WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

DIP Project

Tristen Haverly - 1864691
Michaela Klug - 2393033

June 7, 2023

1 Abstract

This report presents a digit extraction algorithm for images of Sudoku puzzles. The algorithm utilizes digital image processing techniques to accurately identify and isolate individual digits from the puzzle grid. Extensive experimentation and evaluation demonstrate the effectiveness and robustness of the proposed algorithm when tested on different Sudoku grids. The goal of extracting the digits from the grid is to present them in a sensible way so as to be fed into a Sudoku solving algorithm or model.

2 Introduction

Sudoku is a puzzle solving game which is played on a 9 x 9 grid. The puzzle starts as a grid with numbers in the range [1,9] in random squares of the grid. The objective of the game is to fill in the empty squares with numbers in the range [1,9] so that no number is repeated in each row, column and 3 x 3 square block [1].

While many Sudoku puzzles are presented along with their solutions in various newspapers, magazines or dedicated activity books, there may be a situation in which an individual would like to check the solution to the puzzle when the solution has not been presented. Various Sudoku solving algorithms and models have been developed to solve this problem. However, they usually require one to manually enter the grid into the model/algorithm. Manually entering the digits of the Sudoku grid into some solvers is tedious and so it would be much easier to simply upload a picture of the Sudoku problem in question, have some program which extracts the digits for you and passes them into a Sudoku solver.

It is for this reason algorithms such as the one presented are being developed. Here, we demonstrate a non-learning approach for extracting the digits out of an image of a Sudoku puzzle along with their corresponding positions in the puzzle. We then use a machine learning model to predict the values of the digits in the image and present the Sudoku puzzle in a text-based grid.

3 Terminology

Before delving into the algorithm's details, it is essential to define the following terminology:

- Sudoku puzzle: A 9x9 grid containing partially filled digits from 1 to 9.
- Digit extraction: The process of identifying and isolating individual digits from the Sudoku puzzle grid.
- Pre-processing: The initial stage of the algorithm, which enhances the visibility of digits and removes noise or unwanted elements from the puzzle image.

[1] https://sudoku.com

4 Methodology

In this section, we provide a detailed description of the methodology employed in our Sudoku puzzle digit extraction system. The methodology consists of several key steps, including pre-processing, grid detection, digit extraction, and digit recognition.

Figure 1: Example image

4.1 Pre-processing of the image

All techniques used in this subsection are from the DIP course

By applying the following pre-processing techniques, we aim to improve the quality of the input image, making it easier to detect and extract the digits.

1. Once we have read in an image from the data set, we convert the image to gray-scale. This allows us to process many types of images including those in RGB format.

Figure 2: Gray-scale example image

2. We then perform a thresholding on the image using Otsu's thresholding method. This converts the gray-scale image into a binary image of 0s and 1s and allows us to process the image more easily. This has a benefit in that morphological operators are simpler and may work quicker for binary images. Since we frequently make use of various morphological operators, we decided to convert the image into a binary image at this stage.

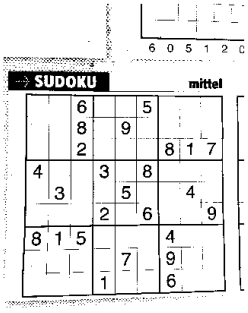

Figure 3: Binary version of example image after Otsu's thresholding

3. We take the inverted image so that the background pixels are black and the foreground pixels are white and then we use the function binary_fill_holes from Skimage[2] to perform a flood-fill on the binary image. This function identifies the foreground (white) pixels and fills in the gaps between those foreground pixels- i.e it modifies the standard flood-fill function for binary images. We perform this binary flood-fill in order to fill in the region of the Sudoku grid and use that area for further grid isolation.

[2] https://scikit-image.org/

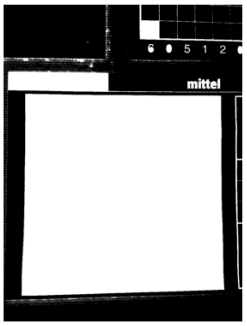

Figure 4: Image depicting results of binary flood-fill

4. We perform an opening on the result of the binary flood fill, with a disk-shaped structuring element with a radius of 50, to remove smaller details and noise. The goal is that only the larger grid area should remain after the opening.

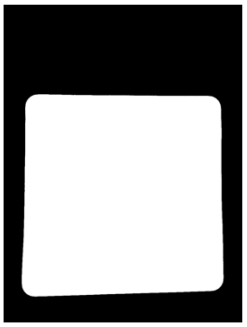

Figure 5: Result of opening

5. Next we obtain the convex hull of the image. The result is the approximate shape/polygon of the grid shape. We find the 4 extreme points (corners) of this convex hull and use those points to crop the image around the Sudoku grid, isolating the grid from the rest of the image. We also obtain a cropped version of the inverted version of the original image for later use.

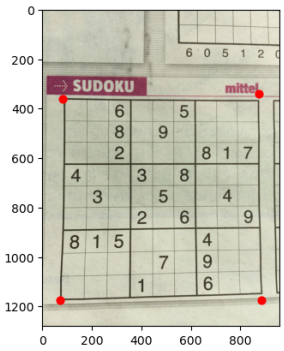

Figure 6: The extreme points of the convex hull

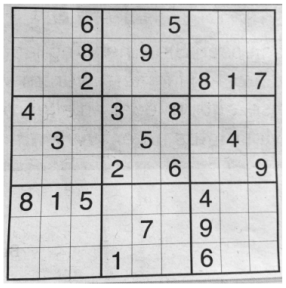

Figure 7: The resulting cropped gray-scale image

4.2 Grid detection

All techniques used in this subsection are from the DIP course

1. Hysteresis thresholding is performed in order to detect the edges in the image. Those edges being the grid lines and some numbers. The value of the lower threshold is set to 0.05 and the high threshold is set to 0.38. We take the inverted result of the thresholding and perform an opening with a disk-shaped structuring element with a radius of 3 on that result. This is to fill in any gaps in the edges and make the rest of the processing more successful.

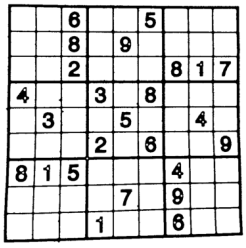

Figure 8: Result of hysteresis thresholding

2. The next step is to isolate the grid from the image and exclude any numbers present. We achieve this in 2 steps. The first step is to remove any grid lines from the image. Here we use a flood-fill. We iterate through the image in order to find the first black pixel and this pixel corresponds to the first pixel in the outer grid of the Sudoku puzzle. Once we have

located the first black pixel we use those coordinates as the coordinates of our seed point and then perform a flood-fill, filling in the 'flooded' region with the value 1 (white)- the colour of the background of the image. The result of this first step is the image with numbers and no grid lines whatsoever.

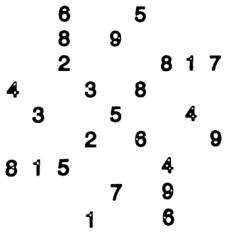

Figure 9: Result of the flood-fill, no grid lines remain

Step 2 involves taking the 'difference' of the image from step 1 and the original cropped image. We use a bit-wise xor function from NumPy[3] to achieve this effect and the result is now only the grid of the Sudoku puzzle.

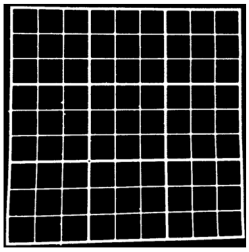

Figure 10: Result of the bit-wise xor operation- only the grid lines remain

3. We then perform a dilation on the grid to thicken the lines and make them clearer for contour detection. We also get the inverted result for later.

[3]https://numpy.org/
[4]https://www.tech-quantum.com
[5]https://scikit-image.org/

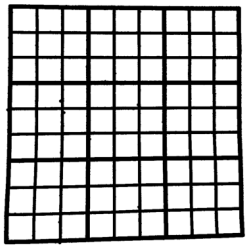

Figure 11: Dilation of grid lines. Inverted image displayed

4.3 Isolate the small squares of the Sudoku grid

Now we wish to isolate the 81 small squares in the grid.

1. The first step is to find the contours in the cropped image consisting only of the grid.

   Finding the contours in an image is an extension to the DIP course content. The process involves finding the boundary of objects and creating a 'curve' which joins all the points along the boundary[4]. This technique is used for object detection. The Skimage[5] function for detecting contours, used in this code, uses the 'marching squares' method to find the contours in an image.

   We store all contours which have an area in the range [200,500]. These boundary values were based on the average size of the small squares in the cropped images of the grids.

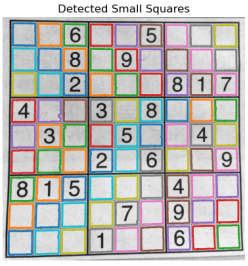

Figure 12: Contours of the small squares in the grid

2. We then need to sort the contours in the order in which they appear in the image (starting at the top

left corner, travelling right and then repeating this for every row). In order to do this, we create a list of sub-lists in which the first element of each sub-list is the contour and the second element of the sub-list is the top left corner of that contour. We then divide this big list into 9 groups. This groups all those sub-lists whose top left corners are most similar to each other in the x co-ordinate i.e collects all the contours which appear in the same row. We then sort the sub-lists in each group/row according to the y coordinate of the top left corner of the contour. This orders the row contours in order in which they appear (in order of increasing y co-ordinate- left to right).

Now we have detected the small squares of the Sudoku grid in the order in which they appear.

The rest of the techniques used in this subsection are from the DIP course

3. In the next bit of code we determine whether each small square is empty or occupied (it contains a number). We iterate through all the contours (of the small squares) and crop the inverted version of the original image to the shape and size specified by the co-ordinates of that specific contour. We then erode the image to get the smallest number of white pixels required to represent the contents of that block, we use this version of the image to work out the sum of the pixel values in the image. If the sum of the pixel values is in the range [0,100] (meaning there are few white pixels in the block and so it is unlikely that a number is in the block), then we classify that block as empty, otherwise we classify that block as occupied. We store these classifications in lists.

```
[['empty' 'empty' 'occupied' 'empty' 'empty' 'occupied' 'empty' 'empty' 'empty']
 ['empty' 'empty' 'occupied' 'empty' 'occupied' 'empty' 'empty' 'empty' 'empty']
 ['empty' 'empty' 'occupied' 'empty' 'empty' 'empty' 'occupied' 'occupied' 'occupied']
 ['occupied' 'empty' 'empty' 'occupied' 'empty' 'occupied' 'empty' 'empty' 'empty']
 ['empty' 'occupied' 'empty' 'empty' 'occupied' 'empty' 'empty' 'occupied' 'empty']
 ['empty' 'empty' 'empty' 'occupied' 'empty' 'occupied' 'empty' 'empty' 'occupied']
 ['occupied' 'occupied' 'occupied' 'empty' 'empty' 'empty' 'occupied' 'empty' 'empty']
 ['empty' 'empty' 'empty' 'empty' 'occupied' 'empty' 'occupied' 'empty' 'empty']
 ['empty' 'empty' 'empty' 'occupied' 'empty' 'empty' 'occupied' 'empty' 'empty']]
```

Figure 13: Lists storing whether each square is empty or occupied

We perform a dilation on that eroded image to thicken the numbers in order to improve the performance of the number prediction model through which we pass this image. We store these dilated images in another list.

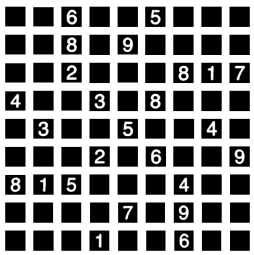

Figure 14: The extracted small squares from the grid displayed as individual images

4.4 Representing each square in the grid as text

We load in, build, train and save our machine learning model from TensorFlow[6].

We iterate through the list of images from the stage above and we save those images with unique file names and store the image paths in a list. This is needed in order to be able to pass the relevant images through the model to predict which digit is in each image (small square).

Finally, we loop through each of the small squares/images. We check if that square is empty or occupied: if that square is empty, we print a "-". If that square/image is not empty and is rather occupied, we read in the image using its image path we stored earlier, we convert it to gray-scale, normalise it and reshape it according to what is required by the model. We pass this image through the model and obtain a predicted value of the digit in the image. We print this value.

The result of this step is a text-base grid of the original Sudoku puzzle in the image.

[6]https://www.tensorflow.org/

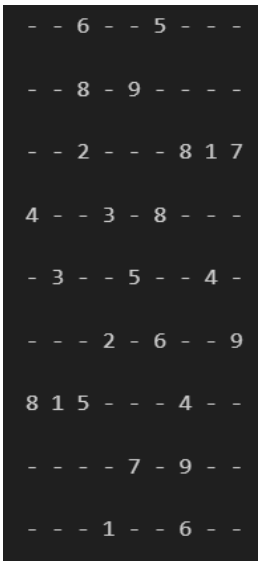

Figure 15: Output: text-based grid

5 Experimental setup

The data used to test this code consists of images of various Sudoku puzzles photographed from different newspapers and from various distances and angles. These images were obtained from a Sudoku data set on Github (Wicht und Hennebert [2014]) . The full data set consists of images with many different properties such as different lighting, angles, colours, grid style, noise etc.. However, we chose a subset of the images which matched the brief as 'images from a standardised source (for example the same newspaper, with the same font, scanned in exactly the same way each time, with even lighting)'.

To recognize the digits in the images extracted from the Sudoku puzzle, we employ a Convolutional Neural Network (CNN) model trained on the MNIST dataset[7], which is trained on images of handwritten digits.

We utilize the TensorFlow and Keras libraries for the digit prediction model. The steps to use the model include loading the MNIST data set[8], pre-processing the data, converting the labels to one-hot encoding, building the CNN model architecture, compiling the model with an appropriate optimizer and loss function, training the model on the pre-processed data set, and finally saving the trained model to a file. We utilised this model as it was proven to be accurate on the handwritten MNIST data set. Although the images in our data set do not contain handwritten numbers, but rather printed numbers, this was the best model we could find for this assignment. We do expect there to be some incorrect predictions for the digits, however for the purpose of this report it is sufficient. As the digit prediction was not a core part of the assignment we did not focus too much on it. However, this does present room for possible improvement of the algorithm in the future.

We pre-process the image we wish to feed into the model as required. We convert the image to gray-scale and resize the gray-scale image to be 28 x 28. We then normalize the image and reshape it to have the shape (1,28,28,1) as is required by the model. We pass this processed image into the model and obtain the predicted digit.

6 Analysis and Discussion

To assess the Sudoku extraction algorithm, we conducted real-world testing on a diverse range of images, replicating various scenarios commonly encountered in Sudoku puzzles. The test images included puzzles captured from different camera angles, images with background "noise" and puzzles with varying font styles.

Our objective was to evaluate the algorithm's performance in accurately extracting the Sudoku digits from these images. We deliberately selected images that represented common obstacles faced during Sudoku puzzle solving, ensuring a comprehensive assessment of the algorithm's capabilities.

During the testing process, the algorithm was applied to each image to extract the Sudoku grid and identify the individual digits. We compared the algorithm's extracted digits with the ground truth solutions for each puzzle, enabling us to determine the accuracy of the extraction process.

The results of our practical testing indicated that the algorithm consistently produced correct and reliable digit extraction across the varied images that we tested. These findings demonstrated the algorithm's effectiveness in handling real-world Sudoku puzzle examples.

[7] https://nextjournal.com/gkoehler/digit-recognition-with-keras
[8] http://yann.lecun.com/exdb/mnist/

6.1 Limitations

While we tried to make this algorithm generalise to as many Sudoku images as possible, there are several limitations to our code.

As stated in the project brief, this code focused on 'images from a standardised source (for example the same newspaper, with the same font, scanned in exactly the same way each time, with even lighting)'. And therefore, while we do have some variation in our data set with respect to content in the image, grid type, folded pages, odd angles etc. , this code does not generalise well to images taken with uneven, bad lighting with shadows, images in which a lot of excessive extra information is present, the image is blurry or unfocused, the grid is faded etc.. The code also has a problem with grids in which some squares were blue for example. While it did work for grids which had pale green squares, the images with the pale blue squares failed during the thresholding process.

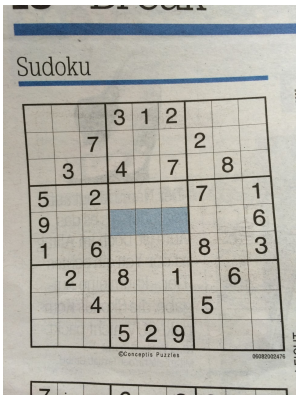

Figure 18: Image with blue blocks in Sudoku grid

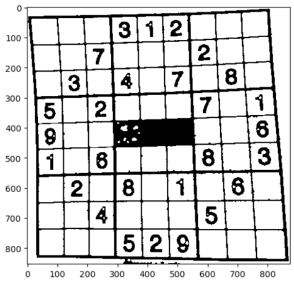

Figure 19: Incorrect thresholding of grid

Examples of images for which our algorithm is not successful

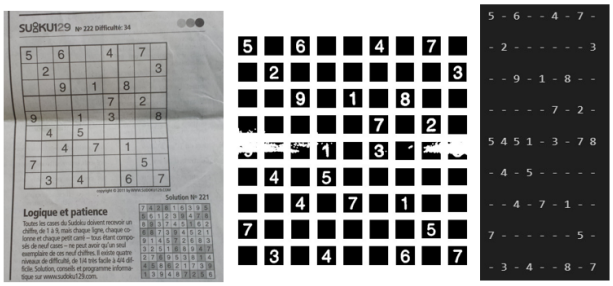

Figure 20: Effect of bent page on our algorithm

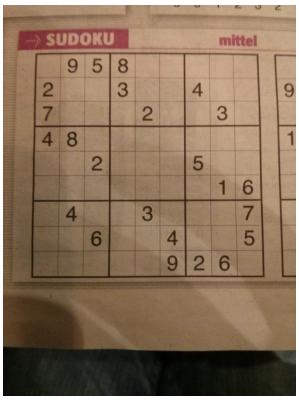

Figure 16: Image with shadows. This image fails at the thresholding step

Figure 17: Blurry image. Due to the dullness of the lines in the image, it fails at the hysteresis thresholding step

6.2 Possible Improvements

Sometimes the cell was incorrectly classified as empty or occupied when it was the opposite. This could be improved by using a more generalised approach as opposed to hard coding the sum of images as is done in our code. One could estimate the size of a square on the grid and then create a threshold based on the area of each square. One could take the ratio of white to black pixels and depending on that result as compared to the dynamic threshold, classify the cell as empty or occupied.

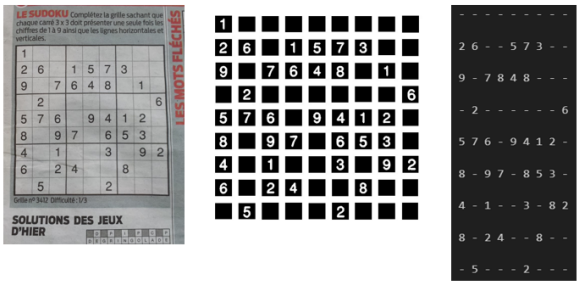

Figure 21: Incorrect empty/occupied classification leads to incorrect results in the text-based grid as can be seen by the missing one in the first square of the grid.

Where we used Otsu's thresholding, a local adaptive thresholding may have been better suited to account for the background noise. We tried to use this in our code however it did not work with the existing algorithm. In theory however, it could be used and the rest of the algorithm should be adjusted accordingly. We could have also tried to use a Sobel filter to detect initial edges before thresholding.

Additionally, we could add more steps in the pre-processing of the image in order to improve the performance of the proceeding operations. Such pre-processing could include removing noise by smoothing the image using something like Gaussian smooth and then sharpening the image so as to increase details and emphasise the lines and numbers. We could have also used histogram equalization to make steps such as thresholding more effective.

As mentioned previously, this algorithm focused on detecting the digits correctly in images of Sudoku puzzles which were scanned in consistently, from the same source with even lighting; and so it was not built to handle extreme changes in lighting, angles, shadows, blurriness and such. Therefore, this algorithm could be extended generally to accommodate such changes in image quality.

We could also improve the predictions of the model by rejecting any '0' predictions since 0 is not allowed in Sudoku grids. We could instead take the next most likely digit as the prediction from the model.

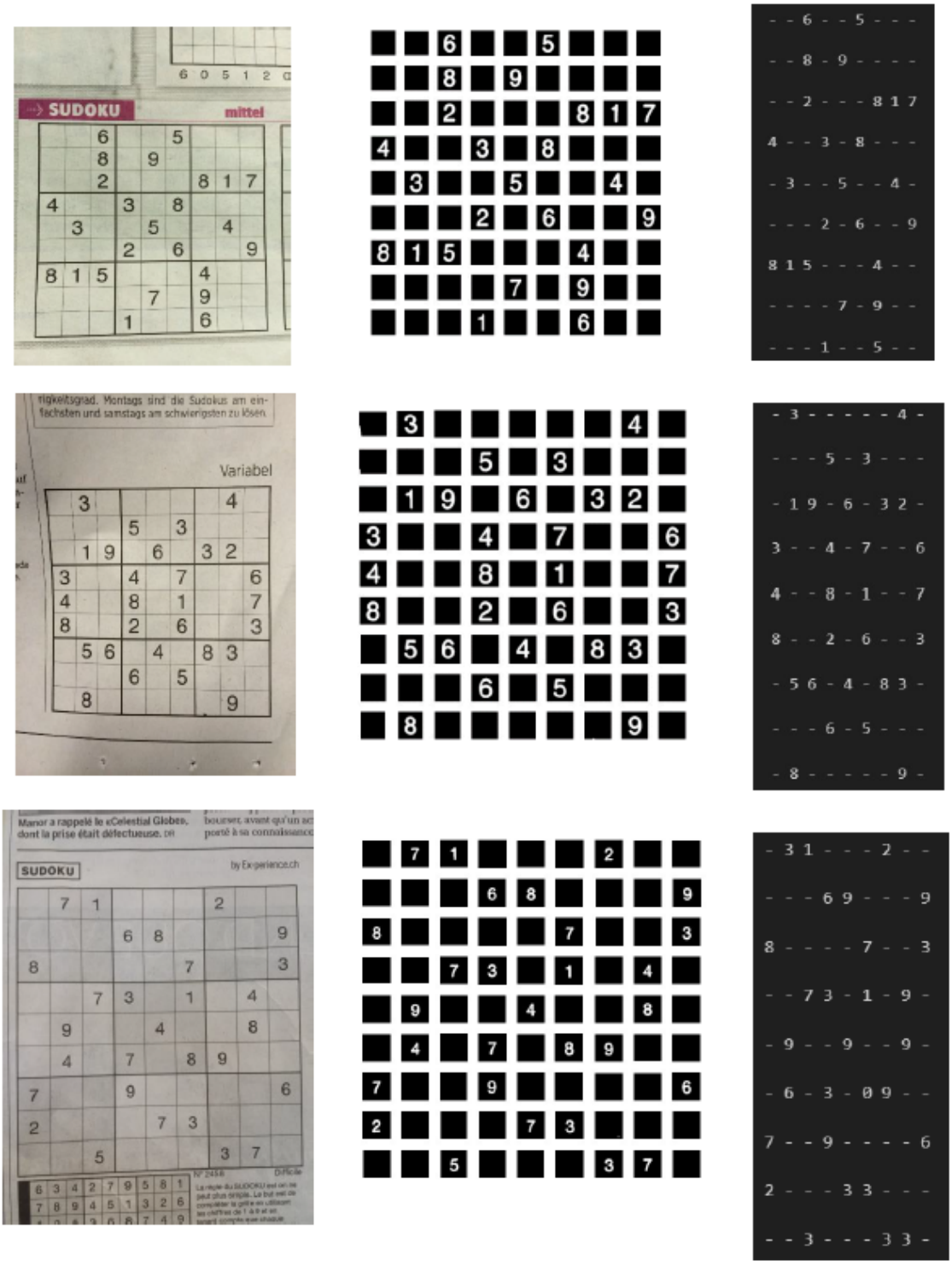

Figure 22: Successful digit extraction of 3 different Sudoku grids using our algorithm (NOTE: the model does not predict the digit correctly every time, even though we pass in good quality images shown in the middle grids)

7 Conclusion

In this report, we presented a comprehensive digit extraction algorithm for Sudoku puzzle images, so that the grid could be passed into existing Sudoku solvers as text. The algorithm employed various techniques, including preprocessing, grid detection, digit extraction, and digit recognition, to accurately identify and recognize individual digits from the puzzle grid. Experimental evaluation demonstrated the effectiveness of the proposed algorithm on standardised images. Future work may involve refining the digit recognition model and exploring additional image processing techniques to handle more challenging scenarios.

References

[Wicht und Hennebert 2014] WICHT, Baptiste ; HENNEBERT, Jean: Camera-based Sudoku recognition with deep belief network. In: *Soft Computing and Pattern Recognition (SoCPaR), 2014 6th International Conference of* IEEE (Veranst.), 2014, S. 83–88