**Reinforcement Learning-Based Auto Chess Implementation**

Kaya Daylor

Aleksei Glebov

1. *Introduction and Background*

Auto Chess games are a genre of strategy games that combine elements of drafting, positioning, and real-time combat resolution. Players build a team of units over several rounds, balancing economy, unit upgrades, and positioning strategies. Given the complexity of decisions and long-term consequences of early actions, these games present an excellent testbed for reinforcement learning techniques.

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 capable of learning how to manage resources, select units, and prepare for battle with minimal hard-coded logic. The project combines custom environment design with reinforcement learning to investigate how well a learning agent can compete in this strategic setting.

1. *Methods and Implementation*

We implemented a simplified Auto Chess environment using the Gymnasium API to allow for reinforcement learning integration. The environment simulates core game mechanics, including:

* **Shop system**: Units are randomly drawn based on the player's level and unit cost probabilities.
* **Economy and leveling**: Players earn income each round and may choose to spend gold on buying units, refreshing the shop, or leveling up to unlock higher-tier units.
* **Unit management**: Units can be moved from the bench to the board, repositioned, and upgraded into their stronger versions through a star-level system (1★, 2★, 3★).
* **Combat**: After each preparation phase (which lasts for 50 actions), units automatically engage in combat, 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:

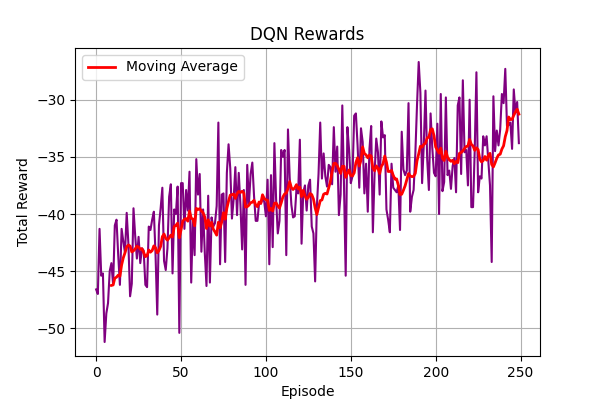
* Player’s gold, level, and health;
* Current shop units (encoded by their cost, type, star);
* Bench and board layout.

The DQN architecture includes embedding layers for unit types, fully connected layers for each section (shop, bench, board), and a final combined feature layer that outputs Q-values for each possible action type. To guide the agent’s learning, we designed a reward system that provides positive reinforcement for strategic actions (e.g., successfully placing units, winning fights) and negative reinforcement for poor decisions (e.g., ending a turn with an empty board, losing fights).

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.

At this early stage of development, the agent struggles to consistently produce winning strategies. 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 were addressed by refining the environment several times. However, over a longer period of time (250 episodes), our agent began to produce better visual results. Of course, the rewards still stick to negative values, but the average reward has visually increased.



Despite the low reward performance, our agent demonstrated exploration behavior by sampling a range of legal actions. The epsilon-greedy exploration strategy decays over time, but the agent has not yet converged to a strong policy. We believe that further training and reward shaping are necessary to improve his learning.

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. One of the most notable issues is the sparse reward structure: many of the agent's actions provide no immediate feedback, and the final reward (winning or losing a battle) may occur several steps after a key decision. Additionally, the agent must learn long-term strategy involving unit combinations, gold management, and board control, which is far more complex than environments with short action-reward loops. This makes convergence slower and requires carefully designed reward shaping to guide learning.

Another critical factor is the large action space. Our agent must choose between shop purchases, rerolls, board placements, unit movements, and leveling actions. Although we used structured observation inputs and embedded unit features, the action space remains difficult to explore efficiently without prior knowledge or curriculum learning. We also observed that non-beneficial actions (such as buying units or leveling up without enough gold or trying to move non-existent units) occurred frequently during early episodes. We mitigated this through small negative rewards for failed actions, but the issue remains a significant barrier to efficient learning.

Despite these challenges, the agent began to show patterns of exploration and attempted reasonable behaviors, such as consistently buying units early in the episode. With additional training time, further reward tuning, and architectural adjustments, we believe the agent’s performance can be significantly improved.

1. *Conclusion and Future Work*

In this project, we developed a custom Auto Chess environment and applied a Deep Q-Network (DQN) agent to learn strategic decision-making through reinforcement learning. Our environment captured essential aspects of the genre, including gold economy, unit selection, upgrades, and combat mechanics. While the agent has not yet demonstrated consistently strong performance, our results show the feasibility of applying RL to such complex, multi-phase games. The project highlighted important challenges related to reward sparsity, exploration in large action spaces, and the need for robust environment design and error handling.

Moving forward, we plan to improve agent performance through several key enhancements:

* **Revising the reward function** to encourage specific desirable behaviors (e.g., board population, unit upgrading).
* **Making separate actions** for selecting shop indices, movement cells, and unit positions.
* **Using more advanced RL methods**, such as Double DQN, Dueling DQN, or policy gradient approaches.
* **Going for longer training runs** and experimentation with different hyperparameters.

Finally, our project demonstrates the potential of reinforcement learning in complex, high-level decision-making environments and sets the foundation for more advanced Auto Chess agents in the future. Working on this project was both challenging and deeply rewarding. It allowed us to explore the practical aspects of reinforcement learning while designing a custom game environment, and it was genuinely exciting to watch our agent learn and interact with the system we built.

*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>