

DLIP Final LAB Report

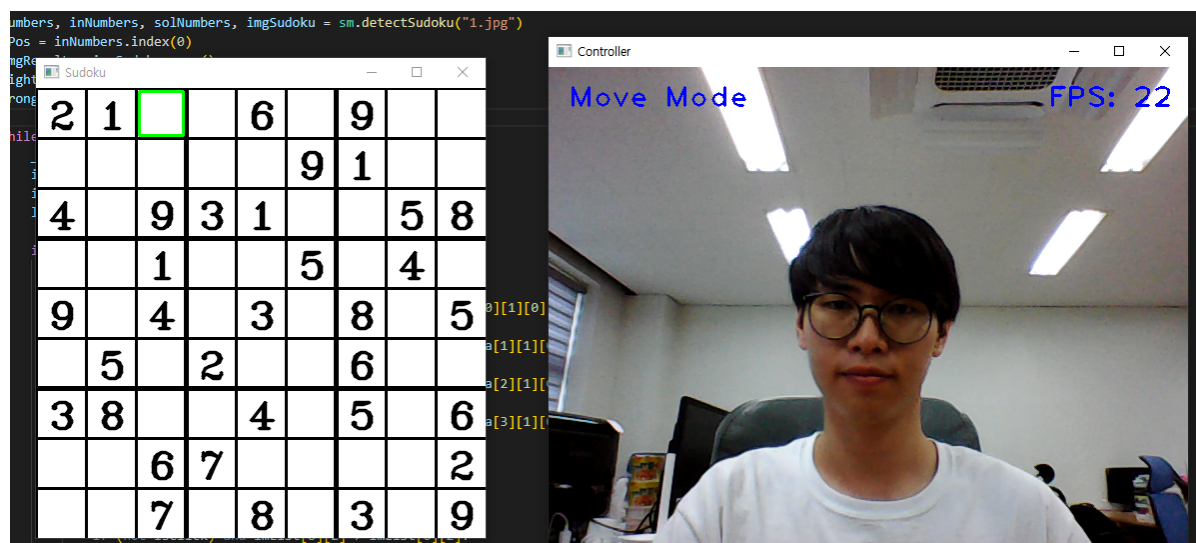
Date: 20-06-2022

Author: 김승환(21600102)

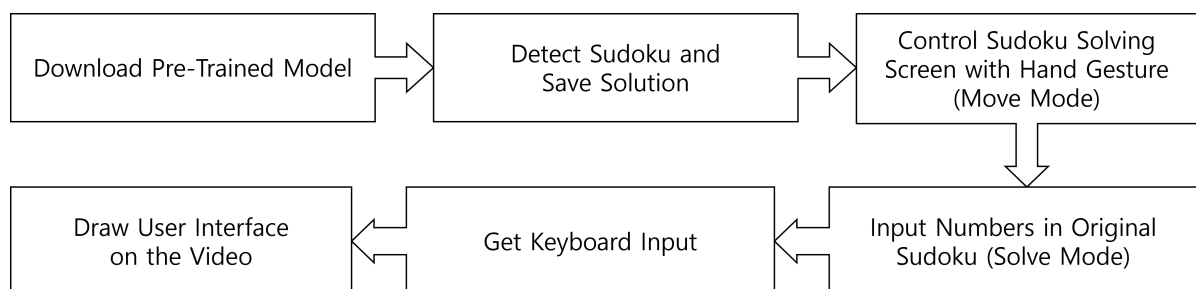
Github link: <https://github.com/SH-Kim97/DLIP/tree/main/FinalLAB>

Introduction

In this project, I will create a program that detect and solve sudoku puzzle. In this program, user can control sudoku screen with hand gesture and input numbers with fingers. When user inputs 'f' key, solve result and solution are displayed.



1. Flow Chart



Requirement

1. Hardware

- Intel Core i5-7300HQ
- GTX 1050 Ti

2. Software Installation

Follow the link below to see and install all the requirements

<https://ykkim.gitbook.io/dlip/installation-guide/installation-guide-for-deep-learning>

Then install Mediapipe, Tensorflow, Keras at py39 in anaconda prompt

```
pip install mediapipe
```

```
pip install tensorflow
```

(optional, when keras did not installed with tensorflow)

```
pip install keras
```

Below list is the overall requirements

- Python 3.9.12
- Open-cv 4.5.5.64
- Numpy 1.21.5
- Pytorch 1.9.1
- Cudnn 7.6.5
- Tensorflow 2.9.1
- Keras 2.9.0
- Mediapipe 0.8.10
- Protobuf 3.19.4

***Copy and paste 4 source codes in the Appendix below with exact same file name**

Dataset

Two test sudoku images (1.jpg, 2.jpg)

Download link: <https://github.com/murtazahassan/OpenCV-Sudoku-Solver/tree/main/Resources>

***Test images must be in the same folder with source codes**

Tutorial Procedure

1. Download Pre-Trained Model

1. For hand detection, use model in Mediapipe module
2. For sudoku detection, download and use pre-trained model in Keras (myModel.h5)

Download link: <https://github.com/murtazahassan/OpenCV-Sudoku-Solver/tree/main/Resources>

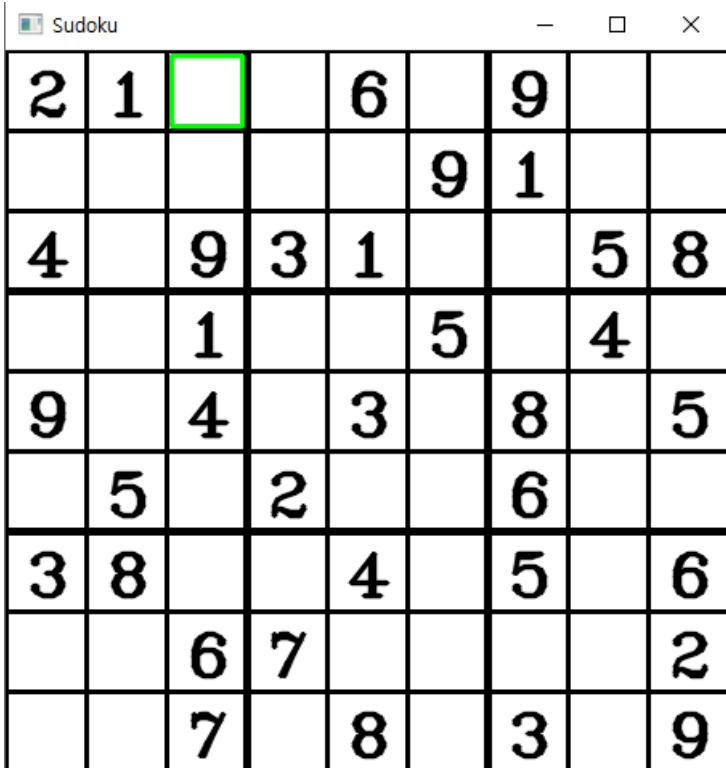
***Model must be in the same folder with source codes**

```
model = keras.models.load_model('myModel.h5')
```

2. Detect Sudoku and Save Solution

1. Preprocess the source image
2. Find contours
3. Find the biggest contour
4. Reshape image with the biggest coontour
5. Detect and save the digits in the sudoku
6. Solve and save solution

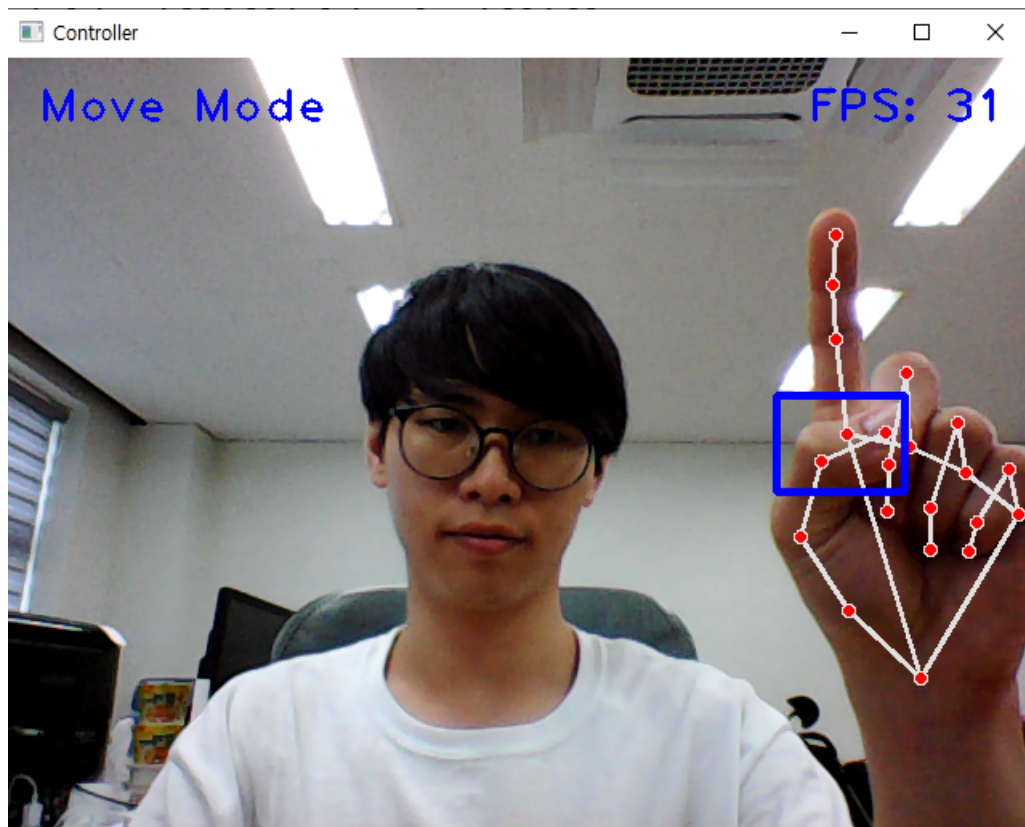
```
numbers, inNumbers, solNumbers, imgSudoku = sm.detectSudoku("1.jpg")
```



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

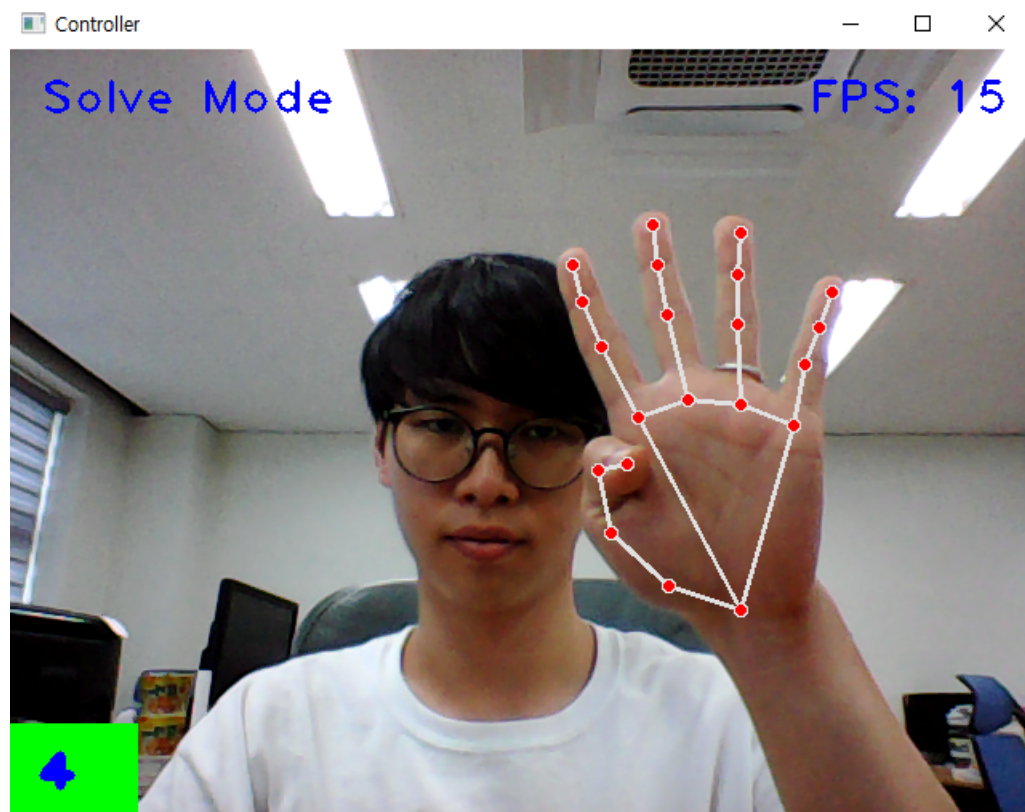
3. Control Sudoku Solving Screen with Hand Gesture (Move Mode)

In this mode, there are 4 hidden buttons(up, down, left, right) on the webcam screen. When the middle point of thumb is on button, the button is activated and user can click by folding index finger.



4. Input Numbers in Original Sudoku (Solve Mode)

In this mode, program detects number of fingers. When the number maintained for 20 frame, the number inputed in the selected position. When user inputs 0, the position is reset.



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

5. Get Keyboard Input (Mode Change)

When user inputs 'm' key, the mode is changed. One is move mode and the other is solve mode.

```
# Change mode
if inkey == 77 or inkey == 109:
    moveMode = not moveMode
    detector = htm.handDetector(maxHands=1+(not moveMode), detectionCon=0.75)
```





6. Get Keyboard Input (Finish Solving)

When user inputs 'f' key to finish solving, solve result and solution are displayed. 'Success!' is displayed when all the digits fit with solution. Otherwise, 'Fail!' is displayed.

```
# Finish and display result
elif inkey == 70 or inkey == 102:
    if inNumbers == solNumbers:
        imgResult = sm.inNumbers(imgResult, solNumbers, color = (0, 255, 0))
        putText(Sudoku, "Success!", (50, 240), FONT_HERSHEY_PLAIN, 5, (0, 255,
0), 5)
    else:
        for i in range(81):
            if solNumbers[i] == inNumbers[i]:
                rightNumbers[i] = solNumbers[i]
            else:
                wrongNumbers[i] = solNumbers[i]
        imgResult = sm.inNumbers(imgResult, rightNumbers, color = (0, 255, 0))
        imgResult = sm.inNumbers(imgResult, wrongNumbers, color = (0, 0, 255))
        putText(Sudoku, "Fail!", (150, 240), FONT_HERSHEY_PLAIN, 5, (0, 0, 255),
5)
    imshow('Result', imgResult)
    imshow('Sudoku', Sudoku)
    waitKey()
    break
```

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

Results and Analysis

1. For the number of fingers and hand gesture detection, it well works 100%
2. For the sudoku detection, it well works 5~6 times in 10 times with random sudoku image

Reference

<https://ykkim.gitbook.io/dlip/installation-guide/installation-guide-for-deep-learning>

<https://www.computervision.zone/courses/hand-tracking/>

<https://www.computervision.zone/courses/finger-counter/>

<https://github.com/murtazahassan/OpenCV-Sudoku-Solver>

Appendix

Entire Source Code

[DLIP_FinalLAB_21600102_김승환.py]

```
import math
import numpy as np
import cv2
from cv2 import *
from cv2.cv2 import *
import time
import mediapipe as mp
import HandTrackingModule as htm
import sudokuMain as sm
```

```
pTime = 0
```

```

wCam, hCam = 640, 480
cap = VideoCapture(0)
cap.set(CAP_PROP_FRAME_WIDTH, wCam)
cap.set(CAP_PROP_FRAME_HEIGHT, hCam)

moveMode = True
detector = htm.handDetector(maxHands=1, detectionCon=0.75)

buttonArea = [[(280, 90), (360, 150)], [(280, 330), (360, 390)], [(80, 210),
(160, 270)], [(480, 210), (560, 270)]] # 0: Up, 1: Down, 2: Left, 3: Right
moveDirection = 0 # 0: None, 1: Up, 2: Down, 3: Left, 4: Right
isClick = False
isMove = False

tipIds = [4, 8, 12, 16, 20, 25, 29, 33, 37, 41]

numStack = 0
pNum = 0

numbers, inNumbers, solNumbers, imgSudoku = sm.detectSudoku("1.jpg")
cPos = inNumbers.index(0)
imgResult = imgSudoku.copy()
rightNumbers = [0] * 81
wrongNumbers = [0] * 81

while True:
    _, img = cap.read()
    img = flip(img, 1)
    img = detector.findHands(img)
    lmList = detector.findPosition(img, draw=False)

    if moveMode: # Move mode
        if len(lmList) != 0:
            # Find position of thumb middle point
            if buttonArea[0][0][0] < lmList[3][1] < buttonArea[0][1][0] and
buttonArea[0][0][1] < lmList[3][2] < buttonArea[0][1][1]:
                moveDirection = 1
            elif buttonArea[1][0][0] < lmList[3][1] < buttonArea[1][1][0] and
buttonArea[1][0][1] < lmList[3][2] < buttonArea[1][1][1]:
                moveDirection = 2
            elif buttonArea[2][0][0] < lmList[3][1] < buttonArea[2][1][0] and
buttonArea[2][0][1] < lmList[3][2] < buttonArea[2][1][1]:
                moveDirection = 3
            elif buttonArea[3][0][0] < lmList[3][1] < buttonArea[3][1][0] and
buttonArea[3][0][1] < lmList[3][2] < buttonArea[3][1][1]:
                moveDirection = 4
            else:
                moveDirection = 0

        # Check click and move state
        if (not isClick) and lmList[8][2] > lmList[6][2]:
            isClick = True
            isMove = True
        elif (isClick) and lmList[8][2] > lmList[6][2]:
            isMove = False
        else:
            isClick = False

```



```

        isMove = False

        # Display direction and click state
        if moveDirection == 1:
            if isMove:
                img = rectangle(img, buttonArea[0][0], buttonArea[0][1], (0,
255, 0), 3)

                cPos = cPos - 9
                if cPos < 0:
                    cPos = cPos + 81
            else:
                img = rectangle(img, buttonArea[0][0], buttonArea[0][1],
(255, 0, 0), 3)
        elif moveDirection == 2:
            if isMove:
                img = rectangle(img, buttonArea[1][0], buttonArea[1][1], (0,
255, 0), 3)

                cPos = cPos + 9
                if cPos > 80:
                    cPos = cPos - 81
            else:
                img = rectangle(img, buttonArea[1][0], buttonArea[1][1],
(255, 0, 0), 3)
        elif moveDirection == 3:
            if isMove:
                img = rectangle(img, buttonArea[2][0], buttonArea[2][1], (0,
255, 0), 3)

                cPos = cPos - 1
                if cPos % 9 == 8:
                    cPos = cPos + 9
            else:
                img = rectangle(img, buttonArea[2][0], buttonArea[2][1],
(255, 0, 0), 3)
        elif moveDirection == 4:
            if isMove:
                img = rectangle(img, buttonArea[3][0], buttonArea[3][1], (0,
255, 0), 3)

                cPos = cPos + 1
                if cPos % 9 == 0:
                    cPos = cPos - 9
            else:
                img = rectangle(img, buttonArea[3][0], buttonArea[3][1],
(255, 0, 0), 3)

        putText(img, "Move Mode", (20, 40), FONT_HERSHEY_PLAIN, 2, (255, 0, 0),
2)

    else: # Solve mode
        # Detect fingers
        fingers = []
        if len(lmList) != 0:
            for id, cx, cy in lmList[:]:
                if id in tipIds:
                    if id in [4, 25]: # Thumb
                        if lmList[id - 1][1] < cx < lmList[id + 13][1] or
lmList[id + 13][1] < cx < lmList[id - 1][1]:
                            fingers.append(0)
                    else:

```

```

        fingers.append(1)

    else: # 4 Fingers
        if cy < lmList[id - 2][2]:
            fingers.append(1)
        else:
            fingers.append(0)

    totalFingers = fingers.count(1) # Count fingers

    # Stack same number of fingers
    if pNum == totalFingers:
        numStack += 1
    else:
        numStack = 0
    pNum = totalFingers

    # Input the number when number of fingers maintained for 20 frame
    if numStack == 20 and totalFingers != 10 and inNumbers[cPos] != -1:
        inNumbers[cPos] = totalFingers
        try:
            cPos = inNumbers.index(0)
        except:
            pass

    # Print number of fingers
    rectangle(img, (0, 420), (80, 480), (0, 255, 0), FILLED)
    putText(img, str(totalFingers), (20, 460), FONT_HERSHEY_PLAIN, 2,
(255, 0, 0), 5)

    putText(img, "Solve Mode", (20, 40), FONT_HERSHEY_PLAIN, 2, (255, 0, 0),
2)

    # Display sudoku image
    sudoku = imgSudoku.copy()
    sudoku = sm.inNumbers(sudoku, inNumbers)
    sudoku = sm.selectedGrid(sudoku, cPos)

    cTime = time.time()
    fps = 1 / (cTime - pTime)
    pTime = cTime

    putText(img, f'FPS: {int(fps)}', (500, 40), FONT_HERSHEY_PLAIN, 2, (255, 0,
0), 2)

    imshow('Sudoku', sudoku)
    imshow("Controller", img)

    inkey = waitKey(1)
    # Change mode
    if inkey == 77 or inkey == 109:
        moveMode = not moveMode
        detector = htm.handDetector(maxHands=1+(not moveMode),
detectionCon=0.75)
    # Finish and display result
    elif inkey == 70 or inkey == 102:
        if inNumbers == solNumbers:
            imgResult = sm.inNumbers(imgResult, solNumbers, color = (0, 255, 0))

```

```

        putText(Sudoku, "Success!", (50, 240), FONT_HERSHEY_PLAIN, 5, (0,
255, 0), 5)
    else:
        for i in range(81):
            if solNumbers[i] == inNumbers[i]:
                rightNumbers[i] = solNumbers[i]
            else:
                wrongNumbers[i] = solNumbers[i]
        imgResult = sm.inNumbers(imgResult, rightNumbers, color = (0, 255,
0))
        imgResult = sm.inNumbers(imgResult, wrongNumbers, color = (0, 0,
255))
        putText(Sudoku, "Fail!", (150, 240), FONT_HERSHEY_PLAIN, 5, (0, 0,
255), 5)
        imshow('Result', imgResult)
        imshow('Sudoku', Sudoku)
        waitKey()
        break
# Break
elif inkey == 27:
    break

destroyAllWindows()

```

[HandTrackingModule.py]

```

import math
import numpy as np
import cv2
from cv2 import *
from cv2.cv2 import *
import mediapipe as mp

class handDetector():
    def __init__(self, mode=False, maxHands=2, complexity=1, detectionCon=0.5,
trackCon=0.5):
        self.mode = mode
        self.maxHands = maxHands
        self.complexity = complexity
        self.detectionCon = detectionCon
        self.trackCon = trackCon

        self.mpHands = mp.solutions.hands
        self.hands = self.mpHands.Hands(self.mode, self.maxHands,
self.complexity, self.detectionCon, self.trackCon)
        self.mpDraw = mp.solutions.drawing_utils

    def findHands(self, img, draw=True):
        imgRGB = cvtColor(img, COLOR_BGR2RGB)
        self.results = self.hands.process(imgRGB)

        if self.results.multi_hand_landmarks:
            for handLms in self.results.multi_hand_landmarks:
                if draw:

```

```

        self.mpDraw.draw_landmarks(img, handLms,
                                    self.mpHands.HAND_CONNECTIONS)

    return img

def findPosition(self, img, draw=True):

    lmList = []
    if self.results.multi_hand_landmarks:
        for handNo, handLms in enumerate(self.results.multi_hand_landmarks):
            for id, lm in enumerate(handLms.landmark):
                h, w, _ = img.shape
                cx, cy = int(lm.x * w), int(lm.y * h)
                lmList.append([id + handNo*21, cx, cy])
                if draw:
                    circle(img, (cx, cy), 15, (255, 0, 255), FILLED)
    return lmList

```

[sudokuMain.py]

```

import os
import cv2
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
from sudokuSolver import *

def detectSudoku(pathImage):
    heightImg = 450
    widthImg = 450
    model = initializePredectionModel() # Load the CNN model

    img = cv2.imread(pathImage)
    img = cv2.resize(img, (widthImg, heightImg))
    imgThreshold = preProcess(img)

    contours, hierarchy = cv2.findContours(imgThreshold, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    biggest, maxArea = biggestContour(contours) # Find the biggest contour

    if biggest.size != 0:
        biggest = reorder(biggest)
        pts1 = np.float32(biggest)
        pts2 = np.float32([[0, 0],[widthImg, 0], [0, heightImg],[widthImg,
heightImg]])
        matrix = cv2.getPerspectiveTransform(pts1, pts2)
        imgwarpColored = cv2.warpPerspective(img, matrix, (widthImg, heightImg))
        imgwarpColored = cv2.cvtColor(imgwarpColored,cv2.COLOR_BGR2GRAY)
        imgDetectedDigits = np.ones((heightImg, widthImg, 3), np.uint8) * 255

        # Split the image and find each digit available
        boxes = splitBoxes(imgwarpColored)
        numbers = getPredection(boxes, model)
        imgDetectedDigits = displayNumbers(imgDetectedDigits, numbers)
        numbers = np.asarray(numbers)
        posArray = np.where(numbers > 0, 0, 1)

```

```

        inNumbers = np.where(numbers > 0, -1, 0)

        # Find solution of the board
        board = np.array_split(numbers, 9)
        try:
            solve(board)
        except:
            pass

        # Save the solution
        flatList = []
        for sublist in board:
            for item in sublist:
                flatList.append(item)
        solvedNumbers = flatList*posArray
        solvedNumbers = np.where(solvedNumbers == 0, -1, solvedNumbers)

        # Draw grid and Return
        imgDetectedDigits = drawGrid(imgDetectedDigits)
        return numbers.tolist(), inNumbers.tolist(), solvedNumbers.tolist(),
imgDetectedDigits

    else:
        print("No Sudoku Found")

# Draw selected grid
def selectedGrid(img, pos):
    secW = int(img.shape[1] / 9)
    secH = int(img.shape[0] / 9)

    xpos, ypos = pos % 9, pos // 9
    pt1 = (xpos*secW+3, ypos*secH+3)
    pt2 = ((xpos+1)*secW-3, (ypos+1)*secH-3)

    cv2.rectangle(img, pt1, pt2, (0, 255, 0), 2)
    return img

# Display input numbers
def inNumbers(img, numbers, color = (255, 0, 0)):
    secW = int(img.shape[1]/9)
    secH = int(img.shape[0]/9)
    for x in range (0,9):
        for y in range (0,9):
            if numbers[(y*9)+x] not in [-1, 0] :
                cv2.putText(img, str(numbers[(y*9)+x]),
                            (x*secW+int(secW/2)-12, int((y+0.8)*secH)),
cv2.FONT_HERSHEY_COMPLEX_SMALL,
                            2, color, 2, cv2.LINE_AA)

    return img

```

[sudokuSolver.py]

```

import cv2
import numpy as np
from tensorflow import keras

```

```

def solve(bo):
    find = find_empty(bo)
    if not find:
        return True
    else:
        row, col = find
    for i in range(1,10):
        if valid(bo, i, (row, col)):
            bo[row][col] = i
            if solve(bo):
                return True
            bo[row][col] = 0
    return False

def valid(bo, num, pos):
    # Check row
    for i in range(len(bo[0])):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False
    # Check column
    for i in range(len(bo)):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False
    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3
    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if bo[i][j] == num and (i,j) != pos:
                return False
    return True

def print_board(bo):
    for i in range(len(bo)):
        if i % 3 == 0 and i != 0:
            print("- - - - -")
        for j in range(len(bo[0])):
            if j % 3 == 0 and j != 0:
                print(" | ", end="")
            if j == 8:
                print(bo[i][j])
            else:
                print(str(bo[i][j]) + " ", end="")

def find_empty(bo):
    for i in range(len(bo)):
        for j in range(len(bo[0])):
            if bo[i][j] == 0:
                return (i, j)
    return None

# Read the model weights
def initializePredictionModel():
    model = keras.models.load_model('myModel.h5')
    return model

```



```

# Preprocessing image
def preProcess(img):
    imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    imgBlur = cv2.GaussianBlur(imgGray, (5, 5), 1)
    imgThreshold = cv2.adaptiveThreshold(imgBlur, 255, 1, 1, 11, 2)
    # _, imgThreshold = cv2.threshold(imgBlur, 120, 255, 1)
    return imgThreshold

# Reorder points for Warp Perspective
def reorder(myPoints):
    myPoints = myPoints.reshape((4, 2))
    myPointsNew = np.zeros((4, 1, 2), dtype=np.int32)
    add = myPoints.sum(1)
    myPointsNew[0] = myPoints[np.argmin(add)]
    myPointsNew[3] = myPoints[np.argmax(add)]
    diff = np.diff(myPoints, axis=1)
    myPointsNew[1] = myPoints[np.argmin(diff)]
    myPointsNew[2] = myPoints[np.argmax(diff)]
    return myPointsNew

# Finding the biggest contour
def biggestContour(contours):
    biggest = np.array([])
    max_area = 0
    for i in contours:
        area = cv2.contourArea(i)
        if area > 50:
            peri = cv2.arcLength(i, True)
            approx = cv2.approxPolyDP(i, 0.02 * peri, True)
            if area > max_area and len(approx) == 4:
                biggest = approx
                max_area = area
    return biggest, max_area

# Split the image into 81 different images
def splitBoxes(img):
    rows = np.vsplit(img, 9)
    boxes=[]
    for r in rows:
        cols= np.hsplit(r, 9)
        for box in cols:
            boxes.append(box)
    return boxes

# Get predictions on all images
def getPredection(boxes, model):
    result = []
    for image in boxes:
        # Prepare image
        img = np.asarray(image)
        img = img[4:img.shape[0] - 4, 4:img.shape[1] -4]
        img = cv2.resize(img, (28, 28))
        img = img / 255
        img = img.reshape(1, 28, 28, 1)
        # Get predection
        predictions = model.predict(img)
        classIndex = np.argmax(predictions, axis=1)
        probabilityValue = np.amax(predictions)

```

```

        # Save to result
        if probabilityvalue > 0.8:
            result.append(classIndex[0])
        else:
            result.append(0)
    return result

# Display the original sudoku
def displayNumbers(img, numbers, color = (0, 0, 0)):
    secw = int(img.shape[1]/9)
    sech = int(img.shape[0]/9)
    for x in range (0,9):
        for y in range (0,9):
            if numbers[(y*9)+x] != 0 :
                cv2.putText(img, str(numbers[(y*9)+x]),
                            (x*secw+int(secw/2)-12, int((y+0.8)*sech)),
                            cv2.FONT_HERSHEY_COMPLEX_SMALL,
                            2, color, 2, cv2.LINE_AA)

    return img

# Draw grid
def drawGrid(img):
    secw = int(img.shape[1] / 9)
    sech = int(img.shape[0] / 9)
    for i in range (0, 9):
        pt1 = (0, sech * i)
        pt2 = (img.shape[1], sech * i)
        pt3 = (secw * i, 0)
        pt4 = (secw * i, img.shape[0])
        if i % 3 == 0 and i > 0:
            cv2.line(img, pt1, pt2, (0, 0, 0),3)
            cv2.line(img, pt3, pt4, (0, 0, 0),3)
        else:
            cv2.line(img, pt1, pt2, (0, 0, 0),2)
            cv2.line(img, pt3, pt4, (0, 0, 0),2)
    return img

```