100% accurate Sudoku solving with deep learning algorithm

Sébastien Guissart [1]

Abstract

Building a deep learning model capable of solving a Sudoku grid is a real challenge. Solving involves either backtracking or completing the grid step by step following logical reasoning (each completion must be done with certainty). The techniques used are: the use of Sudoku grid symmetries to drastically limit the number of model parameters, training on all grid resolution steps with continuous adaptation of the thresholds applied to the model outputs leading to the prediction, and a trial-and-error algorithm that allows the model to formulate assumptions and test them when the step is too complex. The model we have built solves 100% of the grids available to us. To our knowledge, it is the first to achieve this level of performance.

1. Introduction

AI and particularly deep learning have made significant advances in recent decades on games such as chess, go or the atari suite (Schrittwieser et al., 2020). However, one popular game, sudoku, has so far failed to find a deep learning algorithm giving solutions as accurate as human ones (human accuracy being 100%). There are several reasons for this: firstly, the challenge is less rewarding, given that a multitude of algorithms can be used to find an exact solution. This hasn't discouraged everyone, however, with a number of articles available on the subject (Park, 2018; Palm et al., 2017; Wang et al., 2019). The main reason, in our opinion, is the poor treatment of system symmetries in solution attempts. We also believe that solving such a problem of this type requires a step-by-step approach, where each step must provide a precise improvement, which makes standard deep learning algorithms unsuitable.

The code of the paper is available there (Guissart, 2024)

[1]Citizen Scientist. Correspondence to: Sébastien Guissart <sebastien.guissart@gmail.com>.

2. Take advantage of sudoku symmetry

First we need to define the representation of the sudoku grid. The trivial choice of the matrix representation in which it is displayed is not the right one: the digits of the cells appear as quantities. A better option is a one-hot encoding of these digits. Here we also choose to duplicate the dimension of the digits into two channels, one for the presence and one for the absence of the digit. A sudoku grid is therefore represented by a (9,9,9,2) tensor shape for the column, row, digit and absence/presence respectively. For each digit-cell pair (DCP) we therefore have a vector of dimension 2 representing the DCP; if its value is [0,1] we know that the digit is present in that cell, if its value is [1,0] we know that it is absent and if its value is [0,0] we don't know whether it is present or absent. The aim of the deep learning algorithm we'll be implementing is to find a value for all the empty DCPs.

A Sudoku grid has several symmetries. Many different permutations can be applied to a grid while maintaining the validity of the grid.

Permutations are: digit permutation, inside block row and column permutation, column and row block permutation.

For each DCP, we can construct a set of linear combinations that are invariant under these permutations. This set of linear combinations is used to construct a feature vector. Indeed, if we expect the same result for a DCP regardless of the permutation performed, then the feature vector should also be invariant under permutation. This concept appears in many other deep learning models, such as convolutional neural networks or transformers (which are based on invariant translation symmetry), *i.e.* a convolutional kernel will give the same output on the same image patch, whatever the patch localisation.

We combine these Sudoku group invariant linear combinations as a symmetry feature engineering tensor (SFET) Figure1.

The dot product of the SFET and the Sudoku grid returns a feature tensor of the shape $(9, 9, 9, f_{dim})$, where the first 3 dimensions are the column, row, digit dimension and the last one is the feature vector dimension.

The choice of the linear combination is quite arbitrary, many

different choices exist and we choose a set that follows some intuition. We try to keep it simple by using DCP masks where each value of the mask has the same position (permutation wise).

This symmetry treatment drastically reduces the number of parameters of the deep learning model. without it, the first layer should naturally be a dense layer with $9 \times 9 \times 9$ dimensions in input and output, leading to a $\sim 1M$ parameter dummy model.

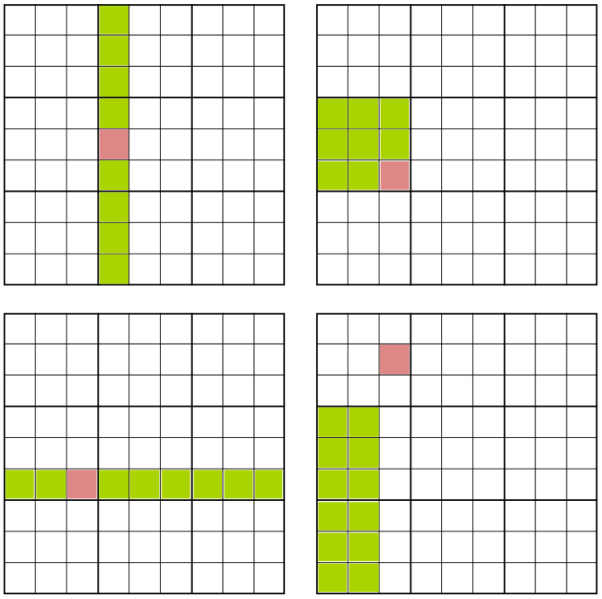

Figure 1. Example of linear combination based on the feature vector. The red cell corresponds to the DCP target cell, the green cells correspond to the summed DCP. Each spatial mask corresponds to 2 linear combinations, one with summed digits equal to the target digits and the other with summed digits different from the target digits.

3. The model architecture

We design a very simple and robust model with the symmetry tensor followed by 1 kernel 3D convolution layers. In this model architecture, after obtaining the symmetry feature tensor, each DCP is treated separately through its own channel. The model contains a very small number of parameters (512) but allows a robust training Figure2.

4. Model training

We train our model using the 3 million Sudoku puzzle grids dataset available on Kaggle (Radcliffe, 2020; Dachev, 2010). We use only 1280 grids for training and validation, as the use of system symmetries considerably reduces the parameter

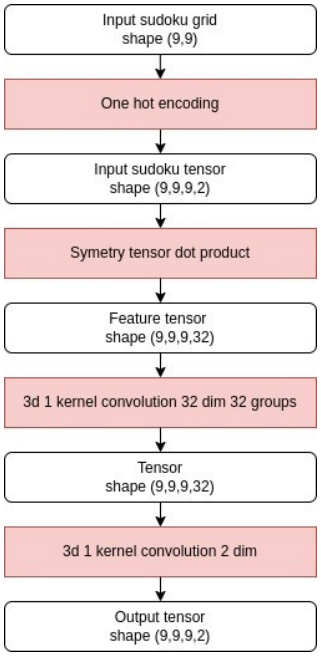

Figure 2. model architecture schema.

space to be explored.

We train the model via a standard gradient descent algorithm using a binary cross entropy loss (BCE). However, we aim to build the solution iteratively, at each step applying a threshold to our output and increasing our grid by a few DCP values.

The new grid is then used to train with the same strategy. We stop the iteration when there is no improvement over the old grid.

The thresholds are initialized first to the most permissive values and adjusted during training so that the new grids are error-free. The validation phases are used to reset the thresholds so that they remain as close as possible to the optimal solution. See Algorithm1.

This model training description allows us to solve 579/ 1280. We used 2 strategy to improve the number of solved grid and reach 100% accuracy.

5. Model boosting strategy

In the training presented above, all the DCPs are trained to the same level, yet for some the model can find an exact solution and for others it can only give a probability. We assume that model training is degraded by this fact. We then seek to vary the training sample. To do this, we take a list of models of the same architecture, the first of which is trained in the standard way, the second is trained on the grids not

Algorithm 1 Threshold adaptation

$margin \leftarrow 0.1$
function ADAPT_THRESHOLD(th, $batch$)
  $X, y \leftarrow batch$
  $output \leftarrow model.forward(X)$
  $pred \leftarrow output > th$
  if $any(pred = 1)$ & $(y = 0)$ **then**
    $th \leftarrow max(output[y = 0]) + margin$
  end if
end function
$LV \leftarrow -10$
Initialize $th \leftarrow LV$
for every training and validation epoch **do**
  train epoch loop
  for each batch **do**
    train batch
    ADAPT_THRESHOLD(th, $batch$)
  end for
  Initialize $th_c \leftarrow LV$
  validation epoch loop
  for each batch **do**
    validate batch
    ADAPT_THRESHOLD(th_c, $batch$)
  end for
  $th \leftarrow th_c$
end for

Algorithm 2 Trial error algorithm

function PREDICT WITH TRIAL ERROR(x, $model$)
  $x_{new} = model.predict(x)$
  if $x_{new} \neq x$ **then**
    return x_{new}
  end if
  $tried_{DCP} = []$
  while true do
    $DCP_{to\_try} =$ **GET DCP TRIAL**$(x_{new}, tried_{DCP})$
    $tried_{DCP}.append(DCP\_to\_try)$
    Initialize x_{pres}, x_{abs} **with** x
    $x_{abs}[DCP_{to\_try}] = [1, 0]$
    $x_{pres}[DCP_{to\_try}] = [0, 1]$
    for x_{supp}, x_o **in** $[(x_{pres}, x_{abs}), (x_{abs}, x_{pres})]$ **do**
      while true do
        $x_{supp} = model.predict(x_{supp})$
        if x_{supp} **not valid then**
          return x_o
        end if
        if x_{supp} **complete then**
          return x_{supp}
        end if
        if x_{supp} **not improving then**
          break while
        end if
      end while
    end for
  end while
end function

resolved by the first and so on. By varying the sampling of each model, they learn more easily from specific situations. This strategy improves the result to 597 solved grids over 1280 on the validation set.

6. Trial error Algorithm

The present model manages to solve simple Sudoku grids, but fails to predict more complex ones. Indeed, some advanced Sudoku grids involve the use of techniques such as 'forcing chain' (Day, 2000). This kind of technique cannot appear in the abstraction of our model. So we need to add a step when a Sudoku grid gets stuck at some point: we choose an unsolved DCP and try both grids (a grid with presence of digit assumption and a grid with absent digit assumption), if one of the solutions leads to a non valid grid we know the second option is the good one, if both grids get stuck at solving we try a new DCP. See Algorithm2.

This simple trial-and-error procedure is sufficient to solve all the Sudokus we have tried.

To secure the result, we include a model that proposes the best candidate to the trial-error system. This model has the same architecture and is trained to predict whether a candidate DCP will lead to one invalid grid (success) or two stuck grids (failure).

7. Results

It is difficult to be exhaustive about the precision of our best model, but we have tested it on :

- 100k randomly selected grids from the 3 million grid dataset.

- 4k of the top 15k grids according to the dataset difficulty rating.

- some evil grids found on the internet.

- a grid with only 17 digits (the minimum number of digits a sudoku grid can have in order to have a unique solution).

For each grid tested, the algorithm gave us an exact solution except one: the Arto Inkala grid (Inkala, 2012). For this grid the trial error model at some step tries every available DCP but stops improving. We built a backtracking algorithm to solve this grid using the trial error model and it worked, showing the robustness of the algorithm. In the near future we may write another article describing this algorithm.

8. Conclusion

We have built a deep learning model capable of solving all sudoku grids, up to the hardest. This model is based on the symmetries inherent in sudoku, which allow us to strongly restrict the number of model parameters. Finally, as the first model was robust but did not solve all grids, a trial error mechanism was implemented. This approach should be applicable to other NP problems of the same type.

References

Dachev, B. Sudoku generator and solver for node.js. https://github.com/dachev/sudoku, 2010.

Day, S. O. T. Sudoku techniques: Forcing chains. https://www.sudokuoftheday.com/techniques/forcing-chains, 2000.

Guissart, S. Demo and code of current paper. https://huggingface.co/spaces/SebastienGuissart/deeplearning_sudoku_solver, 2024.

Inkala, A. Arto inkala sudoku grid. https://www.sudokuwiki.org/sudoku.htm?bd=800000000003600000070090200050007000000045700000100030001000068008500010090000400, 2012.

Palm, R. B., Paquet, U., and Winther, O. Recurrent relational networks for complex relational reasoning. *CoRR*, abs/1711.08028, 2017. URL http://arxiv.org/abs/1711.08028.

Park, K. Can convolutional neural networks crack sudoku puzzles? https://github.com/Kyubyong/sudoku, 2018.

Radcliffe, D. 3 million sudoku puzzles with ratings. https://www.kaggle.com/datasets/radcliffe/3-million-sudoku-puzzles-with-ratings, 2020.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-03051-4. URL http://dx.doi.org/10.1038/s41586-020-03051-4.

Wang, P., Donti, P. L., Wilder, B., and Kolter, J. Z. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *CoRR*, abs/1905.12149, 2019. URL http://arxiv.org/abs/1905.12149.