ECE 415: Image Analysis and Computer Vision I
Image Scanning Sudoku Solver

Professor Ahmet Enis Cetin

Stefan Mijalkov
Department of Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, United States
smijal2@uic.edu

Hamza Waseem
Department of Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL, United States
hwasee3@uic.edu

***Abstract*—This document is a synthesis of the final project for the Image Analysis and Computer Vision course. The project focuses on core image analysis methodologies and its real world application using Python and widely used deep learning library tensorflow.**

***Index Terms*—deep learning, image analysis, computer vision**

I. Introduction

The purpose of this project is to create a software that takes an image of a Sudoku grid as an input, solves the grid, and writes the solution back to the image.

II. Method Description

A. Procedure

To achieve the desired output, we decided to go with the flowchart shown in Fig. 1. The program takes a partially completed image of a sudoku grid. We then resize the image to 1200x1200. This size is experimentally determined, because it was producing the best results. Then we convert the image to grayscale, and binarize to make it black and white. Usually the sudoku grids have black digits, and white background. Then we invert the image to make the background black and the digits white. The previous step is important for the training phase to match the MNIST dataset.

The next step is to find the square-like shapes using an edge detection filter and separate the cells. Each cell of the sudoku grid is saved as a new image maintaining the same order and size that matches the MNIST dataset. We feed the extracted cells in the pre-trained convolutional neural network to predict the digit. The outputs are stored in a 9x9 Numpy matrix. At this point, we have the complete image mapped into an integer matrix. Then we apply a backtracking algorithm to get a solution. Reading from the solution matrix, we map the integer values to the corresponding labeled images of the digits. Finally, we fill up the empty grid cells in the original image with the mapped digits from the solution matrix.

The flowchart below describes the steps in order:

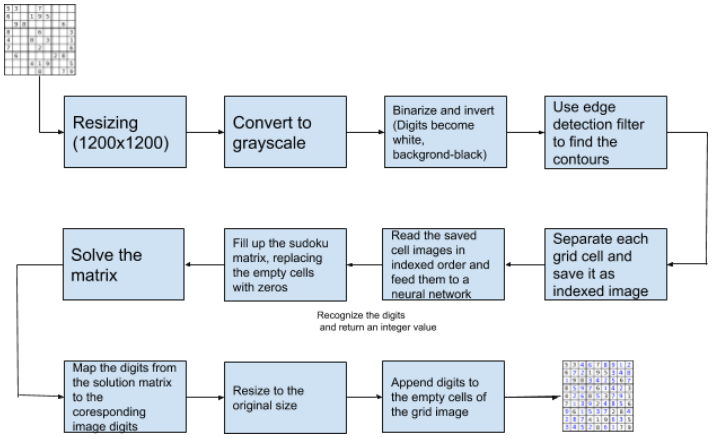

Fig. 1. Sudoku solver software flowchart

III. Experiment Results

Below are the images of the extracted cells from the first row of the original grid:

Fig. 2. Extracted Cells

These are the images of the labelled digits that are used in the final stage to write back the solution.

Fig. 3. Labelled Digits

*Note: When the program is run, the input is the default sudoku.png grid. To try with different sudoku grids modify the path to the image.

Here is what the terminal output looks like when we run the program:

```
(1200, 1200, 3)
(1200, 1200, 3)
All cells scanned
Checksum
[[1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1]]
Sudoku matrix:
[[5 3 0 0 7 0 0 0 0]
 [6 0 0 1 9 5 0 0 0]
 [0 9 8 0 0 0 0 6 0]
 [8 0 0 0 6 0 0 0 3]
 [4 0 0 8 0 3 0 0 1]
 [7 0 0 0 2 0 0 0 6]
 [0 6 0 0 0 0 0 2 8 0]
 [0 0 0 4 1 9 0 0 5]
 [0 0 0 0 8 0 0 7 9]]
Solved Matrix
[[5 3 4 6 7 8 9 1 2]
 [6 7 2 1 9 5 3 4 8]
 [1 9 8 3 4 2 5 6 7]
 [8 5 9 7 6 1 4 2 3]
 [4 2 6 8 5 3 7 9 1]
 [7 1 3 9 2 4 8 5 6]
 [9 6 1 5 3 7 2 8 4]
 [2 8 7 4 1 9 6 3 5]
 [3 4 5 2 8 6 1 7 9]]
```

Fig. 4. Terminal Output

For ease of use, we also get a solution image that is the modified image of the original sudoku table. This resultant image fills in the empty cells of the original sudoku.png image.

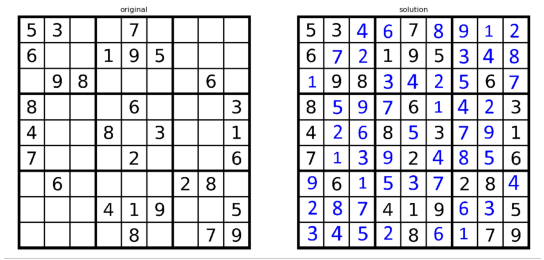

Fig. 5. Solved Image

IV. Conclusion

In conclusion, image processing together with machine learning is a very powerful tool for computer vision and automation. Sudoku solver is a simple concept and example of how useful this area of study is in real-life applications. Similar techniques can be applied to various different projects with the desired requirements a user may have. This allows for easy, fast, and user-friendly applications.

The system however is not perfect and cannot guarantee 100% accuracy. If only one digit is wrongly classified by the neural network, the whole solution will be wrong. Similarly, if the image is blurred, or has a very low resolution, additional image processing filters should be applied to get the desired results. The program at its current state is not capable of extracting sudoku grids from images where there are other objects present unless the grid is the main focus of the image. A possible fix to improve the accuracy is to train a neural network using a different synthetic digits dataset instead of the standard MNIST handwritten digits dataset.

V. Contribution of each member

Both partners equally shared the responsibilities of this project. We discussed various topics for our final project and ended up with two contenders, Sudoku Solver and Facial Recognition. We created a Github repository to work on our code and then combined and tested the final program. The report aspect of the project was distributed the same way. We divided the workload into equal portions and finished our section for the report.

We chose this project as our main project because it goes in depth regarding the principles we have learned throughout the course, and we implement it by ourselves while not heavily depending on a library to do most of the work for us. Since facial recognition is a topic that requires a lot of in depth knowledge about certain topics, we were dependent on a library that does the recognition, whereas we compare the results using different models and tolerance levels for facial recognition. We will be using that project as our extra credit option.

References

[1] Browser-Based Augmented Reality Sudoku Solver using TensorFlow and Image Processing, URL: https://www.youtube.com/watch?v=cOC-ad0BsY0

[2] Edward Pie, Automatic Number Plate Localization, URL:https://www.youtube.com/watch?v=UgGLo_QRHJ8&t=3321s

APPENDIX

sudoku_solver.py

```
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt
from matplotlib import image
from PIL import Image
import math
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
import glob


#Path
current_directory = os.path.dirname(os.path.abspath(__file__))
image_path = os.path.join(current_directory , 'sudoku.png')
image = cv2.imread(image_path)
im_copy = image
print(image.shape)
image = cv2.resize(image,(1200,1200))
print(image.shape)
cell_size = image.shape[0]//9

#To create a new directory for saving the extracted cells from the sudoku table
current_directory = os.path.dirname(os.path.abspath(__file__))
final_directory = os.path.join(current_directory, r'extracted_cells')
if not os.path.exists(final_directory):
    os.makedirs(final_directory)

#Add neural network for digit recognition
#Open extracted_cells folder, read image one by one -> recognize -> write to the matrix
model_path = os.path.join(current_directory, 'saved_model/my_model12')
new_model = tf.keras.models.load_model(model_path)

#final matrix
sudoku_matrix = np.zeros((9,9), dtype='int')
check_bool = np.zeros((9,9), dtype='int') #works like a boolean checksum -> when all cells are extracted becomes 9x9 matrix of Logic 1's
mask = np.ones((9,9), dtype='int')

#to make it grayscale
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

#to plot 2 images side by side
def plot_images(img1,img2, title1, title2):
    fig = plt.figure(figsize=[15,15])
    ax1 = fig.add_subplot(121)
    ax1.imshow(img1, cmap="gray")
    ax1.set(xticks=[], yticks=[], title=title1)

    ax2 = fig.add_subplot(122)
    ax2.imshow(img2, cmap="gray")
    ax2.set(xticks=[], yticks=[], title=title2)
    plt.show()

#for giving the correct name in the extracted cell files
def calculate_index(indexes):
    return str(indexes[1]) + str(indexes[0])

# to get rid of accidental frames in the extracted pictures and make it easier for recognition
def blendFrame(image):
    for i in range(3):
        image[i,:] = 0
        image[27-i,:] = 0
    for i in range(7):
        image[:,i] = 0
        image[:,27-i]=0
    return image

#resize to 28x28 to match the MNIST dataset pictures
def resize_tf(img):
    basewidth = 28
    wpercent = (basewidth/float(img.size[0]))
    hsize = int((float(img.size[1])*float(wpercent)))
    img = img.resize((basewidth,hsize), Image.ANTIALIAS)
    np_img=np.asarray(img)
    return np_img

#treshold some pixel for more clear view
def treshold(image):
    b = np.where(image < 140)
    c = np.where(image>=180)
    image[b] = 0
    image[c] = 255
    return image

#crop the center of the image to make the digit biger on the 28x28 frame
def cropND(image):
    image = image[:,13:126-13]
    image = image[13:126-13,:]
    return image

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) #make it gray
inverted = np.invert(gray) #invert to black background -> white number
contours = cv2.Canny(gray, 20,200)  #contours
cnts, new = cv2.findContours(contours.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
cnts = sorted(cnts, key=cv2.contourArea, reverse=True)  #sort by intensity of contours
image_copy = image.copy()
_ = cv2.drawContours(image_copy, cnts, -1, (255,0,255), 2)



cell = None
for c in cnts:
    perimeter = cv2.arcLength(c,True)
    edges_count = cv2.approxPolyDP(c, 0.02*perimeter, True)
    if(len(edges_count)==4):
        x,y,w,h = cv2.boundingRect(c)
        if(w>180 or h>180 or h<100 or w<100):
            print("Ignore")

        else:
            cell = image[y:y+h, x:x+w]
            cell=cropND(cell)
            cell = Image.fromarray(np.uint8(cell))
            resized = resize_tf(cell)
            resized = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY) #make it gray
            cell = np.invert(resized)
            cell = blendFrame(cell)
```

```
            cell = treshold(cell)
            indexes = [math.floor(x/w), math.floor(y/h)]
            if(indexes[0]>8):
                indexes[0]=8
            if(indexes[1]>8):
                indexes[i]=8
            else:
                check_bool[indexes[0]][indexes[1]] = 1
                cell_name = os.path.join(final_directory, calculate_index(indexes) + ".jpg")
                cv2.imwrite(cell_name, cell)
            if(np.array_equal(mask,check_bool)):
                print("All cells scanned")
                break
print("Checksum")
print(check_bool)

if(not np.array_equal(mask,check_bool)):
    print("Failed to scan all the cells in the grid")
    exit()
cells = []
files = glob.glob(os.path.join(final_directory,"*.jpg"))
files.sort()
for filename in files:
    img = cv2.imread(filename)
    img = rgb2gray(img)
    cells.append(img)
cells = np.array(cells)
cells.reshape(81,28,28)


#new_model.summary()
predictions = new_model.predict(cells)
i=0
for pred in predictions:
    perc = max(pred)
    if(perc*100>95):
        value = np.where(pred==perc)
        sudoku_matrix[int(i/9),i%9]=value[0]
    i+=1

print("Sudoku matrix: ")
print(sudoku_matrix)
to_fill = (sudoku_matrix==0)*1

empty_loc=[0,0]

def checkRow(sudoku_matrix, row_indx, num):
    row = sudoku_matrix[row_indx,:]
    if num in row:
        return True
    return False
def checkColumn(sudoku_matrix, col_indx, num):
    column = sudoku_matrix[:,col_indx]
    if num in column:
        return True
    return False

def checkBox(sudoku_matrix, row_indx, col_indx, num):
    sr = row_indx//3*3
    sc = col_indx//3*3
    box = sudoku_matrix[sr:sr+3,sc:sc+3]
    if num in box:
        return True
    return False

def isSafe(sudoku_matrix,ri,ci,num):
    return (not checkRow(sudoku_matrix,ri,num) and
            not checkColumn(sudoku_matrix,ci,num) and
            not checkBox(sudoku_matrix,ri,ci,num) )

def findEmpty(sudoku_matrix,empty_loc):
    for i in range(9):
        for j in range(9):
            if(sudoku_matrix[i,j]==0):
                empty_loc[0]=i
                empty_loc[1]=j
                return True
    return False

#solution with backtracking
def solve(sudoku_matrix, empty_loc):
    if(not findEmpty(sudoku_matrix,empty_loc)):
        return True
    else:
        row=empty_loc[0]
        column = empty_loc[1]
    for num in range(1,10):
        if(isSafe(sudoku_matrix,row,column,num)):
            sudoku_matrix[row,column]=num
            if(solve(sudoku_matrix,empty_loc)):
                return True
            sudoku_matrix[row,column]=0
    return False

check_sol = solve(sudoku_matrix,empty_loc)
if(not check_sol):
    print("No solution found")
    exit()
print("Solved Matrix")
print(sudoku_matrix)

dig_directory = os.path.join(current_directory, r'digits')
files = glob.glob(os.path.join(dig_directory,"*.jpg"))
files.sort()
print(files)

def findDigit(files, digit):
    for filename in files:
        l = len(filename)
        name=filename[l-6:]
        if(name == "d"+str(digit) + ".JPG"):
            d=cv2.imread(filename)
            d=cv2.resize(d,(100,100))
            # cv2.imshow("d",d)
            return d

def writeSolution(sudoku_matrix, to_fill, image_copy):
    for i in range(9):
        for j in range(9):
            if(to_fill[i,j]==1):
                d=findDigit(files,sudoku_matrix[i,j])
                image_copy[i*130+30:i*130+30+100, j*130+30:j*130+100+30,:]  = d

writeSolution(sudoku_matrix,to_fill,im_copy)
plot_images(image,im_copy,'original','solution')

cv2.waitKey(0)
cv2.destroyAllWindows()
```