**Reinforcement Learning-Based Auto Chess Implementation**

[GitHub Repository](https://github.com/AGlebov-atbu/ds543-project)

Kaya Daylor

Aleksei Glebov

*1 May 2025*

1. *Introduction and Background*

Auto Chess is a strategy-based game that involves deploying units with different costs, abilities, and synergies, requiring the agent to make decisions on unit placement, itemization, and positioning. These decisions are based on game states which are constantly changing. Our goal of this project was to develop a Reinforcement Learning (RL) model that can autonomously play Auto Chess while learning optimal strategies for the game. The challenge lies in the complex nature of the game, the large state and action spaces, and the need for the agent to adapt to its strategies based on both random and opponent-driven elements in the game. The RL agent must navigate these challenges without relying on predefined strategies but rather by learning directly from its interactions with the environment.

In this project, we explore the application of a Deep Q-Network (DQN) to a simplified Auto Chess environment. Our goal is to train an agent so that it is capable of learning how to manage resources, select units, and prepare for battle with minimal hard-coded logic. The project combines a custom environment design with Reinforcement Learning to explore how well an RL agent can compete in an Auto Chess setting.

1. *Literature Review*

This project draws inspiration from several previous works and related articles. Like many RL models, we will incorporate Q-functions and expected outcomes into our implementation. We will utilize these previously established methods in conjunction with our original Auto Chess environment.

“Simulating Team Fight Tactics” by Riot Games (Cao, 2023) serves as a valuable resource that highlights unique challenges in implementing Auto Chess and how to mitigate these obstacles. This article is especially relevant as it is specific to our project domain.

“Large-scale deep learning to augment production RL workloads at Riot Games” by Anyscale YouTube channel (Anyscale, 2023) describes Riot Games’ implementation of RL bots in the Team Fight Tactics (TFT) production environment. This piece of media helped to discover the methods and techniques used by one of the most popular Auto Chess implementations.

“Reinforcement Learning in Chess,” an article on Medium made by Aditya (Gill, 2023), offers useful insights about implementing RL in traditional chess. Gill discusses several key approaches that influenced our work:

1. State Representation: The article highlights the importance of effective state encoding in chess, which informed our own approach in representing the complex state space of Auto Chess.
2. Self-Play Learning: Gill emphasizes the importance of self-play in developing strong chess agents. This reinforces our decision to implement a training regime where our agent plays against different versions of itself (i.e. random, rule-based, etc.).
3. Value Function Approximation: The article details how neural networks can effectively approximate the value function in chess positions, which parallels our approach to evaluating board states in Auto Chess. Gill specifically states how convolutional layers can capture spatial relationships between pieces. We adapted a similar concept for evaluating unit positioning on our 2D board.

Although traditional chess and Auto Chess are different, the fundamental challenges of representing complex states, balancing exploration vs. exploitation, and learning long-term strategies are common in both domains. Gill’s analysis of these challenges and the corresponding solutions provide valuable guidance for our project.

1. *Methods and Implementation*

We implemented a simplified Auto Chess environment using the Gymnasium API to allow for the integration of Reinforcement Learning. The environment simulates several core game mechanics. Our environment incorporates a shop system where units are randomly drawn based on the player’s level and pre-defined unit cost probabilities. The economy and leveling mechanics allow players to earn income each round and make strategic decisions about spending gold to buy units, refreshing the shop, or leveling up to unlock higher-tier units.

Unit management is a key feature enabling units to be moved from the bench to the board, repositioned, and upgraded to stronger versions through a star system (1-star, 2-star, 3-star). The combat system activates after each preparation phase (which lasts for 50 actions), with units automatically engaging in combat by attacking the nearest enemy unit within range. The losing player takes damage based on the number of enemy survivors.

We created two players, each with their own gold, health, board, and bench. During each environment step(), the agent selects a discrete action such as buying a unit, placing a unit, leveling up, or ending the round.

The agent is trained using a Deep Q-Network (DQN), which takes a structured observation dictionary as input. Observations include encoded representations of the player’s gold, level, and health; current shop units (encoded by their cost, type, star); and the bench and board layout.

The DQN architecture includes embedding layers for unit types and fully connected layers for each section (shop, bench, board). These feed into a final combined feature layer that outputs Q-values across the action space. Our reward system is carefully calibrated to promote strategic decision-making, offering positive reinforcement for effective actions such as optimal unit placement and combat victories, while discouraging suboptimal choices through negative feedback when the agent ends turns with unoccupied board spaces or suffers defeats in battle.

1. *Experimental Results*

To evaluate the effectiveness of our DQN agent, we trained it over a series of episodes within the Auto Chess environment. During each episode, the agent performed a sequence of actions such as buying units, leveling up, and managing the board, culminating in a simulated battle at the end of each round.

In the early stages of our development, the agent struggled to consistently produce an effective winning strategy. In the initial 100 episodes of training, most episodes yielded a negative reward, and the agent frequently ended turns with an empty board or non-optimal unit placements. Additionally, we encountered several errors due to invalid actions or poor state handling. These issues were addressed by refining the environment several times. However, over a longer period (250 episodes), our agent began to produce better results. While the rewards remain negative, the average reward clearly increases. Figure 1 shows this increase in reward over 250 episodes with the moving average seen in red.

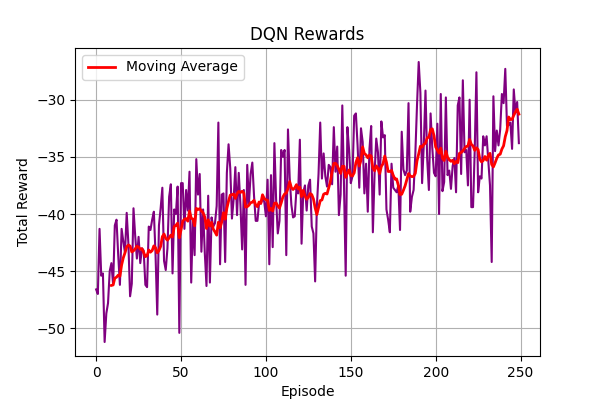


Figure 1. DQN Rewards

Despite the low reward performance, our agent demonstrated exploration behavior by sampling a range of valid actions. The epsilon-greedy exploration strategy successfully showed a decay in epsilon over time, but the agent was not yet able to converge to a strong policy. Analysis of the agent’s action distribution revealed a tendency to favor certain actions, particularly buying units when available, while sometimes neglecting strategic positioning or economic considerations. This behavior likely stems from the immediate positive reward associated with unit acquisition compared to the delayed rewards of optimal board positioning or economy management.

In terms of combat performance, the agent’s win rate remained below 50% when facing our rule-based opponent, which follows simple yet effective heuristics. The most common failure modes included:

1. Poor unit positioning with incorrect units placed in vulnerable front-line positions
2. Suboptimal gold management, often depleting resources early rather than building economy
3. Failure to create synergistic unit combinations that would activate bonuses

These observations provide clear directions for refining our approach in future work, particularly in terms of reward shaping and positional improvements to better capture the long-term strategic elements of Auto Chess.

1. *Analysis and Discussion*

Our experiments highlight several challenges that arise when applying reinforcement learning techniques to a complex, multi-action environment like Auto Chess. The most notable issue is the sparse reward structure—many of the agent's decisions provide no immediate feedback, with the final reward (winning or losing a battle) occurring several steps after critical decisions have been made. This temporal disconnect between actions and outcomes makes it difficult for the agent to associate specific decisions with their long-term consequences, particularly when learning strategic elements like unit synergies, economic management, and optimal board positioning.

The expansive action space presents another substantial hurdle. Our agent must navigate a complex decision landscape including shop purchases, strategic rerolls, board placements, unit repositioning, and economy-based leveling actions. Despite our efforts to structure observation inputs and embed unit features effectively, the action space remains challenging to explore efficiently without incorporating domain knowledge or implementing curriculum learning approaches as mentioned in some prior arts. During early training episodes, we frequently observed the agent attempting non-beneficial or even impossible actions—such as purchasing units or leveling up without sufficient gold or attempting to move non-existent units. While we introduced small negative rewards to discourage these invalid actions, this exploration inefficiency remains a significant barrier to learning.

The multi-stage nature of Auto Chess further complicates the learning process. The agent must master distinct yet interconnected phases of play (economic development, army building, and positioning) while balancing short-term tactical decisions against long-term strategic planning. Our current reward model, while functional likely requires more sophisticated shaping to better guide the agent through these interconnected decision layers.

Despite these challenges, we observed encouraging signs of learning. As training progressed, the agent began demonstrating coherent patterns of exploration and increasingly consistent behaviors, such as prioritizing unit purchases in early rounds and attempting basic positioning strategies. With additional training iterations, refinements to the reward structure, and architectural optimizations—particularly in how the agent processes the game state—we believe the performance can be substantially improved. Future work might explore implementing hierarchical Reinforcement Learning approaches to better handle the distinct decision layers inherent in Auto Chess.

1. *Conclusion and Future Work*

We successfully developed a custom Auto Chess environment and implemented a Deep Q-Network agent to learn strategic gameplay through Reinforcement Learning. The environment we engineered effectively captures the core mechanics of the Auto Chess game—including resource management, unit acquisition and positioning, tier upgrades, and combat resolution. While our agent has not yet achieved mastery of the game, the progressive learning behaviors observed demonstrate the viability of applying Reinforcement Learning techniques to complex, multi-phase games with delayed rewards.

Throughout this work, we identified several critical challenges inherent to applying Reinforcement Learning in such environments: the difficulty of learning from sparse, delayed rewards; the inefficiency of exploration in expansive, combinatorial action spaces; and the technical complexities of designing robust simulation environments that accurately model strategic gameplay while maintaining computational efficiency.

For future development, we have identified several promising directions:

1. Enhancing the reward architecture to provide more granular feedback on specific strategic elements, such as effective gold utilization, successful unit combinations, and optimal board positioning.
2. Restructuring the action space by decomposing complex actions into discrete components (separate actions for shop selection, unit placement, and board movement) to simplify the learning process.
3. Implementing advanced Reinforcement Learning methodologies including Double DQN, Deuling DQN, and actor-critic architectures to improve learning stability and efficiency.
4. Extending the training duration and conducting systematic hyperparameter optimization to enhance convergence and performance.
5. Introducing curriculum learning approaches that gradually increase game complexity as the agent develops competence.

This project not only demonstrates the capabilities of Reinforcement Learning in complex strategic decision-making environments but also establishes a foundation for future research into Auto Chess and similar strategy games. The process of designing and implementing this system—from environment modeling to agent architecture—provided valuable insights into the practical challenges of applying Reinforcement Learning to real-world problems characterized by delayed rewards and complex state-action relationships.

The experience of watching our agent gradually develop strategic behaviors, despite the inherent complexity of the game, was particularly rewarding and offers encouraging evidence that Reinforcement Learning can be effectively applied to sophisticated decision-making systems that require both tactical awareness and long-term planning.

*References*

Anyscale. (2023). Large-scale deep learning to augment production RL workloads at Riot Games [Video]. YouTube. <https://www.youtube.com/watch?v=8EsQkFxWYhU>

Cao, R. (2023). Simulating Teamfight Tactics [PowerPoint slides]. Game Developers Conference. <https://media.gdcvault.com/gdc2023/Slides/Simulating++Teamfight+Tactics_Cao_Ran.pdf>

Gill, S. (2023). Reinforcement learning in chess. Medium. <https://medium.com/@samgill1256/reinforcement-learning-in-chess-73d97fad96b3>

silverlight6. (2025). TFTMuZeroAgent [GitHub repository]. GitHub. <https://github.com/silverlight6/TFTMuZeroAgent>