Capstone Project Report

By - Pratyush Bhatnagar

Solving Sudoku with Convolution Neural Network | Keras

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.

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.

- Here are the 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.**

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>
```

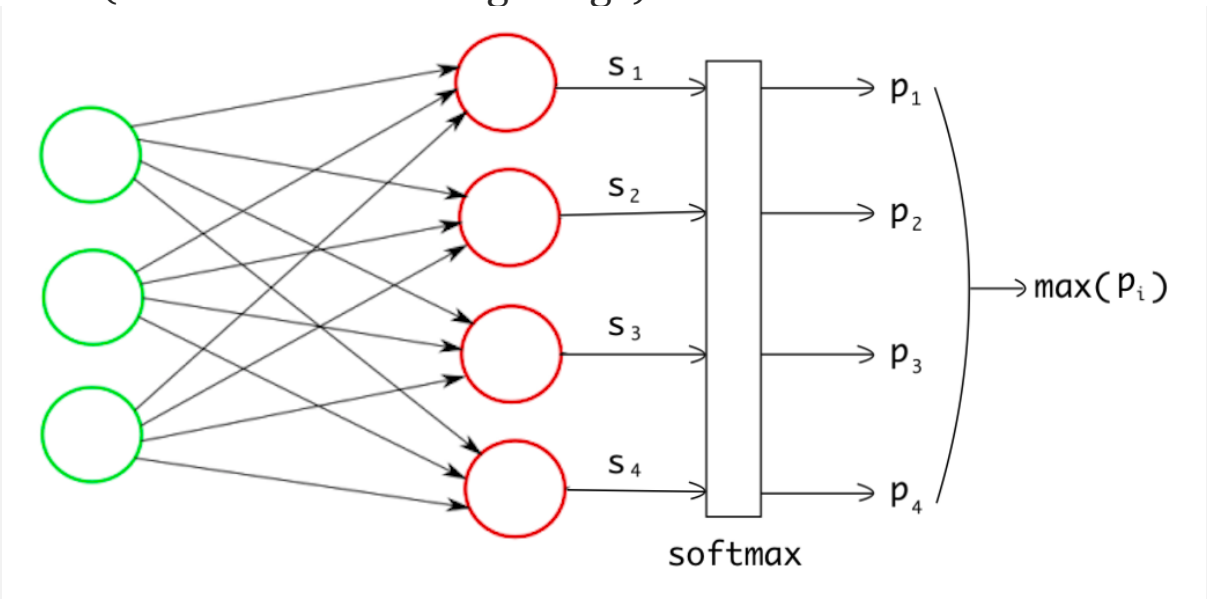
I trained the network for 2 epochs, with batch size 64. The learning rate for the first epoch was 0.001 and for second epochs I reduced it to 0.0001. The final training loss settled down to 0.34. I tried a few different network architecture and strategies but could not reduce the loss further so I went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

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
```

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 following 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
```

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.

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
```

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.

Stage 5- Testing the model

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
```

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

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

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-

```
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]]
```

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.