UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

Automatic Tire-markings Detection & Recognition

Robert-Mihai Lică

Thesis advisor:

Conf. dr. ing. Iuliu Vasilescu

Conf. dr. ing. Daniel Rosner

BUCHAREST

2022

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

Detectarea și Recunoașterea Automată a Marcajelor de pe Cauciucuri

Robert-Mihai Lică

Coordonator științific:

Conf. dr. ing. Iuliu Vasilescu

Conf. dr. ing. Daniel Rosner

BUCUREȘTI
2022

CONTENTS

ABSTRACT

This work shall provide an image processing pipeline for automatic detection and recognition of tire-markings from images. Because tires can be one of the most expensive consumables of a truck, they are of interest to fleet managers who want to monitor their placement, parameters and possible illegal swap. My solution shall provide an easy way to extract from an image the serial code and/or certification number in order to aid an individual to almost uniquely identify a tire. The system can identify a wheel in an image and unwraps it. Then, it detects the text regions on the tire by using a band-pass filter on the frequency domain of the image, and lastly Microsoft Cognitive Services' OCR is employed to perform the character recognition.

SINOPSIS

Această lucreare o să ofere o bandă de procesare a imaginilor pentru detectarea automată și recunoașterea marcajelor de pe marginea unui cauciuc. Deoarece cauciucurile pot fi unul dintre cele mai scumpe consumabile ale unui tir, este în interesul managerilor de flote să monitorizeze utilizarea cauciucurilor, parametrii lor și eventuale schimbări ilegale. Soluția mea va oferi o metodă ușoară de a extrage din imagini codul de serie și/sau numărul certificării pentru a identifica aproape unic un cauciuc. Sistemul poate să identifice o roată într-o imagine și să o desfășoare. Apoi, aplică un filtru bandă pe domeniul de frecvență a imaginii și OCR-ul de la "Microsoft Cognitive Services" este utilizat pentru a efectua recunoașterea caracterelor.

1 INTRODUCTION

1.1 Motivation

In an ideal world, tires would not be that expensive, especially in the truck market, and their theft or illegal swapping wouldn't be that profitable. Because we do not pay much attention when it comes to our tires, aside from: in the winter to swap them with the winter ones and in summer put the summer ones, if somebody was to change our tires for other similar looking ones (just the same color is enough) the majority of the people would not even notice. You would have to keep track of what tires you are using, probably mark down their serial number and certification number (if present) and from time to time check that the ones on your car or trucks have still the same information.

The tire-markings consists of embossed letters on the side of tires and are put by manufacturers to represent the characteristics of the tire, show its certification and to distinguish between different production batches. A common serial code found on a tire is the DOT code (Figure 1a), the acronym meaning "The Department of Transportation". It is a marking that is mandated by the United States of America to be present on all the tires that are commercialized in the country and is as close as possible to a serial number. Because having different production lines is expensive, manufacturers print this DOT code also on tires that are sold in other regions of the world. This code is usually composed of: DOT marking (Figure 1a – i), tire manufacturer or manufacturing plant code (Figure 1a – ii), the size code (Figure 1a – iii), tire manufacturer (Figure 1a – iv) and finally the week and the year the tire was produced in (Figure 1a – v). Unfortunately, some of these groups of letters are sometimes missing. Anyway, this code is not enough to uniquely identify a tire as the most unique part is the date which has weekly increments only.

Another code present on the tire is the E-mark used in Europe (Figure 1b) to mark the certification which the respective tire follows and complies with. It consists of a circle with letter 'e' or 'E' followed by a number (representing the country who issued the approval that the tire meets the certification) inside a circle and next to the circle is a code or 2 lines of code representing the certification itself that the tire complies with.

The problem of tire theft and illegal swapping is more predominant in the truck market as there tires are more expensive and wear out faster. For a truck fleet owner this is a big problem because swapped lower quality tires can be a road hazard and in case of failure they can have catastrophic results. The solution at the moment is a labor intensive, slow and prone to human error or ill will: a person, that can be bribed, has to manually write down the

information after buying a new tire and at some intervals of time it has to manually check again that the tire on the truck is still the same one, with the same specifications, and that nobody illegally swapped it for an older or lower quality one. In the case of tire theft, there isn't much to do as there is no way to determine where the tires end up utilized. If we were to apply the same solution to tire theft – as we apply to tire illegal swapping – we would need people that would inspect at the tires passing on a road, mark down the information and enter it in a database. If the respective tire was declared stolen, there would be at least a starting point in the theft investigation.

But this human inspection approach doesn't scale when it comes to hundreds of millions of vehicles. So, this is a great task for automation. It could be replicated indefinitely and the limiting factor would only be the hardware required. An automatic system for collecting information from the side of tires and reporting to a database that the serial number was seen in a particular location at a certain moment would help in identifying stolen tires that are put again in use on the roads. Or can check that the tires on a truck have the serial numbers that they should have and were not swapped.

1.2 Problem Statement

To further create a system that is able to almost uniquely recognize a tire, firstly it is needed to extract the markings from the image of a wheel. These are under the form of embossed letters on the side of the tire. They are black in color just as the background, so the contrast is not great. The tire-markings that are quite unique and of interest to us are the DOT serial code (Figure 1a) and E-mark certification number (Figure 1b). The manufacturer name(Figure 1c), maximum ratings (Figure 1d), construction materials, the ISO metric tire codes (Figure 1e) and other supplementary markings only contain information to the physical characteristics of the tire itself and would be nice to obtain, but not necessary.

At the moment, such recognition systems require extra equipment to create special environment conditions [8] or are computationally intensive and require processing on an external server [2]. My solution is to create a more robust image processing pipeline, that would not require supplementary incident lights for better letter contrast with the background. It would also be best to move as much as possible of the processing in situ to be as independents as possible from a central server.

1.3 Overview of the Solution

My solution is to create a more scalable and easier to deploy image processing pipeline which would not require supplementary incident lights (that provide better letter contrast with the background) and would do as much processing as possible in situ. All of this comes in the

Figure 1: Tire-markings: a) DOT code, b) E-mark, c) Manufacturer, d) Maximum ratings, e) ISO metric tire codes

Figure 2: Dataset entry example

Figure 3: DOT marking with incident sunlight

detriment of precision, the pipeline having a text detection rate of 0.76 but a precision when it comes to detecting only the DOT code and E-mark of 0.07 (more details in chapter 5). In my image processing pipeline I will be feeding photos of car tires captured with a Cannon EOS 1300D with a resolution of 5184 by 3456 pixels which will be cropped around the center to have a resolution of 3456 by 3456 pixels. The images were taken from approximately 115 centimeters and the hole wheel was always in the shot (Figure 2). With this resolution and distance from the captured object, the characters composing the serial number have around 60 by 20 pixels in size, so not very much. I divided my pipeline in 3 big stages that tackle the problem sequentially: tire unwrapping, text detection, and text recognition.

a) Tire Unwrapping

Consists of the process of determining where in the image the tire is situated, detecting its outer edge, inner edge and the center of the tire, and converting the circular shape of the tire in a rectangular one (unwrapping). In Figure 2 we can see the image of a tire which will have its circles detected. The circles are used to obtain an unwrapped image of the tire (Figure 15). In the unwrapped version it can be seen that the tire is not perfectly straight because

of the perspective of the captured image. If the camera was not perpendicular on the tire's plane and in line with the wheel's axle, the tire has a slight oval shape that is accentuated by the unwrapping as an oval has 2 centers and not only one.

b) Text Detection

The scope of this stage, is to determine the regions where text might be present. Because the final stage, text recognition, can be one of the most computationally intensive parts and the image's pixel count is still quite high after the unwrapping (4,804,166 pixels in average), we want to reduce the space where we try to recognize characters.

I opted this stage to not use machine learning and pre-trained models and instead to have a deterministic approach by using a combination of processing techniques and greedy components detection. My approach was to pass the image in the frequency domain in order to remove the high frequencies from the image – as those tend to represent noise – and the low frequencies – who usually represent the uniform background color of the tire, leaving the markings behind. This was difficult to accomplish as the tire-markings – being embossed letters – are not very prominent compared to the background of the tire (for example Figure 1a) without controlling lighting conditions. I needed through multiple tests to come up with a series of heuristics to propose the areas that contain text. At this stage I accepted to allow a higher number of false positives in the regions than a higher number of false negatives (who would have meant to miss on some text regions). It will be the task of the next stage to deal with the falsely voted regions of text. In the end, I output a binary image (Figure 27) with the same size as the unwrapped one. The white pixels represent supposed text area that the next stage should attempt to recognize.

c) Text Recognition

This is the stage where the information is practically extracted from the image. So far I have only prepared the image or selected the regions of interest. Taking into consideration the hardships in extracting the letters' regions in the previous stage, I opted to use a pre-trained OCR algorithm in order for the pipeline to be more robust and creating an OCR from scratch was not the scope of this project. The OCR used is part of the solutions suite of Microsoft Cognitive Services [11]. Applied on the found regions of interest, it gave a character error rate of 0.26 when the image was not blurry and could recognize at least a few letters.

1.4 The Document's Structure

In this chapter I presented the problem as a hole and then focused on the problem of automatically recognizing tire-markings. I showed what a solution to this problem would require

and my approach to solving it.

The next chapter – 2. State of the Art – will present other work in the field, what their approaches were, their setups and results and how I am bringing new contribution to the field.

In chapter 3. The Proposed Solution I will go in depth in the image processing pipeline. I will present each stage, its steps, sub-steps and each action performed. I will provide information on each action in a consistent pattern: the output, how it's done – the algorithm in pseudo-code and its explanation – and optionally other approaches that didn't work and motivated me to chose this action in the end.

In chapter 4. Implementation Details I will present and explain the code, the libraries used and how the data is managed. When is the case, I will motivate the choice of an implementation over another one.

The overall performance of the system is outlined in chapter 5. Evaluation alongside the results of other works in this field. The dataset collection details and the system on which the pipeline was run on are also presented.

The final chapter – 6. Conclusions – will present an overview of this work, the novelty of the approach in comparison with other contributions in the field and possible future improvements.

2 STATE OF THE ART

In designing my solution I've taken into account other work performed in the field of automatically recognizing tire-markings. The difficulty of the task is twofold. On one hand, the markings on tires' sides are rarely printed because they would fade in time or under the effects of the elements. Instead, the approach employed by manufacturers is to have embossed letters on the side of the tire that would fade slower than their painted counter parts. They are not foolproof either as they can also fade in time as the letters dull and become indistinguishable from the normal side of the tire, but are usually enough to outlive the tire grooves that dull the first because of the tire's usage. One characteristic of these embossed letters is that they are practically rubber on rubber, so black on black (Figure 1a). Their visibility is not great, even with the human eye can be hard to distinguish the letters in some lighting conditions. For the human eye, it can be beneficial to have an incident source of light that would make the letters cast a shadow on the side of the tire like can be seen in Figure 3 and a person could accomplish this with a flashlight that he positions at an optimal angle.

This was also the approach of Wajahat Kazmi et alia in their work [8]. They had a 2 camera setup that each would capture half of the wheel and a supplementary source of "Strobe light incident at steep angles with respect to the plane of the sidewall" to quote them. This arrangement would help with the resolution of the images and character detection in the later stages of their pipeline. As a first step, they also performed an unwrapping of the tire using Circular Hough Transform [19]. After, they focused on detecting only the DOT code by using their crafted features in order to keep a low memory footprint (the average sizes of their images were 500x2800 pixels) and extracted a Histogram of Oriented Gradients. This output they would feed in a Convolutional Neural Network based Multi-Layered Perceptron and would have as output regions of the image where text is present. The training was done with a synthetic data-set. Then, they would localize in the proposed areas the DOT code by using a deep neural network trained on a synthetic data-set of DOT foregrounds and different tire backgrounds. The same model was used also for character recognition and was trained on a 700,000 synthetic data-set of characters on black background to mimic the low contrast appearance of the embossed markings. They claim their pipeline obtained an accuracy of 80% in images that are considered acceptable to human standards, but make a note that an objective benchmark is not available. The lower quality images that would pose difficulties even to a human to recognize the text obtained an accuracy raging from 73% to 14%. On the 9 images they tested their pipeline on, they obtained a character error rate of 0.23 while I obtained a character error rate of 0.26 on the images which were not blurry and 0.51 on my dataset. Because of the inconsistency in bench-marking the accuracy of such a system, it would be beneficial to make dataset I collected public for the benefit of the community.

While the past paper's goal was for an industrial system that would have controlled conditions when performing the tire-markings recognition, there is also work in the field, by Anton Katanaev et alia [2] for a consumer solution that would try to extract the tire specifications in order to help with ordering new ones. Their goal is to obtain the "ISO metric tire codes" that specify the type of tire, the width, aspect ratio, construction, diameter, load index and speed rating. Their first step was to collect a data-set of tire images by using internet scrappers and then filter through them using a classification model based off ResNet64. Compared to the previous work, Anton Katanaev et alia found the Circular Hough Transform unsuitable because of parameter tuning and opted for a segmentation approach for detecting the tire from the background. After this, they focused on image preprocessing to combat the different illumination in the images collected in the data-set and also perform circle correction for when the tires were appearing with an oval shape because of the camera angle. To reduce the space in which to perform the character recognition, the team also employed a step in which regions of text are searched in the image. They compared 14 pre-trained text detection models and chose the best performing one to provide supposed regions of text. After this, the proposed regions are fed into the best performing text recognition model they've tried: SEG OCR. This model obtained a character error rate of 0.16 on the original images, performing worst on the ones where the contrast was adjusted. As a final step to combat errors that might appear in the character recognition part and to identify a tire by its properties, they used a database of more than 15,000 tire characteristics for correcting incomplete or erroneous data. Because of the many learning models used, the processing part of their application is in cloud (they leverage the computational power of AWS Cloud). On the user's end is a smartphone application to submit the captured tire image to the server to start processing and provide suitable tire candidates.

In the Problem Statement I stated the desire of a system that would not require complex setups, a controlled light environment and that would be nice to be as much as possible self contained (to not rely completely on a server for processing). This is why in my approach I preferred deterministic mathematical algorithms over learning models to accomplish the task and provide a robust system which can perform in natural lighting conditions. I consider I succeeded in doing this as I am able to obtain the DOT code and the E-mark in 61.74% of the images and the character error rate of the pipeline is just 0.26 on the non blury images.

3 THE PROPOSED SOLUTION

The solution is an image processing pipeline that would extract from the side picture of a tire the markings written on it. These are present in the form of embossed letters on the tire's exterior walls. By desiring to not have a complex setup and not require supplementary light sources, the system was tested on images taken only with ambient light. This proved to increase the problem's difficulty quite considerably when it came to detecting the regions of text, because the contrast between the embossed letters and the background image got even smaller. Multiple approaches were considered in section 3.2 Text Detection and in the end one proved fruitful in delivering acceptable results.

The system's stages are described in the following sections:

3.1 Tire Unwrapping (Stage 1)

As a first stage, I wanted to extract only the tire from the input image. This would help because the information I am interested in is located on it. I would also like to unwrap the tire from its disk shape (a circle with a smaller circle as a hole inside of it) into a rectangular one in order to have the writing from left to right rather than in all directions like if it was in a circle.

To better explain how this was accomplished, this stage was split in multiple steps:

3.1.1 Circle Detection

I detect where in the image are the circles representing the outer and inner borders of the tire and get their center's coordinates. The two detected circles might not have a common center because of the imperfections in taking the images, but I account and try to correct for that through a series of heuristics.

Because this is the beginning of the pipeline, I introduced a step to equalization the images and have some independence from the lighting conditions. Then I detected the first circles in the image using Hough Circle Transform and filtered its output through a series of heuristics.

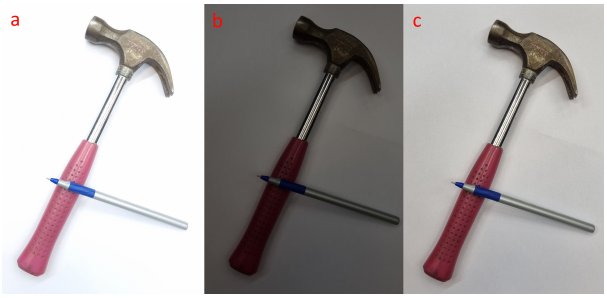

Figure 4: Equalization demonstration: a) over-exposed image, b) underexposed image, c) equalized image

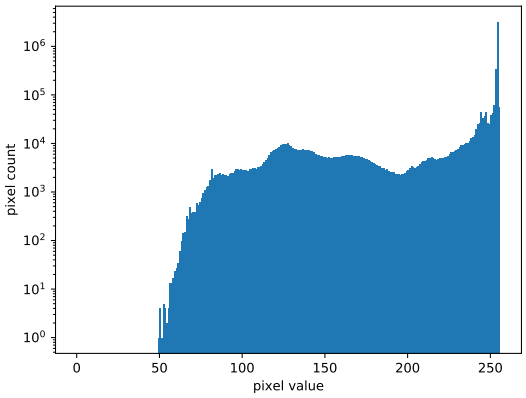

Figure 5: Hist. of Fig. 4a

a) Equalization

Accomplishes:

Lighting conditions are not controlled and this might result in low contrast in some areas of the image. Here we ensure a constant contrast across the entire image and that the entire histogram is used. This increases the robustness of the overall system.

Reasoning:

The pixel in a black and white image is a representation of how bright that particular part of the image should be (0 is for no light and 255 is for maximum brightness on a 8 bit sensor). If we take a photo in broad daylight, the values of all the pixels could all be over 200 lets say. The image would appear as being washed-out and lacking details like Figure 4a and would have a histogram looking like Figure 5. If the photo is taken in a dark environment and all pixel values are under 100, the image appears again quite dark and lacking detail, this is the case for Figure 4b whose histogram is drawn in Figure 6. A solution is to use the space that remained unused in pixel values and stretch the existing pixel values to also cover those, thus increasing the contrast of the image and its level of detail. This action is called equalization and can be used to enhance the contrast in an image, like can be seen in Figure 4c. Furthermore, it spreads the pixel values across their entire value range so that a under-light and an over-light picture's histogram would look similar (Figure 7). The histogram is a plot that has on the X axis all possible pixel values and on the Y axis the number of times that value appears in an image and usually is on a logarithmic scale.

A deviation from this standard equalization is Adaptive Histogram Equalization (AHE) that is not applied on the entire image and just on a small portion of it. When calculating the new value for a pixel, only a neighborhood around it is taken into consideration. Thus method while useful in bringing more contrast in to an image that has lighter and darker regions, it also increases the noise in the image. To counteract this, Contrast Limited AHE (CLAHE)

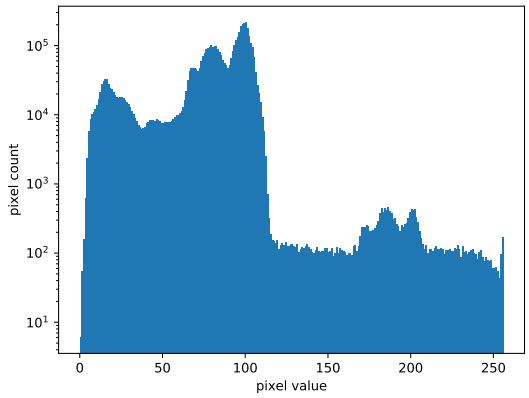

Figure 6: Hist. of Fig. 4b

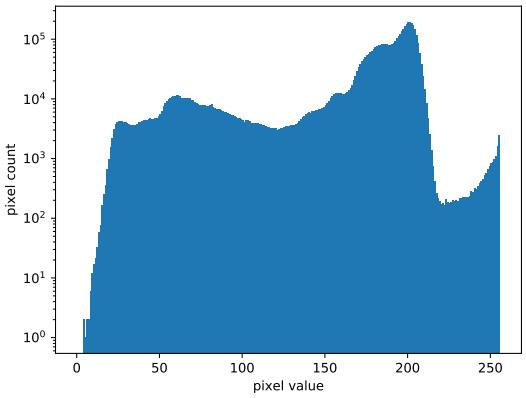

Figure 7: Hist. of Fig. 4c

was created. The addition is that it has a clipping upper limit and redistributes the pixels that appear too often in the image into another ranges.

Algorithm:

Algorithm 1 CLAHE

Require: img be a $X \times X$ matrix; $cl \in [0, 1]$; k

Ensure: $k \mid X$

1: $S \leftarrow$ (img split into $k \times k$ cells)
2: **for** s in S **do**
3: $\quad H_s \leftarrow histogram(s)$
4: $\quad H_{res}, N_{remain} \leftarrow clip(H_s, cl)$
5: $\quad H_{res} \leftarrow redistribute(H_{res}, N_{remain})$
6: $\quad s \leftarrow adjust\_cell(s, H_{res})$
7: **end for**
8: $img \leftarrow$ interpolate cells S

CLAHE is described in pseudo-code in Algorithm 1 in accordance to the implementation found in [26] and the explanation in the paper [10].

On line 1 we split the image in multiple cells of size k by k because CLAHE is an adaptive algorithm. Then, for each of the cells we calculate their histogram to find out how many different pixel values they contain (N_{gray}). To perform the clipping, we need to calculate a few things:

$N_{avg} \leftarrow (k \times k) \div N_{gray}$, that is the number of pixels in the cell divided by how many pixel values are present in the cell;

$N_{CL} \leftarrow cl \times N_{avg}$, the clipping limit calculated based on input parameter cl;

$N_{\Sigma\_clip} \leftarrow \sum_{i=0}^{N_{gray}} max(0, H_s[i] - N_{CL})$, is the number of pixels that need to be

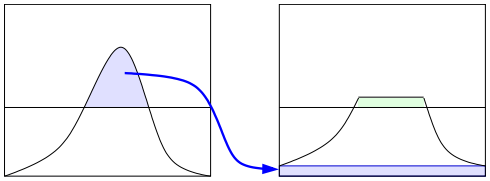

Figure 8: The histogram after first clipping, source: [24]

clipped, $H_s[i]$ is the pixel count for a certain pixel value;

$N_{avg\_gray} \leftarrow N_{\Sigma\_clip} \div N_{gray}$, is how many pixels in average we will modify in addition to have a certain value.

Then, we iterate through all the pixel values in the histogram and create a new one. If the pixel count is $\geq$ than N_{CL}, we will clip that pixel value. Else, we will add to the pixel count N_{avg_gray} and if we exceed the clipping limit, we floor to it and take note of this in the form of N_remain. After this step, the histogram would look like in figure 8.

The limit we excede with will be redistributed after this on line 5. Here, we go through the pixel values in the histogram again and at a simmilary random step we add one more pixel to the count that overflowed earlyer. The step is computed by the formula: $step \leftarrow N_{gray} \div N_{remain}$.

Before starting to compute the next cell, we adjust the current one acoording to the new histogram we just computed: H_{res}.

After all the cels were modified and equalized, we can stisch them back together. But because we had a cell by cell approach to calculating the histogram, border pixels might have very different values. This is why an extra step of bilinear interpolation is performed on the border pizels of each of the cells.

Results:

If before the equalization the histogram of the image looked like Figure 9, after the equalization it looks like Figure 10. It can be observed it is smoother and part of the pixels were brightened.

b) Hough Circles Transform (HCT)

Accomplishes:

It detects circular shapes present in an image. The ones we are interested in are the outer and inner rims of the tire.

Reasoning:

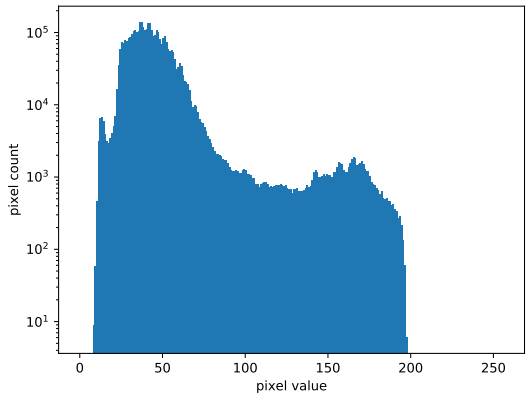

Figure 9: Histogram without equalization

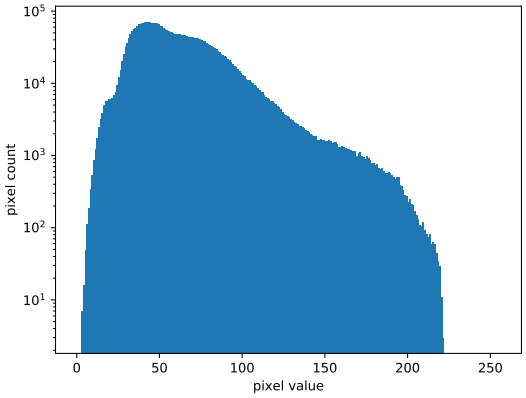

Figure 10: Histogram with equalization

The reasoning for why we want to detect the tire in the image is twofold. Firstly, the text searching part is a computational expensive one and we want to reduce the search space from the whole 3456 by 3456 pixels to usually a forth of that. The second reason is that the text is present in the image in a circular form, while the vast majority of optical character recognition (OCR) software is performing best on text that is aligned along one dimension only. So, by detecting the tire's circular shape in the image, we can transform the tire into its unwrapped version (that also has a smaller pixel count).

Algorithm:

Algorithm 2 Hough Circles Transform

Require: img; r_{min}; r_{max}; dp; $canny\_params$

$blur \leftarrow gaussian\_blur(img)$
$gray \leftarrow color2gray(blur)$
$edges \leftarrow canny(gray, canny\_params)$
for $r = r_{min} : +1 : r_{max}$ **do**
    $acc[i] \leftarrow matrix(shape(img) \div dp)$
    for each pixel $p > 0$ in $edges$ **do**
        $acc[i] \leftarrow accumulate(acc[i], p, r)$
    end for
end for
$center, radius \leftarrow extract(acc)$
return $(center, radius)$

Hough Circles Transform is particularization of the classical Hough Transform [19] and its scope is to extract features from images. The original algorithm is searching for a circle with a known radius, but as can be seen in Algorithm 2, it can be generalized to an interval of radii.

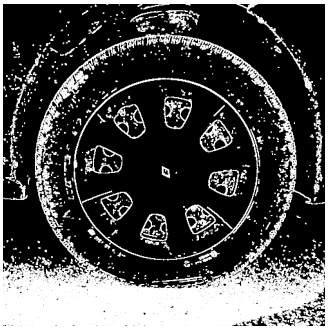

Figure 11: Canny output

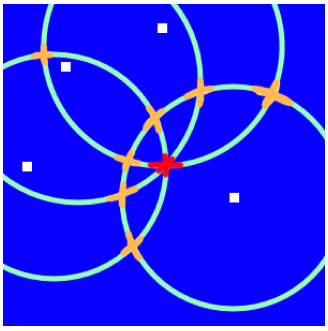

Figure 12: HCT accumulator

Figure 13: HCT result

The first 2 actions in the algorithm are to apply Gaussian Blur [25] (to remove the noise that follows a Gaussian distribution) and convert the color image into gray-scale.

Then it uses Canny Edge Detector [22] to obtain a new image which contains only the outlines where the algorithm considered a boundary between 2 objects exists. An example can be seen in Figure 11, where each white pixel represents an edge.

On line 4, it can be seen the $loop$ that will probe for the radius interval that was requested.

For each radius that the algorithm will probe, will create an accumulator matrix that is dp times smaller than the original image. The purpose of this matrix will be explained shortly.

The central part of the algorithm is the for loop on line 6. It goes through all the pixels that resulted in being edges and finds their corresponding position in the smaller $acc[i]$ matrix. Then, all around that point in the accumulator matrix at distance r, it increments the corresponding cell. Four probed pixels (the white squares) of this can be seen in in Figure 12 where it can be seen that when circles overlap the value in that cell increases.

Such an accumulator is created for each probed radius and it will result in a 3D space. So, the last action of the algorithm (line 10) is to search in this 3D space of accumulators and find the most incremented cell (the red dot in Figure 12). That cell is considered the center of the circle and the radius is corresponding to the accumulator matrix it was found in.

Results:

Because tires have different dimensions and an outer and inner rim, I have to search for multiple circles in the image. So, I probed for multiple radii and the algorithm found multiple circles in the image, as can be seen in Figure 13. This will be the input for the next step, where through a series of heuristics the circle count will be reduced to only the most concentric ones and hopefully only 2.

c) Heuristics

Accomplishes:

Reduces the number of detected circles in the previous step (Figure 13) and remains only with 2 concentric circles representing the tire's inner and outer rims (Figure 14). If his step fails to remain with only 2 circles, then the image is discarded.

Reasoning:

There are many types of tires in the wild with different shapes and sizes, so it was better in the previous step to detect more circles in the image than to look for only 2 circles. When looking only for 2 circles using the Hough Circular Transform, the car's wheel arc can be falsely detected as a circle or the wheel cap can have a pattern on it that would falsely trick the circle detection algorithm. Because of this reason, I preferred to detect many false positives and at this stage filter through them and remain with only the mot concentric circles that have at least a certain radius difference. Another possible problem would be that the tire is not a perfect circle because of the contact point with the ground or because when taking the image, the camera sensor was not in line with the wheel's axle.

Algorithm:

The first heuristic I apply is to reduce the circles that are very similar one to another. I start from the most prominent circle found by the previous step. If the conditions $dist(c_1, c_2) < 0.5 \times size(img)$ and $\left|r_1 - r_2\right| < 0.37 * size(img)$ are found true, that means the circles are probably the same one and I calculate a mean one. The mean is calculated by taking the average of the centers and the average of the radii. In checking other circles, thisone will be used. The values for the thresholds were found experimentally.

The second heuristic's goal is to create 2 concentric circles. At first I was taking the biggest and smallest circles and using the average of their center's coordinates to calculate a new center. This gave unsatisfactory results as the computed center would be quite far from the true one and in the unwrapping phase, it would not make the tire a continuous horizontal strip. The improvement I ended up finding was that the smaller radius circle would have the center closer to the wheel's true center, so it would be better to use the inner circle as the base rather than to calculate a new center. The reason to why the smaller circle has a better center is that the inner circle usually represents the inner rim of the tire that is a perfect circle, while the outer rim is deformed at the contact point between the ground and the wheel. That deformity can trow off the center calculated at the previous step. If more than 2 circles are present, this step is skipped and the image will be discarded as we couldn't detect the inner and outer parts of the tire. An improvement that could be made is that if more circles are found, to select the ones that have the closest centers and a radius bigger than a certain value.

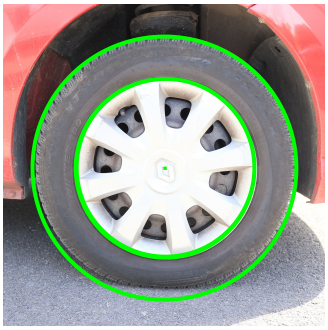

Figure 14: Filtered Concentric Circles Detected

Results:

The resulted concentric circles that can be seen in Figure 14 are an approximation of what the true circle might be, but it is close enough so that the tire can be unwrapped. The circle might not follow perfectly the rim of the tire because of the distortion of the wheel on the ground, or the distortion was introduced when taking the picture if the camera's sensor wasn't horizontal and in line with the wheel's axle. If we couldn't remain with 2 concentric circles at this step, the image is discarded. Otherwise, it is passed to the next step.

3.1.2 Convert to Polar Coordinates

Accomplishes:

This step unwraps the tire and outputs it as a continuous horizontal strip as can be seen in Figure 15. This can be done because we have detected the supposed center of the tire and its radius.

Reasoning:

The horizontal strip will help in text detection and recognition, as it is easier to work with text going in only one direction.

Algorithm:

A normal image representation is in the Cartesian domain. In this domain, a pixel is characterized as having 2 coordinates: x and y that we are familiar with. So a pixel is defined by 2 distances and the 2 distances have an unique pixel tied to them. Another domain to represent images in, is the Polar domain. In this domain, a pixel is characterized also with 2 coordinates: $angle$ and $radius$. The entire image is constructed around a center point and if information in the Cartesian domain was arranged in as the height of the image is the y and the width is the x, here they are the $angle$ and $radius$ respectively.

Because of this correlation, we can convert from one domain to another. We are

Figure 15: Successfully Unwrapped tire

Figure 16: Failed unwrapping of tire

interested now in converting from a Cartesian system into a Polar one as the tire has a circular shape. We can use the found circle center to make it the origin of the new Polar domain and start taking points from the image starting at the small radius until the big one. For each pixel in the image, now with the found circle's center as the image's one, we can find its new x and y coordinate and from them, we can calculate its representation as $angle$ and $radius$ using the following formulas.

$$radius = \sqrt{x^2 + y^2}$$

$$angle = \arctan(y/x)$$

There is a problem though. Closer we are to the new selected center, less points are present on the surrounding circle than on a bigger one. The resulting image after the Polar transformation will have as it's width the pixel count of the pixels that were on the most outer circle's perimeter. The points closer to the center must be stretched out in order to fill in the gaps and this is solved using interpolation. New computed pixels are used to fill in the gaps.

Results:

Multiple interpolation levels were used to observe if it affects in any way the output, but the letters on the tire are far away from the detected center around which the polar transform si applied that even a bilinear interpolation is good. Now, I have the unwrapped tire as a new image (Figure 15) and it is ready to be processed further in order to obtain the regions of text. It must be mentioned that some wheel centers are falsely recognized and after the polar transform an unusable image is generated, like can be seen in Figure 16. These are found and removed manualy and are not used in the next stages.

3.2 Text Detection (Stage 2)

Now that the initial image was processed and just the tire region was kept and unwrapped to have a rectangular form with the text spanning from left to right, the text detection phase can begin. The goal now is to obtain a bitmap of the rectangular tire with regions where text is present. This phase is useful because it will further reduce the space where we have to perform text recognition, and because the goal is to reduce, false-positives are permitted in an acceptable limit. If we can reduce the image search space by at least 80%, this stage is considered successful and the text recognition can begin. Otherwise the image is discarded.

3.2.1 Segmentation

Accomplishes:

Splits the image in smaller cells in order to perform operations on each one of them. These cells are overlapped a certain percentage to be able to vote on them regions of interest.

Reasoning:

While searching a way to obtain the text region in an image, frequency filtering proved useful and not needing parameter tuning. The less ideal part is that the last step is OTSU thresholding that is a global algorithm. While not performing useful if we were to process the entire image at the same time, it proved very helpful in binarizing smaller areas of the big image. So, I decided to split the original image in overlapping cells.

The cells overlap because in the end a voting process is employed to remove regions that were considered text in just a few of the segments.

Algorithm:

I need to calculate the corners of cells that have a certain overlap between them. This overlap is present on the x and y axes. Firstly, I calculate the number of cells that will be needed on each of the axes with the formula:

$$N = \frac{L - o}{no}$$

where L is the number of pixels, o is the size of the overlap region and no is the region of a cell that is not overlapped.

Results:

Now the big picture was split in multiple smaller overlapping cells (Figure 17) that will be further processed individually.

Figure 17: Segment

3.2.2 Segment Binarization

a) Image Frequency Filtering

Accomplishes:

This is a band-pass filter applied to the frequency domain of the segment. The remaining frequencies will consist of letter shapes (the tire-markings) and some artifacts.

Reasoning:

Because the background of the tire is fairly uniform in blackness and noise, these can be considered low frequency color variations and high frequency noise respectively. By eliminating certain bands from the frequency domain of the image, we are left with the letter shapes that could be processed further to enhance them and filter out any artifacts that remained.

Algorithm:

Algorithm 3 Band-pass Filtering

Require: img

1: $f\_rep \leftarrow FFT(img)$
2: $f\_rep \leftarrow bandpass\_filter(f\_rep, low, high)$
3: $img \leftarrow IFFT(f\_rep)$

Any analog signal is composed from multiple signals of different frequencies. These frequencies could be extracted using Fourier Transform [3] or reconstruct the signal using its inverse. Because it is a computationally expensive algorithm, in practice Fast Fourier Transform [3] is used because it usually gives acceptable results.

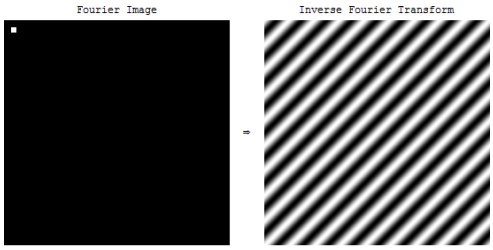

Figure 18: Frequency domain representation

This step can be described by Algorithm 3. But firstly, it must be made clear what passing an image into the frequency domain means. The easiest is to take a one dimensional example (a line) and then generalize for a two dimensional case (an image).

On the line, each pixel value could be considered to represent the amplitude of a signal. The signal progresses in time to one direction of the line. This signal can be decomposed in its core frequencies, to the left the low ones and the high ones to the right. The same thinking could be also applied to a 2D image, just that we get a 2D frequency image with real and imaginary components. For example, Figure 18 represents what happens if we have only a frequency present, what the resulting image would look like. The distance from the corner represents the frequency, while the position of the dot is the angle the bands will appear after applying the Inverse Fourier Transform.

To be easier to visualize the frequencies, the image is usually shifted so that the 0 frequency is in the middle of the image rather than in the corners like in Figure 19. This is useful also because it makes it easier to apply a band-pass filter. To apply this, we just have to blackout the middle of the image and the outer parts. In the end we are left with a disk shaped frequency representation of the image (Figure 20). Experimentally I got usable results by keeping a disk with the small radius 14% and the big radius 36% of the found frequencies in the image.

After applying the Inverse Fast Fourier Transform, we can observe that the low (the background) and the high (the noise) frequencies were removed. I successfully removed the background and the noise and still remained with the general letter outline in the image as can be seen in Figure 21 and 22.

The huge benefit of this method is that it can be applied to more images using the same band-pass filter, without changing the cutoffs.

Results:

Outputs an image like in Figure 21, where letter shapes could be observed. There are also artifacts remaining that will be dealt with at a later stage.

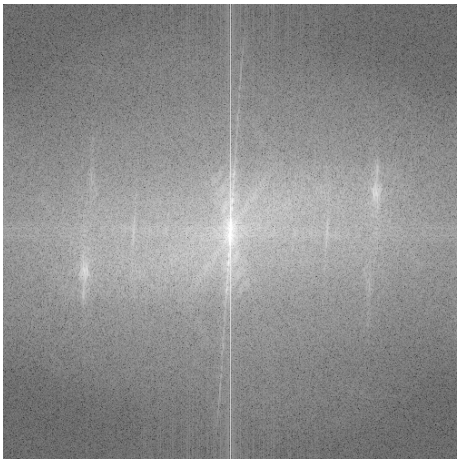

Figure 19: Frequency domain of Figure 17

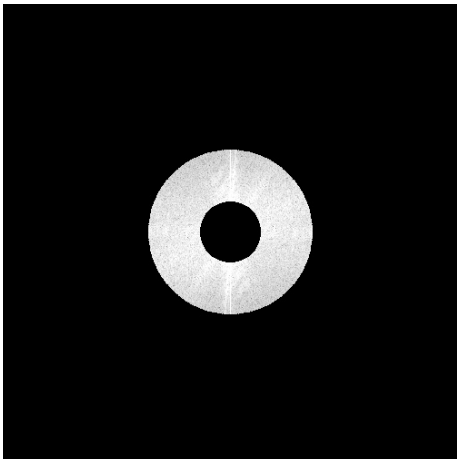

Figure 20: Filtered frequency domain of Figure 17

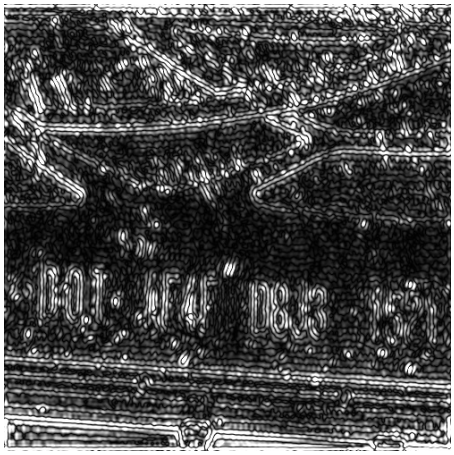

Figure 21: Filtered segment 17

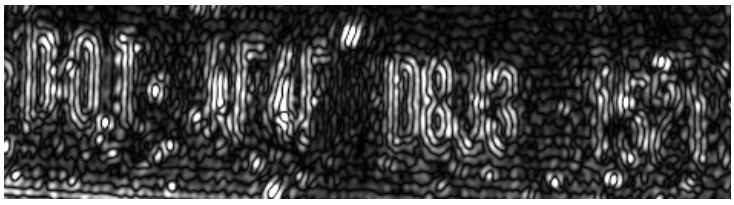

Figure 22: Zoom on text in filtered segment 21

b) Black Hat Morphological Operation

Accomplishes:

Enhances the darker regions representing the letter contours that are surrounded by brighter regions.

Reasoning:

It can be seen that in the resulted image at the previous step (Figure 21), the letters have an outline in dark pixels surrounded by bright regions. It can be more clearly seen in Figure 22. By extracting this outline of the letters, we will be able to have the letters more prominent in the image than other features.

Algorithm:

A morphological operation is based on mathematical morphology [5]. The main components to it are an object A (our image) and a structuring element B. Normally, these operations are applied on binary images but variations exist that let them to be used also on gray-scale ones. Firstly, a path must be created for our structuring element (B) to traverse the edge of the object (A). This path if formally defined as a function that maps points from our image to a real line.

$$f : E \to R$$

The Black Hat Morphological operation is defined mathematically as:

$$T_b(f) = f \bullet b - f$$

where b is a structuring element and $\bullet$ means the closing operation that is defined as a dilation followed by an erosion of a set A by a structuring element B:

$$A \bullet B = (A \oplus B) \ominus B$$

Results:

The result of this step can be seen in Figure 23. The figure has the brightness increased for viewing, but even if it's a darker image, it would not be a problem for a thresholding algorithm that can select his own threshold. This is why, in the next step we apply OTSU thresholding.

c) OTSU Thresholding

Accomplishes:

Figure 23: Blackhat applied on Figure 21

Computes automatically a threshold for the image and applies it, outputting a binary image. Because of the processing previously done, the output consists of tire-markings and some artifacts.

Reasoning:

When trying different methods to make the regions of interest (the tire-markings) remain in the image and remove the parts I am not interested in, OTSU Thresholding happened to do just that. Before the voting could begin, I anyway would have had to binarize the image so this is even more helpful. The fact that OTSU computes automatically its threshold is also very helpful as it prevents the need to experimentally fine tune the algorithm and perform overfitting. The output would have some artifacts that I removed as much as possible through a set of heuristics later.

Algorithm:

The novalty in the OTSU Thresholding (Algorithm 4) is that it computes the threshold by minimizing the intra-class variance (the variance between the white and black pixels that would result).

For each possible threshold value computes the binary image. After which the algorithm computes on the respective image the variance in white and black pixels and returns the weighted variance of the two. This can be seen on line 4 of the algorithm. We store the resulting image with the threshold and the weighted variance in 2 vectors (I and WV) that are self incrementing.

As a last step (line 6) we select the threshold that minimizes the weighted variance

Figure 24: Supposed text areas (OTSU applied on Figure 23)

and retuns the corresponding computed binary image with its corresponding threshold.

Algorithm 4 OTSU Thresholding

Require: img

1: **for** t in possible thresholds **do**
2: $img_{bin} \leftarrow threshold(img, t)$
3: $I[i] \leftarrow (img_{bin}, t)$
4: $WV[i] \leftarrow weighted_varianve(img_{bin})$
5: **end for**
6: $it \leftarrow pos\_of\_min(WV)$
7: **return** $I[it]$

Results:

It can be seen in Figure 24 that the letters or text remains almost as a connected component while the artifacts span across the entire segment or are small dots. This is very helpful as we can filter for the components that have a certain size and shape.

3.2.3 Filtering Artifacts

Accomplishes:

The artifacts that can be seen in Figure 24 are removed as much as possible, while retaining the letters' shapes in the binary map. This process is a combination of heuristics and morphological operations.

Reasoning:

Because the output of this step will be used in voting the regions of interest the text recognition will take place on, we would like to discard as many artifacts as possible. A big majority of them can be identified and removed because of their shape and size. The regions of text we are interested in are elongated and not very tall, while the letters have a common supple shape.

Algorithm:

Algorithm 5 Filtering Artifacts

Require: img

1: $img \leftarrow dilate(img, se, 1)$
2: $CCs \leftarrow contours(img)$
3: $img \leftarrow filter(img, CCs, heu_1)$
4: $img \leftarrow dilate(img, se, 4)$
5: $CCs \leftarrow contours(img)$
6: $img \leftarrow filter(img, CCs, heu_2)$
7: **return** img

The pseudo-code for the filtering process can be seen in Algorithm 5.

It starts off with a dilation, which is a morphological operation [5] which results in the thickening of the white connected components. It can be described mathematically as:

$$A \oplus B = \{z \in E | (B^s)_z \cap A \neq \emptyset\}$$

where A is our image with sets in it, B is a structuring element that is applied across the entirety of A and B^s is the symmetric of B. This is used in order to unite any artifacts and letter parts that might be disconnected because of the initial lighting conditions. For now I apply a single iteration.

The operation on line 2 looks in the image and finds all the connected components (CCs).

Then on line 3 we apply the first series of heuristics. These were determined experimentally and through the observation of the sizes of letters. On each component, a bounding box is placed. Any of the following rules (heu_1) would force the component to be discarded: $h < 10p \mid w < 10p$ (even if this was a letter, it would be too small to be recognizable), $h/w > 1.5 \mid h/w < 0.33$ or $h > 0.33 * h_{seg}$ & $w > 0.66 * w_{seg}$. (h - height of the component, w - width of the component, h_{seg} - height of the segment, w_{seg} - width of the segment).

After this first series of filtering noise, I've decided to apply again dilation operation but for 4 iterations in order to thicken up the remaining sets. I do not want necessarily to close

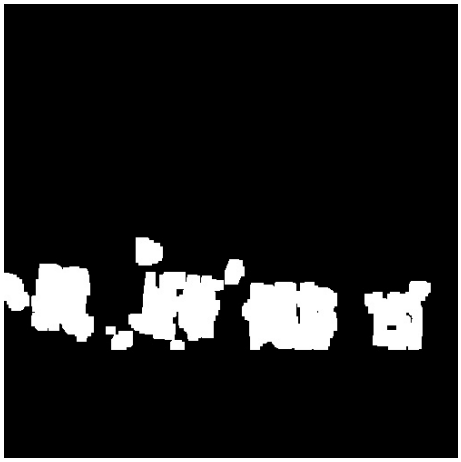

Figure 25: Resulted binary image with supposed text areas after artifact filtering

the spaces between them, as I would have chosen to use the closing morphological operation if I wanted that.

I search again for the connected components (CCs) in the image as they might have changed quite drastically from the last time.

And now, as a last step (line 6), I apply another filtration with other rules. In my experiments, I observed that some more artifacts could be eliminated at this step so I chose to do just that. The rules for removing these artifacts (heu_2) are: $h < 26p \mid w < 10p$ (even if this was a letter, it would be too small to be recognizable), $h/w > 1.5 \mid h/w < 0.33$ or $h > 0.33 * h_{seg}$ & $w > 0.66 * w_{seg}$. Practically, almost the same rules as in the first round of filtration, just that now the height of the component must be bigger as we have applied dilation a few times already.

Results:

Even if the filtering process gives some false negatives and lets some artifacts to pass, it would be way worse to have false negatives and start removing from the tire-markings themselves. I am satisfied with the current state of the filter because as can be seen in Figure 25 I could eliminate the vast majority of the artifacts present in Figure 24. This bitmap will be used for voting in the accumulator that has the same size with the unwrapped image.

Finding connected components is a quite expensive operation, so further improvement could be done on this step in order to reduce the number of searches for connected components to speed up the pipeline.

3.2.4 Voting

Now that the segment was processed (revealed the zone where text is present and removed as much as possible of the artifacts present in the image), the white zones that remained

will have their corresponding place in the accumulator matrix (that is the same size with the unwrapped image) increased by one. This is the action that I called voting. Because of the overlap between segments this voting step can be used to further reduce the number of false positive regions of interest.

After all the segments were processed, it results a voting map like the one in Figure 26 (it was enhanced to be more visible), where the brighter the region, the more votes it got. But this voting accumulator can be further processed in order to reduce the zones that will remain as regions of interest for the text recognition part of the pipeline.

The filtration could be described by the pseudo-code in Algorithm 6. I start off with a threshold operation, where I discard any region that did not get at least 3 votes. This step is done to reduce the artifacts that appeared only in some segments. The thresholding operation also makes the pixels that passed the check to have the maximum value, resulting in the accumulator becoming a binary image.

Algorithm 6 Cleaning Accumulator Matrix

Require: acc

1: $acc \leftarrow threshold(acc, 3)$
2: $CCs \leftarrow contours(acc)$
3: $acc \leftarrow filter(acc, CCs, heu_3)$
4: $acc \leftarrow dilate(acc, se, 4)$
5: $CCs \leftarrow contours(acc)$
6: $acc \leftarrow filter(acc, CCs, heu_4)$
7: **return** acc

Then, on line 3 I am filtering through the connected components found in the accumulator matrix (the operation on line 2). The component is discarded if its bounding box has $h < 40p \mid w < 10p$ (heu_3) as it would be too small to be even a letter. At this step the height limit was increased compared to heuristic hre_2 because the components that are voted by multiple segments tend to thicken up. This might be because of the frequency filtering leaves different aspects of a feature (tire-marking or artifact) visible as it is very dependable on the content of the segment.

On the filtered accumulator matrix, 4 iterations of the morphological operation dilation were applied to connect the letters. Other artifacts that are still present would not connect with anything else if they are far away enough from other connected components.

Now, on the newly diluted accumulator matrix and the new connected components, we apply another filtration using the heuristics heu_4 that consist of: heuristic hre_3, $h > 200p$ (the text region is too tall so it is a region that we are not interested in) or $h > 100p \ \& \ h/w > 0.85$ (a region too tall that doesn't have a prominent enough shape).

The result of this step is a bitmap (Figure 27) that when applied over the wrapped tire shows regions of text (Figure 28). This bitmap will be used in the text recognition stage

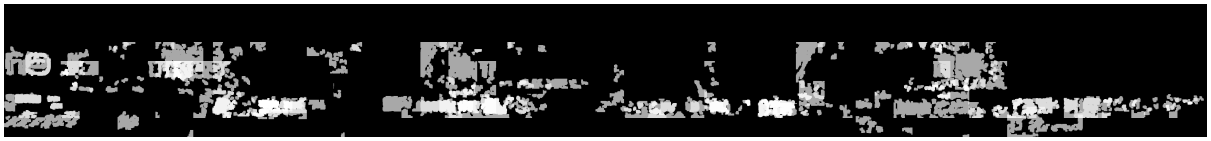

Figure 26: Voting map

Figure 27: Binary map with text regions

to indicate the regions of interest in the image and reduce the space on wich the algorithm will run on.

3.3 Text Recognition (Stage 3)

Accomplishes:

Now that we have reduced our unwrapped image to a few regions of interest, we can use an OCR to start extracting the text. This will be the output of our system and it will define our performance. The quality of the input images is also an important factor because during the evaluation of the system I observed that blurry and darker images (like Figure 29) – which translates in a lower contrast in the image – have the text not recognized at all.

Reasoning:

It could be considered that the reason is stated in the Motivation section of this work, but to reiterate, we want to extract the serial DOT code and the E-mark certification in order to almost uniquely identify a tire. I chose to use an OCR for this stage because they are the most common approach to text recognition and the availability of ready made solutions only increased in the past years [23]. There is also a wide variety to the OCR solutions that are offered as services, accessible through APIs.

Algorithm:

Figure 28: Text Regions on Unwrapped Tire

Figure 29: Blurry and lower contrast image

The OCR I used is a service from Microsoft Cognitive Solutions [11] and it uses "deep-learning-based models and works with text on a variety of surfaces and backgrounds" (source: [13]). So, it is not specially tailored to our case (tire images). This is a good thing as it provides versatility at the expense of performance and openness of the system.

The Read API (that is the OCR's name) [12] is accessed through API calls to a configured endpoint. For each region of interest extracted in the Text Detection 3.2 section, a request is sent to the endpoint which returns its guess on the text in the image. While not being able to perform all the processing on only one local machine (as we had to rely on the computation power and OCR solution hosted on a remote server), the impact this has on performance is reduced because we submit only the regions of interest (which have a smaller size than the original unwrapped tire image).

Results:

The results from this stage are in the form of annotations that are coupled with the original detected region of interest. The recognized text returned by the OCR will be categorized in the following categories: DOT serial code, E-Mark certification and miscellaneous.

4 IMPLEMENTATION DETAILS

This chapter will provide the details of the implementation in regards of the developing environment and libraries used. I chose to develop the pipeline in python3 [21] as it provides easy access to the OpenCV [16] library and a relaxed programming (because it is dynamically typed). The performance hit is minimal as the computationally intensive algorithms used are implemented in C/C++ and python is used only for interfacing and managing the flow of data. The project could just as well be developed in C++ using the OpenCV library. Another library that I used is numpy that provides the ability to perform matrix operations efficiently.

The developing environment was Jupiter Notebooks [6] as it provides easy code modularization and the possibility to test only certain sections, without having to rerun the entire pipeline.

Now, I will mention for each step described in The Proposed Solution 3 chapter its implementation if it contained an important algorithm. While the algorithms used were part of the OpenCV library, the interfacing of their outputs and the flow of data was of my own doing. I designed the pipeline that makes use of these algorithms.

a) CLAHE

This algorithm is performing Clip Limited Adaptive Histogram Equalization (CLAHE) and is provided by the OpenCV library. It is used as can be seen in the code snippet 4.1.

Code Snippet 4.1: CLAHE

```
clahe_obj = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
clahe_obj.apply(img_gray)
```

Firstly, it is needed to create a CLAHE object and provide to it the clipping limit and the tile grid size which will be used by the algorithm 1. The clip limit is a double representing a percentage and the tile grid must be a tuple of two integers. The returned object is of type $cv :: CLAHE$ and has a method called $apply$ which takes as input a gray-scale image and returns the processed image. The library has also the possibility to use the CUDA framework to run on the GPU the algorithm, but I did not use it as it would make the code not portable and complicate the setup. Anyhow, it is an interesting opportunity to study the speedup of the entire pipeline if some algorithms are run on the GPU.

b) Hough Circles Transform

In order to detect the circles in the equalized image, we employ the Hough Circles Transform [19] algorithm (that was described in 2) from the OpenCV library. We probe for multiple ranges of radii and later we will reduce their number and combine them. In order to sped up the algorithm, we reduce the size of the image to a thumbnail of 300 by 300 pixels. The code snippet 4.2 shows the usage of the provided algorithm.

Code Snippet 4.2: Hough Circles Transform

```
C, R = cv2.HoughCircles(thumbnail, cv2.HOUGH_GRADIENT, 2, param1=140,
    param2=130, minDist=thumbnail.shape[0], minRadius=70, maxRadius=140)[0]
ret = [(C, R)]
for new_r in range(80, 130, 10):
    nc, nr = cv2.HoughCircles(thumbnail, cv2.HOUGH_GRADIENT, 2,
        param1=140, param2=130, minDist=thumbnail.shape[0], minRadius=new_r
        - 5, maxRadius=new_r + 5)[0]
    ret.append((nc, nr))
```

The Hough Circles function from OpenCV takes in many parameters as it also applies internally Canny Edge detection [22]. The first parameter is the image (thumbnail) to be processed, the second is the algorithm to be used and the third is how many times smaller the accumulator matrix shall be compared to the image's resolution. The fourth parameter ($parameter1$) is the higher threshold for the Canny Edge Detector and the lower one is half of it. The fifth parameter ($parameter2$) is the accumulator's threshold after which a circle will be considered a circle. The sixth parameter is the minimum distance between the circles found by the algorithm in the image (as it can find multiple ones and return a list of them). The seventh and eighth parameters are the minimum and maximum radius of the circles to search for. The function returns a list of found circles in the passed image. The parameter values were found by trial and error and taking into account that the wheel should be at least 140 pixels in diameter. Anything smaller would have the text on the tire too small to be able to recognize it, so it would be better to just discard the image now than later.

After finding the main circle in the image, we start looking for other circles in different intervals of radii around the main one. The idea was to extract the most predominant circle in each interval and later combine them if there are too many found. All the found circles are pushed in a list which has the first element the main circle of the image.

c) Heuristics for Reducing the Number of Circles

In this part, I have implemented my heuristics for reducing the number of circles that we have detected in the previous step. The heuristics are based on the assumptions that the detected circles, if they have the centers close enough to each other and the radii similar,

they are probably the same circle. This combination of circles also helps with the distortion of the tire that causes it to appear as oval shaped. The grater the threshold to consider two circles the same one, the more circles will be reduced and we will be able to detect even more oval shaped ones. The problem is that we may end up combining the outer and inner rim's detected circles if the values are too big. So through trial and error, I've come up with the values that can be seen in code snippet 4.3.

Code Snippet 4.3: Reducing the Number of Circles

```
poped: bool = False
    while poped == True:
        poped = False
        combined_circles = []
        for i in range(len(circles) - 1):
            c1, r1 = circles[i]
            c2, r2 = circles[i + 1]
            if math.dist(c1, c2) < 20 and abs(r1 - r2) < 15:
                combined_circles.append(((c1 + c2) / 2, (r1 + r2) / 2))
                poped = True
        circles = combined_circles
```

It must be noted that I decided to create a new circle from the 2 ones that I considered to be the same one by averaging their centers and radii. The complexity of this algorithm is $O(N^2)$ where N is the number of circles, which is in average 3.36.

d) Conversion to Polar Coordinates

If the number of detected circles could be reduced to only 2 circles in the image, we can determine the center of the wheel, around which we will unwrap the tire. The unwrapping is performed by converting a section of the image around the center of the wheel to polar coordinates (the reasoning was described in subsection 3.1.2 Convert to Polar Coordinates). A function from the polarTransform [20] python module is used. The code snippet 4.4 shows the usage of the function.

Code Snippet 4.4: Polar Coordinates

```
((c1, r1), (c2, r2)) = circles
 r_min = min(r1, r2)
 r_max = max(r1, r2)
 c = c1 if r1 == r_min else c2

 unwrapped, _ = polarTransform.convertToPolarImage(img, center=c,
     initialRadius=r_min, finalRadius=r_max, hasColor=True, order=1,
     useMultiThreading=True)
 unwrapped = cv2.rotate(unwrapped, cv2.ROTATE_90_COUNTERCLOCKWISE)
```

I observed that usually the inner circle is better aligned with the center of the tire so we select it as the center. This might be because the outer rim of the tire is deformed at the contact point with the ground. This might trick the circle detection part into believing more circles are present and when reducing their number, the newly calculated center might drift away from the true center.

The first parameter of the function to transform the image to polar coordinates is the image to be transformed. The second parameter is the center of the wheel. The third parameter is the inner radius of the tire and the fourth parameter is the outer one. The fifth parameter is a boolean that indicates if the image has color channels. The sixth parameter is the order of the interpolation step that is needed to figure out what color are some pixels that are stretched and must be calculated. The seventh parameter is a boolean that indicates if the function could run multi-threaded.

As a last step, the image must be rotated 90 degrees counterclockwise to get the tire unwrapped in a horizontal shape with the text written from left to right.

e) Image Frequency Filtering

The core of the system that enabled the pipeline to detect text on the tire is the Image Frequency Filtering algorithm (a)). It is applied on a segment of the unwrapped tire image, segments stat have a certain overlay between them. The idea is to search for tire-markings in all segments of the image and then count the position of the supposedly text in an accumulator to perform the voting process. The code snippet 4.5 shows my implementation of this band-pass filter.

Code Snippet 4.5: Image Frequency Filtering

```
def frequencyFilter(segment, lpf = 7, hpf = 18):
    fshift = numpy.fft.fftshift(numpy.fft.fft2(segment))

    # create the mask that will be the bandpass filter
    rows, cols  = fshift.shape
    disk_big_r  = int(rows * hpf / 100)
    disk_small_r = int(rows * lpf / 100)
    disk_r = abs(int(disk_big_r - (disk_big_r - disk_small_r) / 2))
    mask = numpy.zeros(fshift.shape, numpy.uint8)
    cv2.circle(mask, (rows//2, cols//2), med_r, color=1,
        thickness=abs(big_r - small_r))

    fshift = fshift * mask # the actual filtering
```

```
    filtered_seg = numpy.fft.ifft2(fshift)
    filtered_seg = numpy.abs(filtered_seg)
    return filtered_seg
```

Firstly I use the numpy library to apply a 2 dimensional Fast Fourier Transform on the segment to pass it into the frequency domain. The function outputs a matrix of complex numbers which is fed into the function $numpy.fft.fftshift$ to bring the zero frequency part of the array into it's center (Figure 19). After this, I create a mask which is a disk with the small radius 7% of the dimension of the image and the big radius 18% of the dimension of the image. The mask is then multiplied with the complex numbers in the frequency domain. This action acts as a band-pass filter. Then the remaining frequencies are used to apply the Inverse Fast Fourier Transform to get the filtered segment.

f) Black Hat Morphological Operation

To perform the Black Hat Morphological Operation (which was used in step 3.2.1 in the pipeline), I used the function provided by the OpenCV [15]. The code snippet 4.6 shows the usage of the function.

Code Snippet 4.6: Black Hat Morphological Operation

```
filtered_seg = cv2.morphologyEx(filtered_seg, cv2.MORPH_BLACKHAT,
    np.ones((3,3), np.uint8))
```

The first parameter passed to the function is the image on which the operation will be performed. The second parameter is an enumerate type that indicates what operation will be performed. The third parameter is the structuring element to be used in the operation. The function returns the image after the operation was applied.

g) OTSU Thresholding

This algorithm was described here 4 and in the implementation I used the function provided by the OpenCV library [17].

Code Snippet 4.7: OTSU Thresholding

```
thresh, bin_img = cv2.threshold(filtered_seg, 0, 255, cv2.THRESH_BINARY |
    cv2.THRESH_OTSU)
```

This function is a general threshold function that internally selects what kind of thresholding to apply accordingly to the 4th parameter passed to it. In my case I wanted it to perform OTSU Thresholding in order to automatically detect the threshold value in the segment. The

first parameter is the image which has resulted from the previous step 4.6. The second parameter is not used in OTSU mode and the third parameter determines to what value the pixels are set when they pass the threshold value.

h) Dilation (Morphological Operation)

When I needed to apply the dilation operation (in the 5 and 6 algorithms) I used the function provided by the OpenCV library [14].

Code Snippet 4.8: Dilation

```
img = cv2.dilate(img, kernel, iterations_no)
```

Like any other morphological operation, the first parameter is the image on which the operation will be performed. The second parameter is the structuring element to be used in the operation. The third parameter is the number of iterations we want the operation to be performed. The function returns the modified image.

i) Finding Connected Components

There are multiple moments in the algorithm when I decided to find the connected components that are in the binary image in order to filter out artifacts that I do not consider text (Algorithm 5) or when I filtered the voting accumulator matrix for regions that are not probably text (Algorithm 6). I used the implementation provided by the OpenCV library [18] to find the connected components inside an image.

Code Snippet 4.9: Connected Components

```
_, cnts, _ = cv2.findContours(binary_img, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)
```

The first parameter of the function is a binary image in which we want to detect the connected components. The second parameter tells the function how to pack the connected components and how to set their hierarchy. The third and final parameter tells the algorithm if we want to optimize the contours that define the connected component (if a line is found, only its endpoints are worth to be stored). The function returns a tuple of three values. The first value is a modified image. The second value is a list of all the connected components in the image. The third value is a list of the hierarchy of the connected components. I do not make use of the first and third parameters as I am interested only in having the bounding box of the components.

j) Performing OCR with Read API

As the selected OCR solution is an online service provided by Microsoft [12], I interact with it through their defined API and using the Python libraries [4] that were provided by them. There is also a Docker deployable solution that Microsoft provides to business entities that handle sensitive information or want to be in charge of their own data. The code snippet 4.10 shows the usage of the API.

Code Snippet 4.10: Read API OCR

```
def OCR(file: str) -> Any:
    imageData = open(file, "rb")
    client = ComputerVisionClient(config.ENDPOINT_URL,
        CognitiveServicesCredentials(config.SUBSCRIPTION_KEY))
    operationID = client.read_in_stream(imageData,
        raw=True).headers["Operation-Location"].split("/")[-1]
    # the API response is checked with busy waiting
    while True:
        results = client.get_read_result(operationID)
        if results.status.lower() not in ["notstarted", "running"]:
            break
        time.sleep(10)
    return results
```

The first step is to open the image we wish to perform OCR on in binary mode. The file descriptor obtained will be used later to submit the payload directly to the API without needing from us to load the image into memory by hand.

The second step is to obtain a client token from the service. The $ENDPOINT_URL$ and $SUBSCRIPTION_KEY$ are defined in another file and represent my credentials for the Azure Cognitive Services. At the moment I am using the free tier of the service which allows me to perform a maximum of 10 requests per minute and maximum 5000 per month. I also obtained 200$ of free credit to test out the OCR and I am billed out of them for every request I make to the server.

Then, I send my payload by specifying the opened image I wish to perform OCR on. The operation ID is extracted in order to poll the API for the result. Doing busy waiting is not always a great idea to implement an API, but Microsoft's reasoning while designing it was that you can submit multiple requests and you won't have to wait for their results to come all at once.

5 EVALUATION

As I developed a processing pipeline to extract the tire-markings off images of tires, I needed a dataset. To build this dataset I took photos with a Cannon EOS 1300D that has a resolution of 5184 by 3456 pixels, but the picture was cropped around the center to 3456 by 3456 pixels so that the image would be square. The lens used is EF-S 18-55mm Canon Zoom which has diameter of 58mm. When taking pictures I used a focal length of 23 millimeters and the aperture, exposure time and ISO were kept on "auto". In the process of building the dataset, the distance from the lens to the tire's side was approximately 115 centimeters and the average exposure time was 1/93 seconds. The dataset size is of 107 images.

While capturing the images, I was careful to catch the entire wheel in the image because detecting half wheels or arcs would have proven difficult. One more adjustment I did was the camera placement in regards to the wheel's axle. I decided to be approximately in line with it in order for the wheel to appear circular in the image. If I wouldn't have done so, the tire would have had an oval shape which would not affect very much the unwrapping faze as it has some slack and it doesn't require the tire to be perfectly circular. Anyway, the outer rim of the tire doesn't have a circular shape because it is deformed at the contact point with the ground. The oval shape affects the unwrapped output like Figure 16, compared to Figure 15.

By controlling the distance between the camera and the wheel, as well as the camera position in regards to the wheel's axle, I was be able to detect the tire in the image more reliably and ensured the letters composing the tire-markings had at least 20 pixels in height.

When it comes to evaluating the running time, the system the algorithm is tested on contains an Intel i7-6700HQ CPU and 16GB of DDR4 RAM at 2133 Mega-transfers/s. The operating system is Ubuntu 20.04.4 LTS 64 bit with Linux Kernel 5.5.9-050509-generic. In measuring the time, I do not take into account the writing time to the storage media.

There are three main points in which the pipeline can be evaluated and they coincide with its steps. The first is the percentage of the dataset in which the tire could be identified and unwrapped. The second is the percentage of codes found in average on a tire and the average quantity of false positives in the regions of interest extracted. The third is the performance of the OCR in correctly recognizing the regions of interest.

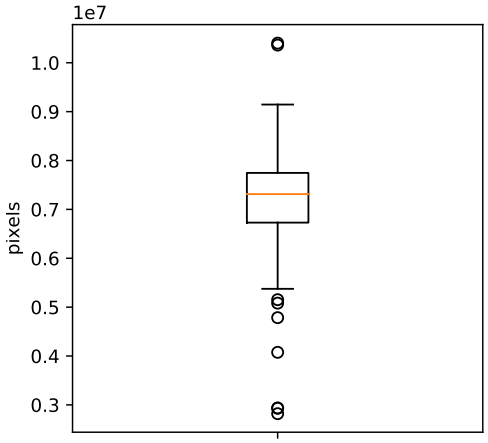

Figure 30: No. of pixels reduced by Stage 1

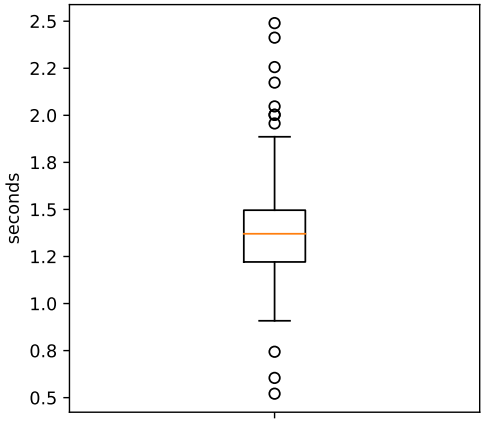

Figure 31: Running time of Stage 1

5.1 Evaluating the Tire Unwrapping (Stage 1)

I measured what percentage of the input images my algorithm is able to unwrap successfully. This is partially automated and partially manual. My algorithm tries to obtain the circles approximately matching to the outer and inner rims of the tire. If more circles are detected and the algorithm is not able to reduce their number to only two or not even 2 are found, the respective image will be discarded automatically. The manual part is to pass through each successfully unwrapped image and check if the wheel's center was correctly detected. If not, I will discard the image before proceeding to the next phase.

It turned out that the pipeline could identify and unwrap the tire in 86 images out of the 107 in the dataset. Out of these 86 unwrapped images, 5 were false positives that I observed while manually passing through them. So the first stage of the algorithm could find and unwrap the tire in 75.7% of the dataset.

By performing the action of unwrapping and keeping only the tire, I succeeded in reducing the number of pixels in the image by an average of 7,139,770 pixels (59,77% of the squared original size), the distribution of the pixel reduction count could be seen in Figure 30.

This stage on the hardware configuration previously stated has an average running time of 1.39 seconds (Figure 31). It must be noted that the IO operations with the long term storage medium are not taken into account when it comes to measuring the running time.

5.2 Evaluating the Text Detection (Stage 2)

When designing the pipeline part to detect text regions, I allowed for a higher number of false positives in order to detect as many codes and text regions as possible. In the successfully

Metric	False Alarm Rate	Detection Rate	False Negative Rate	Precision	Recall
Value	0.92	0.76	0.23	0.07	0.76

Table 1: Evaluation metrics for Stage 2

unwrapped images (numbering 81) the second stage detected a total of 1623 regions of interest, out of which . Thus, the precision of this step is (36.97%).

Before measuring the performance of detecting the tire-markings I am interested in – the DOT code and E-mark certification –, it is needed to define a few edge cases. If the text of a marking, for example the DOT code, is split in multiple segments instead of single one, it will be considered as a valid recognition and a new metric for segmented detections will be used. If a text region is incomplete recognized, it will be considered as being valid and this will accounted by another metric for incomplete recognitions. If the past 2 cases combine, it will be considered the tire-marking is also segmented and incomplete. For example if there is the DOT code "DOT ABCD EFG 0123" and it was detected as 2 regions "OT ABCD" and "01", they will be considered as one valid recognition, one segmented recognition and one incomplete recognition. If in the detected region of interest are present other markings besides the DOT code or the E-mark, it will be accounted as a combined detection.

In the successfully unwrapped images (amounting to 81), 1623 zones were considered to contain text, 600 contained any kind of text (true positives) and 1023 were false positives (63.03%). 151 identification tire-markings (DOT code and E-mark certification) were present in the unwrapped images and the algorithm successfully detected 116 (76.82% out of all the codes) markings, out of which 16 were segmented (13.79% out of the detected markings), 45 were incomplete (38.79% out of the detected markings) and 24 were combined with other markings(20.68% out of the detected markings).

To measure the performance of the pipeline in detecting only the tire identification codes and not any other text in the image, I need to consider true positives only the 116 detections and the rest of 1507 false positives. The number of false negatives is 35. I used the metrics described in the work of Afzal Godil et alia [1] to evaluate this step in the pipeline. The results are summarized in Table 1.

By finding regions of interest in the images, the space in which OCR must be performed later is reduced in average by 91.36% of the unwrapped image size. The average running time of this stage is 4.51 seconds (Figure 32).

5.3 Evaluating the Text Recognition (Stage 3)

When it comes to evaluating the OCR, I will use the Character Error Rate (CER) [9] metric. This will show the performance of the algorithm in recognizing the letters out of the good

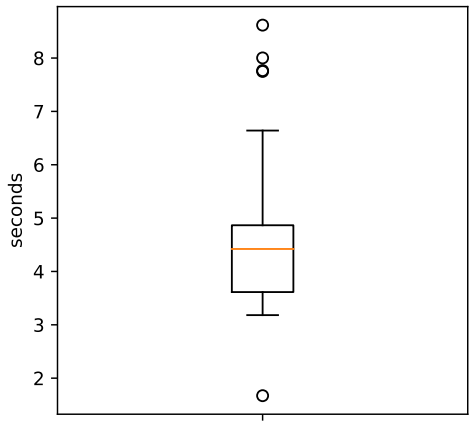

Figure 32: Running time of Stage 2

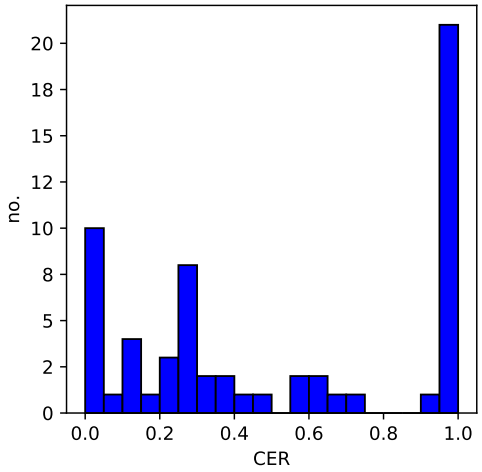

Figure 33: CER dist. on TP_S

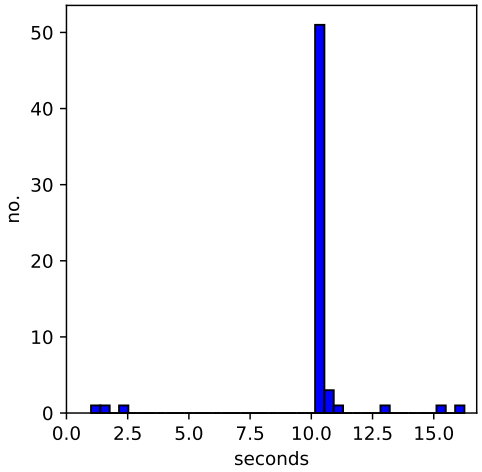

Figure 34: Running time of Stage 3

regions of interest, lower being better.

$$CER = \frac{S + D + I}{N}$$

Where S is # of substitutions, D is # of deletions, I is # of insertions and N is # of characters in ground truth.

I will hand pick the true positive images containing text of an acceptable image quality and discard the false positives detected at the 3.2 Text Detection section stage because I have limited a limited number of credits to try out the OCR (called Read API) from Microsoft Cognitive Services [11].

I have selected a subset of 61 true positives (called TP_S) detected at the 3.2 Text Detection stage. An interesting thing happened. If the OCR was able to find text in the image, it could recognize pretty good the characters in the image and usually almost all of

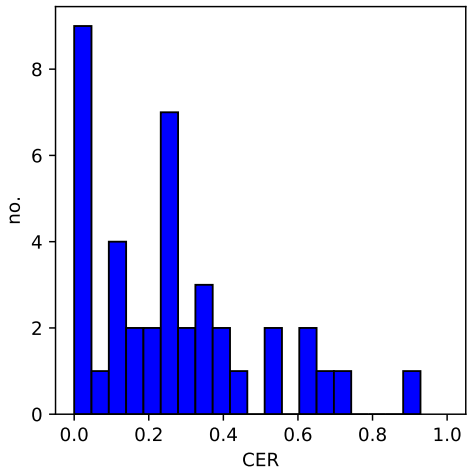

Figure 35: CER dist. on any recognized TP_S

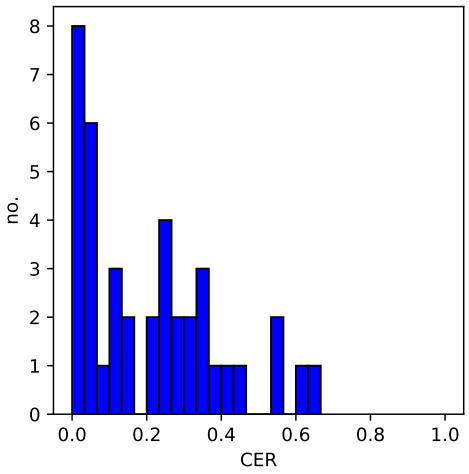

Figure 36: CER dist. on detected words

it. However, there is a discrepancy in the number of characters recognized in lower quality photos or blurry ones as the OCR fails to recognize any characters. This can be observed in the distribution of CER values for the 61 true positives in Figure 33 and has an average CER of 0.51. If we remove the images where no text could be recognized we get a distribution like in Figure 35 with an average CER of 0.26. This is a good sign that the OCR is working well on images that are not blurry or too dark (like Figure 29) and without contrast. I also measured the CER only on the words recognized in a image from TP_S and gave a distribution like in Figure 36 with an average CER of 0.20.

Measuring the running time is not a precise task as the OCR is a web service and it would depend on the network congestion and the service's load at the time. Anyhow, I still measured the time it takes from the submission of the request to the response. It resulted in an average of 10.17 seconds and as can be seen in Figure 34, the majority of the response time values is around this value with a few outliers. This may be an artificially induced delay in the service provided by Microsoft Cognitive Services.

5.4 Overall Evaluation

The pipeline created consists of 3 stages which take in total an average of 16.07 seconds or 5.9 seconds without the last stage that could not be exactly measured. The average time for processing an image and extracting its text is difficult to be compared to the other works [8] [2] because of the the differences in hardware, input size and the lack of a standardized benchmark.

However, in terms of performance, we can measure our results with the other works in the field only in averages. My pipeline, before reaching the OCR step, is able to parse 61.74% of the dataset. This means that in 61.74% of cases it is able to identify in an image the

DOT code and the E-mark (with the mentions specified in section 5.2 regarding incomplete detections). This is comparable to the average precision of 64.7% obtained by [2] in the text detection stage, though it must be mentioned that they are interested in tire-markings containing specifications about the tire rather than the identification codes.

If the image that is being processed by my pipeline is not blurry, the OCR provided by Microsoft Cognitive Services [12] is able to recognize the text in the image with an CER of 0.26. The performance on the entire dataset when the blurry images are introduced is lower, the CER reaching 0.51. This is still comparable to the average CER of 0.39 obtained by [2] on all tire-markings they were interested in. The work of Kazmi et alia [8] obtained an average CER of 0.23 on the 9 images they hand picked and tested on.

6 CONCLUSIONS

In this work, I introduced a full pipeline for detecting and recognizing the tire-markings that are able to identify a tire (the DOT code and the E-mark certification). The pipeline is able to recognize these markings in 61.74% of cases by using the deterministic processing steps employed in section 3.1 and section 3.2. The approach in detecting the tire-markings is a novel approach compared to other works in the field which employ a combination of Histogram of Oriented Gradients with Convolutional Neural Network based multi-layered perceptron [8] or pre-trained text detection models which are specialized later for tire-markings detection [2]. For the text recognition step I employed an OCR offered as a service by Microsoft [11] which in was able to obtain a character error rate (CER) of 0.26 in images where any text was detected. The OCR was not able to recognize the text in images which we could consider blurry and with low contrast (like Figure 29), this bringing the CER to 0.51 on the entire collected dataset.

Difficulties

I could say that the most difficult part of the pipeline to implement was the text detection step. The main culprit of this was the low contrast between the embossed letters (that make up the tire-markings) and their background. Because the letters are not a distinctive feature as can be seen in Figure 17, if they appear in the output of a processing algorithm, they are always accompanied by noise or other features on the tire that do not interest us.

For the beginning, I tried with Sobel edge detector [7] to obtain a gradient of change in the image, but it did not look useful as can be seen in Figure 37.

Then I tried further by applying Canny Edge Detector [22] but in order to obtain usable results, I had to fine-tune its parameters and these were extremely influenced by the color of the tire, the lighting conditions and noise. A somewhat usable output after fine-tuning can be seen in Figure 38. Noise was a real issue because if too much blurring was applied to remove the noise the letters would lose shape.

I also tried some thresholding techniques. OTSU Thresholding proved useful because it is a global algorithm and different materials on the wheel reflect light in different amounts so a single threshold is not practical. I tried also the option of adaptive Thresholding and it proved useful as can be seen in Figure 39, but it suffers also from parameter tuning, just like the Canny Edge Detector and because the system must be robust this is not acceptable.

Before resorting to Machine Learning models to solve this step (a thing that I avoided

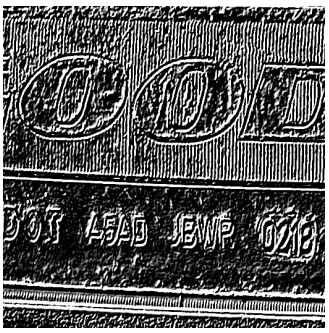

Figure 37: Sobel trial

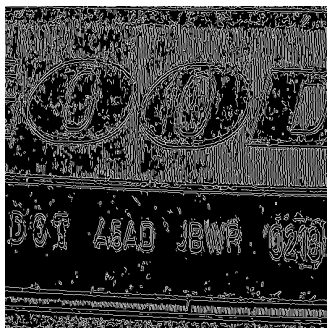

Figure 38: Canny trial

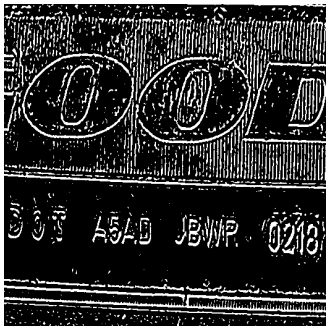

Figure 39: Adaptive Thresholding trial

in order to keep the processing pipeline simple and not computationally intensive), a solution appeared. Filtering certain frequencies in the frequency domain of the image showed that only the letters can be left prominent in a segment of the image. From this stepping stone, the text detection stage was developed in section 3.2. It proved to be quite accurate in detecting the identification markings on tires, 76.82% of them to be more precise.

Future Work

Even if the proposed system is a functional one, more work could be done in improving its performance. The most efficient approach would be to start with the quality of the input images and experiment with other cameras or lenses that might reduce the exposure time. The reduction of the exposure time would also reduce the noise and blurryness in the images.

Another aspect of the pipeline that could be improved is the text detection step which at the moment 63.03% of the reported regions do not contain any kind of text. The reduction in false positives would further reduce the pixel count on which the text recognition algorithm is applied.

And in regards of the text recognition stage, it would be best if I could migrate away from the Microsoft Cognitive Services OCR service and use a solution which could be run locally. Such a solution would most probably be an OCR which employs machine learning under the hood and could be specialized for tire-markings recognition. This means a bigger dataset will be needed to be collected and labeled accordingly.

BIBLIOGRAPHY

[1] Godil Afzal, Bostelman Roger, Shackleford Will, Hong Tsai, and Shneier Michael. Performance metrics for evaluating object and human detection and tracking systems. *NISTIR*, 7972, 2014.

[2] Katanaev Anton, Ukhmylov Yaroslav, Chekmareva Alfiya, and Khalili Roman. How to identify vehicle tires using deep learning visual models. `https://blog.griddynamics.com/how-to-identify-vehicle-tires-using-deep-learning-visual-models/`. Last accessed: June 2022.

[3] Ronald Newbold Bracewell and Ronald N Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-Hill New York, 1986.

[4] Microsoft Corporation. azure-cognitiveservices-vision-computervision 0.9.0. `https://pypi.org/project/azure-cognitiveservices-vision-computervision/`. Last accessed: June 2022.

[5] Robert M. Haralick, Stanley R. Sternberg, and Xinhua Zhuang. Image analysis using mathematical morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(4):532–550, 1987.

[6] Project Jupyter. Jupyter. `https://jupyter.org`. Last accessed: June 2022.

[7] Nick Kanopoulos, Nagesh Vasanthavada, and Robert L Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988.

[8] Wajahat Kazmi, Ian Nabney, George Vogiatzis, Peter Rose, and Alex Codd. An efficient industrial system for vehicle tyre (tire) detection and text recognition using deep learning. *IEEE Transactions on Intelligent Transportation Systems*, 22(2):1264–1275, 2021.

[9] Kenneth Leung. Evaluate ocr output quality with character error rate (cer) and word error rate (wer). `shorturl.at/muATY`. Last accessed: June 2022.

[10] J. MA, X. Fan, S.X. Yang, X. Zhang, and X. Zhu. Contrast limited adaptive histogram equalization based fusion for underwater image enhancement. *Preprints 2017*, 2017030086, 2017.

[11] Microsoft. Microsoft cognitive services. `https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/`. Last accessed: June 2022.

[12] Microsoft. Microsoft cognitive services read api. https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/overview-ocr. Last accessed: June 2022.

[13] Microsoft. What is computer vision? https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/overview. Last accessed: June 2022.

[14] OpenCV. Image filtering. https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html#ga4ff0f3318642c4f469d0e11f242f3b6c. Last accessed: June 2022.

[15] OpenCV. Morphological transformations. https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html. Last accessed: June 2022.

[16] OpenCV. Opencv about page. https://opencv.org/about/. Last accessed: June 2022.

[17] OpenCV. Otsu thresholding. https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577199e9c57. Last accessed: June 2022.

[18] OpenCV. Structural analysis and shape descriptors. https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga95f5b48d01abc7c2e0732db24689837b. Last accessed: June 2022.

[19] OpenCV-Docs. Hough circle transform. https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html. Last accessed: June 2022.

[20] polarTransform. Reference guide. https://polartransform.readthedocs.io/en/latest/polarTransform.html#polarTransform.convertToPolarImage. Last accessed: June 2022.

[21] Python. Python about page. https://www.python.org/about/. Last accessed: June 2022.

[22] Adrian Rosebrock. Opencv edge detection (cv2.canny). https://pyimagesearch.com/2021/05/12/opencv-edge-detection-cv2-canny/. Last accessed: March 2022.

[23] Han Ted and Hickman Amanda. Our search for the best ocr tool, and what we found. https://source.opennews.org/articles/so-many-ocr-options/. Last accessed: June 2022.

[24] Vswitchs. Excess redistribution in contrast-limited adaptive histogram equalization. https://en.wikipedia.org/wiki/File:Clahe-redist.svg. Last accessed: June 2022.

[25] Wikipedia. Gaussian blur. https://en.wikipedia.org/wiki/Gaussian_blur#cite_ref-ShapiroStockman_1-0. Last accessed: June 2022.

[26] Karel Zuiderveld. C ansi implementation of contrast limited adaptive histogram equalization. http://www.realtimerendering.com/resources/GraphicsGems/gemsiv/clahe.c. Last accessed: June 2022.