Navigating the Maze of Reinforcement Learning: A Journey through Tic-tac-toe Dimensions

1st Talha Ahmed
Mathematics Department, School of Science and Engineering
Lahore University of Management Sciences
Lahore, Pakistan
24100033@lums.edu.pk

2nd Nehal Ahmed Shaikh
Department of Economics, School of Humanities and Social Sciences
Lahore University of Management Sciences
Lahore, Pakistan
24020001@lums.edu.pk

***Abstract*—This report delves into an explorative journey in the realm of reinforcement learning (RL), specifically applied to the classic game of tic-tac-toe in various dimensions. Our project was structured in three distinct phases, each higher in complexity and dimensionality. The first phase involved applying Value Iteration to a standard 2D 3 $\times$ 3 tic-tac-toe grid, a foundational step that allowed us to grasp the basics of RL in a controlled environment. In the second phase, we scaled our approach to a 2D 4 $\times$ 4 tic-tac-toe, utilizing Q-learning to tackle the enhanced complexity. The final and most challenging phase saw us confronting a 3D 4 $\times$ 4 $\times$ 4 tic-tac-toe, employing an RL algorithm of our choice to navigate this intricate landscape.**

In this report, we candidly recount our journey, detailing the plethora of challenges, setbacks, and breakthroughs encountered along the way. It includes discussions of the strategies we considered, the ones we adopted, and those we discarded. The report is also an honest reflection on our failed attempts and the invaluable lessons learned from them. By sharing our exhaustive research process, experimentation, and methodical approach to problem-solving, we aim to provide insights into the practical applications of RL in progressively complex scenarios.

Keywords — Tic-tac-toe, Value Iteration, Online Learning, Q-Learning, DQN

I. Introduction

The field of artificial intelligence (AI) has witnessed remarkable advancements in recent years, with RL emerging as a pivotal area of study and application. RL, characterized by its ability to learn optimal behaviors through trial-and-error interactions with an environment, offers profound implications for both theoretical understanding and practical applications in AI. This report presents a comprehensive account of our foray into the application of RL techniques, using the universally recognized and seemingly simple game of tic-tac-toe as our testing ground.

Tic-tac-toe, a game traditionally played on a 2D grid, serves as an ideal platform for exploring the fundamentals of RL due to its well-defined rules and finite number of actions and states. However, to fully harness the potential of RL, we extended the game's complexity across three phases: from a standard 2D 3 $\times$ 3 grid to a 2D 4 $\times$ 4 one, and finally to a 3D 4 $\times$ 4 $\times$ 4 version. This division allowed us to progressively build our understanding and expertise in the field.

We began with the implementation of Value Iteration in the standard 3 $\times$ 3 grid to obtain a framework that can be later modified to work on variants of the game. We then advanced to the 4 $\times$ 4 case, employing Q-learning to address the new challenges, particularly those regarding memory constraints. The culmination of our project was the final phase, where we tackled the most complex scenario: a 3D 4 $\times$ 4 $\times$ 4 tic-tac-toe. Here, we had the freedom to select an RL algorithm of our choice, pushing us to innovate and experiment beyond the conventional frameworks.

Our report is not just a technical exposition of RL algorithms and their applications. It is a narrative that encapsulates the entire spectrum of our experience — the initial hypotheses, the iterative process of trial and error, the setbacks and breakthroughs, and the critical insights gained. We provide an honest reflection of our learning curve, highlighting both our successes and the obstacles we encountered. Through this report, we hope to contribute to the broader discourse on the application of AI and machine learning, particularly in the context of games, which continue to serve as effective platforms for exploring and advancing these technologies.

Outline Section II delves into the mechanics and nuances of tic-tac-toe, with a particular focus on the 3D version and its underlying principles. In Section III, we explore *Phase 1 - 2D Tic-tac-toe (3 $\times$ 3)*, detailing the approach, findings, and significant insights gained. Following this, sections IV and V examine *Phase 2 - 2D Tic-tac-toe (4 $\times$ 4)* and *Phase 3 - 3D Tic-tac-toe (4 $\times$ 4 $\times$ 4)* respectively, each with its methodology and outcome. The report culminates in Section VI with our concluding observations and reflections.

II. Understanding the Game of Tic-tac-toe

Tic-tac-toe, a classic zero-sum game, is a simple yet intriguing challenge that has been studied extensively in game theory and AI. In its basic form, the game consists of two players

taking turns to place their marks (traditionally 'X' and 'O') on a grid. The objective is to align a straight line of consecutive marks, either horizontally, vertically, or diagonally. The game ends in a win for the player who achieves this alignment first and a draw if all spaces are filled without an alignment.

A. The 3D Variant

In the 3D variant of tic-tac-toe, the game extends to a multi-layered grid ($4 \times 4 \times 4$ in our case). Here, winning lines can be formed in three dimensions, providing a vastly increased number of winning combinations and significantly elevating the game's complexity.

B. Representation of Players' Moves

In our study, players' moves are represented numerically for computational ease. In Phases 1 and 2 (2D variants), we use '2' for Player 2, '1' for Player 1, and '0' to denote empty spaces (the '*why*' of it will be dealt in Section III). In Phase 3 (3D variant), we switch to '1' for Player 1, '-1' for Player 2, and '0' for empty spaces. This representation facilitates the implementation of RL algorithms and state evaluation.

C. The Curse of Dimensionality in Tic-tac-toe

A significant challenge in applying RL to tic-tac-toe, especially in its 3D form, is the "curse of dimensionality." As the game extends to three dimensions, the number of possible states increases exponentially and poses a great computational and analytical challenge.

D. Online Learning and Pruning of States

To address this issue, two strategies become crucial: online learning and pruning of states. Online learning, while prone to stochastic variations, allows the RL algorithm to adapt and learn from each unique game state. Pruning, on the other hand, involves strategic elimination of certain game states based on predefined conditions, thereby reducing the computational load.

Considerations for Pruning States

1) The number of marks for Player 1 must either be equal to or one more than that for Player 2.
2) The number of marks for Player 2 must either be equal to or one less than that for Player 1.
3) States where a player is winning in multiple ways are invalid.
4) States where a player is both winning and losing are invalid.

E. The Computational Challenge

Even after rigorous pruning, the computational challenge remains. In Phase II, for instance, nearly 10 million states survived pruning, underscoring the complexity involved in the game's analysis.

F. Optimal Play and Nash Equilibrium

In an optimal game scenario, where two fully trained agents are pitted against each other, the outcome gravitates toward a Nash equilibrium, typically resulting in a draw. This is due to the symmetric nature of tic-tac-toe, where perfect play from both sides often leads to an impasse. Therefore, an initial thought of solving the game of tic-tac-toe can involve a *min-max* optimization formulation where one agent is trying to maximize its future expected reward, while the other is trying to minimize that.

III. Phase 1 - 2D 3×3 Tic-tac-toe

A. Value Iteration

The solution to this problem features Value Iteration, an algorithm grounded in dynamic programming and well suited for discrete, finite-state environments. Thus, it was an ideal choice for our initial exploration.

The motivation for selecting Value Iteration also lies in its simplicity and effectiveness. The algorithm iteratively estimates state values, which aligns well with the turn-based nature of tic-tac-toe. Each state's value is a measure of the game's favorability toward a player at any given turn.

The theoretical foundation of Value Iteration is based on the Bellman Equation, offering a recursive solution to the value function in a Markov Decision Process (MDP). The Bellman Optimality Equation for Value Iteration is given by:

$$V^*(s) = \max_{a \in A} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')], \quad (1)$$

where $V^*(s)$ is the optimal value of state s, A represents the set of all possible actions (of either player), $P(s'|s,a)$ denotes the transition probability, $R(s,a,s')$ is the immediate reward, and γ is the discount factor, balancing immediate and future rewards. A high discount factor makes immediate rewards more valuable relative to future rewards, thereby incentivizing the agent to reach the optimal solution faster.

In our tic-tac-toe application, actions correspond to placing a mark in an empty cell. Rewards are configured to reflect the outcomes of the game: wins, losses, and draws. The algorithm iteratively updates the value of each state based on these rewards until it converges to an optimal policy. The optimal policy dictates the most favorable action in each state and hence maximizes the probability of winning. The deterministic nature of the game is note-worthy, as it ensure that there is no concept of transition probabilities any phase. Therefore, **Equation 1** becomes relatively simple:

$$V^*(s) = \max_{a \in A} R(s,a,s') + \gamma V^*(s'), \quad (2)$$

The implementation of Value Iteration in this context highlights both its utility and the challenges in adapting it to a game environment. Our subsequent discussion covers the specifics of this implementation, the encountered challenges, and the key insights gleaned from this phase.

B. Looking for Alternatives

Since our objective was to reach the *Nash* Equilibrium, our first thought led us to believe that the algorithm specified above is insufficient for training two competing agents to play optimally and hence consistently resulting in draws. Therefore, we started researching on variants of value iteration, specifically for *zero-sum* or two-player games. We came across the concepts of *Value Set Iteration* (VSI) and *Policy Set Iteration* (PSI) in [1] and [2], followed by a culmination of the two concepts in the paper [3], where the author uses VSI in tackling zero-sum games. Below is a quick formulation of their idea that stems from the point of view of a min-max optimization problem for getting the optimal policy for both the maximizer and the minimizer.

A two-person zero-sum *deterministic* Markov game is defined by the tuple $M = (X, A, B, C)$, where X is the state set and $A(x)$ and $B(x)$ are action sets for the minimizer and the maximizer, respectively, at state x. The value function for policies $\pi \in \Pi$ and $\phi \in \Phi$ - where Π, Φ represents the set of all policies for the maximizer A and the minimizer B - with an initial state x is given by

$$V(\pi, \phi)(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(X_t, \pi(X_t), \phi(X_t)) \middle| X_0 = x\right], \tag{3}$$

where X_t is the state at time t, and γ is the discount factor (with same interpretation as **Equation 1**) representing the expected future (discounted) sum of rewards. $V^*(\pi, \phi)$ denotes the *Nash* equilibrium of M. A policy π is ϵ-optimal if

$$\max_{x \in X} |V^*(x) - \sup_{\phi \in \Phi} V(\pi, \phi)(x)| \leq \epsilon. \tag{4}$$

In words, this equation means that if we are able to approximate the optimal value function V^* by the game value obtained when the minimizer plays a policy π and the maximizer plays the best-response policy to π with an error at most by ϵ at every x, π is ϵ-optimal. With these preliminaries, the VSI's cornerstone equation is the contraction mapping $T : B(X) \times P(\Pi) \to B(X)$, where $B(X)$ is the set of all real-valued functions (including V^*) - defined as

$$T(u, \Delta)(x) := \inf_{f \in F(x)} \sup_{g \in G(x)} \Big(R(x, f, g) + \gamma \min \left\{ u(y), \min_{\pi \in \Delta} \sup_{\phi \in \Phi} V(\pi, \phi)(y) \right\} \Big),$$

for $u \in B(X)$, $\Delta \in P(\Pi)$ (representing the power set of the set of all policies Π) and *admissible policies* $F(x), G(x)$. From the above equation, we infer that VSI operates with a sequence $\{\Delta_k\}$ of policy sets. At iteration k, an initial estimate of V^* is derived using the value functions, where each policy in Δ_k is played against its best-responsive rival policy. This estimated function is then refined by comparing with V_k, the current VSI estimate of V^*, which in turn is utilized to generate V_{k+1}, the updated estimate of V^* for the next iteration $k + 1$. Moreover, the above formulation under the *max norm* contraction operator implies that V^* is a unique fixed point of T, satisfying ϵ-optimality conditions. Note that in the event $\Delta = \emptyset$, it simply reduces to the value iteration equation.

In terms of the motivations of this approach, the paper finds that while retaining the dimensionality challenges of VI, this approach offers more flexibility and potentially faster convergence. It demonstrates a linear convergence rate to the equilibrium value function under certain conditions.

Although the author ingeniously uses the concepts of policy/value set iteration, i.e., generalized versions of policy and value iteration and contraction mappings to pose the standard Bellman optimality algorithm as an exact mini-max game that shows at least linear convergence for standard MG's, unlike some variants of VI that require exponential/polynomial convergence. We decided not to go ahead with this specifically for the reasons it had mentioned in the limitations before due to it being an *exact* algorithm.

Before reading this paper, we had only come across 'exact' terminology when studying gradient descent and its various learning step methodologies, such as backtracking line search, inexact line search, and exact line search.

Given an objective function $f(\mathbf{x})$, the gradient descent update with exact line search at iteration k is defined as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k), \tag{5}$$

where α_k is the step size, determined by solving the line search problem:

$$\alpha_k = \arg\min_{\alpha} f(\mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)). \tag{6}$$

In this process, α_k is chosen to minimize the objective function along the direction of the negative gradient $\nabla f(\mathbf{x}_k)$. This approach, despite ensuring the step size that maximally decreases the function value by optimizing over all possible step sizes, easily becomes intractable for complicated functions or large scale systems.

Similarly, in the VSI formulation, the optimization problem is over the set Δ that represents the power set of all possible policies Π and hence becomes intractable for huge state spaces. Its effect may not be that pronounced in 3×3 2D tic-tac-toe, but it is still a valid concern in general. Therefore, though exact value set iteration, i.e., choosing optimal expected future discounted reward/cost at each iteration, does seem to work quite well, we believe it is precisely due to the exact approach the author expresses that the algorithm is vulnerable to the curse of dimensionality and becomes intractable for large state/action spaces. Therefore, the computation and access of the set Δ can become tedious. To conclude this alternative method search, we did not utilize the above approach, but instead made sure to use it as a learning experience and drew inspiration from it by considering the simple case of $\Delta = \emptyset$.

C. Value Iteration Applied - Gym to Base 10 conversion

As we now had resorted again to value iteration, the problem was to implement it in our zero-sum game. Our

first thought was to switch the min-max operators for the two agents, thereby making the two agents learn in an adversarial fashion. We ran experiments for a small number of iterations and observed improved performance/policy for both. We also observed that, given enough iterations, a Nash equilibrium was achievable. However, then came the computational side of matters. The original implementation drew inspiration from the *Farama Foundation's Gymnasium* environment where we first made the state transition graph in the form of a dictionary. The graph's purpose was to provide us with the transitioned state, its reward, and its termination status (i.e., whether it is a win, lose, or draw) when given a state-action pair. However, this became computationally infeasible when the dictionary grew too large to search for optimal actions by comparing the value of the transitioned states that were indexed by the states themselves (recall that states were originally numbered as 1 for player 1 mark, -1 for player 2 mark, and 0 for no mark). Therefore, we resorted to the idea of *hash tables*, toward which we were nudged by our esteemed professor, Dr. Hassan Jaleel. If we had a unique identifier for each of the states, a lookup table would be possible with *O(1)* time complexity. To obtain the unique identifier, we indexed the states as mentioned in Section II, as that allows us to treat the states as unique values, and therefore identifiers in a look-up table, in a ternary numeral system. Furthermore, this made *Gymnasium*'s implementation unnecessary, as now we could simply compute each state's transition, reward etc. on the fly. The same approach is also used in Phase II. In conclusion, we not only achieved computational efficiency, but also observed fine convergence as pitting the two learned agents against each other always resulted in a draw, the Nash equilibrium. Moreover, when either player played against a human, the game still ended in a draw if not a loss for the human. However, it was surprising to note that, once we had learned their optimal policies, Player 1 won more games (though more than 60% of the games resulted in a draw) if we employed an ϵ-greedy strategy for choosing actions. This is indicative of the fact that, in tic-tac-toe games, Player 1 can either win or draw with optimal play, whereas Player 2 can only draw. This implies that that slightly sub-optimal moves have a greater chance of effecting Player 2's chance of losing. The code for the implementation can be found here: **Github Code**.

IV. Phase 2 - 2D 4 × 4 Tic-tac-toe

A. Increased Complexity

The 4 × 4 tic-tac-toe presents over ten million states, an exponential leap from the 3 × 3 variant, posing significant computational challenges.

B. Q-Learning Formulation

Q-learning, a model-free reinforcement algorithm, was chosen for its ability to operate in this vast state space. The update formula for the Q-value is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \quad (7)$$

where s is the state, a is the action, r is the reward, α is the learning rate, γ is the discount factor, and s' is the subsequent state.

C. Temporal Difference Learning: TD(0), TD(1), and TD(λ)

Temporal Difference (TD) learning bridges the gap between dynamic programming and Monte Carlo methods. In Q-learning, the TD gap is defined as the difference between the estimated Q-value and the observed reward plus the discounted estimate of subsequent state-action pairs:

$$\text{TD gap} = r + \gamma \max_{a'} Q(s',a') - Q(s,a). \quad (8)$$

The TD(1) approach, also known as the Monte Carlo method, updates the Q-values at the end of each episode. Its update rule is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G_t - Q(s,a)), \quad (9)$$

where G_t is the actual return following time t.

The TD(λ) approach generalizes both TD(0) and TD(1) by considering all possible n-step ahead predictions in the update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \sum_{n=1}^{\infty} (\lambda^{n-1} \delta_t^{(n)}), \quad (10)$$

where $\delta_t^{(n)}$ is the n-step TD error and λ is the trace decay parameter, blending the information from all time steps.

As the value of λ in TD(λ) approaches 0, the updates become more stochastic and noisier, similar to the updates in stochastic gradient descent. This introduces more variance in the updates, but allows for faster learning as each step is computed using less data. This is akin to the trade-off observed in machine learning optimization, where stochastic gradient descent offers fast but noisy updates, while full-batch gradient descent provides stable but computationally expensive updates. Mini-batch gradient descent strikes a balance between these extremes. The choice of λ thus reflects a similar trade-off between the speed and stability of learning in TD learning.

In our implementation, we chose TD(0) for its immediate update feature, favoring speed over the potential accuracy gains from waiting until the end of the episode in TD(1), and avoiding the complexity of TD(λ)'s full spectrum of multi-step predictions.

D. Controlling Exploration vs Exploitation Trade-off and Noisy Updates

The ϵ-greedy strategy is employed to manage the exploration versus exploitation trade-off and is defined as

$$a_t = \begin{cases} \arg\max_a Q(s_t,a), & \text{with probability } 1-\epsilon \\ \text{a random action}, & \text{with probability } \epsilon \end{cases} \quad (11)$$

where a_t is the action taken at time t, $Q(s_t,a)$ is the Q-value of action a in state s_t, and ϵ is the exploration rate.

In our implementation, ϵ is decayed exponentially to encourage less exploration as the model becomes increasingly confident in its predictions. This is represented by

$$\epsilon = \epsilon_0 e^{-\kappa t} \quad (12)$$

where ϵ_0 is the initial exploration rate, e is the base of the natural logarithm, κ is the decay rate, and t is the episode number.

Similar to the Adam optimization method [4], which adapts the learning rate for frequently updated weights, our approach employs a counter for each state-action pair to adjust the learning rate α. This counter acts to decrease α for state-action pairs that are updated more frequently, thereby mitigating noise and improving the balance of the model:

$$\alpha = \frac{\alpha_0}{1 + \text{count}(s, a) \cdot \text{adap\_lr}} \tag{13}$$

where α_0 is the initial learning rate, $\text{count}(s, a)$ is the number of times the state-action pair has been updated, and adap_lr is the adaptation rate for the learning rate.

E. Min-Max Switch for Q-value Updates

In standard Q-learning, the model is typically designed for a single agent. However, in our two-agent tic-tac-toe game, the Temporal Difference (TD) gap is based on the other player's move. This necessitates a switch in the Q-value update rule: we use a maximization for Player 2 and minimization for Player 1. This switch mirrors the adversarial nature of the game, akin to Generative Adversarial Networks (GANs). Such an approach is similar to what we implemented in value iteration, where we adapted the strategy to optimize for both agents, ultimately steering the game toward a Nash equilibrium.

F. Limitations and Observations

While implementing the Q-learning algorithm for the 4×4 tic-tac-toe, we encountered significant limitations. The requirement to maintain a Q-table for over 10 million states resulted in extensive memory usage and inefficiency. Furthermore, our focus was primarily on the implementation rather than the precision of the algorithm.

In testing, the two agents, each following their learned policy, competed against each other in a series of 20 games. Surprisingly, these games frequently ended in draws. This outcome suggested that the agents' policies were sub-optimal rather than reflective of having achieved Nash equilibrium. Closer examination revealed that the agents often failed to execute moves that would guarantee a win, indicating a need for further refinement in the learning process and policy optimization. See code here: **Github Code**.

V. Phase 3 - 3D $4 \times 4 \times 4$ Tic-tac-toe

A. Deep Reinforcement Learning Challenges in 3D Tic-tac-toe

In the landscape of deep reinforcement learning (RL), the 3D $4 \times 4 \times 4$ tic-tac-toe stands as a classic challenge, marred by the curse of dimensionality, with 3^{64} possible states, and the problem of episodic, sparse, or delayed rewards. The late revelation of action utility, typical in many board games, complicates the learning process, especially when paired with a colossal state space akin to those in Go or Chess.

Although research to address these issues obstacles is being done, TD(0) for now has shown great promise for in dealing with high dimensionality or sparse rewards as shown in Chess [5] or Othello [6]. Therefore, our methodology for Phase 3 leverages a Deep Q-Network (DQN) with TD(0) framework, a data-driven paradigm propelled by the successes in neural networks across fields in applied mathematics, like signal processing and wireless communications. DQNs are adept at approximating the optimal action-value function Q^*, which is crucial for navigating environments with extensive state spaces and sparse rewards.

B. DQN Approach and Architecture

Traditionally, one might consider a fully connected Multi-Layer Perceptron (MLP) for neural network-based learning tasks. However, for complex games such as 3D $4 \times 4 \times 4$ tic-tac-toe, this approach does not inherently account for the relational significance of board positions. Understanding that certain sequences of marks hold more strategic value, we have employed a 'structured' DQN borrowed from [7]. This approach, inspired by domain-specific insights, focuses on the crucial rows of marks that determine winning conditions, significantly reducing the number of trainable parameters while preserving the network's ability to discern critical patterns. Many such 'structured' neural networks have been proposed before such as those involving convolutional neural networks (CNN) [8]. They, though similar to our approach that we will soon discuss, employ sparse connections between inputs and hidden layer units. However, they are made invariant to small translations of patterns in an image (in the context of vision problems, especially CNNs). But we require the neural network to not be translation invariant due to aforementioned reasons. Hence, we adopt a DQN data-driven approach to approximate the optimal Q-values. The architecture of our network is shown in **Figure 1**.

Our DQN's input layer, representing the 64-state board, is subdivided into 76 distinct rows—horizontal, vertical, and diagonal—reflecting all possible winning lines. To adeptly capture the patterns within these rows, we introduce four 'detector' neurons per row, resulting in a structured hidden layer with 304 neurons. Each detector is dedicated to recognizing specific configurations within its corresponding row. This design not only streamlines the parameter space but also enhances the network's pattern recognition capability, crucial for discerning the strategic state of the game.

Following the structured hidden layer, the network converges into a fully connected layer and subsequently to the output layer, maintaining computational efficiency and pattern-detection efficacy.

C. TD(0) Algorithm in the Context of DQN

In the realm of DQN, the TD(0) algorithm serves as the backbone for updating the Q-values. It starts with the standard Bellman equation for Q-learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) \tag{14}$$

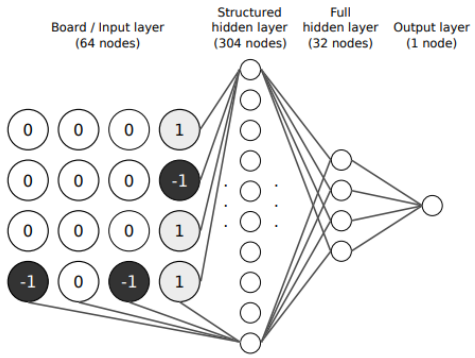

Fig. 1. The architecture of the DQN used in our implementation.

However, within our neural network framework, there is already a learning rate for the optimizer therefore, we assume $\alpha = 1$, simplifying the update to:

$$Q(s,a) \leftarrow r + \gamma \max_{a'} Q(s',a') \tag{15}$$

This positions the neural network as a function-approximator for the Q-value given a state-action pair.

The TD-learning algorithm with a DQN is applied as follows:

Algorithm 1 Learn-From-Game TD-learning with DQN

1: Compute $Q(s_t,:)$ by forward pass through the NN for each possible action a and corresponding afterstate s'_t
2: Select action leading to afterstate s^a_t using policy ϵ-greedy exploration. (See 11)
3: Use the Q-update equation 15 to compute the target value $Q^{\text{new}}(s^a_{t-1})$
4: Forward propagate the Neural Network for the current value of the previous afterstate $Q(s^a_{t-1})$
5: Backpropagate the error between $Q^{\text{new}}(s^a_{t-1})$ and $Q(s^a_{t-1})$
6: Save afterstate s^a_t as s^a_{t-1}
7: Execute the selected action leading to afterstate s^a_t
8: Let the opponent perform an action and increment time t

The algorithm updates the values of afterstates (s'_t - the state resulting from the player's move, before the opponent's turn), as these are the state values that are used for move selection. Because tic-tac-toe 3D is played against an opponent, we must wait for our opponent to make its move before learning an afterstate's value. The TD-error is the difference between subsequent afterstate values in between which the opponent also makes a move. As we are using online TD-learning, the update rule is applied on each of the player's turns (except the first one). In 3D tic-tac-toe, there is only a reward at the end of the game. The player receives a reward of 1 upon winning, -1 upon losing, and 0 in case the game ends in a draw. For initial moves ($t = 1$), where there is no preceding afterstate, steps 1, 2, and 6-8 are conducted. We set $\epsilon = 0.1$ in our implementation to balance exploration with exploitation.

Training against a static opponent with a benchmark policy, the TD player learns exclusively from its own moves. When engaging in self-play, the algorithm is applied to both players, doubling the learning data per game but at the cost of not learning directly from the opponent it will eventually face.

D. Training, Network and Coding Specifics

The state representation in our DQN framework uses 1 for Player 1, -1 for Player 2, and 0 otherwise. To align the neural network's output with this scheme, we employed the hyperbolic tangent (Tanh) activation function, which scales the values to the range [-1, 1]. This choice is empirically shown to be effective with Xavier initialization for weights [9], due to its variance conservation properties. For a weight W belonging to some hidden layer, Xavier initialization draws the weight matrix from the normal distribution as defined below.

$$W \sim N\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right) \tag{16}$$

where n_{in} and n_{out} are the number of input and output units of the hidden layer.

Moreover, as we discussed before that asynchoronous/online learning especially in TD-learning can result in noisy updates/more variances. Therefore, the loss function (characterized by the TD gap) has to be such that the supervisory signal generated during backpropropgation when the agent is training (either against itself or against a benchmark policy) has to be robust to noise/outliers. For this reason we used the Huber loss. The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy. It has been employed in many estimation problems in machine learning such as [10]. See **Figure 2** where the loss is shown for a meager 1000 episodes which depicts this behaviour.

Fig. 2. TD Gap against number of episodes. Shows noisy updates of $Q(s,a)$

$$L_{\text{Huber}}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise,} \end{cases} \tag{17}$$

where $\delta = Q(s,a) - (r + \gamma \max_{a'} Q(s',a'))$ is the TD error/gap.

During training, the opposing player followed a random policy for 5000 episodes before we shifted to a benchmark policy specified by the networks own trained weights. This

strategy encouraged Player 1 to learn strategic moves rather than just adapting to randomness, paving the way for more sophisticated game-play.

Although every outlined step here flows easily from one to another, one of the most trickiest part in implementing the DQN was making the architecture. All our code is done using python and *PyTorch* framework. Although PyTorch is extremely powerful for building and training neural networks, for cases (layers in our case) where a somewhat non-orthodox structure is needed, PyTorch requires some customization. And that specific customization we had to do for developing the 'structured' hidden layer. The classic

```
from torch import nn
nn.Linear(76, 304)
```

falls short. This is because, as explained, the above functionality makes fully connected layer networks, which in our case is not needed. Due to each row being connected to their respective set of four nodes/neurons/detectors in the hidden layer, we had to customize this layer. After hours of trial-and-error work, we were able to accomplish this. The trick lied in realizing that the connections between the inputs, i.e., 76 rows to the 304 neurons in the hidden layer was just a *element-wise* dot product of the respective elements. Consider the figure below which shows the weight matrix of the structured hidden layer and the input *feature map*.

$$\overset{1\times 76}{\begin{bmatrix} x_1 & \cdots & x_{76} \end{bmatrix}} \quad \overset{304\times 304}{\begin{bmatrix} \boxed{\cdot} & 0 & \cdots & 0 \\ 0 & \boxed{\cdot} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \boxed{\cdot} \end{bmatrix}}$$

The diagram above shows the rows $x_1 \cdots x_{76}$ (each consisting of 4 input nodes) arranged as a 1×76 vector. But note that since we are customizing our layer, this is not a typical vector. Each element in it is a 1×4 vector corresponding to the dimensions of an individual row. Similarly observe the weight matrix of the structured hidden layer. Sharp minds will not only note that this a block diagonal matrix but a special type of *Jordan* form of a matrix where the blocks (of dimensions 4×4) represents the non-zero weight connections of each row with its 4 hidden layer nodes in the structured layer. After this realization, the rest was simple, customizing an element by element dot product. The output, once flattened, is then of the dimensions of 1×304. This is then fed to the rest of the fully connected (FC) layers as a typical neural network. Once we have a clear idea of how to mathematically formulate the connections before and after the structured layer, PyTorch and its powerful back-propagation algorithm is able to compute the weight updates.

E. Testing Performance and Limitations

The training process involved running episodes against a random policy and then against a benchmark policy. The ϵ-greedy strategy and Huber loss function were employed to navigate the exploration-exploitation trade-off and to provide robustness against noise, respectively. To elaborate on the benchmark policy a bit, at first the agent was trained against the other player's random policy. The idea was that given enough number of episodes, the agent would have come up with an approximate best policy for each possible state. However, since the algorithm was turning out to be a bit computationally expensive, we started focusing on optimizing the other player's moves or providing it with a policy that is at least better than random. Intuitively, this would force our agent to come up with better move strategies hence resulting in fine-tuning of the weights. Therefore, after 5000 episodes, we used the model's weights as a policy for the other player while our current agent was tuning its parameters. We saw a jump in the improved performance when played (in a series of 100 games) against a bot (with pure random actions) which was quite surprising since typically models involving such a huge space require a huge number of episodes (typically in the order of 10^6). Not to say that our agent was making optimal decisions, but the trend in the improved performance was a indication our approach was at least theoretically correct, which was what most important for us. The graph below in **Figure 3** shows the episode number against average win $\%$ when tested against a bot with pure random actions as the policy. See code here: **Github Code**.

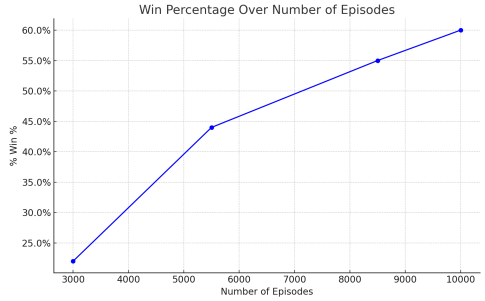

Fig. 3. Win Percentage against Number of Episodes

VI. Conclusion and Limitations

This study ventured through the complex domain of reinforcement learning, particularly applying various algorithms to different forms of tic-tac-toe. From the foundational 2D 3 $\times$ 3 game in Phase 1 to the intricate 3D 4 $\times$ 4 $\times$ 4 variant in Phase 3, we explored Value Iteration, Q-Learning, and DQN approaches. Our journey revealed significant insights into the application of these algorithms, showcasing their effectiveness and adaptability to varying game complexities and dimensions.

Despite the successes, we encountered limitations such as the computational demand and memory inefficiency, especially in larger state spaces. Additionally, the inherent challenges of sparse rewards in reinforcement learning were pronounced in the more complex game versions. These limitations underline the need for continued research and development in efficient

algorithmic strategies and resource management in deep reinforcement learning environments.

REFERENCES

[1] Hyeong Soo Chang, "Value set iteration for Markov decision processes," *Automatica*, 49, 3687–3689, 2013, DOI: 10.1016/j.automatica.2014.05.009.

[2] Hyeong Soo Chang, "Policy set iteration for Markov decision processes," *Automatica*, 49(12), 3687-3689, 2013, DOI: 10.1016/j.automatica.2013.09.010.

[3] Hyeong Soo Chang, "Value set iteration for two-person zero-sum Markov games," *Automatica*, 76, 61-64, 2017, DOI: 10.1016/j.automatica.2016.10.010.

[4] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," *International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[5] Sebastian Thrun, "Learning to play the game of chess," *Advances in Neural Information Processing Systems*, volume 7, pages 1069–1076, 1995.

[6] M. van der Ree and M. A. Wiering. "Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play," *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, April 2013, pages 108–115.

[7] Michiel Steeg, Madalina Drugan, and Marco Wiering, "Temporal Difference Learning for the Game Tic-tac-toe 3D: Applying Structure to Neural Networks," *Proceedings of the 2015 IEEE Symposium on Computational Intelligence*, December 2015, DOI: 10.1109/SSCI.2015.89.

[8] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Back-propagation applied to handwritten zip code recognition," *Neural Computation*, 1(4), 541–551, 1989.

[9] Katanforoosh & Kunin, "Initializing neural networks," deeplearning.ai, 2019.

[10] Shouyou Huang and Qiang Wu, "Robust pairwise learning with Huber loss," *Journal of Complexity*, 66, 101570, 2021, DOI: 10.1016/j.jco.2021.101570.