The Minesweeper - Techniques of artificial intelligence

Anthony Zhou[1], Hamza El Miri[1] & Julien Baudru[1]

[1]Université Libre de Bruxelles, Brussels, Belgium

1 Introduction

Following its integration by standard on Windows 3.1 systems in 1992, the Minesweeper quickly became one of the most popular puzzle games of the century. The goal of this single-player game is simple, the player must discover as many tiles as possible in the game while trying to never click on a mine. The player has clues about the potential positions of the mines, thanks to the number given on a discovered tile, he can know how many mines are in its neighborhood.

In these pages, we will present the different points we needed to create an artificial intelligence capable of playing the game described above in the most successful way possible.

1.1 Motivation

Minesweeper has been shown to be an **NP-Complete**[1] puzzle. As a consequence, there is no (*deterministic*) algorithm that can solve it efficiently (*in polynomial time*), in fact, the time complexity of such algorithm is a function of the board size and mine density. Our goal in this project is to use AI to solve any Minesweeper board efficiently.

2 Models

To solve the given problem, we first reproduced the game with its original rules and a scoring system based on the total number of tiles discovered on the board. Then, we implemented different solutions to solve this game, the models we selected are the following:

1. A solver (an algorithmic logical approach)
2. A convolutional neural network (CNN)
3. A reinforcement learning algorithm (RL)

Some screenshots of our application can be found in the section 4.

[1]R. Kaye, Minesweeper is NP-complete, The Mathematical Intelligencer 22, nr. 2, pp. 9–15, 2000

2.1 Solver

Before implementing any machine learning technique, we first designed a solver for the game. The interest of proceeding in this way is to establish a reference base to compare the results obtained by the different models implemented.

2.1.1 Algorithm

The naive solution would be an exhaustive search implementation but this would be too slow. While an efficient solution does not exist, we can use the rules of the game to mitigate the computation time. To do this, we identify two rules for the algorithm.

Rule 1: *If the number of neighboring tiles equals the value of the revealed tile, flag all tiles around it as mines.*

In this case, we know for certain where the mines are so no further computation is necessary as illustrated in figures 1a and 1b. Similarly, if the value of the tile minus the number of flags around it equals the number of remaining hidden tiles, then all of the remaining tiles must be mines.

(a) Only one possibility (b) After applying rule 1

Figure 1: Rule 1

Rule 2: *If there are an equal number of tiles flagged in the neighborhood of a revealed tile to its value, the remaining tiles are safe.*

This rule relies on the flags placed by the first rule to determine the safe tiles as illustrated in figure 2.

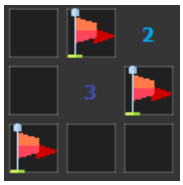


Figure 2: Rule 2: The remaining hidden tiles are safe

These two rules allow us to solve a large number of situation in linear time. However, there are certain tile arrangements that do not give us enough information to apply them as shown in figure 3.

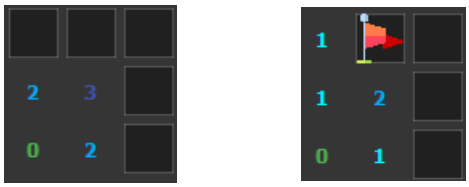


Figure 3: Cases without enough information

The only way to solve such cases is through the use of exhaustive search to determine the probability that a tile contains a mine or not.

Exhaustive Search We used a binary tree approach to implement the search which is generated tile by tile as seen in figure 4. A branch from the root node to a terminal node represents a possible combination. At each insertion, we verify that the rules of the game are respected, if the combination is invalid, we reject the branch significantly cutting down the possibilities. Furthermore, at each insertion we update the probability that each tile contains a mine.

This method has two major advantages over a traditional backtracking method:

- If at any point, the probability of a tile a containing a mine is 0 (*or 1*) we immediately stop the execution because this can only happen if all right splits (*or left splits*) at tile a have been rejected, meaning that all final combinations do not contain a mine (*or contain a mine*) at tile a. Thus, unlike a traditional backtracking method, we do not always have to explore every possible combination before making a decision.
- The tree representation is memory efficient. Each branch represents a combination and differences in combinations are determined by the splits in the branches, therefore, there is no duplicate information stored.

2.1.2 Results

With these rules, the algorithm can solve a large number of games relatively quickly. We evaluated its performance over 500 games in each difficulty. Figure 5 outlines the results.

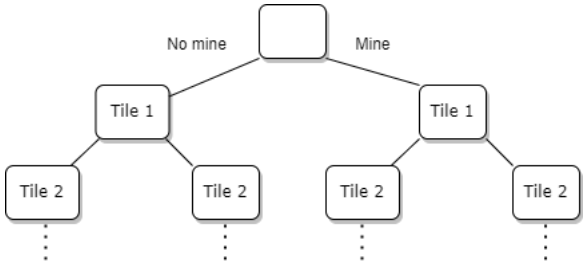


Figure 4: Tree generation: 2 new nodes representing the tile are inserted for each terminal node.

	Board size	Mines
Beginner	8x8	10
Advanced	16x16	40
Expert	24x24	99

Table 1: Difficulty breakdown

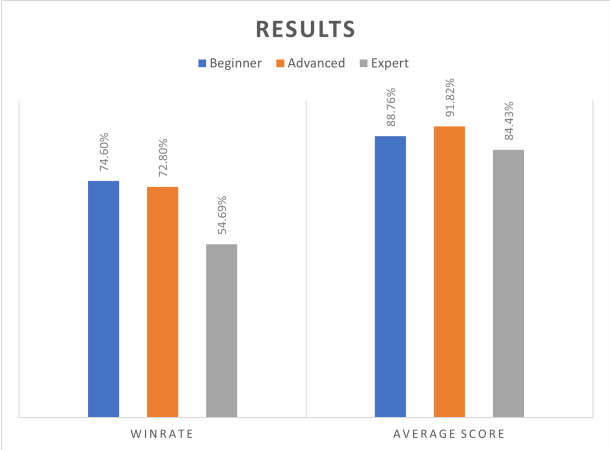


Figure 5: Results over 500 games played for each difficulty

At its core, Minesweeper is a game of luck which means that a 100% winrate is impossible to achieve. Nevertheless, the solver performs very well in beginner and advanced difficulties but struggles in expert (see 1). The win rate and the average score are the metrics we will use to judge the performance of our AI models.

2.2 Convolutional neural network (CNN)

To solve the given problem, we have chosen a convolutional neural network (CNN). We chose this type of architecture because they are usually used in image processing and are therefore effective when it is a question of detecting features of the matrices of values that are given to them as input. More precisely, the chosen architecture is close to the famous *AlexNet* network, the different layers of our network are detailed on figure 6.

This CNN then receives game boards as input (details in section 2.2.1) and returns a matrix giving probability of find-

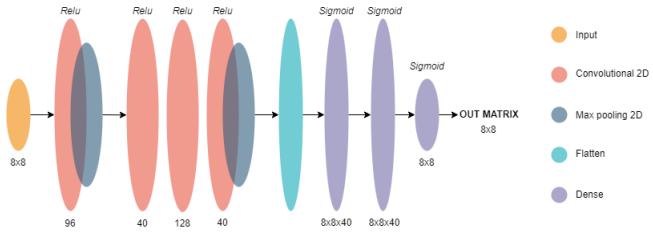
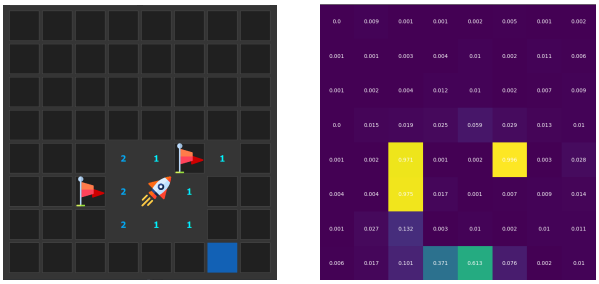


Figure 6: Architecture of our model based on the AlexNet network

ing a mine at each position (*1 if mine, 0 otherwise*). Then, for the positions in the surroundings of the revealed tiles, an algorithm is in charge, on the one hand, of finding the minimal probability (violet tiles in 7b), i.e. the position of the next move (blue tile in 7a), and on the other hand, of finding the maximal probability (yellow tiles in 7b) and placing a flag there, i.e. the most probable position of a mine at this moment of the game.



(a) Board game (b) Board game probabilities

Figure 7: Input and output matrices

2.2.1 Training

There are several possible options for creating the set of inputs required for the training of the CNN. The method we have chosen is to generate many winning games and for each sequence of boards of these games, we have kept only one board every 15 moves, this allows us to obtain varied game situations, namely beginning of games, middle of games and end of games. Initially, we only generated a set of beginning of game, which prevented the model to progress further in the game. These boards, or matrices, are constituted of the values of the revealed tiles (*from 0 to 8*) and of the unknown tiles for which we have fixed the value to -1. Moreover, during the training, for each of these generated boards, the corresponding solutions are also provided to the CNN, which take the form of matrices in which the positions of the mines in the perimeters of the revealed tiles are marked by a 1, the values of the other positions are set to 0.

Thus, we managed to reach an accuracy of 62.35% and an error loss of 0.0424%, to do this we trained our model on 5,000,000 boards and this for 10 epochs, figure 8 shows the evolution of the model accuracy as a function of epochs and figure 9 shows the evolution of the error loss as a function of epochs.

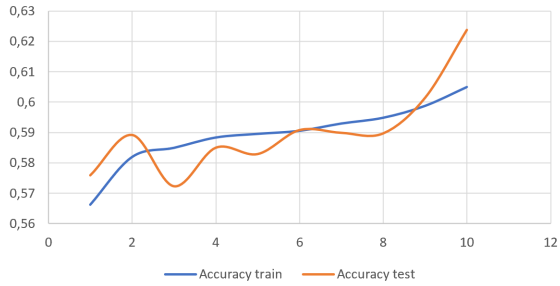


Figure 8: Accuracy evolution per epochs

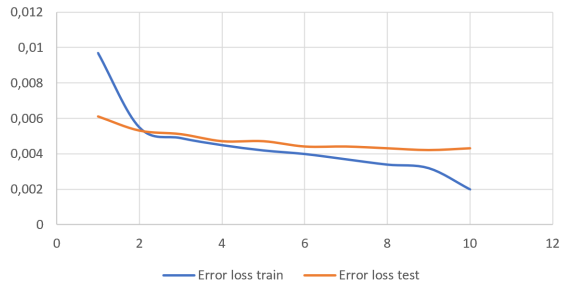


Figure 9: Error loss evolution per epochs

2.2.2 Testing

Then, in order to evaluate the real performance of our model, we tested it on a set of 1,000 games that it has never been confronted with, in other words a validation set. Under these conditions, the model reaches a win rate of 73.6%. The figure 10 shows the evolution of the percentage of games won by our model as a function of the number of boards on which it trained during 10 epochs. We notice that this value seems to follow a logarithmic function.

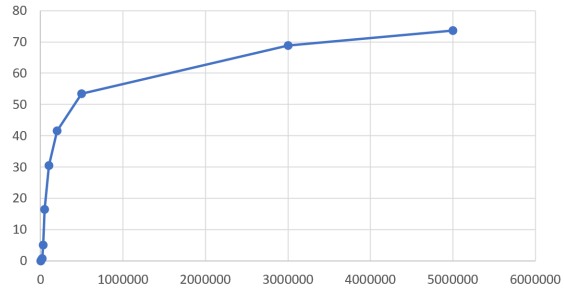


Figure 10: Win rate evolution (%) per number of boards

2.3 Reinforcement learning (RL)

For our reinforcement learning model, we use a modified version of Q-learning. After learning the Q values, the agent would use the values obtained to play minesweeper. The standard Q-learning algorithm is as follows:

$$Q(s_t, a_t) \leftarrow Q(s, a) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (1)$$

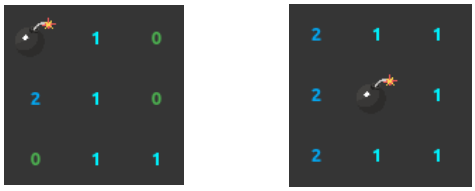
where s_t is the state at time t, a_t is the action at time t, α is the learning rate, r_t is the reward at time t and γ is the discount factor. This structure allows the agent to learn not just about the direct reward of a particular action, but whether a particular action is more likely to lead to reward in the long-term. While this is crucial for many games like chess, we are not as concerned with the endgame result in minesweeper. Instead, we are more interested in the immediate reward, whether a particular move will uncover a mine on a specific board configuration. For that, we simply remove the red part in equation 1 and we get:

$$Q(s_t, a_t) \leftarrow Q(s, a) + \alpha * (r_t) \quad (2)$$

We exclusively used the 8x8 board with 10 mines for our training and testing.

2.3.1 Q-table

We will use a cluster of 3x3 around a tile to see if it gives enough information for the RL agent to make the right decision, examples of cluster can be seen in figure 11.



(a) With a normal tile (b) With a mine

Figure 11: Examples of 3x3 clusters

The different clusters will be the states of our Q-table and each time a new cluster is discovered, it will be added in the Q-table. Two actions will be available for our agent: *click* or *ignore* the tile. A reward will be given depending on both the tile and the action (cfr table 2).

States	*Click*	*Ignore*
Tile is **not** a mine	+1	-1
Tile is a mine	-1	+1

Table 2: Reward matrix

For every move, the agent looks at the Q-table to see if it has already recorded the state and select the action with the highest reward, called greedy action.

2.3.2 Training

We train our agent using the epsilon greedy algorithm to balance exploration and exploitation. We give our algorithm 1,000,000 episodes to converge and we evaluate our agent every 10,000 episodes. We trained the agent with $\alpha = 0.1, \varepsilon_{max} = 0.9, \varepsilon_{min} = 0.1$ and $\varepsilon_{decay} = 0.99$.

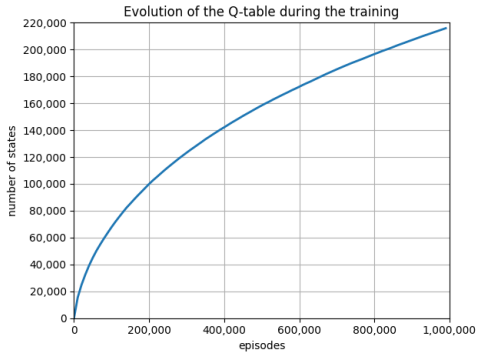


Figure 12: Evolution of number of states in the Q-table

Figure 13: Evolution of number of wins of the agent

We can see in figures 12 and 13 that the numbers of states and wins keep increasing meaning the agent learns.
We now make a comparison on the number of states between an agent trained over 500,000 episodes, an agent trained over 700,000 episodes and the agent trained over 1,000,000 episodes in the figure 14.

We can see that augmenting the number of episodes increases the number of states discovered in our Q-table.

2.3.3 Testing

We now evaluate the trained agent on 1,000 games. Note that undiscovered states during the testing phase are ignored.

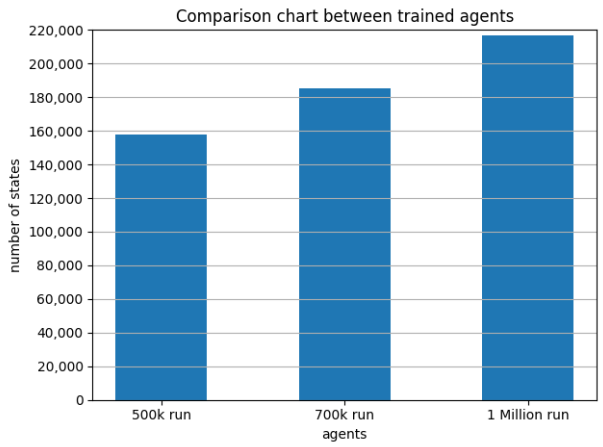


Figure 14: Comparison graph between an agent trained over 500,000 episodes, an agent trained over 700,000 episodes and the agent trained over 1,000,000 episodes

We reach a win rate of 0.8%. In figure 15, we can see the evolution of number of wins of the agent during the evaluation.

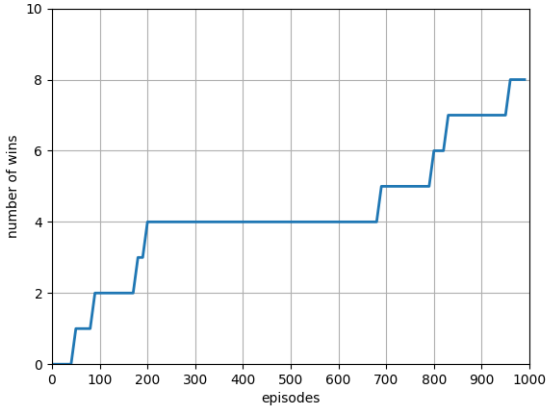


Figure 15: Evolution of number of wins of the agent during the testing phase

2.3.4 Further improvement

We can see that our reinforcement learning performs poorly, to enhance it we could possibly increase the number of episodes for the training phase or increase the size of the cluster. As the essence of this game consists in a pattern matching, another idea could be to move to deep Q-learning and use a convolutional neural network instead of the Q-table as it performs better for pattern matching (cfr section 2.2).

3 Conclusion

To conclude, we notice that for the same size of board, i.e. 8x8, the algorithmic solver is the most performant with 75.2% of win rate, followed closely by the CNN with 73.6%. Then comes the model using reinforcement learning with 0.8% of win rate (see figure 10). It is important to note that the CNN and RL models can still be improved either by providing more training boards or by extending the duration of their training.

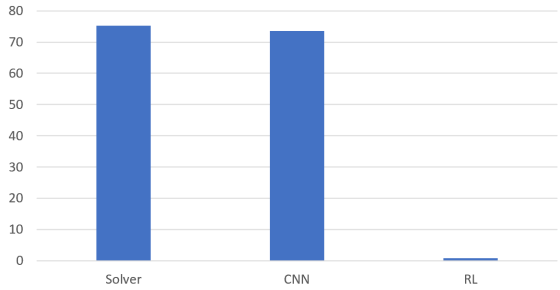


Figure 16: Comparison of the different models' win rates (%)

In terms of speed, we note that the fastest model is the RL, followed by the CNN and as expected, the solver is the slowest as shown in figure 17. Finally, we've shown that a well-trained AI can solve Minesweeper games in an efficient manner.

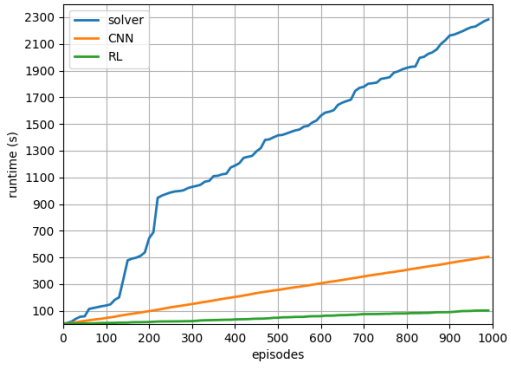


Figure 17: Comparison of the different models' runtime

4 Screenshots of our application

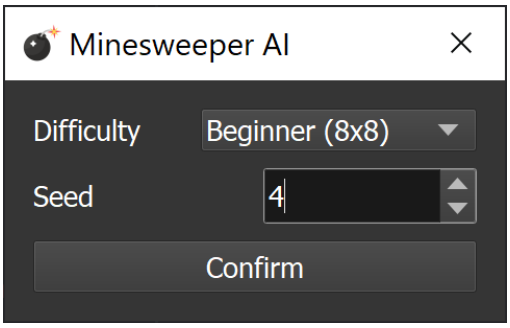


Figure 18: Difficulty and seed selection menu

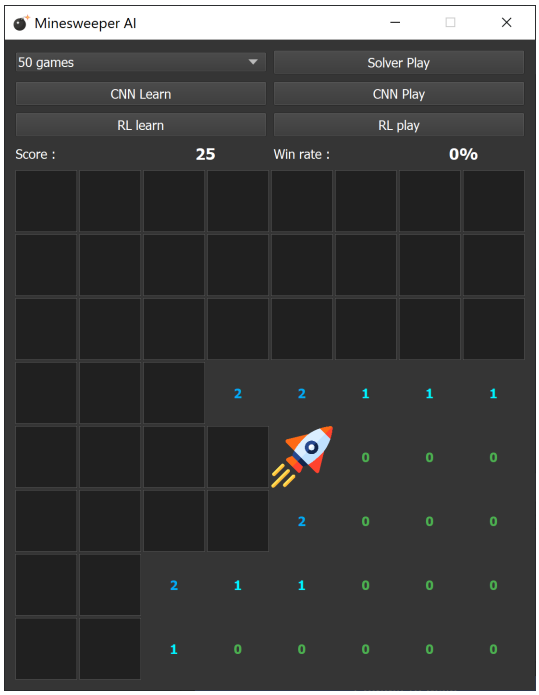


Figure 19: Game configuration with a board size of 8x8

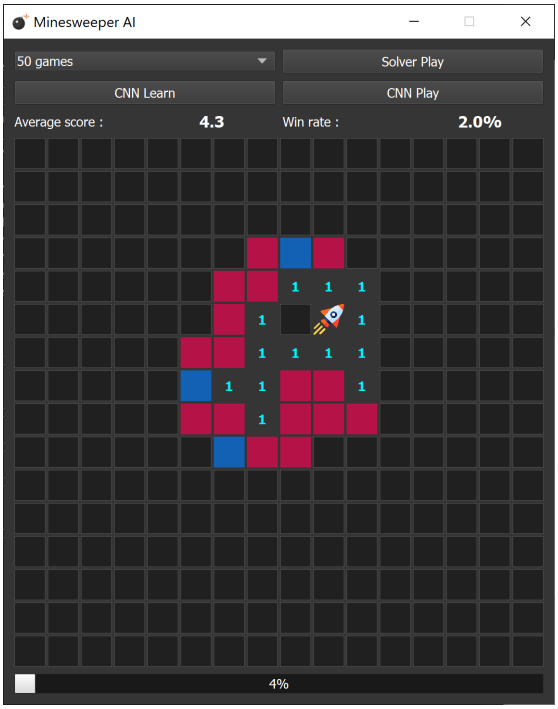


Figure 20: Solver running on a 16x16 board. Red = No mine is being considered at that tile. Blue = A mine is being considered

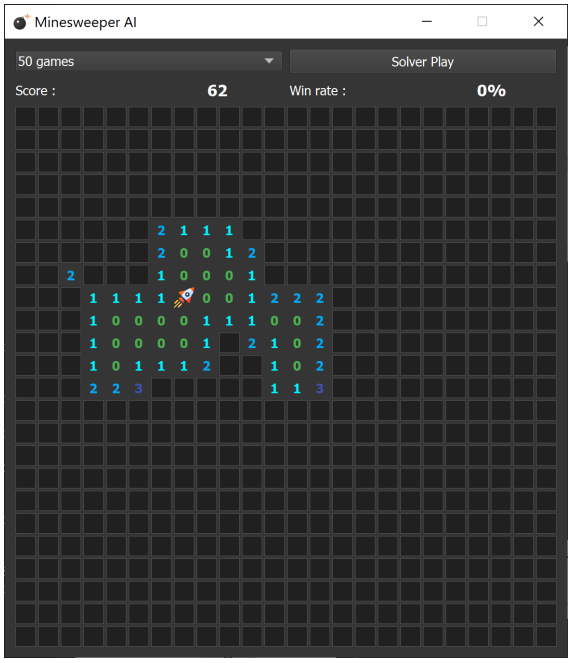


Figure 21: Game configuration with a board size of 24x24