Capstone Project Report

By - Pratyush Bhatnagar

Solving Sudoku with Convolution Neural Network | Keras

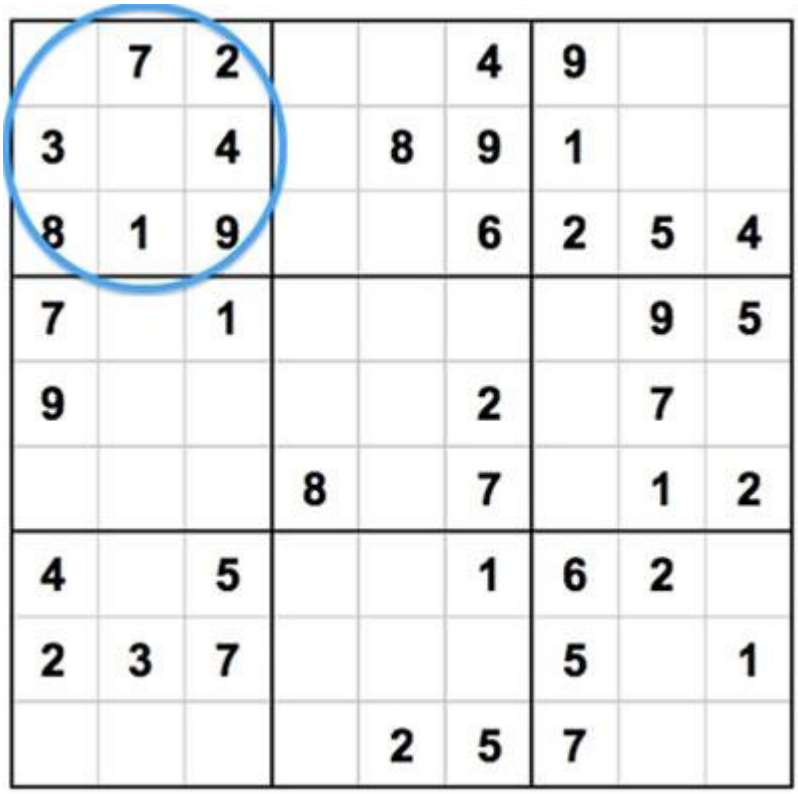
What is Sudoku?

Sudoku is played on a grid of 9 x 9 spaces. Within the rows and columns are 9 “squares” (made up of 3 x 3 spaces). Each row, column and square (9 spaces each) needs to be filled out with the numbers 1-9, without repeating any numbers within the row, column or square. Does it sound complicated? As you can see from the image below of an actual Sudoku grid, each

Sudoku grid comes with a few spaces already filled in; the more spaces filled in, the easier the game – the more difficult Sudoku puzzles have very few spaces that are already filled in.

Sudoku is a game of logic and reasoning, so you shouldn't have to guess. If you don't know what number to put in a certain space, keep scanning the other areas of the grid until you seen an opportunity to place a number.

Eg.- How to solve?



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

As you can see, in the upper left square (circled in blue), this square already has 7 out of the 9 spaces filled in. The only numbers missing from the square are 5 and 6. By seeing which numbers are missing from each square, row, or column, we can use process of elimination and deductive reasoning to decide which numbers need to go in each blank space.

For example, in the upper left square, we know we need to add a 5 and a 6 to be able to complete the square, but based on the neighbouring rows and squares we cannot clearly deduce which number to add in which space. This means that we should ignore the upper left square for now, and try to fill in spaces in some other areas of the grid instead.

In this way, we can solve the whole sudoku very easily.

Metrics used to check performance

For evaluating the performance and accuracy, we want to output the correct solved sudoku as an array of string. So,

Accuracy %age = (total no. of correct outputs / total no. of inputs) x 100

Testing 100 games

In [44]:
```
def test_accuracy(feats, labels):

    correct = 0

    for i,feat in enumerate(feats):

        pred = inference_sudoku(feat)

        true = labels[i].reshape((9,9))+1

        if(abs(true - pred).sum()==0):
            correct += 1

    print(correct/feats.shape[0])
```

In [45]:
```
test_accuracy(x_test[:100], y_test[:100])
```

```
1.0
```

- Network was able to solve the puzzles with 99% accuracy.

- **Way to verify your output**

```
In [10]: np.sum(game, axis=1)
Out[10]: array([45, 45, 45, 45, 45, 45, 45, 45, 45], dtype=int64)
```

- The sum of every row will be (1+2+3+4+5+6+7+8+9) = 45.
- Therefore, we can check the sum of every row as 45.

Problem Statement

A project to create a Sudoku solver from inputs for educational purposes to explore themes of:

- Puzzle Solving Algorithms
- Machine Learning
- Deep Learning

This is a **Supervised Learning** technique where we have a pair of array of string as input and the corresponding output. The model gets trained on the dataset we provide.

The Sudoku class can take an input as an array of natural numbers and produce solved puzzle as an output. Output can be as a formatted string, dictionary or an array of string.

Strategy or Solution path(all steps elaborated below)

Step 1 - For the code to work I have to import the required libraries.

Step 2 - Next, I collected the Dataset from Kaggle

Step 3 – Pre-processing it and loading it into training set.(80%)

Step 4 - I trained the network for 2 epochs, with batch size 64. The learning rate for the first epoch was 0.001 and for second epochs I reduced it to 0.0001. The final training loss settled down to 0.34. I tried a few different network architecture and strategies but could not reduce the loss further so I went ahead with this network. Its time to test the network.(Elaborated below)

Step 5 – Testing the accuracy of the model(testing set – 20%)

- Here is the or development stages that describes the path followed in making a **Sudoku Solver**.

Stage 1- Importing the required libraries

In [1]:
```
import copy
import keras
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from model import get_model
```

- **Keras** is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear & actionable error messages. It also has extensive documentation and developer guides.

- **Tensorflow** can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. Tensorflow is a symbolic math library based on dataflow and differentiable programming.

- **NumPy** is a python **library** used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. **NumPy** stands for Numerical Python.

- **Pandas** is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

- **Scikit-learn** features
various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, *k*-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Stage 2- Collecting the Dataset

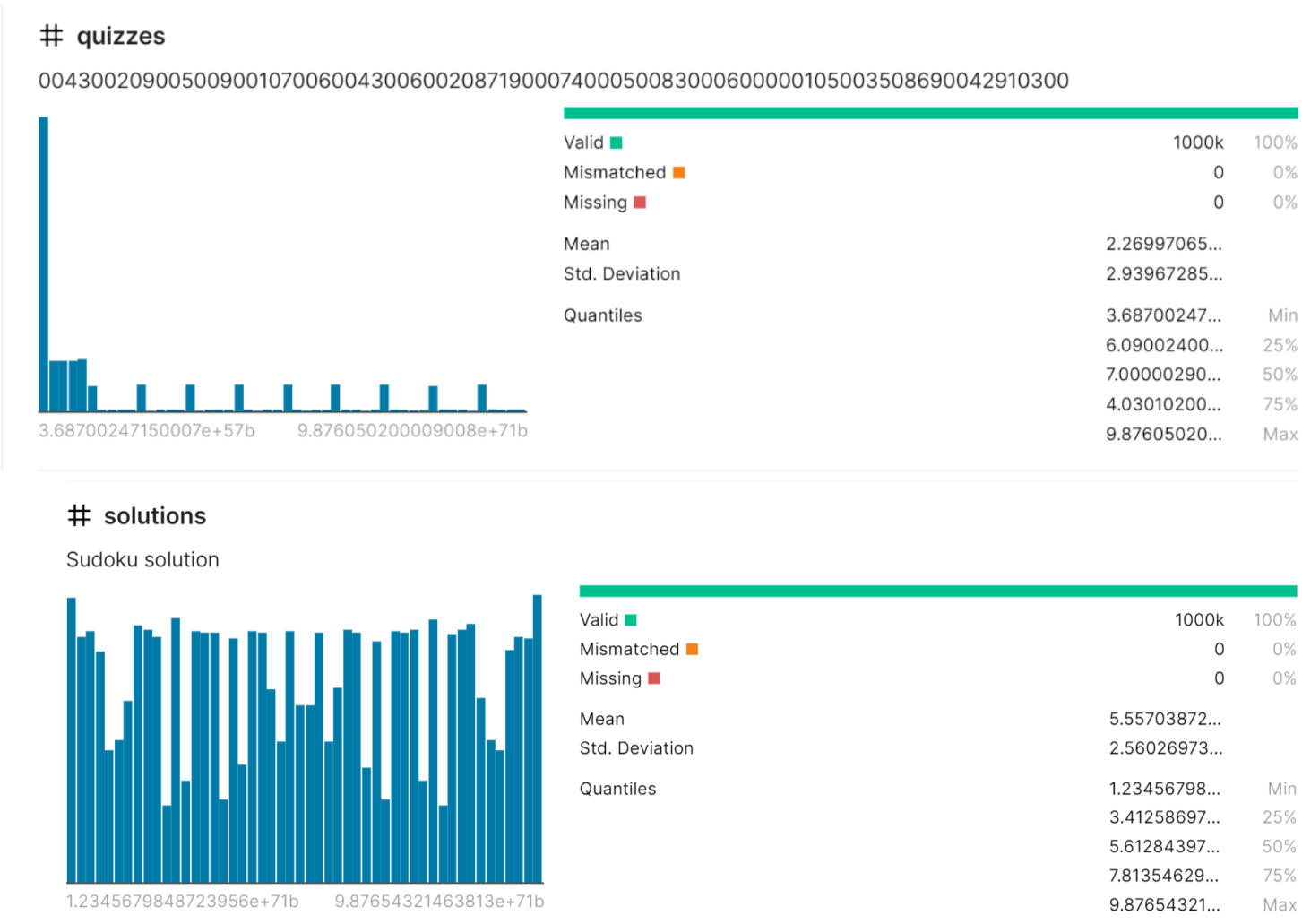
```
In [2]: def get_data(file):

            data = pd.read_csv(file)

            feat_raw = data['quizzes']
            label_raw = data['solutions']

            feat = []
            label = []
```

- Download the **dataset** for this project. I found the following data on Kaggle, which contains 1 million unsolved and solved Sudoku games. Please take a look at the data below.

	quizzes	solutions
0	004300209005009001070060043006002087190007400...	864371259325849761971265843436192587198657432 2...
1	040100050107003960520008000000000017000906800 8...	346179258187523964529648371965832417472916835 8...
2	600120384008459072000006005000264030070080006 9...	695127384138459672724836915851264739273981546 9...
3	497200000100400005000016098620300040300900000 0...	497258316186439725253716498629381547375964182 8...
4	005910308009403060027500100030000201000820007 0...	465912378189473562327568149738645291954821637 2...

Sudoku Dataset

The dataset contains 2 columns. The column quizzes has the unsolved games and the column solutions has respective solved games. Each game is represented by a string of 81 numbers. Following is a 9x9 sudoku converted from the string. The number 0 represents the blank position in unsolved games.



Stage 3- Pre-processing and Loading the Dataset

```
    for i in feat_raw:

        x = np.array([int(j) for j in i]).reshape((9,9,1))
        feat.append(x)

    feat = np.array(feat)
    feat = feat/9
    feat -= .5

    for i in label_raw:

        x = np.array([int(j) for j in i]).reshape((81,1)) - 1
        label.append(x)

    label = np.array(label)

    del(feat_raw)
    del(label_raw)

    x_train, x_test, y_train, y_test = train_test_split(feat, label, test_size=0.2, random_state=42)

    return x_train, x_test, y_train, y_test
```

Load the data

```
In [3]: x_train, x_test, y_train, y_test = get_data('sudoku.csv')
```

After reading the data from a (.csv) file and splitting the training(80%) and testing data(20%), our task is to feed the unsolved sudoku to a neural network and get the solved sudoku out of it. This means we have to feed 81 numbers to the network and need to have 81 output numbers from it.

Normalization step-

We have to convert the input data(unsolved games) into a 3D array since we have to feed it to the CNN. I have converted each string of 81 numbers in a shape of (9,9,1). Then I normalized the input data by dividing it with 9 and subtracting 0.5. By doing so data becomes zero mean-centred and in the range of (-0.5 - 0.5). **Neural networks generally perform better with zero centred normalized data.**

Visualization of Project

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

Initally

```
In [12]: game = '''
           1 2 3 0 8 5 4 0 0
           0 0 0 0 3 4 0 2 6
           0 0 6 0 1 0 0 0 3
           0 0 7 9 2 0 0 0 0
           3 9 0 0 0 0 0 6 2
           0 0 5 4 7 3 0 0 9
           0 7 2 0 0 0 9 0 1
           0 0 0 1 0 7 0 4 0
           9 5 0 3 4 2 0 0 8
         '''

game = solve_sudoku(game)

print('solved puzzle:\n')
print(game)
```

```
solved puzzle:

[[1 2 3 6 8 5 4 9 7]
 [5 8 9 7 3 4 1 2 6]
 [7 4 6 2 1 9 5 8 3]
 [8 1 7 9 2 6 3 5 4]
 [3 9 4 8 5 1 7 6 2]
 [2 6 5 4 7 3 8 1 9]
 [4 7 2 5 6 8 9 3 1]
 [6 3 8 1 9 7 2 4 5]
 [9 5 1 3 4 2 6 7 8]]
```

Finally

These figures represents the same games-

- **Left one represents the initial puzzle,**
- **Right one represents input & its output solution.**

Stage 4- Training the model

Train your own Model

In [39]:
```
model = get_model()

adam = keras.optimizers.Adam(lr=.001)
model.compile(loss='sparse_categorical_crossentropy', optimizer=adam)

model.fit(x_train, y_train, batch_size=32, epochs=2)
```

```
Epoch 1/2
25000/25000 [==============================] - 1846s 74ms/step - loss: 0.4464
Epoch 2/2
25000/25000 [==============================] - 1643s 66ms/step - loss: 0.3598
```

Out[39]:
```
<tensorflow.python.keras.callbacks.History at 0x1b20255b248>
```

I trained the network for 2 epochs, with batch size 64. The learning rate for the first epoch was 0.001 and for second epochs I reduced it to 0.0001. The final training loss settled down to 0.34. I tried a few different network architecture and strategies but could not reduce the loss further so I went ahead with this network. Its time to test the network.

Algorithm and Techniques (Using CNN model)

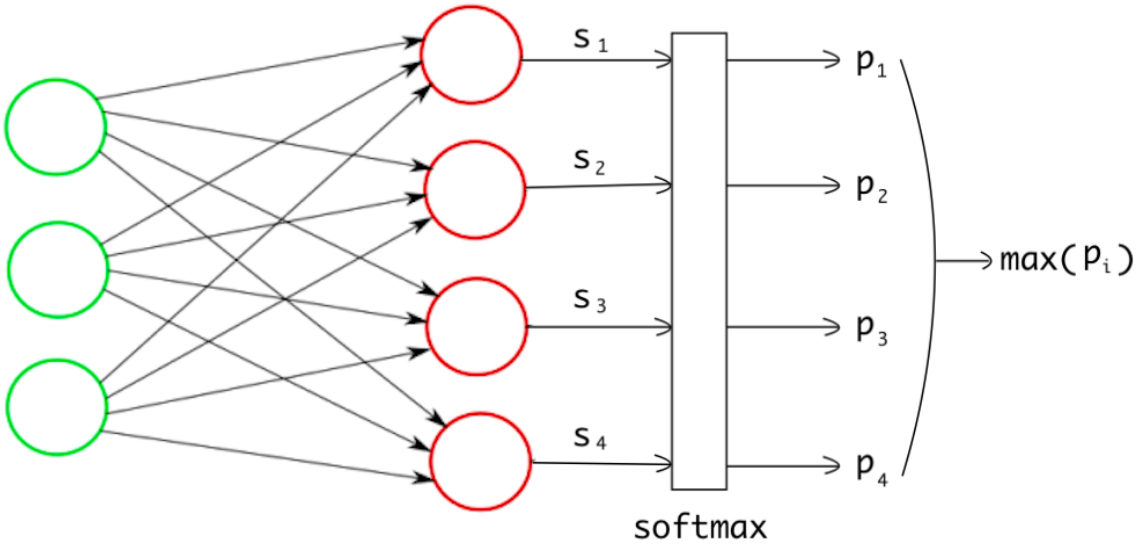
```
import keras
from keras.layers import Activation
from keras.layers import Conv2D, BatchNormalization, Dense, Flatten, Reshape

def get_model():

    model = keras.models.Sequential()

    model.add(Conv2D(64, kernel_size=(3,3), activation='relu', padding='same', input_shape=(9,9,1)))
    model.add(BatchNormalization())
    model.add(Conv2D(64, kernel_size=(3,3), activation='relu', padding='same'))
    model.add(BatchNormalization())
    model.add(Conv2D(128, kernel_size=(1,1), activation='relu', padding='same'))

    model.add(Flatten())
    model.add(Dense(81*9))
    model.add(Reshape((-1, 9)))
    model.add(Activation('softmax'))

    return model
```

CNNs are an extension of multilayer perceptrons, which can learn filters that need to be computed by the machine learning models. Since the training process involves learning about patterns from smaller patterns, computations usually are time-consuming and may require GPU-based implementation.

A **convolutional neural network** consists of distinct hidden layers in addition to the input and output layers. These distinct layers usually consist of convolutional layer with filters that can be learned, rectified linear unit layer for application of activation function, pooling layer for down-sampling and loss layer for specification of penalization for incorrect output. We use Keras with TensorFlow as backend to process the CNN model and provide a comparison between the distinct CNN implementations formed by choosing different hyperparameters associated with each layer.

get_model method description-

In a typical multi-class classification, the neural network outputs scores for each class. Then we apply softmax function on the final scores to convert them into probabilities. And the data is classified into a class that has the highest probability



value(refer to the above image).But in sudoku, the scenario is different. We have to get **81** numbers for each position in the sudoku game, not just one. And we have a total of **9** classes for each number because a number can fall in a range of 1 to 9.

To comply with this design, our network should output 81x9 numbers. Where each row represents one of the 81 numbers, and each column represents one of 9 classes. Then we can apply softmax and take the maximum along with each row so that we have 81 numbers classified into one of the 9 classes.

I created the following simple network for this task. The network consists of 3 **Convolution** layers and one **Dense** layer on top for classification.

Note I am reshaping the output of the Dense layers in a shape of (81, 9) then adding a softmax layer on it. I compiled the model with sparse categorical crossentropy loss and adam optimizer.

Since we are using scc loss, we don't need to provide a one-hot encoded target vector. Our target vectors shape is (81, 1) where the vector elements represent the true class of 81 numbers.

Solve Sudoku by filling blank positions one by one

In [5]:
```
def norm(a):
    return (a/9)-.5
```

In [6]:
```
def denorm(a):
    return (a+.5)*9
```

In [7]:
```
def inference_sudoku(sample):

    '''
        This function solve the sudoku by filling blank positions one by one.
    '''

    feat = copy.copy(sample)

    while(1):

        out = model.predict(feat.reshape((1,9,9,1)))
        out = out.squeeze()

        pred = np.argmax(out, axis=1).reshape((9,9))+1
        prob = np.around(np.max(out, axis=1).reshape((9,9)), 2)

        feat = denorm(feat).reshape((9,9))
        mask = (feat==0)

        if(mask.sum()==0):
            break

        prob_new = prob*mask

        ind = np.argmax(prob_new)
        x, y = (ind//9), (ind%9)

        val = pred[x][y]
        feat[x][y] = val
        feat = norm(feat)

    return pred
```

As humans when we solve Sudoku, we fill numbers one by one. We do not simply look at the sudoku once and fill all the numbers. The advantage of filling the numbers one by one is that each time we fill a number we keep getting a better idea about the next move.

I implemented the same approach while solving the sudoku now. Instead of predicting all 81 numbers at once, I am picking just one number among all blank position that has the highest probability value, and filling that number in the sudoku. After filling one number we again feed this puzzle to the network and

make a prediction. We keep repeating this and filling the blank positions one by one with the highest probability number until we are left with no blank positions.

This approach boosted the performance and the network was able to solve almost all the games in this dataset. Test accuracy on 1000 games was 0.99.

Difference in Performance of Benchmark model and this model

Most of the sudoku solvers use **Recursion** or **Backtracking** as a way to solve sudoku problems. One such example can be using MATLAB-

Solving a Sudoku puzzle in MATLAB may not be a challenge, but to solve it quickly is not so easy. More than 20 Sudoku solvers have been submitted in MATLAB File Exchange. Most of them use recursive approaches, which, without non-recursive algorithm support, could be very inefficient.

It maintains a Candidate Map (C, a 81 x 9 logical array) to discover the solution. Initially, without any number on the Board (B, 9 x 9 array), all elements of C is true, indicating that at any position, there are 9 candidates. Each of 9 x 9 grids belongs to a row, a column and a block, marked as (r,c,b). Let 81 x 1 vectors I, J and K represent row, column and block indices of all grids respectively. When a specific number, n is

put into the grid (r,c,b), i.e. B(r,c)=n, then all grids in L=(I==r | J==c | K==b) should not have candidate n, hence C(L,n)=false. If a row of C has only one candidate, i.e. L=sum(C,2)==1, then, the corresponding grid is solved, i.e. B(L(k))=find(C(L(k),:),'first'). Many other rules have been implemented to update C.

If a puzzle cannot be completely solved by the non-recursive solver, a recursive solver will be called. The recursive solver will always try the grid, which has the minimum number of candidates indicated by C. For example, a grid m has two candidates, n1 and n2. The recursive solver will set B(m)=n1 and call the non-recursive solver to find the rest solutions. If an error is returned by the non-recursive solver, then the recursive solver will set B(m)=n2 to call the non-recursive solver again. If both branches are failed, then the puzzle is not valid.

Due to Recursion and Backtracking, the Time Complexity of the model gets increased and gives delayed outputs. So, for the sake of time and other factors, I chose Deep Learning to be the best approach to solve sudoku puzzles. This model would be the best benchmark for other sudoku solvers.

Complexity Analysis of Backtracking approach:

- **Time complexity:** O(9^(n*n)).
  For every unassigned index, there are 9 possible options so the time complexity is O(9^(n*n)).
- **Space Complexity:** O(n*n).
  To store the output array a matrix is needed.

Complexity Analysis of our CNN model approach:

Time complexity: O(n^4)
Space complexity: O(n*n)

Results (Implications)-

Stage 5- Testing the model

Test your own game

In [8]:
```
def solve_sudoku(game):

    game = game.replace('\n', '')
    game = game.replace(' ', '')
    game = np.array([int(j) for j in game]).reshape((9,9,1))
    game = norm(game)
    game = inference_sudoku(game)
    return game
```

This method (solve_sudoku) takes an array of number string and outputs the solved sudoku.

One such example is given of my model-

```
game = '''
          0 8 0 0 3 2 0 0 1
          7 0 3 0 8 0 0 0 2
          5 0 0 0 0 7 0 3 0
          0 5 0 0 0 1 9 7 0
          6 0 0 7 0 9 0 0 8
          0 4 7 2 0 0 0 5 0
          0 2 0 6 0 0 0 0 9
          8 0 0 0 9 0 3 0 5
          3 0 0 8 2 0 0 1 0
      '''

game = solve_sudoku(game)

print('solved puzzle:\n')
print(game)
```

```
solved puzzle:

[[4 8 9 5 3 2 7 6 1]
 [7 1 3 4 8 6 5 9 2]
 [5 6 2 9 1 7 8 3 4]
 [2 5 8 3 4 1 9 7 6]
 [6 3 1 7 5 9 2 4 8]
 [9 4 7 2 6 8 1 5 3]
 [1 2 5 6 7 3 4 8 9]
 [8 7 6 1 9 4 3 2 5]
 [3 9 4 8 2 5 6 1 7]]
```

Testing 100 games

In [44]:
```
def test_accuracy(feats, labels):

    correct = 0

    for i,feat in enumerate(feats):

        pred = inference_sudoku(feat)

        true = labels[i].reshape((9,9))+1

        if(abs(true - pred).sum()==0):
            correct += 1

    print(correct/feats.shape[0])
```

In [45]:
```
test_accuracy(x_test[:100], y_test[:100])
```

```
1.0
```

Accuracy %age = (total no. of correct outputs / total no. of inputs) x 100

Our model is robust because it has 99% accuracy rate and gives the right solution for the data not in the provided dataset too.

- **Way to verify your output**

```
In [10]: np.sum(game, axis=1)
Out[10]: array([45, 45, 45, 45, 45, 45, 45, 45, 45], dtype=int64)
```

- The sum of every row will be (1+2+3+4+5+6+7+8+9) = 45.
- Therefore, we can check the sum of every row as 45.

Complexity Analysis of Backtracking approach(Benchmark model):

- **Time complexity:** O(9^(n*n)).
  For every unassigned index, there are 9 possible options so the time complexity is O(9^(n*n)).
- **Space Complexity:** O(n*n).
  To store the output array a matrix is needed.

Complexity Analysis of our CNN model approach:

Time complexity: O(n^4)

- Because of 3 CNN layers and 1 dense layer

Space complexity: O(n*n)

#A Backtracking program

```
# in Python to solve Sudoku problem
import time

# A Utility Function to print the Grid
def print_grid(arr):
        for i in range(9):
                for j in range(9):
                        print arr[i][j],
                print ('n')
def find_empty_location(arr, l):
        for row in range(9):
                for col in range(9):
                        if(arr[row][col]== 0):
```

```
                        l[0]= row
                        l[1]= col
                        return True
        return False

# Returns a boolean which indicates
# whether any assigned entry
# in the specified row matches
# the given number.
def used_in_row(arr, row, num):
        for i in range(9):
                if(arr[row][i] == num):
                        return True
        return False

# Returns a boolean which indicates
# whether any assigned entry
# in the specified column matches
# the given number.
def used_in_col(arr, col, num):
        for i in range(9):
                if(arr[i][col] == num):
                        return True
        return False

# Returns a boolean which indicates
# whether any assigned entry
# within the specified 3x3 box
# matches the given number
def used_in_box(arr, row, col, num):
        for i in range(3):
                for j in range(3):
                        if(arr[i + row][j + col] == num):
                                return True
        return False

# Checks whether it will be legal
# to assign num to the given row, col
```

```
# Returns a boolean which indicates
# whether it will be legal to assign
# num to the given row, col location.
def check_location_is_safe(arr, row, col, num):

        # Check if 'num' is not already
        # placed in current row,
        # current column and current 3x3 box
        return not used_in_row(arr, row, num)
        not used_in_col(arr, col, num)
        not used_in_box(arr, row - row % 3,
                                        col - col % 3, num)

# Takes a partially filled-in grid
# and attempts to assign values to
# all unassigned locations in such a
# way to meet the requirements
# for Sudoku solution (non-duplication
# across rows, columns, and boxes)
def solve_sudoku(arr):

        # 'l' is a list variable that keeps the
        # record of row and col in
        # find_empty_location Function
        l =[0, 0]

        # If there is no unassigned
        # location, we are done
        if(not find_empty_location(arr, l)):
                return True

        # Assigning list values to row and col
        # that we got from the above Function
        row = l[0]
        col = l[1]

        # consider digits 1 to 9
        for num in range(1, 10):
```

```
            # if looks promising
            if(check_location_is_safe(arr,
                                      row, col, num)):

                # make tentative assignment
                arr[row][col]= num

                # return, if success,
                # ya ! if(solve_sudoku(arr)):
                return True

                # failure, unmake & try again
                arr[row][col] = 0

        # this triggers backtracking
        return False

# Driver main function to test above functions
if __name__=="__main__":

        # creating a 2D array for the grid
        grid =[[0 for x in range(9)]for y in range(9)]

        # assigning values to the grid
        grid =[[3, 0, 6, 5, 0, 8, 4, 0, 0],
               [5, 2, 0, 0, 0, 0, 0, 0, 0],
               [0, 8, 7, 0, 0, 0, 0, 3, 1],
               [0, 0, 3, 0, 1, 0, 0, 8, 0],
               [9, 0, 0, 8, 6, 3, 0, 0, 5],
               [0, 5, 0, 0, 9, 0, 6, 0, 0],
               [1, 3, 0, 0, 0, 0, 2, 5, 0],
               [0, 0, 0, 0, 0, 0, 0, 7, 4],
               [0, 0, 5, 2, 0, 6, 3, 0, 0]]
        start=time.time()
        if(solve_sudoku(grid))
print_grid(grid)
      end=time.time()
      print(end-start)
```

Time taken by backtracking algorithm- 6.19888305664e-05 sec

See the same input and output in both models below -

Difference of time complexities

```
    # assigning values to the grid
    grid =[[1, 2, 3, 0, 8, 5, 4, 0, 0],
           [0, 0, 0, 0, 3, 4, 0, 2, 6],
           [0, 0, 6, 0, 1, 0, 0, 0, 3],
           [0, 0, 7, 9, 2, 0, 0, 0, 0],
           [3, 9, 0, 0, 0, 0, 0, 6, 2],
           [0, 0, 5, 4, 7, 3, 0, 0, 9],
           [0, 7, 2, 0, 0, 0, 9, 0, 1],
           [0, 0, 0, 1, 0, 7, 0, 4, 0],
           [9, 5, 0, 3, 4, 2, 0, 0, 8]]
    start=time.time()
    if(solve_sudoku(grid)):
        print_grid(grid)
        end=time.time()
        print(end-start)

[[1 2 3 6 8 5 4 9 7]
 [5 8 9 7 3 4 1 2 6]
 [7 4 6 2 1 9 5 8 3]
 [8 1 7 9 2 6 3 5 4]
 [3 9 4 8 5 1 7 6 2]
 [2 6 5 4 7 3 8 1 9]
 [4 7 2 5 6 8 9 3 1]
 [6 3 8 1 9 7 2 4 5]
 [9 5 1 3 4 2 6 7 8]]
6.00814819336e-05
```

Backtracking model-
Time taken- 6.00814819336e-05 s

```
In [10]: game = '''
                1 2 3 0 8 5 4 0 0
                0 0 0 0 3 4 0 2 6
                0 0 6 0 1 0 0 0 3
                0 0 7 9 2 0 0 0 0
                3 9 0 0 0 0 0 6 2
                0 0 5 4 7 3 0 0 9
                0 7 2 0 0 0 9 0 1
                0 0 0 1 0 7 0 4 0
                9 5 0 3 4 2 0 0 8
            '''
start=time.time()
game = solve_sudoku(game)

print('solved puzzle:\n')
print(game)
end=time.time()
print(end-start)

solved puzzle:

[[1 2 3 6 8 5 4 9 7]
 [5 8 9 7 3 4 1 2 6]
 [7 4 6 2 1 9 5 8 3]
 [8 1 7 9 2 6 3 5 4]
 [3 9 4 8 5 1 7 6 2]
 [2 6 5 4 7 3 8 1 9]
 [4 7 2 5 6 8 9 3 1]
 [6 3 8 1 9 7 2 4 5]
 [9 5 1 3 4 2 6 7 8]]
1.6369132995605469
```

Deep Learning model
Time taken-1.6369132995605469 s

This shows that Deep Learning model is far better in calculating or solving the puzzle very fast. The final model and solution is significant enough to have adequately solved the problem.