Computer Vision - Sudoku Visual Information Extractor

ANDREI-CRISTIAN RUSU

November 28, 2021

1 Introduction

In this document you will find implementation details about the Sudoku Scanner application. To make everything as clear as possible, we'll go through each file, and explain most of the relevant functions, and the idea behind their usage.

2 Reading the data

2.1 Training

For training, the application expects a directory with the following structure:

```
training
└── [classic / jigsaw] // sudoku type
    ├── {index}.jpg // image
    └── {index}_bonus_gt.txt // answer file
```

Here is an example of an answer file:

```
o68ooo5oo
o542o8ooo
7ooo5o8oo
ooo491ooo
oooooo3oo
oooo32o41
o9o3oo168
oo6o2oooo
41oo7ooo2
```

The directory for the **training data** can be changed in *constants.py*.

2.2 Testing

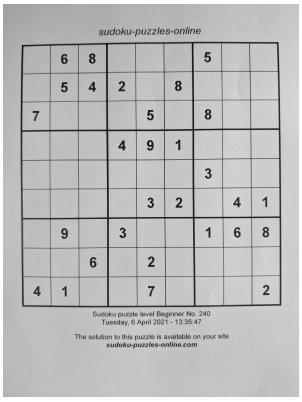
The folder structure should be the same - but of course, without the answer files.

3 Processing the image

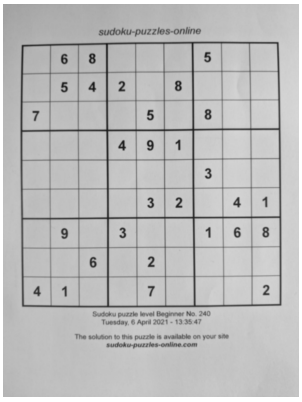
At first, every image goes through a pipeline of transformations, in order to bring the image to an easier version to work with.

The pipeline goes like this:

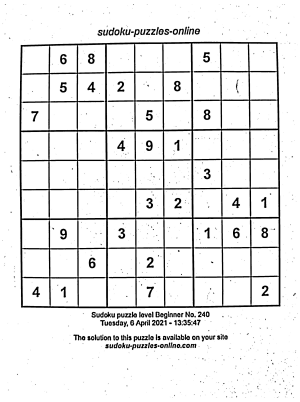
At this point, we need to find the contour, and try to get a *bird-eye view* of the image.



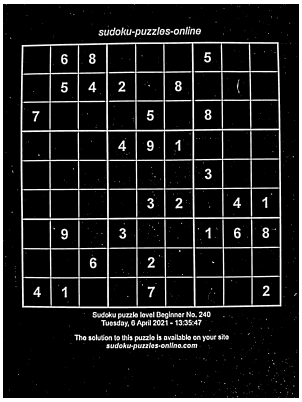
(a) Convert to grayscale



(b) Add gaussian blur



(a) Add adaptive threshold



(b) Invert

For this, we'll use find_contours, which will make use of *OpenCV's findContours* method. It works in the following way:

```
Find the contours of the image

Sort them by contour area

Loop through each contour
    Calculate the perimeter using arcLength

    Find an approximation of the contour using approxPolyDP

    If we find a contour with 4 vertices, return it

Raise an error
```

After we have the contour, we can apply *get_bird_eye_view*, which receives the image and the four corner points, representing the ROI, and returns the image as seen from above. Internally, it works in the following way:

```
Sort the corner points clockwise (top left, top right, bottom right, bottom left)

Calculate the width of the new image (i.e. max. distance between top left-right or botton

Calculate the height of the new image (i.e. max distance between left top-bottom or righ

Create a set of "destination points", to obtain a top-down view
```

Compute the perspective transform, using the destination points

Return the warped perspective

This is the result:

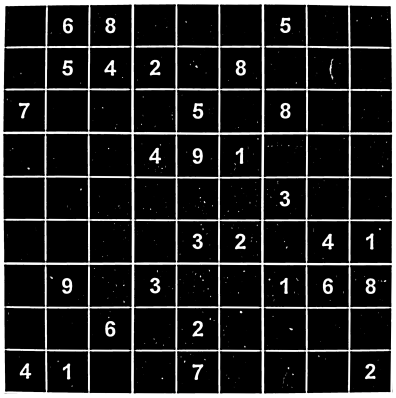


Figure 3: Bird eye view

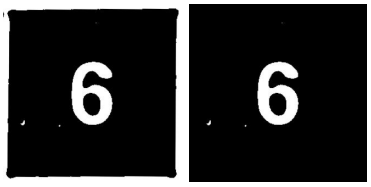
4 Extracting the data

4.1 Idea

At this point, we know that we have an image that should be (mostly) a square. For this reason, we can consider that a cell will have $size = (image\_size/9, image\_size/9)$. Using this, we can iterate through the image, using a patch-by-patch approach, and extracting the digits from every patch.

4.2 Implementation

For every patch, we need to do a little bit more processing.



(a) Initial cell image

(b) Remove border

(c) Find the biggest contour and create a mask

(d) Do a bit-wise AND between the mask and the cell

(e) Find the "bounding rect" and extract the ROI

There are two cases when this process can return *None* instead of the ROI:

1. We can not find a contour
2. We find a contour, but the filled percentage of the mask is less than a small threshold (i.e. 2%).

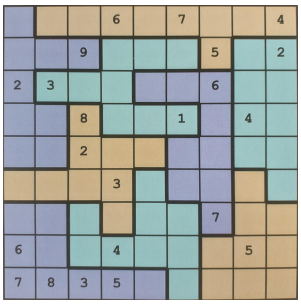
4.3 Okay, but what about Jigsaw?

For Jigsaw Sudoku's, there is another story - the boundaries. We need to be able to identify the regions for each cell in the game. For this, we will have to do a few things.

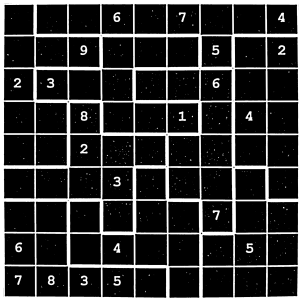
4.3.1 Keeping only the boundaries in each image

1.For this, we will use *keep_only_boundaries*, which receives the image and a boolean - is_color - which will be true only for color Jigsaw games. This boolean is needed because, based on a few tests, the parameters can't be the same for all three types of games.

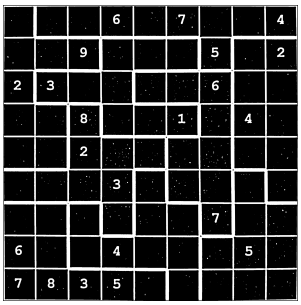
Here, we add a 10 pixels black padding to the whole grid, then we follow up with a few rounds of erosion / delation, ending with a threshold.



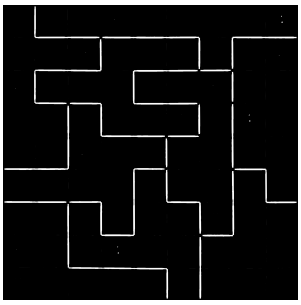
(a) Initial grid image



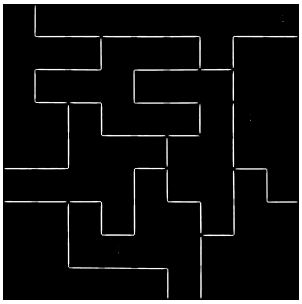
(b) After processing



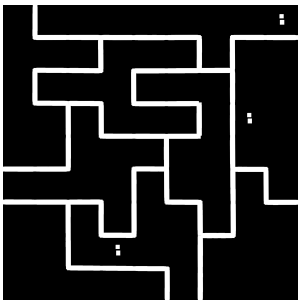
(c) After applying 10 pixels of (0, 0, 0) padding



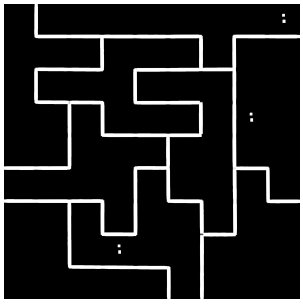
(d) After first round of erosion



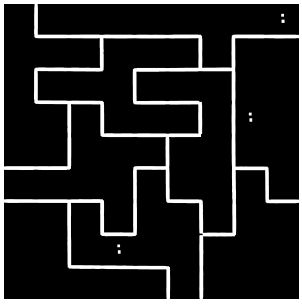
(e) After second erosion



(f) After first round of dilation

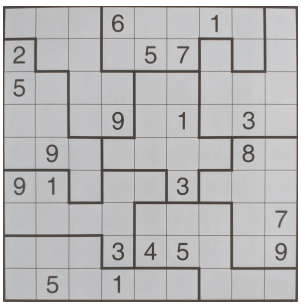


(g) After third round of erosion

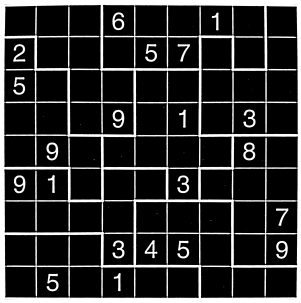


(h) After applying the final threshold

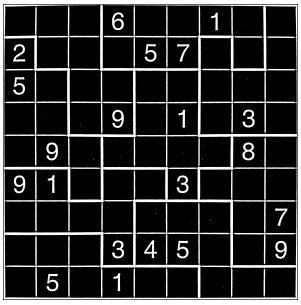
This is very similar for grayscale games:



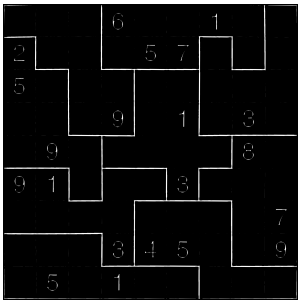
(a) Initial grid image



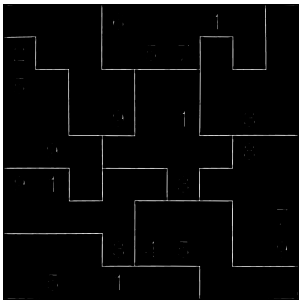
(b) After processing



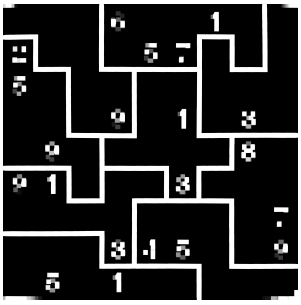
(c) After applying 10 pixels of (0, 0, 0) padding



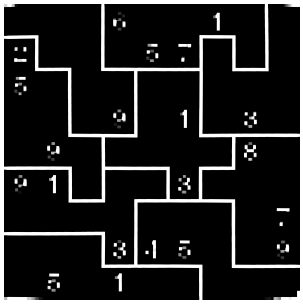
(d) After first round of erosion



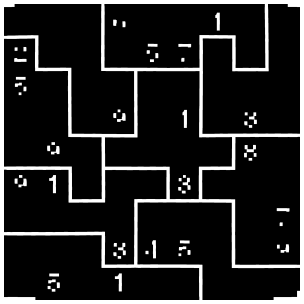
(e) After second erosion



(f) After first round of dilation



(g) After third round of erosion



(h) After applying the final threshold

Yes, it would seem like we still have a few traces of digits in the cells. But this won't be a problem.

4.3.2 Creating matrices for vertical / horizontal edges

We will need two matrices: *vertical_edges_matrix*, of shape $(9, 8)$ and *horizontal_edges_matrix* of size (8, 9). $M[i][j] = True$ will mean that there is a vertical edge on the i-th line, between columns j and $j + 1$, respectively that there is a horizontal edge on the j-th cell, between rows i and $i + 1$.

For these, we will use the same approach of going through patches, but this time, we won't start from the beginning of the cell, but rather from half of it. This way, we can try to identify vertical / horizontal lines.

For each such patch, we try to *erode* it with a **MORPH_RECT** structuring element, of size $(1, edge\_height)$ for vertical edges, respectively the same element, of size $(edge\_width, 1)$, for horizontal edges. (Note: $edge_width$ has been chosen to be 250, as it had the best accuracy, based on a few tests).

This way, after eroding the "cell" with such an element, if we still have white pixels, it means we have such a structure. Id est, we have a vertical / horizontal edge.

Examples:

(a) A patch without a vertical edge

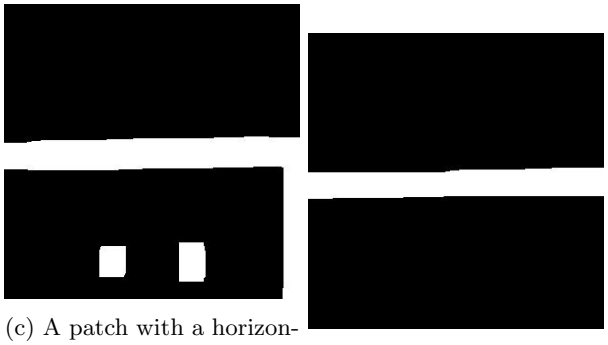
(b) After erosion

(c) A patch with a vertical edge

(d) After erosion

(a) A patch without a horizontal edge

(b) After erosion



(c) A patch with a horizontal edge

(d) After erosion

4.3.3 Creating the "boundary matrix"

Now that we have matrices for *vertical_edges* and *horizontal_edges*, we can create a "boundary matrix" M, where $M[i][j]$ = the region where the cell on the i_{th} row and j_{th} column belongs.

For this, we will first create a *neighbourhood matrix*, where $M[i][j] = [bool, bool, bool, bool]$, and $M[i][j][k]$ represents whether or not the respective cell has a top, right, bottom and respectively left neighbour ($k \in 0, 1, 2, 3$).

Our main function, *get_boundaries_matrix* will have a simple implementation - it iterates through all rows / columns uses an auxiliary function, *fill*, that does a *depth-first* technique to fill the patch, with the correct number.

After we have all these ingredients, we can start working on the classifier.

5 Extracting images for the digits (used by the Classifier)

5.1 Idea

For identifying the digits in a sudoku grid, instead of training a CNN, we will be using **template matching**.

It has proven efficiency, when the *font* that is used for the images doesn't change (which is true in our case).

Since we have two major types of sudoku, Classic and Jigsaw - with the catch that Jigsaw Sudoku's can be either color or grayscale - and every type uses different fonts for writing the digits, we need to extract and store images for every digit, for all three types of Sudoku's.

5.2 Implementation

For this, we make use of the training data. For each sudoku type, we will run the processing pipeline that transforms an image to a bird eye view, and extracts the ROI from all cells. We will create dictionaries, with $keys \in (1, 2, ..., 9)$, where the values are the ROI for the corresponding digit. Having these maps, we can easily use *template matching* to identify digits in our images.

6 Classifier

This is the simplest classifier ever. It's a function that, given an image and the *digit_to_image_map*, it iterates through all the key-value pairs in the map, template matching the given image with the image for the current digit.

It stores everything in a dictionary, where the key is the digit, and the result is the probability of the image being that digit.

At the end, it returns the key, for which the probability is the maximum.

7 Parameters Tuning

In this whole paper I never mentioned parameters and hyperparameters.

Rather than using a technique like *grid search*, they have actually been generated by trial and error. It felt overkill to try to setup such a search, since we're not dealing with a huge amount of them.

Everything can be found in the implementation.