

Sudoku RNN in PyTorch



Josef Lindman Hörnlund · [Follow](#)

6 min read · Dec 28, 2018



27



1



Medium



Search



Write

Sign up

Sign in



The other day I went to see a friend who is working at a startup that isn't going that well. The truth is that he needed to cheer up a little, hence he spent some time solving Sudokus while feebly hoping some generous angel would walk by throwing money at them. Being a sympathetic heart I joined him in his bold endeavour, throwing my brains at some "evil" puzzles downloaded from the internet. Halfway through it I realised it would be fun to try to solve it using some neural network. Having a few hours to spare this Christmas I thus took a tiny stab at it. This short post describes how it went.

The full code is available at https://github.com/modulai/pytorch_sudoku.

TL;DR

With a simple strategy we are able to train a neural network in PyTorch that can solve Sudoku puzzles in a Kaggle dataset perfectly.

Background

Since Sudokus appeared on the scene in the 80s there has actually been some theoretic investigations into their properties: Peoples have shown that they need at least 17 clues to have a unique solution. Take a peek at [McGuire et al](#) "*There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem*" — <https://arxiv.org/abs/1201.0749> — great paper! It has also been

shown that they can be described as an exact cover problem and solved with Knuth's dancing link algorithm for example.

This method is deterministic so how about stochastic methods?

There are a couple of those as well, of course! See for example Moon et al *"Multiple Constraint Satisfaction by Belief Propagation: An Example Using Sudoku"*, 2006.

For a fairly recent approach, which from a superficial review looks very powerful, see <https://arxiv.org/abs/1711.08028>: *"Recurrent relational networks"*, Rasmus Berg Palm et al.

A Sudoku as a sequence problem

It seems reasonable to assume that a Sudoku can be solved in the framework of a constrained recurrent neural network: The input sequence is the available clues, and the output sequence is a sequence of numbers to be entered in the empty cells to complete the puzzle. It should be recurrent since filling in numbers help you solve for the remaining empty cells. The problem is furthermore constrained since no row, no column and none of the nine 3x3 boxes should contain duplicate integers.

Dataset

The dataset we used was downloaded from Kaggle (<https://www.kaggle.com/bryanpark/sudoku>) and consists of a CSV with one million Sudoku puzzles, and two columns: one with the clues, and one with

the solution.

```
>>> dataset.quizzes.head(1)
0043002090050090010700600430060020871900074000...

>>> dataset.solutions.head(1)
8643712593258497619712658434361925871986574322...
```

It should be noted that this dataset contains very simple Sudokus since they in average have around 33 clues. Hard Sudokus should have no more than 25 (roughly).

To handle this data in a network we transform each quiz to a matrix of size $(81, 9)$ where the numbers are one-hot-encoded, and empty cells are given by empty vectors. The whole dataset hence become a tensor with dimensions $(bts, 81, 9)$ where **bts** is the dataset size or the batch size. We do this transformation to be able to run a softmax on the numbers, thinking of the distribution in each cell as a categorical distribution.

Constraints

The most important rule of a Sudoku is the constraint that no row, no column and no 3x3 box can contain duplicate numbers. To describe this constraint in a form digestible by the network we build a “*constraint mask tensor*”. This tensor is of dimension $(81, 3, 81)$, where the first index enumerates the 81 cells of the puzzle, the second one enumerates the constraints (row column and box). The last index enumerates the cells that

constrain the cell in question. A row constraint for the first cell is therefore described by ones in the first nine cells for example.

To make it more concrete, let us look at some code. The box constraints are constructed with the following python snippet.

```
...
constraint_mask = torch.zeros((81, 3, 81), dtype=torch.float)
...
# box constraints
for a in range(81):
    r = a // 9
    c = a % 9
    br = 3 * 9 * (r // 3)
    bc = 3 * (c // 3)
    for b in range(9):
        r = b % 3
        c = 9 * (b // 3)
        constraint_mask[a, 2, br + bc + r + c] = 1
```

The above loop will fill in a vector of length 81 with ones for cells constraining the cell in question by the box rule, and leaves zeros everywhere else.

Building the network

To fill in an empty cell in an unfinished puzzle we score each cell with the probability of this cell being a certain number. This is done by a *softmax* activation function for each cell over the nine possible numbers. At each step in the output sequence of the recurrent network, the cell with the

highest estimated probability will be filled with the maximum from the softmax. As we loop over the empty cells, the puzzle will be solved one cell at the time.

(note that the python code in this post is not 100 % correct, but only here to convey the idea. Look at the [github](#) repo for the correct code)

To accomplish this we build a basic MLP with one hidden layer:

```
class SudokuSolver(nn.Module):
    def __init__(self, constraint_mask, n=9, hidden1=100):
        super(SudokuSolver, self).__init__()
        self.constraint_mask = constraint_mask
        self.n = n
        self.hidden1 = hidden1

        self.input_size = 3 * n

        self.l1 = nn.Linear(self.input_size,
                             self.hidden1, bias=False)
        self.a1 = nn.ReLU()
        self.l2 = nn.Linear(self.hidden1,
                             n, bias=False)
        self.softmax = nn.Softmax(dim=2)
```

The input vector for scoring each cell is constructed from the constraint masks as

```
def forward(self, x):
    c = self.constraint_mask
    ...
    constraints = (x * c).sum(dim=3)
```

This way the network knows what cells are filled and what numbers are possible to enter in the cell in question ,without violating the constraint. The input vector for each empty cell is then run through the net

```
f = constraints.reshape(bts, n * n, 3 * n)
y = self.l2(self.a1(self.l1(f)))
s = self.softmax(y)
```

The model is trained with MSE loss on the difference between the models guess and the solution, so the score from the softmax from each iteration is saved for the loss function, used in the backprop step.

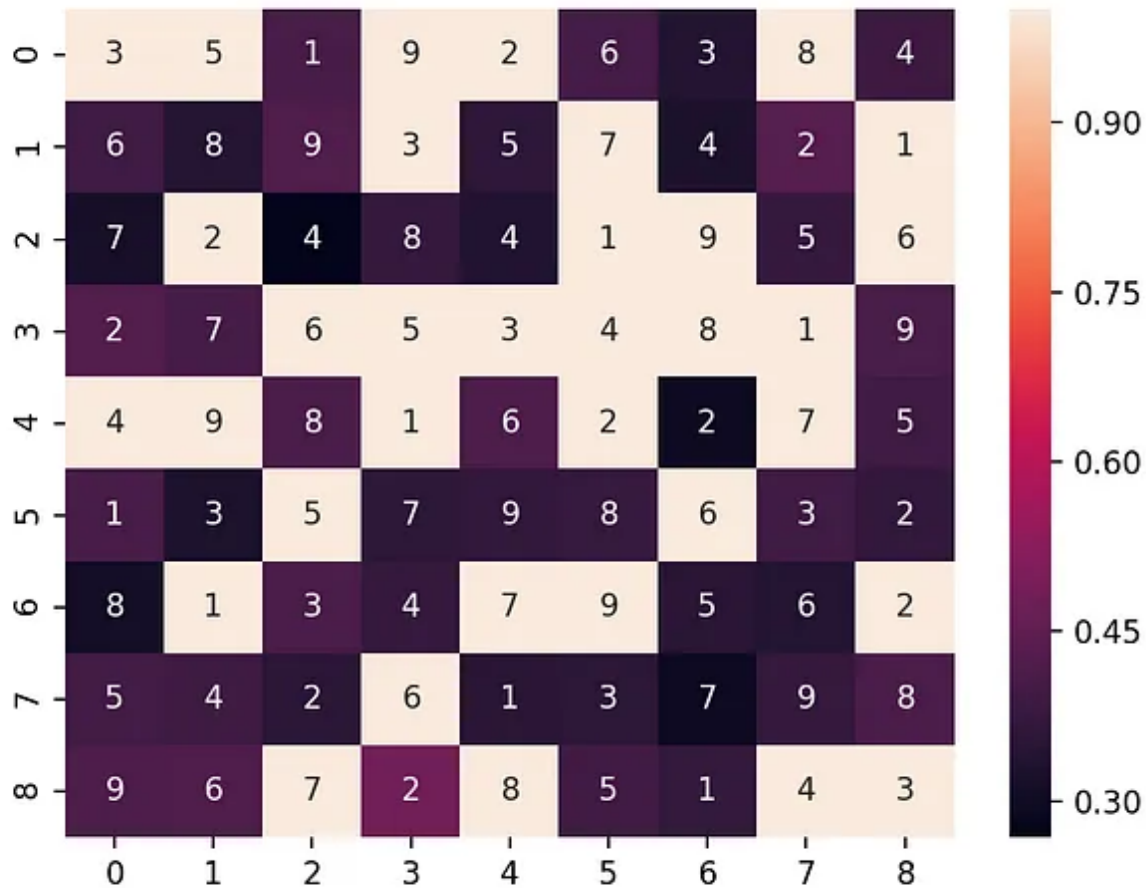
For the next iteration of the puzzle we fill in the best guess in the matrix and start over:

```
# find most probable guess
max_row_score = s.max(dim=2)[1]
puzzle_max = max_row_score.max(dim=1)[1]
# fill it in
x[puzzle_max] = 1
```

Training

Having the basic ingredients ready we can start to train the network! Below are three puzzle solutions given by the network after ten batches (with a

batch size of 100 puzzles), after 20 batches and after 50.



Guesses after 10 batches. The model is very unsure, and make a lot of errors.

Guesses after 20 batches . We see that the network is very unsure about the 8 in the middle. This number is wrong as well.

After 50 batches we see that the network is a lot more confident about its guesses, and has managed to solve the puzzle.

Validation error

Validation error (in terms of cells in error) while training. After 70 batches, the error rate has gone down to zero.

After around two epochs (100 batches with a dataset of 10k puzzles) the model can confidently solve all puzzles in the validation set. Yay!

Remarks

The Sudoku puzzles in the Kaggle dataset contain a lot of clues and this makes them quite simple. Not a lot of involved reasoning is needed by the net, which is probably why this simple approach works so well. When

facing puzzles with around 20 clues the model probably need to be extended with more information. This can be done by adding information to the feature vector or by using the recurrent network structure better to reason a few steps ahead, for example rescoring guesses that turned out wrong.

It should furthermore be pointed out that the network described in this post is quite tailor made for the problem at hand. No attempt is made at building a network that can solve general puzzles/exact cover problems.

About Modulai

Modulai is a machine learning bureau that help companies create tailored machine learning products at the core of their business. Contact us if you are interested in learning more!

Machine Learning



Written by Josef Lindman Hörnlund

9 Followers · 1 Following

Follow

Machine learning specialist at Modulai. We are hiring. www.modulai.io

Responses (1)



Write a response

What are your thoughts?



Jjjj

Jun 18, 2022



Excuse me,sir

Could I ask you some questions about it?

I have some problems during doing the program 🙏



Reply

Recommended from Medium



 noplaxochia

Numpy multilayer perceptron from scratch

With backpropagation

★ Oct 14, 2024



In The Deep Hub by Jorgecardete

Convolutional Neural Networks: A Comprehensive Guide

Exploring the power of CNNs in image analysis

Feb 7, 2024




2.8K



38

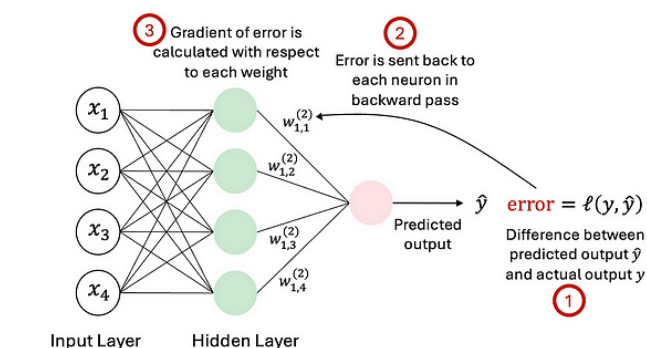


 D.H. Jang

Interpreting Support Vector Machine Coefficients: A...

In the rapidly advancing landscape of artificial intelligence (AI) and machine...

★ Nov 3, 2024



LM Po

Backpropagation: The Backbone of Neural Network Training

Backpropagation, short for “backward propagation of errors,” is a fundamental...

★ Sep 14, 2024

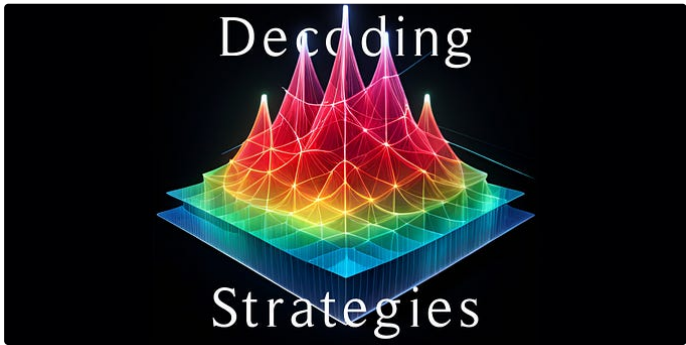
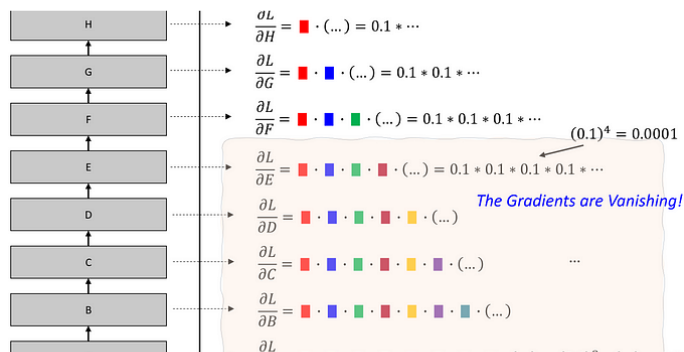


20



3





 Hugman Sangkeun Jung

 In TDS Archive by Maxime Labonne 

Understanding Backpropagation and Vanishing Gradients

A Comprehensive Guide to Backpropagation and the Vanishing Gradient Problem

★ Nov 14, 2024  16 

Decoding Strategies in Large Language Models

A Guide to Text Generation From Beam Search to Nucleus Sampling

★ Jun 4, 2023  663  4 

See more recommendations