Comparative Analysis of Reinforcement Learning Strategies for Minesweeper

Sawyer, Shane

ABSTRACT – This paper applies reinforcement learning techniques. to the game Minesweeper without aiming to use neural networks. Four different strategies were used over two different algorithms. Two of the strategies were able to achieve notable win rates after training. Furthermore, when more data was used in training, a strategy using multiple agents was able to compete with the performance of neural networks.

I. Introduction

THE game of Minesweeper has been a popular logic game for decades, providing players with the challenge of revealing hidden mines on a grid-based board without detonating any of the mines. Minesweeper requires strategic decision making and logical deduction. Since the board state begins with each space being covered, it presents a unique challenge to try and solve the board using reinforcement learning.

Minesweeper is played on a rectangular grid, where each cell either does or doesn't contain a mine. For clarification, Minesweeper is a digital game played on a device, and not a physical game with pieces. In order to win the game, a player must reveal all empty cells without revealing a mine. When an empty cell is uncovered, it will have a number indicating how many mines are adjacent to it. A mine is adjacent to a space if it resides within the spaces directly to the sides, or the diagonals, of the space. Below are three figures, which represent the initial board, what the board may look like after the first player interaction, and a solved board. The board used was a 9x9 grid with 10 mines. The flags are also placed by the player, but only serves as a tool for the player to remember what spaces he/she thinks to be a mine.

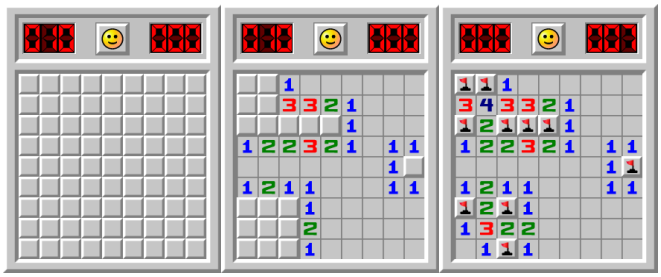

Figure 1: Minesweeper board in different points of gameplay

Reinforcement learning (RL) has gained significant attention as a powerful approach to machine learning, particularly in the domain of autonomous decision making. RL algorithms learn to make sequential decisions by interacting with an environment and receive feedback in the form of rewards. Understanding how different reinforcement learning algorithms perform in such environments improves our understanding of decision-making agents.

Complex unknown environments pose unique challenges for RL algorithms. These environments may have high dimensional states and uncertain reward structures. Furthermore, the true state of the environment may be obscured to the agent. Additionally, limited prior knowledge about the environment further complicates the learning process. Consequently, assessing the performance of different RL strategies in these scenarios can be attributed to a test of their effectiveness and reliability in real-world applications.

Minesweeper constitutes a complex unknown environment for RL. Additionally, it is proven to be NP-complete, which indicates it as a complex computing problem for any approach [1].

II. Reinforcement Learning

Reinforcement learning is a machine learning approach that enables an agent to learn optimal decisions by interacting with an environment and receiving feedback in the form of rewards. At its core, RL involves an agent, environment, states, actions, rewards, and a learning process [2]. The agent is given the state of the environment, from which the agent chooses an action based on that state. The environment is then updated according to the agent's action, transitioning into a new state, and a reward is produced and given as feedback to the agent. The agent's goal is to learn the optimal actions to take in the environment in order to maximize the reward it earns.

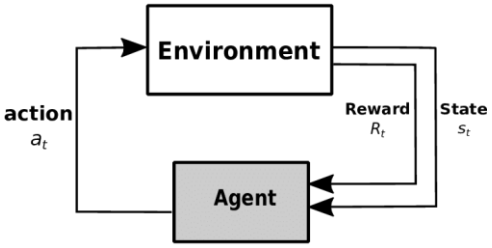

Figure 2: Diagram of RL cycle [3].

This paper uses the formalization of RL as a Markov Decision Process, which is defined as a tuple (S, A, P, R, γ) [2].

- S: The set of possible states of the environment.
- A: The set of possible actions the agent can make.
- P: The state transition probability function, or the probability of one state transitioning to another.
- R: The reward function.
- γ: The discount factor, a hyperparameter that determines the importance of future rewards.

Now with this formal definition, it is easier to see and define how complex Minesweeper can be for reinforcement learning. Firstly, the set of possible states for Minesweeper can become very large. This paper evaluates performance on a 9x9 grid with 10 mines. The following is an equation that is an estimation of the number of possible states:

$$|S| \approx \frac{81!}{71!\,10!} \cdot 2^{71} = 4.43522867 \times 10^{33}$$

The first term represents the number of ways we can place 10 mines on the 9x9 board, and the second term represents that each space, besides the mines, can either be covered or uncovered. Furthermore, the possible actions that can be taken by the agent contributes to the complexity. The agent will not be allowed to place flags, but only attempt to uncover each unique space. For each of the possible states, the agent needs to learn which of the 81 locations to uncover for optimality. Thus:

$$|A| = 81$$

For the purposes of this paper, the state transition probability function won't be prevalent in the used algorithms, which will be introduced shortly. The reward function varies depending on the environment and can heavily influence the behavior of the agent. This is because the agent's sole purpose in reinforcement learning is to maximize the rewards earned. For Minesweeper, this is the reward function used in this paper:

$$R(s, win) = 1000$$
$$R(s, lose) = -100$$
$$R(s, uncover) = 5$$
$$R(s, repeat) = -1$$

As seen by the reward function, the agent is rewarded for winning the game, or uncovering a space. The agent is penalized for uncovering a mine, or attempting to uncover a space that has already been uncovered, denoted by "repeat."

III. Challenges

There are various challenges that will need to be addressed in applying RL to Minesweeper. The first challenge is managing exploration versus exploitation. Exploration refers to how the agent randomly interacts with the environment. Since the agent aims to maximize the rewards it earns, it typically will always choose the action it believes is optimal. Consequently, the agent repeatedly chooses the best action, which is exploitation. At the beginning of training, an RL agent typically begins with a high chance to choose random actions. This random chance will be decreased over time, to allow the agent to exploit more. In Minesweeper, having any random chance can be detrimental to the performance of the agent, as it could randomly choose a mine. To address this, the exploration versus exploitation strategy will need to be handled carefully to allow the agent to explore without randomly choosing a mine.

Secondly, the Minesweeper board consists of sparse rewards. Most cells on the board are empty, and the agent will typically begin by repeatedly trying to uncover cells that are already revealed. At the beginning of training, the agent has not yet learned much about each choice to be made on the board, and very few actions may be taken which result in the agent earning a positive reward rather than being penalized. Mines are rather infrequent, but winning the game is even more infrequent. Thus, the agent will encounter few rewards contributing to the challenge of applying RL to Minesweeper.

IV. Strategies

A. Q-learning

Q-learning is a fundamental RL algorithm that uses a value-based approach. It learns an action-value function to estimate the expected reward for each state-action pair [4]. Q-learning typically employs a strategy that varies between exploration and exploitation, which was discussed in the challenges section. It is known for its simplicity and ability to converge to an optimal policy with discrete state and action spaces [5].

More specifically, Q-learning involves learning an action value function, referred to as the Q-value function. Each Q-value represents the expected reward of taking a specific action and subsequently acting optimally. The algorithm iteratively updates its Q-values using something called the Bellman equation [6]. Furthermore, Q-learning uses the exploration and exploitation strategy. Denoted by "epsilon," the agent will act randomly with probability "epsilon" which is exploration, otherwise the agent will choose the optimal Q-value, or exploration. The Q-values are updated with the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (r + \gamma * \max_{a'}(Q(s',a') - Q(s,a))$$

In this equation Q(s, a) represents the Q-values for a state s and action a, r is the reward, alpha is the learning rate, gamma is the discount factor, and s' and a' represent the optimal

future action state pair. This equation will be applied using three different strategies.

Strategy #1, Normal

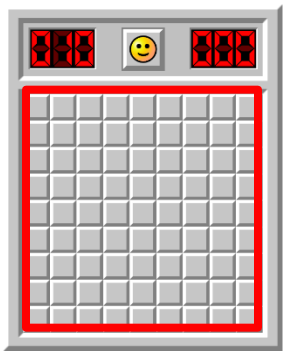

Figure 3: Board showing observation for normal Q-learning.

In this strategy, the state, or the observation given to the agent of the environment, will be the entire 9x9 board. This will include the same state space, action space, and reward function as described earlier. Ideally, this would allow the agent to take the best action possible as it is able to observe the entire board. However, to be covered in detail later, this strategy paired with Q-learning fails to solve randomly generated 9x9 boards due to the extremely high state space.

Strategy #2, Sliding Window

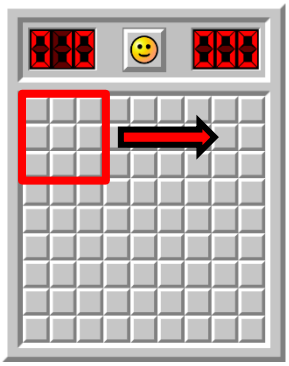

Figure 4: Board showing observation for sliding window.

This new strategy provides the agent with a much smaller observation of the environment. A 3x3 window starts in the top left corner of the board, the observation of this 3x3 window is given to the agent, in which the action is chosen and applied to the environment. Before moving the window, the agent is given the next state of the environment and the rewards it received. This strategy aims to reduce the state space and the action space. The new sizes of the state space and action space are as follows:

$$|S| \approx \sum_{n=0}^{9} \frac{9!}{n!\,(9-n)!} \cdot 2^{9-n} = 19683$$
$$|A| = 10$$

You may notice that the action space is 10, rather than 9 as would be expected on a 3x3 grid. In order to allow the agent to play the game without being forced to decide on each 3x3 section, a new action called “no operation” or “noop” is added. In the case that the agent cannot make an informed decision on the small observation, it is allowed to choose to do nothing. A small penalty is given to ensure the agent will not continuously choose to not act and instead seek out to uncover spaces. In order to improve the optimization of rewards, the reward function of the environment is altered to:

$$R(s, win) = 5$$
$$R(s, lose) = -100$$
$$R(s, uncover) = 5$$
$$R(s, repeat) = -1$$
$$R(s, noop) = -0.1$$

A large reward for winning is no longer given, because it would violate the properties of the MDP. For example, the agent may get an identical observation of a 3x3 area. In case 1, all other possible 3x3 sections are already cleared, the agent only has one last decision to make. In case 2, the 3x3 section is identical but the other possible 3x3 sections may remain uncleared. If the agent was still rewarded 1000 for winning, it would cause inconsistent rewards to be given, because the same action would produce a reward of 1000 in case 1, but a reward of only 5 in case 2 despite the observations being identical.

Strategy #3, Multi-Agent

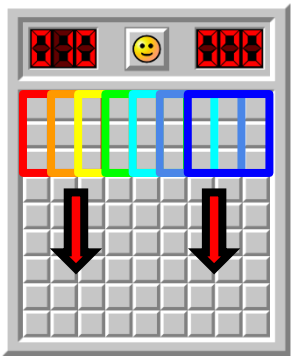

Figure 5: Board showing observation for each agent.

The third and final strategy used for Q-learning involves an identical state and action space to the second strategy. However, instead of moving a window around for a single agent, this strategy involves creating 49 individual agents for every unique 3x3 section of the board. This is a strategy known as multi agent Q-learning [7]. The state and action spaces are identical because of the 3x3 grid observation, and that each agent still has the option to not make an action. Additionally, the reward function remains identical.

During training, one agent is chosen randomly from the 49 agents to make an action. This ensures that each agent has an equal opportunity to learn and act on the board, rather than following a fixed order that may result in some agents getting less opportunities from other agents during training.

B. SARSA

The second algorithm that will be used is known as State-action-reward-state-action (SARSA). It is very similar to Q-learning but has the distinct difference that it is on-policy rather than off-policy [8]. It updates the policy, or the Q-values based on the actions taken. This is different from Q-learning, which updated its values on the maximum expected Q-value for future actions, rather than actions taken. The algorithm is very similar:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (r + \gamma * (Q(s',a') - Q(s,a))$$

The only difference in the algorithm is that the future Q-values Q(s', a') is the actual state and action pair of the environment based on the changes made by (s, a).

The method of exploring is different, however. Q-learning randomly chooses when to explore whereas SARSA initializes the Q-values to a low value, to encourage exploration [2]. For the purposes of this paper, the state representation, action space, and reward function will remain identical to that of Strategy #1 used for Q-learning.

However, due to being an on-policy algorithm, SARSA will not be able to be implemented into the Strategy #2 and Strategy #3 used in Q-learning. The sliding window strategy will not work because SARSA learns off the actual best next action, which is disrupted or inconsistent due to the moving observation space. Similarly, in the third strategy there is an overlap between the agents' observation spaces. Another agent acting within the space of a different agent will not disrupt Q-learning but will disrupt SARSA from choosing the correct actual action.

V. Analysis

The first form of analysis will use the same conditions for each strategy. The environment will be identical for each strategy, besides the changes to the reward function discussed for the second and third Q-learning strategies. The board will be 9x9 with 10 mines randomly placed throughout the board. Each strategy will be trained in over 10,000 different episodes with 5 different random boards. One episode is the agent playing on one board until it wins or loses.

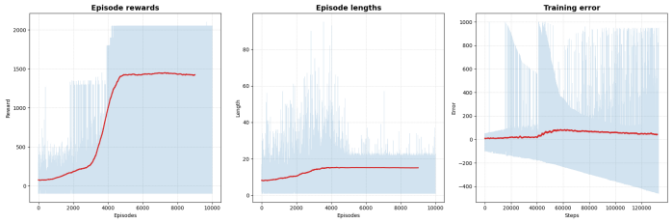

Figure 6: Episodes rewards, length, and training error during training for normal Q-learning strategy.

The three graphs represent the data of how the agent performed and interacted with the environment. The first graph shows the total sum of rewards earned for each episode. Over the 10,000 episodes it can clearly be seen that the agent was able to earn higher rewards before reaching a plateau. The second graph shows the episode lengths, or how many moves the agent made before it won or lost. Over each episode, the length tends to increase because the agent begins to learn to avoid the lines. The blue lines represent the raw data and not the average, so at the beginning there are certain episodes that become lengthy. This is because the agent takes time to learn that trying to uncover spaces that are already uncovered leads to a loss in rewards. The training error represents the difference between the expected reward of the action and the actual reward earned. The high spikes where the error rises to 1000 is around when the agent begins to win games, and the expected reward at the state is 0, which was the initial Q-value. Once the agent begins winning, the average error rises slightly before gradually decreasing to 0.

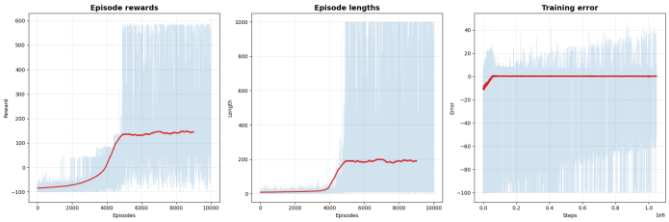

Figure 7: Episodes rewards, length, and training error during training for sliding window Q-learning strategy.

Due to the different observations given to the agent the data in these graphs differ slightly. Like the first strategy for Q-learning, the episode rewards clearly rise and plateau, showing the agent has learned to increase the earned rewards. One difference is that the episode lengths rise much higher. This shows how the agent decides to not make an action many times, each time it doesn't make an action counts as a step in the episode. The reason why the longest episodes only reach 1000 steps is because the environment is forcibly ended if the agent isn't making progress. If you compare the red line in the episode length to the previous strategy, it averages at around 200 rather than below 20. This is because the sliding window strategy typically forces the agent to not choose to make an action as the window moves, resulting in much longer episodes.

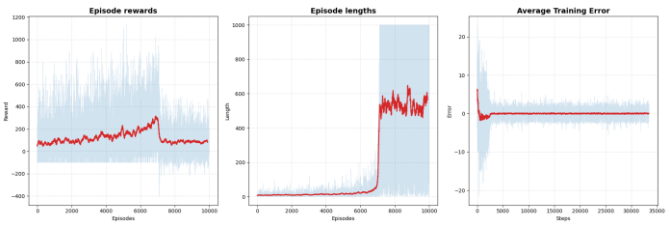

Figure 8: Episodes rewards, length, and training error during training for multi-agent Q-learning strategy.

The graphs for this strategy vary greatly from the previous two. Due to the nature of having 49 individual agents and each one being chosen randomly; it produces the challenge of ensuring that the agents can interact with the environment without randomly choosing a mine. As a result, the exploration versus exploitation strategy is varied to heavily favor exploitation over exploration. This can easily be seen in the graph for the episode lengths, which experiences a much larger rise in the lengths. By heavily restricting the exploration strategy the episode lengths become much longer as the agents don't randomly detonate mines.

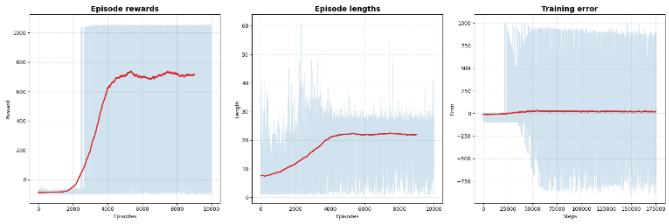

Figure 9: Episodes rewards, length, and training error during training for SARSA.

The final algorithm I will be going over in this analysis section is the results from the SARSA training. By simply comparing the shape of the graphs, the results seem the most like the normal Q-learning strategy. As was discussed in the Strategies section, SARSA is very similar to Q-learning. The episode rewards noticeably start at a much lower value. Compared to the normal Q-learning strategy, SARSA takes a longer time to begin earning any rewards in the environment. However, once SARSA begins earning rewards it can learn at a slightly faster rate than Q-learning. Furthermore, looking at the raw data in blue, the SARSA agent can begin winning games much earlier than the Q-learning agent.

The results from this show that the SARSA agent can learn quickly off less data. While this is not an issue with the Minesweeper game, it may be more applicable in real world applications. For example, in a situation where a robot learns to flip pancakes in the real physical world [9]. Although the robot can technically flip pancakes for as long as it wants, the time to physically perform the action may take long. If one were to have to choose between Q-learning or SARSA for this situation, it would be more suitable to choose SARSA. This would allow the robot to learn faster off less data, causing it to need less flips overall.

VI. Larger Scale

The previous analysis only used 5 different game boards over a rather small number of episodes. However, it shows insights into how the agents learn and interact with the environment.

In order to properly evaluate their efficacy to win games of Minesweeper, we trained the agents on a much larger scale, only using randomly generated boards. For those more familiar with the intricacies of Minesweeper and board cycles [10], this large-scale test did not use board cycles. Each board was completely randomized. The following table represents the win rate of each strategy after being trained on 1,000,000 random boards, and then tested further on another 10,000 boards once training finished.

Agent Type	**Win Rate**
Normal	0.00%
SARSA	0.00%
Sliding Window	1.53%
Multi-Agent	4.18%

Figure 10: Win rate for each agent type over 10,000 games after training

The normal Q-learning approach, as expected, exhibited a win rate of 0.00%. Given the vast number of possible Minesweeper games, it is highly improbable that any of the training boards coincided with those encountered during the final 10,000 games. Furthermore, even if the same board did arise during training, the normal agent likely failed to achieve victory while playing it.

In contrast, both the sliding window and decentralized methods demonstrated an ability to win some games after training. This success can be attributed to the fact that their state representations only encompassed 3x3 grids. The multi-agent approach appeared to be more effective in solving Minesweeper boards. This could be attributed to the fact that each agent remained static in its designated 3x3 area. Agents positioned in the corners exhibited less overlap with other agents, thus enabling them to learn and make more informed decisions. On the other hand, the sliding window method encountered limitations in making contextually nuanced decisions based on its relative position within the Minesweeper board, such as whether it occupied a corner, edge, or central region.

The findings indicate that the localized nature of the sliding window and multi-agent methods facilitated improved performance in Minesweeper. By focusing on smaller, more manageable regions of the board, these approaches allowed the agents to develop strategies that were better suited to specific localized contexts. Consequently, the multi-agent method, which emphasized independent learning within designated 3x3 areas, exhibited greater adaptability and effectiveness in achieving successful outcomes.

It is worth noting that the win rates achieved by both the sliding window and decentralized methods are relatively low. This can be attributed to the inherent complexity and uncertainty of the Minesweeper game, even when employing Q-learning techniques. Nonetheless, the results highlight the potential of tailored Q-learning approaches in tackling large state space environments, such as Minesweeper, while offering valuable insights into the interplay between state representation and decision-making capabilities.

Other papers that have explored the comparative performance of machine learning algorithms typically use the time to train as a metric [11], [12]. However, they do not provide enough justification as to why this is a valid measure of

performance. While it provides some insight into the time required to train a model, it is susceptible to various factors that can render it inaccurate. For example, the performance of a computer can vary depending on other tasks running concurrently, system load, available resources, and hardware specifications.

I present a table of time to train for my methods, but to ensure the information can be trusted I trained the agents in a sandbox environment on my own computer. Each agent was given 4 cores on an Intel Core i9-9900K CPU clocked at 3.60 GHz. Furthermore, each agent was given 16GB of memory. The training does not nearly use this much memory, but it is to ensure the agent is not restricted by hardware or other processes that may be running on my system.

Agent Type	**Time To Train (hh:mm:ss)**
Normal	00:15:43
SARSA	00:31:06
Sliding Window	02:34:21
Multi-Agent	03:45:53

Figure 11: The time it took to train each agent type over 1,000,000 games.

It's worth noting that the time to train could be improved via software optimization and hardware acceleration. The presented times should only be compared to each other relatively, rather than accepted as the time to train on all systems.

Considering the previous note, the normal approach or the first strategy took much less time to train. This is because the agent always has the entire board as the observation and can always make an action anywhere. Regardless, the normal Q-learning agent was unable to achieve a single win after training. On the other hand, the other two strategies took much longer to train. This is because the episode lengths are much longer as most actions on the environment are the agent/agents choosing to not make any action.

Finally, the second notable times to compare are between SARSA and the normal Q-learning strategy. The main difference in the algorithms is that the SARSA agent learns based off the actual best next action. In contrast, the Q-learning agent assumes it will act optimally in future decisions. Due to this, the SARSA agent takes a longer time to train over all the games. In the analysis section, it was seen that the SARSA agent was able to learn quickly off less data. This shows the tradeoff between using SARSA or Q-learning. In cases where you may have less data or have a limited number of times to act upon an environment, it would be better to use SARSA.

VII. Neural Networks

Although I will refrain from going into detail on the complexities of how neural networks work, it is worth mentioning that other projects exist where reinforcement learning was applied to Minesweeper [13], [14]. In these projects, these GitHub users applied deep learning techniques using neural networks to the game Minesweeper. In the first project [13], the user used the same sized board as mine, along with the same number of mines. Since their code is open source, I was able to download their code and run it against my own environment to see the results. After training on 1,000,000 games in the same environment as discussed in the analysis section and running on a further 10,000 games, I was able to achieve better results than my own. Their agent was able to win 9.05% of games played after training. However, the time to train on my own system was longer than 24 hours. Similarly, applying the code written in the second project [14], I was able to achieve a win rate of 10.01% over a training time longer than 24 hours.

Although the win rates may be impressive compared to my own tactics which do not use neural networks, the training time is much too long to use on my own system. In a similar test, I attempted to train my Multi-Agent strategy over 10,000,000 games in order to match the longer training time of neural networks. In the end, my Multi-Agent strategy was able to achieve a win rate of 8.78%.

Despite taking 10x the amount of data, my Multi-Agent strategy was able to achieve an increased win rate and compare to the strategies of neural networks. This shows that although neural networks can learn patterns and solve games with a smaller amount of data, the computations required to do so take a long time. As a result, applying that same amount of training time but using more data to a simpler agent can result in a similar win rate.

VIII. Conclusion

In conclusion, employing strategies that simplify an agent's perception of the environment, such as in the context of Minesweeper, has demonstrated the ability of Q-learning methods to solve boards effectively without resorting to deep learning techniques. By reducing the problem to the smallest possible state representation, such as a 3x3 grid, it becomes feasible to make informed decisions within the game.

The findings of this research shed light on the potential of Q-learning to handle large state space environments while circumventing the complexities associated with deep neural networks. By leveraging localized observations and limiting the agent's focus to a smaller region of the board, it becomes possible to distill the problem into a more manageable and comprehensible form.

The use of a sliding window or multi-agent approach in Q-learning enabled agents to develop strategies that were tailored to specific localized contexts within Minesweeper. This approach fostered a deeper understanding of the game dynamics within the limited scope of the agent's perception, ultimately leading to improved performance.

The ability of Q-learning to tackle Minesweeper, even in its reduced state representation, highlights the adaptability and effectiveness of this reinforcement learning technique. By leveraging temporal difference learning and value iteration principles, Q-learning demonstrates its capability to navigate complex environments and make informed decisions based on limited observations.

Future research that focuses on reinforcement learning without using deep learning or neural networks can focus on making logical inferences and simplifications to the environment to reduce the state space. This could reliably increase the performance of the agents on the environment, with the tradeoff being an increase in training time. The projects used in [13] and [14] rely on the pattern learning of neural networks but could likely see an increase in win rates if forced to make logical decisions on subdivisions of the environment.

IX. References

[1] R. Kaye, "Minesweeper is NP-complete," Math. Intell., vol. 22, no. 2, p. 9, Spring 2000, doi: 10.1007/BF03025367.

[2] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge: The MIT Press, 1998. Accessed: Jun. 03, 2023. [Online]. Available: https://muse.jhu.edu/pub/6/oa_monograph/book/60836

[3] R. Amiri, H. Mehrpouyan, L. Fridman, R. Mallik, A. Nallanathan, and D. Matolak, "A Machine Learning Approach for Power Allocation in HetNets Considering QoS," Mar. 2018.

[4] C. Watkins, "Learning From Delayed Rewards," Jan. 1989.

[5] C. J. C. H. Watkins and P. Dayan, "Q-learning," Mach. Learn., vol. 8, no. 3, pp. 279–292, May 1992, doi: 10.1007/BF00992698.

[6] T. G. Dietterich, "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," arXiv.org, May 21, 1999. https://arxiv.org/abs/cs/9905014v1 (accessed Jun. 04, 2023).

[7] S. Sen, M. Sekaran, and J. Hale, "Learning to coordinate without sharing information," in Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence, in AAAI'94. Seattle, Washington: AAAI Press, Aug. 1994, pp. 426–431.

[8] G. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," Tech. Rep. CUEDF-INFENGTR 166, Nov. 1994.

[9] P. Kormushev, S. Calinon, and D. Caldwell, "Reinforcement Learning in Robotics: Applications and Real-World Challenges," Robotics, vol. 2, pp. 122–148, Sep. 2013, doi: 10.3390/robotics2030122.

[10] "Board Cycles - MinesweeperWiki." http://www.minesweeper.info/wiki/Board_Cycles (accessed Jun. 05, 2023).

[11] N. Gholizadeh, N. Kazemi, and P. Musilek, "A Comparative Study of Reinforcement Learning Algorithms for Distribution Network Reconfiguration With Deep Q-Learning-Based Action Sampling," IEEE Access, vol. 11, pp. 13714–13723, 2023, doi: 10.1109/ACCESS.2023.3243549.

[12] J. Ngarambe, A. Irakoze, G. Y. Yun, and G. Kim, "Comparative Performance of Machine Learning Algorithms in the Prediction of Indoor Daylight Illuminances," Sustainability, vol. 12, no. 11, Art. no. 11, Jan. 2020, doi: 10.3390/su12114471.

[13] S. Lee, "sdlee94/Minesweeper-AI-Reinforcement-Learning." May 25, 2023. Accessed: Jun. 13, 2023. [Online]. Available: https://github.com/sdlee94/Minesweeper-AI-Reinforcement-Learning

[14] J. Hansen, "Minesweeper Solver - Using Deep Reinforcement Learning." May 25, 2023. Accessed: Jun. 03, 2023. [Online]. Available: https://github.com/jakejhansen/minesweeper_solver