Minesweeper AI Using Neural Networks

Alex Knutsson[1], Jakob Unnebäck[2]

Abstract
This report is the result of a project in the course Artificial Intelligence for Interactive Media. It will cover the topic of neural networks in machine learning and describes an implementation of a minesweeper game played by the network. The network uses Deep Q-learning to calculate the agent's next move in the game in order to reveal as many tiles as possible without clicking on the mines. The evaluation of the AI's result is then discussed and possible improvements are presented.
Source code: https://github.com/3DJakob/sweeperAI

Video:

Authors
[1] *Media Technology Student at Linköping University, alekn425@student.liu.se*
[2] *Media Technology Student at Linköping University, jakun708@student.liu.se*

Keywords: Neural Network — Reinforcement Learning — Minesweeper

Contents

1. Introduction

Minesweeper is a classic logic puzzle game from the 1990's with the objective to strategically uncover tiles on a board with clickable squares without clicking on mines[1]. The player gets clues from the revealed tiles in the form of integer numbers on the tile that tell how many mines are connected to that tile. The goal of the game is to uncover all the tiles that does not have mines. The player loses if they clicks on the wrong tile with a mine behind it. When the player presses a tile with zero adjacent mines all other adjacent zero tiles also gets revealed. In that way one move can reveal a big chunk of tiles.

This report covers an implementation that tries to solve the game with the use of machine learning and neural networks. The game can however be beaten with rules with a combination of if-else statements that exploits certain game states and statistics, but we wanted to try to implement a neural network that learns a strategy from a clean slate and progressively gets better and better.

Figure 1 shows the interface of how a minesweeper game. The numbers on the tiles show the number of surrounding mines.

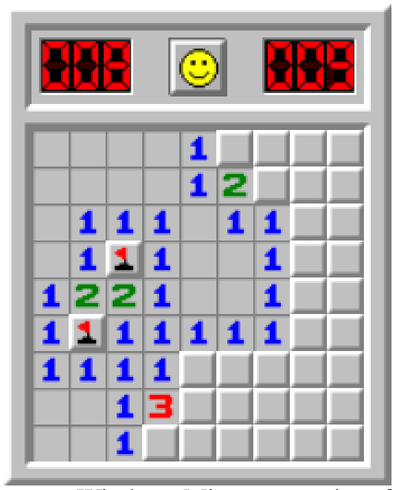

Figure 1. Windows Minesweeper interface

2. Theory

The game minesweeper is completely deterministic, a human with knowledge of the game can perfectly predict the effect of an action. This makes the game a good candidate to be played by an AI as the AI can over time learn the patterns that result in a certain output. For a given setup of the game each tile's probability of being a mine can be calculated. Furthermore in most cases there are one or more tiles which can be determined to be a mine or not with 100% certainty. A good player will always press the completely safe tiles if possible. As the network is trained with all possible combinations of game maps should converge to all the probabilities that the tiles have of being a mine and when that happens the AI will have reaches its maximum potential.

Deep Q-learning is a model-free reinforcement learning algorithm. In reinforcement learning the AI is improved by rewarding good actions and giving penalties for bad actions. The network consist of one input layer one output layer and an arbitrary many parallel och serial hidden layers. Depending on how complected the input and output is correlated more hidden layers are required to sufficiently solve that task. While in theory a large network could always solve the same problems a smaller one can one problem that can occur with a large network is that the AI may overfit to the training data and learn the training set rather than general rules. In this case this is not an issue however since there is only a limited possible inputs and they always yield the same output probabilities. Furthermore since training data can be generated by generating new games so the AI should get the full range of input combinations. As the network size is increased so is the required computing to update and use the network. By finding the optimal network size the training speed can be increased without effecting the performance potential.

The Q-learning formula can be seen in equation 1 where the returned Q includes the current state s_t and the action to take a_t.

$$Q(s_t,a_t) = r_t + \gamma \cdot \frac{\text{estimated future value}}{maxQ(s_t,a))} \qquad (1)$$

The reward r_t is decided based on a reward structure. This is added with the highest possible Q-value of the next state multiplied the γ value which is the learning-rate of the network.

A popular way to feed the AI game information is by reading screenshots of the game[2]. However this is very demanding as each frame has to be interpreted. By coding a game from scratch the AI can access the underlying game functions to speed up the playing process. This also helps with the creation of the reward function as information from the underlying game can be accessed.

3. Method

This chapter describes the method used to create the implementaion of the game.

3.1 The game

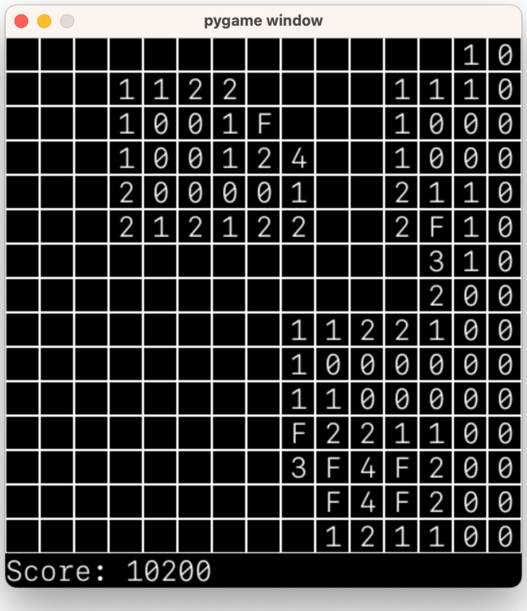

Figure 2. Minesweeper implemented in pygame

The first step was to create the minesweeper game. This was done using pygame as this meant both the AI and game could be ran in the same python environment. Pygame is a simply framework for creating 2D games. First the game map is defined, it contains the information about what every tile contains. For a 15x15 game this is a 15x15 array where the numbers 0-8 coresponds to number of mines, 9 if it is a mine, 10 if unexplored and 11 if flagged. It is created by first setting all tiles to zero then randomly placing mines until all are placed then finally the whole map is looped to update the numbers with the corresponding adjacent mines. This map is hidden for the player of the game as knowing it would defeat the purpose of the game. Once the game map is defined a user map is created. The user map is what the player sees. This is initiated with only 10s as all tiles are unexplored from the beginning of a game.

The game is rendered by looping trough the user map while drawing tiles. Once a tile is pressed the game looks up that tile in the game map and reveals it. If the tile was a mine the game is lost. The function that executes the players move returns the information if the game was lost, if the game was won, the updated score and also the reward associated with that move.

Initially the reward was calculated as $100 * n$ where n is the number of tiles that got revealed. This was however was adjusted after some testing to simply return 100 when the move did not result in a mine. When a mine is pressed the reward -50 is returned.

3.2 The agent

The agent is responsible for extracting game data from the mine sweeper game, transforming the data, feeding the network and interpreting the output to a game move. As stated in 3.1 the user map is for a 15x15 game a 15x15 matrix. In order for the network to differentiate number of mines from flags and unexplored tiles the matrix has to be split into multiple matrices. This is done using one-hot encoding [3]. In one hot encoding each input value is corresponds to a specific input that only state if that value is present i.e. if it is hot. As described in 3.1 there are 11 different possible values for a tile, each of these are connected to an input. So for a network that is trained on the whole game map on a 15×15 game that would mean creating a $15 \times 15 \times 11$ tensor input. However since mines never have to be inputed to the AI as the game is already lost by then the input only needs to accommodate 10 layers.

During development 2 networks were tested. Version 1 using the full $15 \times 15 \times 10$ input with a 15×15 output were the output correlates to how much it wants to press the different tiles on the board. In this case the move using the following equation $(x, y) = Max(output)$ where x,y is the coordinates of the highest coefficient in the neural network output.

Version 2 instead evaluated every tile separately. In this case the AI is feeded the 5×5 enclosing area to the tile that should be evaluated. This results in a $5 \times 5 \times 10$ input size. The output in this case is how much the AI want to press that specific tile and therefore of size 1. When the agent uses version 2 it loops trough all tiles on the board that are not revealed, gets the 5×5 enclosing area feeds that to the AI and saves the output in a matrix with the size of the board. Once all tiles have been computed it uses the same equation as version one making the move with the maximum coefficient. Since there is no point of computing already revealed tiles as clicking them has no effect on the game this algorithm can be very effective especially later stages of a game.

3.3 The network

The neural network's task was to calculate the optimal Q-policy for the next action based on previous knowledge. This was tested with two different input techniques, listed below. The first 200 moves have a decreasing probability of being random from 1 to 0. This helps the network explore how different neurons are correlated. This is then exploited and improved upon once the 200 first moves have been decided.

Version 1

The first version utilized one-hot encoding for each integer case as described in section 3.1. This led to a $15 \times 15 \times 10$ tensor input. The hidden layer size was 2048 and the output size was 225 (i.e 15×15, the size of the board) with ReLU activation as illustrated in 4. The output decided what tile to click with a Q-value for every tile on the board.

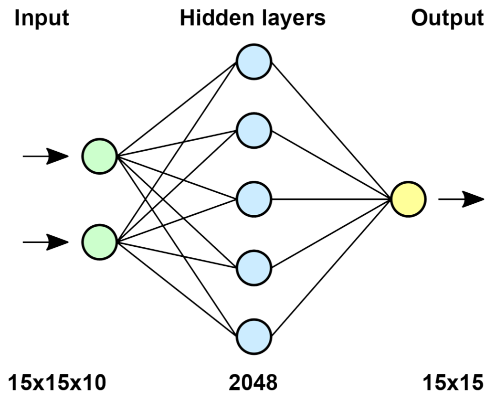

Figure 3. Version 1 of the neural network.

Version 2

The next version of the network had a different approach. It utilized convolution of the board with a 5×5 kernel with one hot encoding. This meant that the network handled a $5 \times 5 \times 10$ input with a hidden size of 1280 and the output size of 1 using ReLU activation as seen in figure 4.

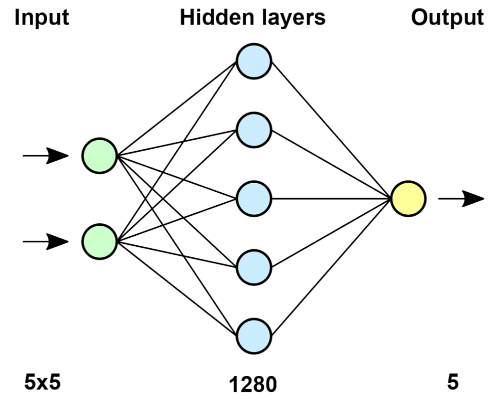

Figure 4. Version 2 of the neural network.

3.4 Heat map

To better understand how the network determines its next move we implemented a heat map to visualize the Q-values for every tile with a red color as illustrated in figure 5.

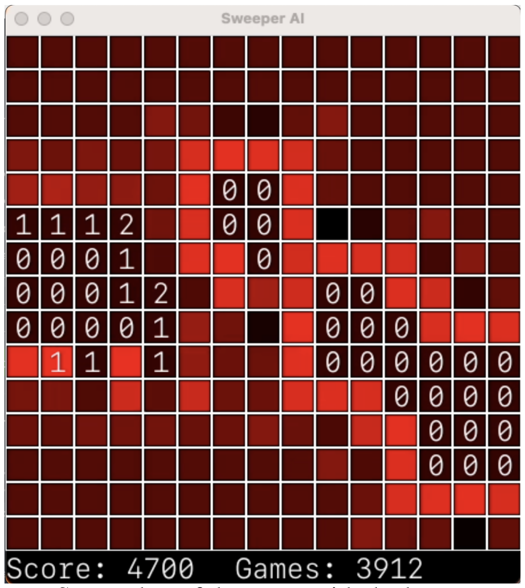

Figure 5. Screenshot of the game with the heat map active.

The bright red tiles are the safest according to the network. The black tiles are likely to reveal a mine. The red tile with a newly revealed 1 was the chosen tile at this state. This was correctly classified since it is impossible to be a mine in that spot because of the neighboring zero above it. In this particular state we can also see that the agent has learned that tiles surrounding zeros are safe.

4. Result

In this section the results of the testing will be detailed. As explained in 3 two different networks were used and therefore both results will be listed.

4.1 Version 1

Version one is using the whole game map as input.

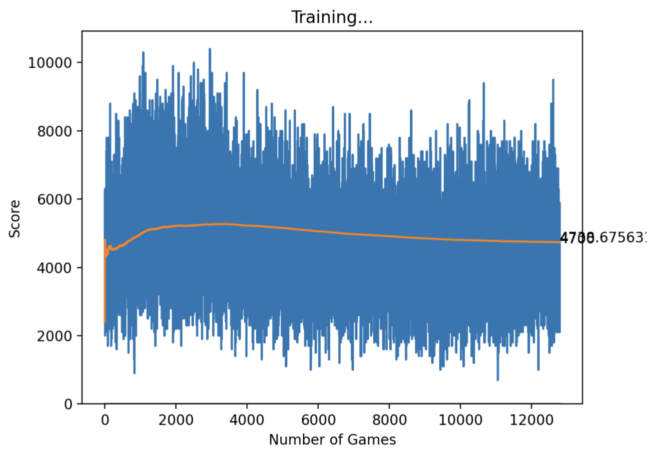

Figure 6. A plot of the score (blue) and mean score (orange) over the course of around 13 000 games with the reward function *relative* to explored tiles

In the resulting graph above we see an initial improvement of the mean score around 2000 iterations. However past that we see it slightly decline without any further improvement.

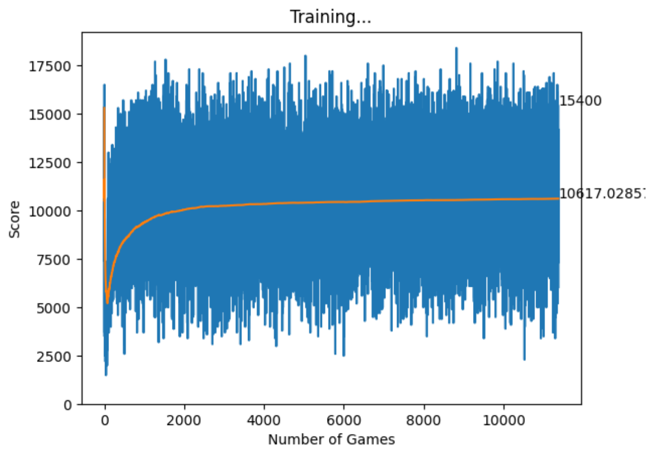

Figure 7. A plot of the score (blue) and mean score (orange) over the course of around 12 000 games with the reward function *unrelated* to explored tiles

With the adjusted reward function the AI quickly surpasses the mean score of the AI in 6. By the time it reaches around 12 000 games the mean score has climbed to 10600. In this case the maximum possible score is 18500.

4.2 Version 2

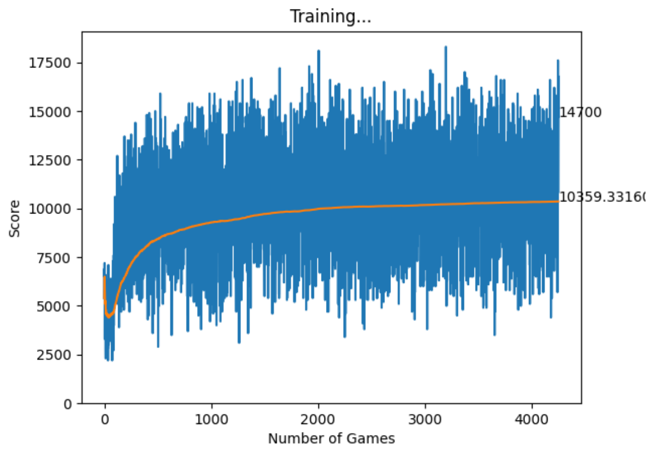

Figure 8. A plot of the score (blue) and mean score (orange) over the course of around 4200 games with the version 2 5x5 network.

Version 2 initially learns much faster and gets way less really low scores in the 2000-4000 range.

5. Discussion

The results gathered show a lot of different interesting aspects of the behavior and potential of the neural net. In this chapter the results will be discussed and the different problems and solutions with the different models.

5.1 Version 1

In figure 6 we can observe how mean scores start to decline after around 2000 in score. This is because the the AI is playing the short game it simply want to maximize the reward for a particular move. And since presses out in the unexplored can potentially reveal a lot of tiles and therefore give high reward it makes up for the risk. We realized however that this was a flaw in the reward system as when playing mine sweeper no extra points are given for revealing the map in the fewest moves. Instead safe moves should always be a priority. This was adjusted by setting the reward function to a constant +100 if the move did not lose the game.

Changing the reward function greatly increased the performance of the AI as can be seen in 7 where the mean score increases constantly. If this model were to be trained for a very long period the potential could be very high.

However a big problem with version 1 is that the AI often draws the conclusion to press a already revealed tile. Even when doing so is associated with a negative reward the AI has a very hard time drawing the parallel to what makes a revealed tile. The reason why this is a huge problem is that the AI often gets stuck trying to press tiles without any effect which in turn makes the training process really slow. In an attempt to solve this another input layer was added with the map of explored and unexplored tiles only. However in our testing this did not help the AI draw the parallel. Another solution tested was to from the output discard any coefficients related to explored tiles. This did speed up the process but might have other unforeseen consequence as the AI may instead then learn to press already explored tiles as it thinks that is why it is succeeding.

Version 1 is also very specific. It only works with the board size as that correlates to the number of inputs in the network. It has basically learned all the different patterns of inputs for that game size. Changing the board size means that the AI has to be trained from scratch. Also changing the mine count could have a significant impact since it changes the probabilities of high and low numbers on the board.

5.2 Version 2

The fact that the version 2 model yields a lot less low scores tell us that the model has a greater understanding of the meaning of 0 tiles. Identifying that all tiles around a 0 are safe is the first step to consistently securing scores over 6000. This is further proved in 5 where we can clearly observe how the AI have identified safe tiles around zeros. Version 2 is also a lot more adaptable since the input size of the network is not related to the game size. This means that the AI can be trained on a smaller board to then move on to a harder one down the line.

Since version 2 is a significantly smaller network it can be trained a lot faster. Since each tile is evaluated to network for each move the AI very quickly learns common patterns like how zeros and ones work.

While initially it may seem like only a 5×5 surrounding area is used to determine a tile we later realized that this is not the case. In certain situation it might be necessary to conclude if one of the tiles around the edge is a mine as that could eliminate numbers on tiles closer to the center. This way information can propagate over the playing field to the tile you are evaluating. This is likely why the improvements of this method converges to zero without reaching a very good win rate. One possible way to combat this was implementing another input layer with flagging data where flags were set from a threshold of the lowest scoring coefficients from last round. The issue with this method is that the coefficients vary greatly in range between runs and it proved hard to find a good threshold even when using percent. In our testing the improvements of this method was within margin of error and therefore not worth pursuing. Other possible solutions were considered but not tested and can be found in section 5.3.2.

5.3 Future improvements

While the AI works and is pretty impressive to watch playing games there are many more improvements that could be made. In this chapter these improvements will be discussed.

5.3.1 Out of bounds handling

As of now, the agent does not distinguish between blank tiles and tiles outside the boundary of the game. This could help the agent to make better evaluations at edge tiles and get a better understanding of the game state when using the convolution method (version 2). This could be solved by passing another layer as input since they never can contain a mine.

5.3.2 Spatial understanding

As a human playing, we can draw conclusions based of the knowledge of if other tiles in the area are mines. This type of conditional probability is only accounted for the adjacent tiles in the 5x5 network.

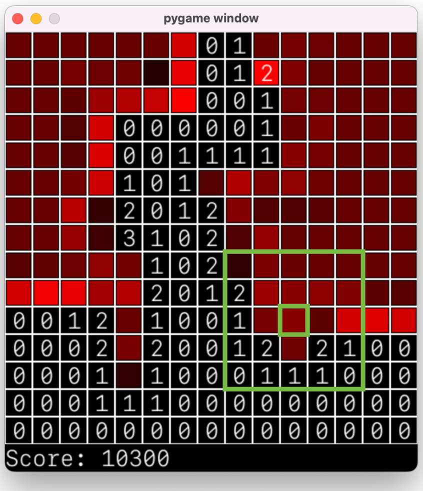

Figure 9. Screenshot of the game with a tile with surrounding 5x5 area highlighted in green.

Figure 9 have one of these situations where you need to look at the surrounding 7x7 to determine that the tile in fact is not a mine. This tile can therefore never be evaluated with 100% certainty with the current configuration.

Could need larger kernel size (7x7)

One possible solution to this could be to simply increase the surrounding input area. This would help the AI interpret propagating probability of mines one tile further away. While in theory mine propagation could effect moves on the other side of the board this gets increasingly unlikely the more tiles away we step. Therefore it would require some testing but most likely 7x7 would be enough but testing 9x9 could also be of interest.

Back feeding probabilities

Another possible solution much like the flagging solution described in 5.2 could be back feeding the network with another layer with the corresponding coefficients from last computation. This could help the AI identify tiles it thinks are mines which could help it make a better guess on the current tile. If this method is better than simply increasing the input size to a 7x7 could be interesting to explore in the future. In the case figure 9 the highlighted tile could then be evaluated to be safe given that the tile under and to the left of it have a very high previous mine probability.

5.4 Diversifying the training data

In the current state the AI is trained by playing the game until it loses and then starting over. While this is very satisfying to look at it is not very efficient as the AI trains a lot on tile combinations present in the early game and also gets exposed to common numbers like 0 and 1 often while it might take thousands of iterations before a 6 or 7 is discovered. Certain combinations with very few unexplored tiles might take a very long time to be learned as a lot of correct moves have to made to get there. It would be interesting to create a function that generates possible map training cases with a more constant distribution. This could potentially help the AI progress faster.

6. Conclusion

The conclusions from this projects are threefold

1. Using a neural network to play the game minesweeper does work quite well in practice.
2. The best move in minesweeper is always the safest, not the move that progresses you furthest.
3. By using a local area as network input instead of the whole game map the network can be much more adaptive to different maps.
4. Mine probabilities can propagate trough tiles and therefore some method to account for this must be implemented to get the full potential of the local method.

References

[1] Wikipedia. Minesweeper (video game).

[2] Bai Li. How to write your own minesweeper ai.

[3] Wikipedia. One-hot.