Computer Vision Sudoku OCR Project

Abdullah Elrefaey
MagdElDin AbdalRaaof
Mark George
Noran Askar

README

Sudoku Visual Extractor and Solver

Abdallah Elrefaey

MagdElDin AbdalRaaof

Mark George

Noran Askar

Introduction

The project uses computer vision techniques like contrast manipulation, rotation, morphological operations, and Hough transforms to extract the Sudoku board, recognize the numbers using OCR powered by a hybrid model of rules-based detection and template matching, and solve the Sudoku puzzle if it is legal by using a backtracking algorithm. Developing the pipeline took hours of trial and error from figuring out how to preprocess as many images as possible with the same pipeline to testing different methods of board extraction and OCR.

To run it properly, we recommend having a folder called inputs for the images and use the folder for the templates.

Methodology

First Image Preprocessing

Rotation

Check

- We extract the edges using Canny edge detection
- Followed by extracting the straight lines using Hough Transform
- Detect all lines with an angle between 20 and 160 (aka the non vertical and non horizontal lines)

- Rotate the image based on the median of the angles of the lines with an angle between 20 and 160
- Rotation angle is by default equal to the median of the detected angles
- Rotation angle is subtracted by 90 in instances where the median is greater than 45 degrees to rotate relative to the vertical axis

Action

- Rotate around the center of the image
- Encode the rotation and translation to keep the center fixed
- Use nearest neighbor interpolation by default for the rotation for speed and to avoid blocky visual artifacts
- `warpAffine` allows us to rotate the image while maintaining parallel lines and linearity

Low Contrast (Washout) and Low Lighting

Check

- It finds the pixel intensity value of the bottom 5th percentile
- If the darkest 5% of pixels are brighter than the threshold we set, mark the image as "washed out"
- Then we measure the standard deviation of the pixel intensities to determine contrast
- If the standard deviation is below the contrast threshold, report low contrast
- Furthermore, we check if the image is too dark on average by calculating the average values of the image pixels
- If there is a large variance then uneven shadows are detected and action is taken

Action

- Dilate the image with a 7×7 kernel
- Apply a large median blur (21×21 kernel) to smooth out details and mimic the brighter parts of the background
- Calculate the absolute difference between the original image and the estimated background, isolating the foreground content (text, grid lines) from the lighting
- Stretch the pixel values to use the full 0-255 range
- After background removal, the image might only use a narrow range of values (e.g., 80-180)

- Normalization maps the darkest pixel to 0 and brightest to 255, maximizing contrast
- Use contrast limited adaptive histogram equalization (CLAHE) to enhance contrast using a 10x10 kernel with a maximum level of 3.8 to limit noise amplification

Noise

Check

- Create a smoothed version using median blur
- It gets the absolute difference pixel by pixel between the smoothed and the original versions
- Pixels that differ a lot from their local median are marked as noise
- If the ratio of noisy to total pixel is greater than the threshold, we mark the image as noisy

Action

- We apply median blur iteratively with progressively larger kernels until the noise ratio is below the threshold

Laplacian Variance (Sharpness vs Blurriness)

Check

- We apply the Laplacian operator with a kernel size
- We calculate the variance of the Laplacian values
- Sharp images have high variance while blurry images have low variance

Action

- Measure sharpness using the check function
- If the variance from the check is beyond the threshold then the image is blurred using a Gaussian blur with a 3x3 kernel

Sobel Variance

Check

- Check the combined Sobel (`sqrt(sobelx**2 + sobely**2)`)
- If Sobel variance is too low, perform action

Action

- Begin by applying a Gaussian blur with a 5x5 kernel on an image copy
- Sharpened is approximately 1.5x the original image - 0.5x the blurred image to achieve sharpening

Binary Thresholding

Check

- We extract the image histogram then calculate the image's entropy (measure of information)
- If the entropy is higher than the entropy threshold then the image has many shades of gray and isn't close to purely black and white

Action

- We apply adaptive Gaussian thresholding
- If the percentage of dark pixels is below 5% or above 50% then we fall back to using Otsu thresholding (aka a single boundary below which all pixels are considered one color and above which all pixels are considered another color)

Repairing The Board

Check

- We extract all contours in the binary image and filters for contours with more than 300 pixels to try and identify the grid structure
- This isolated line structure is then slightly dilated to strengthen weak connections.
- Using Canny edge detection followed by Probabilistic Hough Line Transform, it identifies individual line segments
- The Hough transform parameters are tuned to find lines at least 30 pixels long with gaps up to 30 pixels, which helps detect fragmented lines
- We count the fragments by first identifying if they are horizontal or vertical then determine if they are a fragment by whether or not they are smaller than a third of the expected length/width of a vertical/horizontal fragment

- If more than 8 fragments are detected then action needs to be taken

Action

- We use morphological dilation using a vertical and a horizontal kernel to fill the gaps
- Opening then dilation are used to clean up the excess
- If more than 1.5% of pixels are bright after repair (indicating excessive noise), a median blur is applied
- The repaired lines are merged back with the original image using bitwise OR

Pipeline Operation

- Check and correct for rotation
- Check for and remove noise
- Check if there are shadows, if the image is dark, or if the image is washed out and fix that as defined above
- Check Sobel variance and increase sharpness if needed
- Check if the lines are broken and try to repair them as defined above

Grid Detection

- The images are loaded, converted to grayscale, and blurred slightly to remove noise
- Adaptive thresholding is used to isolate the grid
- We extract horizontal lines with a 40x1 kernel and extract vertical lines with a 1x40 kernel and combine them to get the grid
- Detect line segments in the grid mask
- Parameters require lines at least 100 pixels long with gaps under 10 pixels
- The result is a list of (x1, y1, x2, y2) coordinates for each detected line segment
- We group nearby parallel lines together by averaging their positions
    - For example, if the same grid line is detected as 3 slightly different positions (y=100, y=102, y=98), it clusters them into a single line at y=100
- `find_grid_corners_from_hough(lines, img_shape)` Analyzes all detected lines to find the grid boundaries
    1. Classifies each line as horizontal (angle near 0° or 180°) or vertical (angle near 90°)
    2. Clusters similar lines to get the main grid lines
    3. Takes the **outermost** lines (first and last in each direction) as the grid boundaries
    4. Returns the four corner points where these boundaries intersect

- `order_points(pts)` Orders four corner points in consistent clockwise order: top-left, top-right, bottom-right, bottom-left. Uses sum and difference of coordinates to identify each corner.
- `perspective_transform(image, corners)` Applies perspective transformation to "unwarp" the grid:
  1. Orders corners consistently
  2. Calculates the maximum width and height from corner distances
  3. Uses the larger dimension to create a square output
  4. Maps the original corners to a perfect square's corners using `cv2.getPerspectiveTransform()`
  5. Warps the image to the new perspective with `cv2.warpPerspective()`
- If corner detection using Hough transform fails, fall back to **contour detection**:
  - Finds all contours in the grid mask
  - Sorts by area (largest first)
  - Looks for the first 4-sided polygon approximation
  - Uses those points as corners

Making the Image Square

- There are 3 strategies used for this portion
- Each strategy is a fallback for the prior strategy

Strategy 1: Reuse Previous Corners (`extract_corners_from_previous`)

Use the corner visualization from the previous step of the pipeline pipeline:

- Extract red circles (HSV color detection for red)
- Find their centroids using moments
- Validate that exactly 4 corners were found
- Check that the grid is not too small and that it is roughly square

Strategy 2: Parse Hough Overlay (`extract_lines_from_hough_overlay`, `corners_from_hough_overlay`)

- If the corner images don't work, then the grid found from the Hough transform images are used as base

- The green color is extracted
- We redetect the lines from scratch
- We pick the most evenly spaced 10 lines horizontally and vertically to match a Sudoku grid
- `select_best_uniform_10()` does this by
  - Trying all possible windows of 10 consecutive lines from detected clusters
  - Scoring them based on the ratio of maximum and minimum spacing between consecutive lines and the standard deviation of the spacings
  - Windows that are too uneven (max/min ratio > 2.2) are rejected
- We use linear regression for precision
- We find the corners where the lines intersect

Strategy 3: Detect from Scratch (Fallback)

If reuse and Hough parsing both fail, performs full detection:

Step 1: Grid Line Detection (`detect_grid_lines`)

- Applies adaptive thresholding
- Uses morphological operations (erosion + dilation) with directional kernels to extract horizontal/vertical lines separately
- Combines them and applies Hough line detection
- Has a fallback to detect on raw threshold if grid mask fails

Step 2: Line Classification (`classify_lines`)

Analyzes each detected line segment:

- Horizontal: angle < 10° or > 170°
- Vertical: angle 80°-100°
- Records vertical angles for rotation estimation

Step 3: Rotation Correction (`estimate_rotation`, `apply_rotation`)

Analyze vertical line angles:

- Require at least 6 vertical lines for reliability
- Calculate mean angle deviation from 90°

- Only apply rotation if:
  - Standard deviation < 2.5° (lines are consistent)
  - Correction is 2°-12° (not too small/large)
- Rotates the entire image and re-detects lines

Step 4: Clustering & Smart Selection

- Clusters nearby parallel lines (threshold: ~1/80 of image dimension)
- We pick the most evenly spaced 10 lines horizontally and vertically to match a Sudoku grid like before

Step 5: Line Completion (`complete_lines`)

If fewer than 10 lines detected:

- Use detected lines as anchors
- Interpolate missing lines with uniform spacing
- Fall back to dividing image into 9 equal parts if very few lines found

Step 6: Validation & Fallbacks

- Check if derived corners are reliable using:
  - **Area ratio**: Grid should cover > 35% of image
  - **Aspect ratio**: Should be 0.75-1.35 (roughly square)
  - **Spacing uniformity**: Max/min spacing ratio < 2.5
- If unreliable, tries two final fallbacks:
  1. **Contour detection** (`contour_outer_corners`): Finds largest 4-sided polygon
  2. **Bounding box** (`bounding_box_corners`): Uses min/max coordinates of all grid pixels

Helper Functions Explained

`select_best_uniform_10()` - The key innovation:

- Tries all possible windows of 10 consecutive lines from detected clusters
- Scores each window based on:
  - **Uniformity**: Ratio of max/min spacing between consecutive lines

    - **Stability**: Standard deviation of spacings
- Rejects windows where max/min ratio > 2.2 (too uneven)
- Returns the most uniformly-spaced subset

This solves the problem where extra lines (from noise, thick grid lines detected as doubles, or table edges) would throw off the grid.

`spacing_uniform()`:
Validates that line spacing is consistent (max/min ratio < 2.5)

`enhance_warped()`:
Post-processing on the final warped image:

1. CLAHE for local contrast enhancement
2. Canny edge detection
3. Reinforces edges by adding them back (15% weight)
4. Unsharp masking for final sharpening

`warp_to_square()`:
Performs perspective transformation to 450×450 square

Drawing Black Lines Where the Grid Should Be

- We detect lines in much the same way as back when detecting the grid and then draw black lines on top of it

Complete Grid Lines

- We use Otsu thresholding to make the numbers white and the background black
- We find connected components and keep those within reasonable threshold and then proceed to smooth out the image and fill any gaps
- Finally, we invert the image again to go back to black text with a white background

Draw Clean Sudoku Grid

- Begin by loading templates of each number from 0 to 9
- Then Apply binary thresholding for inversion

- Then use opening and closing to clean the template image
- Extract just the digit using contouring
- Center the image using warping
- Resize to a standard 200x200
- Process done to achieve consistency with the main Sudoku board being processed

Extract and Save Cells

- Divide the image into 81 cells based on image dimensions
- Enlarge each cell by 4x
- Use cubic interpolation to maintain edge details while smoothing the image
- Save each cell in a directory for the image

Preprocess cell for digit detection

- Convert to grayscale
- Denoise using N1 means denoising
- Use CLAHE to improve contrast with a clip limit of 2.5 and a kernel of 4x4
- Try various thresholding methods like Otsu, Adaptive Gaussian, and Adaptive Mean then select a particular method based on density (given density is neither too high nor too low)

Cleaning Binary Cell Image

- Use opening and closing to remove noise from the cell image like with the template image
- Remove the edges very carefully to avoid cutting off digits

Check if a Cell is Empty

- If image density is too low or too high then the cell is empty
- Too low means the cell is empty
- Too high a density means the cell has garbage
- Just in case, we also check if there is only 1 connected component in the image (the background)
- We also check the size of the largest component
- If it is too small or too big then it is noise to be ignored

Find Content Bounds

- Collect all non-zero pixels
- Get the bounding rectangle

Center Cell Content

- Assuming the image passes the checks that determine whether or not the cell is empty, begin centering
- Pad the image to avoid clipping the digit
- Calculate the shift necessary and calculate the translation matrix to apply the warp to center the image

Extract Digit Component

- Extracts all contours in an image
- If a contour is too big or too small, it is not counted as a digit
- There are minimum size constraints and aspect ratio constraints to help filter noise as well
- Store tuple of contour, area, and index for each candidate
- Select the largest valid contour
- Draw the selected contour on a purely white background
- Find the smallest bounding box containing the entire contour
- Add some padding to the image to prevent the image from being too tight by having the pad be 10% of the smaller dimension (or 2 pixels if 10% is 1 pixel)
- Count holes using the internal contours and only count them if the area of the hole accounts for over 2% of the area

Compute Digit Features

- Calculate the aspect ratio where different numbers have different aspect ratio ranges
- Split the image into 6 regions (top, bottom, middle, left, right, center)
- Calculate the density of each region (number of white pixels / total pixels)

OCR

- Find content bounds

- Add padding to content to avoid clipping
- Calculate the transformation to make the ROI centered
- Center the ROI
- Resize to template size
- Compute template matching score
- Then there is feature based classification
- We use a decision tree format
- The biggest determining factor is the number of holes
- Then the narrowness (aspect ratio)
- Then whether more pixels are in the top compared to the bottom or in the left compared to the right
- How we use rules-based recognition vs template matching depends on template matching confidence
- Very high template matching confidence makes that our primary approach
- If there is a strong confidence in template matching, but not enough to warrant exclusive use
    - We check if there is a gap between the template matching winner vs the runner up. If the gap is large, take the template matching result
    - If the gap isn't large, but template matching and rules based agree, then the situation is settled
    - If otherwise rules are very confident (above 90%) then the rules override
    - Otherwise resort to template matching
- If template matching confidence is less than 50% but more than 40%
    - The gap necessary to blindly trust template matching is larger
    - Agreement between template and rule approaches remains the immediate fallback
    - If the second place instead agrees with rules, choose that
    - Then check if the third place matches
    - If template is significantly more confident than rules, choose template
    - If rules are significantly more confident than template, choose rules
    - Otherwise default to template
- If template matching is less than 40% but greater than 30%
    - Check if rules are more than 80% confident and choose that outcome if so
    - If any of the top 3 results agree with rules and its probability is greater than 25% then choose the agreement
    - Else choose the template if it is significantly more confident than rules
    - Otherwise default to rules
- In cases of weak template matching confidence
    - Choose rules if they are more than 70% confident or if the best template match has less than 20% confidence

- Choose template if rules have a less than 70% confidence but template has over 20% confidence

OCR Pipeline

- Apply all these steps in order to get the numbers and create a perfect image with the recognized numbers
- Use a function to run the OCR on every image in a file path

Sudoku Solver

- Check if the puzzle layout is legal before solving, abort if not
- Use a simple backtracking algorithm that operates recursively to fill the puzzle
- Undo if no step works
- Keep undoing until the next choice becomes available
- Stop when either the board is solved or all solutions are exhausted

Results

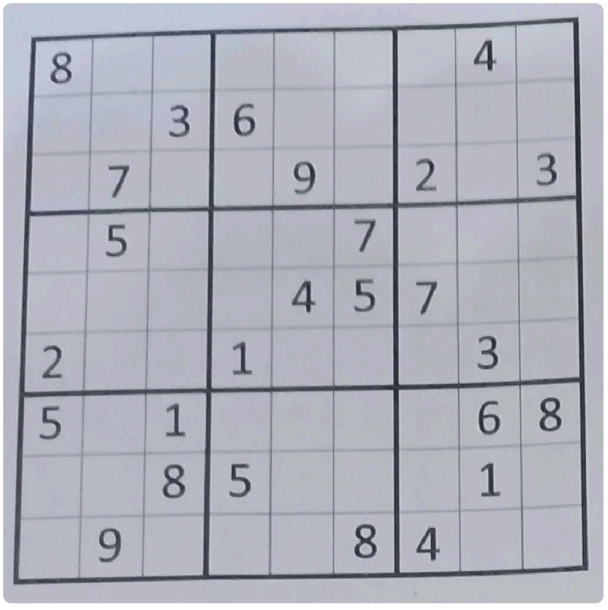
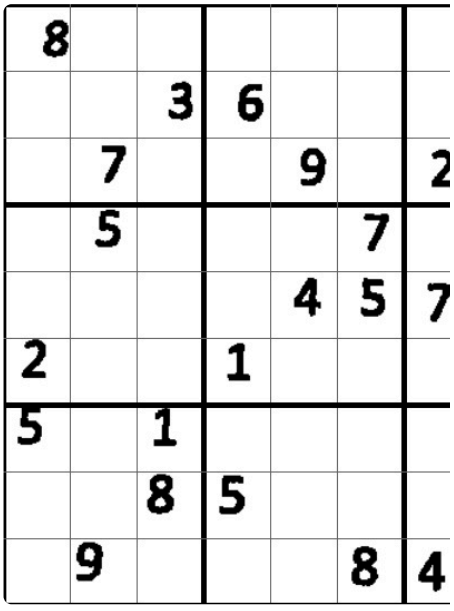
Image Number	Original	Processed
1		

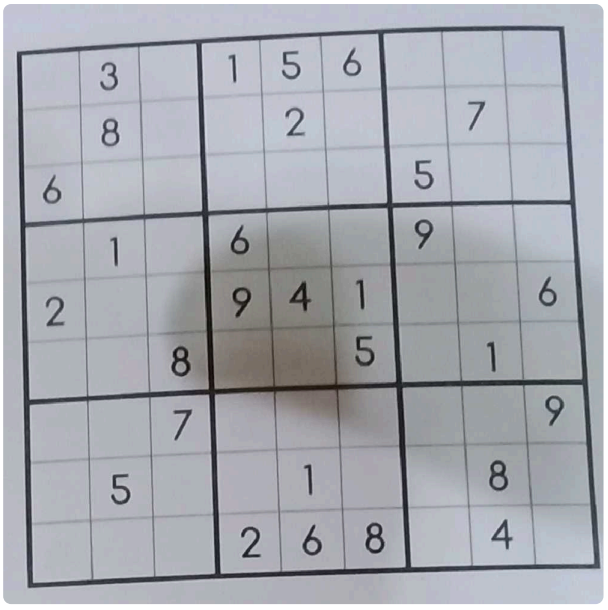
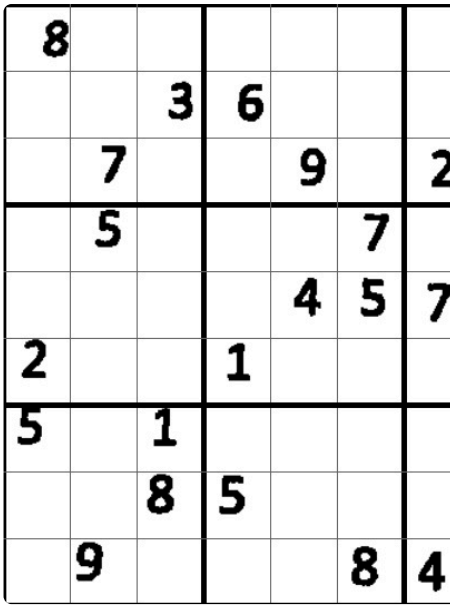
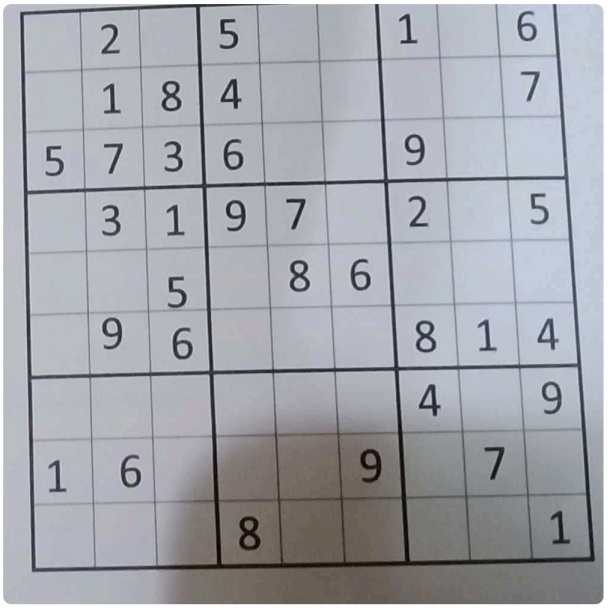
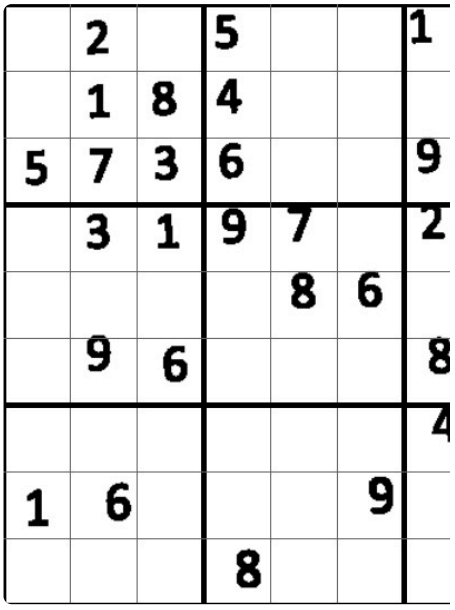
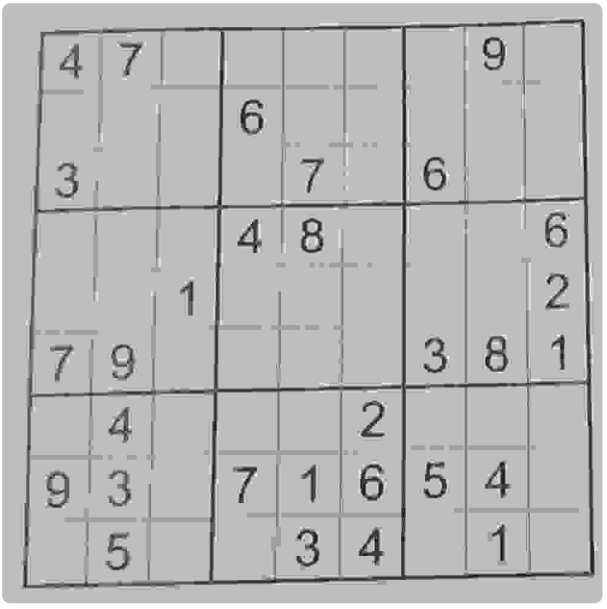
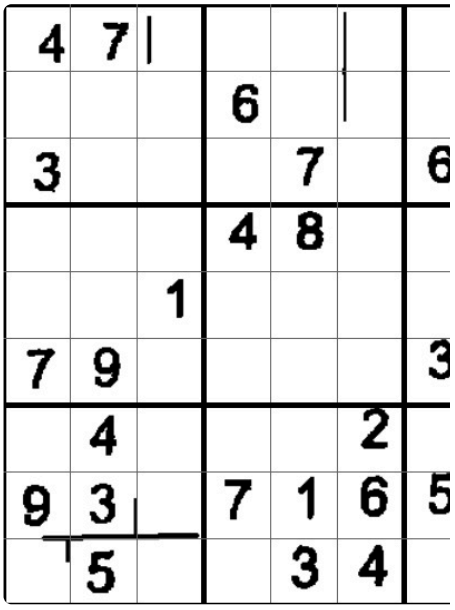
Image Number	Original	Processed
2		
3		
4		

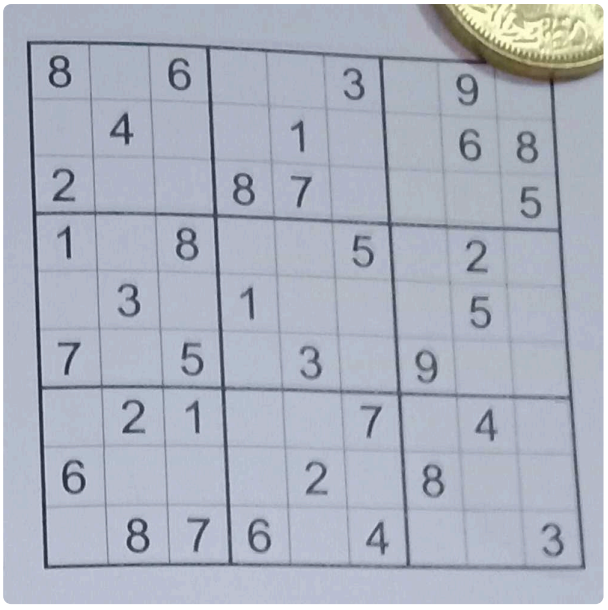
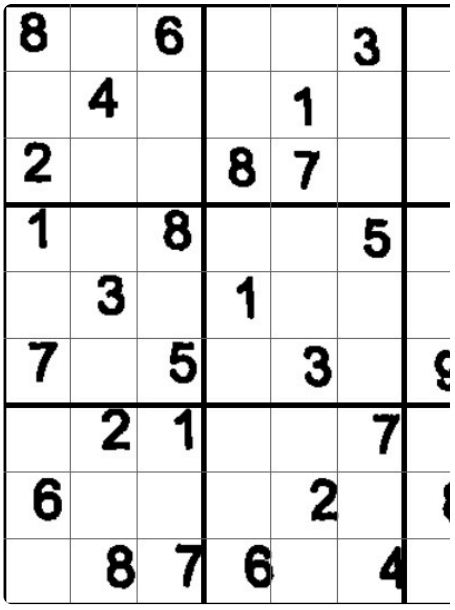
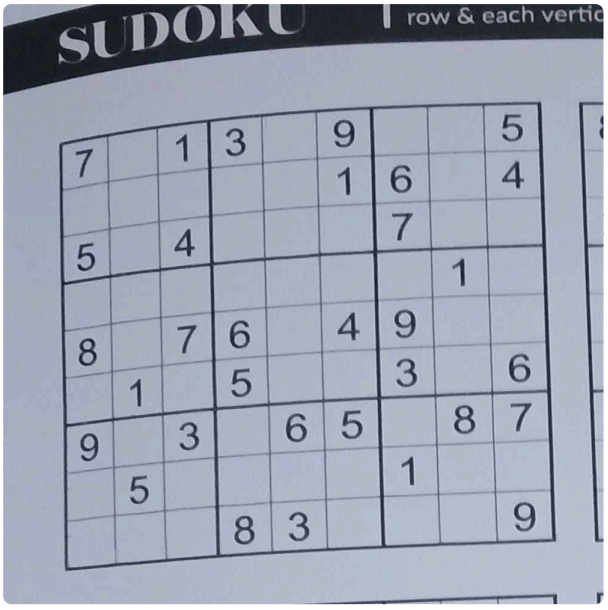
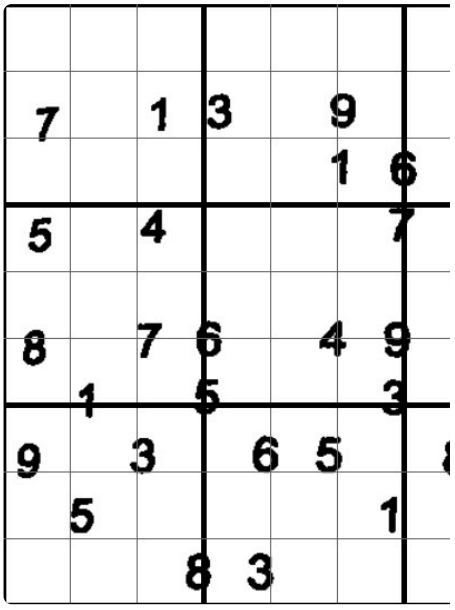
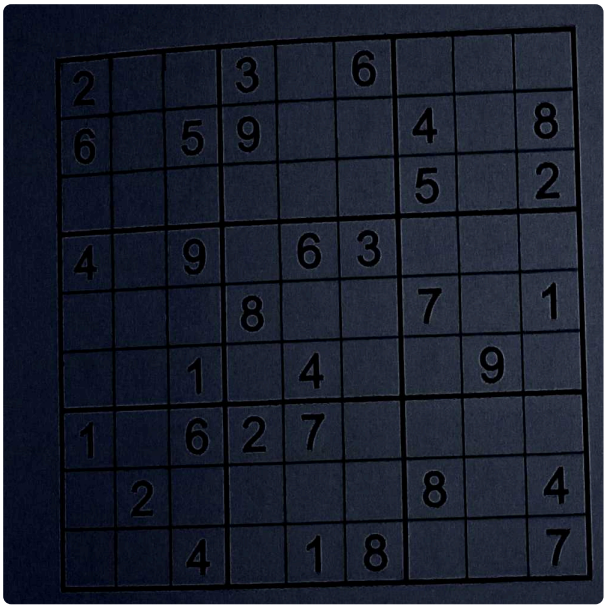
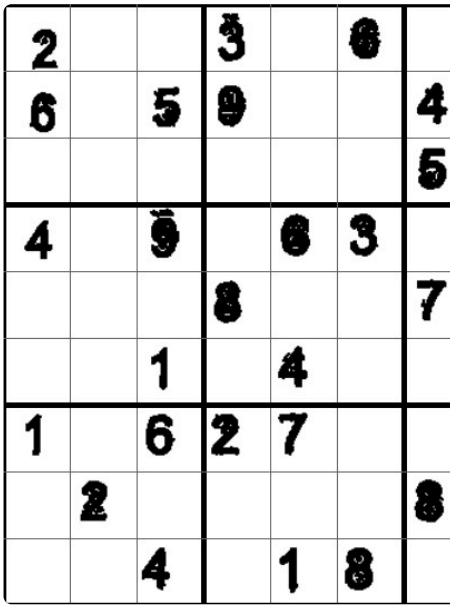
Image Number	Original	Processed
5		
6		
7		

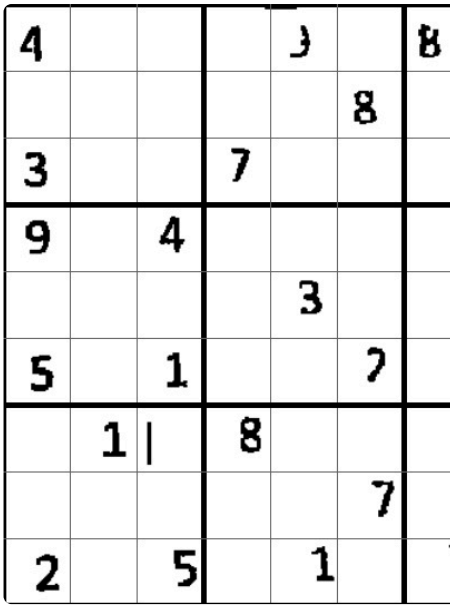
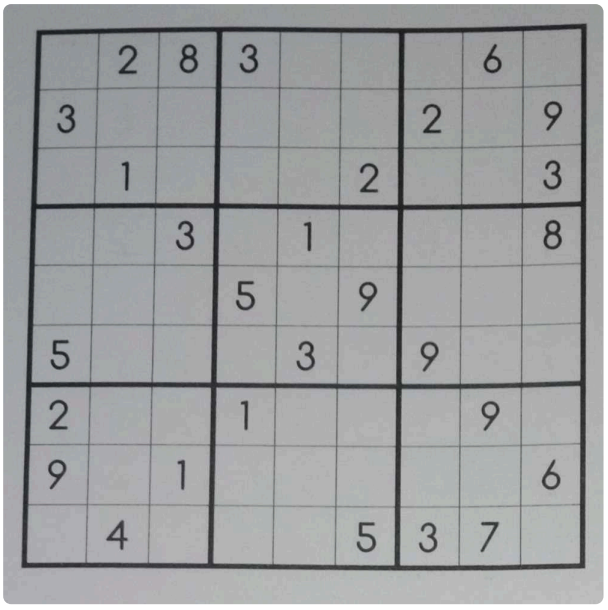
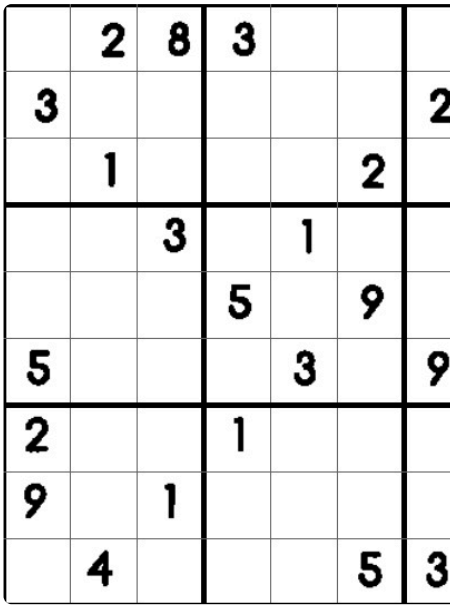
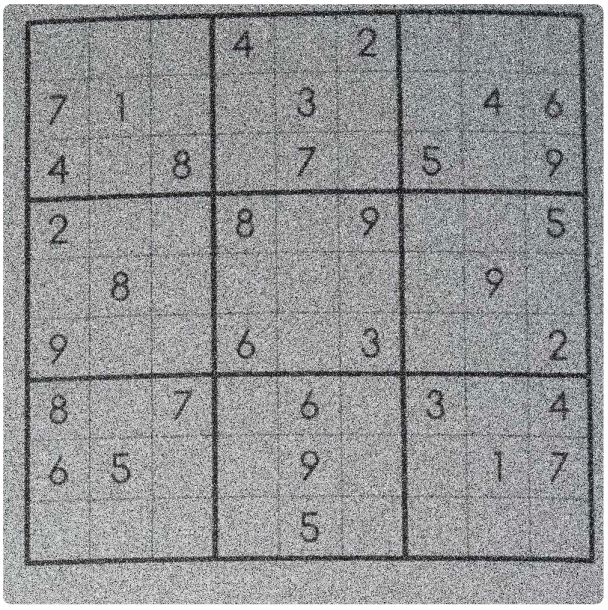
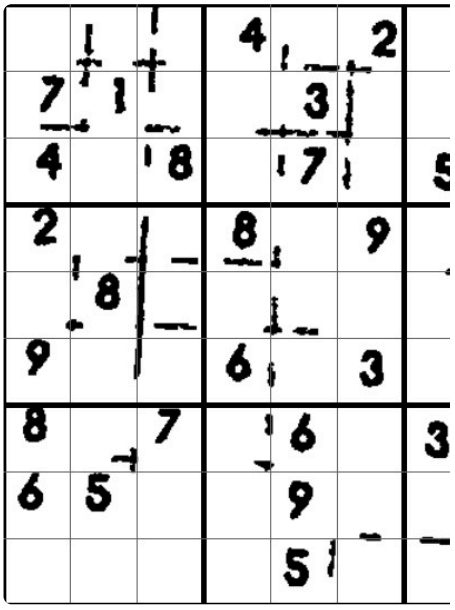
Image Number	Original	Processed
8		
9		
10		

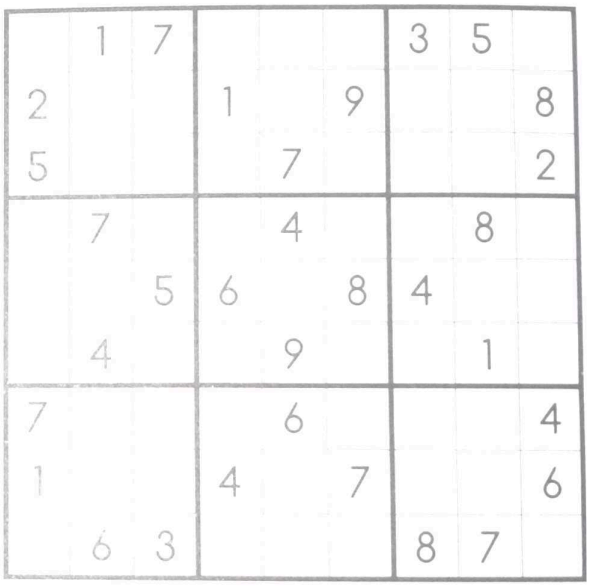
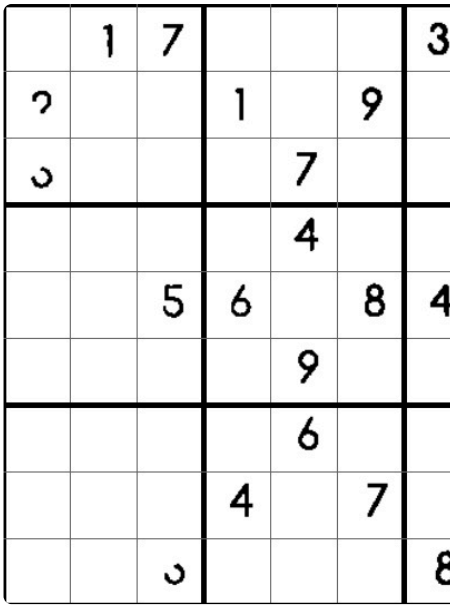
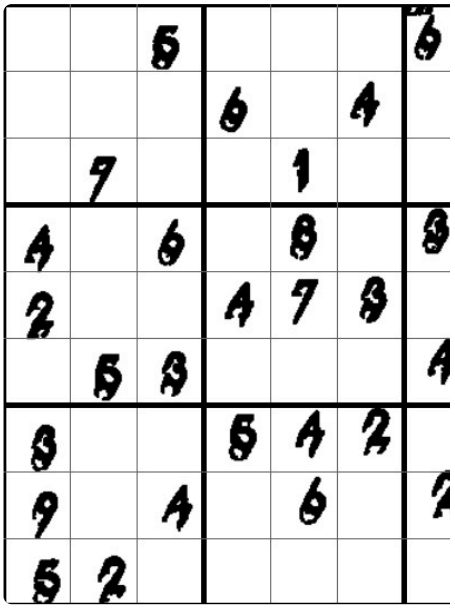
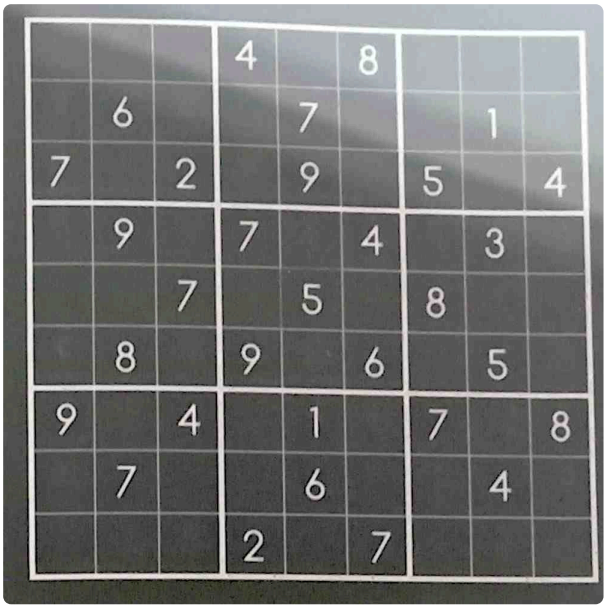
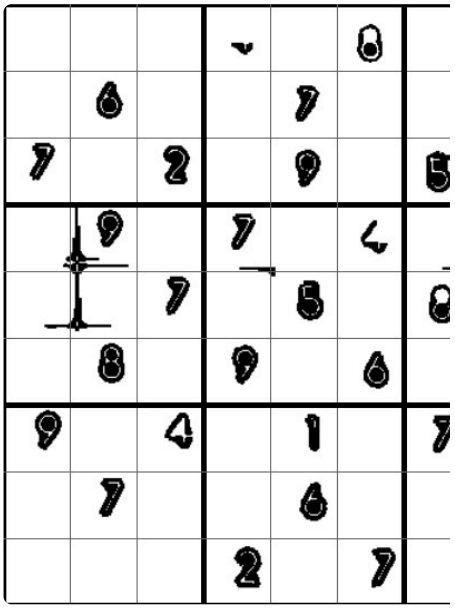
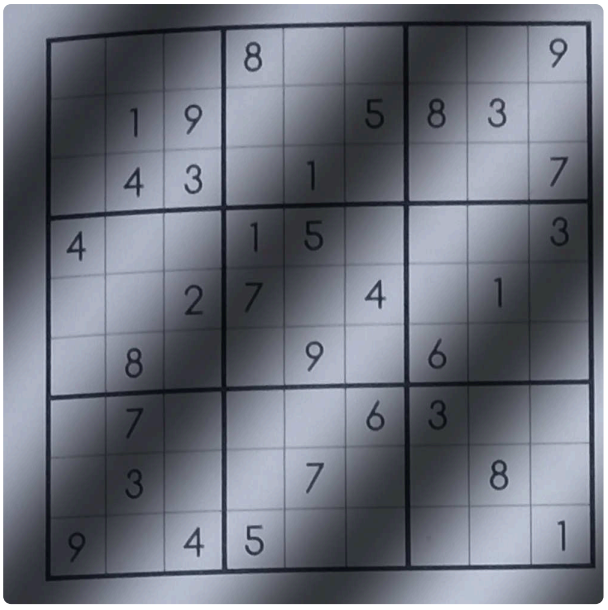
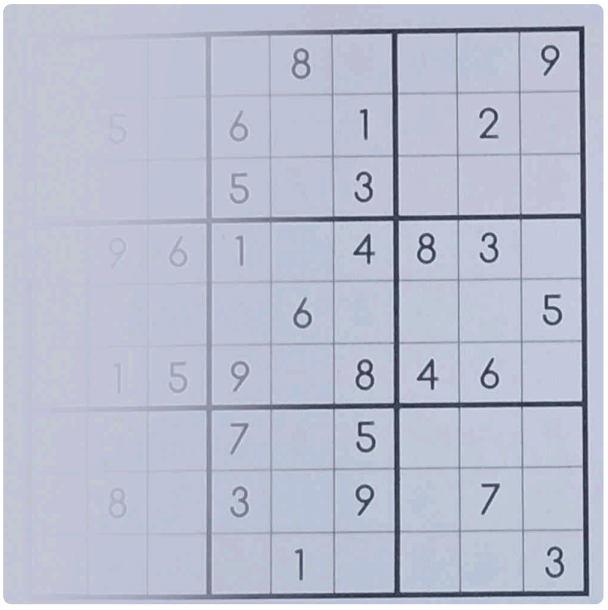
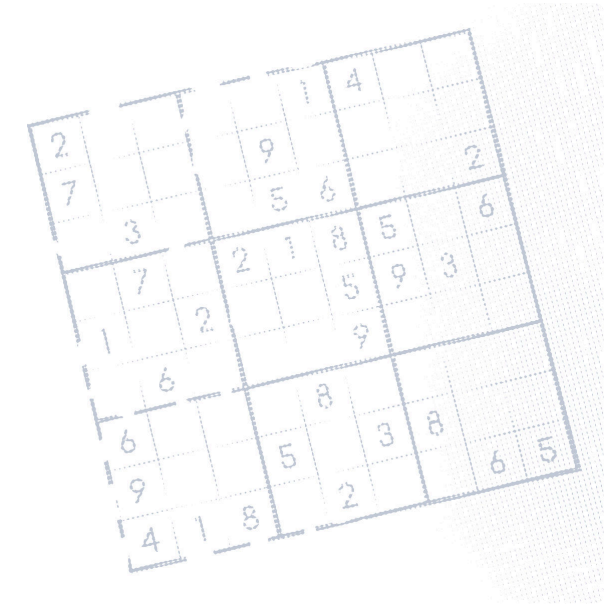
Image Number	Original	Processed
11		
12		
13		

Image Number	Original	Processed
14		
15		
16		

References

- LLM generation
- https://efcms.engr.utk.edu/ef230-2023-08/modules/matlab-algorithms/iPhone%20Sudoku%20Grab%20%20How%20does%20it%20all%20work%20.htm