Solving nonograms using neural networks

Balázs Patrik Csomor
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
csomorbp@gmail.com

Márton Bendegúz Bendicsek
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
bendicsekb@gmail.com

***Abstract*—A nonogram (*also known as Paint by Numbers, Picross, Griddlers, Pic-a-Pix*) is a Japanese logic puzzle game, where the player needs to color cells of a grid based on numbers describing the number of colored cells in rows and columns. While this puzzle may seem like an easy task at first sight, it is a great tool to showcase different theories in mathematics and computer science. During our work we aimed to create a neural network that can solve most nonograms of a fixed size. This included generating puzzles, which have unique solutions and experimenting with densely connected and convolutional neural networks. In the process we encountered multiple pitfalls of machine learning, which we also describe in this document. The result of our research is a model, which can predict the correct solutions for 6 by 6 sized nonograms in almost 70% of the cases. We also deployed a web-based application, where users can interact with the machine learning model, available at https://nonogram-solver.web.app/**

Keywords—Nonograms, convolutional neural network

I. NONOGRAM PUZZLES

Nonograms first appeared at the end of the 1980s in Japan. These puzzles consist of a rectangular grid with empty cells and sequences of numbers describing the rows and columns of the grid. A sequence of numbers for a line shows the length of blocks of cells which need to be colored in that line.

For instance, the sequence [2 1] at the first row on Fig. 1. means that there must be 2 blocks of colored (black) cells, one with the length of 2 and one with the length of 1. These blocks of cells must be separated with at least one empty (white) cell. The order is important as well, the block with length of 2 must be the first colored piece when looking at the puzzle from left to right. A nonogram puzzle is solved, when the colored cells fulfill each condition set by the row and column descriptors. The solution of this nonogram can be seen at Fig. 2.

This logic puzzle gained popularity in both printed and digital forms. Most of the time, players who successfully solve a nonogram get an image as a reward, consisting of the black and white cells of the puzzle. In our example, the black cells spell the word *AI*.

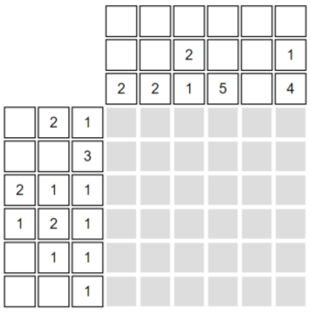


Fig. 1. A nonogram puzzle in its original state

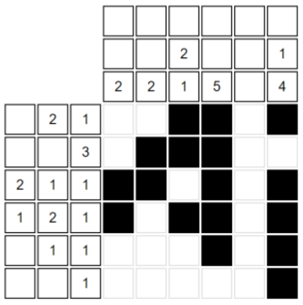


Fig. 2. The nonogram puzzle from Fig. 1. in its solved state

II. SOLVING NONOGRAMS

The problem of solving nonogram puzzles is NP-complete [1]. This fact may not be a surprise if we consider that solving these puzzles is a constraint satisfaction problem (CSP), where the descriptors and of rows and columns are constraints and the cells must be in a specific state to solve the puzzle [2] [3]. Other logic puzzles like Sudoku also belong to this category of problems. One difficulty of CSP-s is the phenomenon of combinatorial explosion. This is also the case with nonograms, since an 8 by 8 sized grid for instance has $2^{(8*8)}$ possible colorings. Thus, an exhaustive search is out of question.

As it is described in the paper "An Efficient Approach to Solving Nonograms" [1], the current best algorithm for solving nonograms has a time complexity of $O(k * I)$, where the grid has a size of $I * I$ and k is the average number of integers in one constraint. The algorithm is not related to our topic of machine learning and we did not aim to outperform it, however it is the state-of-the-art solution for solving these kinds of puzzles.

Regarding the practical side of the problem, human players use a technique, which is easy to implement and can solve most puzzles with acceptable runtimes. This algorithm solves puzzles line by line. Looking at a line, we need to consider the descriptors of that row or column and the cells, which may already be given a color. As a next step, all solutions for that line need to be generated, which fulfil the constraints of the descriptors and the previously colored cells. At last, we can color the cells of the line, where all the solutions have a common color. The algorithm continues until the full puzzle is solved. As most nonograms published as logic puzzles in newspapers or online have a simple unique solution, players are advised to go along this logic.

Considering previous works for solving nonograms with the help of machine learning tools, the paper "Solving nonogram puzzles by reinforcement learning" [2] describes an interesting approach. The authors use the simple algorithm described above for the base of their solution. However, they state that the order of lines in which the puzzle is solved heavily affects run times. Due to combinatorial explosion, finding the optimal order of lines to solve is impossible with traditional algorithms like breadth-first search. The authors propose a reinforcement learning based model, where states are the cells of lines, actions are choosing the next line to solve, and rewards are the ratio of cells colored in the given line after an action. Their model showed great results in predicting the optimal order of lines, however the solution was not entirely done with neural networks.

Considering the existing methods for solving nonograms, our goal was not to overcome those algorithms in terms of runtime or accuracy. We rather wanted to find out if neural networks can be applied to solving logic puzzles and constraint satisfaction problems in general.

III. Network Architecture

Our neural network for solving 6 by 6 nonograms consists of a single 1-dimensional convolutional layer and three densely connected layers. The convolutional layer aims to help the network understand how descriptors of a row or a column need to be looked at as one unit [6] [7].

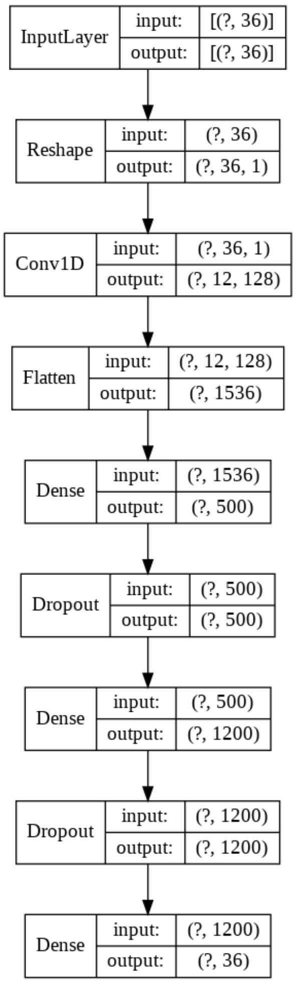


Fig. 3. Visualization of our neural network model

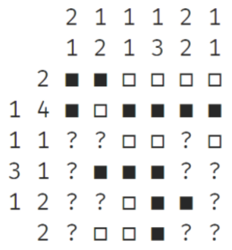
Since the inputs of the neural network are the row and column descriptors, the input shape is a vector with 36 elements. An explanation of this is that in a line with the length of 6 cells, there can only be 3 colored blocks of pieces at maximum.

The output shape of our network is a vector with 36 elements as well since the prediction describes the individual cells of the solved nonogram.

IV. Generating Nonograms

The first step for creating a neural network which can solve nonograms was generation of training data. We considered web scraping from puzzle sites as an option for data collection, but we quickly gave up on the idea, since by artificially generating the data we get way more control over it.

The task of data generation started by implementing a nonogram class in Python. We tried to make the implementation as generalized as possible, so we could generate both square shaped and rectangular nonograms. The *Nonogram* class stores cells and the values of each cell, which can be -1 for not yet colored, 0 for white and 1 for black cells. A variety of functions were added to be able to calculate descriptors, check validity of a solution or to show nonograms in a visual way on the command line.



```
    2 1 1 1 2 1
    1 2 1 3 2 1
  2 ■ ■ □ □ □ □
1 4 ■ □ ■ ■ ■ ■
1 1 ? ? □ □ ? □
3 1 ? ■ ■ ■ ? ?
1 2 ? ? □ ■ ■ ?
  2 ? □ □ ■ ? ?
```

Fig. 4. Output of the print function of the nonogram class

Generating random nonograms with the implemented class is simple. We just need to randomly choose the colored cells and calculate the descriptors. However, there are more aspects to consider. As we aim to replicate real puzzles, almost entirely black or entirely white nonograms are not helpful. Thus, we chose to generate puzzles, where the number of colored cells follows a normal distribution, with half the number of cells being the mean value.

Another issue is the uniqueness of nonograms. There are descriptors, which correspond to two or more different solutions. At first, we included these non-unique nonograms in the training data, but they led to inaccurate predictions. Therefore, we decided to generate unique nonograms, which are defined as puzzles which can be solved with the simple algorithm described in the “Solving Nonograms” chapter [4]. This heavily influenced the generation time, generating 100.000 nonograms of 6 by 6 size took over 8 minutes on a Google Colab machine.

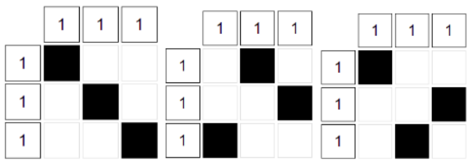


Fig. 5. Nonogram with multiple solutions

The generated nonograms were reshaped from their inner data structure to input and output vectors with the length of 36 in the case of 6 by 6 puzzles. However, our generation functions can be parametrized to generate nonograms of any shape.

V. Training

During training, we started with a relatively simple model, then tried to add more and more along the way. Our loss function was mean absolute error (MAE), which is equal to the ratio of incorrectly predicted cells in a nonogram.

We used early stopping, so we could quickly determine if a model was not learning as expected. With this method we could rapidly iterate through different configurations. Checkpoints were also saved during the training process; thus, we were able to always keep the best performing model and use it as a baseline for further training.

One problem we encountered was overfitting, which we identified by the ratio between the training and the validation losses. Dealing with this problem, dropout layers have been introduced after the fully connected layers consisting of the greatest number of nodes. This, however, soon proved not to be enough. Besides, generating quadruple amount of training data resulted in a ~10% improvement in model performance. Additionally, the original model has been pruned, as we found, that deeper architectures are prone to overfitting.

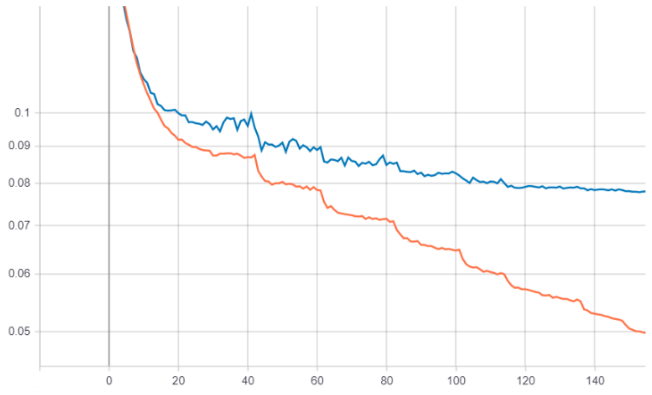


Fig. 6. Train (*orange*) and validation (*blue*) losses showing overfitting

We also noticed, that when the loss plateaued, there was still room for learning, so we used an adaptive learning rate, which made sure that the model could find important minima. This involved reducing learning rate if the validation loss has not decreased in the last 10 epochs. The optimal reduction factor turned out to be 0.75. With this method we were able to make the model learn for longer, thus further reducing loss.

VI. Hyperparameter Optimalization

To get an overview about the ideal model, we utilized Keras Tuner, which is a hyperparameter optimalization package for Keras models [5]. To use it, one should simply define a function that returns a compiled model. The only argument of this function is an optimizer instance; inside the function the parameters can be sampled from a range given by the optimizer. Parameters can be integers or floating-point numbers of a certain range, Booleans, and choices of predefined values.

We used random space search hyperparameter optimization to find optimal network parameters. Random space search essentially tests every possible network with given parameters in a random permutation. The set of these parameters is called hyperparameter space. When too many values are parametrized, the hyperparameter space becomes unreasonably big, which results in high computation times. Because of this, the search is restricted to the following parameters for each layer type:

1) *Fully connected: Number of nodes per layer*
2) *Convolutional: Pool size; Number of layers*

VII. Evaluation

As mentioned in the previous chapter, the loss function shows us how the ratio of incorrectly predicted cells compared to the total number of cells. While this measurement works for training, what we aim for is a model that predicts most nonograms without mistakes, not one that misses few cells in all of them.

During the evaluation process, we predicted outputs from a test dataset and counted the number of missed cells for each nonogram. We displayed these values on a histogram and calculated the ratio of mistake free predictions. As expected, as the validation loss went down, the ratio of correctly solved nonograms rose.

With all the already described methods for optimizing the training process and tuning the parameters, we could create a network, which can predict solutions for unique 6 by 6 sized nonograms almost 70% of the time. We can also state that most of the missed nonograms include only 1 or 2 incorrectly labeled cells.

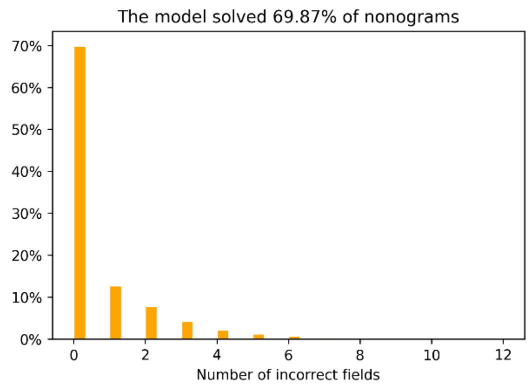


Fig. 7. Prediction results for solving 6 by 6 nonograms

We learned an interesting lesson at the start of our work. For better chances at succeeding, we started building a model for predicting cells of 4 by 4 nonograms. It did not take much time to get to a neural network, which could correctly predict the solutions of these puzzles over 99% of the time.

We soon realized that this may be an example of a neural network memorizing each input-output pair. This could happen due to the fact, that there are only $2^{(4*4)} = 65536$ possible nonograms of this size and at the same time we used a network with a higher number of parameters and ~70.000 nonograms as training data.

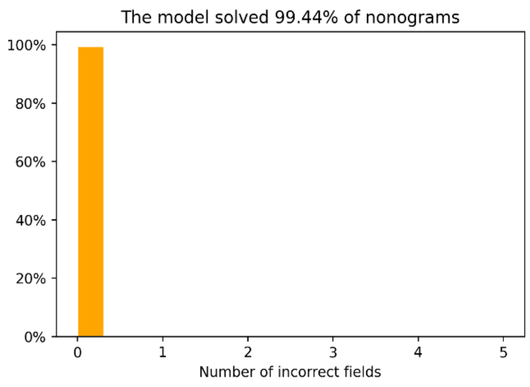


Fig. 8. Prediction results for solving 4 by 4 nonograms

For further evaluation we used ROC analysis, which showed, that even with low thresholds, false positive labels do occur. Starting from this, we checked whether the model was not able to decide between the two classes, or if it learned the wrong patterns. Unfortunately, the latter has been proven to be true, as the absolute difference between the predictions and the actual labels are mostly over 0.99. This means, that the model is perhaps overly confident in the result.

VIII. DEPLOYMENT

To be able to present our results in an easily accessible way, we decided to create a web application, which uses our model to provide a solver for 6 by 6 nonograms.

The prediction runs in the user's browser, using Tensorflow.js, a JavaScript library for creating and interacting machine learning models [10]. Since we used Keras based on Tensorflow to build our neural network, we could effortlessly save our model to a format compatible with the JavaScript library. This solution also has the benefit, that we can simply serve our web application through a CDN, without significant server costs.

The user can enter row and column descriptors for nonograms through the web interface and the website displays the predicted solution from the model.

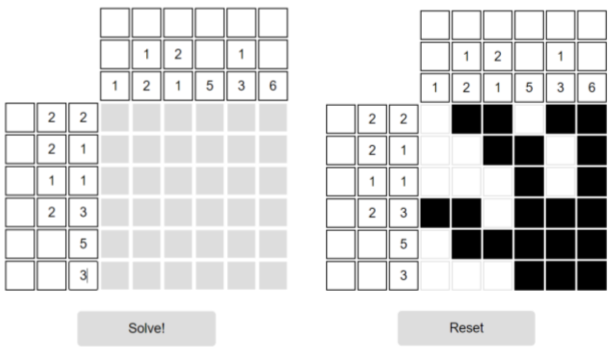


Fig. 9. Our web application showing solving of a puzzle

IX. CONCLUSION

In this document, we presented a neural network-based architecture for solving 6 by 6 nonograms, which proved to be acceptable in accuracy. We also made a web-based solver using our model. During this engineering process we gained valuable experience in solving machine learning related problems.

The resulting codebase is general enough, so that it can easily be adjusted to deal with differently sized or even non-quadratic nonograms. Future improvements could include generalizing the architecture or creating different models for different nonograms.

REFERENCES

[1] N. a. N. T. Ueda, „NP-completeness results for NONOGRAM via Parsimonious Reductions," 1996.

[2] K. J. a. K. W. A. Batenburg, „Solving Nonograms by combining relaxations," *Pattern Recognition,* volume 42, pp. 1672-1683, 2009.

[3] S. a. N. P. Russell, Artificial intelligence: a modern approach, 2002.

[4] I.-C. W. a. D. S. a. L. C. a. K. C. a. C. K. a. H. K. a. H. Lin, „An Efficient Approach to Solving Nonograms," *IEEE Transactions on Computational Intelligence and AI in Games,* pp. 251-264, 2013.

[5] F. a. C. D. a. S. T. R. Dandurand, „Solving nonogram puzzles by reinforcement learning," *Proceedings of the Annual Meeting of the Cognitive Science Society,* 2012.

[6] S. a. A. O. a. A. O. a. I. T. a. G. M. a. I. D. J. Kiranyaz, „1D convolutional neural networks and applications: A survey," *arXiv preprint arXiv:1905.03554,* 2019.

[7] S. a. C. P. a. K. A. L. Abdoli, „End-to-end environmental sound classification using a 1D convolutional neural network," *Expert Systems with Applications,* volume 136, pp. 252-263.

[8] E. G. a. S.-S. S. a. L.-M. J. M. a. P.-B. A. M. a. P.-F. J. A. Ortiz-Garcia, „Automated generation and visualization of picture-logic puzzles," *Computers & Graphics,* pp. 750-760, 2007.

[9] T. a. B. E. a. L. J. a. C. F. a. J. H. a. I. L. e. a. O'Malley, „Keras Tuner," 2019.

[10] D. a. T. N. a. A. Y. a. Y. A. a. K. N. a. Y. P. e. a. Smilkov, „Tensorflow.js: Machine learning for the web and beyond," *arXiv preprint arXiv:1901.05350,* 2019.