SudokuRecognizer

Aim of Project

The aim of this project is to solve various boards of the game sudoku simply from images or videos taken of them. Solutions can then be projected back onto the board in the image in real-time. This could be useful as an application for phones to quickly get a solution to a sudoku by only taking a picture or video instead of manually entering the board.

Prerequisites

All the code for the project is located in the `src` folder. Before running the project, make sure Python 3 is installed (the project has only been tested with Python 3.7 and 3.10 but should work for other versions) and install the required libraries by running the command `pip install -r requirements.txt` from the project root folder.

Running The Project

The project can then be run using the command `python src/main.py`. It will prompt to either run on an image or a video. There are 5 images and 2 videos located in the `images` directory that can be used to test the program. The convolutional neural network will automatically use the weights located in the `model` directory if they exists, or will otherwise train the model for 100 epochs (training takes some minutes).

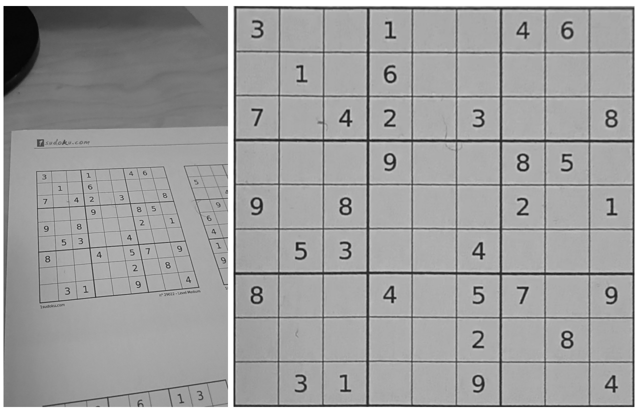
Recognizing Sudoku Boards

The task of recognizing Sudoku boards can be divided into two separate sub-tasks:

- Locating the grid and finding each individual grid cell
- Classifying the numbers contained in each non-empty cell

Locating Sudoku Grid

The first function for locating the Sudoku grid is the `find_grid` function in `main.py`. It takes a grayscale image of the Sudoku board as input and returns a the transformed Sudoku grid as a 900x900 image in addition to an inverse transformation matrix to project the solution back onto the image later. An example input and output from this function is shown below:



The first step of the process is applying a Gaussian filter with kernel size (7, 7) and $\sigma = 3$ to de-noise the image. Then, since the only important information is the Sudoku grid which is binary in nature, a binary threshold is applied. The threshold applied is an adaptive threshold to better binarize images with varying lighting, using a block size of 7 pixels.

```
NOTE: The parameters chosen for the filters were chosen since they gave the best
results for the camera and images that were used. For other cameras or images
with more variation in lighting other parameters might give better results.
```

Once the image is binarized, the next step is to find the 4 corners of the Sudoku grid, which is done in the `find_sudoku_corners` function. The first step of this process is finding the contours of the image, which is done using OpenCV's implementation of Suzuki's contour finding algorithm for binary images. Assuming that the image is centered on the Sudoku grid, the contour of the grid is then the largest of the contours (though in practice the second largest since the algorithm returns the contour of the image as the largest). This contour is then used to find the four corners of the grid by approximating it as a rectangle.

Having found the 4 corners of the grid, they can now be used to transform the grid to a 900x900 image of the grid by finding the 3x3 matrix of the perspective transform. This is done in the function `transform_grid`, which first finds the transformation matrix, inverts it for projection later, and then applies it to the image. Since the coordinates of the 4 points are known in both the source and destination, the `cv2.getPerspectiveTransform` function is used instead of `cv2.findHomography`.

The 900x900 image is then passed to the function `find_cells` which splits the grid into individual cells. This is done by first splitting the image into 9 rows and 9 columns before applying a binary threshold. The binary threshold is applied to easier separate empty cells from non-empty ones, and an adaptive threshold is not necessary since the lighting is not varying much inside each cell. Only the central part of the cell is used to avoid including grid lines in the resulting images, and the cells are returned as a list of images for the non-empty cells and *None* objects for the empty cells.

Classifying Numbers

Having found the individual cells of the Sudoku grid, the next step is classifying the numbers contained in the non-empty cells. This is done through the use of a convolutional neural network.

Neural Network Model

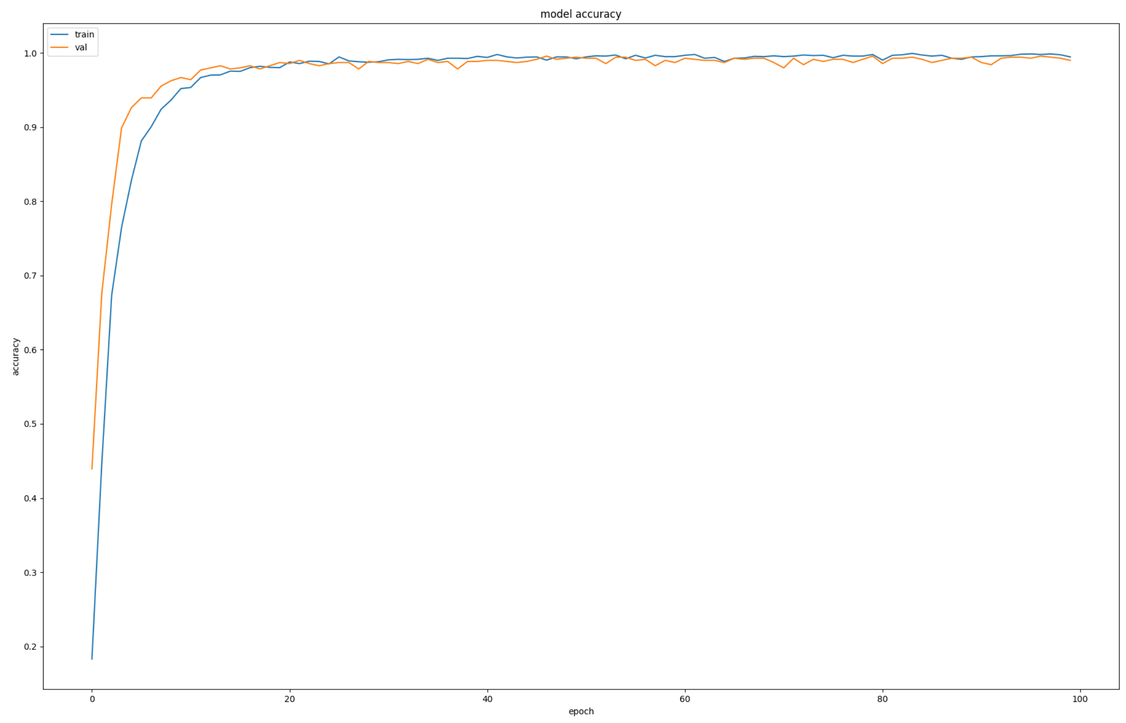
The network model can be found in the `get_model` function in the file `digit_model.py`. It uses two convolutional layers both with a kernel size of 3x3 and filter sizes 32 and 64. They are each followed by a max pooling layer to better extract features and dropout to reduce overfitting. The last layer before the output is a dense layer with 128 units. Since the neural network is solving a multi-class classification problem where the output is an integer, the `sparse_categorical_crossentropy` loss function is used.

Dataset

The first attempt at classifying the cells was done by training the network on the MNIST dataset which consists of hand drawn digits. However, the results were not good enough when classifying the printed digits of the Sudoku board, where the network especially had difficulty differentiating between 9's and 4's and between 1's and 7's. Therefore, the Printed Digits Dataset was used instead, which contains printed digits between 0 and 9, conveniently in the same format as the MNIST dataset. This gave significantly improved results.

Training Results

After training the network for 100 epochs, these were the accuracy results:



In the end, the model reached an accuracy of over 99% on the validation set.

Classifying

The cells are classified in the function `classify_grid` in `main.py`. This function iterates through all the non-empty cells, blurs them with a Gaussian blur to better fit the dataset and resizes them to 28x28. The pixels are converted to floating point numbers between 0 and 1 and the cells are classified by the model. The classified Sudoku grid is then returned as a numpy array.

Solving Sudoku

Having classified the Sudoku board it can now be solved. The code for solving the Sudoku board can be found in the file `sudoku_solver.py`. It is a simple recursive Sudoku solver that tries each valid option at each cell and backtracks when no valid option is longer available. It is by no means an optimal Sudoku solver, however the focus of this project was more on the computer vision and image processing parts.

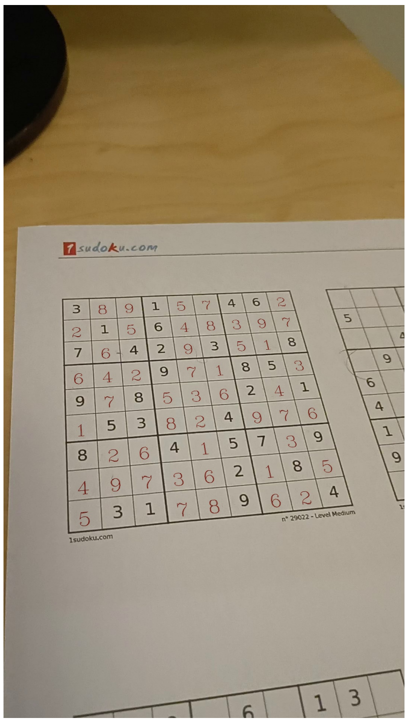
Projecting Solution

Having solved the Sudoku, the solution can now be projected back onto the original image. This is done in the function `project_solution` in `main.py`. This function iterates through all the elements of the solved Sudoku grid, and if it was not part of the initial numbers puts the corresponding digit onto an empty 900x900 image. Since the inverse of the homography found in `transform_grid` was stored, this is used to transform the 900x900 grid to the space of the original image grid, and then subtracted from the original image to overlay the numbers.

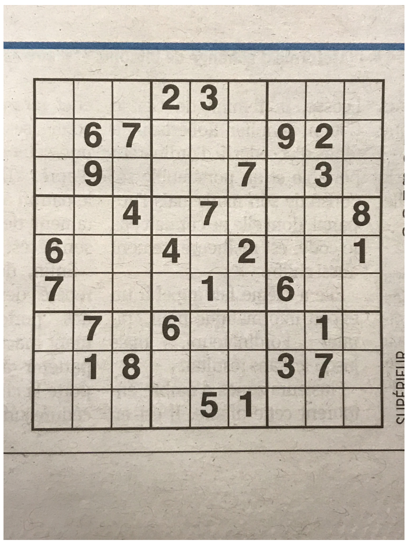
Conclusion

Image Results

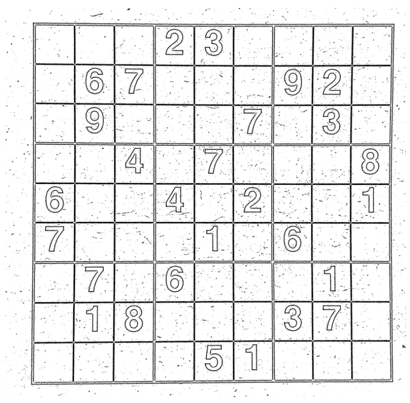
The results were very good, and the program was able to find and project the solution to every image it was tested on except 1. An example of the result of an image is shown below.



The image it was not able to solve was the following (located in `images/sudoku.jpg`):



The reason it failed seemed to be because the Gaussian filter and adaptive binary thresholding had parameters not suited for this specific image, as the result below has black spots inside the cells and non-solid digits:



			2	3				
	6	7				9	2	
	9				7		3	
		4		7				8
6			4		2			1
7				1		6		
	7		6				1	
	1	8				3	7	
				5	1			

Video Results

The results of the 2 videos it was tested on were very good, as it seemed to solve each frame where a complete Sudoku board was present. The main issue with the videos were performance related, where during playback the video would freeze for a moment before showing the solution to a Sudoku board. Through profiling it was found that the main bottlenecks were the recursive Sudoku solver (which can be seen in that more difficult boards makes it freeze for longer) and the classification. There is a performance-accuracy tradeoff in the neural network model where a simpler model could predict the digits faster (though with less accuracy), however for this project accuracy was deemed as more important.