# How to Write a Auto Sudoku Solver Using Python

March 13, 2020

Author: Sumeng Wang

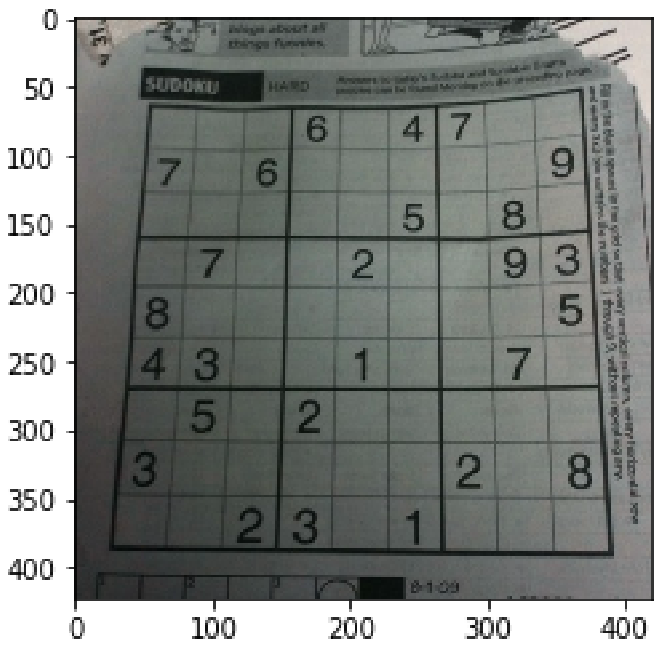
A Sudoku puzzle is completed when each row, column and 3x3 square contain the numbers 1-9 exactly with no repeats. Our goal is to take a picture of the Sudoku puzzle, then process the image to detect the puzzle, recognize the digits in each cell, and finally solve the puzzle

## 1 Scan the Sudoku

Let's first pick a Sudoku Puzzle as following.

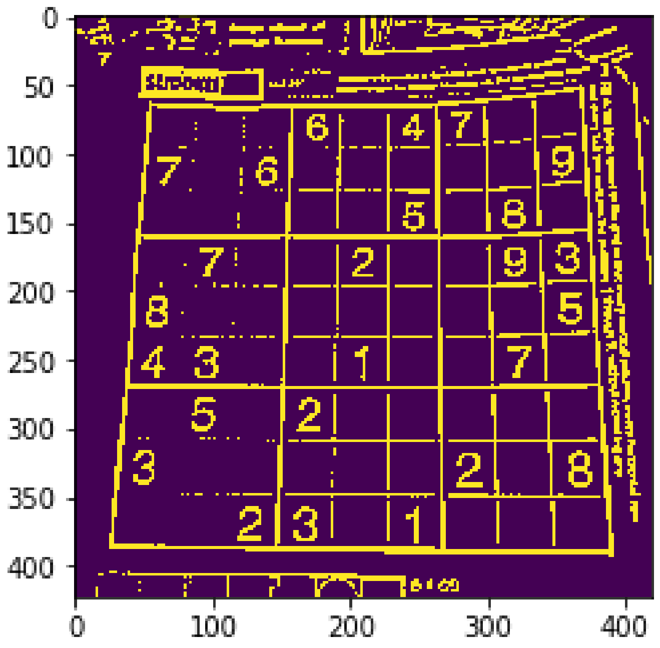
[30]:
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('puzzles/puzzle4.jpg')
plt.imshow(img)
plt.show()
```



First we want to convert the color into grayscale, remove some noise and blue it. Then do bitwise inversion to it. This gives us a better picture to process

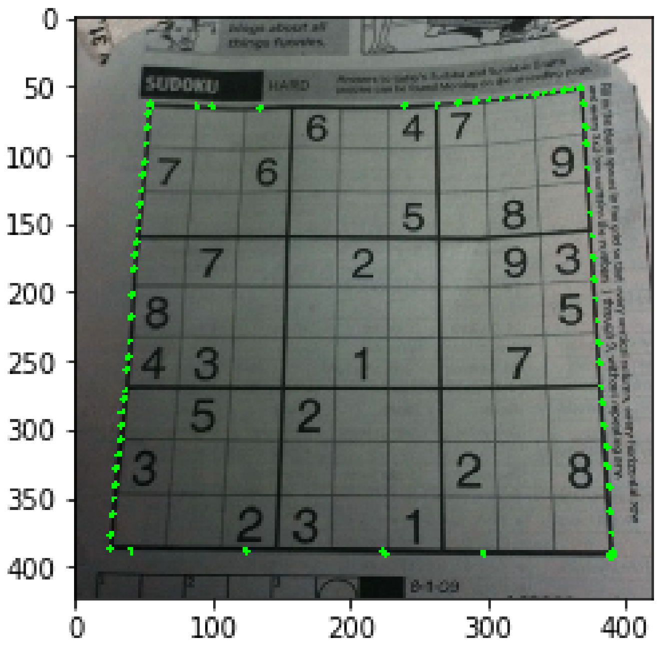
[31]:
```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = cv2.fastNlMeansDenoising(gray, h = 10)
outerBox = np.zeros(img.shape, np.uint8)
gray = cv2.GaussianBlur(gray, (5,5), 0)
outerBox = cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.
 ↪THRESH_BINARY, 5, 2)
outerBox = cv2.bitwise_not(outerBox)
plt.imshow(outerBox)
plt.show()
```



After that, we use findContours function to find all closed areas. Since the puzzle itself is the biggest closed area in the figure, we take the max of all contours to find the border of the sudoku. This is shown below:

[38]:
```
%%capture
ret,thresh = cv2.threshold(outerBox,127,255,0)
contours, hierarchy = cv2.findContours(thresh,cv2.RETR_EXTERNAL,cv2.
 ↪CHAIN_APPROX_SIMPLE)
maxContour = max(contours, key=cv2.contourArea)
temp = img.copy()
cv2.drawContours(temp, maxContour, -1, (0,255,0), 3)
```

[39]:
```
plt.imshow(temp)
plt.show()
```



Since the picture we take might be tilted, like this example we are using, even though we are able to extract the most left, right, top, bottom point, we need to figure out a way to reliably determine the four corners of sudoku puzzle. The way I did it is to first determine the midpoint from the most left, right, top, bottom point. After we found the midpoint, we can divide the figure into four quadrants. And the corner in each quadrant will be the point that is farthest from the midpoint. This way eventually gives us a reliable way to find the corners of the puzzle. After that, we transform the figure to get rid of the extra background and only leaves the puzzle.

[50]:
```
extLeft = tuple(maxContour[maxContour[:, :, 0].argmin()][0])
extRight = tuple(maxContour[maxContour[:, :, 0].argmax()][0])
extTop = tuple(maxContour[maxContour[:, :, 1].argmin()][0])
extBot = tuple(maxContour[maxContour[:, :, 1].argmax()][0])
midpoint = (int((extLeft[0] + extRight[0])/2), int((extTop[1]+extBot[1])/2))

first_quadrant = [p[0] for p in maxContour if (p[0][0] > midpoint[0]) and␣
 ↪(p[0][1] < midpoint[1])]
second_quadrant = [p[0] for p in maxContour if (p[0][0] < midpoint[0]) and␣
 ↪(p[0][1] < midpoint[1])]
third_quadrant = [p[0] for p in maxContour if (p[0][0] < midpoint[0]) and␣
 ↪(p[0][1] > midpoint[1])]
fourth_quadrant = [p[0] for p in maxContour if (p[0][0] > midpoint[0]) and␣
 ↪(p[0][1] > midpoint[1])]
```

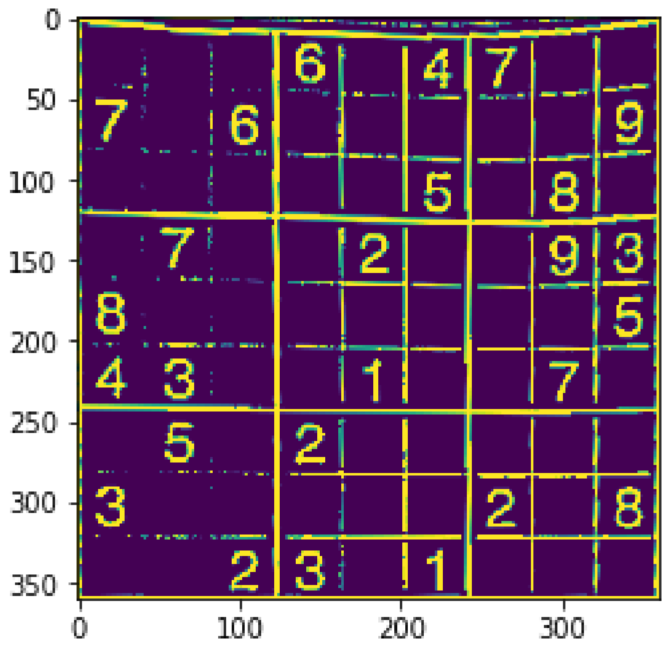
```
def distance(p):
        return (p[0] - midpoint[0]) ** 2 + (p[1] - midpoint[1]) ** 2

# determine corner positions by quadrants
cornerRT = tuple(max(first_quadrant, key = distance))
cornerLT = tuple(max(second_quadrant, key = distance))
cornerLB = tuple(max(third_quadrant, key = distance))
cornerRB = tuple(max(fourth_quadrant, key = distance))

# crop out puzzle and transform
pts1 = np.float32([cornerLT,cornerRT,cornerLB,cornerRB])
pts2 = np.float32([[0,0],[360,0],[0,360],[360,360]])

M = cv2.getPerspectiveTransform(pts1,pts2)
gray_dst = cv2.warpPerspective(outerBox,M,(360,360))

plt.imshow(gray_dst)
plt.show()
```



As we can seee in the above figure, there are many vertical and horizontal lines. We want to get rid of those lines because those lines add noise to our digit recognition part later. To do this we can use the contour method again, since each vertical or horizontal line is a contour. And then we use the background color to draw those contours out to get a cleaner figure.

[51]:
```
horizontal = np.copy(gray_dst)
vertical = np.copy(gray_dst)
```

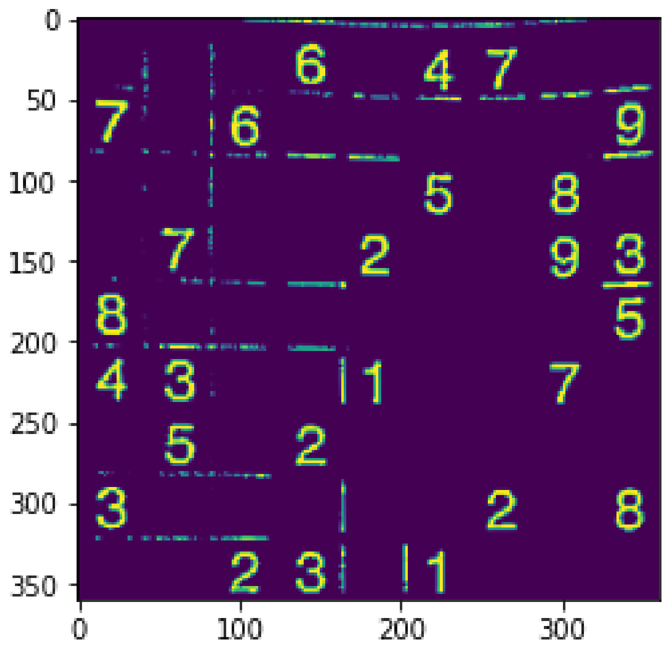
```
# Specify size on horizontal axis
cols = horizontal.shape[1]
horizontal_size = int(cols / 12)
# Create structure element for extracting horizontal lines through morphology␣
↪operations
horizontalStructure = cv2.getStructuringElement(cv2.MORPH_RECT,␣
↪(horizontal_size, 1))
# Apply morphology operations
horizontal = cv2.erode(horizontal, horizontalStructure)
horizontal = cv2.dilate(horizontal, horizontalStructure)

# [horiz]
# [vert]
# Specify size on vertical axis
rows = vertical.shape[0]
verticalsize = int(rows / 12)

# Create structure element for extracting vertical lines through morphology␣
↪operations
verticalStructure = cv2.getStructuringElement(cv2.MORPH_RECT, (1, verticalsize))
# Apply morphology operations
vertical = cv2.erode(vertical, verticalStructure)
vertical = cv2.dilate(vertical, verticalStructure)
# Show extracted vertical lines

ret_v,thresh_v = cv2.threshold(vertical,127,255,0)
contours_v, hierarchy_v = cv2.findContours(thresh_v,cv2.RETR_TREE,cv2.
↪CHAIN_APPROX_SIMPLE)
ret_h,thresh_h = cv2.threshold(horizontal,127,255,0)
contours_h, hierarchy_h = cv2.findContours(thresh_h,cv2.RETR_TREE,cv2.
↪CHAIN_APPROX_SIMPLE)
cv2.drawContours(gray_dst, contours_v, -1, (0,0,0), 3)
cv2.drawContours(gray_dst, contours_h, -1, (0,0,0), 3)

# [vert]
# [smooth]
smooth = cv2.blur(gray_dst, (2, 2))
smooth = cv2.fastNlMeansDenoising(smooth, h = 10)
plt.imshow(smooth)
plt.show()
```



Up to this point, we are done with our initial image processing. Next we can move on to our digit recognition model training.

## 2 Train Digit Recognition Model

We use keras library to train our digit recognition model. The reference I followed is a tutorial on dogs and cats classification (https://towardsdatascience.com/image-detection-from-scratch-in-keras-f314872006c9). To prepare the labeled data, I processed about ten sudoku puzzles in different fonts, crops each cell out and manually save it into corresponding folders named from one to nine. As a result, I have about 1000 labeled pictures and I used 80% for training and 20% for validation. The final model comes out with a 99% percent accuracy.

[ ]:
```
import random
import gc
from sklearn.model_selection import train_test_split
from keras import layers
from keras import models
from keras import optimizers
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing.image import img_to_array, load_img

path = "D:/UCLA/project/soduku/trainimage/"
keys = ['blank/', 'one/','two/','three/','four/','five/','six/','seven/','eight/
 ↪','nine/']
blanks = [(path + keys[0] + "{}").format(i) for i in os.listdir(path + keys[0])]
ones = [(path + keys[1] + "{}").format(i) for i in os.listdir(path + keys[1])]
```

```
twos = [(path + keys[2] + "{}").format(i) for i in os.listdir(path + keys[2])]
threes = [(path + keys[3] + "{}").format(i) for i in os.listdir(path + keys[3])]
fours = [(path + keys[4] + "{}").format(i) for i in os.listdir(path + keys[4])]
fives = [(path + keys[5] + "{}").format(i) for i in os.listdir(path + keys[5])]
sixs = [(path + keys[6] + "{}").format(i) for i in os.listdir(path + keys[6])]
sevens = [(path + keys[7] + "{}").format(i) for i in os.listdir(path + keys[7])]
eights = [(path + keys[8] + "{}").format(i) for i in os.listdir(path + keys[8])]
nines = [(path + keys[9] + "{}").format(i) for i in os.listdir(path + keys[9])]

train_imgs = blanks + ones + twos + threes + fours + fives + sixs + sevens +␣
 ↪eights + nines
random.shuffle(train_imgs)
# clear garbage
del blanks
del ones
del twos
del threes
del fours
del fives
del sixs
del sevens
del eights
del nines

gc.collect()

def read_and_process_image(list_of_images):
    X = []
    y = []

    for image in list_of_images:
        img = cv2.resize(cv2.imread(image, cv2.COLOR_BGR2GRAY), (150,150),␣
 ↪interpolation = cv2.INTER_CUBIC)
        img = np.expand_dims(img, axis=-1)
        X.append(img)
        if 'blank' in image:
            y.append(0)
        elif 'one' in image:
            y.append(1)
        elif 'two' in image:
            y.append(2)
        elif 'three' in image:
            y.append(3)
        elif 'four' in image:
            y.append(4)
        elif 'five' in image:
            y.append(5)
        elif 'six' in image:
            y.append(6)
        elif 'seven' in image:
            y.append(7)
        elif 'eight' in image:
            y.append(8)
        elif 'nine' in image:
            y.append(9)

    return X, y

X, y = read_and_process_image(train_imgs)
print(X[0].shape)
X = np.array(X)
y = np.array(y)
# split the data into train and test set

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size = 0.2)

del X
del y
gc.collect()

ntrain = len(X_train)
nval = len(X_val)

batch_size = 32

model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation = 'relu', input_shape = (150,150,␣
 ↪1)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64,(3,3), activation = 'relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128,(3,3), activation = 'relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128,(3,3), activation = 'relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation = 'relu'))
model.add(layers.Dense(10, activation='softmax'))

# use the RMSprop optimizer with a learning rate of 0.0001
# use sparse_categorical_crossentropy loss because 10 classes
model.compile(loss ='sparse_categorical_crossentropy', optimizer = optimizers.
 ↪RMSprop(lr = 1e-4), metrics = ['acc'])
```

```python
# create the augmentation configuration
train_datagen = ImageDataGenerator(rescale=1./255,
                                  width_shift_range = 0.1,
                                  height_shift_range = 0.1,
                                  shear_range = 0.1,
                                  zoom_range = 0.1)
val_datagen = ImageDataGenerator(rescale=1./255)

# create the image generators
train_generator = train_datagen.flow(X_train, y_train, batch_size=batch_size)
val_generator = val_datagen.flow(X_val,y_val,batch_size=batch_size)

# training
history = model.fit_generator(train_generator,
                                steps_per_epoch = ntrain // batch_size,
                                epochs = 128,
                                validation_data = val_generator,
                                validation_steps = nval // batch_size)

model.save('models/digit_rec_model.h5')
```

## 3 The Solving Algorithm

The solving algorithm implement an optimized recursive method. At each state of completion of the puzzle, the function checks the possible choices of each empty cell. Then it fills in the cell with only one possible choice and update the possible choices of other empty cells. It repeats until it does not have any cell with only one possible choice. Then it finds the one with least possibilities and recurse on all possible choices.

[55]:
```python
from copy import copy, deepcopy
def column(matrix, i):
    return [row[i] for row in matrix]

def block(matrix, i, j):
    return [row[i:(i+3)] for row in matrix[j:(j+3)]]

def possibilities(i,j,board):
    # to generate possibilities of each entry of the board
    begin = [1,2,3,4,5,6,7,8,9]
    for col in range(0,9):
        if board[i][col] in begin:
            begin.remove(board[i][col])
    for row in range(0,9):
        if board[row][j] in begin:
            begin.remove(board[row][j])
    row_index = int(i/3)*3
    col_index = int(j/3)*3
    for row in range(row_index, row_index+3):
        for col in range(col_index, col_index+3):
            if board[row][col] in begin:
                begin.remove(board[row][col])
    return begin

def generate_poss(board):
    out = [[[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []],
            [[], [], [], [], [], [], [], [], []]
           ]
    for i in range(0,9):
        for j in range(0,9):
            if board[i][j] == 0:
                out[i][j] = possibilities(i,j,board)
    return out

def find_hidden_single(poss, board):
    sol = []
    for i in range(0,9):
        # check row
        rows = sum(poss[i],[])
        cols = sum(column(poss, i),[])
        x = int(i/3)
        y = i//3
        blocks = sum(block(poss,x,y),[])
        for k in range(1,10):
            if rows.count(k) == 1:
                for j in range(0,9):
                    if k in poss[i][j] and [i,j,k] not in sol:
                        # this is the hidden single
                        sol.append([i,j,k])
            if cols.count(k) == 1:
                for j in range(0,9):
                    if k in poss[j][i] and [j,i,k] not in sol:
                        # this is the hidden single
                        sol.append([j,i,k])
            if blocks.count(k) == 1:
                for ix in range(x*3, x*3+3):
                    for iy in range(y*3, y*3+3):
                        if k in poss[ix][iy] and [ix,iy,k] not in sol:
                        # this is the hidden single
                            sol.append([ix,iy,k])
    return sol

def update_board(board, sol):
    for key in sol:
        board[key[0]][key[1]] = key[2]
    return board

def get_min_poss(poss):
    min_poss = [9,9,9]
    for i in range(0,9):
        for j in range(0,9):
            if len(poss[i][j]) < min_poss[2] and len(poss[i][j]) != 0:
                min_poss = [i,j,len(poss[i][j])]
    return min_poss

def check_complete(board):
    success = True
    for row in board:
        if 0 in row:
            success = False
    return success

def check_wrong_path(board):
    wrong_path = False
    poss = generate_poss(board)
    for i in range(0,9):
        for j in range(0,9):
            if board[i][j] == 0 and poss[i][j] == []:
                wrong_path = True
    return wrong_path

def solve(board):
    hidden_single = True
    while hidden_single:
        poss = generate_poss(board)
        sol = find_hidden_single(poss, board)
        if sol == []:
            #can't find hidden single any more
            hidden_single = False
        else:
            board = update_board(board, sol)
    if check_complete(board):
        #result = board
        return board
    elif check_wrong_path(board):
        #do nothing
        #pass
        return None
    else:
        poss = generate_poss(board)
        guess = get_min_poss(poss)
        results = [0] * guess[2]
        for i in range(0, guess[2]):
            #print(str(i) + ' ' + str(guess[2]))
            temp = deepcopy(board)
            temp[guess[0]][guess[1]] = poss[guess[0]][guess[1]][i]
            results[i] = solve(temp)
        return merge_results(results)

def merge_results(mylist):
    n = len(mylist)
    for i in range(0, n):
        if mylist[i] != None:
            return mylist[i]
    return None
```

## 4 Solve the Puzzle!

[76]:
```
from keras.models import load_model
model = load_model('models/digit_rec_model2.h5')

X = []
for i in range(0,9):
    for j in range(0,9):
        crop_img = smooth[i * 40: (i+1)*40 - 1, j * 40: (j+1)*40 - 1]
        crop_img = cv2.resize(crop_img, (150,150), interpolation = cv2.
 ↪INTER_CUBIC)
        crop_img =  np.expand_dims(crop_img, axis=-1)
        crop_img =  np.expand_dims(crop_img, axis=0)
        X.append(crop_img)

x = np.array(X)

board = []
for i in range(0,9):
    y = [0] * 9
    for j in range(0,9):
        y[j] = model.predict_classes(x[i*9+j])[0]
    board.append(y)
```

```python
solution = solve(board)
for i in range(9):
    if i % 3 == 0 and i != 0:
        print('------|-------|------')
    for j in range(9):
        if (j+1) % 3 == 0 and j != 8:
            print(solution[i][j], end = ' | ')
        else:
            print(solution[i][j], end = ' ')
    print()
```

```
5 8 3 | 6 9 4 | 7 2 1
7 1 6 | 8 3 2 | 5 4 9
2 9 4 | 1 7 5 | 3 8 6
------|-------|------
6 7 1 | 5 2 8 | 4 9 3
8 2 9 | 7 4 3 | 1 6 5
4 3 5 | 9 1 6 | 8 7 2
------|-------|------
1 5 8 | 2 6 7 | 9 3 4
3 6 7 | 4 5 9 | 2 1 8
9 4 2 | 3 8 1 | 6 5 7
```

## 5 Future Goal

Currently I am self learning Django and AWS. In the future I will convert this project into a real web app that everyone can use.