**Training A Minesweeper Neural Network**

Introduction

Minesweeper is a puzzle-solving game that was introduced in the 1980s. The aim of minesweeper is to reveal all cells on a grid without revealing a bomb. If a bomb is revealed the game is lost. When a cell is revealed, if it is not a bomb, will display a number that tells the user how many bombs it are around. Minesweeper contains many combinations and strategies that are not immediately obvious. However, it also contains some guesswork and probability. This project aimed to create an AI-driven neural network that would provide aid to the user to help them complete a game of minesweeper. Two goals for this AI were to have a generalised understanding of how to play minesweeper (total mean squared error < 0.1) and to perform at a level that is comparable, if not better, than humans. The AI was considered above human standard if it could predict the outcome of a cell more accurately and consistently than that of a human payer.

Background

Several different methods have been explored in the past to solve games using AI. Taweh Beysolow II implemented a general reinforcement learning approach using q-learning to teach AI how to play games in different environments [1]. Unfortunately, these environments are extremely complex to set up in a way that a neural network can recognise and would be an overengineered approach for a simple game of minesweeper. Andrew Adamatsky implemented cellular automation to play minesweeper which identifies mines on an grid in time [2]. Luis Gardea created a minesweeper trainer that uses supervised learning and q-learning, comparing the results [3]. A multi-layered perceptron was chosen due to its simplicity and reproducibility compared to other AI methods.

AI Method and Tools

The neural network was implemented using the Java framework and Neuroph Studio GUI. Minesweeper was created in the java framework so that lower-level control of the program could be accessed. The program contains a limited feature set compared to the normal minesweeper to reduce complexity and usability for the user. The difference between this version of minesweeper and any other is that it contains an option to encode a cell when the middle mouse button is clicked.

The neural network implemented for this AI is a multilayered perceptron comprised of 288 input neurons, 144 hidden neurons and 1 output neuron. This neural network requires a hidden layer as the data used in minesweeper is not linearly separable, all tiles encoded for input are not known at the time. Another reason is that minesweeper requires the XOR function to predict bomb positions (if there is a flag or a number but not both). When the program probes a cell, it encodes all cells in a 5x5 grid excluding the cell probed. Each cell can be one of twelve options. The value of a cell would be 0-8 if it contains a number, 9 if it had a flag, 10 if unopened or 11 if it is outside the game grid size. Each cell value was considered a different input into the neural network. Since the neural network can only have binary inputs (1 or 0) the values were converted into an input vector where the value of the cell is represented by a one, placed in the [] position in the vector.

See **Appendix A**. The program checked a 5x5 grid around the probed cell encoded 24 cells (excluding the probed cell) resulting in 24\*12 inputs. See **Appendix B**. This output vector containing 288 inputs was exported to a CSV file. The neural network contained 1 output neuron which had an output of 1 (bomb) or 0 (no bomb). The encode function outputs a 288-integer long vector. The vector was inputted into a Neuroph Studio GUI multilayered perceptron with backpropagation to output a number between 0-1. The output is how likely the cell is to contain a bomb.

Data were cleaned using notepad++. Firstly, the data was sorted by lexicographically ascending order (generalised alphabetical order). This meant that duplicate lines of input were placed next to one another. Next, the find and replace tool was used with the regex expression “^(.\*?)$\s+?^(?=.\*^\1$)” which removes the current line if the next line is the same as itself.

Evaluation & Conclusion

The AI was evaluated using three key metrics: the number of iterations required to meet the maximum allowed error, the total mean squared error on the training set and the number of cells revealed before failing. The neural networks’ leaning rule, bias, number of hidden neurons, learning rate and maximum allowed error were manipulated to maximise the performance of these metrics.

Bias in a neural network allows it to shift the sigmoid of the activation function so that a more preferred output can be obtained. According to fig. 1, the non-biased network performed better as it had a lower total mean squared error. Unfortunately, Neuroph studio GUI does not show the bias it uses or an option to change it which for this reason a biased neural network will not be used.

To find the number of hidden neurons the pruning technique was implemented [4]. The network was tested with a high number of hidden neurons (288) and reduced until a significant change was detected. Fig. 2 shows that 144 neurons performed the best with a total mean squared error of 0.100886648. A high number of neurons did not perform well due to Widrow’s rule of thumb:

Where is the number of training examples, is the number of weights in the network and is the maximum allowed error. This rule states that as the number of neurons increases so will the amount of training data required to reach a generalised neural network. Therefore, if the neural network were to use 288 neurons it would require a lot more data to become generalised. In fig. 2 even though 50 neurons performed the quickest it did not have the lowest mean squared error when used with the test data.

The learning rate of a neural network is how much the weights will change through each iteration. It is defined in the following backpropagation learning rule:

Where is the iteration, is the weight, is the input, and is the error. If the learning rate is too high, it can learn too quickly and become unstable creating undesirable weight changes. If it is too low, it can take a long time to train or can get stuck. According to fig. 3 a learning rate of 0.1 performed the best however took 38% longer to compute than a learning rate of 0.2. As compute time was not an issue a learning rate of 0.1 was chosen.

Multiple training rules can be implemented in this neural network. This project focused on three: resilient propagation, backpropagation, and backpropagation with momentum. Resilient propagation is one of the more complex learning rules as it does not require user variables such as learning rate. It uses partial derivative magnitudes to adjust the weights automatically. This rule is generally quicker than backpropagation but can cause the opposite effect if the maximum error is too low. As seen in fig. 4 resilient propagation overtakes backpropagation at ~0.065 total network error but quickly slows down taking twice as long as backpropagation. Backpropagation is the simplest learning rule. The network generates an output, and if that output is not desired the weights are adjusted to reduce this error. It does this by starting at the output layer and propagating backwards to the input layer adjusting the weights during each layer. Backpropagation with momentum is like backpropagation except its values are not easily changed. If a network generates several desired outputs, it becomes harder to change these weights if the next output is undesired. This reduces the amount of variation between weights between iterations making the network more stable. Overall backpropagation with momentum had the lowest mean squared error and finished the quickest.

The total error is the max amount of error in a network allowed whilst training. According to fig. 5, a max error of 0.01 is best suited for this network. This might seem counter-intuitive but if the max error of a network is too low than the network is not generalised enough to account for unexpected inputs. The network will only accurately provide the desired output if it had encountered that input previously.

Based on all previous test the most optimal neural network configuration was determined to be one with 144 hidden neurons, backpropagation with momentum learning rule, a learning rate of 0.1, a max error of 0.01 and bias turned off. This configuration is depicted in fig. 6 with a total mean squared error of 0.087337245 over 48 iterations. This network is the second fastest and had the lowest mean squared error out of all previously tested configurations. The network met the goal of being generalised as it could predict most combinations that it had not seen before and had a total mean squared error of 0. 087337245 (<0.1). Unfortunately, it is unclear if it had met its goal of competing at a human level as data was not able to be collected in time for this evaluation. The original intent was to have several images displaying a 5x5 grid of cells with various combinations of cells. If the AI were able to correctly identify the same or more correct cells than a human, it would be considered human comparable. The data would have been collected with a survey that presented the user with several images, with each image having the options “there is a mine”, “there isn’t a mine”, and “not sure”.

Currently, the neural network is limited by the amount of data that was collected. Collecting data was time-consuming which if repeated could be collected automatically with the need for manual control. Another aspect of the neural network that could be improved upon is the overall network architecture. If the number of neuron inputs could be reduced that would also reduce the amount of data required for a generalised system.

Results

Data was collected using the minesweeper program which collected 3105 records of data. This data was split into a 90% training set (2795 records) and 10% testing set (310).

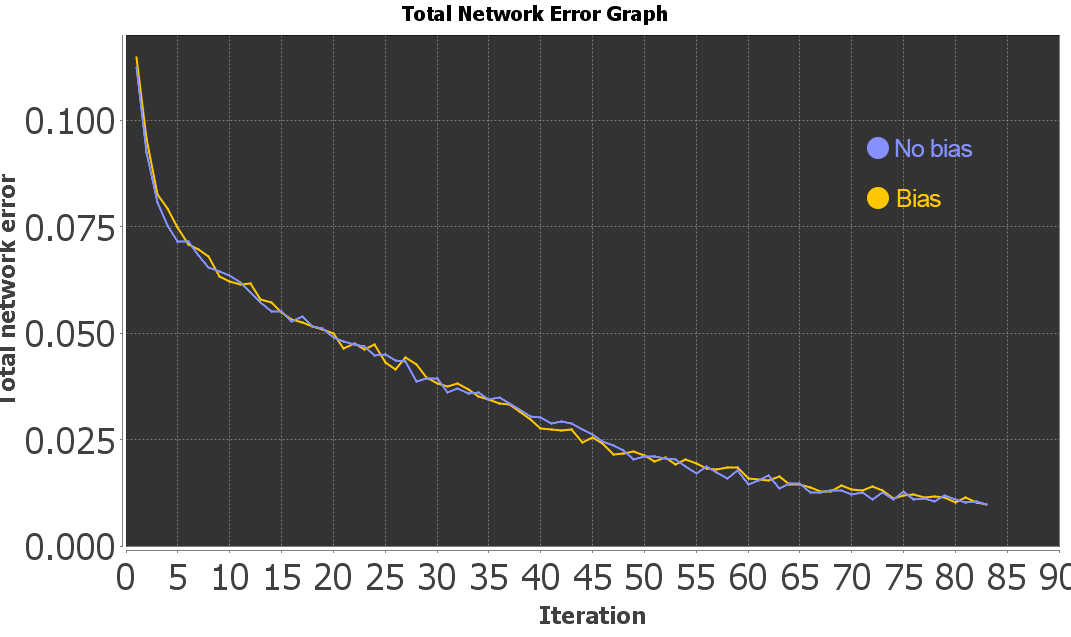


Fig. 1. Total network error of bias vs non-bias networks

In Fig. 1 the bias network and nonbiased network both completed the training set in 83 iterations. When testing the neural network, the bias network received a total mean squared error of 0.100886648 whilst the non-biased network received a total mean squared error of 0.091579768.

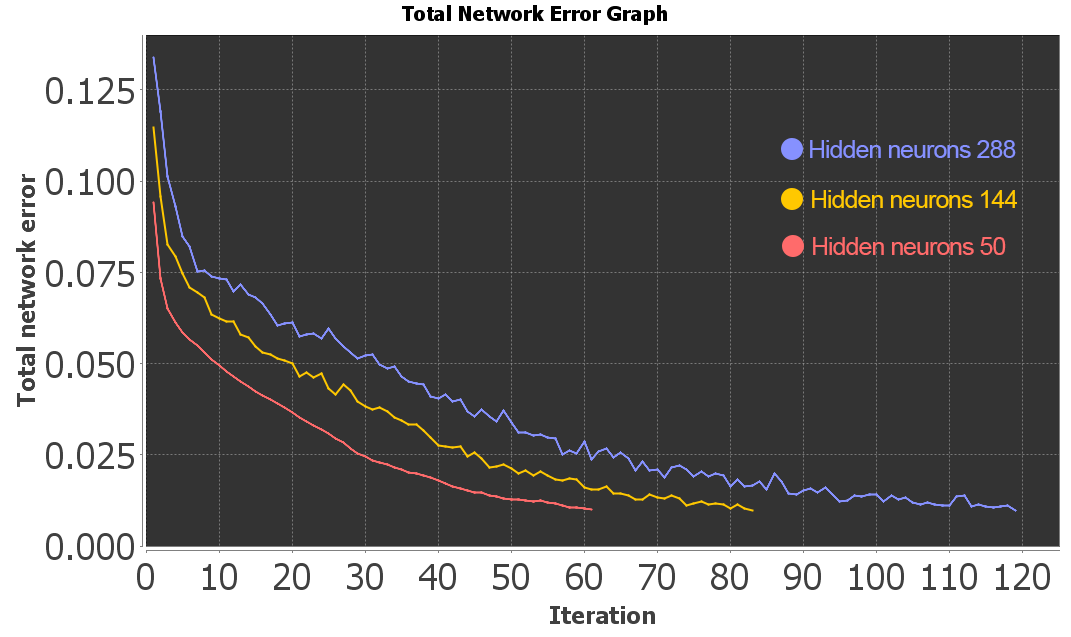


Fig. 2. Total network error over iterations with networks of  
 288, 144, and 50 hidden neurons

In Fig. 2 the network with 288 hidden neurons completed the training set in 119 iterations. The network with 144 hidden neurons in 83 iterations and the network with 50 neurons completed in 61 iterations. Neural networks with 288, 144, and 50 received total mean squared errors of 0.204106838, 0.100886648, and 0.112549848, respectively.

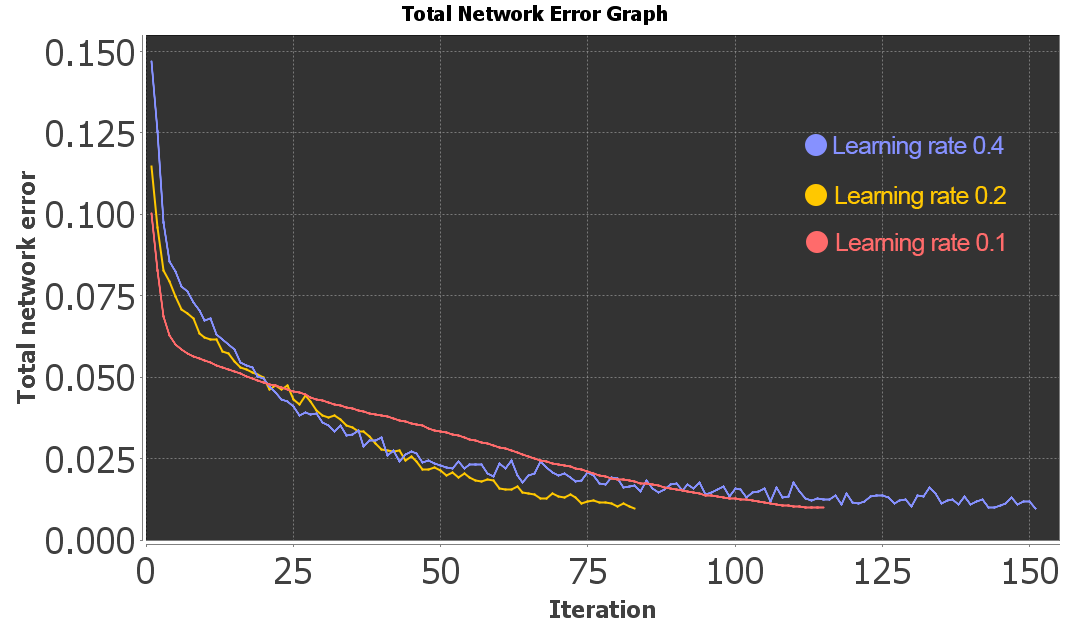


Fig. 3. Total network error over iterations with neural networks   
of 0.4, 0.2, and 0.1 learning rates

In Fig. 3 the network with 0.2 learning rate completed the fastest at 83 iterations and the network with 0.4 learning rate completing the slowest with 151 iterations. The network with 0.1 learning rate completed in 115 iterations. Neural networks with 0.4, 0.2, and 0.1 learning rate received total mean squared errors of 0.103220804, 0.100886648, and 0.097156093, respectively.

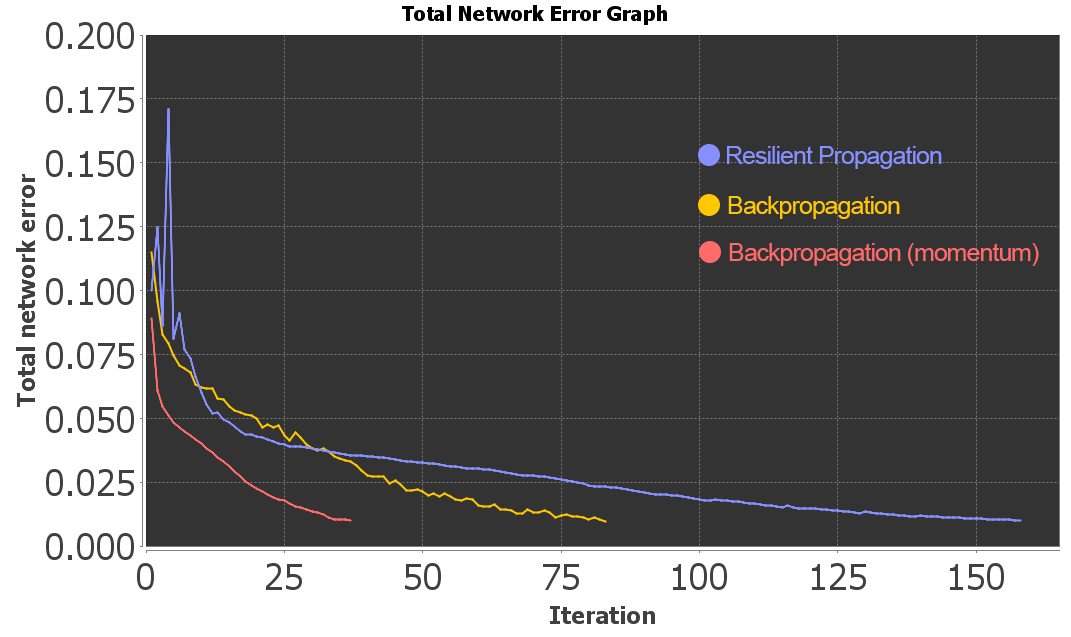


Fig. 4. Total network error over iterations with resilient propagation,   
backpropagation, and backpropagation with momentum learning rules

In Fig. 4 the network with resilient propagation completed the training set in 158 iterations. The network with backpropagation in 83 iterations and the network with backpropagation with momentum completed in 37 iterations. Neural networks with resilient propagation, backpropagation, and backpropagation with momentum received total mean squared errors of 0.094629169, 0.100886648, and 0.097436501, respectively.

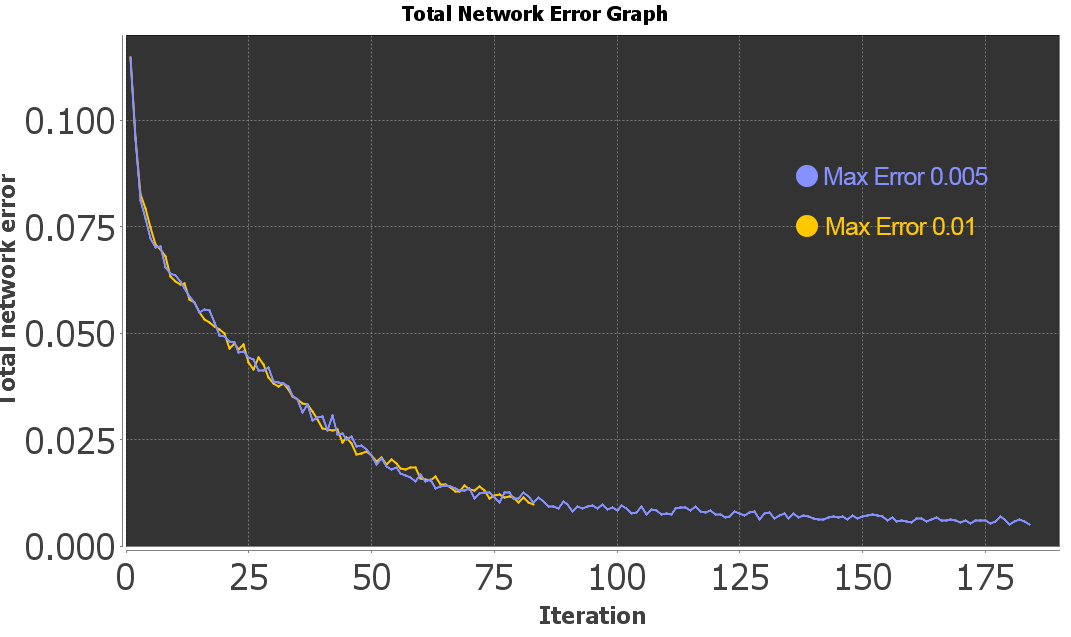


Fig. 5. Total network error over iterations with neural networks of  
 0.005 and 0.01 max error

In Fig. 5 the network with 0.005 max error completed in 184 iterations and the with 0.01 completed the training set in 83 iterations. Neural networks with 0.005 and 0.01 received total mean squared errors of 0.109714657, 0.100886648, respectively.

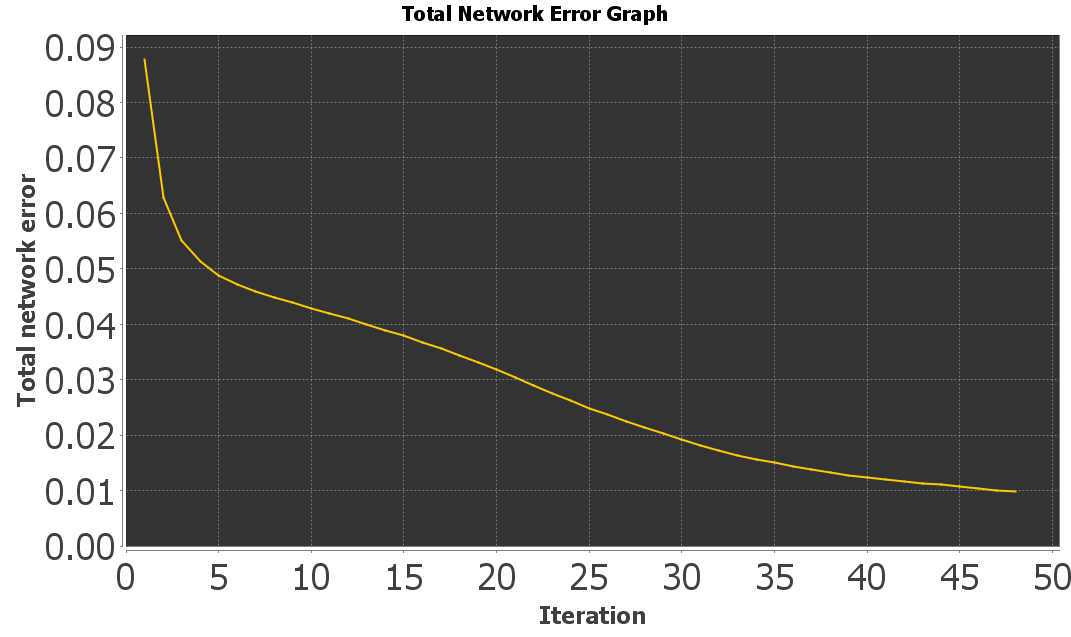


Fig. 6. Total network error over time with complete neural network

Conclusion

Using a multi-layered perceptron, cells were predicted in a game of minesweeper. The neural network created was able to reach a total mean squared error of 0.087337245 over 48 iterations. The network reached is the goal of becoming a generalised neural network, however, with insufficient time it was not able to meet its goal of becoming human comparable. Although this is a great achievement, things can still be improved such as the amount of data collected, how it was collected, and refining the overall architecture of the neural network.

References

[1] T. Beysolow, Applied Reinforcement Learning with Python: *With OpenAI Gym, Tensorflow, and Keras*, New York: Apress

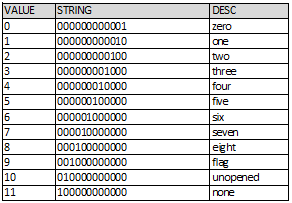
[2] A. Adamatzky, “How cellular automaton plays Minesweeper,” *Applied Mathematics and Computation,* vol. *85*, no. *2,* pp. 127-137, 1997

[3] L. Gardea, G. Koontz and R. Silva, “Training a Minesweeper Solver,” *Standford,* Autumn 2015. [Report]. Available: <http://cs229.stanford.edu/proj2015/372_report.pdf>. [Accessed: 21/08/2020]

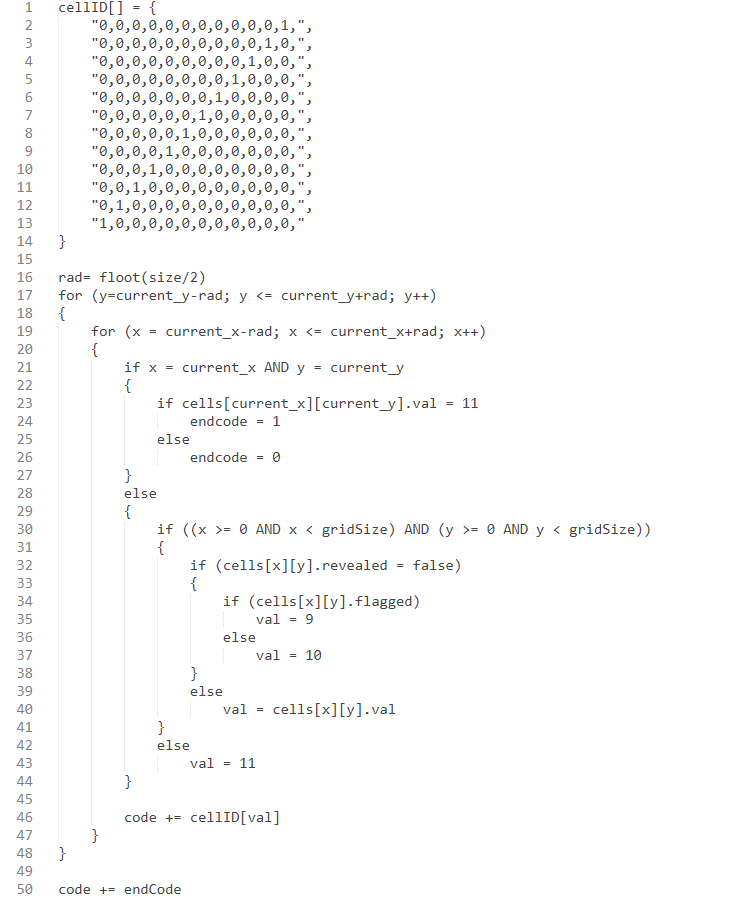
[4] X. Zeng and D. S. Yeung, "Hidden neuron pruning of multilayer perceptrons using a quantified sensitivity measure," Neurocomputing, vol. 69, no. 7, pp. 825-837, 2006/03/01/ 2006, doi: https://doi.org/10.1016/j.neucom.2005.04.010.

Appendix

**APPENDIX A: CELL INPUT VECTOR VALUES**

****

**APPENDIX B: CELL ENCODING ALGORITHM**



User Guide

To re-generate the results described in this report, two pieces of software are needed.

= Java: to run the minesweeper helper tool.

= Neuroph Studio GUI: to run and calculate the neural network.

* Firstly, open Neuroph Studio and load the provided neural network project.
* Double click the neural network.
* Drag the training set (90%) onto the neural network and press train.
* Once trained opened the minesweeper jar file.
* Press middle click to encode a cell. This is copied to your clipboard.
* Go back to Neuroph Studio and click the set-in button with the neural network selected.
* Press ctrl+v and press ok.
* A number should be presented in the output of the neural network. This is the likelihood of a bomb being present in that cell.