Image Processing

Assignment

2020

Candidate: 213120

Degree: MSc Intelligent & Adaptive Systems

University of Sussex – School of Engineering and Informatics

Contents

1. Introduction

We were tasked with writing a function capable of reading a colour pattern image from the hard disk, before returning an array that was representative of that pattern. With there being five possible colours; white, blue, yellow, green and red – a typical input and corresponding output should be of the form seen below.

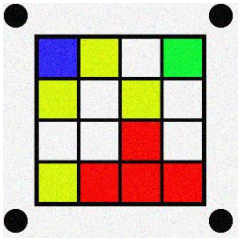

Figure 1.1 - Typical Basic Input

$$\texttt{Result} = \begin{bmatrix} b & y & w & g \\ y & w & y & w \\ w & w & r & w \\ y & r & r & r \end{bmatrix}$$

Figure 1.2 - Corresponding Output

It must be noted that my proposed solution is designed around the slightly more challenging *SimulatedImages* dataset as opposed to *images2*. Although the high-level approach is transferrable, there would be differences in implantation during the portion responsible for tile location, outlined in section 2.2; due to the absence of the black boarders in *images2*. Another caveat that must be raised is that the input images may be subject to any range of rotation, skew or noise that is not present in Figure 1.1.

2. Approach

2.1 Transformation

Although there exists a certain extent of variation amongst the target images, we are able to follow a fairly uniform structure to achieve the desired output for each image. This uniformity between the images can only be achieved through proper pre-processing, the first goal of which is to account for any skew or rotation within the original image. This would be a necessary step for any robust implementation of these methods, as in many applications the camera and target are unlikely to remain perfectly aligned.

We will be using the 4 black circles that are present in each image as reference points to apply the appropriate correcting transformations to each image. In order to locate these, I first converted the image to inverted black and white, before filling holes using matlabs inbuilt `imfill` function. The results of these steps being applied to *proj1_2.png* can be seen in Figure 2.

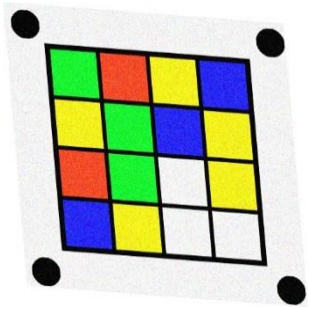

Figure 2.1 - Original Image

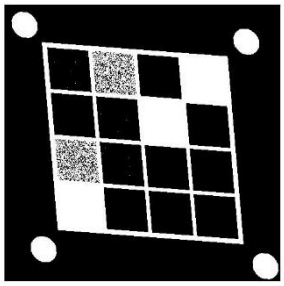

Figure 2.2 - Inverted B&W

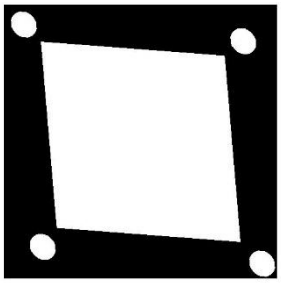

Figure 2.2 - Holes Filled

Matlabs inbuilt `regionprops` function can then be used on the result displayed in Figure 2.3, with *Centroid* and *Area* also being passed in as parameters. This will return a data structure with a tuple for each 8-connected component within the image, and fields to match our chosen additional parameters. There will therefore be 5 tuples, corresponding to each of the 4 circles as well as the white square seen in figure 2.3. We can then isolate the circles by iterating through the tuples, removing that which has the greatest area – as this will be the one that corresponds to the square.

Since we also passed *Centroid* through as a parameter, we now have the x and y co-ordinates of the centre of each circle, we will use these values as the moving points for our transformation. Since we want to translate all images such that they are uniform, we can hardcode the corresponding fixed points based on those in a non-skewed or rotated image such as the one seen in Figure 1.1. We can then use `fitgeotrans` to create a transformation matrix based on these two sets of points. Once applied to the original image with `imwarp`, we will be left with the below:

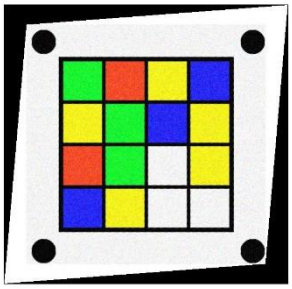

Figure 3 - Transformed Image

2.2 Tile Location

We will employ similar techniques and pre-processing methods to locate each tile as we did in section 2.1 for the circles. The transformed image is converted to black and white before being eroded. The threshold about which we binarize the image in order to convert it to black and white, as well as the width of the structuring element used during the erosion are two very key parameters when we consider the systems ability to work with the noisier input images. I will discuss this in more detail later, but there exists a trade off here - tuning these values to achieve better success on noisy images, comes at a detriment to its performance on those with less noise, and vice versa.

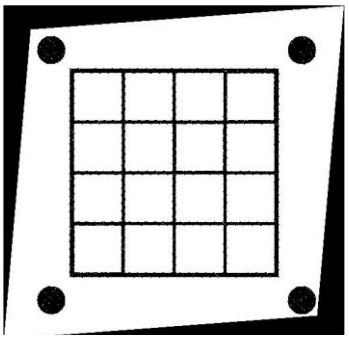

Figure 4.1 - Transformed + B&W

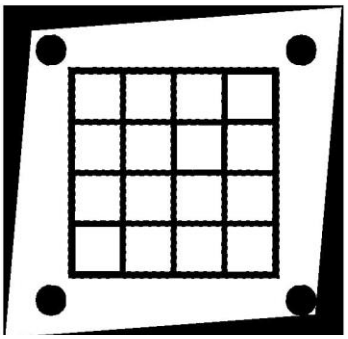

Figure 4.2 - Results after Erosion

The results can be seen in Figure 4. Notice the presence of many small noisy pixels in Figure 4.1 that are removed by the erosion process and are thus no longer visible in Figure 4.2. This is necessary as we are about to once again employ the `regionprops` function to retrieve the area and centroid co-ordinates of each component, however now we will be aiming to isolate the tiles. I chose to do this by first calculating the median area for all components, and removing any components that strayed more than 10% from this value. Although the tiles may appear to all have the same area, there is naturally a low level of variation amongst them due to imperfections within the system – whether they be as a result of noise or any previous pre-processing steps, and so I couldn't just check for matching areas. Comparing each to an average area was also not an option, as the largest component is an order of magnitude larger, skewing this figure beyond any real use.

2.3 Colour Identification

In order to identify the colour of each tile, an obvious approach may have been to simply sample our newly found centre pixel for each tile, however in Figure 5 you can see why this would be insufficient.

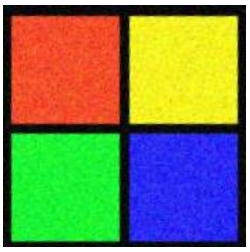

Figure 5.1 - Before Median Filter

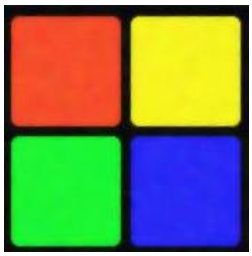

Figure 5.2 - Median Filter Applied

Figure 5.1 gives a slightly closer look at 4 of the tiles from a typical input image. Within each coloured tile there is non-negligible variation in the colours of individual pixels that needs to be accounted for. I made the decision to use a median filter rather than an averaging one for the same reason I used this as a metric to remove potential tiles during section 2.2 - much like the actual tiles made up the vast majority of the tuples within the `regionprops` datastructure, my point of interest here is the natively coloured pixels within each tile, of which they are again the vast majority. An averaging filter would have been influenced by every pixel, increasingly more so for the more anomalous pixels, giving the noise more influence than with a median filter. Figure 5.2 shows the same tiles as figure 5.1, once the median filter has been applied.

My next step was to convert the image to Lab colour space, that is because it is the most exact means of representing colours, and can make otherwise subtle differences in colour more apparent. For now I will not be making use of the L channel, representing lightness of each colour, but will go into further detail on what it may be used for in section 4.2. With the image now stored as a 3d uint8 array of dimensions (x,y,3), I separated it in to its 3 component layers (representing the L, A and B channels). I then wrote code that would iterate through each tile and retrieve their average A and B values using the centroid coordinates found in the previous section. I used this on a range of possible input images to then define typical A and B channel values for each of the 5 colours, to be used as points of reference soon.

Looking for exact matches between colours would not have constituted a robust system due to all the potential external variables at play, and so I assigned a colour to each tile based on their Euclidean distances from those pre-defined typical values.

3. Results

Of the 30 simulated images, my implementation was able to return a result for 22 of them, giving it a successful run rate of 73%. The vast majority of those that were unsuccessful were examples of the *proj_* images, which appear to have a different sort of projection warp to the *proj1_* images for example, as these seem to be stretched only on a 2d plane as opposed to a more complex 3d one. Below you can see the confusion matrix for all returned results:

		Actual Colour				
		White	**Blue**	**Yellow**	**Green**	**Red**
Predicted Colour	**White**	68	14	9	10	12
	Blue	2	40	7	4	2
	Yellow	2	2	49	2	4
	Green	5	6	10	49	2
	Red	1	1	2	1	49

As you can see, my implementation achieved a respectable degree of accuracy, correctly classifying 72% of tiles. This wasn't an even distribution however, whilst it achieved 100% accuracy on the *org_* images, and nearing that on the *rot_* images, it suffered much more on *proj2_* for example, as slightly different shades of the colours are used which led to many being misclassified as white. This highlights one key shortcoming of this system, the manor in which colours are assigned based on comparisons to pre-defined 'typical' values, naturally these values could be tweaked to suit any particular colour palette, and you could only achieve a certain level of success using one set of fixed values. In section 4.2 I outline an alternative approach that I anticipate to yield better results, however I deem this one to be sufficient for the scope of this task.

4. Discussion

4.1 Image Cropping

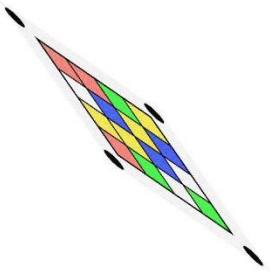

Figure 6.1 - Skewed Target Image

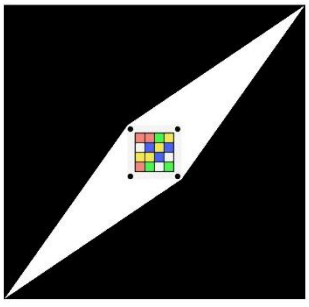

Figure 6.1 - The Same Image after Transformation

Within my submitted code file, you may notice a currently unused function called *findCrop,* this is because my solution described in section 2.2 for locating the tiles was not my first approach and I was initially aiming to transform every image in to a form identical to that in Figure 1.1 so that the tiles would be at the same co-ordinates allowing me to hardcode them in. I failed to foresee the transformation resulting in Figure 6.1 due to the white background, hence my decision to create this function. It worked by finding the centroids much the same way as described in section 2.1, to then use the upper-left one

as a point of reference about which to crop the image. It achieved good results on very skewed images such as that in Figure 6.1, however the system became more temperamental on those images with less skew. Since cropping the images didn't provide any advantage in terms of colour identification, I decided to abandon this approach and find an alternative method to locate the tiles.

4.2 Live Images

After converting our images to LAB colour space and separating it in to its L, A and B components, I mentioned in section 2.3 that I had no use for the L component. Whilst the A channel represents a scale from red to green, and the B value from blue to yellow, the L is the lightness/darkness. This is one key difference in how humans and computers interpret colours – if a human saw a shade of blue inside and outside of a shadow they would see them as the same colour, whereas computers would not. Since our images contain different lightness's of blue that we don't want to distinguish between, I chose not to incorporate the L component of the image during the colour identification. I do anticipate that careful and correct incorporation of this layer into the decision-making process may aid in the disambiguation between the Blue and White tiles that are occasionally confused for one another, but this remains a point of further investigation I hope to pursue.

If we were to extend our system to work on the live photo image set, I believe we may then have to incorporate the L layer during the identification process. Additional changes would also have to be made to the way that colours are assigned to tiles, rather than taking Euclidian distances from fixed 'typical' values, a more relative approach would likely need to be taken. For example blue tiles are likely to have the lowest B value of those present, reds the highest A etc. Alternatively, machine learning approaches would find this task rather trivial – given sufficient labelled training data, a classifier could be trained that takes all 3 channels as inputs and outputs one of the 5 possible colours. This method is certainly the one I would choose if I were tasked to build an implementation that worked reliably with real-life photos, as similar photos could be used to create the training data, making the model very well suited to the task.

5. Appendix

```
clear all
close all


%colourMatrix('SimulatedImages/org_1.png', false, false)
colourMatrix('SimulatedImages/noise_3.png', false, true)
%colourMatrix('SimulatedImages/proj2_1.png', false, false)
%colourMatrix('SimulatedImages/rot_1.png', false, false)

function [results] = colourMatrix(filename, displaySteps, noisy)

    %----------------Read in image--------------------
    img = imread(filename);
    figure('Name', ' Original Image'), imshow(img)
    fixedPoints = [26.5 26.5; 26.5 445.5; 445.5 26.5; 445.5 445.5];

    %========================== TRANSFORMATION ==========================

    %----------------Convert to B+W--------------------
    BW = ~im2bw(img, 0.5); %invert
    if displaySteps
        figure('Name', 'inv B&W'), imshow(BW)
    end

    %------------------Errode----------------------------
    %se = strel('disk',2); %create strel obj to erode with
    %erodeBW = imerode(BW, se); %erode to clear circles
    %imshow(erodeBW);

    %----------------Fill holes-------------------------
    filled = imfill(BW,'holes');
    if displaySteps
        figure('Name', 'Filled'), imshow(filled);
    end

    %---------------Identify Cirlces--------------------
    CC=bwconncomp(BW); %Create data struct, used to be applied to filled
    s = regionprops(filled,'Eccentricity', 'Centroid', 'Area'); %ID circles
    maximum = max([s.Area]);
    %Remove max area as this is the square
    for x = 1:size(s,1)
        if s(x).Area == maximum
            s(x) = [];
            break
        end
    end

    %--------------Retrieve Centroids-----------------
    centroids = [s.Centroid];
    centres = vec2mat(centroids,2); %can reduce to 1 line
    movingPoints = centres;

    %------------Create Transformation Matrix---------
    mytform = fitgeotrans(movingPoints,fixedPoints, 'projective');
    mytform.T;

    %-----------Apply to original--------------------
    %figure(1), imshow(img)
    imgTrans=imwarp(img,mytform);
    %figure(2)
    if displaySteps
        figure('Name', 'Transformed'), imshow(imgTrans)
    end

    %-----------------Crop---------------------
    % ctrs = findCrop(imgTrans);
    % imgCropped = imcrop(imgTrans,[ctrs(1) ctrs(2) 420 420]);
```

```
% figure('Name', 'Cropped'), imshow(imgCropped)



%========================== LOCATE TILES ==========================

%----------------Convert to B+W--------------------
if noisy %Adjust for noisy images
    threshold = 0.4;
else
    threshold = 0.1;
end
BW = im2bw(imgTrans, threshold); %Convert to B&W around threshold
if displaySteps
    figure('Name', 'Transformed B&W'), imshow(BW)
end

%------------------Dilate-----------------------------
% se = strel('disk',1); %create strel obj to dilate with
% dilateBW = imdilate(BW, se); %erode to clear circles
 %figure('Name', 'Dilated'), imshow(dilateBW);

%------------------Errode-----------------------------
se = strel('square',5); %create strel obj to erode with (set to 10 for noisy imgs)
erodeBW = imerode(BW, se); %erode to clear circles
if displaySteps
    figure('Name', 'Erroded'), imshow(erodeBW);
end

%---------------Identify Tile Centroids----------------------
t = regionprops(erodeBW, 'Centroid', 'Area'); %ID circles
med = median([t.Area]); %Average not suitable
margin = med * 0.1;
%Remove all objects with areas 10% from median
x = 1;
while size(t,1) > 16
    if t(x).Area < med-margin || t(x).Area > med+margin
        t(x) = [];
    else
        x = x + 1; %only increment if entry not removed
    end
end



%======================== COLOUR IDENTIFICATION ========================

%------------------------Lab values of colours---------------------------
white = [128,128]; %128,128
blue = [145,95];
yellow = [116,195];
green = [65,180];
red = [170,150];
colours = [white; blue; yellow; green; red]; %colour index = position

%----------------------Median filter rgb channels-----------------------
C = makecform('srgb2lab');
medianFilt = imgTrans;
medianFilt(:,:,1) = medfilt2(medianFilt(:,:,1), [9 9]);
medianFilt(:,:,2) = medfilt2(medianFilt(:,:,2), [9 9]);
medianFilt(:,:,3) = medfilt2(medianFilt(:,:,3), [9 9]);

%---------------------------Lab colour space---------------------------
imgLab = applycform(medianFilt, C); % convert to lab
if displaySteps
    figure('Name', 'Median'), imshow(medianFilt);
    figure('Name', 'Median+Lab'), imshow(imgLab);
end

%--------------Seperate img in to seperate component layers-----------
L = imgLab(:,:,1); %L = lightness, comment on live imgs?
a = imgLab(:,:,2); %A = Red/Green val
b = imgLab(:,:,3); %B = Blue/Yello val
```

```
    %----------Create Datastructures to store tile averages-------------
    aGrid = [];
    bGrid = [];
    lGrid = [];
    colGrid = [];

    %----------------------Iterate over each tile----------------------
    for i = 1:16
        tileDim = sqrt(t(i).Area) * 0.7; %Area to be averaged is 70% of the tile
        centre = round(t(i).Centroid);

        %Identify 3 corners of tile area to be averaged, relative to centre
        A = centre + [round((-1/2) * tileDim), round((-1/2) * tileDim)]; %Top left corner
        B = centre + [round((1/2) * tileDim), round((-1/2) * tileDim)]; %Bottom left
        C = centre + [round((-1/2) * tileDim), round((1/2) * tileDim)]; %Top right

        %Calc avg L, A and B values within each tile
        avgL = mean(mean(L(A(1):B(1), A(2):C(2))));
        avgA = mean(mean(a(A(1):B(1), A(2):C(2))));
        avgB = mean(mean(b(A(1):B(1), A(2):C(2))));
        avs = [avgA, avgB];

        %Collect Values
        lGrid = [lGrid, avgL];
        aGrid = [aGrid, avgA];
        bGrid = [bGrid, avgB];

        %Minimum Euclidean distance to org colours
        [~,colcode] = min(sqrt(sum((colours-avs).^2,2)));
        colGrid = [colGrid, colcode];
    end

    %----------------------Reformat to 4x4 Grid-----------------------------
    %lGrid = reshape(lGrid,[4,4]);
    aGrid = reshape(aGrid,[4,4]);
    bGrid = reshape(bGrid,[4,4]);
    colGrid = reshape(colGrid, [4,4]);
    %disp(colGrid');

    %-----------------Translate Colour Indexes to String--------------------
    results = strings(4,4);
    for i = 1:4
        for j = 1:4
            switch colGrid(i,j)
                case 1
                    results(i,j) = 'White';
                case 2
                    results(i,j) = 'Blue';
                case 3
                    results(i,j) = 'Yellow';
                case 4
                    results(i,j) = 'Green';
                case 5
                    results(i,j) = 'Red';
            end
        end
    end
    %Transpose to match original
    results = results';
    %Output Results
    disp(results);
end




%Used to locate top left centroid to use as crop reference
function [centroids] = findCrop(img)
    %figure(1), imshow(img);
    maximum = 0;
    %---------------Convert to B+W-----------
    BW = ~im2bw(img, 0.5); %invert
    %figure(2), imshow(BW);
```

```
    %----------------Fill holes---------------
    filled = imfill(BW,'holes');
    %figure(3), imshow(filled);
    %---------------Identify Cirlces--------------------
    CC=bwconncomp(BW); %Create data struct
    info = regionprops(BW,'Eccentricity', 'Centroid', 'Area'); %ID circles
    %disp(s.Area);
    areas = info.Area;
    disp(areas);
    maximum = max(areas);
    %Remove max area as this is the square
    for x = 1:size(info,1)
        if info(x).Area == maximum
            info(x) = [];
            break
        end
    end
    %--------------Retrieve Centroids------------------
    centroids = vec2mat([info.Centroid],2);
    %centroids = centroids(1,:);
end
```