Final Project Writeup:

Solving the Problem of Rubiks Cube Competition Data Entry

Computer Vision — CS 4501

Bridget Andersen, Steven Stetzler, Samantha Reid

May 7, 2017

1 Introduction

See the README.md file in our repository (which you can view at https://github.com/slreid/rubiks-scanner as well as in our zip submission) for information about how to run the contents of our project.

We would not mind having our project writeup posted on the course website.

Motivation: We are trying to solve a problem that plagues Rubik's Cube competitions. At these competitions, competitors solve various Rubiks Cube puzzles and their time to solve is handwritten on a paper scorecard. During the competition, someone must read these scorecards and enter them into a database. After the competition, competition organizers are tasked with verifying that the times written on the scorecard match the online database. Both of these tasks require many hours of tedious human labor. We aim to solve this problem using computer vision.

Overall Goal: Our goal was to create a system that takes in a picture of a Rubiks Cube competition scorecard and extracts the competitor ID and solve times to allow for automatic entry into an online database.

Students and Roles

- *Bridget Andersen*: worked on using SIFT to normalize the orientations of the scorecard images before information extraction.
- *Samantha Reid*: worked on training the neural network for digit recognition. She also worked on configuring the online database to allow for the data from the scorecard to be saved online and retrieved at a later time.
- *Steven Stetzler*: worked on extracting the digit boxes from the scorecard image, cropping the digits, and constructing the individual solve times.

2 Tasks and Milestones

(1) *Creating the Scorecard Template*: The left image in Figure 1 is an example of an actual scorecard used at a recent UVA Rubiks Cube competition. The layout of the card gives a lot of leeway as to where the digits and characters can be written. As a result, the digits tend to be overlapping and oddly spaced. To make our eventual digit extraction process easier, we created a new template with individual boxes for each digit. This new template is shown below on the right in Figure 1.

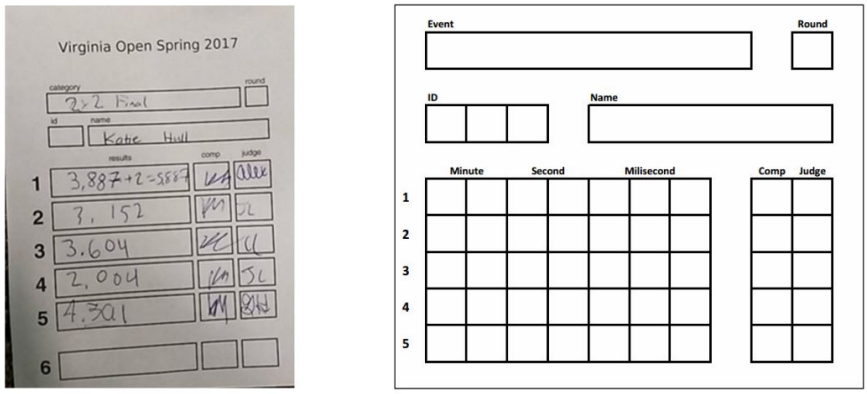

Figure 1: The image on the left shows an actual Rubik's Cube competition scorecard. The image on the right shows our new competition scorecard template.

(2) *Taking the Picture*: Image capture is performed using a phone or computer webcam. We used the Android app IP Camera in order to connect a phone to the computer we ran our code on as a webcam. Video is sent from the camera at a resolution of 800 by 400 pixels at a framerate of 30 frames per second. Every fifth frame from the video output is analyzed. In these frames, we search for contours which match our template image (shown in green in Figure 2). Once a good contour is found, we perform further analysis to extract the scorecard from the current video frame. Performing a contour analysis on every fifth frame, and extracting the scorecard only on frames where a good contour is found allows us to maintain smooth video feedback as the user tries to scan the scorecard.

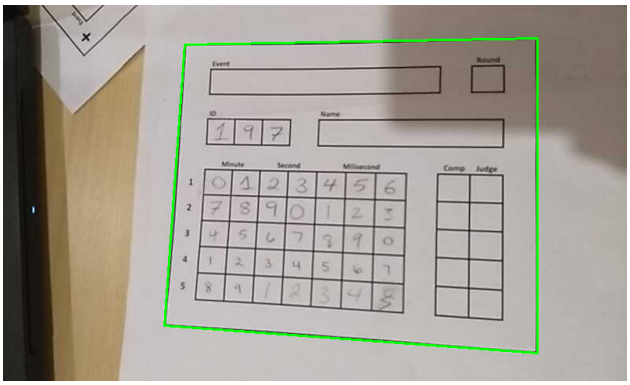

Figure 2: An image of the live view from the phone webcam output with a contour highlighting the scorecard.

(3) *Extracting the Scorecard*: In order to obtain a robust scorecard extraction, we tried using a method similar to what we implemented in Project 2. With this method, we first use OpenCV to extract SIFT features in both the input image and the template. Then we use the Fast Library for Approximate Nearest Neighbor (FLANN) and the ratio test to find good matching features between the images. Finally, we use these features to compute a homography between the images and then use this homography to warp the perspective of the input image to match the template, giving us a normalized top-down view of the scorecard. This process is outlined in Figure 3. We originally had large plus sign markers in the corners of the scorecard to try and guide the matching process. However, FLANN actually did an excellent job of picking up unique matching features between the images like the individual words and numbers, so the plus signs ended up not being necessary and we removed them from our template. We found that this method was really robust to different lighting conditions and perspectives, within reason.

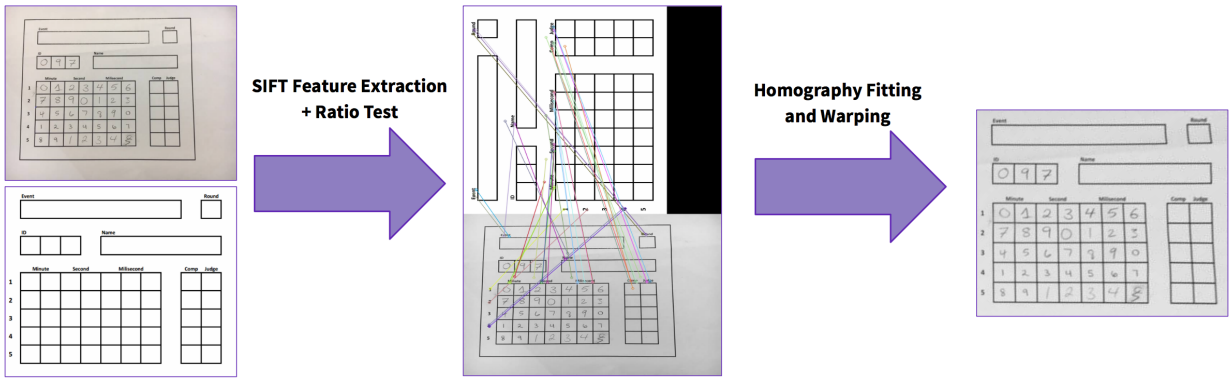

Figure 3: A flowchart showing the images at each step of the scoreboard extraction process. The images on the left show the template and the input image, the image in the center shows the detected matching features, and the image on the right shows the resulting extracted template.

(4) *Digit Cropping and Recognition*: Since each digit is contained within its own box on the scorecard, digit extraction is relatively trivial. To obtain an image of each individual digit, we just cropped out each digit box by hard coding its corresponding coordinates from the scorecard template. We then thresholded the digit image to make it look more like the digits from the MNIST dataset. Finally we used a neural network to classify each of the cropped digit images. At first, we used one of the neural networks that we created in Project 3 (see the `samantha.h5` model file). This neural network consisted of a sequential model containing one convolutional layer, a max pooling layer, and a softmax layer. We trained this network for 12 epochs using 60,000 training samples from the MNIST dataset. A graphic of our digit cropping/recognition process is shown in Figure 4. The purple digits in our results indicate mispredicted numbers.

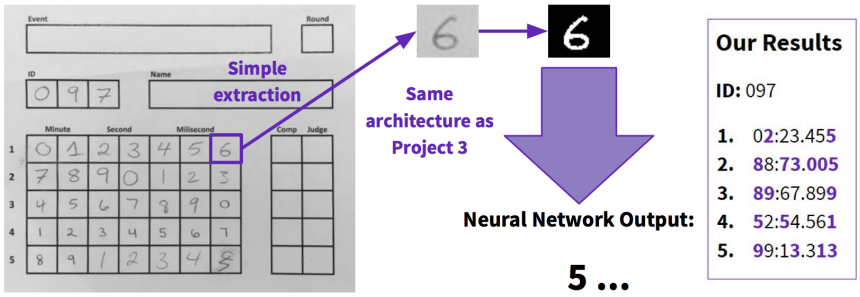

Figure 4: A flowchart of the digit cropping and recognition process along with the predicted results for this particular input image.

As you can see, we had a large number of digits that were not predicted correctly. We tested our network on 10,000 MNIST samples and received the confusion matrix shown in Figure 5. From the matrix, it appeared like the neural network was actually doing a good job classifying MNIST digits. This indicated that our input digit images must have differed significantly from the MNIST testing set. By more closely examining our extracted digit images, we determined that many of the digits written in the scorecard were misaligned or differently sized. Some examples of misaligned digits are also shown in Figure 5. We can see that the zero is oddly placed in the corner of the input image and the six and seven are significantly different sizes.

	0	1	2	3	4	5	6	7	8	9
0	**0.995**	0.001	0.001	0.000	0.000	0.000	0.001	0.001	0.001	0.000
1	0.000	**0.998**	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.001	0.001	**0.993**	0.000	0.001	0.000	0.000	0.004	0.000	0.000
3	0.000	0.000	0.002	**0.995**	0.000	0.000	0.000	0.001	0.001	0.000
4	0.000	0.000	0.001	0.000	**0.996**	0.000	0.002	0.000	0.000	0.001
5	0.001	0.000	0.001	0.008	0.000	**0.987**	0.003	0.000	0.000	0.000
6	0.004	0.002	0.000	0.001	0.001	0.004	**0.987**	0.000	0.000	0.000
7	0.001	0.002	0.007	0.001	0.000	0.000	0.000	**0.988**	0.001	0.000
8	0.005	0.001	0.004	0.001	0.002	0.000	0.000	0.005	**0.978**	0.003
9	0.003	0.002	0.000	0.001	0.005	0.004	0.000	0.007	0.001	**0.977**

Figure 5: The image on the left shows the confusion matrix of the first neural network that we used to predict digits. On the right we have several examples of misaligned digits.

To solve this issue, we created a new algorithm to normalize our digits and make them appear more like the MNIST training data. The digit images were extracted using the same simple cropping method as before. Then, using OpenCV, we calculate the contours in the digit image and find a bounding box that encompasses the contour representing the actual digit. Then we crop the digit at the bounding box and resize it to an approximately 20 by 20 pixel image while maintaining the aspect ratio. Finally, we pad this resized digit with zeros to obtain a 28 by 28 pixel image similar to the MNIST images. This whole process is outlined in Figure 6.

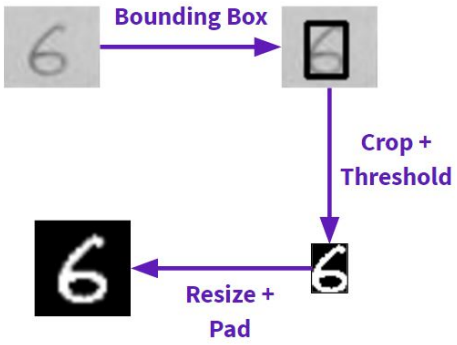

Figure 6: A flowchart of the process we completed to make our input images appear more like the MNIST dataset.

We also trained a new neural network to try and improve our prediction accuracy. This neural network consisted of a sequential model containing two convolutional layers, a max pooling layer, a dropout layer, a densely connected RELU layer, another dropout layer, and finally a softmax layer (see the `new_model.h5` model file). Again, we trained this network for 12 epochs using the MNIST dataset.

During prototyping of our application, we initially chose to input one image of an extracted digit at a time to our neural network for prediction. We noticed that digit prediction took a long time, and was the main bottleneck in the speed of our application. We modified our code to instead take advantage of Keras's ability to predict many inputs in one go. Instead of supplying a single digit image, we instead now supply an array of all of the extracted digits for prediction in one batch. This single modification increased the speed of our application tremendously.

After making all of these changes, our new results ended up being much more accurate. These results are shown in Figure 7. Only two digits were predicted incorrectly using this method, which is a significant improvement over the previous method. The confidences associated with the mispredicted digits are shown in Figure 7 on the right. Note that if a digit is recognized with low confidence (below 75%), we flag that digit for human verification. Under this criteria, only one digit in the fourth row was flagged although it was a correct digit categorization.

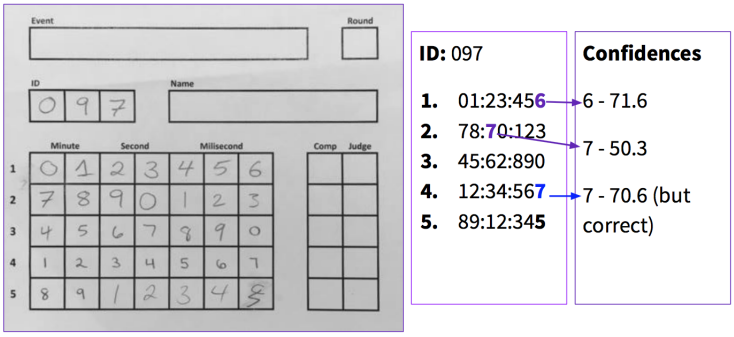

Figure 7: The results for our new image normalizing algorithm and neural network setup.

(5) *Time/ID reconstruction*: Once we had all the digits from the scorecards, we constructed the competitors solve times and unique ID out of the individual digits.

(6) *Sending Data to Database*: We used Google Firebase to simulate the Rubiks Cube competitions database (follow this link to see our database: https://rubiksscanner.firebaseio.com/). After constructing the solve times and competitor ID, the data was sent to the database. Each competitor is stored in the online database by competitor ID. Under each ID are the solve times for each round and the average solve time. The average solve time in rubiks cube competitions is calculated by dropping the fastest and slowest solve times of the five rounds and averaging the remaining three times. The flagged digits for the ID number and each round are stored under the Flagged portion. Each round contains a string of the indices (0-6) in the time string that correspond to the digits that have low confidences.

(7) *Retrieving the Results*: Competitor IDs can be retrieved from the database and sorted in order of increasing average solve time. When the data is fetched from the database, the times are checked in case data was changed manually in the database. New average times are calculated if necessary. This is an easy and fast way to get the ranking of competitors in the Rubik's Cube competition.