

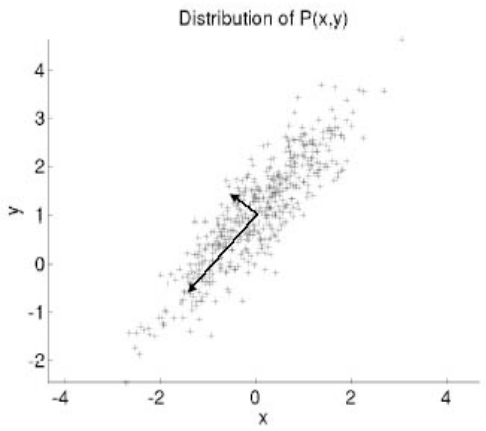


Figure 1

In the example case of the study of height vs. weight the eigenvector "directions" don't actually have any physical meaning, since the data are just abstract values for a physical measurement. But in the case of a face image, by considering the locations of specific features, such as the outline of the eyes, the variations in the data computed with PCA yield eigenvectors whose directions actually have physical meaning – they point along the face. Because of this, we can use the eigenvectors to exaggerate the facial features by morphing the image along those directions.

After generating exaggerated feature components, we used a Feature-Based Image Morphing technique[2] to graphically alter the original face picture to the "caricature."

Next we will explain our method for exaggerating the feature components and go more into depth on the image morphing technique. Then we will explain our results and discuss problems and future improvements that could be made on our technique.

2. Methods

2.1 Creating an average face

First, we defined a set of 99 landmark points to describe the important facial features. These points outlined the eyes, eyebrows, nose, lips, jaw line, hairline, and the top of the hair. Next, we collected a set of fully frontal face images from several image databases[1], and hand marked all of the landmarks on each image. The hand-marking ensured that all of the landmarks were placed very consistently across the face images to produce a better average.

[1]MIT CBCL face set, Carnegie Mellon Computer Vision database, NBA.com player's database, Harvard face database

To do a PCA of the faces, they first had to be aligned and scaled relative to each other, since the images were compiled from several sources and included different sized pictures, and faces that were not all oriented in the same fashion. To align the faces we employed PCA again, by computing the covariance of just the landmarks describing the eyes. The primary eigenvector from this computation points along the axis of the eyes, and its magnitude gives us a scaling factor to use comparatively with those of the other faces.

After aligning the N number of faces, we have a set of N 99x2 matrices, where each row represents the x and y position of one landmark in the face. We then average all of these face matrices together to create an average, or mean face. Figure 2 shows a plot of the mean face landmarks with connecting lines (contours) between the points which define the different features listed above.

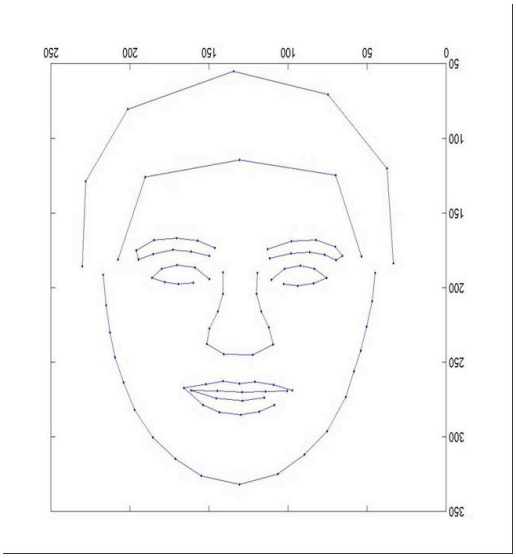


Figure 2

Along with the mean face, we also use the covariance of the faces to describe the average human face. Using the technical computing program MatLab we can easily compute the covariance matrix given by equation (1). We first reshape the 99x2 face matrices into 198x1 vectors, treating each value as a "variable," and each face vector as a "sample," much as the height and weight measurements are both variables sampled from a person. Figure 3 shows an annotated version of the matrix for which we'll compute the covariance.

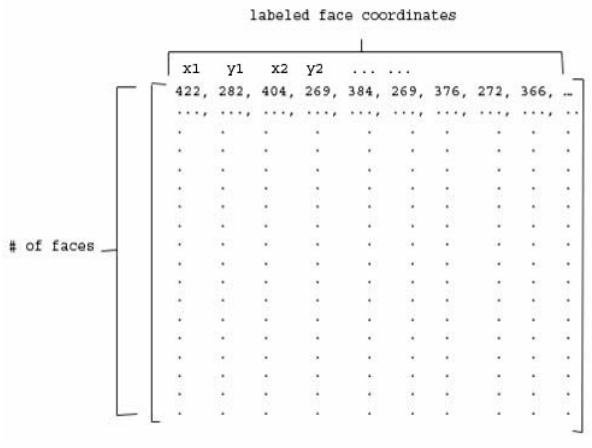


Figure 3

Equation (1) results in an eigenvector matrix with the number of columns (eigenvectors) equal to the number of variables in the data, so in the case of our face vectors, 198 vectors. However, mathematically it can be shown that most of the variation data can be explained with only the first few eigenvectors, whose weight fall off exponentially. Figure 4[3] shows a plot of the relative weight of the first 30 eigenvectors for an average data set, with an inset of a log plot for the same data. It is easy to see that most of the data is contained within the first 10 eigenvectors, so to save calculation time in our program, we only compute and use the first 20 eigenvectors from our PCA analysis of the faces (although we allow the user to use specify the number to optionally compute more).

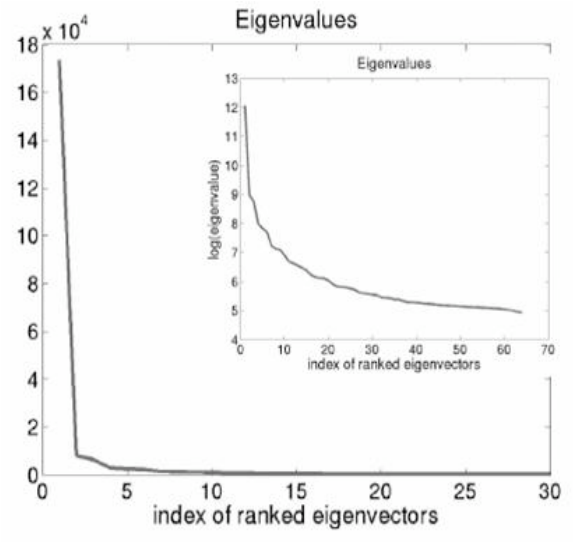


Figure 4

2.2 Computing variances for a new face

After calculating the mean face and eigenvectors for the covariances of the input training face set, we can use these values to measure how a new face varies from the average. The initial step is to label the new face the same as the training set and to align the points. Figure 5 shows a picture of talk show host Jay Leno, with 99 landmarks labeled on his face (as blue x marks).

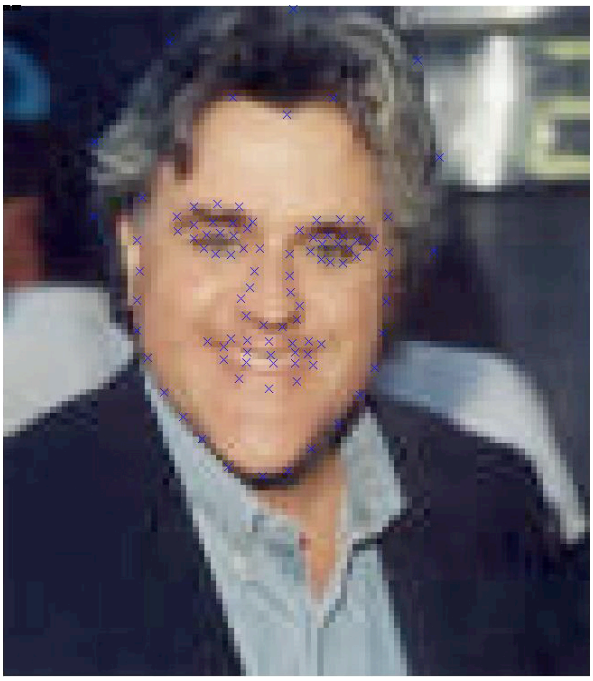


Figure 3

After we have a new face (reshaped to a 198x1 vector), we compute the inner product of the new face vector with the eigenvector matrix, essentially projecting the face vector onto each one of the eigenvectors. This produces a vector whose components are representative of the new face's values with respect to those of the average components of the training set. So, by subtracting the mean face vector from the vector of new face values, we get the differences between each feature on the new and average faces. To compose an exaggerated caricature of the input face, we decided to create a couple of different results by multiplying all or part of this "variance" vector by a scalar to see the different magnitudes of exaggeration they create, and also to better understand how these constant parameters work and affect the final image results.

2.2 Morphing

The morphing algorithm we used to generate the warped image from an exaggerated face vector works on the principal of inverse mapping. The algorithm goes through every pixel of the new, or destination image, and interpolates the pixel in the source image to use in the destination pixel. This assures that every pixel in the destination gets painted.

The interpolation algorithm works by considering matching lines, or contours, in the source and destination images. For each pixel, its distance to each contour is computed, and a weight assigned to this distance, which is given by:

$$w_i = \left(\frac{length^{\rho}}{a + dist} \right)^{b} \quad (3)$$

where a, b, and _ are constants, dist is the shortest distance from the pixel to the line, and length is the length of the line. Currently, our program uses values of a = 1, b = 1, _ = 0.5.

Finally, these values are used to compute the location of the source pixel by mapping each contour in the destination image to a contour in the source image (which may be translated, rotated, and scaled from that in the destination).

In the case of our face vectors, each set of points within one feature (for example, one eye) creates a line with its adjacent landmarks, as shown in Figure 2, and each of these lines is weighted for each pixel in the morphing program.

3. Results and analysis

Looking at the resulting images we received from scaling the face variance vector, we subjectively decided that scaling all of the vectors by a factor of 4 produces the most satisfactory result. By that we mean, it provides a large enough magnitude of exaggeration so that the resulting image resembles a caricature, but at the same time retaining the original face enough so that it is not too distorted and still easily recognizable. Figure 6 is the original input image. Figure 7 is the resulting caricature output with a scalar factor of 4 applied to the variance vector.

Figure 6

Figure 7

The caricature image of Figure 7 displays some positive results in terms of creating some of the exaggerations that would be expected in a caricature of Jay Leno. For example, his chin and cheeks appear to be enlarged and extended from the original, as has his forehead, while his nose appears to have been made thinner and pointier.

There are some clear problems with this output, however. While some of Jay Leno's features have been exaggerated, they are not stretched to the point of an actual caricature, such as shown in Figure 8. With greater values of the scalar multiplier, however, the image begins to show signs of warping not associated with caricature. This is most likely from features which are very close to the average being exaggerated as well, due to the magnitude of the scalar. One way to address this issue in future work would be a relative scaling of each component in the variance vector.

Instead of a constant scaling factor, a linear or even exponential scaling factor could be used, to maximize the exaggeration of varying features while minimizing that of features close to the average.

Figure 8

Another issue of concern with the caricature image are warping artifacts, the most visible of which is an incongruous line from Leno's right eyebrow to below his ear. It is also clear that the top of his head has been cut-off from the image, unfortunately not allowing us to view any exaggerations to his hair.

These problems are most likely caused by the morphing program, as an examination of our covariance matrix and variance vectors did not reveal any problems such as stray landmarks in our morphed images (including others besides the Jay leno photo). These issues could hopefully be addressed by adjusting the parameters of the morphing program, as well as some of its functioning. For instance, we specifically wrote the program to produce an image the same size as the original for better comparison. But this creates cut-off as the image is stretched, and it would be a simple matter for a future program to not be constrained in this manner. Another issue with the morphing program is its runtime, which is on the order of $O(n^3)$ for an image of size nxn. This means a lot of processing time for even small images, such as the Leno picture which is 99x112 pixels. A more efficient morphing program would allow more thorough testing of program parameters to produce the best image caricature.

Other improvements could be made to the landmark set we defined for our faces, which does not include features such as the ears and facial hair. Including more facial features, and using more points to create a more detailed face should provide better results from the training set and more ways to stretch an image from the average.

4. Conclusions

We determined that our method of producing a computerized caricature is a solid beginning framework for this project. By describing an average human face by the mean of a training set and the covariance matrix eigenvectors of that set, and using this average face to compare and exaggerate new faces by scaling their variance vectors, we produced rudimentary caricatures of the new faces. There is room for improvement in the way the exaggeration is calculated, and on the graphical end for morphing the image, but these enhancements can be built upon our method's foundation.

Over the course of this project, we ran into several difficult choices for how to proceed, such as how to landmark the faces, how to align them, and how to exaggerate the variance. We also encountered several difficulties in producing results, mainly with how to accurately define and compute the variance vector, and how to morph the image. This taught us a great deal about Principal Component Analysis and what the covariance matrix and its corresponding eigenvectors and values can tell you about a data set. We also learned a little bit of graphical programming in how to warp an image based on a set of landmark points.

We found that time was a big factor in our research. The individual marking of each face in the training set, and for new faces to be caricatured, was very time consuming and warranted a lot of patience. For future work in this area, a method for accurately marking the landmarks of faces automatically would greatly increase the performance of our method, because a larger training set will produce a better representation of the average face.

The morphing program was another time bottleneck, and an improved morphing algorithm would strengthen our results by allowing us to try more methods for computing variances to observe which methods of exaggeration work the best. An interesting problem we encountered, after we'd determined how to calculate the variance vector, was just how to modify that vector to produce a caricature, which from our experience seems to be a deeper task than one might originally suppose, something connected to our own human abilities of sight and interpretation of images and art, and this provides the

most interesting avenue for future research in this area: how to better explain what we view as "characteristic," not just in human faces but in all things we observe.

5. References

[1] Lee, Tai Sing, *Principal Component Analysis Lecture*, Carnegie Mellon University, Pittsburgh, PA, Spring 2004, p. 9

[2] Beier, Thaddeus, Neely, S, *Feature-Based Image Metamorphosis*, http://www.hammerhead.com/thad/morph.html, Siggraph, 1992

[3] Lee, Tai Sing, *Principal Component Analysis Lecture*, Carnegie Mellon University, Pittsburgh, PA, Spring 2004, p. 19.

Authors:

Dylan Goings is in his third year of study for a Bachelor of Science in Physics at Carnegie Mellon University, minoring in Creative Writing with a focus on Poetry. He hails from Ann Arbor, MI., a hometown he holds dear to his heart. His favorite color is blue. Right now, he is probably smiling.

Jean Sun was born in Shanghai, China and grew up in Boston, MA. She is currently a junior majoring in Computer Science and Business Administration.

Real-time Soccer Ball Detection

Daniel Kim
danielki@andrew.cmu.edu
15-385: Computer Vision
Professor Tai Sing Lee
Carnegie Mellon University
School of Computer Science

Kevin Caffrey
kcaffrey@cmu.edu
15-385: Computer Vision
Professor Tai Sing Lee
Carnegie Mellon University
School of Computer Science

Abstract

This paper details an attempt at real-time soccer ball detection in color images using AdaBoost, a powerful boosting algorithm which combines several weak learners to give results with higher accuracy than any of the individual classifiers. Many features were investigated as potential classifiers for AdaBoost, such as color histogram, power spectrum, circle detector, and Gabor wavelets, but in the end, a collection of 15 Haar-like filters was chosen. The final results were 100% accuracy at detecting soccer balls, with a 1.3% false positive rate, when run on the training set. Efforts at real-time processing allowed achievement of an average time of 0.6 seconds to evaluate a 320x240 pixel color image, which is equivalent to 1.7 frames per second.

1. Introduction

The development of a robust, generalized object recognition algorithm is critical to the advancement of robotics and computer vision-related applications. Widely respected international organizations such as RoboCup are striving towards having a fully autonomous team of humanoid robots play soccer with the best human players by the mid-21st century.[1] In order to meet these goals, we must continue to improve and test the limits of computer vision algorithms.

AdaBoost provides a powerful technique for image classification. This algorithm requires several weak classifiers as inputs, though. Some potential classifiers include, but are not limited to, color histogram, power spectrum, circle detection, Gabor wavelets, and Haar filters.

[1] http://www.robocup.org

Color histograms take all the pixels in an image and place them into a defined number of bins, with each presumably covering the same span of pixel values. Color histograms are good detectors of images that have a distinct color signature, for example, a soccer ball. Other applications for color histograms are in image tracking.

The power spectrum is a plot of an image's power divided amongst bins for varying frequencies. Power spectrums are most commonly generated by using a Fourier transform, and taking the magnitudes of the complex coefficients and squaring them.

A circle detector would also be useful soccer ball detection. One possible implementation of a circle detector utilizes the Hough transform. This requires some method of edge detection. The more popular forms of edge detection are the Canny, Prewitt, Sobel, and Roberts edge detectors.

Gabor wavelets are a family of wavelets of particular orientation and scale. It is believed that images in the primary visual cortex are represented in terms of Gabor wavelets, which gives an indication as to their wide ability and range in application.

Haar filters are rectangular filters loosely based on the Haar wavelet bases. They are simple and quick to compute, and provide rough information on edges and borders within images.

2. Methods

2.1. Image collection

The majority of the images used in this investigation were taken around the campus of Carnegie Mellon University. A generic black-and-white soccer ball was used, and pictures were taken using a Fuijifilm Finepix 2800 digital camera, as well as a webcam. Both outdoor and indoor images were taken. Outdoor images were captured on a sunny day,

however, the lighting varied between individual pictures. Indoor images were taken in Mudge dormitory.

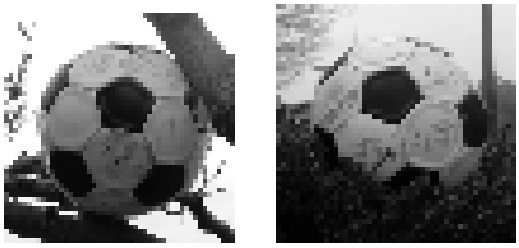


Figure 2.1.1: Sample soccer ball images taken from our training set

Soccer ball images were taken against a variety of backgrounds ranging from grass and flowers to concrete and sky. This was done so that our training set would be as robust as possible. Also, creating a homemade training set ensured that all the images would be of the same soccer ball, vastly improving our detection ability. Using a Matlab function we wrote, soccer ball images were cropped to be made square, smoothed, then subsampled down to a predetermined size of 50x50. Special care had to be taken to ensure the images outputted were exactly 50x50 and not 51x51 or 49x49, regardless of the size of the original image.

(a) (b) (c)

Figure 2.1.2: Shows images downsampled a) without border padding, b) without smoothing and c) with smoothing.

Non-soccer ball images were taken of natural scenery, and effort was made to match these to the backgrounds of the soccer ball images. To quickly and effectively increase our set of non-soccer ball images, random 50x50 sub-windows were taken from the large, high-resolution non-soccer ball images. All in all, 71 soccer ball and 978 non-soccer ball images were collected and used.

2.2. Classifier Learner - AdaBoost

AdaBoost is a method of boosting developed by Freund & Schapire in 1995. Boosting is a method of combining weak classifiers together to form a strong classifier. The concept was introduced by Kearns & Valiant in 1989 when they proved learners with only slightly better than random performance could be combined together to form a strong learner with high accuracy. AdaBoost has since been used in many rapid object detection systems. In particular, Viola & Jones used AdaBoost for face detection with promising results. They succeeded in developing a real-time system with approximately 95% accuracy and a negligible false positive rate. [1],[3]

A popular approach to using AdaBoost is to use a cascade method. A coarse-to-fine search is done through the image to reduce calculation time while retaining performance levels. Initial stages achieve very high accuracy while attaining a relatively low false positive rate, but still rejecting most of the windows from the search. Later stages utilize more weak learners to reduce the false positive rate. Viola & Jones used 38 stages and 6000 features to achieve 95% accuracy with 1 in 14084 false positives, at 15 frames per second. Their approach to choosing the number of classifiers per stage and number of stages involved setting accuracy requirements for each stage, and an overall accuracy goal to decide when to stop adding stages. [1]

AdaBoost is used to choose a small number of classifiers from a large set of classifiers. Each classifier is run on a large training set of training data to determine the errors of each classifier. AdaBoost chooses a classifier and confidence related to the weighted error at each step. After each classifier is chosen, the errors are re-weighted such that the examples that were classified correctly receive less weight for the next choice. This encourages weak learners that compliment each other. The confidence generated is related to how much each learner reduced the error. The confidences are used to build the strong classifier by multiplying each weak classifiers answer by the confidence. If the sum of these weighted classifications is greater than half the sum of the confidences then the final output is true. The algorithm is listed in Table 1.

Although AdaBoost offers promising results, there are several problems involved in the practical implementation. The first, and main problem for us, is that it requires a large training set to achieve good results. Our set of close to 1000 negative and 100 positive images yielded decent but weak results. The second problem is that it is highly dependant on the weak classifiers used. AdaBoost attempts to make each hypothesis independent by assuming that

individual hypotheses will have little correlation between the examples classified incorrectly. If there is a strong correlation between the results of classifiers then one classifier will end up having a much larger confidence than the rest. The problem with this is that the output of the final strong classifier is relying only on one weak classifier, thus boosting is ineffective. This same problem can arise if some classifiers are too strong or complex.

- Given example images (x_1,y_1), …, (x_n,y_n) where y_i = 0,1 for negative and positive examples respectively.
- Initialize weights $w_{1,i}$ = 1/2m, 1/2l for y_i = 0,1 respectively, where *m* and *l* are the number of negatives and positives respectively.
- For $t = 1,\ldots,T$:
  - Normalize the weights to sum to 1
  - For each feature *j*, train a classifier h_j which is restricted to a single feature. The error is calculated with respect to w_t, e_j = Sum over i $(w_i\ |h_j(x_i) - y_i|)$
  - Choose the classifier with the lower error e_t
  - Reduce the weights of examples classified correctly by a factor of B_t, where $B_t = e_t / (1 - e_t)$
- The final classifier outputs 1 if the sum of $\log(1/B_t)\ h_t(x)$ is at least one half of the sum of $\log(1/B_t)$. Here, $\log(1/B_t)$ is the confidence for each classifier *t*

Table 2.2.1: The AdaBoost algorithm for learning classification. Each round selects 1 feature from the set of 7000 features in our implementation. [1]

Another problem associated with the approach Viola & Jones took is the large time required to train using AdaBoost. Viola & Jones used 180,000 features, with 20,000 sample images. For 6,000 stages of AdaBoost, this required on the order of 10^{13} feature evaluations, which is extremely prohibitive for training. McCane & Novins estimates that this would require about a year to train. They present alternatives to feature evaluation using functions to minimize to find the best features. Their results are moderately successful, although they didn't achieve as high success as Viola & Jones, which makes their work questionable. McCane & Novins also propose a better way to choose the number of classifiers per stage is with a maximum computation time allowed per stage, instead of performance levels. [1],[2]

2.3. Classifier selection – Color Histogram

The first feature examined was the color histogram of the image. The belief was that, since soccer balls (and our test ball in particular) are predominately black and white, the color histogram would reflect this by having two sharp peaks at the ends. Non-soccer ball images were predicted to have a relatively uniform distribution of pixels.

The classifier for the color histogram would involve calculating the sum squared error of the color histogram from the average color histogram of a soccer ball image, and thresholding at some value.

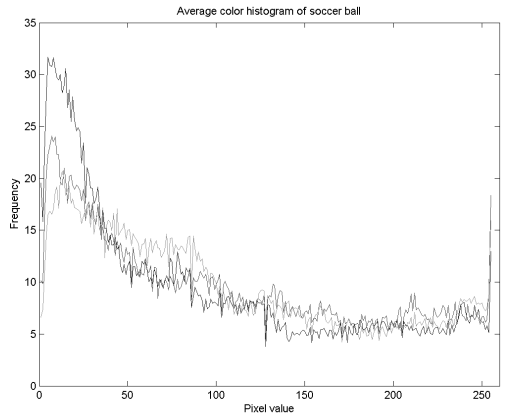


Figure 2.3.1: Average color histogram of a soccer ball

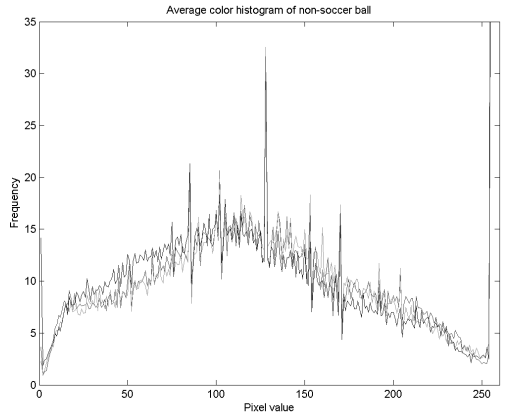


Figure 2.3.2: Average color histogram of non-soccer ball images.

2.4. Classifier selection – Power Spectrum

The next feature examined was the power spectrum of the image. From prior experience, power spectrum was known to be a decent texture discriminator,

particularly for our purposes in this project, due to the nature of a soccer ball. The texture of a soccer ball will be of a given frequency, and varies in a similar manner in all directions due to the symmetry a soccer ball has. Also, regardless of what orientation the ball is in, the power spectrum should come out to be the same.

The classifier for the power spectrum would behave in a similar manner as the color histogram.

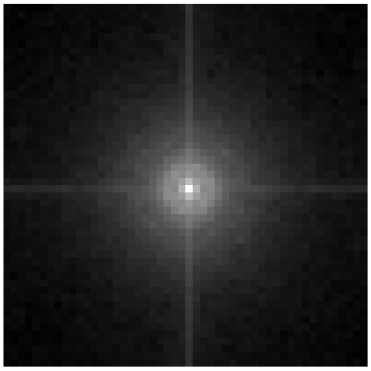


Figure 2.4.1: Average power spectrum of soccer ball

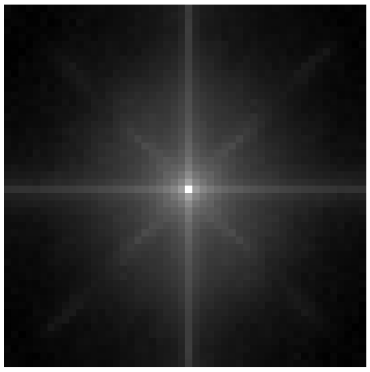


Figure 2.4.2: Average power spectrum of non-soccer ball images

2.5. Classifier selection – Circle Detector

Another feature detector examined was a circle detector. The major shortcomings of a circle detector would be with images where the soccer ball was at least partially occluded, destroying its circular nature, and with images of balls other than soccer balls. Of course, a circle detector would find all kinds of circles other than soccer balls, making it prone to having a high false positive rate, but its effectiveness at finding soccer balls if they exist make it worthwhile.

The circle detector would be implemented using edge detection followed by a Hough transform. Edge detection was done using a Sobel operator, thresholded at .3.

The classifier for the color histogram would involve looking at the result of the Hough transform and determining whether there was a circle or not.

2.6. Classifier selection – Gabor Wavelets

Gabor wavelets were also tried as features, but due to the lack of promising early results, were quickly abandoned. Two Gabor wavelets were used, and the average power of the wavelets and the image was calculated.

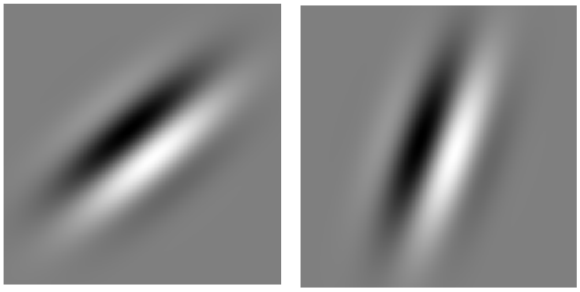


Figure 2.6: The two Gabor filters used: 40 degrees and 70 degrees.

2.7. Classifier Selection – Haar-like Filters

The problem with the previous classifiers tried is two-fold: the lack of gain from boosting, and the computation time involved. With only a few feature detectors, there are images that none of the classifiers got correct. In addition, they seemed to be too strong to benefit from boosting. The strongest detector, the circle classifier, had a vote of over half of the total confidence, giving it the only vote that matters. If the circle detector was taken out, the same held true for the next strongest detector. We needed a larger number of weaker classifiers for AdaBoost to be successful. In order to achieve our goal of near real time results, each evaluation of the classifier must be very fast.

Based on these problems and our requirements, we decided to try the features used by Viola & Jones in their face detection algorithm. These features are Haar-like filters that are applied to a specific region of the image to get a numerical response. These filters are composed of 2, 3, or 4 rectangles. In figure 2.7.1, the white rectangles refer to the positive sum of pixels within that region, which then subtracted by the sum of the pixels in the black regions. Although rough, these features provide an adequate description of boundaries.

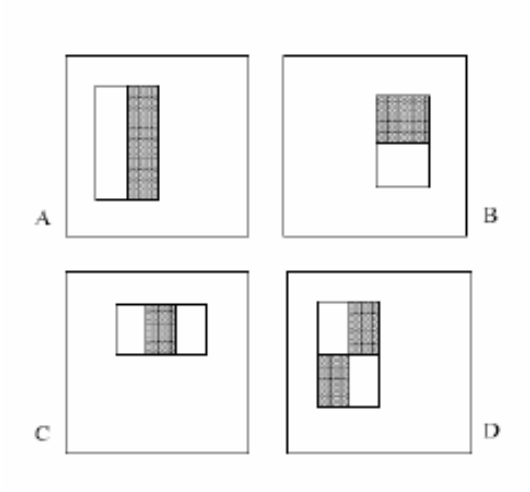


Figure 2.7.1: Haar-like filters of 2, 3, and 4 rectanagles [1]

2.7.1 Integral Image. The main advantage of the Haar wavelet features is that an image representation known as the integral image can be used for efficient computation of rectangle features. The integral image at location *x,y* contains the sum of the pixels above and to the left of *x,y*, inclusive. This representation is particularly efficient as it allows the sum within a rectangle to be computed with only 4 array references. The difference between any two rectangles can be computed in only 8 references. Because the Haar filters involve adjacent rectangles, less array references are actually required.

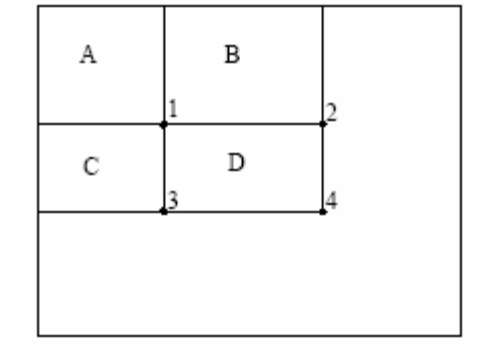


Figure 2.7.2: The value of the integral image at point 1 is the sum of the pixels in region A. The value at location 2 is A+B, and the value at location 3 is A+C. The value at location 4 is A+B+C+D. Thus, the sum of the pixels at D can be computed as 4-3-2+1. [1]

There were 4 variations of the Haar filter, as well as many possible sizes and positions. The parameters for a Haar feature were which of the 4 filters in figure 2.7.1 was used, the position of the upper-left of the filter within the window, and the width and height of the filter. Given these parameters and a 50x50 window, there are on the order of 10^6 possible Haar filters. We reduced the number down to 7000 by sub sampling from the entire Haar filter-space.

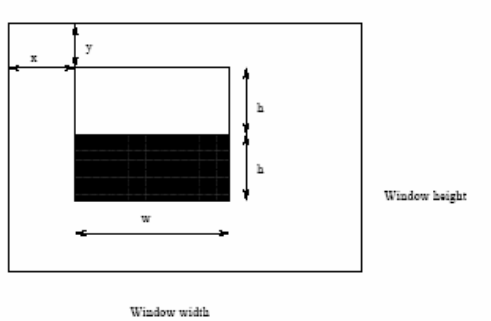


Figure 2.7.3: Illustration of the possible parameters for the Haar filters [1]

For a given sample image, we were able to pre-compute the integral image, which allowed for just a small number of array accesses per feature evaluation after the initial cost to compute the integral image. The training time of AdaBoost was further decreased by storing the error vector of the 7,000 features, allowing each iteration of AdaBoost to take time linear to the number of features, rather than varying with both the number of features and number of images.

The Haar filters generated ranged in accuracy. The best had an 89% accuracy rate with 6.1% false positive. The other filters chosen by AdaBoost ranged from 60-90% accuracy. These filters were of similar strength to the previous feature detectors we used, although they appeared to show greater independence from the large number available and specificness to just one rectangle of the window. Thus, AdaBoost achieved significant confidence values for all the classifiers chosen. Haar wavelets succeeded in meeting our requirements in weak classifiers: they benefit greatly from boosting, and are very fast to evaluate.

3. Results

3.1. AdaBoost Classifier

We trained our AdaBoost classifier with T=15 weak classifiers built from Haar-like filters. The final classifier had significant confidence for each weak feature detector. This classifier was applied to 50x50 windows in a given sample image to attempt to find regions with positive classifications. In the interest of speed, not every possible window was looked at. There was only an overlap of 40 rows/columns for neighboring windows. Due to the way we trained

AdaBoost, we recognized that each soccer ball should return multiple positive windows. We thus group overlapping windows into one region and take the centroid of a region as the location of the ball detected. To help eliminate false positives, we ruled out regions of positive response that did not have enough overlap, or where the region of sufficient overlap was not large enough.

3.2. Accuracy and Speed

We achieved 100% accuracy with 1.3% false positive on our training data. Test images and movies had similar results with proper backgrounds and soccer-ball size. It took on average 0.6 seconds to process a 320x240 image in Matlab. This met our initial goal of 1 second to process an image, as we recognized Matlab would not achieve speed results comparable to a program optimized in a lower level language such as C++. AdaBoost took approximately 1 hour to train on a sample set of a little over 1,000 images and 7,000 features. The majority of this time was the time required to evaluate each feature on each image. The training process was optimized by loading each image only once and computing the integral image only once. All 7,000 features were run once the integral image for a given sample image was computed.

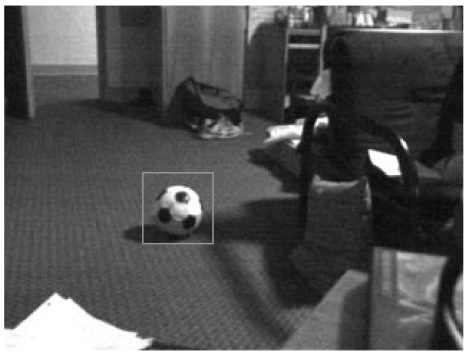


Figure 3.2.1: Detection of a soccer ball in a testing image taken from a webcam.

Figure 3.2.2: Successful detection of a soccer ball in one of our original training images before cropping.

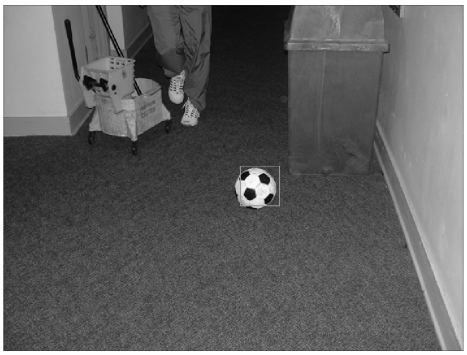


Figure 3.2.3: Successful detection in a test image taken with a digital camera.

4. Limitations

4.1. High False Positive Rate

Although our accuracy was sufficiently high, our algorithm suffered from a high false positive rate. For practical uses 1.3% is much too high, as in an average 320x240 image, there will be 7 false detection windows returned. Our measure of overlapping and discarding regions with not enough overlap worked to remove most of these false windows; however this approach caused accuracy to decline in some situations. In some images, ball detections were lost to regions with a large number of false positive windows due to the threshold we implemented. In the videos we tested, this only occurred in less than 10% of the frames.

To reduce the false positive rate, we could have implemented a cascade approach such as in Viola & Jones. We were limited in the scope of our project

with this respect from time constraints. However, with just 15 features chosen from 7,000 we were able to achieve the promising results we did. If we were to implement a multi-stage cascade choosing from a larger set of features we are confident we would get even stronger results.

4.2. Training Set

Another problem with our soccer ball detector is the training set we used to train AdaBoost. We had only 71 images of soccer balls, and 978 subwindows of non-soccer ball images. The training set size was again limited by time constraints on completion. As is, it took a little over 1 hour to train. On a more desirable set of close to 20,000 images in a training set, AdaBoost would have taken close to a day to train. If we had implemented cascade along with a sufficient training set, training time could have been on the order of weeks.

The main problem with the small training set that we observed was the limited ability to generalize. Our algorithm worked well in environments that had been presented in the sample images, but with unfamiliar backgrounds there were odd and slightly unpredictable results.

Further work on this topic could entail finding more efficient ways of training AdaBoost to allow a sufficiently large training set.

5. Conclusions

Although our exploration into a generic approach to object detection, trained in our case to detect soccer balls, was met with limited success, we were able to learn a lot about the techniques involved. We learned the importance of fast feature detectors in a boosted cascade scheme, where many features are evaluated for every window in question. We also learned about some of the limitations of AdaBoost. In particular, we found out that strong classifiers could possibly not benefit from boosting, and that dependant classifiers would most certainly not benefit from boosting.

Our experiments show that it is possible to generalize the techniques used in face detection for generalized object detection. Boosted cascade could be used for robotics, in particular robot soccer to detect soccer balls in real-time.

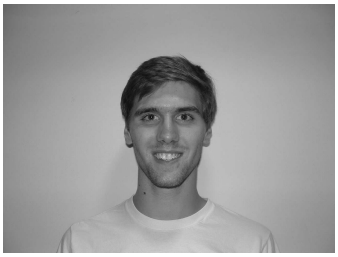
6. References

[1] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition*, volume I, pages 511-518. IEEE Computer Society, 2001.

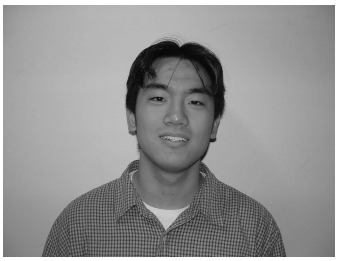
[2] Yoav Freund and Rober E. Schapire. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pages 148-156, 1996.

[3] Brendan McCane and Kevin Novins. On training cascade face detectors. In *Image and Computing NZ*, pages 239-244, 2003.

7. About the Authors



Kevin Caffrey is from Oakton, Virginia and is pursuing a Bachelor's degree in Computer Science as a sophomore at Carnegie Mellon University. He is a member of the varsity track team and a brother of the Kappa Delta Rho fraternity. Kevin is working at the Naval Research Laboratory over the summer.



Daniel Kim is from Silver Spring, Maryland and is pursuing a Bachelor's degree in Computer Science as a sophomore at Carnegie Mellon University. Daniel will be traveling to Korea for two weeks this summer after which he plans on doing research at the National Institute of Standards and Technology.

A Comparison of Background Modeling Techniques

Mike Schultz
Department of Electrical & Computer Engineering, Carnegie Mellon University
schultz2@andrew.cmu.edu

Abstract

This paper describes a comparison of seven background modeling techniques, including four pixel-level only methods, the Wallflower algorithm[1], the ComMode algorithm[2], and a novel technique, RegMode, based on the ComMode algorithm. These methods are tested on a database of videos with a stationary camera. No training data is provided. The performance of the algorithms is judged based on visual criteria alone.

1. Introduction

Background modeling is an attempt to extract out the stationary components from a portion of video. Often, these stationary components can be reduced to a single image. The background information can then be used for a wide variety of applications, including compression, segmentation, and tracking.

Many approaches to this problem have been suggested. The simplest are pixel level models. Purely statistical methods, such as the mean and mode, treat all pixels the same. Other methods attempt to do a crude segmentation to exclude some pixels which are obviously not background. More advanced approaches attempt to combine local pixel information into larger regions and use higher level logic to improve the results.

There are a number of problems and pitfalls inherent to background modeling. These are described in more detail in [1], but I will briefly describe them here. Objects may change states during the course of the video (i.e. background to foreground, foreground to background, or background to foreground to background). These types of problems are called "waking person", "sleeping person", and moved object problems, respectively. Lighting changes can also affect the background. These changes can be gradual, e.g. time of day, or sudden, e.g. light switch. Background objects may not be completely stationary, as in the case of waving trees. Foreground objects may blend in with the background, as in the case of camouflage. Pixels on the interior of a foreground region may not change as rapidly as the borders, and thus may be identified as background ("foreground aperture"). Finally, shadows case by foreground object share properties with both foreground and background objects.

2. Pixel-level only methods

The pixel level only methods tested for comparison purposes were mean, mode, adjacent frame difference, and linear prediction. The mean background was created by replacing each pixel by the mean of its time sequence. The mode background was created by replacing each pixel by the most common value in its time sequence, i.e. its statistical mode. It is important to note that in the implementation that was tested the RGB channels were treated separately for mode calculation. Thus the RGB value that the background pixel takes may not be a value that occurs in its time series. This method may is called the Bayesian Decision since the mode is essentially the value with the highest probability when the normalized histogram is taken as an estimate of the pmf.

The adjacent frame difference method looked at the difference between two consecutive frames. Any pixels that exhibited change above a certain threshold were considered foreground. In this way a crude segmentation of the time series for each pixel was accomplished. This effectively marked only transition regions as foreground and left more temporally consistent regions as background. The final background estimate for each was determined by taking the mean of the values in the time sequence marked as background.

There are many ways that linear prediction may be used to estimate the background. The exact equations and methods may be found in a textbook. In this method, Levinson-Durbin Recursion was used to estimate linear predictor coefficients for a one-step ahead Wiener filter predictor based on the entire sample of video. The results shown later used 30 predictor coefficients. It is important to note that this method

varies slightly from the way that linear prediction is used in the Wallflower algorithm, as will be discussed in the next section.

3. The Wallflower Algorithm

The Wallflower algorithm [1] is a multi-level algorithm consisting of three types of processing. It was designed for use in security camera and time-lapse video situations. The motivation in Wallflower's development was an algorithm that, given some training data, could maintain the estimate of the background over an extended period of time and could handle most of the problems identified in the introduction.

3.1 Pixel-level

At the pixel-level, Wallflower attempts to produce preliminary estimates of the background by utilizing linear prediction. For each pixel, it keeps a history of the last 50 actual values and predicted values. For each new sample, it computes two sets of 30 prediction coefficients for a Wiener filter, one using the actual values and one using the predicted values. If the new actual value is less than a threshold, the pixel is considered to be background.

The model based on the predicted values is used to account for a foreground object corrupting the history of actual values. The coefficients are recomputed at each frame to allow for adaptation, such a time of day. If the new coefficients have an expected squared error less than 1.1 times the previous error then they are kept. Otherwise, the old coefficients are retained.

3.2 Region-level

The region level processing in Wallflower attempts to avoid the foreground aperture problem. Pixels on the interior of a homogeneous moving object may be classified as background by the pixel level processing. To solve this, Wallflower uses the adjacent frame difference method. If this method tags a pixel as foreground in two consecutive frames then it is clearly a foreground pixel. Combining this information with the foreground data from the pixel-level, regions are identified as definitely foreground. Contiguous foreground pixels are then grouped together. The histogram of each group is then used to expand the region. If the histogram value of a pixel adjacent to a group is above a certain threshold that pixel is added to the group. In this way, the foreground region is expanded to include areas that were previously marked background.

Note: In the implementation used for comparison in this paper, the histogram used for each group was not full histogram of with a bin for every value, but a histogram with 64 bins for each channel.

(a) A homogeneous disk moves to the right. Change is visible in the black regions only (J_{t-1} in text).

(b) The same thing happens one frame later (J_t).

(c) Only the intersection ($J_{t-1} \wedge J_t$) is certain to be foreground in the middle image.

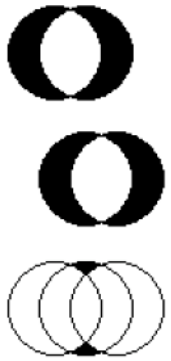


Figure 1: Finding the right foreground in regional level processing

3.3 Frame-level

The frame level processing for Wallflower is designed to handle the light switch problem. To do this Wallflower maintains multiple pixel-level and region-level models at once. Each model is designed for a different set of circumstances (e.g. lights on, lights off), but only one model is active at any time. If the number of foreground pixels exceeds 70% of the total number of pixels, Wallflower considers switching models.

This level of processing requires training data for each set of circumstances. Since the limitations of this comparison were such that there was no training data, this level of processing was eliminated in the implementation.

4. The ComMode Algorithm

The ComMode, "competitive mode estimation", algorithm[2] was designed to do background modeling and object segmentation on any portion of video that had either a stable camera or had been aligned. The basic concept is that pixels were decomposed into a number of operating modes based on self-consistent regions. The most consistent mode was selected as an initial background estimate. Surrounding pixels then "voted" for or against each mode based on the similarity between it and their own background estimate. The background is then chosen to be the mean value of the estimate background mode for each pixel.

4.1 Mode Determination

Each pixel is divided into its modes based on the variance of its time sequence. The standard deviation over a sliding window of pixels is computed. When this deviation is above a threshold the pixels are no longer considered consistent. Contiguous regions of consistent pixels are then grouped into modes, with their intervals being bordered by the inconsistent

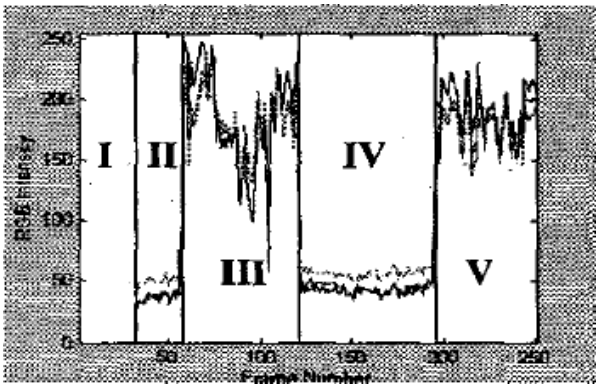
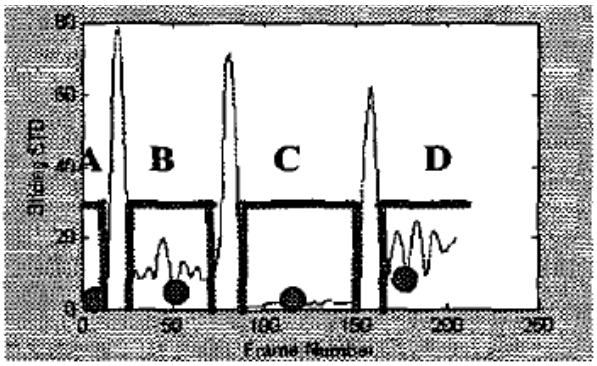


Figure 2: A typical time sequence divided into four modes
Figure3: A standard deviation sequence and the resulting mode division



pixels. These modes may then be combined with other modes if their mean RGB value close enough. In this implementation, a window of 5 pixels was used and the threshold for consistent regions was a standard deviation of 5.

The initial estimate for the background mode of each pixel is then determined and given a confidence. For pixels with only one mode are given a confidence of one. For other pixels, the mode with the least mean standard deviation of its time series, that is the most consistent mode, is chosen as the initial estimate and given confidence of 1/N, where N is the number of modes at that pixel.

4.2 Mode Competition & Voting

Each mode then competes with the other modes through a voting system using surrounding pixels. The set of voting pixels is determined by a radius around the current pixel, or, as in the case of this implementation, by a square of certain size around the current pixel. I used a square of 7 pixels, i.e. +/- 3 pixels in each direction. Each pixel within the voting set casts a vote for each mode in the current pixel of the value of its voting confidence times the similarity of that mode with its own estimate. Each pixel in the voting set then casts votes against each mode of the value of its voting confidence times the similarity of the mode with its own mode that is most similar.

The voting confidence is determined by its own confidence in its estimate times the distance from the current pixel. The similarity between two modes is based on color difference measure and a temporal distance measure. The color difference is 1 minus the sum of the absolute difference in each channel (each value is normalized from 0 to 1, instead of 0 to 255). The temporal distance measure is the length of the intersection of their time intervals divided by the minimum length of the individual intervals.

After each round of voting, the mode with the highest voting value is chose as the new estimate. The confidence is set to the voting value divided by the sum of the voting values for all modes. If the confidence is above a threshold, it is set to 1.

The ComMode paper[2] suggests that the voting process converges after no more than 5-10 iterations, but I found that it converges after about 3 iterations.

The background image is taken to be the mean value of the mode estimate at the end of the voting process.

5. The RegMode Algorithm

The RegMode, “regional mode estimation”, algorithm is essentially an adaptation of the ComMode algorithm, although it takes an approach similar to Wallflower. It attempts to do regional processing on the modes of each pixel in order to utilize larger scale information.

5.1 Motivation

The motivation behind RegMode is to use the best aspects and ideas of each of the previously described algorithms and eliminate their weaknesses. ComMode seems to efficiently represent the ways the pixels behave with its mode methodology. However, the voting process is very computationally expensive and does not seem to produce results good enough to merit the computation time. Wallflower’s pixel level processing is also very computationally intensive and does not provide any information to higher level processing other than a crude foreground/background distinction.

RegMode was developed with the real-world video in mind. Foreground objects in real video behave according to physics. In video, they tend to appear in adjacent pixels and at adjacent times. If these objects are the basis for the modes in a single pixel, then adjacent pixels should have modes that behave similarly. Thus, it would be useful to group these modes together into a larger regional mode to give a better description of the foreground object. Ideally, the background would be grouped into a single mode which would be the most prevalent regional mode.

5.2 Pixel Mode Determination

RegMode begins by determining the modes in exactly the same manner as ComMode. Pixel regions with low variance are said to be consistent and grouped into modes. These modes are then clustered with other modes that have a similar RGB value. The mean RGB value for the new composite modes is recomputed to be used in the next step of processing. In order to save memory, especially for long video files, the modes are represented merely by successive start and end points of the interval.

5.3 Regional Mode Determination

The pixel-level modes are then grouped into regional modes. First, each pixel-level mode is considered to be its own regional mode. Next, each mode is compared to every mode of pixels adjacent to it. Modes with the same pixel are already assumed to not be similar due to clustering in the previous stage. Mode comparison is based on a similarity measure which is based on both color and time interval. The color component of the similarity measure is the sum of the absolute difference in each channel (just as in ComMode, except that these channel values are not normalized to 0 to1). The time interval component measures the percentage of overlap between the two intervals. The overlap is one minus the length of the intersection of the two intervals divided by the length of the longer interval. The final similarity measure is the product of the two components. If the similarity measure between the two modes is less than a threshold, the two modes are considered equivalent.

5.4 Background Synthesis

Once all equivalent modes are determined, each pixel-level mode has a regional mode that is associated with. The regional modes are then sorted based on number of times they occur. At each pixel, the pixel-level mode that has highest rank regional mode is chosen to be the background mode for that pixel. The background image is then composed of the mean RGB value for this mode at each pixel.

6. Results

In many of the clips tested, the results for each algorithm came out very similar. The statistical mode was consistently the best estimate of the four pixel level only models. The mean, frame difference and linear prediction methods produced very similar blurry results.. In uncertain regions Wallflower produced blurred results, whereas ComMode and RegMode produced jumbled and distorted images, very similar to the statistical mode. ComMode consistently produced glaring artifacts as can be seen in Figures 4-5. RegMode produced artifacts as well, although its artifacts were generally limited to areas that other algorithms got wrong (Fig 5). All algorithms had difficulty with regions where the background was visible for only a short period, but the statistical mode and RegMode performed better than the other methods

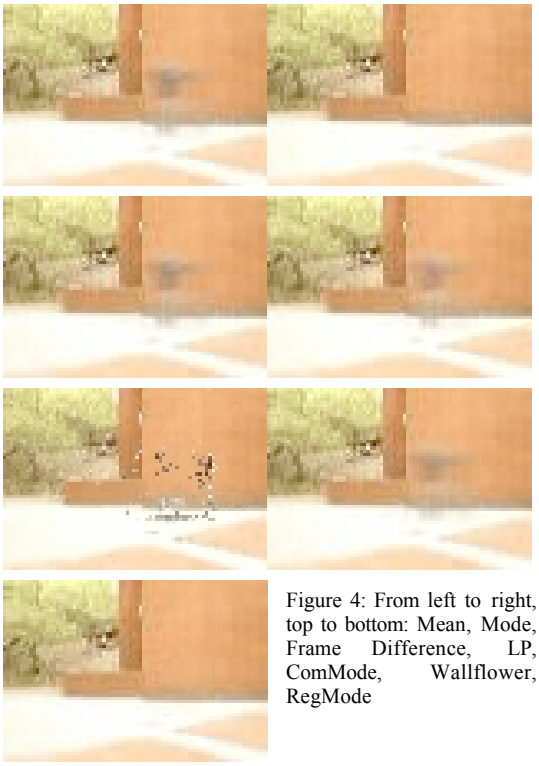


Figure 4: From left to right, top to bottom: Mean, Mode, Frame Difference, LP, ComMode, Wallflower, RegMode

6.1 Computation Time

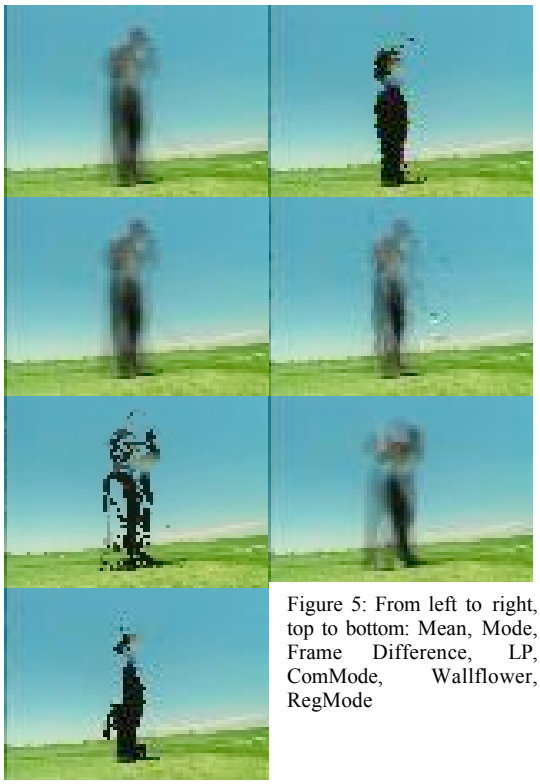


Figure 5: From left to right, top to bottom: Mean, Mode, Frame Difference, LP, ComMode, Wallflower, RegMode

The four pixel-level only methods had relatively insignificant processing time. For larger images and longer videos, linear prediction took a few seconds. The other three methods took less than a second.

The higher level methods had considerably longer processing time. RegMode was always noticeably faster than both ComMode and Wallflower. For small videos of short duration, ComMode finished quicker than Wallflower, although its processing time increased rapidly as the clip size becomes larger. Figure 4 and 5 illustrate the computation time as a function of the total number of pixels (i.e. pixels per frame times the number of frames). Although this is not the ideal axis for each method, I feel it gives a general idea of the computational load.

I would estimate that Wallflower's processing time is roughly linear with respect to total number of pixels (minus the 50 frames necessary for bootstrapping). ComMode and RegMode seem to be some nonlinear function of the total number of pixels. ComMode's processing time is almost entirely dominated by the voting process which is a function of number of pixels and other voting parameters. However, in both ComMode and RegMode, the number of modes found in the image have significant impact.

7. Conclusions

From my results, I have shown that the fastest and most efficient estimate for this type of videos is the statistical mode. The other three pixel-level only methods gave results pretty similar to what I expected. – decent but not great. The three more complicated methods have given me mixed results

Pixel/frame* number of frames	Wallflower	ComMode	RegMode
432000	283.3	146	31.54
1094400	160.1	899	160.1
1361920	1713.4	7542	292.0
1644160	2103.3	12246	474.4

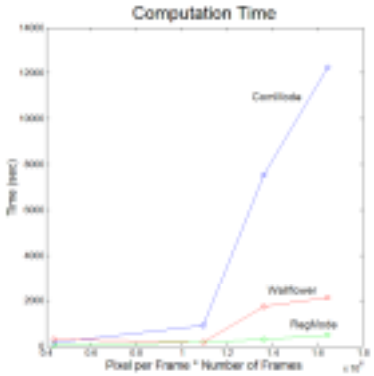


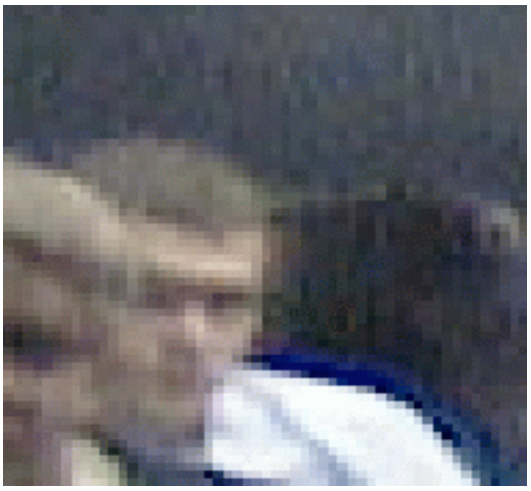
Figure 6 & 7

The ComMode algorithm produces too many artifacts and takes far too much computation to be an effective method. Although the algorithm functions well on small videos, any videos of significant size cause the processing time to increase seemingly exponentially. The results that I got for ComMode are much less spectacular than those mentioned in the ComMode paper, which leads me to believe that I may have made an error in implementation.

The Wallflower method works satisfactorily, but its results are not that much better than simpler methods such as pure linear prediction or frame difference. It seems to perform poorly on high frame rates and probably would work better in its intended application of time lapse security cameras. Wallflower use of linear prediction allows it to adapt and handle oscillating backgrounds, but the algorithm does not take full advantage of the predictions, in that they are only used for a background/foreground segmentation. The processing time is linear with the number of frames and number of pixels, making it a feasible method for most video sizes.

The RegMode method has clearly shown its merits in both its fast processing time and good estimates. It succeeds in some areas where other methods fail. Overall, I think it is a notable method, but my algorithm clearly does not model the background the way I had hope it would. I think the similarity measure needs to be refined and tested a little more.

Due to time constraints I only had time to test a couple of measures.



8. References

[1] Toyama, K. Krumm J., Brumitt, B., Meyers, B., "Wallflower: Principles and Practice of Background Maintenance", *ICCV99* (255-261).

[2] Lipton, A. J. and N. Haering, "ComMode: An Algorithm for Video Background Modeling and Object Segmentation." *ICARCV02* (1603-1608).

The Author:

Mike Schultz is a master student in EECS at Carnegie Mellon. He is taking this course for graduate 15-685 credit, which requires him to tackle with more problem sets and a more substantial term project.

Object Tracking by On-Line Learning of Motion Models

Chytra Pawashe
Dept. of Mechanical Engineering
Carnegie Mellon University, Pittsburgh PA 15289-4377, USA
chytra@cmu.edu

Abstract

We address the problem of tracking multiple identical objects accurately in situations where objects follow motion models. A deterministic and probabilistic model of motion based on a Gaussian distribution was implemented for the moving objects in this study. Furthermore, objects were subjected to collisions among world edges and other objects. By tracking these objects, we hope to be able to determine the underlying models of motion and use this information to improve object motion prediction on-line; this theoretically would allow object tracking to become more accurate and become resilient to collisions. We found that for both the deterministic and probabilistic motion models, prediction accuracy is very close to the actual motion. However the deterministic model shows some resilience to collisions, while the probabilistic model fails under collisions.

1. Introduction

The human visual system is robust when tracking multiple objects. It can analyze motion models, detect collisions, and interpolate data while tracking. For example, if an observer is tracking one car on a busy road, and the observer is temporarily blinded from the car (perhaps by a building), the observer has a good idea of when and where the car will reappear.

The human visual system is limited primarily by attentional resources. On average, the visual system can accurately track four to five objects, depending on the context [1]. Furthermore, the visual system does not derive motion from monitoring objects on the fovea; eye movements do not contribute to motion detection. Instead, it is thought that mechanisms of attention and attentional shifts contribute to object tracking, as well as specific features in the object motions [4].

Computers, on the other hand, do not have limits in resources like humans. However a sophisticated tracking algorithm, which humans possess, is a missing idea.

In many cases, the motions of objects follow patterns. These patterns can follow a deterministic model, which implies that there is only one possible next position of a given object – that its motion is determined. The motion pattern can also be probabilistic, which means that the next position of an object is unknown, but has a certain probability of being at certain points. This motion can appear as being random, but indeed follows a distinct pattern.

The goal of this study is to develop an algorithm that can find the underlying motion patterns of a given set of identical objects that follow varying motion models. Once the motion model is known, the objective is to improve object tracking by creating a prediction model based on the derived underlying motion model. In this study, we worked with a deterministic and a probabilistic motion model.

2. Methods

MATLAB was used to create, using simple computer graphics, movies of objects following certain motion models. MATLAB was also used to read the generated movie and track objects in the movie using the algorithms described in this paper. Analysis data and graphs were also generated to compare motion models and tracking algorithms.

2.1. Creating deterministic motion

In the deterministic motion model, the speed of an object with respect to time is characterized by a 2D 24-element Gaussian with a σ ranging from 1 to 2 (Fig 1); the Gaussian is also normalized to produce adequate movement in the motion field, with velocity being in units of pixels per frame. Typically, a

Gaussian with a higher σ appears smoother and corresponds with smoother, slower object motion.

The Gaussian repeats itself when it reaches its end, or after the 24th element in the Gaussian distribution. This shows movement that is oscillatory. Fig. 2 portrays a typical speed-time graph of an object under this motion model.

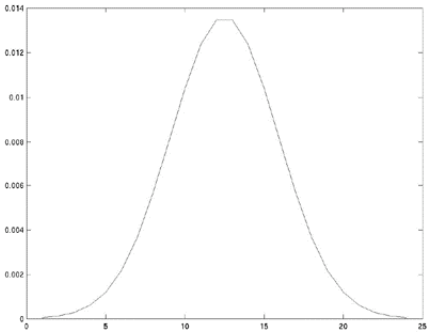


Figure 1. Typical Gaussian Distribution

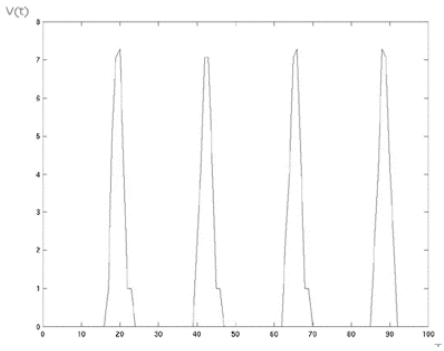


Figure 2. V vs. T for Deterministic Motion Model

Furthermore, objects are initially given a pre-defined starting position, a random propagation angle, and a random starting speed based on the Gaussian model (i.e. a possible random start speed corresponds to starting at element 10 of 24 in the Gaussian distribution). The next position of the object (P') is the current position (P) plus a displacement, which is determined by the speed (S) and propagation angle (θ) with simple trigonometric functions:

$$P'(x) = P(x) + S \cdot \sin(\theta)$$
$$P'(y) = P(y) + S \cdot \cos(\theta)$$

Collisions between balls and edges are processed using simple physics, where the angle of incidence to an obstacle equals the angle of refraction from the obstacle (Fig. 3).

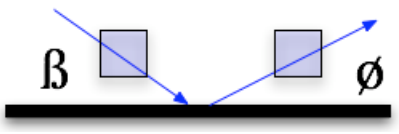


Figure 3. C ollision processing, angle of incidence (ß) equals angle of refraction (ø)

100 frames of motion were produced in a 300x300 pixel movie, for 1-10 5x5 pixel objects with varying σ values. Fig. 4 displays a typical frame of the described motion field.

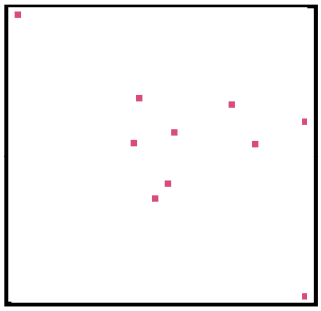


Figure 4. A Frame of objects

2.2. Creating probabilistic motion

In probabilistic motion, the speed and angle of propagation of an object is determined probabilistically based on a Gaussian. A 2D 24-element Gaussian with σ ranging from 1 to 3 was used (see Fig. 1). The next speed or angle is determined as described in the following pseudo-code:

```
1    ΣGaussian = 1
2    X = RandomInteger(1 to 24)
3    T = RandomNumber(0 to 1)
4    P = Gaussian(X)
5    if P > T use P, else goto 2
```

Thus an object's speed and angle, when viewed as a histogram, will resemble the Gaussian it is associated with. Furthermore, the value found from the Gaussian is normalized to achieve adequate motion in speed (in pixels/frame), and between 0 and 2π for angle.

The actual movement of an object in probabilistic motion looks as if it were random. However the object tends to move in one direction with small random movements. Fig. 5 displays the velocity probabilities; longer arrows are more probable, and the object tends to the top-right. In the motion in this study, the objects tend to move to the right side.

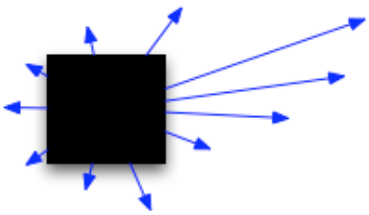


Figure 5. Portrayal of probable velocities for an object; this object will tend to move in the top-right direction

Finally, objects are given a pre-defined coordinate starting position. Edge and collision detection are employed using the physics described in Sec. 2.1, and 100 frames of motion in a 300x300 pixel world for 1-10 5x5 pixel objects were created.

2.3. Primitive Tracking

Initially, underlying motion models are unknown while tracking objects. A simple method to track objects is needed while the underlying model is being deciphered. One method to track without knowing the underlying model is to assign objects using distances. For an object, this simply means that the object in the next frame that has the closest distance to the object in the current frame is the object being tracked. In this method, distance is defined (between objects A and B) as:

$$D(A,B) = \sqrt{(Ax - Bx)^2 + (Ay - By)^2}$$

Another way to primitively track is to use the object's last velocity to create a prediction model. Essentially the predicted position of the object in the next frame will be the object's last velocity added to its current position. Then, the closest object to the predicted position in the next frame will be assigned as the object being tracked. However, this will only work with the deterministic model, as the probabilistic model produces unpredictable velocities (the last velocity is not relevant to the next).

2.4. Finding Underlying Models

Initially, the ball is being tracked using the methods of primitive tracking (Sec. 2.3). While it is being tracked, a speed-time relation is being constructed.

For deterministic motion, the speed-time relation is oscillatory (Fig. 2). Because the speed-time graph directly resembles a Gaussian, we can directly compare it to a Gaussian at the relevant sections to get the σ value. The first point where the Gaussian is known occurs at the first peak, as the varying Gaussians have different peak sizes. These peaks are directly compared to the peaks of Gaussians of different σ until a match is found. Once found, the underlying propagation model is determined.

For the probabilistic model, the velocity appears random, portrayed in Fig. 6. However, the average speed depends on the σ of the Gaussian; the average speed is greater for smaller σ values. With this in mind, the average speed of the object is constantly calculated and compared to a table that relates Gaussians to average speeds (which was pre-calculated, but can be done on the fly). In this fashion, the underlying Gaussian model is determined.

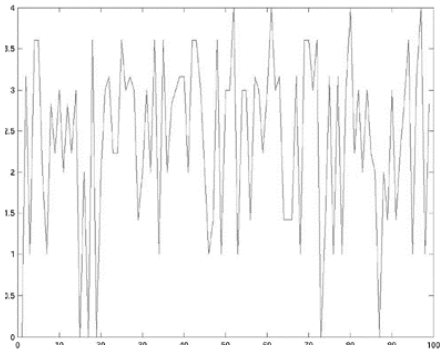


Figure 6. Speed-time graph of probabilistic movement

2.5. Predicting deterministic motion

For deterministic motion, once the Gaussian model is known for an object, velocities and positions are predicted in the same way for creating objects that follow deterministic motion (Sec. 2.1). Simply, the next speed is determined by increasing the index of the Gaussian distribution, and the predicted change in distance in the x and y-axes is determined by using the angle of propagation (calculated with the previous position of the object) and trigonometric functions. Furthermore, collision and edge detection is employed for the prediction model to increase the robustness in the prediction model (Sec. 2.1). Finally, the object in the next frame that is closest to the predicted object will be assigned as the object being tracked.

To correct prediction in the case that the object is using an incorrect index on its Gaussian, the net displacement and velocity associated with the displacement is calculated after the assignment to the next frame is made. Depending on where the index of the Gaussian is and whether this just calculated velocity does not correspond with the prediction, the index on the Gaussian is adjusted.

2.6. Predicting probabilistic motion

In probabilistic motion, once the Gaussian model is known, the next position of an object cannot be determined with high certainty. However since there is a probability model, some possible future positions are more likely than others for an object. To take advantage of this notion, particle filtering is used.

In the idea of particle filtering, small particle objects are used to track objects. Particles that are closer to an object will carry more weight, while particles that are far from an object carry low or no weight. In the next update cycle in the next frame, the particles with low weights are removed. An equivalently removed number of particles are spawned around the particles that had higher weights [2]. By this process, particles constantly chase an object that is tracked.

To use particles to create a prediction model in probabilistic motion, a predicted position is calculated with the known Gaussian model, as described in Sec. 2.2. A particle is then put on the predicted position, and this process is repeated until all particles have been put on a predicted position (20 particles were used in actual prediction). Collision and edge detection are also employed per particle to prevent particles developing on impossible places. More particles will lie on more probable positions, as portrayed in Fig. 7.

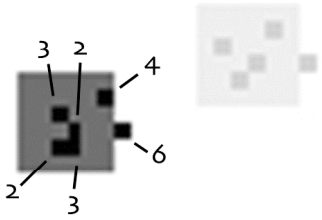


Figure 7. Particle prediction, more particles are on more probable positions

With the particles in their predictive positions, the next frame is considered. First, all objects in the next frame are blurred with a 64x64 $\sigma = 8$ Gaussian so that the objects can bleed to their surroundings – this allows particles that are closer to the centers of objects to acquire a higher weight than particles that are farther away. Once all weights are found, a mean probable position $\overline{P}(x,y)$ is calculated from the particles, and is defined as:

$$\overline{P}(x,y) = \frac{\sum_i P_i(x,y) * W_i}{\sum_i W_i}$$

Where $P_i(x,y)$ is the position of the i^{th} particle and W_i is the weight of the i^{th} particle.

Effectively, the mean probable position is the most likely next position of the object being tracked, thus the object in the next frame that is closest to the mean probable position is assigned as the object being tracked.

3. Results

3.1. Deterministic model results

In the deterministic model, the predictive model using the underlying motion is compared to the primitive model that uses the previous velocity of the object. In object collisions involving two objects, the underlying model prediction rarely missed tracking. However in the primitive prediction model, collisions tended to confuse tracking and incorrect balls were updated while tracking.

With a ball that collides against two wall edges, the predicted positions in the underlying model are compared to the actual positions (Fig. 8). The predictions from the last velocity model are also compared to the actual positions (Fig. 9).

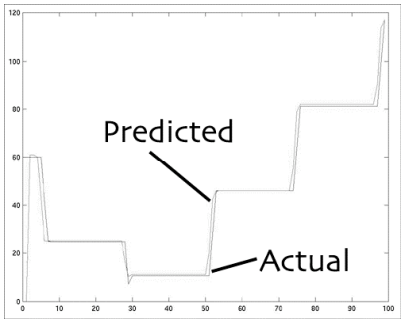


Figure 8. Pr ediction using derived underlying model vs. Actual positions in the y-axis

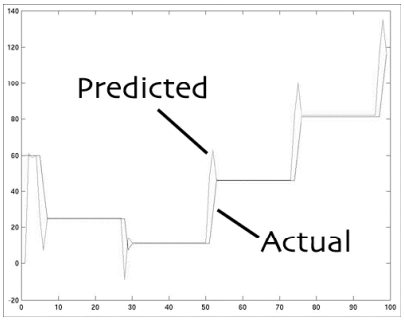


Figure 9. Prediction using last velocity model vs. Actual positions in the y-axis

Furthermore, tracking was tested in a setup with many collisions. 9 balls were clustered initially and set up to converge onto each other. In both primitive and underlying prediction models, significant mistracking was observed. Fig. 10 portrays the clustered setup.

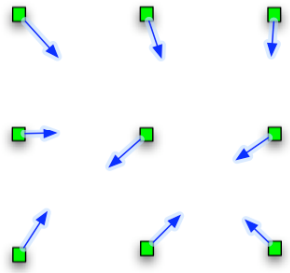


Figure 10. Clustered objects that will collide

3.2. Probabilistic model results

In the probabilistic prediction model with particles, tracking was tested among collisions. Mistracking occurred often during collisions of two or more objects. Mistracking occurred significantly in clusters of objects (Fig. 10).

The probabilistic prediction model was compared to the actual positions of a single, undisturbed object. The mean probable position and the actual position are displayed in Fig. 11.

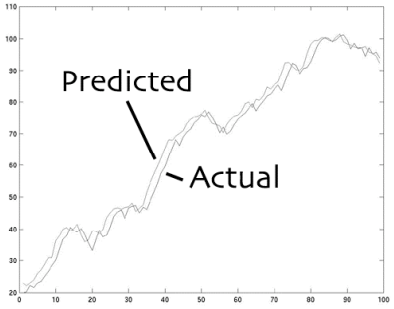


Figure 11. Mean probable prediction vs. Actual positions in the x-axis.

4. Discussion

4.1. Deterministic model

The advanced prediction model that determines the underlying model of deterministic motion produces good results and works well compared to the primitive prediction model using the object's previous velocity. With Fig. 8 and Fig. 9 in mind, the advanced prediction model can almost perfectly characterize the motion of the tracked object.

Collisions are handled decently in the advanced prediction model – much better than the primitive model. For one-on-one collisions, tracking is not a problem. However in areas of many objects, both prediction models fail. It is very possible that the advanced prediction model needs to be optimized and account for the physics of the motion models better. Theoretically, it should be possible that the advanced prediction model produces perfect or near perfect results, particularly among collisions.

Furthermore, the model is not known immediately; it takes several frames of motion for a Gaussian to develop. During this learning phase, if a given object is disturbed, the derived motion model can be heavily skewed from its actual model.

The drawback of this approach is that it assumes a deterministic motion model, and the type of model has to be known (we are using a Gaussian oscillation model). Essentially this algorithm has to be designed for a very specific propagation model and will not work well with any other. However, results can be very accurate in this approach, if a deterministic motion model was the problem at hand.

4.2. Probabilistic model

In the probabilistic model, the mean probable predictions are adequately close to the actual positions (Fig. 11), indicating that prediction is good. However collisions among objects cause the prediction algorithm to fail. It is likely that this occurs because an opposing object moves to the most probable next position of the object being tracked, thus causing mistracking. Optimizing the number of particles and adjusting the way objects bleed (we are using a 64x64 $\sigma = 8$ Gaussian blur) may reduce the mistracking.

However, it seems that the probabilistic tracking algorithm can be very well suited for single object tracking with a lot of noise, i.e. vibrations. We can also limit resources by reducing the number of particles associated to an object, which relates to human attention. If resources were to be limited, object tracking could also occur without having most of the next frame information at all, as particles only need to check weights in very specific locations.

Because the probabilistic model produces noise, a Kalman filter [3] may be useful to reduce the noise and find a more general propagation model. This would allow us to focus on an object's tendencies and global movement, rather than the insignificant local vibrations. However, detecting collisions still has to be considered.

5. Conclusion

We have investigated methods to accurately track objects following deterministic and probabilistic models. In both models, improved tracking is possible if the underlying models are deciphered. However other identical objects, particularly during collisions, tend to confuse the algorithms. Furthermore, the type of model an object follows needs to be known, a limitation the human tracking system is not subjected to. Thus, much work needs to be done in object tracking to approach human ability.

6. References

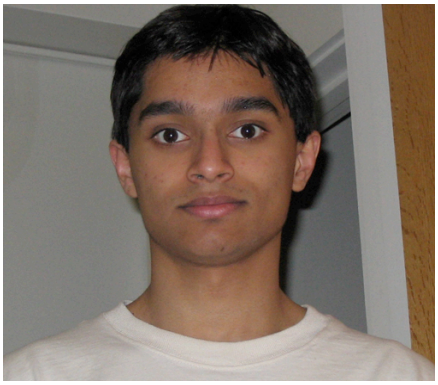
[1] Culham, Jody C., Brandt, Stephan A., Cavanagh, Patrick. "Cortical fMRI Activation Produced by Attentive Tracking of Moving Targets", *Journal of Neurophysiology (80)*, 1998, pp. 2657-2670.

[2] Hue, Carine, Le Cadre, Jean-Pierre, Pérez, Patrick. "A Particle Filter to Track Multiple Objects", IRISA Campus de Beaulieu & Microsoft Research. 2001.

[3] Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems", *Journal of Basic Engineering,* 1960.

[4] Verstraten, Frans A.J., Hooge, Ignace T.C., Culham, Jody, Van Wezel, Richard J.A. "Systematic eye movements do not account for the perception of motion during attentive tracking", *Vision Research (41)*, Pergamon, 2001, pp. 3505-3511.

7. About the Author



Chytra Pawashe is a sophomore enrolled in the bachelors program in Mechanical Engineering at Carnegie Mellon University. In addition, Chytra is pursuing a minor in robotics. His interests lie in the fields of robotics, computing, and nanotechnology and aims to pursue these through an engineering approach. In his free time, Chytra performs research in the Nanorobotics Laboratory at Carnegie Mellon University, studying engineering topics at the micro and nano-scale level.

Filling in missing parts of images

Andres Ivan Jager
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15217

Abstract

This paper describes a way of filling in missing parts of an image. My goal was to make it as un-noticeable as possible, while not requiring any external knowledge, and as little user intervention as possible.

1. Introduction

The purpose of this project is to be able to remove a certain area of a picture, and have it filled in, such that it is not noticeable. The main use for this would be removing unwanted objects from the foreground, such as someone's hand at the edge of the picture, a bug flying in front of the cammera, or a stain on someone's shirt.

Since this is for a Computer Vision class, and not an Artificial Intelligence class, my objective is only to fill in the hole such that it doesn't stand out, not to make it look real even when you know it is not. Because people know a lot about the real world, it would be very hard to fill in a hole such that we couldn't see it even if we were looking for it. Doing so would require a lot of knowledge, ranging from what the Coca-Cola logo looks like and where you would expect it, to who's face you would expect to see next to a picture of the current president.

Originally, my idea was to first decide what part of the hole belongs to each object that surrounds it, and then fill it in with the appropriate color/texture/gradient. As it turns out, texture synthesis ends up doing this implicitly.

1.1. Previous Work

There has been quite a bit of work in Texture Synthesis, but by looking at the results, the best seems to be Efros and Leung's algorithm [1].

There are a lot of other methods, but none of them seemed as well suited for the job. Most of them don't do constrained synthesis, which is what I want, since the generated texture is supposed to look good in the context of the rest of the picture.

There are also a lot of optimizations to Efros and Leung's original algorithm. Efros' Texture Synthesis home page [3] has links to them.

I decided to just implement the original algorithm, since this doesn't need to be real-time, and for this class we need to implement it in matlab, which is the slowest language I've ever seen.

2. Methods

I started by implementing the algorithm almost exactly as described in [2].

Since for this project the source image is always the same as the destination image, the source image has invalid pixels in it. Originaly I compensated for this by simply leting the valid mask in `find_matches()` be the intersection of the valid mask for the template, and the valid mask for the part of the image I was testing. I also ignored the parts of the image where the center pixel was not valid.

Basically the algorithm is like this:
1. Find all invalid pixels with valid neighbors.
2. Randomize that list and sort it in descending order by the number of valid neigbors.
3. For each of these pixels, find the best matches, and pick one randomly.
4. Repeat until there are no more invalid pixels.

`find_matches()`

Finding the best matches is as follows:
1. Let `template` be the square region around the pixel we are finding matches for.
2. For each valid pixel in the image take a window of the same size around the pixel, and calculate the distance in color space between each pixel in that window and the corresponding pixel in `template`.
3. Set the difference between invalid pixels to 0, and weight the others with a gaussian, so that pixels near the center are weighted more.
4. Sum the resulting differences, and normalize it based on how many pixels were valid. This is the error for each given pixel.
5. Return the pixels whose error is near enough to the lowest error found.

One problem I was having was that when trying to fill in the trees in the picture of Efros and Leung, it would end up

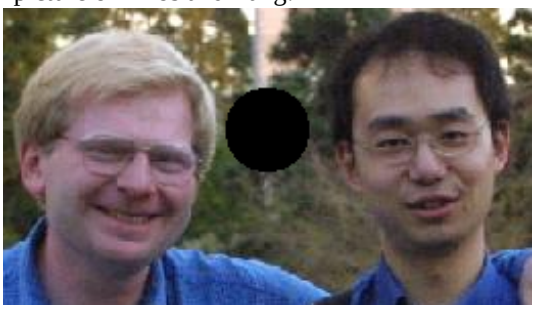
filling in the hole with 4 or 5 solid colors. This seemed to be because the trees are rather blury, causing the best match for a pixel to be one of the pixels right next to it. Since in my implementation the valid mask is just another layer of the image, and as such is a double precicion floating point, I abused it to indicate how "good" that pixel was, rather than just whether it is valid or not. This discourages using the same pixel value over and over again.

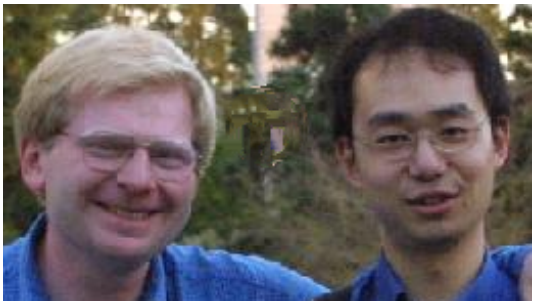
Another problem was picking a pixel where the neighborhood window didn't really have enough valid pixels for the comparison to be accurate. For example, when all the pixels to the left are invalid in the template, and all the pixels to the right of the neighborhood window were invalid, the comparison was entirely based on the pixels directly above and below, which is not really enough information. This problem was solved by requiring at least 1/4 of the pixels in the intersection of the template and the region we are checking to be valid.

3. Results

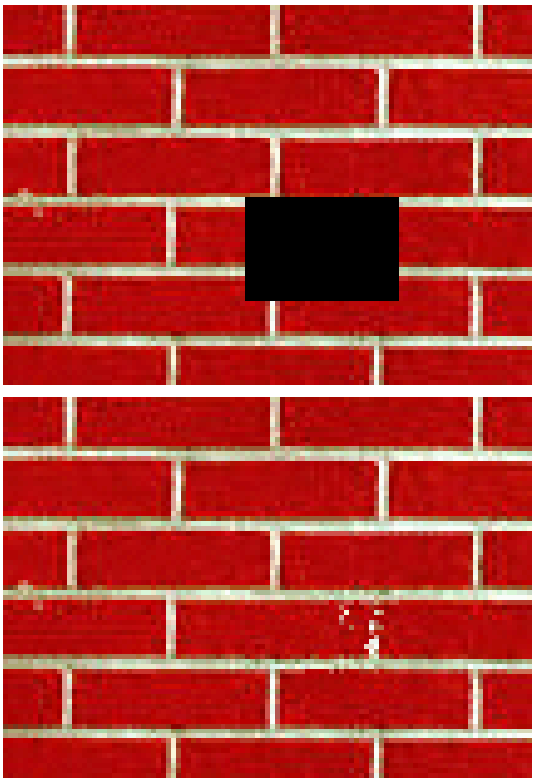
Matlab turned out to be even slower than I thought, so I didn't have much time to try a lot of different test cases. It is probably allocating and freeing a lot of memory it doesn't really need to, but I decided to keep my code legible rather than optimizing it for Matlab. Making a lot of the variables global would probably make it a lot faster, but it would be horrible coding style, so I decided against it.

These were done before requiring at least 1/4 of the intersection to be valid:
A picture of Efros and Lang:

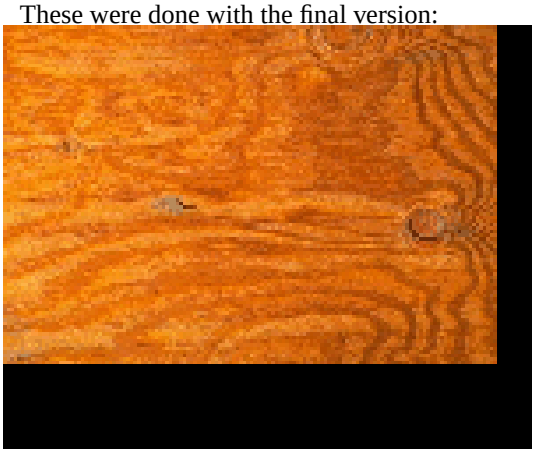


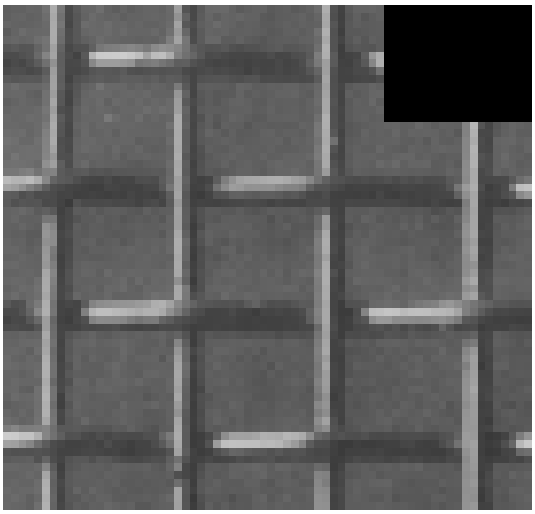


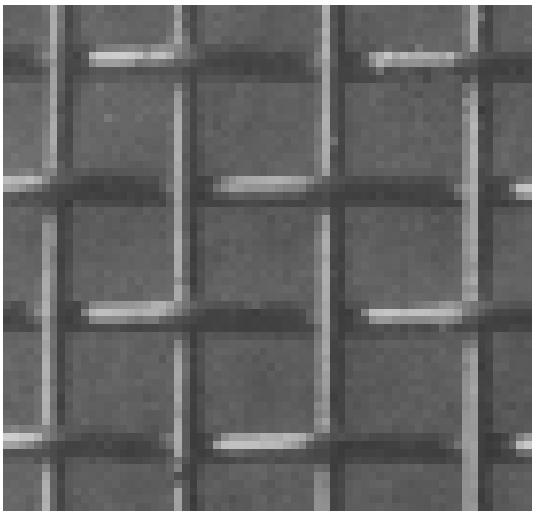
I think the `window_size` wasn't quite big enough for this:

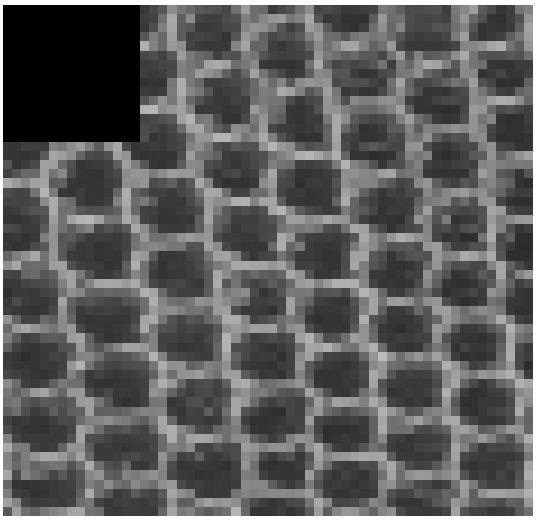


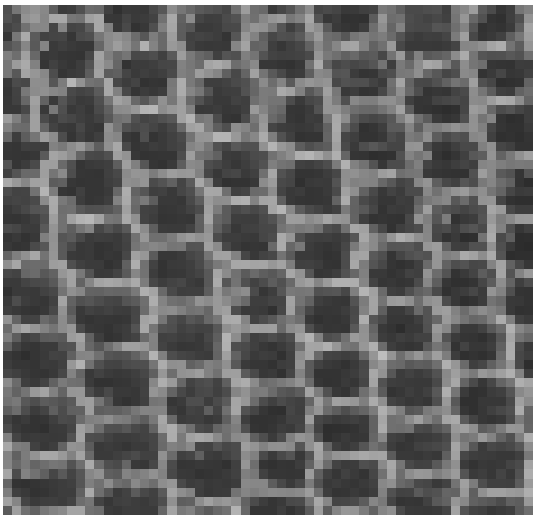
These were done with the final version:



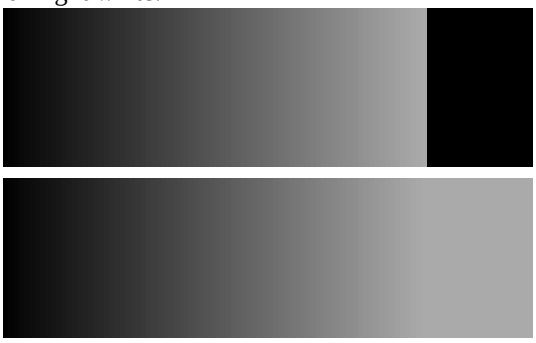








All pixel values must be taken from somewhere in the image, so the algorithm can't continue this gradient, and instead it fills it with a solid gray rather than gradualy making it white.



4. Summary and Conclusions

With a few minor changes, Efros and Leung's algorithm worked quite well. It was also fairly easy to implement, but extremely slow in Matlab. Filling in the picture of Efros and Leung took several hours.

If I were to do it again, I would probably write it in either Python, C, or maybe even ML.

References

[1] Alexei A. Efros and Thomas K. Leung, "Texture Synthesis by Non-parametric Sampling," *IEEE International Conference on Computer Vision*, 1999.

[2] Alexei A. Efros and Thomas K. Leung, "Algorithm details," `http://www.cs.berkeley.edu/ efros/research/NPS/alg.html`

[3] Alexei A. Efros, "Texture Synthesis," `http://www.cs.berkeley.edu/ efros/research/synthesis.html`

Ivan is a sophomore in the Bachelor program in Computer Science in the School of Computer Science at Carnegie Mellon University. He loves to increase entropy.

Common Image Set Compression

Sylvain Paillard
Computer Science
(Junior, Exchange Student)
paillard@sylvain.com

Abstract

The goal of the common image set compression is to save only once 'blocks' (regions) of an image that appear several times in the image, as well as in the other images of an image set. The prosess can be separeted in three steps: find the common blocks, remove blocks from a picture (compression) while saving the positions, replace blocks in a compressed image given a list of blocks and a list of positions. An algorithm to find the block is suggested: it takes small parts of the image randomly and looks for the block in these parts. An algorithm to remove the blocks is suggested: it removes the blocks by shifting the pixels in order to have a smaller number of lines at the end. Results are presented showing the different qualities and compressions, depending on the size of the blocks, and the maximum error allowed to consider whether the blocks are similar.

1. Introduction

Compression is directly connected with the information theory. We can even easily calculate the minimal possible size of most common information (texts, signals, …). Compressing, whether using the the meaning of the information as a compression parameter or not, doesn't change the fact that the compression will be related with the information that the document contains. Compression and understanding the information are, ultimately, two different ways of looking at the same thing.

Indeed, compressing an image using a symbolic approach (which would, for example, save a tree as a symbol with parameters and not as a group of pixels) is not so far in its meaning as the present zone compression like JPEG. Of course, in the way of working, they would be completely different, but the final result is always the same: try to compress an image without losing the information inside.

The idea of this project is to try to make a very small step in this direction by making the following assumption: if we can save the common parts of a set of images once and for all, we will have a compression that is trying more directly to find the common features of these images, and therefore going in the direction of a symbolic compression (even if no expressable knowledge is used or found, because this approach is completely non-parametric).

Due to the complexity of this project and the small amount of time I had, the actual compression I will present in the following section is not so much the information common in a set of images, but the information redundant in a single image, which is also redundant in the other images of the set.

However, this approach is still trying to find common information in the set and, as I explain it in the appendix, this system could be used for image discrimination.

The following sections present the system I used to find the blocks, how I compressed and decompressed images and what different results are found by varying the parameters.

2. Methods

2.1. The image set

What is a good image set for this system? In fact, all kinds of sets can be used, but the more the image differs, the more the results are going to be uninteresting, and the more the algorithm is going to be slow. The image sets I used are extremely similar, but a less similar set would still give good results.

Here is two example images of the set I used :

Reference : Image1

Reference: Image2

2.2. Finding Blocks

As in many search algorithms, a very easy, but completely unrealistic, approach exists :

N = number of pixels
For all blocks Ni in N
For all blocks Nj in N
if Ni and Nj are similar and i!=j
add the block to the set

However, this algorithm is in the order of $O(n^2)$, which overpasses 1'000 billion comparasions for a single image, so it's not realistic to use it.

A good alternative is to take small random subset of the image (for example 100x100 pixels subsets) and run the algorithm on these parts.

This is the method I used and it gave quite good results.

The two majors inconveniences of using this method are:

- We have no guarantees of finding all the possible blocks (for example, a block that occurs twice in an image, once in the left-top and once in the right-bottom, will never be found)
- We can have a very big redundancy of blocks. If we are not using an exact similitude comparison for the blocks (as explained in the next part), we have to filter these blocks if we want to get rid of the many redundancies, because it will cause a big slow down where we are trying to find positions for the blocks in a given image of the set.

2.3. Similar blocks

How to decide if two blocks are similar?

Lossless:

The lossless and easiest way is to consider that they have to be exactly the same. The problem with this method is that the probability of finding blocks that match, even very small blocks, is very small. For example, for a 50'000 pixel grayscale picture, I only found 10 3x3 pixel blocks that matched.

Lossy with maximal error:

Another way to do it is to use a maximal error parameter. The idea is simple: if two blocks have an error smaller than this maximal error, they are considered similar.

There are different ways of calculating this error; I chose to take the sum of the square error of each pixel difference:

Error = 0
For all pixels a_i and b_i of the blocks A and B
Error += $(a_i-b_i)^2$

I used the square of the pixel difference instead of the absolute value of the pixel difference, because I wanted to emphasis big pixel differences. Indeed, a lot of small difference (1 or 2 over the 256 different possible grayscale values) is much less noticable than a few big errors.

I kept the maximal error as a global parameter to be able to modify it, depending on the size of the blocks and the final image quality, but a good value in order to have a good quality is around 25 (for square of two pixel difference)

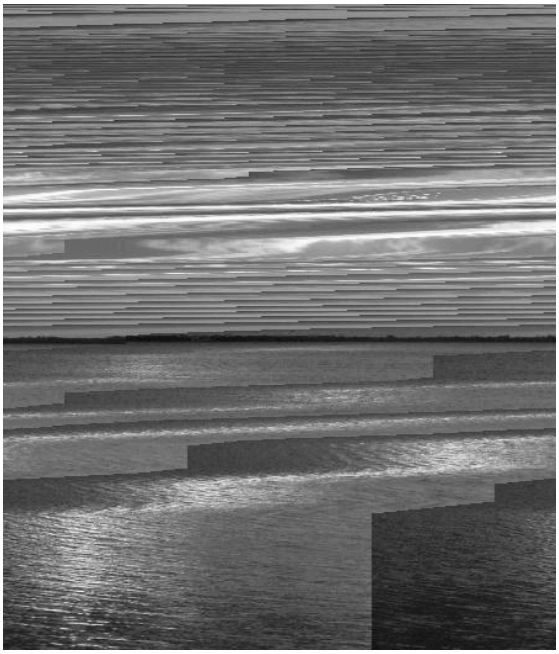
2.4. Remove the blocks

The algorithm to find the blocks is pretty straight forward: it looks through all the possible blocks of the picture and compares them.

There are different ways of dealing with the removal of the blocks:

Keeping a good zone compression:

In order to keep a good zone compression result (using a JPEG compression, for example), it is good to give the mean value as the value of every pixel of the zone. Zone compression algorithms will then be able to compress it very well.



Get a smaller image:

In order to obtain a smaller image, the blocks have to be removed. There are several ways of doing it (take blocks at the end of the pictures, shift the pixels, etc.). I chose to shift the pixels from the right-bottom of the image to the left-top. A 10% compressed image looks then like this:
Reference: Image3

NB: This image it the result of the compression of Image1, compressed with 10x10 blocks with an maximal error of 1'000. We can see that the system found many more blocks in the sky region than in the water region (because of its bigger complexity).

I used the second method (get a smaller image), because it was easier to obtain proportional statistics. Indeed, the JPEG compression works in a surprising way, and was not giving a result proportional to what I wanted to emphasize. However, this would probably be a wrong implementation for an application in the real world, because the JPEG compression gives very bad results with images like *Image3* (because of the way the zone compression algorithms work).

2.5. Positions overlapping

When an image is being compressed, should the overlapping of blocks be allowed or not?

All the results of the next section are given with disjointed positions, which means the blocks defined by the positions on the picture don't overlap. There are several reasons to avoid overlapping:

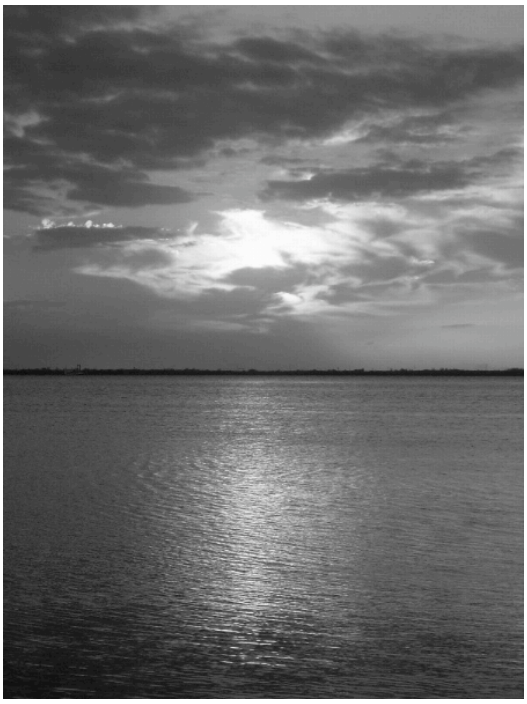
- If a subset of the blocks are ultimately used – which has always been the case – this subset will be bigger, with almost no advantages, if we allow overlapping.
- It's much more difficult to create a shifting function, because you can't sum the pixels defined by blocks to determine the amount of shifting for each pixel.
- The decompressing time becomes fast for a very small compression progression.
- After a certain number of overlapping, the size necessary to keep the positions is going to become bigger than the initial pixels saving size (which is something that really has to be avoided).

However, overlapping could be good sometimes, for example, if two blocks are overlapping by just a few pixels (the corners). Ideally, a complex algorithm should be use to decide for all possible configurations if additional overlapping should be used or not. It could even remove old positions if new overlapping positions give better compression results, because there is less overlapping.

3. Results

I first tried my method with the 10x10 pixel blocks and a maximal error of 1'000. I obtained the blocks from *Image1*.

The decompressed image of the compressed image of *Image1* looks exactly like the original:



Reference: Image4

222 disjointed positions have been found, which gives a compression of 8%.

On *Image2*, 230 disjointed positions have been found, which shows that for a very similar set, the training image doesn't matter. If the set was more diverse, obtaining the blocks from different images that would be representative of the complete set would have been necessary.

N.B.: The number of blocks in the subsets for all the results of this section varies between 50 and 200, depending on the case. The reason for these variations is explained at the end of this section.

The result for 20x20 pixel blocks with 10'000 for the maximal error results in 72 disjointed positions, so a compression of 11%. The result still looks exactly like the original picture:

Reference: Image5

On *Image2*, 52 disjointed positions have been found, so a compression of 8%. This growing difference between the training image and the test images was predictable. Indeed, even if the images are very similar, they are still different enough to give worse results if the blocks are bigger.

Then I tried to augment the maximal error to see when the lossy-ness started to become noticable. I used the following picture (which a subset of *Image1*) .

Reference: Image6

It turned out that the first maximal error that was starting to suffer noticable compression for 10x10 pixel blocks was around 10’000, as illustrated by the following picture:

Reference: Image7

The number of disjointed positions in *Image7* is 100. Which gives a compression of 30%.

Still using *Image6*, using 20x20 pixel blocks with a maximal error of 200’000 gives even lossier results. The number of disjointed positions is 20, which gives a compression of 15%.

Reference: Image8

A very important point in these results is that, even if the compression results are connected with the size of the blocks and the maximal errors, they are not only dependent on each other.

The compression also depends strongly on the following factors :

- The size of the random subsets of the image used to find the block
- The number of blocks in the block set
- The difference (maximal sum of square pixels error) between the blocks of the set.

The reason for these other changes was to avoid very big computational time variations.

For example, using more than 200 blocks for 10x10 pixel blocks and an error of 10’000 found thousands and thousands of positions. I never let the algorithm run until the end, because the necessary time for compressing the image was around 15 hours.

For compression algorithm, three factors count :

- Compression quality
- Compression size
- Compression / Decompression time

Therefore, the time should still be considered, especially for the compression and decompression time (even if the required time for finding the blocks also matters).

4. Conclusion

Reached:

- Found a reasonably good algorithm that finds most of the possible blocks
- Created the compression and decompression algorithm using a pixel-shifting method.
- Succeeded to reach 30% of compression
- Found reasonable parameters for compression.

Possible improvements:

- Create a better algorithm to find the best possible blocks, or a better approximation of the best possible blocks.
- Have different size, shape and orientation blocks in order to first have more compression with a smaller block set.
- Reach a much better compression ratio (I think 80% would be possible in a very similar set, such as the one I used).
- Find the function that describes the time, the compression and the quality of the system with all the possible variations of the parameters (size of the blocks, maximal errors, time necessary for finding the blocks, compression and decompression) to find all the interesting maxima.

The results found in this paper are promising, but I have to admit they are the first step in a very long research program. A very interesting point is that I succeeded to obtain a 30% compression for a small image set during a 40-hour project. This result shows that this subject has a lot of potential ,and should definitely be continued.

It seems that the project is far from a non-parametrical, which would lead to a better information understanding, and I have to admit that it is true.

However, this last goal is supremely difficult to reach, and even if this project was developed to reach all its potential, it would still be a very small step in the understanding of the information.

The compression and its connection to the information – and to artificial intelligence by the same way – is very fascinating subject and I will definitely try to continue to follow this way. I hope some discoveries are going to be made in this direction soon, which will allow computer science to reach a new level with possibilities that would seem unrealistic now.

In the next and last section, you will see that the compression described in this paper can already be used as an understanding of information, which gives some hope about the future of similar compression methods.

5. Appendix: Image discrimination

The idea of this last section is to use the compression ratio as a criterion to decide whether or not a new image belongs to the initial image set.

As explained in the results section, with 10x10 pixel blocks and an error of 1'000, 222 disjointed positions are found for *Image1* and 230 for *Image3*.

No positions are found in the following image :

This image is obviously not part of the image set and the number of positions found gives a very correlated result. Therefore, another possible use of this compression system – maybe with very different optimal compression parameters – would be image set discrimation.

Sylvain Paillard is from Switzerland and is an exchange student at CMU for his Junior year. He is pursuing a diploma at the EPFL in Lausanne (Switzerland).

Texture Characterization

Stephen Roos
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
smroos@andrew.cmu.edu

Sarah Schipul
Dept. of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213
ses@andrew.cmu.edu

Abstract

The ability to characterize textures plays an important role in human visual perception. With this knowledge we can describe, compare, and understand different textures. The computational equivalent of this ability would aid computer vision pursuits profoundly. In this study we examine different methods of translating human characterizations into computational dimensions. We examine the mathematical statistics of the images, and manipulate them with several dimensional reduction techniques. Our results show that it is possible to find computational patterns within image statistics that correlate to characterizations that humans make about the same texture. Future work in this area can lead to a very useful computational texture characterization program.

1. Introduction

Understanding textures plays an important role in human visual perception. Through characterizing textures, we can describe textures to others, compare different textures, and make predictions about textures seen for the first time. Therefore, computer vision systems would be able to learn a lot of valuable information about their visual environment, if it were possible to create a program that could computationally characterize textures in a manner equivalent to the way that people do. The goal of this study is to devise such a program.

Previous work has been done to create a texture naming and classification system [1]. But no such studies have attempted to translate these classifications into computational dimensions, to be implemented in a program. We think it is possible to find correlations to these characteristics in the mathematical statistics of the image matrix. Our goal is to analyze these statistics to find correlations to each of the characteristic terms that we have chosen. We can then use these correlations to create a program that can take a given image and rate it on the characteristics in the same way a human subject would.

2. Methods

2.1. Choosing characterization terms

First, we needed to create a list of terms to describe the different aspects of the textures. We wanted our terms to be independent of each other as much as possible. We also wanted to choose terms that could describe textures out of context. We first considered the work of Rao and Lohse [1]. In their study on identifying relevant dimensions of texture, they used a list of 12 characteristic terms for describing textures. We modified this list to meet the criteria listed above. We ended up with 11 characteristic terms. Each term would be rated for an image on a scale from 0 to 9. Our list of terms can be found in Table 1. The following is a brief description of our terms:

Contrast is a rating of how sharp the edges in the image are. Repetitive vs. Non-repetitive rates what percentage of the image is a repetition of other parts of the image. Structured vs. Random rates the randomness of the distribution of the elements in the image. Directional vs. Non-directional rates whether or not there is a clear direction of the distribution of the elements. Granular vs. Non-granular rates whether or not the texture contains an average shape (which may be thought of as a grain) which is distributed throughout the image. Coarse vs. Fine rates the

relative size of the repeating elements. Regular vs. Irregular rates the consistency in size and shape among the repeating elements. Uniform vs. Non-uniform rates the consistency in color among the repeating elements. Structural complexity measures the degree to which the image adheres to a fixed pattern of organization. Shiny vs. Dull rates degree to which the elements emit specular lighting (as opposed to diffuse lighting). Rough vs. Smooth rates the variation in depth on the texture surface.

2.2. Creating the image database

Then, we needed to create an image database. We did a simple web search to find 100 texture images. In creating this database, we tried to meet several criteria. First, we wanted to represent a broad range of textures. We would not use more than 3 images of the same texture category. Secondly, the images all are relatively easy to characterize using the terms we have chosen. Also, the images should represent a wide range of each of our chosen characteristics.

Table 1. Characteristic terms for describing textures

Low Contrast	0 1 2 3 4 5 6 7 8 9	High Contrast
Non-repetitive	0 1 2 3 4 5 6 7 8 9	Repetitive
Structured	0 1 2 3 4 5 6 7 8 9	Random
Non-directional	0 1 2 3 4 5 6 7 8 9	Directional
Non-granular	0 1 2 3 4 5 6 7 8 9	Granular
Fine	0 1 2 3 4 5 6 7 8 9	Coarse
Irregular	0 1 2 3 4 5 6 7 8 9	Regular
Non-uniform	0 1 2 3 4 5 6 7 8 9	Uniform
Low structural complexity	0 1 2 3 4 5 6 7 8 9	High structural complexity
Dull	0 1 2 3 4 5 6 7 8 9	Shiny
Smooth	0 1 2 3 4 5 6 7 8 9	Rough

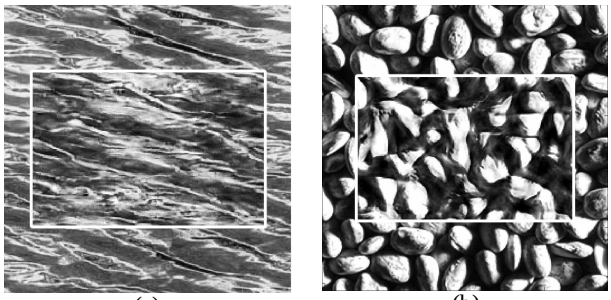
Once we had our image database and our characterization terms, we needed to create human ratings for the images. Because of time constraints and the difficulty of conducting a psychological survey, we decided not to do a survey of random subjects. Instead we simply rated the images ourselves. Because we had 100 images, any personal errors in judgment would be insignificant. Therefore, we went through and rated each texture image on the 0 to 9 scale for each term.

2.3. Analyzing the image database

This database of images and the table of our ratings provided our input data. First, we wanted to statistically analyze our images. Our analysis tool first decomposes input textures into multiple scale and orientation bands using a steerable pyramid implementation based on the work of Karasaridis and Simoncelli [2]. Then, we calculate various statistics on each of these bands and their relationship to each other, based on the suggestions of Portilla and Simoncelli [3]. The analysis tool collects statistics such as texture mean, variance, skew, kurtosis, range, as well as multi-scale and multi-orientation information including cross-correlation and auto-correlation. We also incorporated other statistics, which include histogram analysis of wavelet responses, Fourier transforms, and fractal dimensionality. After performing all of the above statistics, we had 1785 values for each texture.

2.4. Verification through synthesis

Our hypothesis was dependent on the fact that these statistics would contain some representation of the characteristic information. To verify this assumption we created a synthesis package which would recreate a texture using only the statistics for that image. We implemented the synthesis tool described by Portilla and Simoncelli [3]. This program does not take any samples from the original image. Two examples of an original texture image and its synthesized image are in Figure 1. As you can see, the water image in Figures 1 (a) is synthesized very well. Few, if any, differences can be seen. Therefore, all of the characteristics as described by our terms remain preserved. Figure 1 (b) is not a perfect synthesis, because some of the individual nuts are morphed together. Nevertheless, all of the characteristics judged by our terms are preserved. Thus, we proved that our assumption was true. The statistics did contain enough information to predict the characteristic ratings.



(a) (b)

Figure 1. Two examples of synthesized textures. In each image, the outer image is the original texture, and the inner box is the synthesized texture.

2.5. Finding correlations between characterization terms and the statistical data

Once we had a large number of statistics for each texture, we needed to find correlations between that information and our rating scales for the 11 characterization terms. To do this we needed to compare one set of statistics for all 100 images to our ratings on all 100 images. For example, consider we are looking at the values for mean. We have a 1x100 matrix of double values, one value for each image. First we needed to translate these values onto a 0 through 9 scale. To do this, we divided the data into 10 bin histogram. Then, based on which bin a given stat fell into, we changed that stat into a ranking between 0 and 9. We also created a second version of each stat in which the values are, instead, ranked from 9 to 0. By using both matrices, we could look for both a positive and negative relationship with the ratings data. Therefore, our hypothesis is based on the assumption that any statistical data will relate to our ratings data in a linear way. It would be too tedious and time-consuming for us to try to re-rank the data in non-linear ways, in order to find the best fit. So we assume that the relationship will be linear. Once we reformatted the values in this way, we compared them to our ratings values, one characterization term at a time. To compare them we found the difference between the two matrices, and calculated the mean and the standard deviation of the difference across the 100 samples. We added the standard deviation to the mean to get an idea of the range in which the predicted values would fall. We also created a variable called quality, which calculates the accuracy, such that 0% corresponds to random guessing and 100% corresponds to an exact match with our personal ratings. We are looking for the statistics that match our human data with the lowest mean plus standard deviation and the highest quality. This is the method that we use to compare all of our possible correlated rankings to our human characterization rankings.

2.5.1. Marginal statistics. Our first method to find a correlation was to look directly at the marginal statistics. We used our intuition to examine the statistics which we thought would fit the data. The ones that we looked at directly were mean, variance, standard deviation, skew, kurtosis, and range. Some of our predictions and findings follow. Variance looked like a good estimate of contrast for natural images, because sharp lines are defined by large differences in color. The same is true for standard deviation. Skew should be a good prediction for dimensionality, because non-skewed elements will directly follow a straight line. Kurtosis was recommended by other sources as a good estimate of coarseness.

2.5.2. PCA and ICA. Principal Component Analysis (PCA) and Independent Component Analysis (ICA) are two methods that are commonly used for dimensional reduction and simplification. For both of these calculations, we used the FastICA package by Gavert, Hurri, Särelä, and Hyvärien [4]. Initially, we ran PCA and ICA on the images themselves; however we immediately saw that this provided spurious results. Therefore, we instead ran both programs on our collection of statistics for the images. In this way we were hoping to find certain trends or dimensions within the set of statistics, which might correspond to our characteristic terms. PCA reduces the number of dimensions of the input data, by finding the most pronounced axes of variation within the sample space. In slight contrast, ICA reduces the dimensionality by requiring that the components it finds will be mutually independent. After running PCA and ICA on the statistical data, we found some interesting results. Again, PCA generated spurious results, probably because of the diversity of the data. However, ICA was able to find several patterns within the data. The program outputs a given number of sources, which each represent a combination of many different statistics from the set we gave it, reduced into one dimension. It also ranks these sources according to how much they are represented in the original set of data. In order to see what kind of components it isolated, we ran the synthesis program on the top 5 sources to create a new texture. Two of the synthesized textures can be seen in Figure 2. We found the synthesized textures to consist of interesting lines and edges. This may be an artifact caused by mismatched stats (stats which couldn't possibly be generated by a real input image); however, it seems that the importance of multi-scale, multi-orientation edges may, in fact, be a principal source of variation in textures.

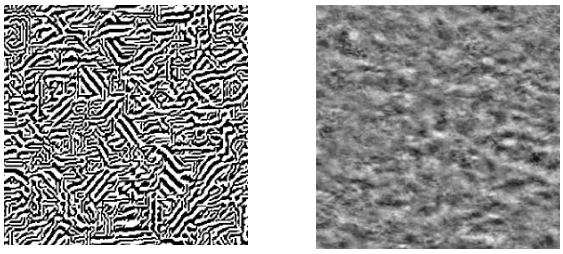


Figure 2. The textures synthesized from two of the independent sources found by ICA.

2.5.3. Local Linear Embedding. Another method of reducing the number of dimensions in a data set is local linear embedding [5]. This algorithm relies on the property of local linearity of continuous curves to

find linear mappings (or embeddings) of input data that may be represented in a complex, non-linear way. For each input data point, the algorithm first finds the K nearest neighbors of that data point. Then it ranks the imaginary lines between the center and the neighboring points based on how collinear they are with other points. In this way, curves with relatively high linear covariance within a given K-nearest neighbor local window are reassigned to a linear space of reduced dimensionality, regardless of the relative complexity of their innate relationship. We ran LLE on the statistical output of our analysis program, and formed rankings from the results based on the dimensions that LLE produced.

2.5.4. Heuristics. Lastly, we used heuristics to find matching data for the terms for which we have not yet found a good match. First, we wanted to improve directionality. To do this, we can look at a directional wavelet response histogram. If a certain bin happens to have significantly more elements than the other bins, then the image is directional. We simply calculated the difference between the number of elements in the most populated bin with the number of elements in the next highest populated bin. Second, we wanted to improve upon repetition. We can improve repetition by doing a Fourier analysis on the texture. This will capture the spatial frequency of a repeating element. This frequency is how repetitive the texture is. Thirdly, we want to improve upon coarseness. Coarseness is often measured by something called the Fractal Dimension of an image. This calculation returns a value between 1 and 2. The value 1 means that the texture is not fractal, and it doesn't contain significant self-similarity. If the value is 2, it means that the texture is very fractal, and is very self-similar, and thus, very coarse. Finally, we need to improve upon Randomness. However, it can have a negative relationship with structural complexity.

2.6. Rating a given test image

Finally, we have a general program that will take in a test image, use the training set of images to create all of the above statistics, and use those statistics to output ratings for each of our 11 dimensions for that image. The program first runs analysis on the 100 training images and the test image. It then performs all of the above analyses on the training set only and chooses the method with the smallest error rate for each descriptive term. It then outputs the corresponding information on the test image for each descriptive term.

3. Results

Our results were fairly impressive. By combining our different methods we were able to result in a prediction model that was fairly accurate.

3.1. Marginal Statistics

The results from our marginal statistics can be seen in Appendix A. Because we only explored the 7 statistics we mentioned above (mean, variance, standard deviation, skew, kurtosis, and range), we did not find much correlation. First of all, mean tells us nothing useful. Although we hoped variance would be a predictor for the sharp edges in contrast, this turns out not to be true because it does not look at the edges at all. For example a slow gradient from black to white would have a high variance, but would not have high contrast.

The same is true for standard deviation. Skew does give us a good prediction for directionality, just as we had hoped. Even though kurtosis was recommended for coarseness, it does not work. This may be because they match up in a non-linear way, which our method does not account for. Finally, range, was not useful at all. Yet, none of these were the best predictor for any category. Therefore, they are not used in the final function.

3.2. PCA and ICA

As I mentioned before, we did not find any useful information using PCA. Nevertheless, ICA provided good results, which can be found in Appendix A. The final function uses ICA for granularity (36% quality), uniformity (47%), and shininess (37%).

3.3. Local linear embedding

Local linear embedding gave us very good results, as can be seen in Appendix A. The final program uses LLE to determine contrast (52%), regularity (39%), complexity (51%), and roughness (40%). As you can see, it had very good quality ratings for the characteristics that it is used for.

3.4. Heuristics

Our method of heuristics worked very well to fill in for those characteristics for which we could not get good results with the other methods. Our heuristic results can be found in Appendix A. Heuristics were

used for repetitiveness (49%), randomness (52%), directionality (66%), and coarseness (49%).

3.5. Final program

Our results for our final program can also be found in Appendix A. By choosing the best method for each characterization term, the program did very well, as our average quality rating was 45%.

4. Discussion

We showed that it is possible to create mathematical representations of human texture characterizations. First, when we showed that our synthesis program can synthesize textures based on statistical information alone, we knew that the characterizations must therefore be represented in those statistics somehow. Although our quality rating was around 50% for most of our characteristics, we were still very successful. These values are significant enough to prove that it is possible for a computer to computationally characterize a variety of texture images in the same ways that humans can. Some ways that the methods may be improved upon in the future is to first look for non-linear matches to the statistics. Also, different uses of PCA/ICA and LLE could be used to get better results.

References

[1] A.R. Rao and G.L. Lohse, "Towards a Texture Naming System: Identifying Relevant Dimensions of Texture", *Vision Res.*, 36(11), 1996, pp. 1649-1669.

[2] A Karasaridis and E Simoncelli, "A Filter Design Technique for Steerable Pyramid Image Transforms", *Int'l Conf. Acoustics Speech and Signal Processing*, Atlanta GA, May 1996.

[3] J Portilla and E P Simoncelli, "A Parametric Texture Model based on Joint Statistics of Complex Wavelet Coefficients", *Int'l Journal of Computer Vision*, 40(1), October, 2000, pp. 49-71.

[4] H. Gavert, J. Hurri, J. Särelä, and A. Hyvärien. "FastICA for Matlab 5.x", 2001.

[5] S. Roweis and L. Saul, "Nonlinear dimensionality reduction by locally linear embedding", *Science*, 290 (5500), Dec.22, 2000, pp.2323--2326.

Stephen Roos

Stephen is from Los Angeles, California. He is currently a Junior in CIT. He is pursuing a double major in Electrical and Computer Engineering and Biomedical Engineering.

Sarah Schipul

Sarah is from Watertown, Connecticut. She will be graduating this May with a B.S. degree in Cognitive Science with a minor in Computer Science. In June she is going to start working full time as a Research Associate at the Center for Cognitive Brain Imaging at CMU.

MARGINAL STATISTICS

Characterization	Difference	Quality
Contrast	3.62	38%
Repetitiveness	4.16	29%
Randomness	3.98	32%
Directionality	4.90	16%
Granularity	4.24	27%
Coarseness	4.68	20%
Regularity	4.29	26%
Uniformity	3.81	35%
Complexity	3.29	44%
Shininess	4.09	30%
Roughness	*4.13*	*29%*
Average	4.11	30%

ICA

Characterization	Difference	Quality
Contrast	2.96	49%
Repetitiveness	3.67	37%
Randomness	3.75	36%
Directionality	4.24	27%
Granularity	**3.71**	**36%**
Coarseness	3.80	35%
Regularity	3.89	33%
Uniformity	**3.07**	**47%**
Complexity	3.14	46%
Shininess	**3.67**	**37%**
Roughness	3.60	38%
Average	**3.59**	**38%**

LOCAL LINEAR EMBEDDING

Characterization	Difference	Quality
Contrast	**2.80**	**52%**
Repetitiveness	3.78	35%
Randomness	3.57	39%
Directionality	4.27	27%
Granularity	3.93	33%
Coarseness	4.10	30%
Regularity	**3.58**	**39%**
Uniformity	3.23	45%
Complexity	**2.86**	**51%**
Shininess	3.77	35%
Roughness	**3.52**	**40%**
Average	3.58	39%

HEURISTICS

Characterization	Difference	Quality
Contrast		
Repetitiveness	**2.95**	**49%**
Randomness	**2.77**	**52%**
Directionality	**2.01**	**66%**
Granularity		
Coarseness	**2.95**	**49%**
Regularity		
Uniformity		
Complexity		
Shininess		
Roughness		
Average	2.67	54%

FINAL RESULTS

Characterization	Difference	Quality
Contrast	2.8	52%
Repetitiveness	2.95	49%
Randomness	2.77	52%
Directionality	2.01	66%
Granularity	3.71	36%
Coarseness	2.95	49%
Regularity	3.58	39%
Uniformity	3.07	47%
Complexity	2.86	51%
Shininess	3.67	37%
Roughness	3.52	40%
Average	3.12	45%

Appendix A. This table contains the results for all of our methods. Textures with difference and quality in bold indicates that that value is used in final results.

Single Image Stereogram Image Extraction Using SSD Disparity Measurement

Daniel Hershey
Carnegie Mellon University
djhershe@andrew.cmu.edu

Abstract

In this paper, I demonstrate the use of the SSD disparity detection algorithm to detect the depth information encoded in a single image stereogram, which is more popularly known as a "magic eye" image. With this method I am able to extract the outline of the image quite well, but the more detailed depth information inside the outline is not extracted clearly. The image is obtained by modifying the standard disparity measurement algorithm so that the same image is used for both the left and the right eye and the algorithm can only match positions to the right along the epipolar line. This modified algorithm is meant to reproduce the illusion seen by a person's eye.

I. INTRODUCTION

THIS paper analyzes single image stereograms (SISs) and single image random dot stereograms (SIRDSs). These images are designed such that when the viewer's eyes are focused beyond the image, the viewer can observe an image popping out of the page. This image can be very simply, such as a flat plane in some shape, or complex, such as a full three dimensional image. The difference between SIRDSs and SISs is that the background image is random dots in a SIRDS and the background is a repeating image in a SIS. The methods for viewing, constructing, and extracting images from both SISs and SIRDSs are the same.

The image observed in a SIS is the result of parallel viewing. Parallel viewing occurs when your eyes each see the same image, but the image is not the result of same object. When your mind notices the same image in each eye, it assumes that the same object caused both images, and calculates the perceived depth accordingly. If separate objects cause the images, your mind perceives them as a single object in the middle of the two objects and deeper than the two objects, closer to where the lines-of-sight from each eye would intersect.

The motivation for using a stereo depth perception algorithm is that the illusion occurs because of the assumptions the brain makes about stereo images. You cannot reproduce the illusion with a single eye. The algorithm I will be using[1] uses sum of square of the difference (SSD) of two windows to determine the points in each image, along the same epipolar lines, which correspond to each other and record the disparity. The depth is inversely proportional to the disparity, meaning that the greater the disparity the less the depth. The objective is to "fool" the SSD algorithm into observing the illusion properly by modifying it slightly.

The images used for this paper are the popular images created by the magic-eye corporation [1]. The information on how the images are created is taken from the SIRDS FAQ [3], and is based on the work done by Dr. Julesz and Christopher Tyler [2].

II. PARALLEL VIEWING

Parallel viewing occurs when the brain sees identical objects in each eye and assumes the images come from the same source. The brain then calculates the depth of the image as if both images came from the same source. If the two objects are closer together, the brain interprets them as having a shallower depth. The farther apart they are, the deeper the brain interprets them.

When your eyes are focused beyond the page, each eye is not focused on the same point on the page. If the image is a repeating pattern and the eyes are focused properly, then the brain sees a pattern, or series of patterns, in each eye and assumes that the pattern focused on by each eye is the same, when they are not. If a pixel is actually shifted slightly more to the center, it will appear shallower, and if it is shifted slightly away from center it will appear deeper. By creating a whole series of shifts in many patterns you can create a very complex three dimensional illusion which is only visible when the eyes are focused beyond the image.

A similar effect occurs when you focus in front of the image, or cross your eyes. This method usually produces a similar illusion, but the depth is reversed. This is because the brain interprets the location of the image at the point where the line of sight from each eye would intersect. Thus in parallel viewing this point is deeper than the actual image and in cross-eyed viewing this point is in front of the

[1] Daniel Hershey is a sophomore in the Department of Computer Science at Carnegie Mellon University. This paper was prepared as a course project for 15-385 Computer Vision

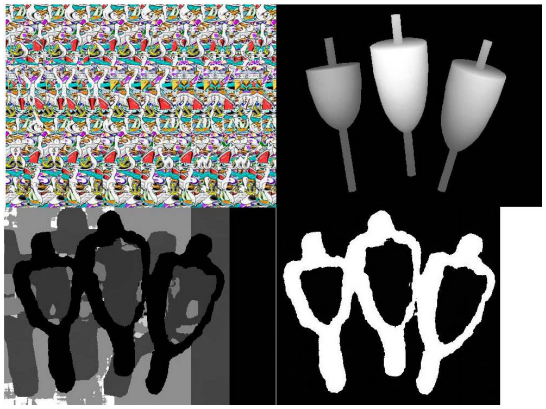


Fig 1. (Top Left) The orginal magic image and its solution (Top right). My algorithm first turns the original image into a disparity map (Bottom Left) and the inverts this map to get a clear image of the outline (Bottom right). Note that the bottom right image is the same size as all the others, just with the right-most column being white.

image.

III. Creating SISs

To create a SIS, you first create a repeating background pattern. This places everything at a default depth of 0. To create a point at depth 1, you add a pixel right before it. This increases the distance between images to the right of the shift and images to the left of the shift. Therefore the perceived depth will decrease, and the points will pop out. To decrease the depth of any pixel by one, you add a pixel right before it. To increase the depth of a pixel you delete a new pixel to its left. The analysis is the same for decreasing the depth. By repeating this process over and over you can create the depth information for every pixel in an image.

The same system can be applied to any image containing a repeating pattern. This offset system allows you to create any picture you wish in the illusion. Because of the difficulty of detecting the patterns in the image, or even the potential absence of a pattern to detect, it is very difficult to simply detect where the shifts in the image are. This is one reason why the stereo depth algorithm needs to be applied. Also, this is just a sketch of how the images are created. The Magic Eye Corporation probably adds more complexity to this algorithm, and detecting the changes to the pattern would require some knowledge of their specific algorithm. The approach of using stereo depth analysis to detect the illusion is probably more robust.

IV. Review of SSD

SSD is a method used for determining the disparity between two identical locations in stereo images. The first task in determining the depth of points in a stereo image is to determine the epipolar lines between the images. These are the lines in which everything in one image along the line

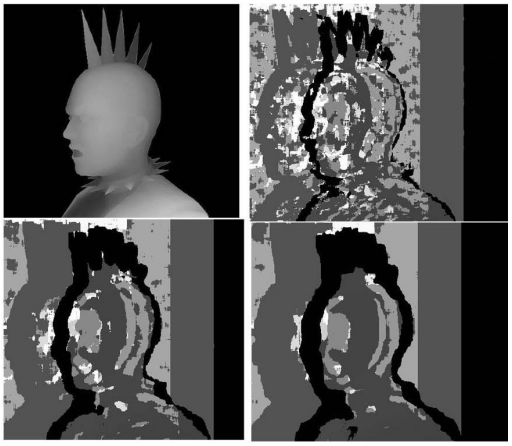


Fig 2. (Top Left) The ground truth of the depth map. (Top Right) The disparity map produced with window size 5. (Bottom Left) The disparity map produced with w=10. (Bottom Right) The disparity map produced with w=15. As the window size increase, the number of artifacts decreases and the detail decreases.

appears in the other image along the same line. These lines can be determined based on the locations of the cameras relative to each other.

SSD is then used to search along the line and determine which points correspond to each other in the image. Points are compared with each other by taking a window of a certain size around the point and comparing the sum of the square of the difference between the color values at each point. Larger window sizes result in more accurate measurement, which reduces artifacts, but loses more detailed information in the process. Smaller windows result in less accurate measurement and more artifacts, but also more detail. The window sizes cited in this paper are half of the length of a window's height and width (the windows are square).

The depth can be calculated from the disparity by the equation

$$\text{Depth} = f*B/d$$

where f is the focal length of the cameras, B is the distance separating the cameras, and d is the disparity. Thus a larger disparity indicates a smaller depth and a larger disparity indicates a deeper depth. This method is known to work fairly well at detecting the depth encoded by a pair of stereo images in a controlled environment. Thus it is appropriate for use in our application because we have a controlled environment (the images are static and well designed) and a pair of images that encode the depth (the images seen by the left and right eye).

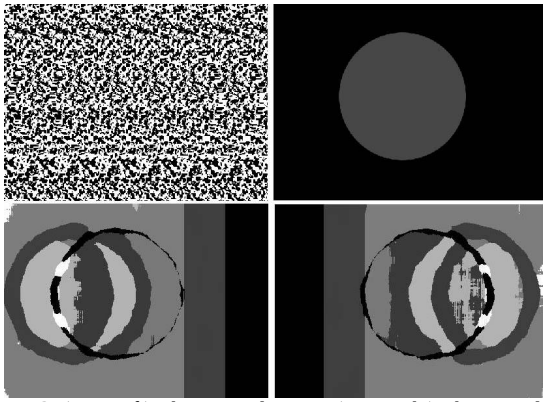


Fig 3. (Top Left) The original image. (Top Right) The ground truth for this image. (Bottom Left) The disparity map when searching to the right. (Bottom Right) The disparity map when searching to the left. All disparities in the bottom right image are negative and all disparities in the bottom left image are positive.

V. The Modified Algorithm

The standard SSD depth perception algorithm will not work in this case because there is only a single image. To solve this, I need to divide the image into two images, one for the left eye and one for the right eye. There is no reason to assume that each eye does not see the same image, so the same image will be used as both the left and the right stereo images. This introduces a problem because every image patch will latch onto itself during the search because the least variance achievable with SSD is the exact same image. An entire image comprised of zero disparity is not desirable.

This problem is addressed by forcing the matching patch to be to the right of the original patch. This is reasonable, as all the objects viewed by the right eye should be to the right. This introduces problems as well though because not all image patches have a corresponding patch to their right. In the actual illusion, this case is realized by the edges of the illusion being blurry and not well detailed. It is caused when the line of sight of one eye lies on the image, but the line of sight of the other eye lies off the image. This method will set a minimum disparity of 1.

The epipolar lines used in the image are horizontal lines because the image is generally close enough to the eyes that no distortion should be caused. This assumption can be checked by observing if the image looks slanted or skewed.

The windows used in the images will be tested with radii ranging from 5 to 15 pixels, which correspond to window dimensions of 10 to 30 pixels. Because of the complexity of the images, detailed information may be lost with the large window sizes, but a large number of artifacts may be inserted with the smaller window sizes.

The disparity information is stored at the left-most points location in the disparity map, instead of the midpoint as it might seem should happen, because each point is not a midpoint of two unique points. This could result in the depth information being shifted left of the location that is found in the real solution.

The returned image is the inverse of the disparity at each point. The constants f and B are not included in the calculation because what is most important is the relative depth of portions of the image. Because the image is an illusion, the literal depth has very little meaning, and thus it is not calculated.

VI. Results

By looking at the disparity measurements you can get the best understanding of how the depth is picked up. The outline of the image, caused by the shifts which produce the parallel viewing illusion, is picked up as a disparity of 1, while the interior information appears to be shifted to the left of the actual image.

The area of the images which is constant corresponds to the area which is not involved in any shifting, which is the background. The constant factor occurs because each point latches onto its counterpart in the next occurrence of the pattern, which is the same distance away for all points in the pattern. The last area of solid black occurs because there are no more occurrences of the pattern, so the closest match for each point is the window which contains the most information of the original point, which is the pixel one space to the right.

The outline of the image occurs because of the shifting points in the image. These points do not clearly correspond to any other points, so the algorithm chooses the pixel directly to the right as the closest match, hence the small disparity. This gives a very accurate outline of the image, down to the resolution allowed by the window size. This limitation on the resolution is because all points with a transition in their window will be a part of this effect.

The outline is also recorded in less extreme disparity measurements. These measurements are shifted to the left of the actual outline. These are likely caused by points which latch onto the shifts because they are close to the pattern. They might not find a better pattern to detect because of the large number of shifts. This information is less precise than the outline and there is no simple automatic method for moving the outline overtop of this information because you have overlapping issues. This extra information becomes irrelevant when you take the inverse because it is very large compared to the outline which has a disparity of 1.

When the inverse of the disparity is taken, the

large values obtained for most of the image go to zero, while the ones recorded for the outline and for the right most pattern remain one. This provides a very clear image of the outline of the image, but negates all aspects of the depth measurements. This is a decent way of viewing the image for the sole purpose of determining what the three dimensional image is. This also retains the problem of the rightmost pattern, but this can easily be cropped from the image if necessary. An algorithm could be applied to fill in the outline so that it appears to be a whole image, but that was not done in this paper.

Due to window size constraints not every image's outline is clearly depicted. This restriction produces poor results when used on images containing fine detail separated by small spaces of short depth. A smaller window could be used to determine the detail, but more than likely this window would introduce too many artifacts to clearly view the image.

In fact, window sizes of 5 and 10 yield images which have large numbers of artifacts. It is not until you reach a window size of 15 that you remove most artifacts. At a window size of 15, however, you do already begin to observe some effects of the finer detail being blurred out.

While these images were designed for use with parallel viewing, most of them can be viewed with cross-eyed viewing. This method of viewing can be modeled in our algorithm by searching to the left of the origin point instead of to the right. This produces an image which is a mirror reflection disparity map, except that the disparities are inversed. The inverse of the cross-eyed disparity map is the same as the outline of the prarllel disparity map except that the outline is -1 instead of 1. In this respect, the algorithm correctly models the solution because in cross-eyed viewing the disparity is the negative of its parallel viewing counterpart.

VII. CONCLUSIONS

It is clear that this algorithm does not detect all the information encoded by the magic eye images. It is, however able to accurately model the illusion in the sense of parallel versus cross-eyed viewing and it does produce the outline of the 3-dimensional images.

Possible future work on this topic would include improving the window size determination in order to reduce the information lost without introducing too many artifacts. A complete change of algorithms to a pattern detection algorithm might produce better results.

It is possible that the information encoded within the image is stored in a different manner than the one described, and more information on that topic would probably lead to better success in detecting the information.

VIII. REFRENCES

[1] http://www.magiceye.com
[2] Tyler & Chang, Vision Research, #17, 1977. Referenced by Tyler, 1983
[3] http://www.cs.waikato.ac.nz/~singlis/sirds.html

Daniel Hershey is a sophomore in the department of computer science at Carnegie Mellon University. He is currently taking the Computer Vision course (15-385) with Tai Sing lee and the Artificial Intelligence course (15-381) with Andrew Moore. His potential research interests include machine learning and applications of machine learning to computer vision.

Recovering Unseen Images: Seeing with the "Magic Eye"

Sean O'Loughlin
Carnegie Mellon University
CIT, ECE Sophomore
solough@andrew.cmu.edu

Abstract

This paper describes a technique which can recover an actual image from a "magic eye" image. A magic eye image is a single image that is capable of storing three-dimensional depth information. It stores this information in by separating two like pixels in such a way that when the viewer focuses past the plane of the image he is able to see depth. Since these separations only occur in the horizontal direction in "magic eye" images, this paper discusses a technique to analyze these images and find repeated patterns in order to recover some information about the hidden depth image.

Background

More commonly known as "magic eye" images, the set of images that encode depth by repeated patterns in the horizontal direction are referred to as digital stereograms or autostereograms. Stereograms were originally two images of the same thing, but each from a different angle or position. Because these two images were of the same thing, corresponding points on each image could be located and some information about the three-dimensional depth of the object could be inferred.

With the development of computers and other digital technology it then became possible to create single images to encode depth in a similar way to stereograms. Thus, the autostereogram was born. Autostereograms are similar to stereograms in that there are two points that correspond to each other. The difference is that the two points in the autostereogram are two pixels on the same horizontal line of the image. Because each pixel corresponds to another pixel with exactly the same value, most autostereograms display some form of repetition or pattern as the image is scanned from left to right or right to left, as seen in the following figure.

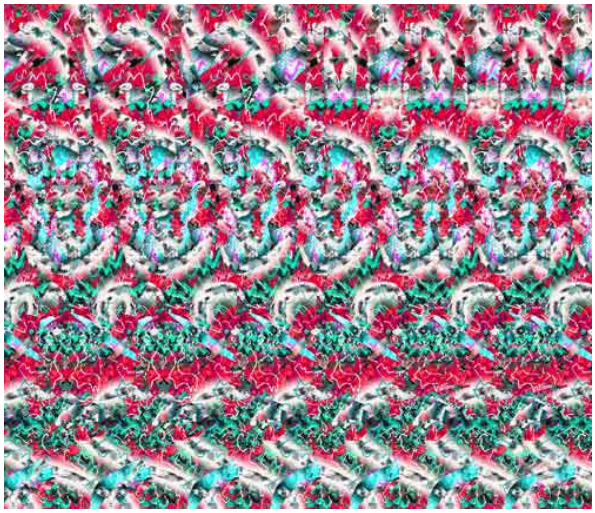


Figure 1. Example of an autostereogram

The method by which humans are capable of extracting the depth information is focusing on a point behind the image so that the corresponding pixels come together when the person is focusing at a certain depth behind the image. Figure 2, constructed from information given in [2], demonstrates how this works. The person's eyes are a certain distance apart and a certain distance away from the plane of the image. When he focuses to a point past the plane of the image, he sees one pixel with his right eye and one with his left eye. The distance between these pixels is known as stereo separation and corresponds to the depth that the combined viewing of the pixels appears to be at. As the figure shows, the three-dimensional image can have an arbitrary shape, yet can still be represented using an autostereogram.

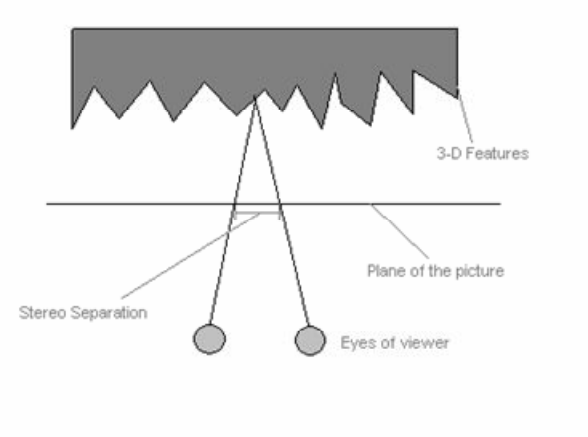


Figure 2. Diagram of stereo separation and viewing depth

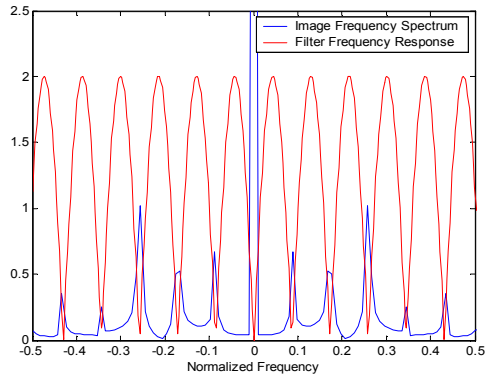


Figure 3. Filter Response and Frequency Response from [1]

Revealing Algorithm

I have termed my algorithm the revealing algorithm because through its execution it reveals some of the information hidden in a "magic eye" autostereogram. The key to the functionality of this algorithm is the horizontal repetition inherent in all autostereograms. By doing a Fourier Transform of the data we can analyze the repetitiveness of the image. A Fourier Transform is a mathematical tool that takes data in the time or space domain and transforms it into data in the frequency domain. That is why it is useful in analyzing repetitiveness. However as Curtis and Zwicker point out in their paper, using a Fourier Transform won't generate integers for discrete data, something that will be necessary for this algorithm's implementation since stereograms are discrete sets of data [1]. However, they point out that the fundamental period in the frequency domain corresponds to the stereo separation of the image. They illustrate this with graphs of frequency response and impulse response versus normalized frequency and it is clear there is a correlation between the two. I've reproduced their graph here as figure 3.

The usefulness of analyzing the frequency response of an autostereogram is that if we take a row of data that we know has pixels mostly at a certain depth, and find the frequency that pixels at this depth appear, we can remove pixels at this frequency and leave the information encoded at other depths. This could be useful for revealing the shape of the hidden image if we extract the frequency information for and remove the background. However, like I mentioned before, the Fourier Transform will not produce integers and we will be unable to remove the all of the information from a single depth. The solution is to use the stereo separation of the image. Since frequency and stereo separation are correlated, the stereo separation (an integer value) can be used to build a filter that will remove all the information from a certain depth. The way to build such a filter is to make the first entry 1, the last entry -1, and the entries in the middle a number of 0's equal to the stereo separation. For example, to build a filter from a stereo separation of 60 pixels, we would make a vector with the first entry a 1 followed by 60 entries of 0 and finally add a -1 as the sixty-second entry.

Once we have our filter built, we are ready to remove the pixels with the same stereo separation by convolving the filter with each row of the image. MATLAB has function that computes convolution but I have included the mathematical representation of convolution here in figure 4 which is described in [3].

$$y(n) = \sum_{i=0}^{n} h(i)x(n-i) = (h * x)(n)$$

$$y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots + h(n)x(0)$$

Figure 4. Convolution

Now we have sufficient information and tools to implement the revealing algorithm. First, we need to choose a row to be representative of the entire image, i.e. it must contain as many depths as possible and it must also contain the background depth. I chose the row exactly in the middle of the image because most of the hidden objects in "magic eye" images are one solid object. Next, we must retrieve the stereo separation information. To do this we start with the first pixel and then count the number of pixels until we find a match and store this number in an array. Then we continue doing this for all the pixels until we reach the end of the row. After that we remove all values from this array that are repeated and we have an array with all the stereo separations from that row. Now we are ready to start filtering.

When we have a filter, we want to convolve it row by row with the image to produce our results. Each row must be convolved with the same filter for consistency. When we convolve the filter and a line of the image, all the pixels at the depth corresponding to the stereo separation that the filter was built from will be equal to zero in the result. Therefore, to remove the background (the largest section of the image), we want to use each filter that we can build from our array of separations and find the one with the largest area of zeros. This will be the result with the background filtered out. In testing, I found that in most "magic eye" images the stereo separation of the background was a value between 80 and 120. Therefore, I restricted the building of filters to values that correspond to that range.

Results

The results turned out rather well. On all but a few "magic eye" images that I tested, I was able to extract the backgrounds and leave a shape corresponding to the three-dimensional hidden image. In most cases this figure is recognizable and you can determine what the object is. Figure 5 is the results of the algorithm applied to Figure 1. The hidden object is a punk rocker. You can clearly see the outline of his head and his spiked Mohawk haircut in the resultant image. The white and black bars on either side are artifacts from the filters and convolution. However, it is interesting to note that when a human views a magic eye image, these same areas are out of focus to him and thus are not important to recover. I've also included Figure 6, the solution to Figure 1 supplied by [6].

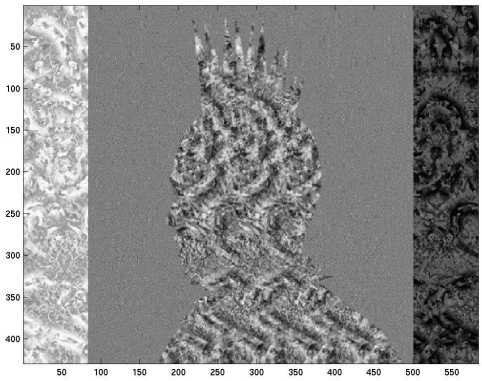


Figure 5. Result of Revealing Algorithm on Figure 1

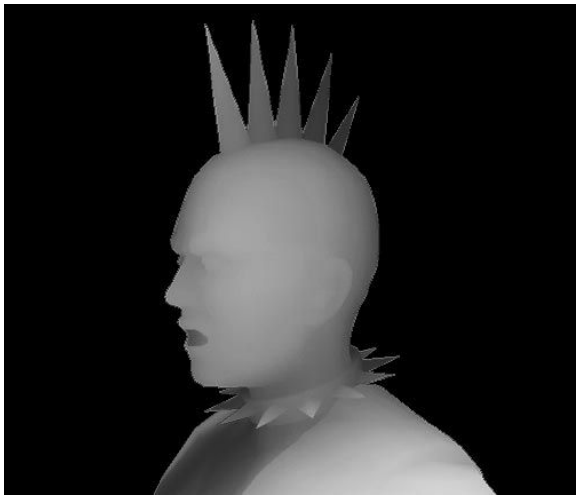


Figure 6. Solution to Figure 1 from [6]

Conclusion

While my algorithm does not recover a three-dimensional model of the hidden object of a "magic eye" image, it does a good job of recovering an outline

of the image and makes the hidden object somewhat recognizable. It also models human sight to some extent, since the black and white useless areas in the result correspond to areas that are out of focus to the human viewer. I find it fascinating that an algorithm based on simple mathematical principles can perform reasonably well at modeling something as complex as human vision and even emulate some human characteristics in the results it produces.

References

[1] Mike Curtis and Sarah Zwicker, "Digital Stereograms", http://faculty.olin.edu/~jcrisman/Teaching/SigSysWeb/Project/stereograms/DigitalStereograms.doc, December 8, 2003. Accessed April 5, 2004.

[2] Wei-Yang Lin, "A Minimal weighted Surface Algorithm for 3D Reconstruction", http://www.cae.wisc.edu/~wei-yanl/cs%20766%20report.pdf, December, 2003. Accessed April 5, 2004.

[3] Julius O. Smith III, "Introduction to Digital Filters", http://ccrma-www.stanford.edu/~jos/filters/, 2003. Accessed April 10, 2004.

[4] "Introduction to Computer Programming with MATLAB", http://www.phon.ucl.ac.uk/courses/spsci/matlab/, Accessed April 10, 2004.

[5] "Autostereograms", http://www.psi.utoronto.ca/~trausti/stereograms/stereograms.html, Accessed April 5, 2004.

[6] Magic Eye", http://www.magiceye.com/, Accessed April 5, 2004.

Biography

Sean O'Loughlin is a 19-year old from West Chester, Pennsylvania. He is currently a student at Carnegie Mellon University, where he is enrolled in the Carnegie Institute of Technology. He is studying Electrical and Computer Engineering. He is a brother of Sigma Nu Fraternity and a member of the CMU Ice Hockey Club.

Finding Waldo

Ilsun Lee and Laura Semesky
Carnegie Mellon University
School of Computer Science
151 N. Craig St. Pittsburgh PA 15213.
Email: ilsunl@andrew.cmu.edu
lsemesky@andrew.cmu.edu

Abstract
In this paper, we present a solution to the finding Waldo problem and discuss its effectiveness. To solve this problem we implemented variations on normalized correlation, PCA, ICA and SIFT algorithms. The only algorithms that yielded acceptable results were normalized correlation and SIFT, so these are the algorithms that are discussed extensively.

1. Introduction
The finding Waldo problem is a variation on classical image detection problems. The goal is to find the character Waldo hidden in cluttered images. Waldo is distinguishable by his red and white striped shirt and hat as well as his hair style and facial features. In order to find Waldo in the sample images confounds such as pose, occlusion, scale and image distracters need to be taken into account. Waldo is not posed the same in all of the images, is often occluded by other people as well as his own accessories and is not guaranteed to be the same size. In addition, the images are filled with a lot of red and white stripes in order to make finding Waldo more difficult.

2. Normalized Correlation
Correlation is the measure of the degree to which two variables agree. It does not necessarily compare the numerical values, but examines the overall trends in behavior. In the case of image correlation, each pixel in the template is compared to the pixels surrounding the corresponding pixel in the source. In the correlation method, the template image is moved pixel by pixel and compared to the output image. However, in this approach the borders are not necessarily checked because of differences in size between the template and source image.

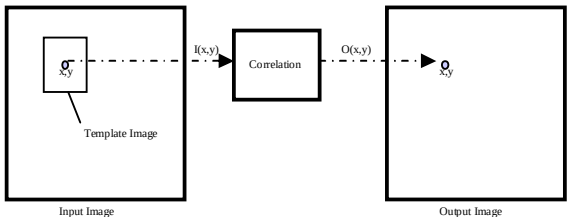


Figure 1.1 Correlation Method.

$$r = \frac{\sum_{i=0}^{N-1}(x_i - \bar{x})\cdot(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{N-1}(x_i - \bar{x})^2 \cdot \sum_{i=0}^{N-1}(y_i - \bar{y})^2}}$$

X is the template gray level image
X bar the average grey level in the template image
Y is the source image section
Y bar is the average grey level in the source image
N is the number of pixels in the section image (section image == template image size) columns * row
R is between –1 and +1, with larger values representing a stronger relationship between the two images.

2.1 Application of Normalized Correlation
We used a data set given to us by Professor Tai Sing Lee of Carnegie Mellon University. The set contains a template of Waldo and various other images which have a reduced Waldo somewhere in the image. In all of the non-template images, Waldo is varied in size and position and is often times occluded by clothing accessories or other objects in the scene. We made a simple program in Matlab to run normalized correlation on our test images. We used the built-in Matlab function corr2(image.file), which does simple correlation. The results were inconsistent, varying among the test images.

2.2 Result of Normalized Correlation


Figure 2.1 Waldo Template Image

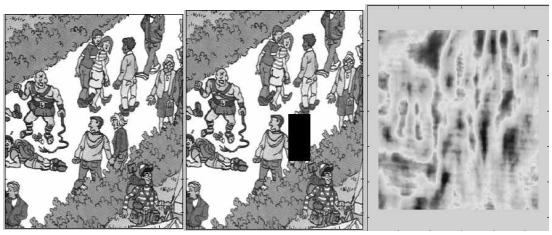


Figure 2.2 Result from Correlation
(Actual Image, Result Image, Distance Image)

The images in figure 2.2 are from a test where correlation did not work. The problem with using the above template image for correlation was that the white-space surrounding Waldo matched with white space in the image. Because the template white-space area was so large and Waldo is usually not surrounded by white space, this led to a lot of false matches. In this example, correlation found the woman with the similarly colored shirt because the white background matched the template. As we can see from the pattern in the distance image, correlation found many false matches along the white road. In the cases where he was in front of a white background, Waldo was detected reasonably well. In order to achieve more accurate results, we had to find a way of dealing with the white-space problem.

2.3 Modified Version of Correlation

To improve the template match, we made a simple modification and considered only certain interesting ranges of pixel values. First, we set the all the white background values to 0. We then multiplied the new template with the source image to produce a new source image that only had the part of the image that we were interested in.

Complicating matters, the corr2 Matlab function was very time consuming. It recalculated the average many times, had summations and used intense math functions such as the square root. To make things more efficient, we replaced the correlation function with a simpler equation:

$$r = (y - x)^2$$

Where y is the source image pixel and x is the template image pixel. Using this function, it is very simple to compute the difference of the sum. I normalized the value of r to be 0 to 1 by dividing by the maximum of r value from Modified Correlation.

2.4 Result from Modified Version of Correlation


Figure 3.1 Modified Template


Figure 3.2 Result from Modified Correlation
(Actual Image, Result Image, Distance Image)

Notice that template in Waldo back ground was set to 0 compared to the original template. In the above case, the exact same source was tested using my modified approach. As you can see, Waldo was found correctly in the picture. Also, we do not see the problem where the white road part was closely related to the Waldo template since we only considered the part where the Waldo should be. However, another problem arose: the new template had a very large portion of red values to consider in correlation case. Thus, if the picture contained a large amount of red, it identified those areas as strong matches.

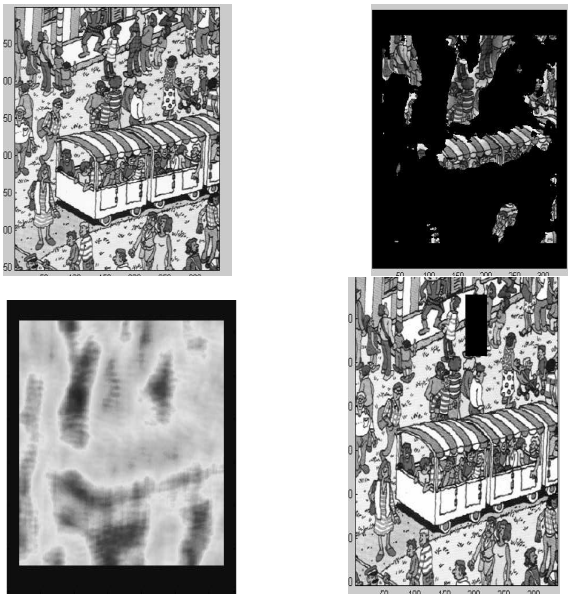


Figure 4. Result from Modified Correlation
(Actual Image, Threshold Image,
Distance Image, Result Image);

The images in figure 4 are from the case where the modified method failed to correctly identify Waldo. As you can see, it detected the area of the woman wearing the red full suit. However, in the distance image, we noticed that Waldo actually showed as a strong match. Thus, we took the threshold of the distance image and multiplied it with the original image to show the general area of where Waldo could possibly exist in the picture. A general threshold value actually worked for most of the pictures and reduced the area to be searched by about 60 – 80 %.
In addition, as I mentioned before time constrain was reduced as well. While it took about 115.0850 sec with corr2 method, my modified method took about 11.7410 sec to compute the result. The result is coming from reducing intense calculation by excluding square root and calculating average sum.

2.5 Red Threshold Version of Correlation

To reduce the time necessary to find matches, we modified the program to exclude the calculations where they were not needed. For example, we found that Waldo always contained some red colors. Thus, if we isolated the red areas of the image using RGB values we could reduce the calculation area. The way it works is that the red values below some threshold will be set to 0 and other areas will contain the original values of red. Then if we sum the pixel values of a source image area before the correlation, we have some idea of whether it contains a lot of red or not. If it is below the threshold, it skips calculation on that area. However, there were a lot of problems with this approach and it did not reduce the running time very much.

2.6 Results from Red Threshold Correlation

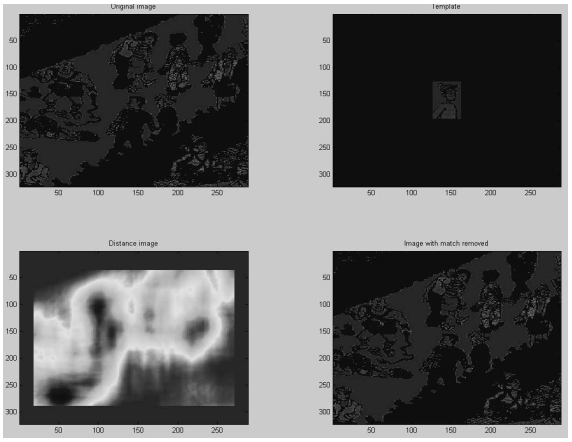


Figure 5. Result from Red Threshold Method
(Red Threshold Source Red Threshold Template
Distance Image, Result Image)

The pictures in figure 5 are from the modified method. At first the threshold excluded about 50% of area to be calculated. There was actually a lot of area that could be skipped. However, the result was very poor. The template contains very high red values since white and red color from RGB contains very high red pixel values. Thus, in actually template matching the area with very high red color was considered as very highly correlated area. In the distance image in the above example, it is evident that with the new threshold, the areas that just happen to have many red colors are detected as highly correlated areas. In contrast, Waldo in the source image lost a lot of its red values because of occlusion from his items. Thus, Waldo matched less than areas with a large amount of red.
In addition, performance time did not decrease by much; it took about 5.7896 sec to produce the image in figure 5. Because our new algorithm needed to threshold and perform additions that were not necessary in the unmodified version, it did not significantly reduce the performance time.

2.7 Other Method Attempts

We tried to apply other various methods to improve the template matching. First we tried to use PCA on the Waldo template. The result was not very good. The reason is that PCA needs to have a data set where Waldo is essentially exactly aligned in each picture. Because Waldo is very small and has a very different orientation in each picture, PCA did not produce reasonable data to use as a template. ICA also had very similar problems.

2.8 Other Approaches

In other approaches, we performed correlation on each Red and Green and Blue image from the RGB image. The result was very similar to the modified method, which just used the gray level correlation. However, it was very time consuming since it needs to run three sets of images. If there were some data sets that could utilize the color differences, this method could most likely be used as well.

2.9 Possible Improvement

There is a lot of room for improvement in this method. We did not actually use any orientation or texture method such as Gabor wavelet or power spectrum to test the image set. The red stripes of the Waldo shirt will respond highly to values of specific Gabor wavelet. Thus, if we use these kinds of wavelets or power spectrums to detect the texture in the test image set we could get better results than just matching the template.

Also, if we could generate a PCA or ICA from the Waldo template to extract its features it would be more efficient in running time and would capture Waldo more effectively. In order to do PCA and ICA, however it would require a lot of data sets and a lot of fixing by hand to actually find matches.

2.10 Conclusion of Correlation

Correlation is method that is good when there is very similar or exact match exists in the picture. However, if the template is very small and it is not an exact match, there could be various problems in template match. Since the small error will account for a big part of the correlation values, it is generally good idea to consider only the area that is really needs to be matched.

3. SIFT: Scale Invariant Feature Transform

SIFT seemed like a good approach to take in solving our problem, because it is robust against changes in conditions such as lighting, scale, rotation, noise and occlusion. Essentially the way SIFT works is that it searches both the template and test images for keypoints that will not be affected by various changes in conditions. It seemed to be ideal for our purposes, because for any given 500x500 image, on average thousands of keypoints are generated, but only a few are needed for a correct match. This way, Waldo being occluded by his accessories or other objects in the image would matter less.

3.1 Application of SIFT

In testing the effectiveness of SIFT in the finding Waldo problem, we used both the C implementation written by David Lowe, and the Intel Matlab code by Scott Ettinger. In both methods, the best-bin-first search method described in the "Object recognition from local scale-invariant features" paper by David Lowe was used to match keypoints in the template to keypoints in the test image.

Using the Intel SIFT code, various keypoints were identified on our Waldo template. The features it seemed to pick up on the most were several of the stripes, Waldo's hat, hair and hand.

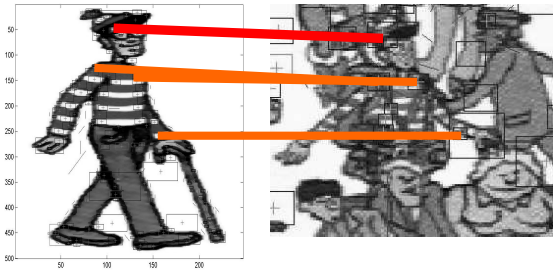


Figure 6.1 Comparison of keypoints found in template image (top) and test image (bottom)

As demonstrated in figure 6, there are several matching keypoints in the template and test images. The most noticeable keypoints were the ones on the hair and stripes of the shirt.

3.2 Results of SIFT

The results of running the SIFT algorithm on the Waldo template were hit and miss. Sometimes Waldo would be detected easily, and other times there were a lot of false matches, or Waldo would not be identified at all. We had to try many different parameters in the algorithm before we found values that actually generated enough keypoints. The other variable we had to change around was the threshold for discarding a match in the best-bin-first matching technique. If the threshold was too low, we got a lot of extraneous matches, but if it was too high, no matches were returned.

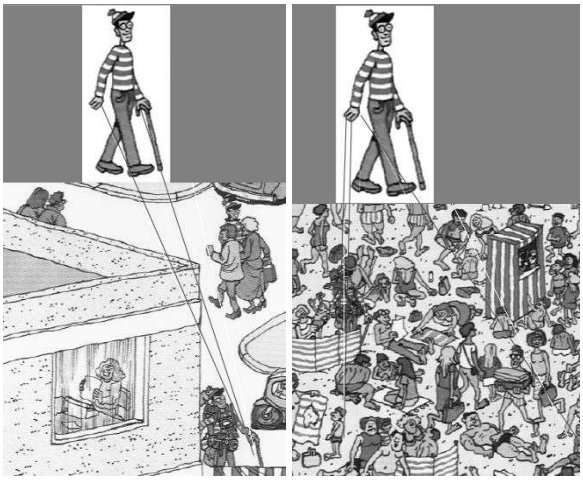


Figure 6.2 Two sample Waldo matches using David Lowe's SIFT code.

The question then became whether the inaccurate matchings were resulting from the implementation of the algorithm itself, or whether the algorithm just wasn't suited for the data. In order to test whether the problem was that the template of Waldo was not consistent enough with the Waldos in the test images, we tried running the algorithm using Waldo templates taken from the test images themselves. This approach yielded much higher success rates, the algorithm finding the correct match each time. The threshold for matching could be a lot lower for the images without returning extraneous matches, thus increasing the certainty of the resulting match.

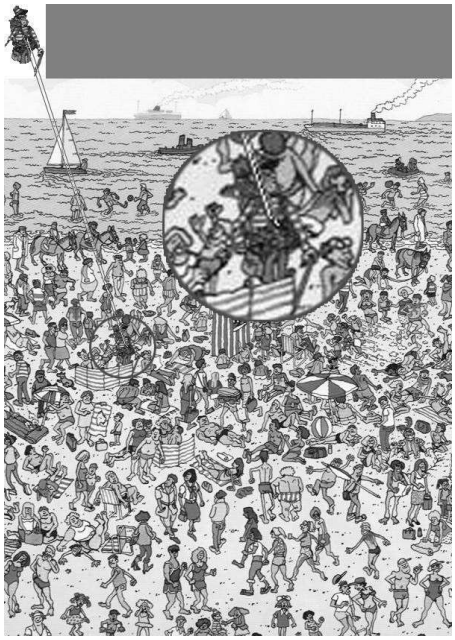


Figure 6.2 Using the extracted Waldo yields much more accurate matches

We also tried using different sizes of Waldo as a template to see if we could improve matches. The larger the template, the more keypoints were generated, which did lead to more accurate matches. However, the Waldos in the test images were too small to generate many keys to compare with. Increasing the size of the image caused our system to run out of memory, so we were not able to see if this might alleviate some of the matching issues.

3.3 Conclusion of SIFT

SIFT turned out to not be the best approach for this problem. There are several possible sources of error that could account for this. The first issue that seems to be one of the biggest sources of error is the size of the templates and the size of Waldo as he appears in the test images. In order for SIFT to work, the template images and test images need to be large enough to generate enough keypoints. We do not feel that we were able to make our test images large enough to be able to find Waldo consistently. A second source of error is that the values used in the keypoint matching algorithm and/or the values used to generate keypoints were suboptimal. Since there is not really a set formula for how to determine these values, the values we used might not be the best values for our dataset. We tried to find the best values through trial and error, but the values still might not be ideal. The answer might just be that SIFT is not a good algorithm to use on this dataset. It could be that the distracter images present in the image had many of the same keypoints, so when other variations occurred, SIFT was not correctly able to find Waldo. Another problem might be that our template of Waldo is not generic enough to find permutations of Waldo in the image. Results listed in other papers that have previously examined SIFT seem to be much more accurate than the results we obtained. Both code implementations of SIFT preformed equally bad. This leads us to believe that the error is either in the image set we have, or in the matching implementation itself.

4, Conclusion

In conclusion, image detection (and matching in general) is a difficult problem. First you there has to be criteria that is easy to find in the image you are looking for that is also unique to the image, As well as robust against noise and occlusion. Once a measurement system is set up to compare the template to the test image, then there has to be an algorithm for deciding whether a hit is actually correct or not. Out of the methods we examined, correlation seemed to yield the best results. Though the running time was about five times as long as the running time for the SIFT algorithm. We feel that

many improvements could be made to the algorithms we used to determine matches.

References

[1] Distinctive image features from scale-invariant keypoints
David G. Lowe, accepted for publication in the *International Journal of Computer Vision*, 2004.

[2] Object recognition from local scale-invariant features
David G. Lowe, *International Conference on Computer Vision*, Corfu, Greece (September 1999), pp. 1150-1157.

[3] Invariant Features from Interest Point Groups,
Matthew Brown and David G. Lowe *British Machine Vision Conference, BMVC 2002*, Cardiff, Wales (September 2002).

Ilsun Lee is a sophomore in the School of Computer Science at Carnegie Mellon University.

Laura Semesky is a sophomore in the School of Computer Science at Carnegie Mellon University. She is getting a minor in Fine Arts. She enjoys art and hopes to be able to incorporate it into her studies in computer science.

Mishkin Face Recognition

Michael Mishkin
Electrical and Computer Engineering
Carnegie Mellon University
mmishkin@andrew.cmu.edu

Abstract

In this project a face recognition system is implemented and trained to recognize images of Mishkin (the author) using PCA with eigenface selection. The classifier software package identifies Mishkin's face with 95% accuracy.

1. Introduction

The ability to recognize familiar faces is inherent to the human visual system and is an ability that is a necessity to our daily social interactions. This skill is so well developed that we can recognize people under different conditions, facial expressions, and even slight changes in facial features such as facial hair or glasses. Modeling such a system has been the focus of study for many computer programmers for the past two decades. With such a multitude of potential practical applications as criminal identification in security systems, a computer user greeting and identification system [1] or other aspects of human-computer interaction, face recognition is quite a fertile area of study. My project for computer vision is a model of such a face recognition system in the Matlab computer programming environment intended to pick out images of my own face among images of other people's faces.

2. Methods

The first aspect to be considered in development of a classifier to identify faces is a method for encoding the faces such that they can be described as a single vector of feature coefficients. An encoding of this form is very useful in the comparison of different faces. For this reason, I chose a PCA based model in which the feature vectors are based on eigenface weights that can be set as the coefficients for a weighted sum of the eigenfaces to accurately reconstruct the original image. Since the eigenfaces used for encoding are the same throughout the image set, the feature vectors found with this method are very good relative representations of each face.

2.1 Principal Component Analysis

Principal component analysis is used to decompose a set of images into a set of principal component images called eigenfaces. Images can be projected into the subspace of eigenfaces which can be considered the face space [2]. Each eigenface can be thought of as an orthogonal axis in this face space such that an image can be mapped to a set of coordinates describing the strength of each principal component. Once a face has been translated into this form, recognition can be done by simply locating the face in the face space and comparing its location to that of the trained set.

With PCA the original image can actually be regenerated with relatively little loss of detail. This is because PCA finds the weighted sum of eigenvectors that best approximates the original image. As more eigenfaces are taken into consideration in this weighted sum, the resulting image becomes more and more similar to the original face. If all of the eigenfaces are considered in the weighted sum then the reconstructed image should be roughly identical to the original image.

2.1.1 PCA Eigenface derivation. The first step in principal component analysis of a set of images is deriving the eigenfaces that best describe that set. A training set is required to base the eigenfaces on and the more images there are in the training set the more eigenfaces will be derived. If the training is set too small, for example if it is only based on a set of images of one person, then the set of eigenfaces will be very good at reconstructing the images of that person but will leave artifacts of the training set in other images and will be a fairly poor reconstructions of the original images. For this reason, in order to gain versatility in the range of images that can be reconstructed, it is necessary to use a sufficiently large database of images.

2.2 The ORL Face Database

The database used in this project is the ORL face database which was collected between April 1992 and April 1994 at the Olivetti Research Laboratory in Cambridge, UK[3]. The image set contains ten images of forty different people for a total of 400 images. All of these images were taken in up-right frontal position on a dark homogeneous background. Otherwise, the images vary in such characteristics as head tilt, facial expression, and lighting conditions. Each of these images is 92 pixels ear to ear and 112 pixels chin to hair.

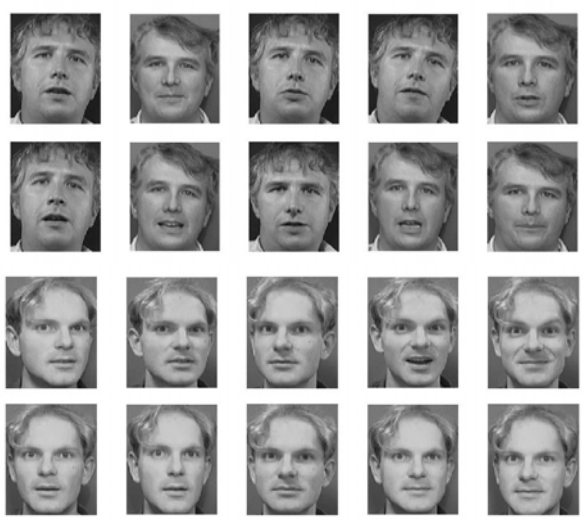


Figure 1. Sample images from ORL Face Database

2.3 Eigenface derivation

The eigenfaces were derived from the first 35 people in the ORL face database. Since there are 350 images total of these 35 people (ten images of each) there were a total of 350 derived eigenfaces. This dataset was sufficiently large to calculate enough eigenfaces to describe a fairly diverse set of images accurately including images outside of the training set.

The eigenface derivation process was loosely based on an adaptation of Matlab code by A.I. Wilmer [4]. His implementation was fairly simple. First the code reads in every image in the training set into a 3d matrix data structure. Once all of the images have been read in then a mean image is calculated. Then the difference between each image and the mean is stored in a second array of images. This step is important since it standardizes the images around a mean face which is a good reference point to be doing calculations from. Each of these difference images is reshaped to a 1d array to be passed into the svd function which calculates the eigenvectors and eigenvalues. The eigenvectors are then reshaped to the dimensions of the images to become the eigenfaces used for image reconstruction.

2.4 Training

Once the set of eigenfaces has been generated the program is ready to begin training. The idea is that after training, an input image can be compared to a mean image for recognition and if the input image is similar enough to this mean it is considered to be a match. The mean image is the resulting image from averaging each of the features of the images in the training set. With the mean image calculated the next step is to find the similarity between each of the training images and the mean image. So the question becomes how to judge similarity to this mean face. By finding the Euclidean distance of each feature vector from the mean feature vector, a value can be associated with how similar the image in question is to the mean. The greatest of these distances among the training images is set as the threshold for a recognized face. Any images found with a distance above this threshold are other people's faces.

Figure 2. Training Set Images

To train for images of my face requires an addendum to the ORL face database. The addendum includes thirty images. Ten of these images are images of me used to train the classifier (Figure 2) and the other twenty are some images of me and some images of my friends to be used as the test set once training is complete. All of these images were standardized to the same image specifications as described earlier. The most significant difference between the new images and the images in the original database is the background color of the images. In order to compensate for this difference the eigenface weights of the addendum may be slightly shifted. Although this may help in discrimination between my face and the faces in the ORL face database this variable is kept constant between the images of me and the images of my friends. Since the main test set is intended to be of me and my friends, the common background color shouldn't have much of an effect on discriminability of my face however it may lead to some false positives

depending on the size of the resulting shift in eigenface weights.

2.5 Eigenface selection

With the training set converted to feature vectors of eigenface coefficients the classifier is essentially ready for recognition. The recognition system described earlier suggests that for the Euclidean distance calculation, the entire feature vector should be taken into account; however, it may be possible to get better results by just selecting those features that are the best at recognizing the faces in the training set. In essence what is needed is a heuristic for eigenface selection. The heuristic that is used is a variation on Fisher discriminants.

2.5.1 Fisher Discriminants. Fisher discriminant analysis is a method for determining how well a feature discriminates between two sets of images. For example, if there are two image sets each corresponding to a different person and the eigenface coefficient feature vectors have already been calculated for each person, the weights of a particular feature in the images from each image set could be analyzed with Fisher discriminant analysis. The Fisher function (which was copied from Michael Schultz's code from homework 4) takes two vectors as input and outputs a single value which is proportional to how well each vector is clustered as well as how well the two vectors are separated. If the values between each vector are separated but the values within each vector are grouped together then the output value will be high and the feature in question is good at discriminating between the two people. However if the two vectors are intermixed and scattered then the output value will be low and the feature in question is not good at discriminating between the two people. Fisher can be applied to every feature for these two people to find an array of values associated with each feature. This array is sorted so as to rank the features in discriminating between the two people.

Since the goal in this project is to be able to recognize images of me this Fisher process is applied to each and every person in the ORL image database and the images of me. So, Fisher compares me to the first person then me to the second person and so on resulting in a series of ranking arrays for each comparison. The top forty features in each of these ranking arrays cast a vote for which features are the strongest overall. At this point, a cutoff is set for how many eigenfaces are to be taken into account in calculation of distance from the trained mean image. Finding this cutoff requires some calibration. Since the selected features are the best at discriminating between the trained person and everyone else this should yield better results than when all of the eigenfaces were taken into account.

3. Results

The first classifier that is tested takes all of the eigenfaces into account in the calculation of the distance from the mean. When the classifier is run on all of the images in the ORL image database none of the images are detected as images of me. Even among the test set, there were no false positives however some of the images of me were not identified. This seemed to only occur in cases when my facial expression was exaggerated. This method was identified 90% of the twenty pictures of me between the test set and the training set. The classifier's test set output can be seen below in figure 3.

Figure 3. Sample output of classifier

Recognition of a few of the people in the ORL database was also tested with this classifier. Since there were no test images for these people and their database images were used as the training set (which is inherently recognized) the classifier test for these people was limited to a test of the precision of the classifier and in most cases there were no false positives.

Before the classifier with eigenface selection could be tested some calibration of the number of eigenfaces to select was necessary. The best amount for the test set turned out to be 23 eigenfaces but since this was the only training set there was no way to check if this would hold true for another training set. Any amount that was chosen above 23 began to introduce false positives while maintaining similar accuracy while amounts less than 23 eigenfaces were less accurate in identification of images of me. With 23 eigenfaces 95% of the images of me were identified with no false positives. So Fisher discriminants for eigenface

selection turned out to result in the best classifier of images of me.

4. Discussion and Conclusions

The classifier using the entire set of eigenfaces is surprisingly good at identifying images of me. As expected slightly better results are possible with eigenface selection but the amount of calibration that was necessary to get such accurate results introduces some doubt as to whether the classifier would be as accurate with a different training set, without necessitating further calibration. None the less, the classifier as it stands can identify images of me from among a set of images in excess of 400 faces of over 40 different people. The next stage for this software package would be to train for more than one person. This would be entirely possible with the current set up of the program and would only require an additional check for the rare case that an image were below threshold for recognition of more than one person. If this were implemented it would also be nifty to implement a nicer GUI for the program and perhaps an interface for loading new images into the database. It may even be possible to classify facial expressions using similar methods but that would be a different project altogether.

5. References

[1] Bansal, Arjun, “User-identification using Face Recognition Algorithms”, CNS 186 Final Project Report, March 2004.

[2] Matthew Turk and Alex Pentland, "Eigenfaces for Recognition", Journal of Cognitive Neuroscience, Volume 3, No. 1, 1991, pp 72-86.

[3] F. Samaria and A. Harter, "Parameterisation of a stochastic model for human face identification", 2nd IEEE Workshop on Applications of Computer Vision, December 1994, Sarasota (Florida).

[4] A. I. Wilmer, “Matlab Code for Eigenfaces”, http://www.ecs.soton.ac.uk/~aiw99r/faces/, Sept 2002.

Michael Mishkin is a junior in the Integrated Masters/Bachelors program in Electrical and Computer Engineering at Carnegie Mellon University. Hailing from New Jersey, he enjoys long nights of code hacking and Krispy Kreme Donuts.

Face Recognition using SIFT features

Rohit Patnaik
Department of Electrical and Computer Engineering
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213
rpatnaik@andrew.cmu.edu

Abstract

A combined face detection and recognition approach is examined. It is based on the scale-invariant feature transform (SIFT) algorithm. The database consists of images containing the target face with variations in illumination, pose, and scale. A simplistic method for performing recognition using the SIFT features is evaluated. A current method of performing 3D object recognition based on the SIFT features is analyzed.

1. Introduction

Object detection is an important step in object classification. Efficient and accurate methods for detection enable the overall classification system to be implemented in real-time with higher accuracy. Object detection is complicated by variations in lighting and pose of the object(s) in the target scene and possible occlusion from other objects. Face detection is even more challenging due to intra-class variations caused by changes in appearance and expression.

Much prior work exists on object detection in general and face detection in particular. In Ref. [1], a probabilistic model of local appearance and spatial relationship is constructed to perform object detection. The algorithm explicitly models and estimates the posterior probability, P(object|image). Another method for face detection uses a boosted cascade of simple features [2] based on the AdaBoost algorithm [3]. Both methods have been shown to provide very good results for object detection (including face detection).

The purpose of this paper is to examine simultaneous face detection and recognition. A two-step detection/recognition method may be employed to solve this problem. Detection can be carried out using one of the methods described before. This needs to be followed by a subsequent recognition step. Face recognition can be performed in one of two main ways – template-based approach and model-based approach. In the template-based approach, one creates a template of the object by extracting feature vectors from a set of training images. One extracts the same features from the test image and computes a distance metric based on which a recognition score is assigned. In the model-based approach, the image is fit to a geometric model. This is achieved by estimating the parameters of the geometric model that best represent the data. Based on the computed parameters, the recognition (verification or classification) score is assigned.

Instead of taking a two-step approach, one can perform simultaneous detection and recognition. One way to combine the two objectives would be to create training data using the methods in [1] or [2] for a specific face. One potential drawback of the algorithms in [1] and [2] is that to recognize an object under different variations, those variations must be included in the training process. Recently, an object recognition approach has been proposed based on the scale-invariant feature transform (SIFT) described in [4]. The features are invariant to image scale and rotation, and are shown to provide robust matching across a substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination. By using these features and only using a few images in the training set, one can hope to achieve reliable recognition.

In this paper, I examine the SIFT-based approach of performing face recognition under illumination, scale and pose variations. I present a simplistic method of performing face recognition with the SIFT features, which is found to give poor results. I examine the cause for the poor matches, and present the method in [5] to perform 3D object recognition. That algorithm combines multiple images of a 2D object into a single model representation. The rest of this paper is organizes as follow. Section 2 describes the SIFT feature extraction algorithm. Section 3 presents my

simplistic algorithm for performing face recognition with the SIFT features. Section 4 presents results using my approach along with an analysis of its failures. Section 5 gives an overview the 3D object recognition algorithm described in [5].

2. SIFT feature extraction

The SIFT features are invariant to image scaling and rotation, and are partially invariant to changes in illumination and 3D camera viewpoint. They are well localized in both the spatial and frequency domains, reducing the probability of disruption by occlusion, clutter, or noise. A cascade filtering approach, in which the more expensive operations are applied only at locations that pass an initial test, minimizes the cost of extracting these features. Following are the major steps in generating the set of image features (keypoints) (the figures in this section have been taken from [4]):

2.1. Detection of scale-space extrema

The first stage of keypoint detection is to identify locations and scales that can be repeatably assigned under differing views of the same object. This is accomplished by searching for stable features across all possible scales, using a continuous function of scale known as scale space. It has been shown by Koenderink (1984) and Lindeberg (1994) that under a variety of reasonable assumptions the only possible scale-space kernel is the Gaussian function. The scale space of an image is defined as a function, $L(x, y, \sigma)$, that is produced from the convolution of a variable-scale Gaussian, $G(x, y, \sigma)$, with an input image, $I(x, y)$:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y), \qquad (1)$$

where * is the convolution operation in *x* and *y*, and

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right). \qquad (2)$$

To efficiently detect stable keypoint locations in scale space, the scale-space extrema in the difference-of-Gaussian (DOG) function convolved with the image, $D(x, y, \sigma)$ is used, which can be computed from the difference of two nearby scales separated by a constant multiplicative factor k:

$$\begin{aligned} D(x, y, \sigma) &= (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma). \end{aligned} \qquad (3)$$

An efficient approach for the construction of $D(x, y, \sigma)$ is shown in Figure 1. The initial image is incrementally convolved with Gaussians to produce images separated by a constant factor *k* in scale space, shown stacked in the left column. Adjacent image scales are subtracted to produce the difference-of-Gaussian images shown on the right. Once a complete octave has been processed, the Gaussian image that has twice the initial value of σ is downsampled by a factor of two in each direction.

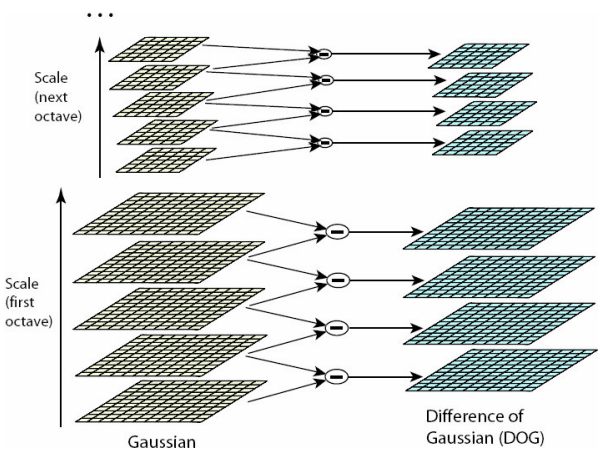


Figure 1. **Efficient construction of $D(x,y,\sigma)$**

In order to detect the local maxima and minima of $D(x, y, \sigma)$, each sample point is compared to its eight neighbors in the current image and nine neighbors in the scale above and below (this is illustrated in Figure 2, the pixel marked X is compared to its 26 neighbors in 3x3 regions in current and adjacent scales, marked with circles). It is selected only if it is larger than all of these neighbors or smaller than all of them. Since most of the sample points are eliminated after the first few checks, the cost of this check is reasonably low.

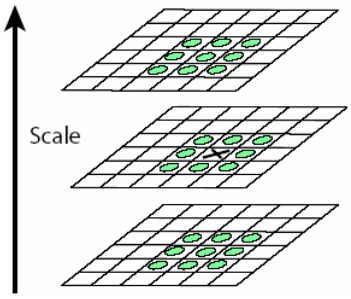


Figure 2. **Detection of maxima and minima in the DOG images**

2.2. Accurate keypoint localization

Once a keypoint candidate has been found by comparing a pixel to its neighbors, the next step is to perform a detailed fit to the nearby data for location, scale, and ratio of principal curvatures. This information allows points to be rejected that have low contrast (and are therefore sensitive to noise) or are poorly localized along an edge. For stability, it is not sufficient to reject keypoints with low contrast. The DOG function will have a strong response along edges, even if the location along the edge is poorly determined and therefore will be unstable to small amounts of noise. A poorly defined peak in the difference-of-Gaussian function will have a large principal curvature across the edge but a small one in the perpendicular direction. The principal curvatures can be computed from a 2x2 Hessian matrix, **H**, computed at the location and scale of the keypoint:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \quad (4)$$

The derivatives are estimated by taking differences of neighboring sample points.

2.3. Orientation assignment

By assigning a consistent orientation to each keypoint based on local image properties, the keypoint descriptor can be represented relative to this orientation and therefore achieve invariance to image rotation. The following approach is used to assign local orientation. The scale of the keypoint is used to select the Gaussian smoothed image, *L*, with the closest scale, so that all computations are performed in a scale-invariant manner. For each image sample, $L(x, y)$ at this scale, the gradient magnitude, $m(x, y)$, and $\theta(x, y)$ is precomputed. An orientation histogram is formed from the gradient orientations of sample points within a region around the keypoint. The orientation histogram has 36 bins covering the 360 degrees range of orientations. Each sample added to the histogram is weighted by its gradient magnitude and by a Gaussian-weighted circular window with a σ that is 1.5 times that of the scale of the keypoint. The highest peak in the histogram indicates the orientation of the keypoint.

2.4. Keypoint descriptor

The previous steps assign an image location, scale, and orientation to each keypoint. These parameters impose a repeatable local 2D coordinate system in which to describe the local image region, and thus provide invariance to these parameters. The next step is to compute a descriptor for the local image region that is highly distinctive yet is as invariant as possible to remaining variations, such as change in illumination or 3D viewpoint. Figure 3 illustrates the computation of the keypoint descriptor. Using the precomputed gradients computed using the algorithm in Section 2.3, the image gradient magnitudes and orientations are sampled around the keypoint location, using the scale of the keypoint to select the level of Gaussian blur for the image. The coordinates of the descriptor and the gradient orientations are rotated relative to the keypoint orientation in order to achieve orientation invariance. The gradients are illustrated with small arrows at each sample location on the left side of Figure 3.

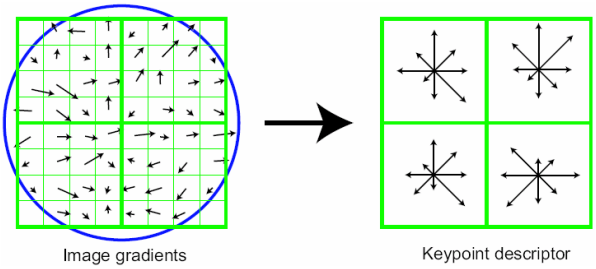


Figure 3. **Computation of keypoint descriptor**

A Gaussian weighting function with σ equal to one half the width of the descriptor window is used to assign a weight to the magnitude of each sample point. This is illustrated with a circular window on the left side of Figure 3. Doing so avoids sudden changes in the descriptor with small changes in the position of the window, and to give less emphasis to gradients that are far from the center of the descriptor, as these are most affected by misregistration errors.

The keypoint descriptor is shown on the right side of Figure 3. It allows for significant shifts in gradient positions by creating orientation histograms over 4x4 sample regions. The figure shows eight directions for each orientation histogram, with the length of each arrow corresponding to the magnitude of that histogram entry. A gradient sample on the left can shift up to 4 sample positions while still contributing to the same histogram on the right, thereby achieving the objective of allowing for larger local positional shifts. The descriptor is formed from a vector containing the values of all the orientation histogram entries, corresponding to the lengths of the arrows on the right side of Figure 3. The figure shows a 2x2 array of orientation histograms, whereas in [4], a 4x4 array of histograms with 8 orientation bins in each is used.

Thus a 4x4x8=128 element feature vector is used to describe each keypoint. Finally, the feature vector is modified to reduce the effects of illumination change.

3. Simplistic recognition algorithm

Keypoints from a set of training images are extracted and stored. Keypoints are extracted from the test image. For a given keypoint in the test image, its Euclidean distance is computed from each keypoint in the training set (using the keypoint descriptor). If the ratio of the distance of the closest keypoint to the second-closed keypoint is below a threshold, the match is rejected. The idea behind this is that since the keypoint descriptors are supposed to be highly specific, if the ratio of the distances is low, it indicates that the keypoint in the test image cannot be matched well to a unique keypoint in the training set. I chose 0.6 as the threshold.

An implementation of this method (the results are shown in the next section) indicated that enough matches were not being obtained in the test image containing the target face. Thus, I was not able to complete the face detection step. During the course of writing this paper, an explanation for the above phenomenon was found. If the training images are sufficiently similar, then they may give rise to keypoints with simlar descriptors. In that case, if a keypoint from the test image has a high ratio of the closest to the second-closest distance, that might be due to it matching similar (correct) keypoints in the training set. I should have used the threshold metric for comparing the keypoint against all keypoints in one training image, and repeated the process separately for each training image. I was not able to find a way to combine the keypoint matching information across various training images.

4. Results

I chose to perform recognition on my face. Figure 4 shows all the images in the database (labeled 1-21 left to right and top to bottom). The database consists of 21 color images of me under different illuminations and 3D viewpoints, and with varying backgrounds. Some pictures are with glasses while others are without; one picture contains another person in the scene. Some of the pictures contain an occluded version of my face. I converted all the images to grayscale. I chose five pictures (images 3-7) as the training set. I cropped the images from 480x640 pixels to 144x192 pixels to only contain the face region.. I used the Invariant Keypoint Detector demo software [6] to extract the keypoints from the training images. Figure 5 shows the training images along with the training images with the arrows overlayed indicating the locations, scales, and orientations of the key features.

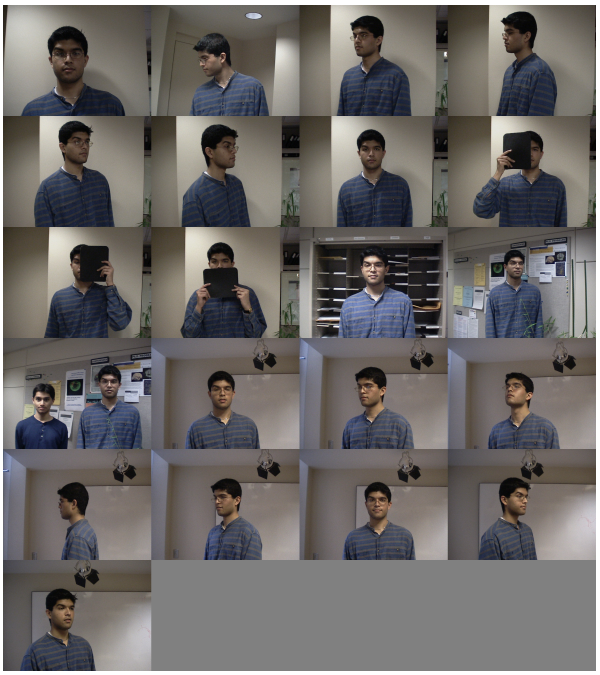


Figure 4. **Image database**

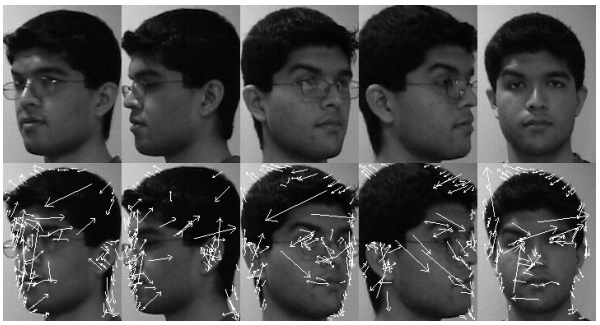


Figure 5. **Training set and extracted keypoints**

Figures 6 and 7 show the keypoint matches for a good match and a poor match. The training images are shown above the test image. The white lines shown in the image connect keypoint in the test image to its best match in the training set (if it passes the distance-ratio threshold test). Note that while the test image in Figure 6 contains the target face that looks very similar to one of the training images (the one containing the matching keypoints), the test image is taken under different lighting and at a different zoom (scale) and the test

image shows the face with glasses. The 3D viewpoint is almost identical. This shows that the keypoints can be matched in the presence of illumination and scale variations. The target image in Figure 7 has a 3D viewpoint that is very similar to one of the training images. Therefore, it is surprising that good keypoint matches could not be found. Increasing the value of the distance-ratio threshold from 0.6 to 0.9 in increments of 0.1 led to an increase in the number of keypoint matches, however most of the additional matches were due to background clutter. Thus, this method of keypoint matching is not a reliable way to perform recognition. If one restricted the pose variations and used more training data at a restricted number of poses, one should be able to obtain better results even with the naive approach.

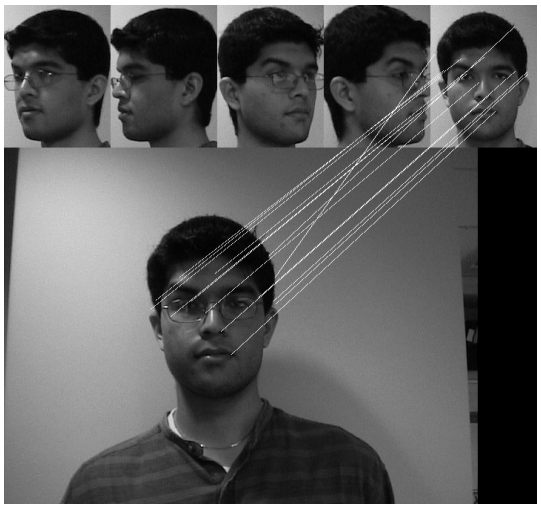


Figure 6. **Example of a good match**

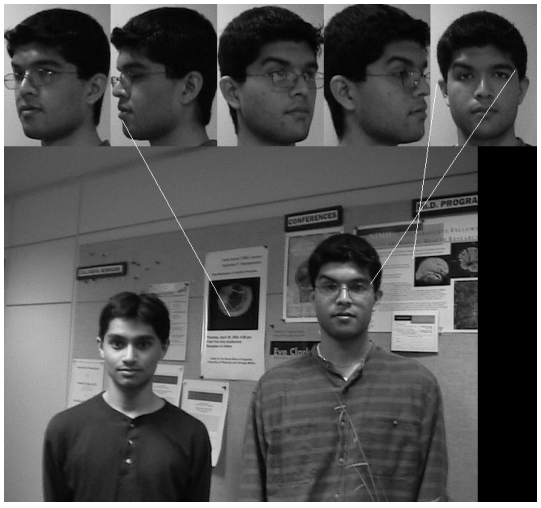


Figure 7. **Example of a poor match**

5. 3D object recognition

The algorithm presented in Section 4 was based on matching keypoints in the test image to individual training images. As such, it is difficult to combine the keypoint matching information across training images at various poses, to identify the pose of the object in the test image. This difficulty is overcome by the algorithm presented in [5], that combines multiple images of a 3D object into a single model representation. This view clustering method provides for recognition of 3D objects from any viewpoint, the generalization of models to non-rigid changes, and improved robustness through the combination of features acquired under a range of imaging conditions. The decision of whether to cluster a training image into an existing view representation or to treat it as a new view is based on the geometric accuracy of the match to previous model views. A new probabilistic model is developed to reduce the false positive matches that would otherwise arise due to loosened geometric constraints on matching 3D and non-rigid models.

It should be noted that the algorithm presented in [5] relies on both a bottom-up and a top-down approach. It is a non-trivial method and requires the building up of a probability model.

6. Conclusion

The scale-invariant feature transform (SIFT) method has been examined for its applicability to for detecting a target face under variations in illuminations and 3D viewpoint. A simplistic method for keypoint matching presented here gives poor results. In order to use the keypoints effectively for face recognition across pose a more complicated method needs to be used which involves both a bottom-up matching scheme, based on a distance metrics and Hough transforms, and a top-down scheme based on prior probability models for the number of matches.

Future work will involve the evaluation of the view clustering method for 3D face recognition. Implementation of a method for performing recognition for small variations in illuminations and pose will serve as good starting point.

Acknowledgments

The author would like to thank Prof. Tai Sing Lee of Carnegie Mellon University, who was the instructor for the Computer Vision course, for providing valuable guidance for this project.

References

[1] H. Schneiderman and T. Kanade, "Probabilistic modeling of local appearance and spatial relationships for object detection," CVPR 1998, pp. 45-51, June 1998.

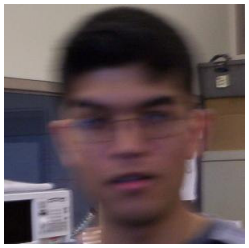
[2] P. Viola and M. Jones, **"**Probabilistic modeling of local appearance and spatial relationships for object recognition,**"** CVPR 2001 **1**, pp. 511-518, December 2001.

[3] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *Computational Learning Theory: Eurocolt '95*, pp. 23–37, Springer-Verlag, 1995.

[4] D. G. Lowe, "Distinctive image features from scale-invariant keypoints", accepted for publication in the *International Journal of Computer Vision,* 2004.

[5] D. G. Lowe, "Local feature view clustering for 3D object recognition," CVPR 2001 1, pp. 682-688, December 2001.

[6] "Invariant Keypoint Detector Demo Software," http://www.cs.ubc.ca/~lowe/keypoints/



Rohit Patnaik is a senior in the Department of Electrical and Computer Engineering (ECE) at Carnegie Mellon University (CMU). He will receive his BS and MS degrees in ECE in May 2004 with college and university honors. He will be pursuing a Ph.D. degree in the area of Signal Processing in the ECE dept. at CMU beginning May 2004.

Winning Patterns in Go

George Fraser
Carnegie-Mellon University
Pittsburgh, PA 15217
gfraser@andrew.cmu.edu

Abstract

In the realm of computer game-playing, the game of Go represents a notable failure of traditional, search-based artificial intelligence algorithms. This is the result of its extremely high branching factor, with up to 361 moves available to a player. Expert human players contend with this by approaching the game in a highly visual, intuitive way. In this paper principal component analysis is used to identify patterns in expert Go play. These patterns are shown to be meaningful by constructing a regression model which uses the patterns to determine the winning player given an end-game board state.

1. Introduction

Go is an unusual game that has eluded traditional game-playing algorithms. Originating in east Asia, it has attracted attention in the western world as markedly different from games such as chess and checkers. For the same reason it has attracted the interest of AI researchers: unlike most board games, it is completely unsuitable for minimax search. The simplicity of play, in which each open space allows a single move, belies the difficulty of the game: the branching factor of search is prohibitively high. This is the result of the large size of the board, and the fact that virtually all plausible moves are available at every turn. Go and chess are often compared as difficult and popular games, and this comparison serves as a good example of what makes a game suitable or unsuitable for minimax search: chess has a much smaller board, with 64 versus 351 available squares, and at any given time many of the pieces will be blocked, limiting the space of legal moves.

Expert human players tend to approach Go in a highly visual way. The only effective Go programs similarly rely on huge databases of patterns encompassing up to millions of rules [1]. These rules can be created using a variety of methods, and in some programs they are manually programmed in by human experts. The identification of patterns which are relevant to the game is an important task in improving the performance of computer Go players.

1.1. Previous Work

There has been extensive work on computer Go. A good review is *Computer Go: an AI Oriented Survey* [1] by Bouzy and Cazenave. The work most immmediately relevant to this paper is *Local move prediction in Go* [3] by Werf et. al. The authors used principal component analysis (PCA) to reduce the dimensionality of Go games and to provide inputs to a neural network which evaluated the strength of game positions. While the network was unable to play with any competency, and was too computationally slow to incorporate into a search algorithm as an evaluation function, it was able to choose the next move given a game in progress with some degree of success. Their work demonstrates that PCA, among other similar algorithms including some of their own making, is capable of reducing the dimensionality of Go board states.

Principal component analysis and independent component analysis are used extensively in this paper. They can be thought of as rotations of the coordinate system in which the data exists. Consider our data: case Go board windows with 49 locations having either a white stone (encoded as 1), a black stone (encoded as -1), or no stone (encoded as 0). We can consider each window to be a point in 49-dimensional space. The axes of this coordinate system are straightforward: each represents to one of the 49 positions varying between white, black, and neutral. We can, however, re-represent this system by rotating the coordinate system such that each axis now represents a combination of several locations varying to different degrees.

Principal component analysis gives a representation which is optimally compact in the least-squares sense: that is, each successive axis minimizes the remaining error in the data set. Independent component analysis attempts to maximize the gaussianity of the data with respect to each independent component; that is, it attempts to identify components which are uncorrelated with each other.

In this study it is hoped that PCA or ICA will be able to identify meaningful patterns in Go boards. The fact that the data is of professional play means that it will be clustered in regions which correspond to meaningful patterns. Component analysis should be able to pick up on these clusters.

2. Methods

The data used in this study were the records of the Kisei tournament matches between 1977 and 1996, a total of 119 games. They are available for download online [5]. The end-game board states were computed using WinMGT, a freely available interpreter for the go game records.

Many 7×7 windows were collected from the boards to provide training data. A few examples are shown in figure 1. In order to prevent the analysis from being confounded by repetitions of the same patterns translated as the window is moving across the board, all windows were centered on a simple corner feature (figure 2). This method is analagous to that of the SIFT algorithm [2]. The boards are rotated and inverted (figure 3) in order to get all the usable data and to generate boards from the perspective of both winner and loser. Windows where the corner feature belongs to the winning player and those where the it belongs to the losing player are analyzed separately. The 50 boards in the set gave about 2000 windows after rotation and inversion.

3. Results

Independent component analysis, ICA with dimensionality restricted to 10, and principal component analysis were applied to the data. In all cases the FastICA package was used to perform the computation. In unrestricted ICA and PCA the number of components was limited by the algorithm to 46 for obvious reasons: a 7×7 windows has 49 spaces, 3 of which are by definition part of the corner feature, leaving only 46 directions of variation.

Unrestricted ICA gave what is in retrospect an expected result: most of the components were simply single points on the board (figure 4). This makes sense, because the windows are a linear sum of 46 different elements.

Restricting the ICA to 10 dimension gave more interesting results (figure 4). The patterns are now more complex. Principal component analysis gives patterns which on their face appear similar.

In order to veryify whether these patterns were actually meaningful, ordinary least-squares multiple regression was used to create a model for the "winning-ness" of 7×7 windows with the corner feature. The same training data that generated the principal components was used to construct the regression model. The coefficients of the regression model are plotted in figure 5.

The regression model was then tested against both the same training data and a new validation set of 10 more end-game boards. To determine the winner given a board, all of the windows with the corner feature are identified and fed into the regression model. The resulting values for each window are summed together and form the overall index for the board. If it is positive, it is indicated that white (encoded as +1) is the winner.

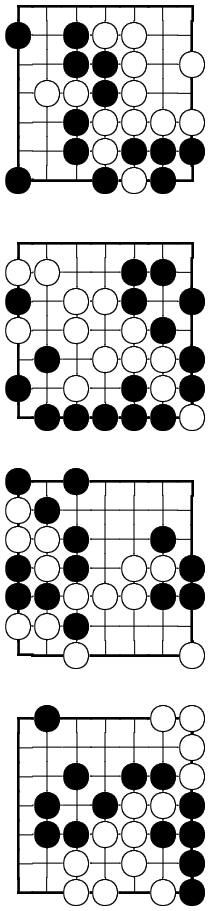


Figure 1: Examples of windows in the data set.

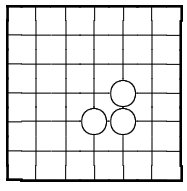


Figure 2: Go corner feature.

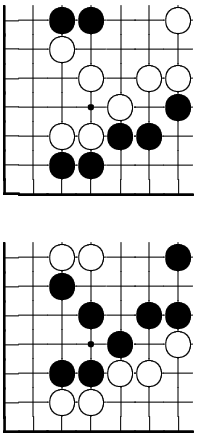


Figure 3: Inverting the board.

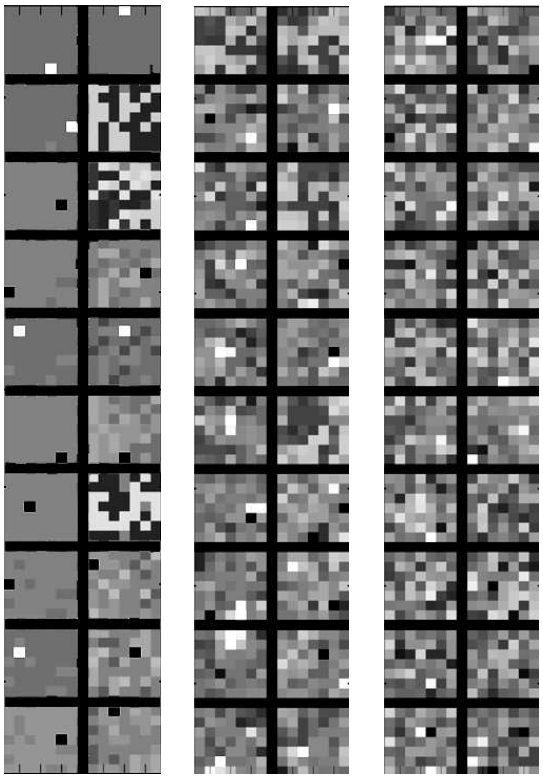


Figure 4: First ten independent components of windows in the data set. Components from independent component analysis, ICA where dimensionality has been restricted to 10, and principal component analysis are shown. Components derived from patterns belonging to the winner are on the left, from the loser on the right.

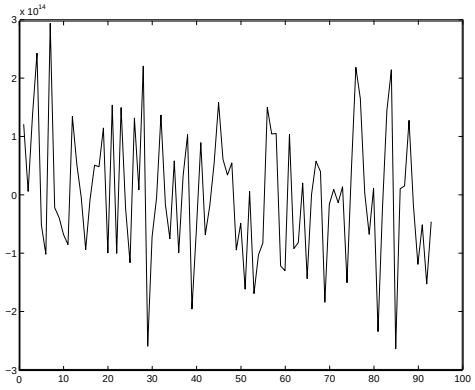


Figure 5: Regression model for winning as a function of the response of corner windows to principal components.

	PCA	ICA (restricted)
Training Data:	$39/50$	$33/50$
Validation Data:	$9/10$	$7/10$

Figure 6: Accuracy results for PCA and restricted-ICA based winning models.

As may be expected the unrestricted ICA was unpredictive. Apparently the patterns in the game are too complex to be captured by a simply linear model of stones generated by regression. However, both restricted ICA and PCA showed predictive power. The results are shown in figure 6. To check that the model was not simply counting stones, a straightforward counting algorithm was tested and showed no significant predictive power.

4. Summary and Conclusions

Determining the winner of a Go game by the state of the board at the end is a definitely not a trivial problem. In the tournament games used in this study, much of the winnings are implicit. Stones are assumed to be lost, territory taken, without the scenario actually having been played out. Professional Go players know when to give up on a region, and so these end-games are highly incomplete. Most of the games in this tournament were won not by counting up territory and prisoners, but by the resignation of one of the players, who sees a hopeless situation. WinMGT, which is capable of computing the score from the record of the game, adding up all prisoners and territory, was not able to determine the outcomes of the games correctly.

In light of this one cannot help but to be somewhat surprised that this algorithm worked at all. It is a rather simple technique applied to a very complex problem. The predicive power of the model indicates that the principal components

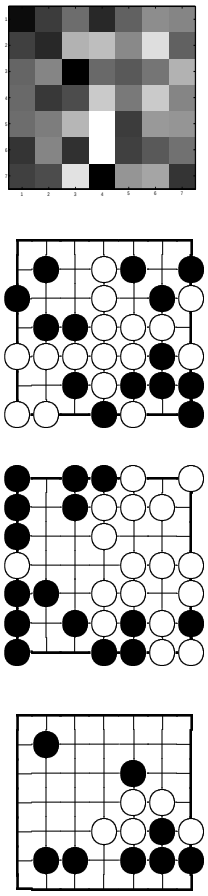


Figure 7: Windows which per-stone garnered the greatest response from the first principal component of the set of winning windows in the data.

must have a significant meaning in terms of the structure of advantageous Go positions. The fact that the unrestricted ICA model, which was effectively just a linear model of the presence of the stones at each location, did not work further indicates that the patterns derived from PCA are meaningful.

We can attempt to find out what the meaning of these patterns is by determining those windows in the data set which are most similar to the principal components and therefore make the greatest contribution to the model. We can refine the criteria to be the greatest similarity per stone in the window, in order to avoid choosing only densely packed windows which are harder to interpret. Presented in figure 7 is one example, showing 9 windows which were most similar to the first principal component of the data. Notably, identical 7×7 windows were found in several cases.

Upon inspecting these results and the windows which

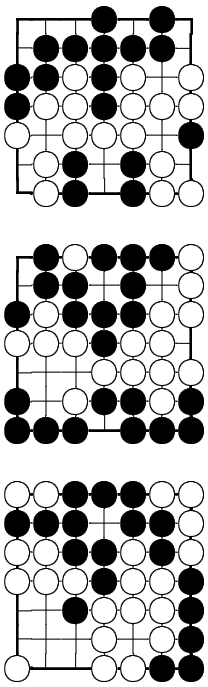


Figure 8: Windows which were determined by the regression model to have the highest likelihood of being part of a board where white won.

responded best to four other principal components I could identify no clear patterns. I then identified those windows deemed by the regression model most likely to be part of a winning board. Several are presented in figure 8. They were also unfortunately rather uninformative.

Future work could focus on using these patterns derived from PCA as part of a computer Go player. They could form the basis of an evaluation function, or a move-ordering function to improve pruning. There are many opportunities for improvement on the existing algorithm, the most immediate of which would be the inclusion of features other than corners, and training on a larger data set.

Acknowledgments

Dr. Tai Sing Lee provided invaluable suggestions along the way for how to turn a rather abstract idea about patterns in Go into a testable algorithm.

References

[1] B. Bouzy, T. Cazenave. “Computer Go: an AI Oriented Survey”. *Artificial Intelligence*, Vol. 132(1), pp. 39–103, October 2001.

[2] D. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints”. *International Journal of Computer Vision*, 2004.

[3] E.C.D. van der Werf, J.W.H.M. Uiterwijk, E.O. Postma, H.J. van den Herik. “Local move prediction in Go”. *In LNCS 2883: Computers and Games*. Third International Conference, CG 2002.

[4] FastICA Toolkit, Helsinki University of Technology. http://www.cis.hut.fi/projects/ica/fastica

[5] http://www.cs.ualberta.ca/ mmueller/go/kisei.html

The Author

George Fraser is from the New York area and is a Junior in the bachelor program in Cognitive Science at Carnegie Mellon University. He is currently working as an undergraduate research fellow under Dr. Daniel J. Simons at the University of Pittsburgh Neurobiology department.