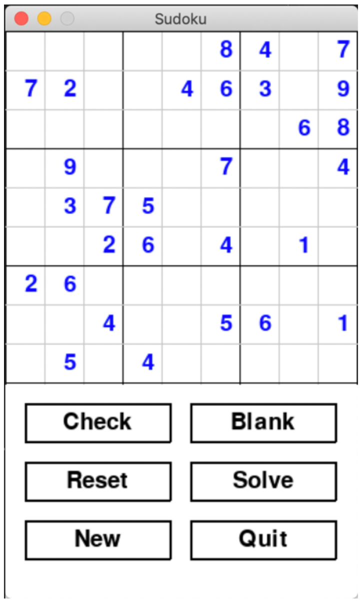
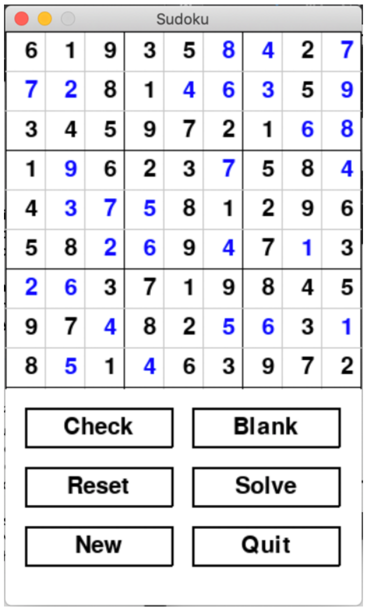
# A Sudoku Solver Combining a Convolutional Neural Network with Backtracking

Aleksey Kocherzhenko, 2020



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



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

## Background

A sudoku puzzle is a 9-by-9 square grid that is subdivided into nine 3-by-3 blocks and is partially filled with numbers. The objective is to fill in the missing numbers in such a way that each row, each column, and each block in the grid, contains all digits from 1 to 9. A well-posed sudoku puzzle has a single solution. There are 5,472,730,538 essentially different Sudoku grids,[1] from which around $6.67 \times 10^{21}$ unique puzzles can be produced using symmetry transformations.[2] The smallest possible number of initially occupied cells in a sudoku puzzle with a unique solution is 17,[3] and at least 8 different digits must appear in these cells.[4]

The general problem of solving Sudoku puzzles on $n^2$-by-$n^2$ grids of $n$-by-$n$ blocks is known to be NP-complete.[5] However, due to the small size of classic 9-by-9 sudoku puzzles, they can be solved quickly on modern computers. A number of deterministic algorithms for constructing and solving such puzzles have been suggested that use constraint propagation or backtracking search.[4]

A sudoku puzzle can be trivially mapped onto a 9-by-9 grayscale image, where each cell is treated as a pixel and its value is its relative intensity. Convolutional neural networks (CNNs) are commonly used for pattern recognition in images. It is, therefore, reasonable to check how well they might perform on solving sudoku puzzles.

*9-by-9 sudoku puzzle (top) and its solution (bottom).*

[1] E. Russell, F. Jarvis. *There are 5472730538 essentially different Sudoku grids... and the Sudoku symmetry group.* http://www.afjarvis.staff.shef.ac.uk/sudoku/sudgroup.html, **2005.**
[2] B. Felgenhauer, F. Jarvis. *Enumerating possible Sudoku grids*. http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf, **2005**.
[3] G. McGuire, B. Tugemann, G. Civario. *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem*. arXiv:1201.0749, **2013**.
[4] P. Norvig. Solving Every Sudoku Puzzle. http://norvig.com/sudoku.html,
[5] T. Yato. *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*. Master's thesis, University of Tokyo, **2003**.

## Data Generation

*This section describes functions in* `sudoku_generator.py` *and* `solve_sudoku.py`*.*

All programs in the current package can use an $n^2$-by-$n^2$ grid with arbitrary integer $n$, but the data was generated and the neural network was trained and tested for $n = 3$ that corresponds to the standard 9-by-9 sudoku grid.

The training data set was generated by running `sudoku_generator.py` that uses a backtracking algorithm that is implemented in the function `fillSolveGrid`. This function can be found in the module `solve_sudoku.py`. It takes 4 parameters: `n` (an integer), `grid` (a list of $n^2$ lists of integers with length $n^2$), `counter` (an integer that counts the number of possible solutions for a partially filled grid – only puzzles with a single solution are valid), and `opt` that can be either `"fill"` or `"solve"`, depending on whether the function is used to create a valid sudoku solution or to solve an existing grid. The function returns two outputs: a Boolean value that is `True` if the grid is completely filled or `False` if there are still empty cells, and the `counter`. The function is called recursively until the grid is completely full.

The check whether this condition is satisfied is implemented in the function `checkGrid` that takes two parameters, `n` and `grid`, and returns a Boolean value that is `True` if the grid is full or `False` otherwise. This function can also be found in the module `solve_sudoku.py`.

Once a valid sudoku puzzle and its solutions have been generated, the function `writeGrid2File` that can be found in the file `sudoku_generator.py` is used to save them to separate files. This function takes 3 parameters: `n`, `grid`, and `filename` (the target filename). All cell values are written as a single line, by row (top left to bottom right), separated by spaces. Zeros in the file with puzzles represent empty cells.

The dataset that was used to train the model consists of **N** puzzles and their solutions that can be found in the files `training_puzzles.txt` and `training_solutions.txt`, respectively. The data set that was used to test the model can be found in the files `test_puzzles.txt` and `test_solutions.txt`, respectively

## Neural Network Architecture

*This section describes functions in* `sudoku_model.py`*.*

The current project uses a simple CNN that consists of three convolutional layers followed by a single fully connected layer (I have tried several different architectures, this one worked best.).[6] The first two convolutional layers each use 64 $n$-by-$n$ filters with same padding and a ReLu activation function. The third convolutional layer uses 128 1-by-1 filters and a ReLu activation

[6] S. Verma. *Solving Sudoku with Convolution Neural Network: Keras*. https://towardsdatascience.com/solving-sudoku-with-convolution-neural-network-keras-655ba4be3b11, **2019**.

function. Batch normalization is applied after each of the first two convolutional layers. No pooling layers are necessary, because of the small size of the “image”. The fully connected layer consists of $n^6$ units, and its output is reshaped to an $n^4$-by-$n^2$ matrix before applying a softmax classifier that returns the probabilities of each of the possible $n^2$ values for each of the $n^4$ numbers on the grid. The model is defined in the `define_model` function in the `sudoku_model.py` module. This function takes a single integer parameter, *n*, and returns the Keras Sequential object `model`.

A requirement for a sudoku solver is that *every* number on the grid must be predicted correctly. However, simply selecting the values with highest probability for each cell from the output of the softmax classifier results in a low accuracy of solutions. A much better result can be obtained by selecting the value for *only one* cell that has the highest probability, adding it to the grid, making a new prediction for the input grid with one extra number filled in, and repeating the procedure until the grid is filled completely.[6] The function `predict_number` in the `sudoku_model.py` module allows to implement one step of this procedure. It takes 3 parameters: `X` (a partially filled sudoku grid represented by a numpy array with zeros representing empty cells), `model` (the output of the `define_model` function), and `n2` (an integer equal to $n^2$, the grid dimension). It then returns 3 outputs: `y_predict` (the value for the next cell to be filled), `index` (the number of the next cell to be filled, with cells numbered by row from left to right), and `flag` (a Boolean value that is `True` if there are still empty cells on a grid and `False` if the grid is completely full).

## Training Procedure

*This section discusses functions in* `import_data.py` *and* `sudoku_trainer.py`.

The training and test data sets are retrieved from the files `training_puzzles.txt`, `training_solutions.txt`, `test_puzzles.txt`, and `test_solutions.txt` using the functions in the module `import_data.py`. The function `import_puzzles` takes 2 parameters: `n` and `filename_x` (a string specifying the path to the data file). It returns 2 outputs: a 4-dimensional numpy array with dimensions (number of puzzles, $n^2$, $n^2$, 1) that correspond to the puzzle number, row on the grid, column on the grid, and cell value, as well as an integer equal to the total number of puzzles. The function `import_solutions` takes 3 parameters: `n`, `num_examples` (the total number of puzzles), and `filename_y` (a string specifying the path to the data file). It returns a 3-dimensional numpy array with dimensions (number of puzzles, $n^4$, 1) that correspond to the puzzle number, cell number on the grid (numbered by row, from top left to bottom right), and cell value.

The program `sudoku_trainer.py` trains the neural network described in the previous section by minimizing the sparse categorical cross-entropy loss using the Adam optimizer[7] for 2 epochs. Setting the initial learning rate to $10^{-3}$ for the first epoch and to $10^{-4}$ for the second epoch yielded the best results. The batch size of 32 resulted in the best compromise between training time and accuracy. The trained weights were saved to the file `sudoku_weights.h5`.

[7] D.P. Kingma, J. Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980, **2014**.

The training set consisted of 400k sudoku puzzles (increasing the dataset size beyond about 300k does not significantly reduce the loss or improve the model's accuracy for a test set). The maximum model accuracy achieved for the test set of 1k puzzles in `test_puzzles.txt` was around 70%.

Interestingly, the model accuracy with the same weights for a test set of 1k puzzles drawn from the dataset of 1M sudoku games at `https://www.kaggle.com/bryanpark/sudoku` was 100%, suggesting that the puzzles generated using the program `sudoku_generator.py` are more challenging for the neural network to solve, whether due to the greater complexity or greater variety of puzzles. Because the times it took for the backtracking algorithm to solve the two sets of 1k puzzles also differed significantly (14 minutes 55 seconds for the custom test set vs. 28 seconds for the Kaggle test set), it is likely that the puzzles in the custom data set are actually more complex, not just more varied. Trying several alternative neural network architectures did not result in improved accuracy for the custom test set.

## Combining the Neural Network with Backtracking for Optimal Performance

*This section discusses the functions in* `check_sudoku.py` *and* `sudoku_cracker.py`.

Once a neural network has been trained, it can be used to solve any sudoku puzzle using a fixed number of operations, known in advance, and at times do it faster than backtracking. However, unlike backtracking, no neural network is guaranteed to find the right solution 100% of the time. Fortunately, checking whether a sudoku solution is correct simply requires making certain that all digits from 1 to 9 are found in each row, column, and 3-by-3 block. If a neural network provides a speedup over backtracking for most sudoku puzzles and solves most sudoku puzzles correctly, it may be practical to first try solving each puzzle using the neural network, then to check the solution, then, if the solution is wrong, to solve the puzzle using backtracking.

The program `sudoku_cracker.py` realizes this approach. It loads a set of sudoku puzzles from a text file with the same format as the training and test sets, as well as the sequential model object described in the Neural Network Architecture section and the trained weights saved to the file `sudoku_weights.h5`. It then repeatedly uses the function `predict_number`, described in the Neural Network Architecture section, to solve each sudoku puzzle by filling it in number-by-number. The function `check_puzzle` that is part of the module `check_sudoku.py` is then used to check whether the solution found by the neural network is correct. This function takes an $n^2$-by-$n^2$ numpy array as an argument and returns a Boolean value: `True` if the solution is correct, and `False` if it is wrong. (The function `check_puzzle` uses two service functions, `row_correct` and `get_blocks` that can also be found in the file `check_sudoku.py`.) If the solution found by the neural network is incorrect, the puzzle is then solved using backtracking, as implemented in the function `fillSolveGrid` that is described in the Data Generation section.

The solution of 1k puzzles in the custom data set using backtracking took 14 minutes 55 seconds. Using the neural network with backtracking as backup, this time was reduced to 11 minutes 26 seconds (2 minutes 35 seconds for the 65% of puzzles that the neural network solved correctly and 8 minutes 51 seconds for the 35% of puzzles for which backtracking had to be applied).

## The Graphical User Interface

*This section discusses the functions in* `sudoku_game.py`.

The simple graphical user interface for this package, implemented using `pygame`, can be seen on the first page. It consists of a sudoku grid, with clues (already filled in numbers) that are shown in blue, with 6 clickable buttons below. The user may click on any empty cell in the sudoku grid and type in a number from 1 to 9 to fill it; these numbers are shown in black. The value of any user-filled cell may be cleared by clicking the cell, then pressing the backspace key ("Delete" on a Mac). A value may also be corrected by clicking the cell and typing in another number. All user-entered values may be cleared by clicking the "Reset" button below the sudoku grid.

Once a puzzle has been filled, the user can click and hold the "Check" button to check whether the solution is correct.

At any point during the game, the user may also click the "Solve" button to automatically complete the puzzle. The solution is attempted using a neural network first, then backtracking if the solution predicted by the neural network is incorrect (this procedure is described in the previous section). *If the user has already entered some values, these values are taken into account in the solution*. If some of these values are incorrect, the attempt at a solution will fail. A time limit of 2 minutes is set to search for a solution; typically, if a solution is not found within this timeframe, it means that the sudoku puzzle is invalid.

A new sudoku puzzle can be loaded by clicking the "New" button. All puzzles are loaded from a file and are guaranteed to have exactly one solution.

Clicking the "Blank" button displays an empty sudoku grid. The user may use this grid to enter clues, e.g., from a sudoku puzzle book, then use the "Solve" button to solve the puzzle. A solution should be found if the puzzle has at least one, however, no check as to whether the puzzle has only a single solution is performed.

Clicking the "Quit" button will close the game window.

## System Configuration

The programs in this package were tested on a 13-inch 2017 MacBook Pro with a 2.3 GHz Intel Core i5 processor and 8 GB of RAM, running python 3.7.6, Tensorflow 2.2.0, Keras 2.3.1 (using Tensorflow backend), and pygame 1.9.6.