Learning to Settle: Reinforcement Learning in Catan

Anirud Aggarwal
University of Maryland, College Park
anirud@umd.edu

Jinhai Yan
University of Maryland, College Park
jyan1236@umd.edu

Serena Huang
University of Maryland, College Park
shuang04@umd.edu

Rohit Kommuru
University of Maryland, College Park
rkommuru@umd.edu

Monish Napa
University of Maryland, College Park
mnapa@umd.edu

Han Lin
University of Maryland, College Park
hlin1238@umd.edu

Abstract

This project investigates the development of AI agents for playing the strategic board game Catan using reinforcement learning (RL). Existing approaches often rely on single-agent RL, treating other players as static elements of the environment, which fails to capture the dynamic inter-player strategies essential for competitive games. To address this, we initially aimed to leverage multi-agent reinforcement learning (MARL) to model these interactions. However, due to the complexity of MARL and time constraints, we refocused on single-agent RL to produce results. Despite suboptimal performance, our findings highlight challenges and opportunities for future applications of MARL in strategic multiplayer settings like Catan.

1 Introduction

In recent years, RL has achieved remarkable success in games, surpassing professional human players in Go, Chess, and others. While most RL research focuses on single-agent systems, our work explores the challenges and opportunities of MARL by applying it to the strategic board game Catan, which features complex mutli-agent interactions.

Catan is a turn-based, multiplayer game where players build settlements, cities, and roads on a hexagonal board to gain resources, trade, and achieve strategic goals. Victory is earned by reaching 10 points through infrastructure development, drawing development cards, and attaining milestones like the Longest Road or Largest Army [5]. Its well-defined rules, reliance on strategy, and lack of real-time constraints make it an ideal environment for studying MARL in competitive settings. Similar to skilled human players, MARL agents should be able to infer opponents' goals and adapt their strategies accordingly.

In a typical two-player scenario, a reinforcement learning agent can be trained by playing directly against an adversarial agent and applying Game Theory techniques. Unlike traditional two-player RL settings, multi-agent environments introduce additional complexity. In the single-agent RL setting, the environment, in which the model is trained in, is assumed to be fully stationary and dependent on the agent's action, but the presence of multiple agents leads to a non-stationary environment. Additionally, agents may operate with different objectives, unlike in a two-player zero sum game, adding additional complexity within a MARL setting and makes the direct application of single-agent RL methods challenging.

We aim to train MARL models using algorithms from MARLlib [11] to evaluate their effectiveness in 4-player Catan. Existing projects, such as Settlers-RL [3], have explored single-agent RL approaches like PPO but fail to account for inter-agent interactions and the non-stationary nature of multi-agent settings. By leveraging MARL techniques, we seek to improve on these limitations and advance RL research in complex multiplayer environments.

2 Related Works

2.1 Settlers-RL

Settlers-RL is an existing approach to apply reinforcement learning to Catan. It uses Proximal Policy Optimization (PPO), which aims to optimize the policy function without causing large deviations from the previous old policy [21]. Its model uses a collection of two-layer neural network that processes vectorized environmental observations. It uses a "semi-asynchronous" PPO approach, where each process maintains four policies (one per agent) and updates after acquiring sufficient experience [3].

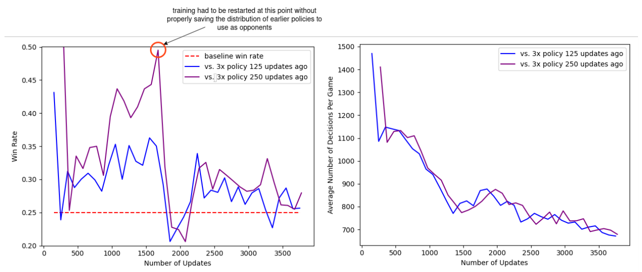

Figure 1: Settlers-RL performance: The left panel shows win rate changes over updates, with performance declining after a crash at 2000 updates. The right panel illustrates a reduction in the average number of decisions per game, indicating increased gameplay efficiency. [3].

While our work builds upon the Settlers-RL Catan implementation, key modifications were made to simplify training and improve scalability. Inter-player trading was removed to eliminate unnecessary decision-making that hindered learning, and the tightly coupled components for human interaction, environment logic, and display rendering were decoupled. Additionally, the PyGame rendering component was made optional, reducing code complexity and speeding up runtime.

2.2 Multi Agent Proximal Policy Optimization (MAPPO)

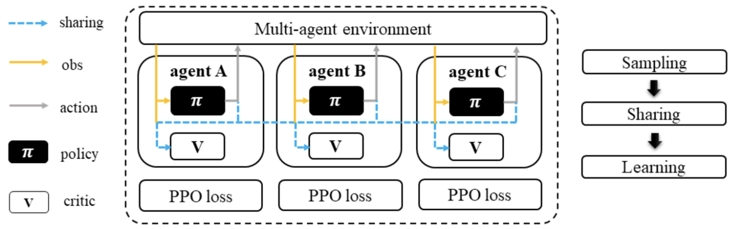

Figure 2: Demonstration of the MAPPO model. [21]

The MAPPO algorithm is a generalized multi-agent algorithm based off of the Proximal Policy Optimization (PPO) algorithm. We choose PPO as it scales better than other algorithms while being more robust to suboptimal hyperparameters. PPO restricts the probability ratio of the new updated and old policy to be between $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter we can adjust.

MAPPO works with a centralized critic and decentralized agents. While each agent only has access to the observation state and their personal development and resource cards, the centralized critic has a view of the entire board and all the agents. It uses this information to create the policy gradients and General Advantage Estimator(GAE). These values are then used to update and improve each agent, as shown in Figure 2. The GAE determines how 'good' the actions played by the agent are compared to a baseline value.

3 Method

3.1 Training infrastructure

All training was done on a cloud virtual machine with a NVIDIA T4 GPU and 32 vCPUs hosted on Google Cloud Platform (GCP). We chose GCP as our training code was CPU bottle-necked while agents collected experience across dozens of games in parallel, while only spending a short time on the gradient calculation and back-propagation step. We encountered quota issues on GCP, not allowing us to request more than 32 vCPUs, and when our GCP credits were used up our virtual machines were immediately (and improperly) closed, interrupting our training and leading to some data loss.

3.2 MARLlib

MARLlib is a Python library implementing several current MARL algorithms, which we use to experiment with the most optimal strategies [12]. The library enables competitive, collaborative, and mixed environments, making it a good fit for our task.

We then determined whether to train our agents using a centralized or decentralized approach. Centralized learning provides agents with access to the global state, the states of other agents, and shared rewards, making it a more efficient learning method. However, it does not accurately reflect the competitive dynamics of Catan. In contrast, decentralized learning emphasizes individual strategies, where each agent operates with its own policy board [1]. To better capture the essence of Catan, we adopted a hybrid approach that combines elements of both methods. In this hybrid environment, agents share some common information while maintaining individual policies for decision-making.

3.3 Environment setup

To simulate Catan, we developed a custom environment by adapting the core gameplay logic from Settlers-RL. This process was challenging due to Settlers-RL's undocumented, tightly coupled, and hard-coded structure, which required substantial restructuring. Despite these obstacles, we successfully simplified the state and action spaces by removing inter-player trading and enabling adjustable victory conditions.

3.3.1 Observation space

The observation space consists of three key components: the board state, the player's state, and other players' states.

Board State: Encodes features such as tile values, resources, and robber placement. Tile features include values and resources, corner features represent buildings, and edge features encode road placements. Bank features compress resource and card counts to reduce complexity while preserving critical information. For example, resource counts are bucketed into ranges ($[0, 2]$, $[3, 5]$, etc.) instead of exact values.

Player State: Includes the agent's achievements (e.g., Longest Road), counts of hidden and public development cards, resources, and building placements. A lookup table maps dice roll probabilities to resource production, summarizing the likelihood of receiving specific resources. This state also tracks the agent's victory points.

Other Players' States: Provide publicly available information such as visible resources, building placements, and relative turn order. Hidden information, like development cards, is excluded.

This comprehensive representation was optimized into a fixed-length array of size 947, with each element tightly bounded. For example, the presence of a robber on a tile is represented by a binary value (0 or 1).

3.3.2 Action space

We designed a simplified action space using seven distinct 'action heads' to represent various decision types in Catan, allowing agents to learn composite actions separately while sharing contextual information. For example, if the agent chooses to roll the dice, we expect—loosely based on Hebbian Theory—that areas of the network associated with related actions, such as moving the robber, will also activate, facilitating more cohesive strategy development. These heads are compressed into a length 172 array similar to the observation space.

Action Type: Encoded as a one-hot vector representing the type of action, such as rolling dice, playing development cards, or trading resources.

Tile Action: Encodes tile selection, used for moving the robber using a one-hot vector of length 19.

Corner Action: Represents settlement or city placement as a one-hot vector of 54 corners.

Edge Action: Encodes road placement decisions using a one-hot vector of 73 edges.

Development Card Action: Encodes the type of development card to be played as a one-hot vector.

Resource Action: Uses two one-hot vectors to represent simultaneous resource selection, such as exchanging or discarding resources.

Player Action: Encodes the target player for actions like resource stealing as a one-hot vector.

3.3.3 Implementation

The environment was implemented using PettingZoo's API [22], an extension of Gymnasium [24] tailored for MARL. Due to the turn-based nature of Catan, we adopted an Agent Environment Cycle (AEC) framework. This allowed agents to perform multiple sequential actions within a single turn, such as rolling dice, buying settlements, and ending their turn. The AEC setup ensured compatibility with our custom environment while supporting dynamic agent interactions.

3.4 Reward Function

Our game environment employs a step-based, dense reward system to provide agents with continuous feedback throughout training episodes, accelerating convergence by encouraging incremental improvements [6].

Through iterative experimentation, we refined the reward function to guide agents toward long-term objectives by rewarding incremental game achievements and promoting an understanding of the rules by penalizing invalid actions.

Algorithm S1 Reward Function Pseudocode

```
for each agent do
    if agent won then
        rewards[agent] += 500
    else if agent lost then
        rewards[agent] -= 100
    end if
    if agent gained victory points then
        rewards[agent] += 5 * (victory points gained)
    end if
    if agent played a development card then
        rewards[agent] += 1
    end if
    if agent moved robber then
        rewards[agent] += 0.5
    end if
    if agent discarded resources then
        rewards[agent] -= 0.25
    end if
    if agent upgraded settlement to city then
        rewards[agent] += 3
    end if
    if agent selected an invalid action then
        rewards[agent] -= 10
    end if
end for
```

4 Experiments

During our experimentation, we underwent multiple changes in direction and hit many dead ends. Our initial goal, applying MARL to Catan, was set aside when we encountered irreconcilable issues with MARLlib. Given our limited remaining time, we decided it would be faster to switch to single-agent reinforcement learning and produce some results, even if it wasn't our original intended objective. Then we attempted to migrate from MARLlib to StableBaselines3 [19], another python library similar to MARLlib, but for single-agent reinforcement learning, but we ran into issues again. In the end, we determined that the migration effort wasn't worth the time, and used a PyTorch PPO implementation for our final results.

4.1 Failed attempts

4.1.1 Mutli-agent training with MARLlib

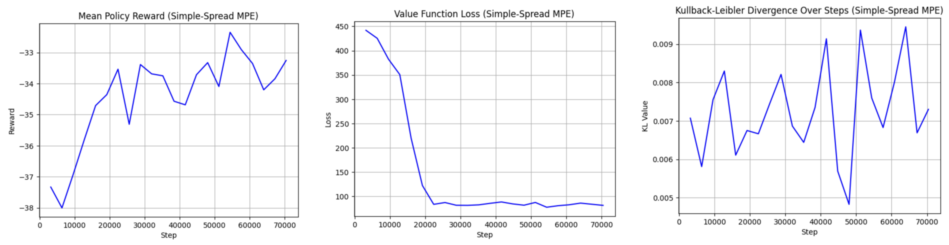

Figure 3: The left panel shows the mean policy reward over timesteps, indicating an overall increase in agent performance as training progresses. The middle panel illustrates the value function loss, which decreases steadily over timesteps, reflecting improved convergence of the value network. The right panel depicts the Kullback-Leibler (KL) divergence between the predicted and true distributions, highlighting fluctuations that suggest periodic overfitting or shifts in exploration strategies.

To evaluate MARLlib's implementation and benchmark results in a controlled setting, we developed a simple MAPPO model within a basic Agent Environment Cycle (AEC) PettingZoo framework. The environment simulated a multi-particle "simple-spread" task, where agents coordinate to cover predefined landmarks while avoiding collisions [7]. This setup was implicitly adapted into a parallel environment to align with MARLlib's requirements, inadvertently masking later challenges posed by turn-based dynamics in non-parallel environments like Catan

The MAPPO model was trained over 100,000 timesteps using the default MARLlib implementation. The underlying architecture consisted of a Multilayer Perceptron with two encoder layers (128 and 256 neurons), demonstrating successful convergence as shown in Figure 3. These preliminary results validated MARLlib's effectiveness in synchronous environments and gave confidence to move forward with MARLlib.

However, integrating MAPPO with our simplified Catan environment proved challenging due to several dependency issues. We resolved these by downgrading to an earlier Python version, rewriting incomparable code, and manually patching key libraries, including Gym and NumPy. Despite these efforts, integrating MARLlib exposed a deeper limitation: it assumes, and requires, parallel agent training, contrary to its documentation. This assumption was particularly problematic for Catan, where agents act sequentially, and the environment state evolves between each agent's turn. MARLlib's parallel training paradigm caused agents to make invalid moves because their observations often became outdated by the time actions were taken. Even with robust action masking to prevent illegal moves, the agents failed to adapt to this dynamic, non-stationary state. As a result, all tested MARLlib algorithms, including MAPPO, MAA2C, MADDPG, and MATRPO, failed to converge during training.

To mitigate these issues, we attempted several adjustments. We introduced penalties for invalid moves to encourage agents to learn valid actions. Additionally, we experimented with simplified reward functions and alternative observation encodings to reduce state-space complexity. Inspired by recommendations in [13], we explored hyperparameter tuning and parameter sharing tricks to improve robustness, but these interventions proved insufficient. Despite these efforts, the parallel training assumption inherent to MARLlib persisted as a fundamental barrier, ultimately rendering the library unsuitable for our turn-based Catan environment.

4.1.2 Single-agent training with StableBaselines3

Given the challenges encountered with MARLlib, we shifted our focus to experimenting with single-agent reinforcement learning using StableBaselines3. Although our original objective was to explore multi-agent reinforcement learning, we hypothesized that insights from single-agent RL could still provide valuable results within the project's time constraints.

However, migrating our code to StableBaselines3 introduced new challenges, particularly with action masking. We observed persistent bugs that likely stemmed from differences in how MARL environments, like ours, dynamically manage the 'active' agent compared to the assumptions made by StableBaselines3. Specifically, StableBaselines3 appeared to expect action masks for a new agent at each step, whereas our AEC framework allowed the same agent to perform multiple sequential actions within a turn. For example, in Catan, Player 1 might roll the dice, play a development card, purchase a city, and then end their turn—resulting in four separate actions within a single agent's turn. This discrepancy led to incorrect masking of invalid actions, hindering effective training. Consequently, we were unable to successfully train an agent in our simplified Catan environment using StableBaselines3.

4.2 Final results

Despite the challenges encountered with MARLlib and StableBaselines3, we successfully implemented a simplified reinforcement learning model using a pure PyTorch implementation of Proximal Policy Optimization (PPO). This approach provided greater flexibility in circumventing the compatibility issues of other frameworks. To adapt our model to the computational complexity of Catan, we developed a custom wrapper for our PettingZoo environment to integrate seamlessly with the PyTorch agent.

We ran two separate experiments: one with a victory point requirement of 10 and another with a reduced requirement of 4 to facilitate faster training. Initially, we intended to employ a curriculum learning strategy by transitioning from 4 to 10 victory points, allowing the model to progressively learn more complex strategies. However, this plan was curtailed due to time constraints, particularly after our GCP credits depleted before the step-up phase began, terminating our training early.

Through iterative experimentation and guidance from prior works [13], we fine-tuned the hyperparameters resulting in the values shown in Table 1. The results, though suboptimal in achieving high performance, provided valuable insights:

Training Loss (10-Point Catan): As shown in Figure 4, the entropy loss decreased significantly over updates, indicating improved decisiveness in the model's policy. Action loss also declined slightly, suggesting some optimization in policy updates. However, validation loss remained stagnant, pointing to limited generalization to unseen states.

Game Efficiency (10-Point Catan): The model demonstrated a clear reduction in average game length and the number of decisions per game, signifying that agents learned to play more efficiently.

Evaluation Win Rate and Victory Points (10-Point Catan): Figure 5 illustrates that while the average win rate showed marginal improvement, the agents consistently achieved higher average victory points across games. This trend suggests that the agents learned to optimize their gameplay strategies even if they struggled to consistently win.

Policy Evaluation Performance Over Time (4-Point Catan): Figure 6 shows that agents trained with a 4-point victory condition improved significantly at defeating random policies over training updates. However, their ability to defeat their own policies from earlier iterations (e.g., 25, 50, and 100 updates ago) diminished over time. This divergence suggests that agents struggled to generalize against progressively better opposing strategies.

Insights on Reward Functions: Across both experiments, reward function design significantly influenced agent behavior. For example, smaller rewards for incremental actions like settlement building encouraged achieving victory points, but larger incentives for winning or achieving strategic goals could potentially enhance end-game performance.

Implications for Single-Agent RL: The decline in self-policy performance highlights a potential limitation of single-agent RL in multiplayer settings like Catan. Without explicit consideration of dynamic inter-player strategies, the agents likely overfit to the static behavior of the random policy, failing to develop generalized strategies, reinforcing the necessity of MARL to address such interdependencies.

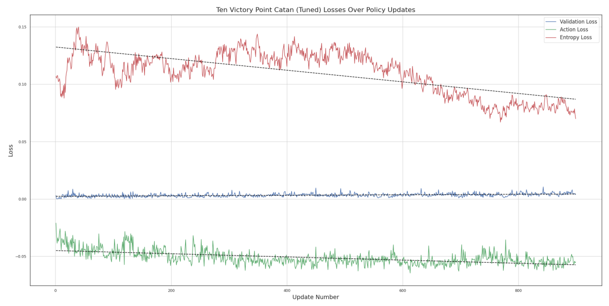

Figure 4: **(above)** Training loss metrics over updates for 10-point Catan using tuned hyperparameters.
(right) Every 25 policy updates our agent plays 32 full games against random policies and earlier iterations (25, 50, and 100 updates prior). The average performances are plotted here, with a full grid available in Appendix A.

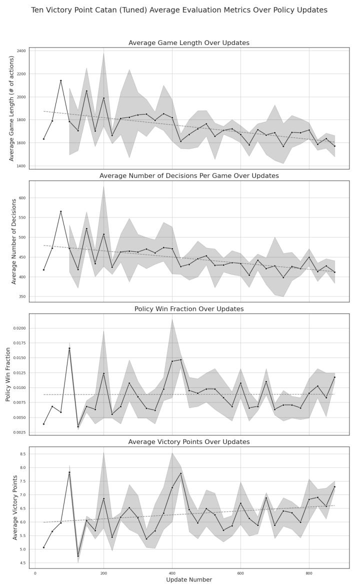

Figure 5: Average evaluation metrics for agents in 10-point Catan with tuned hyperparameters.

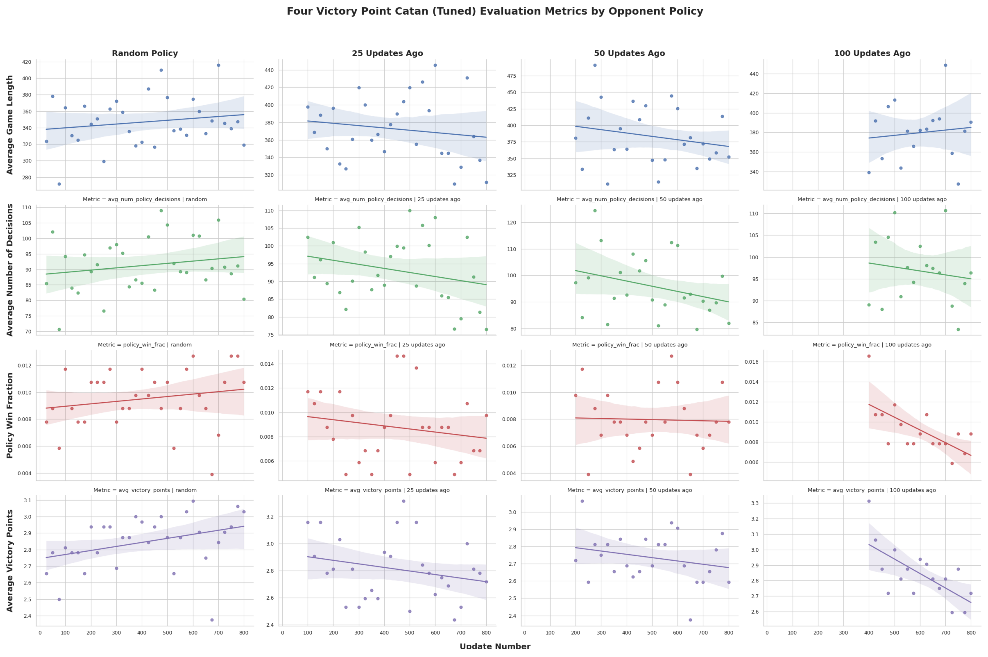

Figure 6: Performance evaluation metrics in 4-Point Catan, tested against random policies and earlier iterations (25, 50, and 100 updates prior). Performance was averaged across 32 evaluation games every 25 policy updates per opponent policy.

Table 1: Hyperparameters used in the model with their descriptions and values.

Hyperparameter	**Description**	**Value**	**Comment**
Environments in Parallel	The number of games to run in parallel for each batch.	160	More processes is better, constrained by cost & GCP CPU types; 32 processes (5 environments each).
Steps	How many steps of experience to collect before a policy update.	30	Higher value reduces stochastically and GPU load.
Epochs	Epochs needed to train PPO actor.	15	Adjusted to fit the model.
Minibatches	Minibatches per PPO epoch.	16	Adjusted to fit the model
Discount Factor (γ)	Discount factor for future rewards: $\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$.	0.999	Slightly less than 1 for stability but high enough to optimize over iterations.
Generalized Advantage Estimation (GAE) Parameter (λ)	Balances bias and variance in expected returns.	0.95	Default value; with more time we might have tuned.
4 Clipping Parameter (ϵ)	Max. allowed gradient update. Prevents overfitting.	0.2	Retained as 0.2 worked well in tests.
Recompute Returns	Boolean determining whether to apply GAE.	True	Ensures estimates are updated with policy and value network changes.
Entropy Loss Coefficient	Adjusts how much entropy influences the overall loss.	0.01	Default value; with more time we might have tuned.

5 Conclusion

This study explored the challenges and potential of using RL to train agents for playing Catan, a complex multiplayer board game. While our initial aim was to implement MARL, the inherent limitations of MARLlib, the sequential dynamics of Catan, and time constraints necessitated a pivot to single-agent RL with PPO.

Despite the constraints, our experiments demonstrated incremental improvements in agent strategy and gameplay efficiency using single-agent PPO. Agents optimized decision-making and achieved higher average victory points over time, though struggled to win in the end-game. Furthermore, agents failed to improve relative to past policies due to difficulty capturing inter-player strategy. Our findings underscore the importance of MARL for capturing non-stationary inter-player interactions in competitive settings.

5.1 Future Work

To address the limitations identified in this study, future work should focus on the following:

- **Short-Term (1–2 weeks):** Finalize a stable single-agent RL baseline using frameworks like StableBaselines3, refining masking mechanisms and action encodings.
- **Mid-Term (1–2 months):** Implement MARL algorithms such as MAPPO or MAA2C in PyTorch, designed specifically for turn-based AEC environments. Gradually increase the complexity of training via curriculum learning, starting with simpler sub-tasks (e.g., road placement).
- **Long-Term (3–6 months):** Incorporate forward search methods like Monte Carlo Tree Search to enhance planning and decision-making [15]. Introduce human-in-the-loop training for interactive debugging and imitation learning.
- **Exploratory Directions (Semester):** Explore asynchronous environments or adapt other strategic games like 7 Wonders to evaluate MARL frameworks in both synchronous and asynchronous settings.

References

[1] Morteza Hashemi Amin Shojaeighadikolaei, Zsolt Talata. Centralized vs. decentralized multi-agent reinforcement learning for enhanced control of electric vehicle charging networks, 2023.

[2] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study, 2020.

[3] Henry Charlesworth. Learning to play settlers of catan with deep reinforcement learning, 2024.

[4] Caleb Compton. The ultimate catan strategy guide - top tips to win more at catan, 2021.

[5] Melanie Davis. How to play settlers of catan: Rules, setup, and strategies explained, 2023.

[6] Andrew Forney. Practical rl, 2020.

[7] Farama Foundation. Simple spread, 2023.

[8] Kornberg Fresnel. Malib: A parallel framework for population-based reinforcement learning, 2022.

[9] Quentin Gendre and Tomoyuki Kaneko. Playing catan with cross-dimensional neural network, 2020.

[10] GSJProgo. Reinforcement learning for catan: Dqn/a3c implementation, 2022.

[11] Siyi Hu, Yifan Zhong, Minquan Gao, Weixun Wang, Hao Dong, Xiaodan Liang, Zhihui Li, Xiaojun Chang, and Yaodong Yang. Marllib: A multi-agent reinforcement learning library, 2023.

[12] Siyi Hu, Yifan Zhong, Minquan Gao, Weixun Wang, Hao Dong, Xiaodan Liang, Zhihui Li, Xiaojun Chang, and Yaodong Yang. Marllib: A scalable and efficient multi-agent reinforcement learning library, 2023.

[13] Xiaoqian Li, Peijun Ju, and Zonglei Jing. Can tricks affect robustness of multi-agent reinforcement learning? In *Proceedings of the 43rd Chinese Control Conference (CCC)*, pages 9022–9027. IEEE, 2024.

[14] Anji Liu, Jianshu Chen, Mingze Yu, Yu Zhai, Xuewen Zhou, and Ji Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search, 2020.

[15] Michael Liu. General game-playing with monte carlo tree search, 2017.

[16] Robert Moni. Reinforcement learning algorithms — an intuitive overview, 2019.

[17] Player One. Analyzing settlers of catan, 2017.

[18] James Hahn Shreeshaa Kulkarni Peter McAughan, Arvind Krishnakumar. Qsettlers: Deep reinforcement learning for settlers of catan, 2021.

[19] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[20] Replicable-MARL. Advanced actor critic family, 2023.

[21] Replicable-MARL. Proximal policy optimization family, 2023.

[22] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.

[23] Klaus Teuber. Catan: Game rules and almanac, 2020.

[24] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[25] Ivan Zhong. Heterogeneous-agent reinforcement learning, 2024.

A Appendix

Supplementary Resources:
The code and additional resources for this project are available at the following GitHub repository:
`https://github.com/Catan-RL/catan-rl/tree/master`

A demonstration of the agent's gameplay behavior is available as a GIF:
`https://github.com/Catan-RL/catan-rl/blob/master/gameplay-rl-agents-full.gif`

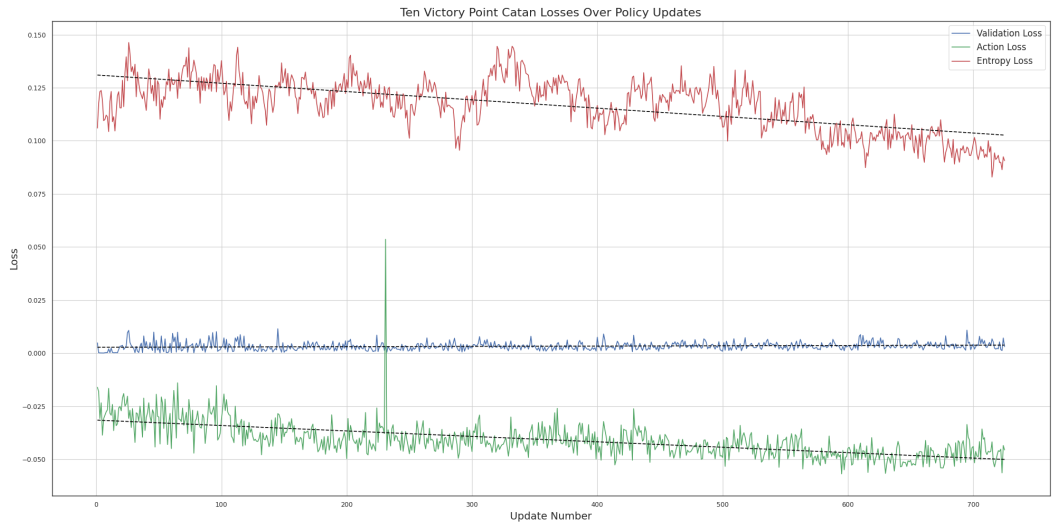

Figure 7: Loss over time during initial training session.

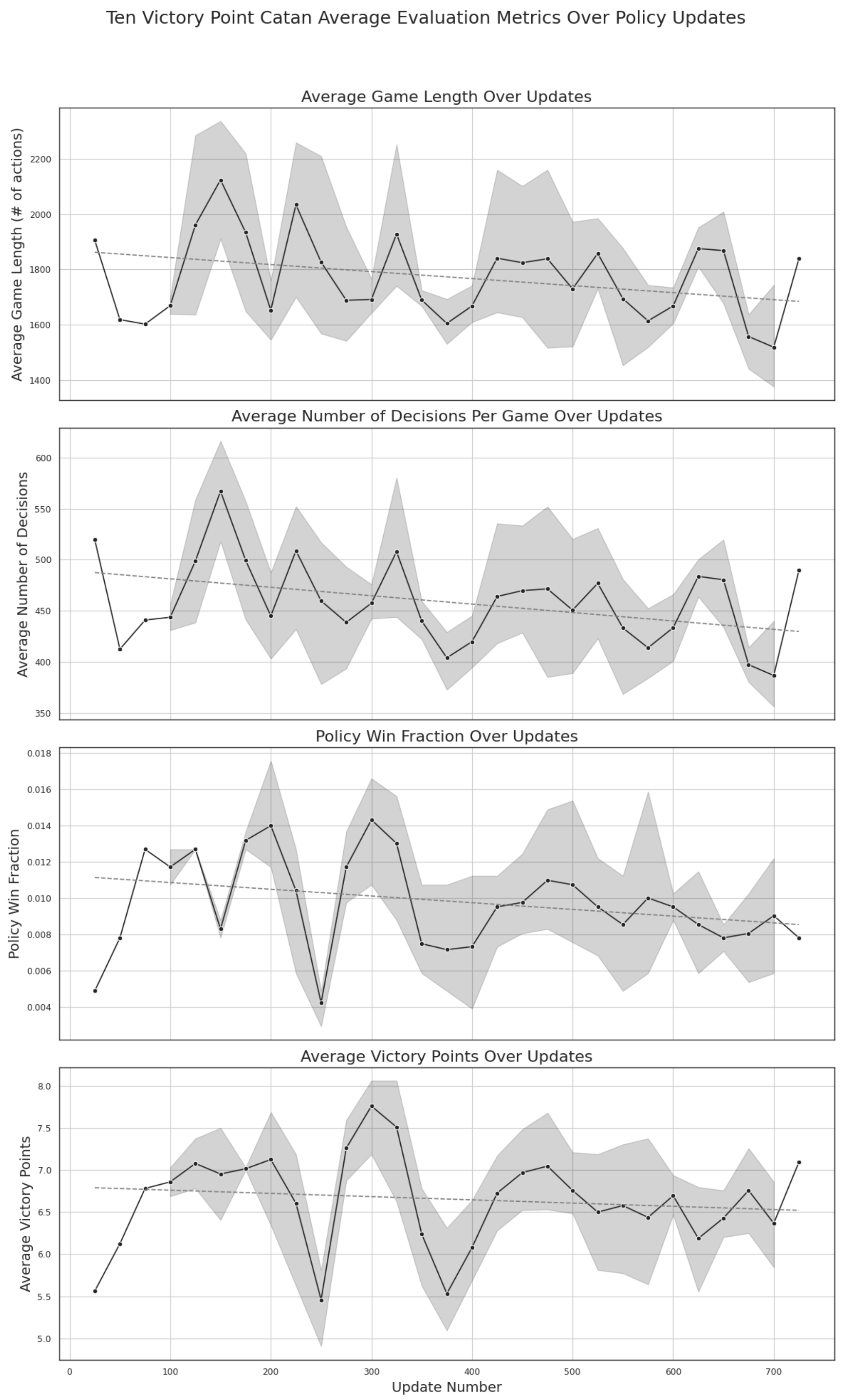

Figure 8: Average evaluation metrics across various points during initial training session, measured every fixed interval.

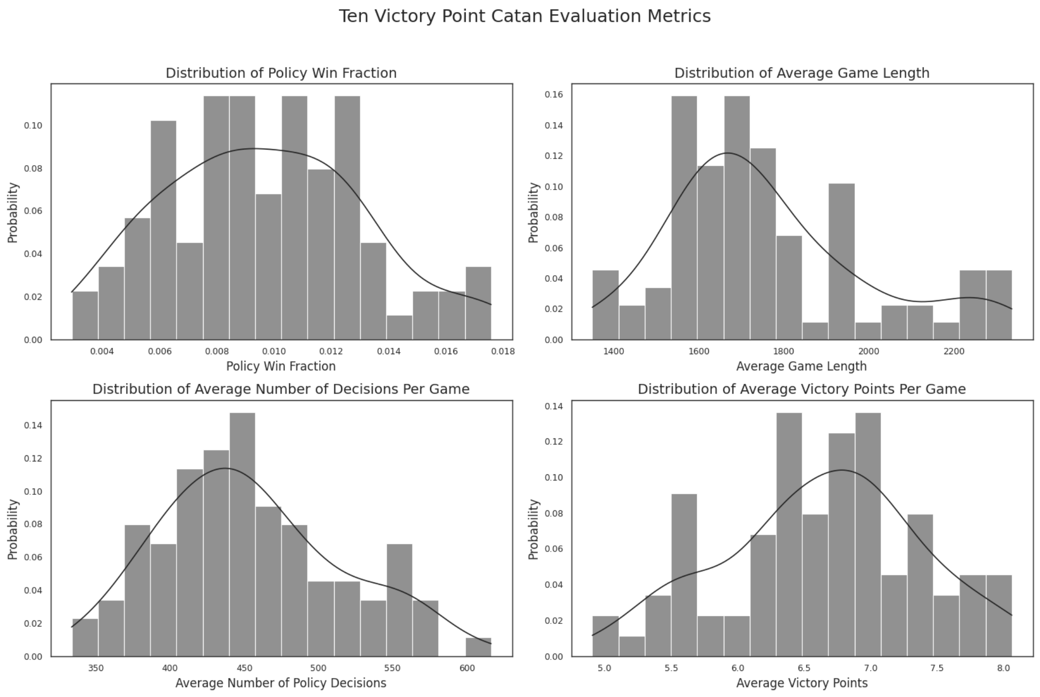

Figure 9: Distribution of various evaluation metrics throughout initial training session.

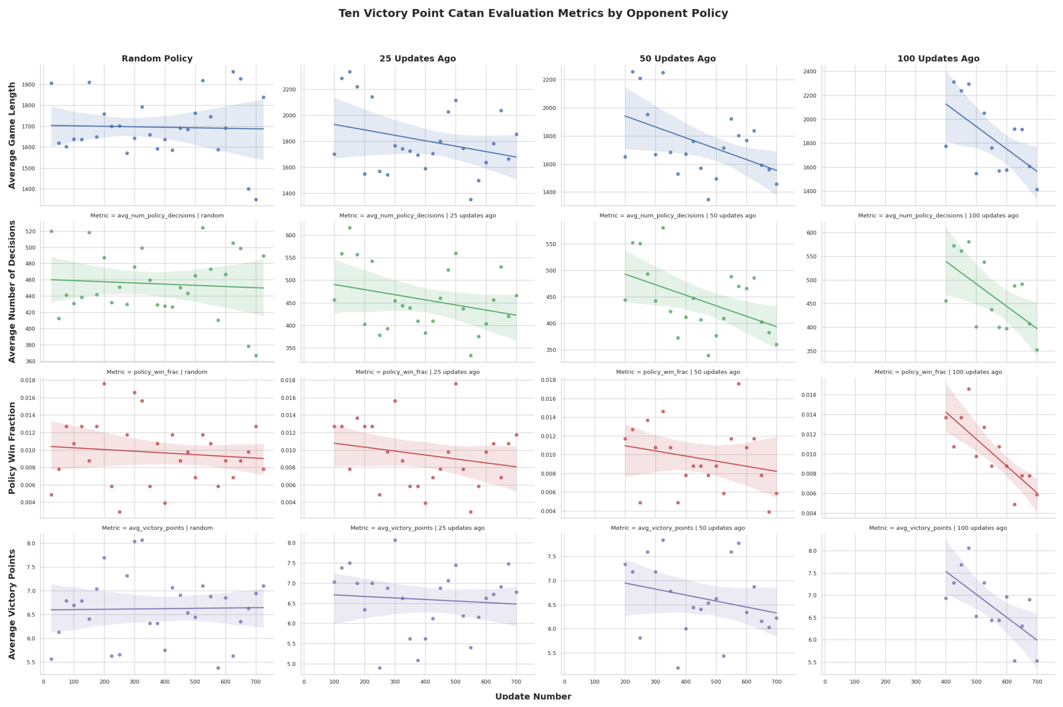

Figure 10: Evaluation performance metrics against random policy and prior updates over time during initial training session.

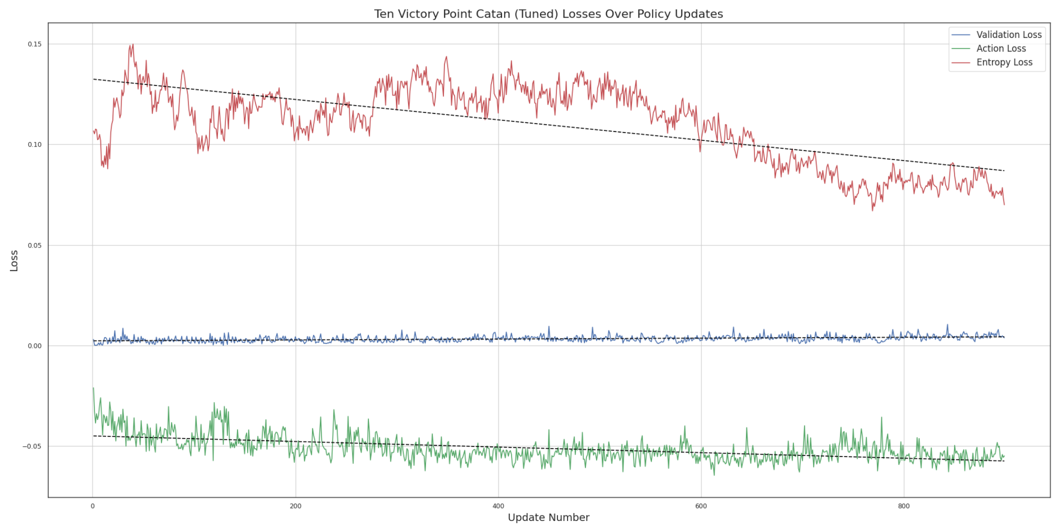

Figure 11: Loss over time during training after tuning parameters.

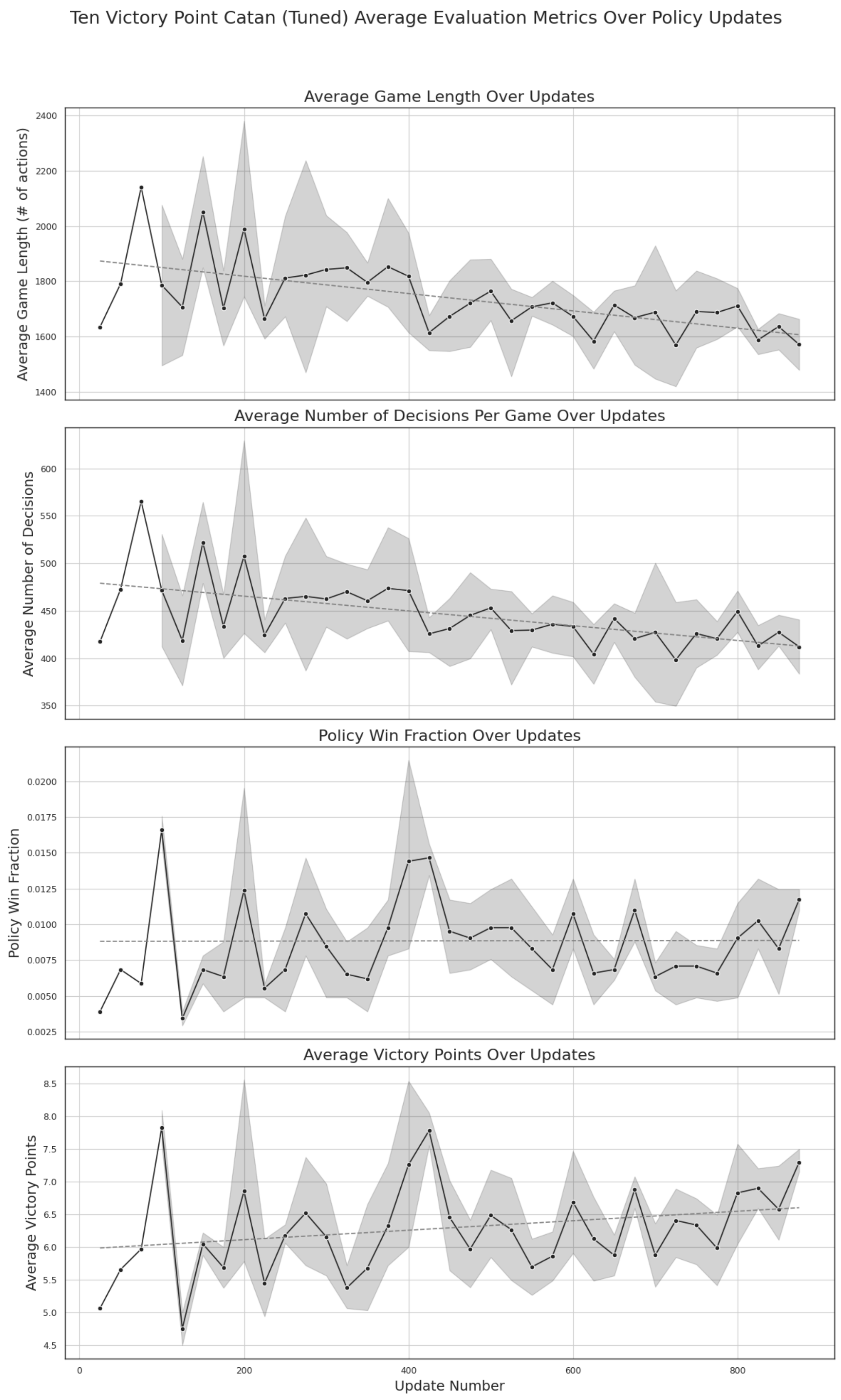

Figure 12: Average evaluation metrics across various points during training after tuning parameters, measured every fixed interval.

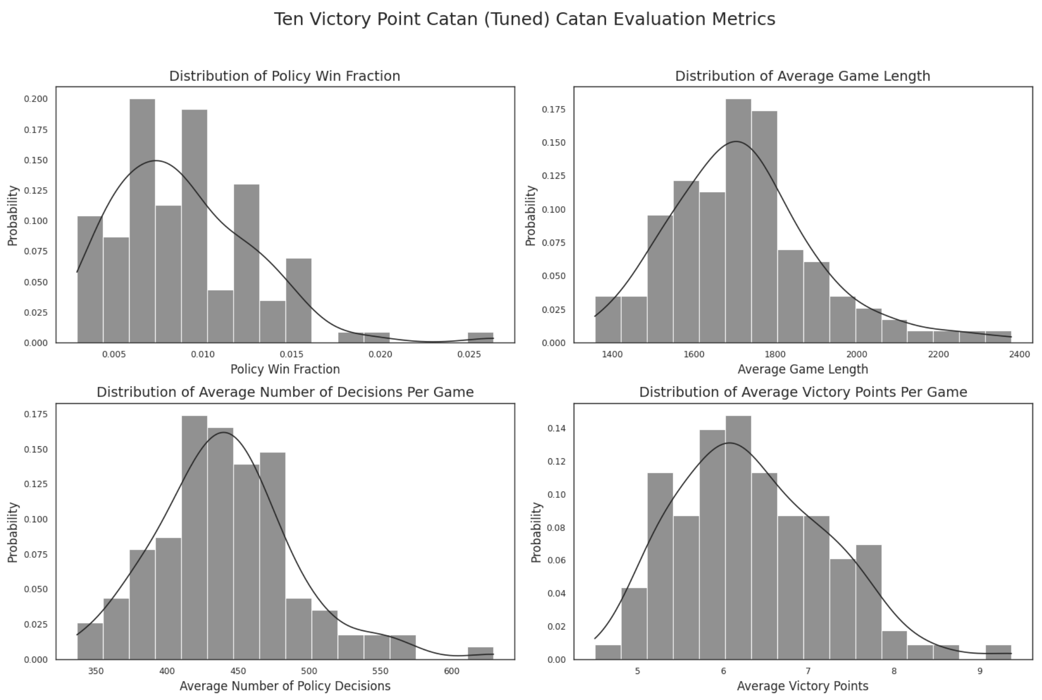

Figure 13: Distribution of various evaluation metrics throughout training after tuning parameters.

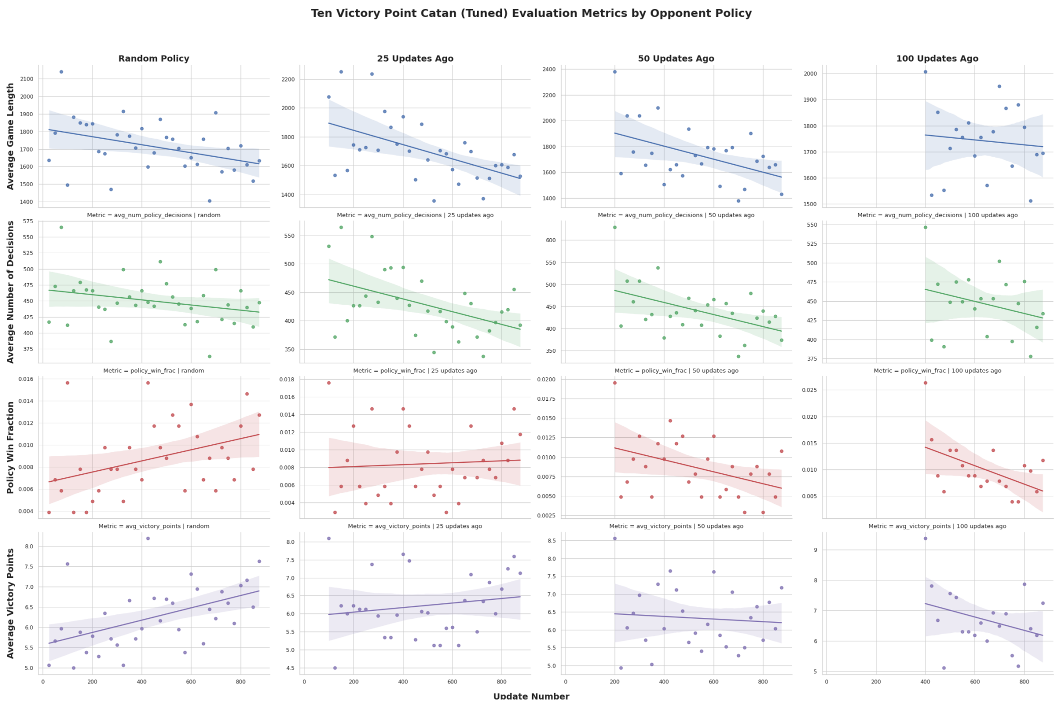

Figure 14: Evaluation performance metrics against random policy and prior updates over time during training after tuning parameters.

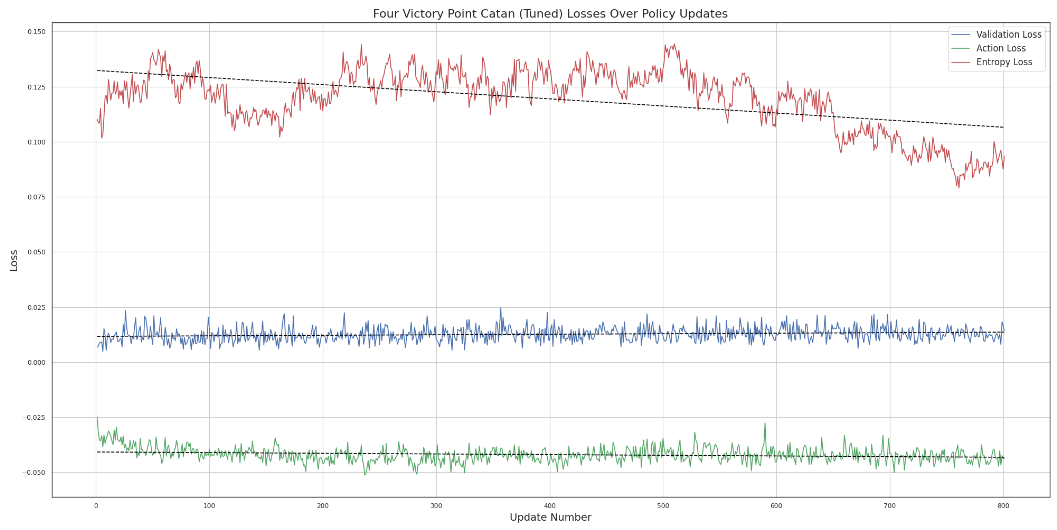

Figure 15: Loss over time during training with accelerated win conditions (reducing victory point requirement from 10 to 4).

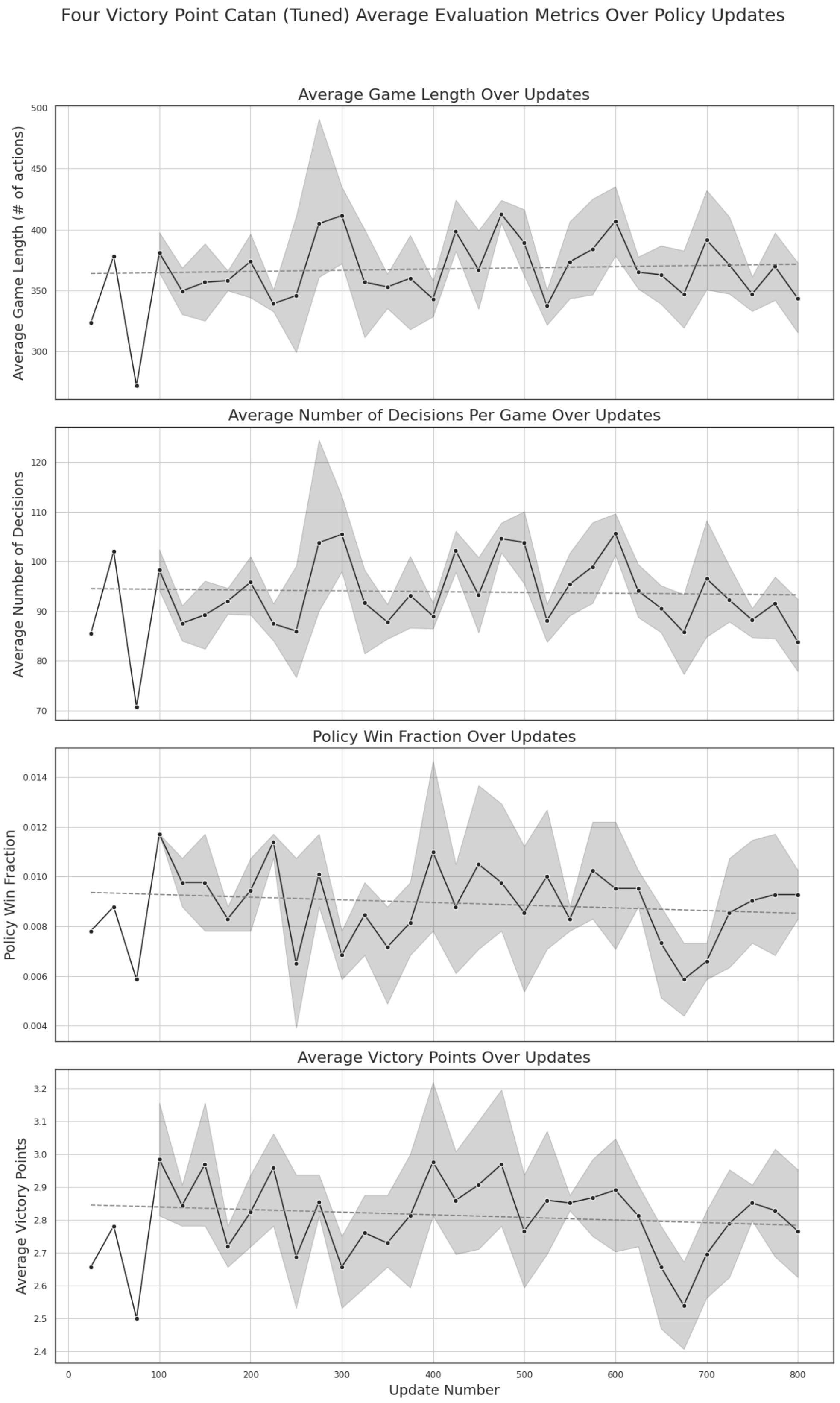

Figure 16: Average evaluation metrics across various points during training with accelerated win conditions (reducing victory point requirement from 10 to 4), measured every fixed interval.

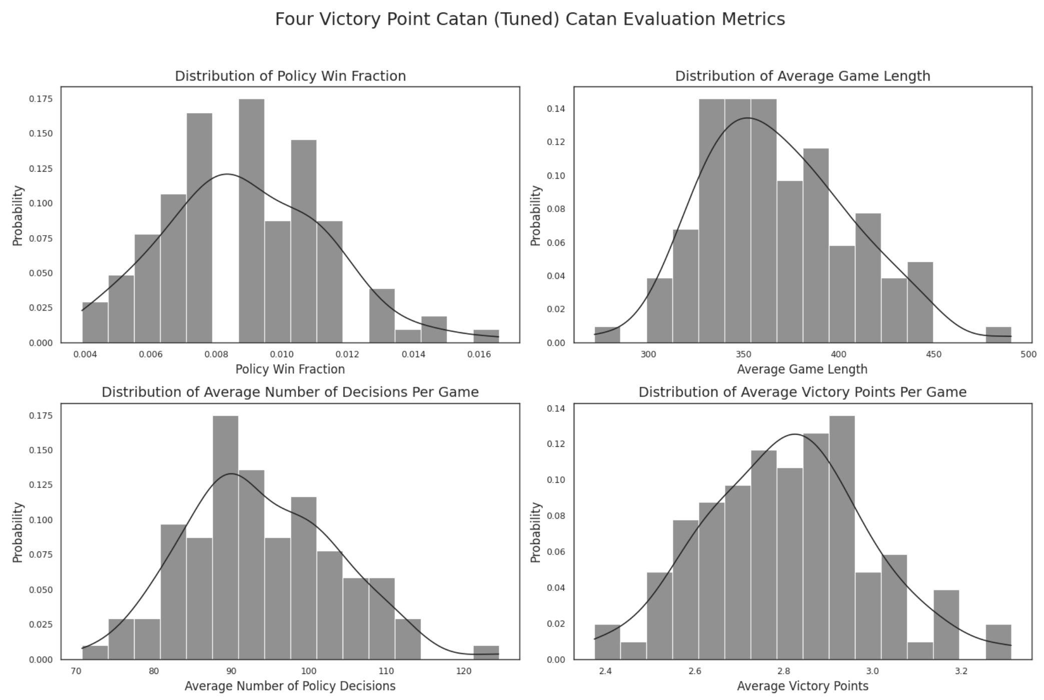

Figure 17: Distribution of various evaluation metrics throughout training with accelerated win conditions (reducing victory point requirement from 10 to 4).

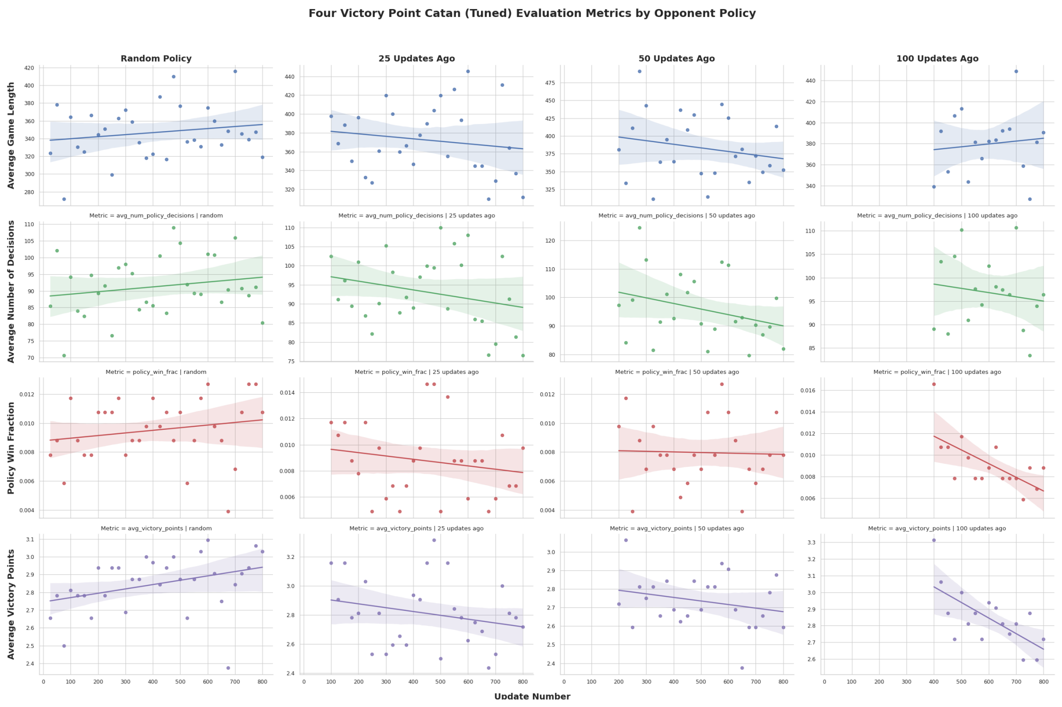

Figure 18: Evaluation performance metrics against random policy and prior updates over time during training with accelerated win conditions (reducing victory point requirement from 10 to 4).