Solving Minesweeper with AI: A Hybrid Methodology

David Kleppang, Maegan Sobel, Eric Williams

Penn State World Campus

AI-801 Foundations in Artificial Intelligence

May 2, 2025

Table of Contents

I. Abstract

Artificial Intelligence (AI) can be used to solve many complex problems without the requirement of brute-force coding or declaration of the rules and environment upfront. Our team developed and trained an AI agent to solve Minesweeper, a partially observable game that requires both logic and risk analysis to win. Because the AI must make decisions in real-time, it must formulate the best-case scenario with only the given context clues to know which is the correct choice and which will lead to failure. This project combines constraint satisfaction techniques, deep reinforcement learning, and the Markov decision processes as a hybrid approach to solving the Minesweeper board. Constraint satisfaction is utilized to make logical comparisons of the available states to determine what the highest reward outcome will be. This is a valuable tool when the data presented in each state reveals an obvious correct choice. When the choices are more ambiguous, the AI can utilize the Markov decision process. The Markov decision process utilized in this project is implemented within a Deep Q-Learning trainer. It selects actions using an epsilon-greedy strategy to choose the best-case scenario and is given positive or negative reinforcement based on the outcome. By leveraging the strengths of each methodology, reinforced through a slow iteration of rewards and tuning of our parameters, we reached a rolling win rate of 42.4%. This showed that the AI model we designed was able to learn the rules of Minesweeper and solve it without the need for direct manipulation or brute-force coding. When the choices are more ambiguous, the AI can utilize the Markov decision process. The Markov decision process utilized in this project is implemented within a Deep Q-Learning trainer. It selects actions using an epsilon-greedy strategy to choose the best-case scenario and is given positive or negative reinforcement based on the outcome. By leveraging the strengths of each methodology, we were able to accomplish a solve rate of 42.4% which scaled regardless of the number of episodes. This showed that the AI model we designed and implemented was able to learn the rules of Minesweeper and solve it without the need for direct manipulation or brute-force coding.

II. Introduction

Minesweeper is a single-player game first released in 1990 by Microsoft as a free and included Windows computer game [7]. It is played on a grid where mines are hidden behind blank cells until the player selects a cell to uncover. The cells can either be safe or a mine, but you will not know for sure until you select it or deduce through logic. The board offers clues to the location of the adjacent safe and mine cells which is where strategy can be used to solve the board. The winning state of the game is to uncover all safe cells without selecting any cells that contain a mine. If a mine is selected at any time during the game, the game is lost. Common difficulty levels released with the game are Beginner, with either an 8x8 or 9x9 grid and 10 mines, Intermediate, which is a 16x16 grid with 40 mines, and Expert, which is 30x16 with 99 mines. The traditional game is timed, but only to note how long it took to solve the game as it is not a countdown before the user must complete the game (see Figure 1) [4].

Figure 1 - Minesweeper for Windows XP, Intermediate Difficulty. This is a winning state. [4]

While Minesweeper is largely regarded as a simple strategy game, it could be better categorized as a stochastic puzzle game. In a stochastic puzzle, the actions by the player are repeatable, the game

environment is impacted and changed by the player, and chance influences the game to some degree [11]. A probability distribution can be generated based on the game states and the game can offer two equally probable or even contradictory options. Since Minesweeper operates in a partially observable state, there may be two equally probable right or wrong outcomes, and the player influences the game states, this classifies Minesweeper as a stochastic puzzle game and not simply a strategy game.

Solving Minesweeper with brute-force algorithms has proven successful, but depending on the algorithm chosen, it can be costly in terms of computing power and storage to utilize a linear method or a recursive search-based method that identifies all possible future board states before deciding. Additionally, there will be instances where the computer will be presented with equally correct or incorrect choices and must guess on the next move. In a brute-force algorithm, this will be enacted via a random guess lacking valuable context clues and not based on any probabilistic outcomes. Training an AI to solve Minesweeper cuts down on risky guesses and teaches it to make smart moves without exhaustively scanning every remaining cell first.

Implementing the Markov Decision Process (MDP) and Constraint Satisfaction Problems (CSP) within a Deep Q-Learning (DQN) framework is a hybrid method to mitigate the challenges of AI learning to solve the Minesweeper game. MDP is often used to solve stochastic problems since the elements of probability enable a more calculated approach to the probabilistic outcomes [11]. The MDP supplies the state that gives the DQN the information required for its next decision, outlines the action space, directs epsilon-greedy action selection, and formalizes the transition model. The weights are then adjusted based on the current reward factor. CSP is used to give quantitative value to the constraints used in the decision process. In Minesweeper, the numbers revealed on the cells are clues that allude to the location of the next-closest mine and constraints are built on the numbers shown. Local propagation is used to enforce the constraints and backtracking information to improve efficiency. The CSP module computes a probability for every hidden cell and blends that estimate with

a positional heuristic to form a score resulting scores from the sub-state the DQN receives, enhancing its inputs and ultimately improving the solve rate.

A successful AI solver for Minesweeper will be able to play the game and achieve higher win ratios over time. The solver will evaluate probabilities and interpret the game states based on limited context to choose the cell with the best outcome that will ultimately lead to a winning game state. The solver must learn over time what inferences to make on the hidden elements and make the safest decisions when uncertainty is present. To successfully implement the proper training environment, multiple variables are tuned including the epsilon decay rate, the positive and negative reinforcement values, and the baseline learning infrastructure which are monitored through a target network. The solver will evaluate probabilities and interpret the game states based on limited context to choose the cell with the best outcome that will ultimately lead to a winning game state. The solver must learn over time what inferences to make on the hidden elements and make the safest decisions when uncertainty is present. To successfully implement the proper training environment, multiple variables are tuned including the epsilon decay rate, the positive and negative reinforcement values, and the baseline learning infrastructure is monitored through a target network.

The combination of methodologies used in the AI Minesweeper solver enables advanced problem-solving over brute-force calculations. This can be applied outside of stochastic puzzle games and used in real-world applications. Natural language processing, puzzle solving, robotics, and scheduling are tasks that use a complex DQN to make real-time decisions in heavily contextual environments [5]. Additional applications include medical diagnosis of disease, robotics, stock trading, and threats in cybersecurity [3, 6, 11].

III. Problem Formulation, AI Model, and Methodology

Minesweeper is more than just a logic-based decision game. Since it requires inferences based on probabilities and experience, traditional brute-force methods can solve complex boards but would take more time, require more initial data, human assistance up-front, and infrastructure built on simple game rules will not afford the solver enough context to make non-random decisions. The problem that must be overcome is training an agent with reinforcement learning to utilize constraint satisfaction and the Markov decision process and ultimately learn the most effective process to achieve a high win rate. To achieve success, five areas are focused on for performance: state space, action space, reward structure and policy optimization, integration of the hybrid learning model, and performance metrics.

The **state space** holds the values that the solver uses to determine the best action. The cells are numbered from 0 to 8 depending on the number and location of the adjacent mines, and unrevealed cells are marked as -1. In the event a selected cell has a value of 0, all adjacent non-mine cells will be revealed in a "flood-fill" manner using the Breadth-First Search (BFS) Algorithm (see Figure B for an example of a flood-filled board after cell (1,2) is selected). Each turn, the matrix of integers updates the state of each cell.

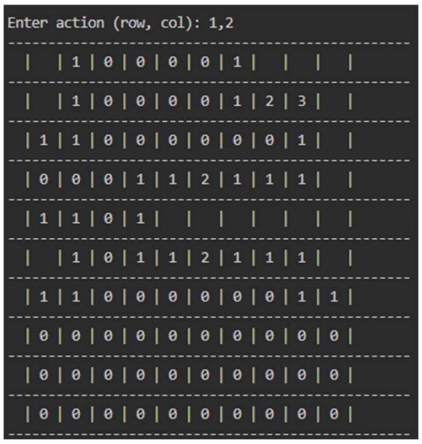

Figure 2 - Interactive Board State after cell (1,2) is selected (row and columns are zero-indexed)

The **action space** is the selection of the cell. Each action will either end the game, win the game, or contribute to the continual reinforcement learning environment through varying negative or positive rewards. The AI must balance risk and opportunity in the action space and appropriate application of the DQN allows for past trials to influence future decision making.

The **reward structure** and **policy optimization** require fine-tuning. Once the main implementation of the DQN and environment are in place, stable advances in the win ratio can be achieved through careful manipulation of the hyperparameters. The reward structure carries three main values, the winning reward, the reward for selecting a safe cell, and the penalty of a loss. The policy can be optimized with changes to the epsilon value that help the solver balance exploration and exploitation. Initially, when the epsilon value is high, the solver will make more random choices which can lead to potential high-risk-high-reward outcomes that it may approach more conservatively at a later time. This must be balanced with the frequency of committing high-risk-high-penalty actions such as losing the game. Additionally, as the epsilon value decays to a set value, the solver will use experience to help provide context for split decisions. This is beneficial for policy optimization if the epsilon does not decay too early, leaving the solver in a state of local optimization instead of learning better strategies. Implementing an epsilon schedule allows the solver to explore more in the beginning trials and exploit prior knowledge in the later trials. require fine tuning. Once the main implementation of the DQN and environment are in place, stable advances in the win ratio can be achieved through careful manipulation of the hyperparameters. The reward structure carries three main values, the winning reward, the reward for selecting a safe cell, and the penalty of a loss. The policy can be optimized with changes to the epsilon value that help the solver balance exploration and exploitation. Initially, when the epsilon value is high, the solver will make more random choices which can lead to potential high-risk-high-reward outcomes that it may approach more conservatively at a later time. This must be balanced with the frequency of committing high-risk-high-penalty actions such as losing the

game. Additionally, as the epsilon value decays to a set value, the solver will use past experience to help provide context for split decisions. This is beneficial for policy optimization if the epsilon does not decay too early, leaving the solver in a state of local optimization instead of learning better strategies. Implementing an epsilon schedule allows the solver to explore more in the beginning trials and exploit prior knowledge in the later trials.

The **integration of the hybrid learning model** (CSP-MDP) into the reinforcement learning framework leverages the strengths of both constraint satisfaction and decision-based planning. CSP relies heavily on logic-based outcomes which are beneficial for the majority of the instances in Minesweeper gameplay but does not adapt well to uncertainty. MDP allows for probabilistic interpretation of risk and performs well when presented with uncertainty. Implementation of both CSP and MDP within reinforcement learning environments like DQNs enables the maximum effectivity of both by rewarding or penalizing the solver based on its ability to apply and implement these methodologies correctly and improve on their utilization over time.

Performance metrics are collected while the AI is training to show valuable information about efficiency and performance. Key metrics include the final reward value, number of squares revealed, win-loss ratio, epsilon value, average time per episode, and average time per move per episode. A final assessment can be performed on the captured metrics and can point to key performance indicators within the code. Tracking the total reward and the epsilon value can show if there is a relationship between a higher reward and decay rate. Additionally, average time per move or episode can highlight computational efficiency or inefficiency. Analysis of the metrics and application of potential improvements is an important step in tuning the solver and optimizing performance. It also shows the successful training of the AI over time.

The Minesweeper board consists of a two-dimensional array that is 10x10 with 8 mines for our “beginner” difficulty and 16x16 with 40 mines for “intermediate” difficulty. Each cell has a value of -1

for hidden, 0 for revealed, and 0-8 for the number of adjacent mines to that cell. The reset() function reinitializes the game state before starting each episode by hiding all the cells and resetting the counts, the alpha parameter, and the win and loss parameter values. To ensure the first move of the next episode will always be safe, self.first_move_done is set to False so the mines can be placed after the first cell selection. The reset() method then runs the CSP logic solver on the new board so the mines can be placed after the first cell selection.

The step(action) method updates the board's state, reveals cells, computes the reward, and returns the new state. Once the first move is complete, it places the mines and sets self.first_move_done to True. It checks for redundant moves and awards a small penalty of -0.1 if the solver attempts to perform the redundant action. If the solver selects a cell with a mine, the variable done is set to True, and won is set to False, -5.0 is given to discourage this action. If the solver selects a cell that is not a mine, the method recursively reveals the adjacent cells us_flood_fill(row, col) and captures the number of revealed cells in the step. The reward is then calculated based on the revealed cells. If all safe cells have been revealed, done is set to True, won is set to True, and a large reward is given. If the game is still going, step(action) returns the information needed to prime the new board state and set up the next step. Metrics collected in the method include win ratio, number of revealed cells, average time per move, and the epsilon value, won is set to True and a large reward is given. If the game is still going, step(action) returns the information needed to prime the new board state and set up the next step. Metrics are collected in the method include win ratio, number of revealed cells, average time per move, and the epsilon value.

The CSP solver is implemented in the apply_csp_solver() function and bridges the constraint satisfaction techniques with our DQN. The function checks the board via our display matrix and identifies all revealed cells. The revealed cells that neighbor hidden cells will have active constraints labeled from 0-8 that describe how many adjacent mines there are. If the cell is hidden, it will have a

value of -1. The constraints are stored as tuples that contain the index of the neighbor cells and the number of adjacent mines. The function uses Deterministic/Decomposition Search (DSS) backtracking to identify cells that are either safe or have a mine and uses forced assignments when there are trivial actions such as three hidden neighbor cells and a constraint that states there are three mines. In this case, all of the neighbor cells will be mines and the assignment can be forced. After the initial force assessment is complete, the solver analyzes the remaining cells using DSS backtracking. The solver will compute the valid solutions in the current state and compute the probabilities of mine locations. Once the solver decides which choice to make, the sub-state is constructed and computes the weighted value for each hidden cell. The weight factor is computed as:

$$\textit{Weight factor} = \alpha\ \textit{(mine probability)} + (1-\alpha)\textit{(location score)}$$

Equation 1

The *mine probability* is calculated using the following rule:

- for each revealed number, record which hidden neighbors must contain exactly that many mines
- if the square is proven safe, p=0
- if the square is proven a mine, p=1
- if uncertain, then generate and count (N) up to a maximum of 20 (*max_solutions*) full board layouts that satisfy the first rule and then divide by N
- if no layouts are generated, return p=0.5

The *location score* is calculated using the following rule:

$$1 - (\frac{m}{l})^x$$

Equation 2

- m = total number of mines,
- l = number of covered cells remaining,
- x = 4, 6, or 8 for a corner, edge, or middle cell respectively

The *alpha* coefficient (α) is a dynamically calculated linear regression value determined using the following equation:

$$\alpha = \theta_1 p + \theta_2 q + \theta_3 \left(\frac{n}{pq}\right) + \theta_4$$

Equation 3

where θ_1 = 0.25, θ_2 = 0.25, θ_3 = 0.2 and θ_4 = 0.05

This information reduces risk when the MDP needs to evaluate probabilistic outcomes based on experience stored in the DQN. The context of these calculations enables the solver to make risk-based decisions with more data than random guessing and are especially helpful when there are multiple possible right or wrong choices.

To integrate the benefits of the hybrid learning model, reinforcement learning is implemented using a deep Q network (DQN), represented by the Q-Network class. This class defines a TensorFlow--based neural network where the input layer receives the sub-state representation of the Minesweeper board, using a ReLU activation function to introduce non-linearity. The architecture also implements three hidden layers, the first using a Tanh function followed by two layers with linear activation functions. The final output layer uses a Tanh activation function to return a scalar Q-value which the DeepQLearner class uses to guide the agent's learning and action selection. The Q-value is a reward estimation value based on the input state and is one of the parameters that can be changed to influence performance. A replay buffer utilizing a deque data structure contains the transition information which includes the current sub-state, the chosen action, the reward, the sub-state after the action, and a Boolean to indicate the end of the episode. To mitigate learning within a narrow scope, the replay buffer pulls random past transitions, and this helps stabilize the network during training. To implement this replay buffer effectively, it must be checked for enough stored past transitions to increase the variance in the data pool.

Two calculations built into the DQN are the target network and the epsilon-decay schedule. The target network serves to re-baseline the learner and acts as a separate network independent of the

rapidly updating DQN. The weights of the target network are updated based on Polyak averaging which reduces the intensity of the impacting Q-values. Implementing a target network reduces noise caused by extreme values and provides more consistency in the training environment. This helps with smoothing the learning process and reduces the likelihood of settling on a bad policy or any policy too early in training. To encourage a better win ratio, a more extreme reward system was implemented to strongly influence the learner before the epsilon decayed to a constant value. While a high reward/high penalty system makes an impression, it also increases the volatility of the DQN's weights. The target network is used to mitigate subsequent shifting with periodic updates that allow for a learning gradient instead of spikes and plateaus. The Polyak averaging can be described as an exponential moving average method to update the target network weights based on the following formula [Equation 4]

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi..$$

Equation 4

```
# Polyak-averaging Target Network update
polyak_var = 0.001
q_weights = self.q_network.get_weights()
target_weights = self.target_network.get_weights()
new_target_weights = [polyak_var * q_w + (1 - polyak_var) * t_w
                      for q_w, t_w in zip(q_weights, target_weights)]
self.target_network.set_weights(new_target_weights)
```

Figure 3- Polyak averaging implementation within the target network

Once the weights are updated, the epsilon value is reduced via a decay schedule. This challenges the solver to reduce exploration and rely on experience as training progresses. The epsilon decay rate is set to 0.001 and updated in the train_step() method. This means that, for each step, the epsilon will reduce by 0.001 until it reaches a target value set in the parameters. The epsilon value for this solver starts at 1.0 and reduces on schedule until it reaches 0.05. The benefit of the epsilon reduction is the slow transition from experimenting with high-risk moves to relying on past moves that

give the trained policy context. The balance of exploration versus exploitation will influence the win ratio of the solver as reducing the epsilon too early can fortify sub-optimal policy but decaying it too slowly can increase the required training time and encourage more random actions over a longer period leading to a less stable learning process and higher variance in the performance metrics.

To assess the appropriate indicators of success, multiple values are stored and calculated for post-training analysis. The metrics are calculated and exported in real time to a .csv file for review and comparison. The captured metrics for this solver are total reward, number of squares revealed, epsilon value, win-loss ratio, total time to complete the episode, and average move time per episode. Additional data that is set in the parameters include the total number of episodes, grid size, and total number of mines, but since these are constant for the entire training session, they are not analyzed within the performance metrics but are instead considered contextually. Since the training rate can fluctuate within the entire training session, stratified metrics are captured to provide inter-milestone assessments. file for review and comparison. The captured metrics for this solver are total reward, number of squares revealed, epsilon value, win-loss ratio, total time to complete the episode, and average move time per episode. Additional data that is set in the parameters include the total number of episodes, grid size, and total number of mines, but since these are constant for the entire training session, they are not analyzed within the performance metrics but are instead considered contextually. Since the training rate can fluctuate within the entire training session, stratified metrics are captured to provide inter-milestone assessments.

The total reward is initially set at a value of 0.0 and is updated with each step. The total reward is calculated as a sum of the reward given for revealing safe cells, the final reward for achieving the winning state, and the negative penalty for hitting a mine or redundantly selecting a cell. Total reward is returned in the run_episode() function along with other important metrics. The reward value is important to monitor as it heavily influences the decisions the solver makes. If the rewards and

penalties are out of balance, the solver may not be able to discern a clear policy or assess risk appropriately. We can visualize improvement via our graphical representation of total reward over time. Initially, we anticipate the rewards will fluctuate heavily while the solver is training since it will be engaging in high-risk decisions and establishing an action policy. As training progresses, we anticipate that the reward value will stabilize towards the highest possible reward since the solver will be utilizing its optimized winning strategy.

The number of squares revealed is a valuable metric because it shows the progress the solver makes in clearing the board. As the solver establishes a good or bad policy, the number of revealed cells will proportionately increase or decrease. Additionally, since rewards are tied to revealing cells, if a high number of cells are consistently revealed but the win ratio does not increase, that could be an indicator that the reward for revealing cells is too low.

The epsilon value is tracked to see the relationship between training duration, exploration versus exploitation, and successful settling on a winning strategy. Trends to note in the epsilon decay rate are a steady decline over a significant portion of the training session and the identification of the plateau. Once the epsilon rate reaches its lowest value, exploitation will become the dominant strategy. If the win-loss ratio and number of revealed cells do not continue to increase after the plateau, it may be an indicator that the solver has settled on a sub-optimal policy. If the win-loss ratio and the number of revealed cells increase, this is a good indicator that the parameters for the solver are performing as expected.

The win-loss ratio is tracked to determine if the solver can achieve a winning state and continue to improve performance throughout the training session. An important data point to note is the rate of increase in the win ratio stratified throughout the training sessions. The win-loss ratio is anticipated to be low during initial exposure to the training environment, but we expect an increase in performance as the DQN gains the context fed from the CSP and MDP methods. We anticipate another capturable data

point when the epsilon plateaus, as that will be a pivotal moment in the win-loss ratio. From the epsilon plateau, the win-loss ratio trend for the rest of the training session will show the effectiveness of the learned strategy. An important data point to note is the rate of increase in the win ratio stratified throughout the training sessions. The win-loss ratio is anticipated to be low during initial exposure to the training environment, but we expect an increase in performance as the DQN gains the context that is fed from the CSP and MDP methods. We anticipate another capturable data point when the epsilon plateaus as that will be a pivotal moment in the win-loss ratio. From the epsilon plateau, the win-loss ratio trend for the rest of the training session will show the effectiveness of the learned strategy.

The average time per episode and average time per move per episode are tracked to showcase computational efficiency. The anticipated trend in total time to winning state should be a gradual decrease as our solver should become more efficient at solving the board over time. Longer episode times may be indicative of cautious decision-making, meaning, the solver lacks the context to decide or is unable to leverage an efficient policy that shortens the time required per move. As with the other metrics, the initial values may fluctuate since the strategy is in development. As training continues, we anticipate the values will steady to some degree. An indicator of success would be comparing shorter average times with the win ratio. If the move time decreases as the win ratio increases, this is indicative of a successful strategy. If the average time does not settle or decrease, and the win ratio does not continue to increase, the solver may be exploring when it should be exploiting or spending too much time in the decision process which would indicate a lack of context from the DQN. If the average time per move is spiking during or after the epsilon decay has reached a plateau, this indicates instability in the strategy and better reinforcement methods that provide consistency will need to be implemented (like a smoother target network and Q value implementation). to some degree. An indicator of success would be comparing shorter average times with the win ratio. If the move time decreases as the win ratio increases, this is indicative of a successful strategy. If the average time does not settle or decrease,

and the win ratio does not continue to increase, the solver may be exploring when it should be exploiting or spending too much time in the decision process which would indicate a lack of context from the DQN. If the average time per move is spiking during or after the epsilon decay has reached plateau, this indicates instability in the strategy and better reinforcement methods that provide consistency will need to be implemented (like a smoother target network and Q value implementation).

The testing methodology was organized to show performance over time and the effectiveness of our training model. The tests performed are outlined in Table 1. Comparable tests were run to indicate changes in performance that isolate a given parameter or combination of parameters. Tuning of the parameters was performed by leveraging successes in certain trials and implementing them in tandem with other efficiency methods to determine if a collaborative effort impacted the win-loss ratio and learning strategy more in coordination or if individual key performance indicators (KPIs) had a greater impact when tuned in isolation. Tuning considerations included stability and longevity, reward implementation, and changes in architecture.

Episodes	**Modification Performed**	**Tuning Consideration**	**Comparable Results**	**KPIs Effected**	**Solve Rate**
10,000	Extreme reward, Polyak averaging, target network implementation	Stability, longevity, reward implementation, architecture changes	Baseline solver with initial rewards, no polyak or target network	All (win-loss ratio, number of cells revealed, average time)	42.4%
4,000	4, 8, 16 mines in 10x10 grid	Weight adjustments depending on mine density	8 mines on 10x10 was default, 4 mines as lower, 16 mines as higher	All (win-loss ratio, number of cells revealed, average time)	74%, 22%, 0%
10,000	16x16 grid with 40 mines	Scalability of solve rate	Beginner board and solve rate	All (win-loss ratio, number of cells revealed, average time)	0%

Table 1: Tests performed on final model with different parameters to show impact on performance

To aid in the visualization of trend data, the metrics are represented visually in the metrics_visInteractive program. When run initially, interactive graphs populate and allow for analysis of

data points at milestones and visual cues by hovering over them. Once the graphs are closed, the snapshots are saved as .png files.

One major challenge we confronted early in testing was a drop in the win-loss ratio after a certain number of episodes, typically around 2,000-3,000 out of 5,000 episodes. Upon further research, we discovered this event is known as catastrophic forgetting. This led to the implementation of a target network and polyak averaging since we believed the plateau was derived from the solver settling on a sub-optimal policy early in the training session. We decreased the epsilon decay rate to expand the training session over more episodes and added stabilization through a target network. While we still experienced a plateau in the win-loss ratio, it did not drop or decrease at any point in the session, even when episode count was increased to 10,000. Additionally, to improve the win-loss ratio for shorter duration tests, increasing the extremity of the reward allowed for a stronger learning model and improved the ratio by 42.2%. Potential future changes that could encourage a higher solve rate are the refinement of the CSP solver probability assignments and more tuning of the target networks and epsilon decay rate. While there was improvement with the initial implementation, more tuning is needed to optimize the learning model.

IV. Experimental Results and Analysis

The testing environment was an average home machine running a processor of 2.90GHz or better. These machines did not use CUDA or have graphics enabled. The code was written in Python-supported editing software (VSCode) and configuration was managed in GitHub. The code was run from the command prompt and the metrics were output in the terminal while the code ran. After the program is completed, a separate initiation of the metrics_visInteractive program must be performed to see the interactive graphical outputs. Once the graphs are closed, the images are stored as .png files. .png files.

The hyperparameters selected for the final performance test are captured in Table 2.

# Episodes	10,000
Grid Size	10x10
# Mines	8
Win Reward	reward = 10.0 * (float(delta)/float(self.safe_cells))
Loss Penalty	-5.0
Redundant Move Penalty	-0.1
Revealed Cell Algorithm	reward = 10.0 * float(delta)/float(self.safe_cells)
Epsilon Decay	epsilon_decay=1e-3
Epsilon Vals	epsilon_start=1.0, epsilon_min=0.05
Polyak Avg Algorithm	polyak_var = 0.001 new_target_weights = [polyak_var * q_w + (1 - polyak_var) * t_w for q_w, t_w in zip(q_weights, target_weights)]

Q-Value Algorithm	for (r, c) in actions:
	sub = self.env.extract_sub_state(r, c)
	sub = np.expand_dims(sub, axis=0) # shape (1, sub_state_size, sub_state_size)
	q_val = self.q_network(sub).numpy()[0][0]
	if q_val > best_q:
	best_q = q_val
	best_action = (r, c)

Weights	self.alpha = 0.25*self.rows + 0.25*self.cols + 0.2*mine_ratio + 0.05
Target Update Frequency	1000
Batch Size	32

Table 2: Final test parameters

Initially, the solver was set with a penalty of -1.0 for a loss, and a reward of 2.0 for a win on 4,000 episodes. This resulted in a rolling win ratio of approximately 39%. When the reward increased to 10.0 and the penalty to -5.0, the rolling win ratio increased to 42.4%. This is one example of how manipulating the reward/penalty system can change the learning rate of the solver.

Comparatively, other AI solvers have accomplished different solve rates for their unique implementations. Other scholarly projects ranged from 50-92% [2,8] for the solve rate or accuracy reporting metrics. One study reported the average success rate for a human playing Minesweeper to be

around 35% on the beginner difficulty [12]. Our rolling win ratio would be considered better than the human average for that study, and our stratified peak performance increase broke the threshold of 50% so these are promising results.

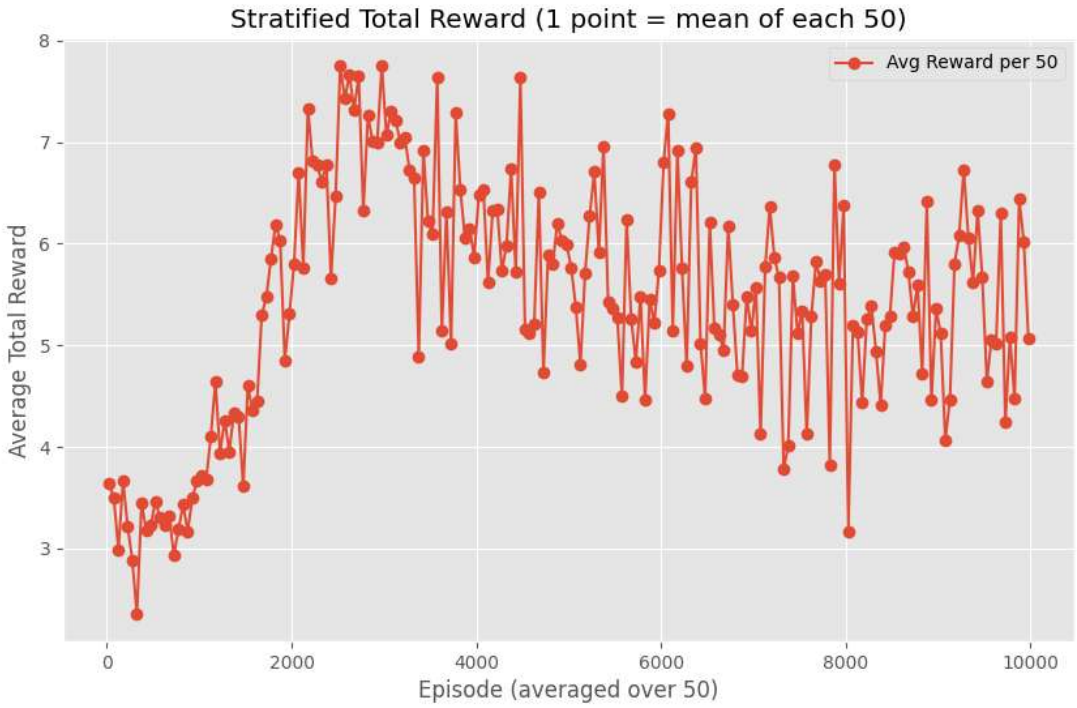

Figure 4: Stratified Total Reward

Total reward was graphed by the average reward per 50 episodes in Figure 4. Total reward was tracked for insight into how we can encourage learning via our reward structure. We also tracked stratified total revealed cells in Figure 5. Revealed cells tell us if our AI was able to increase the average of safe choices made over time. A trend is similar in both visuals, where there is a spike in performance around 1800-3500 episodes and an overall decrease in stratified performance after that. This can tell us if our reward structure was initiated properly to facilitate the best learning environment. Post-testing analysis could point to the disconnect in the first 2000 episodes between the reward given and the cells revealed. It appears that our initial reward for revealed cells was insufficient or at least not proportional and could have led to a decrease in performance later in the training environment.

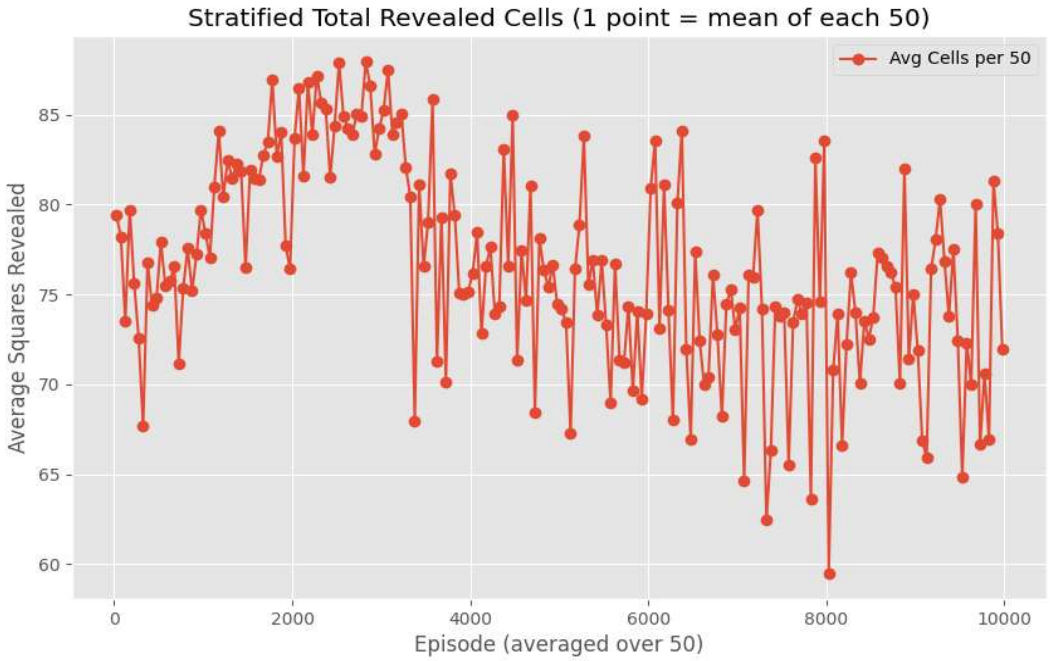

Figure 5: Stratified Total Revealed Cells

Figure 6 shows the rate of decay of our epsilon value over the entire training period. Our epsilon value started at 1.0 and decayed at a rate of 0.001 per action which gradually reached its lowest allowed value of 0.05 at approximately episode 1800. What this data tells us is that the gradual trade-off of experimentation for exploitation occurred over 1800 episodes and the policy was largely established once the epsilon reached its lowest value. We did not identify any anomalies in the decay rate. Our benchmark data point is approximately 1800 episodes for analysis on whether or not the training following this episode milestone settled on an optimal strategy. The training following this episode milestone settled on an optimal strategy.

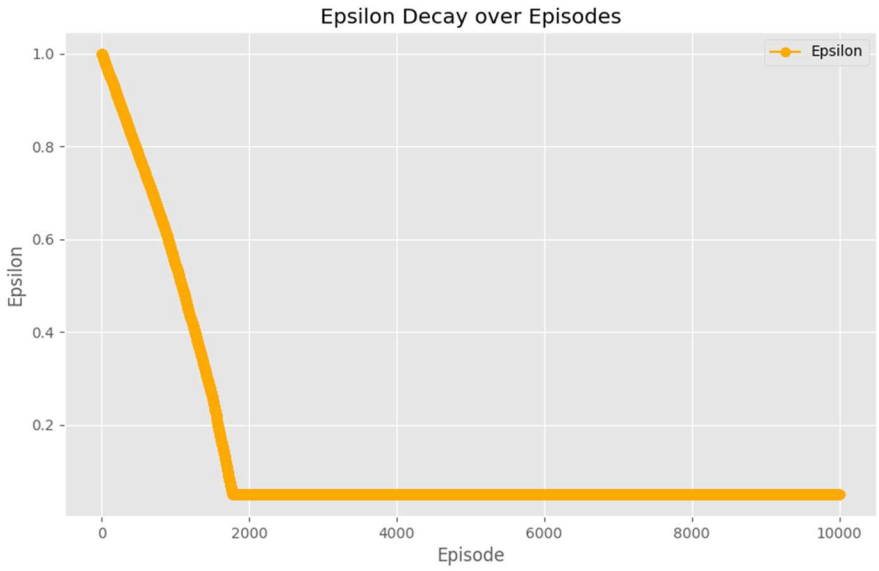

Figure 6: Epsilon Decay over Episodes

The win-loss ratio graph shows the performance improvement over the training period. Figure 7 shows the win ratio over episodes. The data follows a positive sigmoidal trend characterized by near-zero values initially, a rapid increase in performance, and a gradual plateau. We tuned for a longer period of performance increase so the trend is what we anticipated. The largest increase in performance occurred between episodes 1900 and 3500 with a stratified win ratio of 51.4%-53.4% but the rolling ratio was less as it included the initial learning period before the higher success rate. While the surge in performance is promising and could point to an optimal policy, we know that our final win ratio is not as high as comparable models performed by other research teams. The reasoning behind the plateau could be that the solver experienced diminishing returns after a period of training and therefore did not continue to improve the solve rate. Another possible explanation is that the solver settled on a policy that performs at a certain success rate and no amount of additional training will change the resulting solve rate. We could implement a longer exploration period or allow for milestone increases in the epsilon rate so it periodically has opportunities to explore more strategies that may mitigate the plateau.

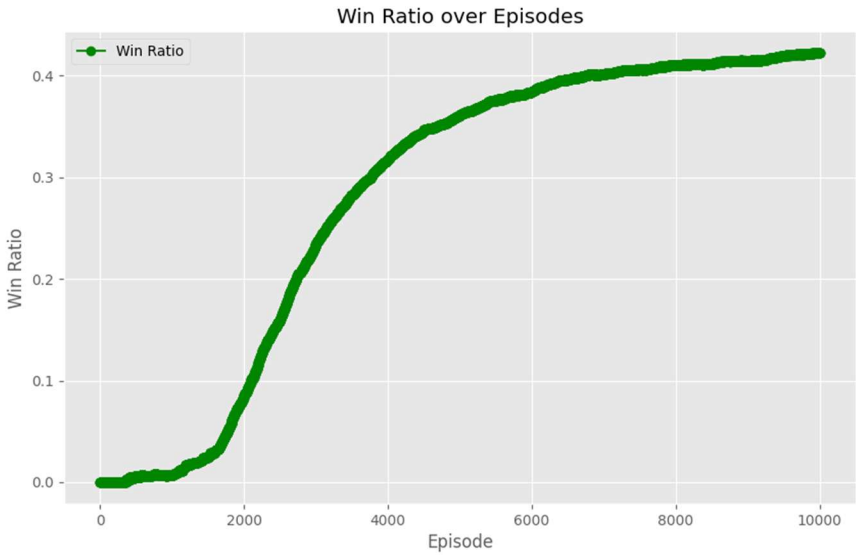

Figure 7: Win Ratio over Episodes

An important relationship to note is that between the rolling win ratio and the epsilon decay. Our epsilon decayed right before we noticed the steepest performance leap in our rolling win ratio. Initially, this was a promising indicator that our model had chosen an optimal policy to implement. Noting the intersection point of the two can be an indicator of how long the model should stay in training, as settling on a policy before the win ratio increases would be sub-optimal and allowing the model to continue training too long into a sustained win ratio could increase riskier moves that negatively impact the learning rate. Fluctuations in reward can create an unstable learning environment, which is something we mitigated with our target network, but is one of many tools to utilize during tuning. We removed the anomaly of catastrophic forgetting, so the overlay of this data is as anticipated.

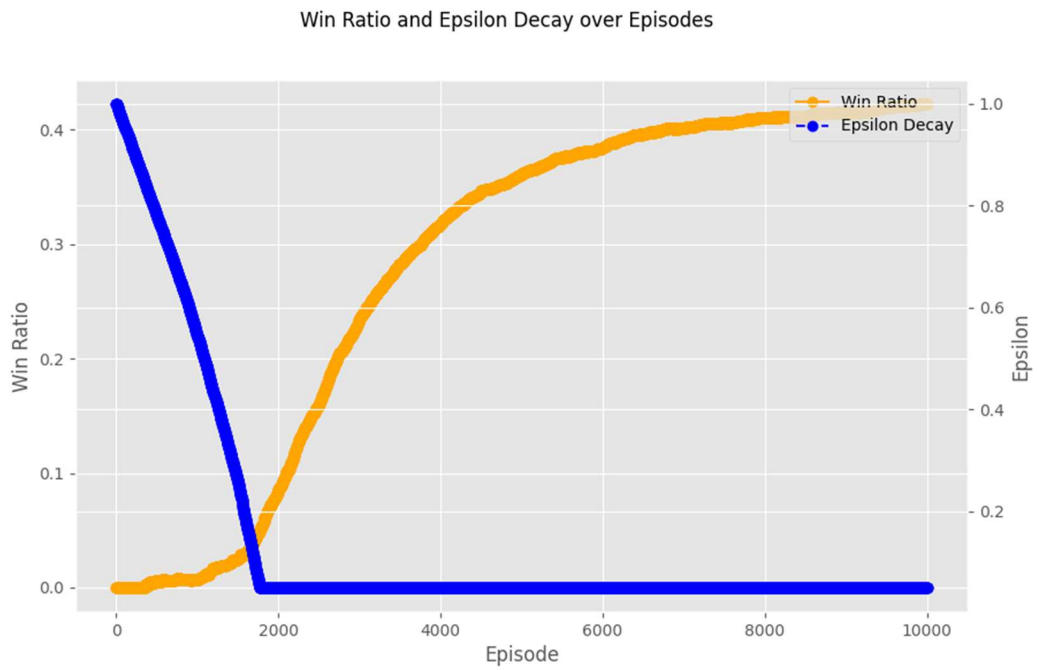

Figure 8: Win ratio and decay rate over episodes

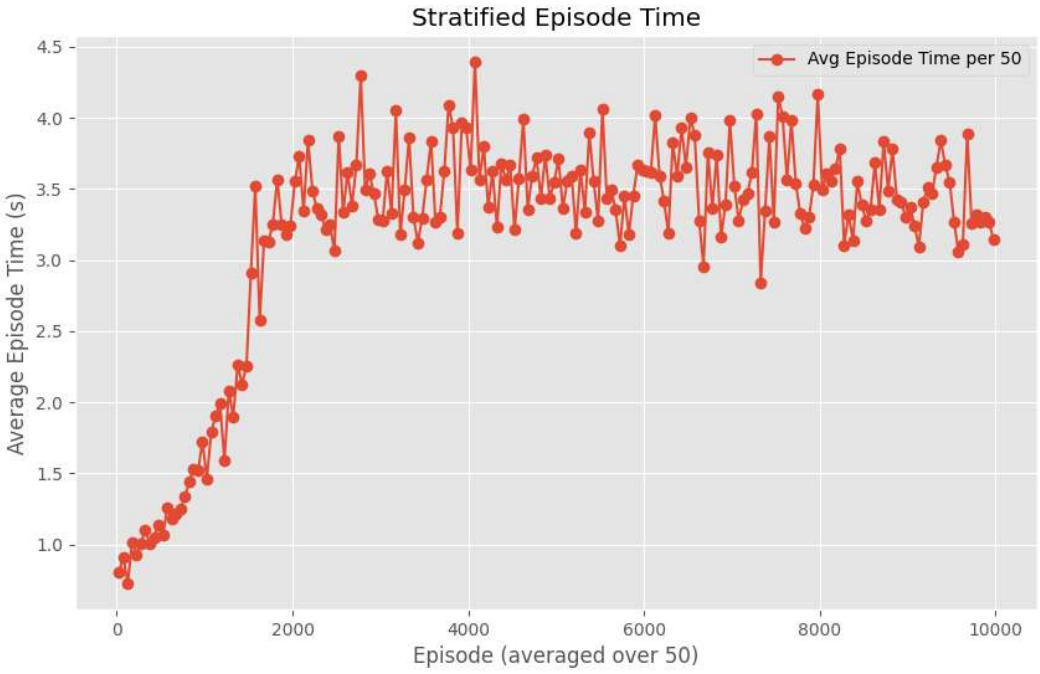

Figure 9: Stratified Episode Time

Average total episode time and average per move per episode analysis are shown in Figure 9 and 10, respectively. Average episode time and move time show the computational expense of these moves, and we anticipate that our AI will have a decreased decision time toward the end of training. Of note, the first 200 episodes show a steep rise from zero until settling near the average that is sustained and very slightly decreased for the remaining 8000 episodes. Those initial times were not evident of extreme computational efficiency but correlate with shorter episode times due to early losses and choices made with less context from the DQN. On the beginner board, computational efficiency was as expected, and the average times did not fluctuate wildly enough to assume our decision model was incapable of forming strategies. Ideally, computational efficiency would be around 200 ms, or 0.2 seconds, per move, so our AI has room for improvement [9, 10]. For CUDA-enabled GPUs or multi-core CPUs, the expectation is that the average would be less than 10 ms [9, 10]. Some ways we can improve our computational efficiency include caching probabilities instead of backtracking, moderating our use of DSScsp, and changing our batching multitasking.

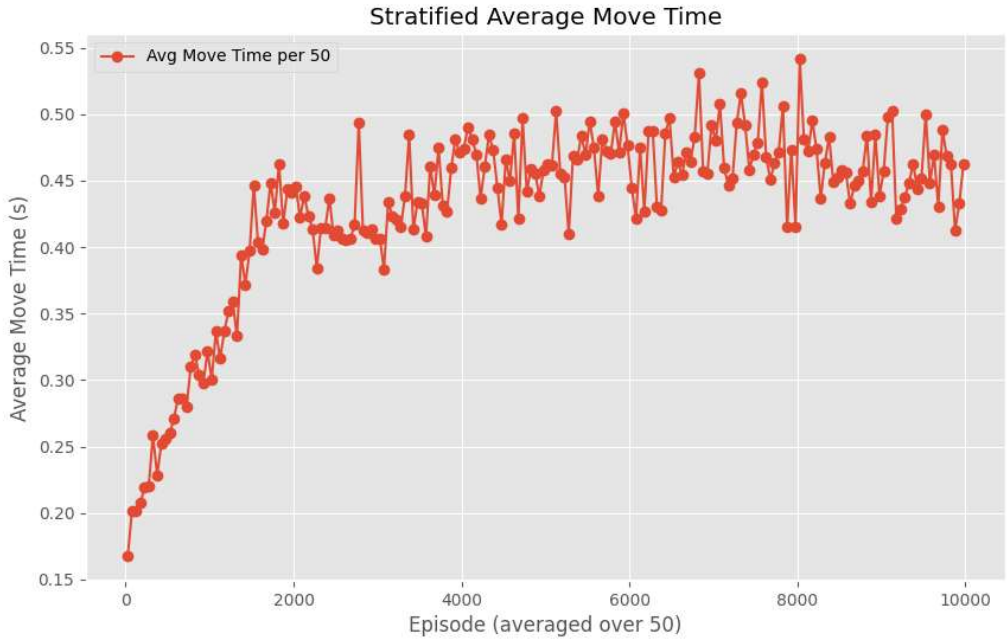

Figure 10: Stratified Average Move Time

V. Discussion, Conclusion, and Future Work

To evaluate the success of our AI solver for Minesweeper, we tested it on multiple trials with different parameters to see if it was able to play the game and achieve higher win ratios throughout training. Multiple KPIs were analyzed to check areas of improvement, and our AI solver increased in efficiency after our tuning efforts. The solver evaluated cell probabilities and interpreted the game states based on limited context and chose the cell with the best outcome that ultimately led to a winning state 42.4% of the time. The solver learned to make specific inferences over time on the hidden elements behind unrevealed cells and made safe decisions when uncertainty was present. To encourage the proper training environment, multiple variables were tuned including the epsilon decay rate, the positive and negative reinforcement values, and the baseline learning infrastructure was modified to support stronger positive policies. These modifications included the implementation of a target network and an epsilon decay schedule. The hybrid approach of CSP, MDP, and the DQN used in our AI Minesweeper solver enables advanced problem-solving to take place within the learned model which showed learning independent of a brute-force rule encoding. rate, the positive and negative reinforcement values, and the baseline learning infrastructure was modified to support stronger positive policies. These modifications included implementation of a target network and an epsilon decay schedule. The hybrid approach of CSP, MDP, and the DQN used in our AI Minesweeper solver enabled advanced problem solving to take place within the learned model which showed learning independent of a brute-force rule encoding.

The strength of our approach lies in the hybrid training methodologies that mitigate gaps in singular problem-solving algorithms. Our implementation of CSP to calculate logical choices, MDP to perform risk analysis and build expertise, and reinforcement learning within the DQN to enhance the effectivity of CSP and MDP, all contributed to a robust AI solver. While our AI solver did not reach

the solve rate of certain other AI solvers, tuning the parameters could bridge the gap in performance as the underlying structures between the programs are similar.

By manipulating KPIs and parameters, we were able to increase the performance of the solver which showed that the learning model is susceptible to influence with minor changes. This is a promising detail that shows that the learning environment we created impacts the solve rate directly. By correlating tuning to improved performance as well as improved performance throughout the training sessions, we showed that what we generated was an AI solver operating in a learning environment and not a random or brute-force application of a Minesweeper solver.

The AI solver operates explicitly in a Minesweeper problem environment, but expanding the methodologies has implications for modern problems. The infrastructure and hybrid learning model can be applied to any problem that has goal states of optimization paired with partial-observability problems that incorporate risk analysis and state/action interactions. implications for modern problems. The infrastructure and hybrid learning model can be applied to any problem that has goal states of optimization paired with partial-observability problems that incorporate risk analysis and state/action interactions.

VI. References

[1] Deep Deterministic Policy Gradient — Spinning Up documentation. (2014). Openai.com. https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[2] Gardea, L., Koontz, G., & Silva, R. (2015). Training a Minesweeper Solver. https://minesweepergame.com/school/training-a-minesweeper-solver-2015.pdf

[3] GenAI for Organizations [Review of GenAI for Organizations]. Deloitte; Deloitte. 2025. https://www2.deloitte.com/us/en/pages/consulting/solutions/generative-ai

[4] How To Play Minesweeper. (n.d.). Minesweepergame.com.

https://minesweepergame.com/strategy/how-to-play-minesweeper.php

[5] Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A., Yogamani, S., & Perez, P. (2021). Deep Reinforcement Learning for Autonomous Driving: A Survey. IEEE Transactions on Intelligent Transportation Systems, 23(6), 1–18. https://doi.org/10.1109/tits.2021.3054625

[6] Nguyen, T. T., & Reddi, V. J. (2021). Deep Reinforcement Learning for Cyber Security. IEEE Transactions on Neural Networks and Learning Systems, 34(8), 1–17. https://doi.org/10.1109/tnnls.2021.3121870

[7] Project, T. F. V. G. (n.d.). The History of Minesweeper. FreeMinesweeper.org. https://freeminesweeper.org/minesweeper-history.php

[8] Sajjad, M. (2022, November 30). Neural Network Learner for Minesweeper [Review of Neural Network Learner for Minesweeper]. Loughborough University Department of Computer Science. https://arxiv.org/pdf/2212.10446

[9] Sinha, Y., Malviya, P., & Nayak, R. (2021). Fast constraint satisfaction problem and learning-based algorithm for solving Minesweeper [Review of Fast constraint satisfaction problem and learning-based algorithm for solving Minesweeper].

[10] Solan, E., & Vieille, N. (2015). Stochastic games. Proceedings of the National Academy of Sciences, 112(45), 13743–13746. https://doi.org/10.1073/pnas.1513508112

[11] Walker, J. (2010). Minesweeper and Hypothetical Thinking Action Research & Pilot Study. https://files.eric.ed.gov/fulltext/ED509464.pdf

[12] Wu, X., Li, R., He, Z. et al (2023). A value-based deep reinforcement learning model with human expertise in optimal treatment of sepsis. npj Digit. Med. 6, 15. https://doi.org/10.1038/s41746-023-00755-5

Github: https://github.com/ekw5123/801-AI-game