

Sudoku Solving using Digit classification, image processing and mathematics

Part 1: Digit Classification Model

Importing Libraries

In [1]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import os, random
import cv2
from glob import glob
import sklearn
from sklearn.model_selection import train_test_split
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing.image import ImageDataGenerator, load_img
from keras.utils.np_utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dropout, Dense, Flatten, BatchNormalizat
ion, Conv2D, MaxPooling2D
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras import backend as K
from tensorflow.keras.preprocessing import image
from sklearn.metrics import accuracy_score, classification_report
from pathlib import Path
from PIL import Image
```

Loading the data

In [2]:

```
data = os.listdir(r"../input/digits/Digits" )
data_X = []
data_y = []
data_classes = len(data)
for i in range (0,data_classes):
    data_list = os.listdir(r"../input/digits/Digits" + "/" + str(i))
    for j in data_list:
        pic = cv2.imread(r"../input/digits/Digits" + "/" + str(i) + "/" + j)
        pic = cv2.resize(pic, (32,32))
        data_X.append(pic)
        data_y.append(i)
```

```
if len(data_X) == len(data_y) :  
    print("Total Dataponits = ",len(data_X))
```

```
# Labels and images  
data_X = np.array(data_X)  
data_y = np.array(data_y)
```

Total Dataponits = 10160

Data Spliting

In [3]:

```
train_X, test_X, train_y, test_y = train_test_split(data_X,data_y,test_size=0.05)  
train_X, valid_X, train_y, valid_y = train_test_split(train_X,train_y,test_size=0.2)  
print("Training Set Shape = ",train_X.shape)  
print("Validation Set Shape = ",valid_X.shape)  
print("Test Set Shape = ",test_X.shape)
```

Training Set Shape = (7721, 32, 32, 3)
Validation Set Shape = (1931, 32, 32, 3)
Test Set Shape = (508, 32, 32, 3)

Preprocessing the images

In [4]:

```
def Prep(img):  
    img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY) #making image grayscale  
    img = cv2.equalizeHist(img) #Histogram equalization to enhance contrast  
    img = img/255 #normalizing  
    return img  
  
train_X = np.array(list(map(Prep, train_X)))  
test_X = np.array(list(map(Prep, test_X)))  
valid_X= np.array(list(map(Prep, valid_X)))  
  
#Reshaping the images  
train_X = train_X.reshape(train_X.shape[0], train_X.shape[1], train_X.shape[2],1)  
test_X = test_X.reshape(test_X.shape[0], test_X.shape[1], test_X.shape[2],1)  
valid_X = valid_X.reshape(valid_X.shape[0], valid_X.shape[1], valid_X.shape[2],1)  
  
#Augmentation  
datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, zoom_range=0.  
2, shear_range=0.1, rotation_range=10)  
datagen.fit(train_X)
```

In [5]:

```
train_y = to_categorical(train_y, data_classes)  
test_y = to_categorical(test_y, data_classes)  
valid_y = to_categorical(valid_y, data_classes)
```

Building the model

In [6]:

```
model = Sequential()  
  
model.add(Conv2D(60, (5,5),input_shape=(32, 32, 1),padding = 'Same',activation='relu'))  
model.add(Conv2D(60, (5,5),padding="same",activation='relu'))  
model.add(MaxPooling2D(pool_size=(2,2)))  
#model.add(Dropout(0.25))
```

```
model.add(Conv2D(30, (3,3),padding="same", activation='relu'))
model.add(Conv2D(30, (3,3), padding="same", activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(500,activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 60)	1560
conv2d_1 (Conv2D)	(None, 32, 32, 60)	90060
max_pooling2d (MaxPooling2D)	(None, 16, 16, 60)	0
conv2d_2 (Conv2D)	(None, 16, 16, 30)	16230
conv2d_3 (Conv2D)	(None, 16, 16, 30)	8130
max_pooling2d_1 (MaxPooling2	(None, 8, 8, 30)	0
dropout (Dropout)	(None, 8, 8, 30)	0
flatten (Flatten)	(None, 1920)	0
dense (Dense)	(None, 500)	960500
dropout_1 (Dropout)	(None, 500)	0
dense_1 (Dense)	(None, 10)	5010
Total params: 1,081,490		
Trainable params: 1,081,490		
Non-trainable params: 0		

Model Compilation and Model training

In [7]:

```
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon = 1e-08, decay=0.0)
model.compile(optimizer=optimizer,loss='categorical_crossentropy',metrics=['accuracy'])

#Fit the model

history = model.fit(datagen.flow(train_X, train_y, batch_size=32),
                    epochs = 30, validation_data = (valid_X, valid_y),
                    verbose = 2, steps_per_epoch= 200)
```

Epoch 1/30
200/200 - 72s - loss: 1.0519 - accuracy: 0.6371 - val_loss: 0.1685 - val_accuracy: 0.9467
Epoch 2/30
200/200 - 70s - loss: 0.3084 - accuracy: 0.9043 - val_loss: 0.0778 - val_accuracy: 0.9751
Epoch 3/30
200/200 - 70s - loss: 0.2047 - accuracy: 0.9338 - val_loss: 0.0367 - val_accuracy: 0.9881
Epoch 4/30
200/200 - 70s - loss: 0.1714 - accuracy: 0.9470 - val_loss: 0.0363 - val_accuracy: 0.9896
Epoch 5/30
200/200 - 70s - loss: 0.1385 - accuracy: 0.9581 - val_loss: 0.0272 - val_accuracy: 0.9922
Epoch 6/30
200/200 - 70s - loss: 0.1256 - accuracy: 0.9627 - val_loss: 0.0450 - val_accuracy: 0.9896
Epoch 7/30

```

200/200 - 71s - loss: 0.1127 - accuracy: 0.9668 - val_loss: 0.0256 - val_accuracy: 0.9907
Epoch 8/30
200/200 - 72s - loss: 0.1162 - accuracy: 0.9650 - val_loss: 0.0545 - val_accuracy: 0.9850
Epoch 9/30
200/200 - 70s - loss: 0.0982 - accuracy: 0.9718 - val_loss: 0.0291 - val_accuracy: 0.9907
Epoch 10/30
200/200 - 70s - loss: 0.0886 - accuracy: 0.9731 - val_loss: 0.0329 - val_accuracy: 0.9943
Epoch 11/30
200/200 - 69s - loss: 0.0960 - accuracy: 0.9735 - val_loss: 0.0240 - val_accuracy: 0.9959
Epoch 12/30
200/200 - 70s - loss: 0.0906 - accuracy: 0.9737 - val_loss: 0.0168 - val_accuracy: 0.9953
Epoch 13/30
200/200 - 72s - loss: 0.0735 - accuracy: 0.9780 - val_loss: 0.0229 - val_accuracy: 0.9933
Epoch 14/30
200/200 - 70s - loss: 0.0920 - accuracy: 0.9721 - val_loss: 0.0266 - val_accuracy: 0.9927
Epoch 15/30
200/200 - 70s - loss: 0.0927 - accuracy: 0.9750 - val_loss: 0.0157 - val_accuracy: 0.9959
Epoch 16/30
200/200 - 70s - loss: 0.0894 - accuracy: 0.9741 - val_loss: 0.0228 - val_accuracy: 0.9948
Epoch 17/30
200/200 - 70s - loss: 0.0783 - accuracy: 0.9775 - val_loss: 0.0375 - val_accuracy: 0.9922
Epoch 18/30
200/200 - 70s - loss: 0.0799 - accuracy: 0.9788 - val_loss: 0.0235 - val_accuracy: 0.9938
Epoch 19/30
200/200 - 71s - loss: 0.0856 - accuracy: 0.9754 - val_loss: 0.0345 - val_accuracy: 0.9902
Epoch 20/30
200/200 - 71s - loss: 0.0793 - accuracy: 0.9765 - val_loss: 0.0194 - val_accuracy: 0.9948
Epoch 21/30
200/200 - 70s - loss: 0.0836 - accuracy: 0.9798 - val_loss: 0.0288 - val_accuracy: 0.9927
Epoch 22/30
200/200 - 70s - loss: 0.0730 - accuracy: 0.9798 - val_loss: 0.0162 - val_accuracy: 0.9953
Epoch 23/30
200/200 - 71s - loss: 0.0804 - accuracy: 0.9789 - val_loss: 0.0646 - val_accuracy: 0.9860
Epoch 24/30
200/200 - 70s - loss: 0.0767 - accuracy: 0.9791 - val_loss: 0.0540 - val_accuracy: 0.9876
Epoch 25/30
200/200 - 70s - loss: 0.0866 - accuracy: 0.9771 - val_loss: 0.0124 - val_accuracy: 0.9959
Epoch 26/30
200/200 - 70s - loss: 0.0819 - accuracy: 0.9788 - val_loss: 0.0297 - val_accuracy: 0.9938
Epoch 27/30
200/200 - 70s - loss: 0.0826 - accuracy: 0.9780 - val_loss: 0.0208 - val_accuracy: 0.9969
Epoch 28/30
200/200 - 70s - loss: 0.0730 - accuracy: 0.9812 - val_loss: 0.0191 - val_accuracy: 0.9959
Epoch 29/30
200/200 - 70s - loss: 0.0910 - accuracy: 0.9751 - val_loss: 0.0327 - val_accuracy: 0.9917
Epoch 30/30
200/200 - 70s - loss: 0.0697 - accuracy: 0.9818 - val_loss: 0.0216 - val_accuracy: 0.9933

```

Evaluating the model

In [8]:

```

score = model.evaluate(test_X, test_y, verbose=0)
print('Test Score = ',score[0])
print('Test Accuracy =', score[1])

```

```

Test Score = 0.0032084500417113304
Test Accuracy = 0.998031497001648

```

Part 2: Reading the Sudoku

Random Selection of an Image

In [79]:

```

folder=r"../input/sudoku-box-detection/aug"

```

```

a=random.choice(os.listdir(folder))
print(a)
sudoku_a = cv2.imread(folder+'/'+a)
plt.figure()
plt.imshow(sudoku_a)
plt.show()

```

_197_2443114.jpeg

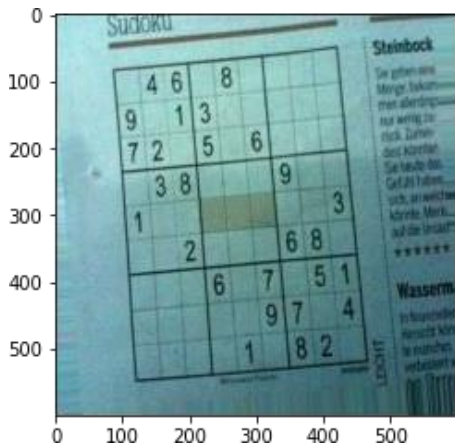


Image preprocessing

In [80]:

```

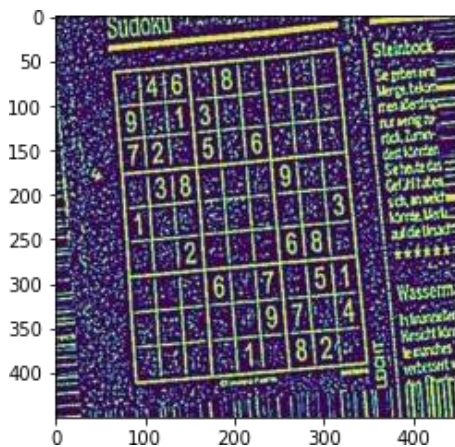
sudoku_a = cv2.resize(sudoku_a, (450,450))

# function to greyscale, blur and change the receptive threshold of image
def preprocess(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (3,3),6)
    #blur = cv2.bilateralFilter(gray,9,75,75)
    threshold_img = cv2.adaptiveThreshold(blur,255,1,1,11,2)
    return threshold_img

threshold = preprocess(sudoku_a)

plt.figure()
plt.imshow(threshold)
plt.show()

```



Contour Detection

In [81]:

```

# Finding the outline of the sudoku puzzle in the image

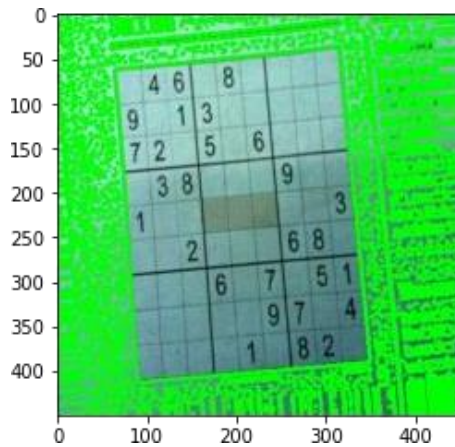
```

```

contour_1 = sudoku_a.copy()
contour_2 = sudoku_a.copy()
contour, hierarchy = cv2.findContours(threshold,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE
)
cv2.drawContours(contour_1, contour,-1,(0,255,0),3)

plt.figure()
plt.imshow(contour_1)
plt.show()

```



In [82]:

```

def main_outline(contour):
    biggest = np.array([])
    max_area = 0
    for i in contour:
        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

def reframe(points):
    points = points.reshape((4, 2))
    points_new = np.zeros((4,1,2),dtype = np.int32)
    add = points.sum(1)
    points_new[0] = points[np.argmin(add)]
    points_new[3] = points[np.argmax(add)]
    diff = np.diff(points, axis =1)
    points_new[1] = points[np.argmin(diff)]
    points_new[2] = points[np.argmax(diff)]
    return points_new

def splitcells(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

black_img = np.zeros((450,450,3), np.uint8)
biggest, maxArea = main_outline(contour)
if biggest.size != 0:
    biggest = reframe(biggest)
    cv2.drawContours(contour_2,biggest,-1, (0,255,0),10)
    pts1 = np.float32(biggest)
    pts2 = np.float32([[0,0],[450,0],[0,450],[450,450]])
    matrix =cv2.getPerspectiveTransform(pts1,pts2)
    imagewrap = cv2.warpPerspective(sudoku_a,matrix,(450,450))

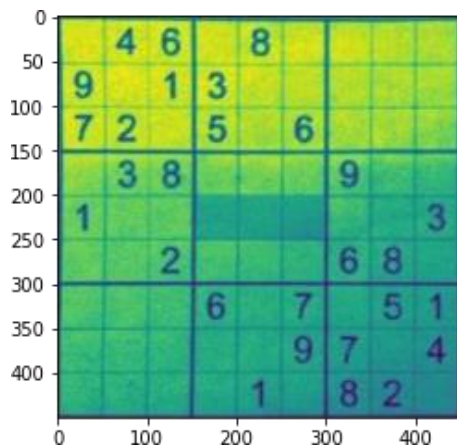
```

```

imagewrap =cv2.cvtColor(imagewrap, cv2.COLOR_BGR2GRAY)

plt.figure()
plt.imshow(imagewrap)
plt.show()

```



Splitting cells and classifying the digits

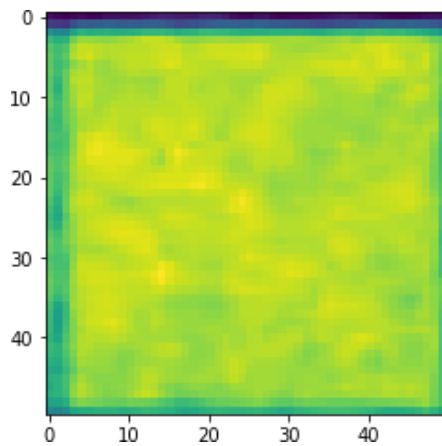
In [83]:

```

sudoku_cell = splitcells(imagewrap)

plt.figure()
plt.imshow(sudoku_cell[58])
plt.show()

```



In [84]:

```

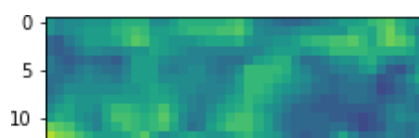
def CropCell(cells):
    Cells_cropped = []
    for image in cells:

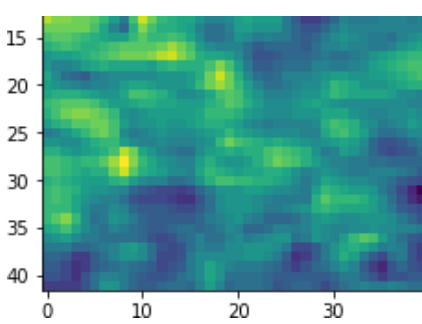
        img = np.array(image)
        img = img[4:46, 6:46]
        img = Image.fromarray(img)
        Cells_cropped.append(img)

    return Cells_cropped

sudoku_cell_cropped= CropCell(sudoku_cell)
plt.figure()
plt.imshow(sudoku_cell_cropped[58])
plt.show()

```





In [85]:

```
def read_cells(cell,model):

    result = []
    for image in cell:
        # preprocess the image as it was in the model
        img = np.asarray(image)
        img = img[4:img.shape[0] - 4, 4:img.shape[1] -4]
        img = cv2.resize(img, (32, 32))
        img = img / 255
        img = img.reshape(1, 32, 32, 1)
        # getting predictions and setting the values if probabilities are above 65%

        predictions = model.predict(img)
        classIndex = model.predict_classes(img)
        probabilityValue = np.amax(predictions)

        if probabilityValue > 0.65:
            result.append(classIndex[0])
        else:
            result.append(0)
    return result

grid = read_cells(sudoku_cell_cropped, model)
grid = np.asarray(grid)
```

Part 3: Solving the Sudoku

In [86]:

```
grid = np.reshape(grid, (9,9))
grid
```

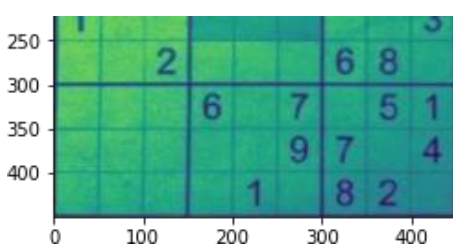
Out[86]:

```
array([[0, 0, 6, 0, 0, 0, 0, 0, 0],
       [9, 0, 0, 0, 0, 0, 0, 0, 0],
       [7, 2, 0, 5, 0, 6, 0, 0, 0],
       [0, 3, 8, 0, 0, 0, 9, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 3],
       [0, 0, 2, 0, 0, 0, 6, 8, 0],
       [0, 0, 0, 6, 0, 7, 0, 0, 0],
       [0, 0, 0, 0, 0, 9, 7, 0, 4],
       [0, 0, 0, 0, 0, 0, 8, 2, 0]])
```

In [87]:

```
plt.figure()
plt.imshow(imagewrap)
plt.show()
```





In [88]:

```
#This function finds the next box to solve

def next_box(quiz):
    for row in range(9):
        for col in range(9):
            if quiz[row][col] == 0:
                return (row, col)
    return False

#Function to fill in the possible values by evaluating rows collumns and smaller cells

def possible (quiz,row, col, n):
    #global quiz
    for i in range (0,9):
        if quiz[row][i] == n and row != i:
            return False
    for i in range (0,9):
        if quiz[i][col] == n and col != i:
            return False

    row0 = (row)//3
    col0 = (col)//3
    for i in range(row0*3, row0*3 + 3):
        for j in range(col0*3, col0*3 + 3):
            if quiz[i][j]==n and (i,j) != (row, col):
                return False
    return True

#Recursion function to loop over untill a valid answer is found.

def solve(quiz):
    val = next_box(quiz)
    if val is False:
        return True
    else:
        row, col = val
        for n in range(1,10): #n is the possible solution
            if possible(quiz,row, col, n):
                quiz[row][col]=n
                if solve(quiz):
                    return True
            else:
                quiz[row][col]=0
        return

def Solved(quiz):
    for row in range(9):
        if row % 3 == 0 and row != 0:
            print(".....")

        for col in range(9):
            if col % 3 == 0 and col != 0:
                print("|", end=" ")

            if col == 8:
                print(quiz[row][col])
            else:
                print(str(quiz[row][col]) + " ", end="")
```

In [89]:

```
solve(grid)
```

```
Out[89]:
```

```
True
```

```
In [90]:
```

```
if solve(grid):  
    Solved(grid)  
else:  
    print("Solution don't exist. Model misread digits.")
```

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

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

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