1. INTRODUCTION

This project is based on the previous work of Jay Hack and Kevin Shutzberg at Stanford University, Stanford, USA. Author’s goal was to create software that can transfer state of a Rubik’s cube (highly mathematical puzzle) to a computer from a computer vision perspective. They decided that this problem is good for convolutional neural networks due to cube’s faces are highly structured and have distinct pattern and textures.

They presented an approach that is based on two separately trained convolutional networks. One network is used for localization of the cube on a video stream and another is used to detect centers of the cube faces from cropped images. Furthermore, they presented some techniques from classical computer vision: featuring SLIC superpixel segmentation, projective transforms and color histogram classifications. In addition, they used technique for gathering large amount of labeled data for similar problems that is based on iterative training and supported by labeling procedure. In the end, they analyze the performance gains. Authors have their code hosted on GitHub, it is fully open source. Link: https://github.com/jayhack/ConvCube.

2. RELATED WORK

2.1 Cube Localization via Convolutional Neural

The main idea and probably the most innovational step in their algorithm was to use a convolutional neural network to extract bounding box of Rubik’s cube from an image with one on it. This approach allows to ignore noisy background for future work that can be affected by noise, reduce the amount of needed computational power due to relatively mild investment of CPU time.

They trained a six-layer convolutional neural network in order to get set of numbers that approximate center’s coordinates of any cube on the image as well as ox and oy scale factors. Author’s highest-performing architecture was like that:

**Layer 1**
Conv-Relu-Pool 16 3x3 filters; Max Pooling; Pool stride of 2, kernel of 2.

**Layer 2**
Conv-Relu-Pool 16 3x3 filters; Max Pooling; Pool stride of 2, kernel of 2.

**Layer 3**
Conv-Relu-Pool 32 3x3 filters; Max Pooling; Pool stride of 2, kernel of 2.

**Layer 4**
Conv-Relu-Pool 16 3x3 filters; Max Pooling; Pool stride of 2, kernel of 2.

**Layer 5**
Affine-Relu 64 neurons.

**Layer 6**
Affine 4 neurons.

Raw input data consisted of 640x360 RGB images captured from a web camera under different lighting conditions: indoors, outdoors, dimly-lit rooms, night time and in direct sunlight. They resized the images to 100 by 100 pixels for this step in order boost accuracy (empirically was determined that for this case 100 by 100 resolution improved accuracy) and significantly lower the computational cost. First of all, they trained their neural network against hand-labelled bounding boxes (Euclidian loss function). Usually, these kind of tasks are quite high-performing and the speed is much better than real time under a variety of lightning conditions.
2.2 Finding Cube Pieces via SLIC Superpixel Classification

The authors were thinking about Rubik’s Cube structure and since the standard cube’s pieces are all colored, sized and shaped uniformly they came to the point that SLIC superpixel segmentation is the best choice to detect pieces via segmenting the image into rectangular superpixels. The next step was to classify all the detected pieces using internal color histograms and shape descriptors. Actually, their approach proceeded following steps. The first one was to run MiniBatch Kmeans algorithm on pixel values and delegate each pixel to its corresponding centroid in order stimulate clustering of similar pixels, that was needed as a support for superpixel segmentation. The next one was to run SLIC superpixel segmentation algorithm on the results from Kmeans image which produced a set of candidates for Rubik’s Cube pieces. The following step was to collect histogram of HSV pixel values, to compute the third moment of the covered area and to add computations to the resulting vector. The last step was to classify all superpixels as a cube piece or otherwise using a random forest classifier trained on hand-labelled ground-truth data.

Fig. 1 Original cropped image

![Kmeans / SLIC superpixels](image1.png)

Fig. 2 Classified superpixels

![Candidate piece’s centers](image2.png)

Fig. 3 Candidate piece’s centers
2.3 Pinpointing Centers of Cube Faces

They accomplished quite reliable performance in identifying locations of cube piece’s candidates, as can be seen from the previous section, but still cannot be meant as a full reconstruction of Rubik’s cube. With a view to grasping the structure of Rubik’s cube they had find point-wise locations correspondences between the pieces of cube and locations in the image. Their approach contained training of a convolutional neural network in order to identify the location of center pieces for each visible face. After that they assigned the cube piece surrounding candidate centers their cube coordinates based on relative positions to the identified face centers. The authors used a six layer neural network with exactly the same architecture as in the section 2.1 with a small change, last layer was reverted to two coordinates. To predict the centers of faces they trained 3 networks, one for yellow and white, second for blue and green, and the last one for orange and red. As can be seen they picked opposite faces of the cube in order to take the advantage of that opposite sides would never be seen at the same time so they did not require separate network for prediction that is why only 3 neural network were used.

2.4 Bootstrap Learning for Data Acquisition

The authors thought on such a problem that their algorithm is running in the real time and any additional data for the machine learning portions made a benefit so they decided to use an output from the various classifiers on real data as additional training data. As a next step, they developed a module that allows to record person with a Rubik’s cube, after that sanitize the output of the classifiers for rejecting labels and inferences that are far away of the mark in order to keep training data precise and at high-quality level. The used 10 – 15 seconds long videos, in average, and it took about 15 minutes to hand-label one video, this process produced from 150 to 300 labelled data samples. Actually, drawing bounding boxes on each frame consumes the majority of the time in hand-labelling. And, finally, the bootstrap learning step is meant to ask one to decide whether to accept or reject labels produced by the algorithm, which is closely immediate for each frame; this process shortens the time to label 10-15 seconds video to 2-3 minutes.

2.5 Extracting Cube Face Information

Using resulted predictions of the face’s centers they were able to compute a projective transform to get the cube orientation and to extract face’s colors. These procedures made possible the classification of each square’s color by a manually-tuned support vector machine (SVM).

3. MY APPROACH

3.1 Thresholding

Nowadays, computer vision application can’t process colored images so they need to convert input image to grayscale (monochrome). There are two basic methods for thresholding: global thresholding and adaptive thresholding. Global thresholding is the simplest method. It takes an image, goes through all the pixels, calculates intensity ((R+G+B)/3) and compare it to a global thresholding value (256/2 = 128). If a pixel intensity is larger than thresholding value than the pixel is converted to white, otherwise to black. Global thresholding it is not useful in real world.

![Comparison of image thresholding](image.png)

Adaptive thresholding is used to get a better result. It does not use a fixed thresholding value (128), the value is calculated for each pixel separately. For instance, it takes 11*11 (121) surrounding pixels, computes the sum of their intensities and divides it by the number of pixels (121). If the mean intensity of the current pixel, located in the center of that area, is greater than the mean thresholding value, it will become white, otherwise black. This algorithm needs to
do all of that for each pixel so it is much slower than the previous but it gives much better results.

Fig. 5 Finding logic of adaptive thresholding value

3.2 Contour and corner detection

In order to start the image processing adaptive thresholding was applied so the returned image was suitable for finding and extracting contours. Basically, lines on that image were turned to white while the background and other light pixels were turned to black. I assumed that image or video contain standard Rubik’s cube, which is 9 by 9.

Code responsible for finding contours iterates through all the contours and stores threshold-passed contours to an array. A contour is a list of points. Its representation can be different depending on the circumstances. In order to find a contour in a binary image all connected components (pixels connected to each other) in the image should be considered.

From scratch, I ran into a problem that edges of cube are not very clear so I was not able to extract faces just by finding bounding boxes for faces contours. In contrary, I decided to use contours that were detected with better quality, contours of each piece of Rubik’s cube. So as a first step I was trying to find all the contours that are closed and whose approximation gives 4 corners, which means that they might be our diffused square pieces of cube’s face. Then I decided to compute remove duplicates, they occur sometimes if OpenCV sees a couple of different contours around the same object, so I took all contours with almost the same center of mass and remove all of them but one. At this point I computed corners of these distinct contours to the horizon and if there 9 of them with the same angle then we found our face. The next step is quite simple, find the extremums or top-left, top-right, bottom-left, bottom-right points, depending on the angle of our face, these points will be the corners of the face.

3.3 Extraction

Since detection of corners for each face is described in the previous section, at this point, we have 4 corners and simply need to be passed to the next step where linear transformation will be applied. As a result, plane image of a face will be stored.

Limitations. It is not possible to detect face corners and extract the actual face from the image if an image or video quality is quite poor, contours are broken or etc. However, if the source quality is good enough than this algorithm is quite sufficient because of its simplicity and performance.

3.4 Hough Transformation

Hough transformation is a numerical method algorithm used to extract features from an image. In this case, it extracts lines. It skips all the white pixels and draws 180 virtual lines through each black pixel. Pixels ‘vote’ for the lines and the virtual line with the largest number of votes in the accumulator is the winner and most probably the real line.

This algorithm is another alternative option for extracting face from the source (image or video).

3.5 Transforming Image

After face corners coordinates were received, program extracts them and apply linear transformation in order to get a square shape image. For this purpose was decided to use perspective transformation equations:
There are 8 unknown variables in these two equations but they could be found with construction of 8 equations system (four for each of two equations). With this system of equations it is possible to map points from distorted image to a flat square image. Basically, in the program, OpenCV’s GetPerspectiveTransform and WarpPerspective methods were used in order to apply perspective transformation to a source.

3.6 Identifying unique faces

Firstly, it was thought to use SURF to determine image features and compare faces on this basis but since in this project only the solved cube was used it turned to use a Histogram comparison as a better choice for this case.

It was decided to keep all uniquely detected faces in the list and compare each newly detected one with all from the list. In order to do that it is needed to convert images to HSV, then calculate Histogram from HSV images and normalize them. In the end, just compare images with one of possible methods, in this case correlation was used, it worked fine. Basically, if the correlation is more than 0.9 it is assumed that faces are to similar and it could be the same face.

This algorithm might not be the better choice but in the current project it worked as it was expected.

3.7 Color detection

Color detection is nearly the easiest part of the program. In order to detect RGB color representation for each piece of cube face it is simply needed to segment image into n equal squares, since in this project 3 by 3 Rubik’s cube was used, segmentation produced 9 pieces and this code won’t work for larger cubes.

After the piece of face is cut out just take the pixel in the center and get its intensity, that is all, RGB color detected. Collect them to some data structure and, basically, the process is finished.

4. RESULTS

Unfortunately, only a few samples of poor quality were used to test the algorithm, so the results are not satisfactory but even though, they showed that the algorithm can do what is expected of it. Here is an example of detected corners of the face from the video stream:

![Fig. 7 Detected corners](image)

After that this face was extracted and transformed:

![Fig. 8 Detected face cut out](image)

Since the video dimensions are low, cube was about 87 by 87 pixels that is not very good conditions and quality for the real working program, but for the prototype it was fine and still showed that algorithm is working.

As a next step, image is segmented into 9 pieces like the following one:

![Fig. 9 Sample of segmented face](image)
And in the end, RGB color from the center of this sub image is extracted. That is all for the recognition part.

Rendering part was also done, here is an example of rendered 3D cube:

Fig. 10 Rendered cube model

5. FUTURE WORK

Basically, there are a lot of ideas how the program could be improved. The main thing that is left to do is to map the recognition part to the rendering part. The next important goal is to ensure that algorithm recognizes unsolved Rubik’s cube or to modify face comparison part in order to work as expected for unsolved state of cube.

Frankly speaking, each part of the program needs some improvements. Corners detection is pretty universal but it would be nice to improve performance, detecting of unique faces works fine with histograms but may collapse for unsolved cube though it is better to add SURF detection to the workflow and so on. Since, the main goal of this mapping is to solve the cube, in the future it is also needed to solve the cube after it was mapped to a model and visualize the process.

6. CONCLUSION

To sum up, I have developed a software for reading Rubik’s cube from an image or a video and recognize it. If the source quality is quite good, made by a modern smartphone, program should produce good results. In case of a bad quality, recognition failures occur more frequently, that happens because source may contain a lot of unnecessary information, for instance, video stream that was used for testing contained a person and a hand rotating the cube and under a low video quality algorithm may not work very well.

7. ACKNOWLEDGMENT

I would like to express my special thanks of gratitude to my supervisor Artjom Lind as well as the head of Distributed Systems research group professor Eero Vainikko who gave me good opportunity to do this wonderful project, which also helped me to learn about so many new things.

8. REFERENCES

2. SudokuOCR - Project Report (MTAT.08.024 Distributed Systems Seminar). Vitalii Zakharov
5. Contours - 1 : Getting Started (http://opencvpython.blogspot.com.ee/2012/06/this-article-is-tutorial-which-try.html)
6. Contours - 3 : Extraction (http://opencvpython.blogspot.com.ee/2012/06/contours-3-extraction.html)
7. Thresholding (http://opencvpython.blogspot.com.ee/2013/05/thresholding.html)
8. Histogram Comparison (http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/histogram_comparison/histogram_comparison.html)