Reinforcement Learning in Catan

Matthew Whelan
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
mw3shc@virginia.edu

Nitin Maddi
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
nm2eed@virginia.edu

Sidhardh Burre
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
ssb3vk@virginia.edu

Abstract

This paper presents an in-depth analysis of various Reinforcement Learning (RL) models applied to the strategic board game, Settlers of Catan. The study investigates the efficacy of different Deep Reinforcement Learning architectures including Deep Q-Learning (DQN), Double DQN (DDQN), and Dueling Networks (DN) across several model sizes and with distinct reward policies. By simulating thousands of games, we generate data to train our models, focusing on optimizing their decision-making capabilities in a game characterized by a large action space and stochastic elements without resource trading. Key findings suggest that the algorithmic choice significantly impacts performance, with Dueling Networks showing superior results. The study further highlights the importance of reward function design, revealing that simpler terminal-based rewards often perform as effectively as more complex schemes. Additionally, the effectiveness of Monte Carlo Tree Search (MCTS) in improving strategic decision-making in RL is demonstrated, albeit at a high computational cost. This paper contributes to the understanding of RL in board games with complex strategic interactions and provides a foundation for future explorations into advanced RL applications in similar environments.

1 Introduction

Reinforcement Learning (RL) distinguishes itself from traditional machine learning paradigms by its ability to learn optimal actions from trial and error interactions within a dynamic environment. Unlike supervised learning that relies on labeled data, RL utilizes an agent that makes decisions, receives feedback through rewards or penalties, and adjusts its strategy accordingly. Notably, RL has been successfully applied to strategic games like Poker and Go, where artificial agents have reached superhuman performance levels. Inspired by these achievements, our research focuses on adapting RL techniques to excel in "Settlers of Catan", a game that combines strategy, randomness, and player interaction in a unique setup.

"Settlers of Catan" presents a multifaceted challenge for AI due to its combination of random elements, such as dice rolls that influence resource availability, and a complex interaction system involving trade and player negotiation. Unlike deterministic games with complete information, such as Chess, Catan incorporates hidden information and chance, adding layers of complexity to the decision-making process. The game is played with a goal of accumulating victory points, which are gained by building settlements and cities, acquiring development cards, and trading resources effectively. The path to

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

victory is nonlinear and can vary significantly with each game session, reflecting a dynamic strategic landscape that is ideal for studying RL's adaptability and strategic depth.

Large action space, sparse feedback, and complex state spaces are highly analogous to modern recommendation systems with one large difference: Catan has verifiable rewards. In recommendation systems it is difficult to make arguments about the optimality of a recommendation [1] due to the lack of ground truth feedback. But in the Catan game, a win or loss provides a ground truth indicator of the model's performance. Therefore, understanding the dynamics of Reinforcement Learning applied to Catan can provide an indicator towards more difficult settings.

This studies highlights a variety of key results based on the experiments ran. **Algorithm Efficiency**: The choice of algorithm is crucial, with more powerful algorithms like DN significantly outperforming more simple ones like Vanilla DQN. **Reward Functions**: Constructing effective reward functions is challenging, and simpler reward functions tend to be more effective. **Model Size**: Bigger models do not automatically lead to better performance, indicating that efficiency doesn't always scale with size. **MCTS Usage**: Monte Carlo Tree Search (MCTS) is highly effective despite its high computational demands.

2 Related Works

2.1 Reinforcement Learning for Board Games

A pivotal work in the domain of game-solving algorithms is AlphaZero, developed by Google DeepMind. AlphaZero represents a general-purpose algorithm designed to tackle a wide array of games, specifically those classified as perfect information games where all players have complete knowledge of the game state [9]. Unlike these games, Catan features imperfect information, presenting unique challenges not directly addressed by AlphaZero. Nonetheless, the techniques described in Silver's work offer valuable insights for developing highly effective reinforcement learning strategies for Catan, potentially capable of surpassing human performance.

Originally an evolution of the AlphaGo algorithm, which famously defeated European Go champion Fan Hui in 2015 [8], AlphaZero advances beyond its predecessor by integrating a dual-component system comprising a policy network and a value network. The policy network is initially trained through supervised learning to mimic human expert moves, then refined using policy-gradient reinforcement learning to optimize play strategies.

A key innovation in AlphaZero is the application of the Upper Confidence Bound (UCB) approach combined with Monte Carlo Tree Search (MCTS). This methodology prioritizes the exploration of less frequently visited states using a value function augmented by a confidence interval, thus balancing exploration and exploitation efficiently. Moreover, AlphaZero employs deep neural networks (DNNs) to interpret the game board analogously to image processing, enabling it to discern patterns and learn game dynamics without direct exposure to every possible game state.

In technical terms, the DNN in AlphaZero, denoted as $(\mathbf{p}, v) = f_\theta(s)$, processes the board state s to output both a vector of move probabilities $\mathbf{p}$, with each component corresponding to a potential action a, and a scalar value v representing the expected game outcome $\mathbb{E}[z|s]$. Through simulations of self-play using MCTS, AlphaZero generates a probability distribution over potential moves, represented by vector π, choosing actions based on their low visit count, high probability, and high estimated value. The neural network parameters Θ are continually adjusted to minimize the discrepancy between predicted and actual game outcomes, and to align the predicted policy vector $\mathbf{p}_t$ closely with the empirical search probabilities π_t.

2.2 Catan

To give background on how Catan works, it is a hexagonal board with 19 squares of four different resources as seen in Figure 1. Players use these resources to build new settlements, cities, roads, and developments. The goal of the game is to be the first player who gains at least 10 *victory points*. While the goal is relatively simple, there are a plethora of options to take in each turn. In one turn, its possible to have over 200 actions to take, which greatly increases the complexity of the algorithm.

The composition of the Catan game board and reliance on dice rolls to obtain resources and progress in-game introduces a large degree of variance. This greatly impacts decision making within the game.

While mean-wise, certain actions may be optimal (such as placing a settlement on a high-yield tile), simply due to variance, the outcome may never reach the expected outcome of decision. Further, with positioning being such a large component of the game, understanding and interpreting the board to select optimal settlement placements is essential to playing the game with significant skill.

The first online AI interface to play Catan was created by Monin more than a decade ago and is still actively updated. This Java-based program includes a complete implementation of the game and uses a fully rule-based AI with predefined decisions and heuristics. Most current research on Settlers of Catan benchmarks against the JSettlers agent.

Szita et al. explored Monte-Carlo Tree Search (MCTS) in Catan, but removed imperfect information and applied to 4-player settings. The implementation reaches 27% winrate with 1000 simulations, and 49% winrate with 10000 simulations, when playing against 3 JSettlers [10]. The work by Gendre et al. [4] explores the application of Cross-Dimensional Neural Networks to Catan, using JSettlers for evaluation

2.3 Deep-Q Learning in Catan

Cuayáhuitl et al. uses a Deep-Q Learning in specifically the decision-making process of Catan, where agents need to negotiate trades effectively [3]. They use a JSettlers agent as a base, and use an experience replay mechanism to enhance learning. This reaches a 53% win rate against 3 heuristic JSettlers. Xenou et al. also uses Deep-Q Learning but uses an action-dependent Q-value computation by implementing parallel LSTM components in the neural network, each responsible for computing a Q-value associated with a specific action [14]. In this way, the authors avoid using a "target" network or experience replay buffer like common DRL literature, while also reaching a win rate of 53% against 3 JSettlers.

QSettlers only alters the trading mechanics in the game, so the building decisions made are suboptimal and therefore their agent does not reach a high win rate against bots [6]. However, they re-implemented the original JSettlers code in Python with a simplfied infrastructure, which made it easier to use and extend upon. The work by Kim et al. [**?**] builds off of QSettlers by adding multiple specialized DQN models for different phases of the game (e.g., Placement, Building, Trading). This approach, though, only reached around 32% with experience replay and dropout regularization.

One of the most significant advancements in computational strategies for the game of Catan is detailed in the Catanatron repository by Collazo [2]. This work encompasses the development of a comprehensive model of the game implemented in Python, designed to facilitate the construction of algorithms capable of playing Catan. The architecture provided supports numerous functionalities, including the enumeration of all possible moves at any given point, which can number in the hundreds, and an extensive debugging framework to enhance algorithm performance assessment. Moreover, the repository includes a meticulous game log that records game states, actions executed, and rewards obtained, which is instrumental for deploying off-policy reinforcement learning (RL) algorithms to learn from strategies executed by non-RL based models.

Although Collazo did not achieve the creation of a superhuman AI for Catan, his efforts led to the development of algorithms that substantially outperform those making random decisions. One of his most effective models employs a manually adjusted value function to select the most advantageous board position, adhering to a straightforward greedy strategy. This model consistently surpassed both random and weighted random decision-making strategies. Furthermore, he utilized this value function as a heuristic in the implementation of Alpha-Beta pruning to navigate the game tree, considering future moves. However, due to the extensive branching factor inherent in Catan, only a search depth of two was feasible without incurring prohibitive computational costs. Although these methods greatly enhanced the AI's performance, they did not elevate it to a superhuman level.

3 Preliminaries

In this section we introduce some of the algorithms we exercise as part of our experiments including a introduction to the various algorithms used in our research.

3.1 Q-Learning

Q-learning, a fundamental reinforcement learning technique, was introduced by Watkins and Dayan in 1992 and has since become pivotal in the field. This off-policy learning algorithm allows an agent to learn the value of an action a in a particular state s using the Bellman equation, which forms the backbone for many reinforcement strategies [13]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{1}$$

where s_t and s_{t+1} are the current and next states, a_t is the action taken, r_{t+1} is the reward received, α is the learning rate, γ is the discount factor, and $\max_a Q(s_{t+1}, a)$ represents the maximum expected utility for the next state.

3.2 Deep Q-Learning

Instead of using a table to store Q-values for each state-action pair, Deep Q-Learning (DQN) employs a neural network, often a convolutional neural network (CNN), to approximate the Q-value function. This Q-network inputs a state and outputs Q-values for all possible actions, effectively handling large or continuous state spaces without needing a massive Q-table [7].

This paradigm shift introduces techniques to enhance stability and convergence when using neural networks: **experience replay** and a **Q-target network**. The former stores the agent's experiences at each time step in a replay buffer and randomly samples mini-batches from this buffer to update the network. The latter is a separate network to generate the Q-value targets that are used in the update step. This provides a fixed baseline for the Q-values, which helps in stabilizing the updates to the Q-network. These innovations allow Deep Q-learning to efficiently handle complex environments with high-dimensional input spaces, like video games, where traditional Q-learning would struggle due to its reliance on a discrete state-action space representation.

3.3 Double-Q Learning

Double Q-Learning (DDQN) was created by Van Hasselt to address the inherent bias in traditional Q-learning, which tends to overestimate action values, particularly in stochastic environments. This bias can lead to suboptimal policy decisions, as Q-Learning might settle prematurely on overvalued actions without exploring potentially better options [11].

Double Q-Learning employs two distinct Q-value functions, Q_A and Q_B. Each function updates its values using the output from the other function, effectively decoupling the action selection from the target value generation. This method reduces the overestimation bias by averaging the updates from two independently learned estimators. The update rules for Double Q-Learning are as follows:

For Q_A:

$$Q_A(s, a) \leftarrow Q_A(s, a) + \alpha \left[R + \gamma Q_B(s', a') - Q_A(s, a) \right]$$

For Q_B:

$$Q_B(s, a) \leftarrow Q_B(s, a) + \alpha \left[R + \gamma Q_A(s', a') - Q_B(s, a) \right]$$

3.4 Dueling Networks

Dueling Networks (DN) introduce a novel neural network architecture for estimating Q-values. The network is split into two streams: one for estimating the state value function $V(s)$ and the other for estimating the advantage function $A(s, a)$ [12]. This architecture helps in learning which states are valuable without having to learn the effect of each action at each state.

The network outputs both the value $V(s)$ and the advantages $A(s, a)$ for each action, which are then combined to estimate the Q-values using a special aggregation formula:

$$Q(s, a) = V(s) + \left(A(s, a) - \max_{a'} A(s, a') \right)$$

An alternate formulation uses the mean instead of the max:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|} \sum_{a' \in A} A(s,a') \right)$$

This structure allows the network to make more informed decisions by balancing the importance of the state and the actions.

4 Methods

In the following section, we describe our methods in relation to each aspect of the project. This includes our Catan Simulator, the different RL algorithms used, the different Model Sizes, and the Reward Functions. We begin with an overview of our approach.

We experimented with Reinforcement Learning based approaches to achieve a reasonable playing ability in the Catan board game. We aim to improve upon the performance of the strongest strategy developed by Collazo, the Alpha Beta Pruning algorithm. We retrieve the final reward based on the win or loss outcome of the game. By simulating thousands of games against the Alpha Beta strategy, we learn the value of context dependent actions. Finally, we deploy trained models in an MCTS algorithm and inspect win rates.

4.1 Catan Simulator

A Github repo created by Bryan Collazo [2] features a faithful implementation of the Catan board game with all the core aspects of the game implemented. Further, the repository is well-documented enabling modification for our needs. Finally, Collazo implements his own pre-coded players which are show in Table 1.

One of the players that Collazo uses, the MCTS player, never exceeded the performance of Random or Weighted Random (a very low baseline) without a significant number of rollouts (simulated game turns). As visible in Table 1, the MCTS player only obtains a 60% win-rate against a weighted-random player despite rolling out 100 turns. Collazo attributes this error due to the hand-tuning of the value functions used within his MCTS rollout. Because these value functions were hard-coded based on player heuristics and didn't account for board composition or player state, he was unable to display significant win rates over even a Weighted Random player. We use the MCTS player as plug-in adaptor for our DQN, DDQN, and DN algorithms.

4.2 Models

Three different models were used in the experiments: DQN, DDQN, DN. A DDQN based model was used for MCTS testing.

4.3 Model Design

The provided environment outputs state tensors of dimensions $(21, 11, CHANNELS)$ for each player, where each channel serves as a separate layer that represents a board state plane for each player. For our implementation, our tensor dimensions are $(21, 11, 26)$, an extremely high dimensional space. Due to the size of the state tensor, we elected to use two different model sizes to see if larger models could extract greater information from the state tensor.

The smaller models featured an additional convolutional layer within the image analysis layer of the models. This can be seen in the difference between Figures 2 and 3 for the DQN models and the difference between Figures 4 and 5 for the Dueling Network models.

4.4 Reward Function

Using the guidance from [5], we elected to build a set of reward functions that would allow us to tests some hypothesis for optimal reward. We began by using the control reward function which provided a +1 reward in the case of a win and a -1 reward in the case of a loss. This composes reward function 5.

In addition to the control reward function, four other reward functions were crafted, each with a different win loss reward ratio. At each step, the algorithm would be provided a reward that consisted of the difference between the algorithm's victory points and the largest victory points in the game. In the case that the model is losing, it would experience a negative reward and in the case that the model is winning, it would see a positive reward. Each of the reward functions are summarized in Table 2.

4.5 Simulation Simplifications

Due to the complexity of the Catan board game and computational limits we took liberties in minimizing some aspects of the game. Namely, we removed the inter-player trading mechanic, lowered the victory point limit, and reduced the number of opponents.

4.5.1 Trading

Trading is very complex mechanic with significant down-stream implications. There are a number of trading opportunities throughout the game including trading with the bank, through ports, and with other players.

Trading with the bank is relatively simple, at any point during the player's turn, the player can exchange four of the same resource for one of any other resource. This mechanic allows for players with a surplus of a single resource to be able to obtain other resources and progress in the game.

There are two forms of port-based trading. Both forms require the player to build structures towards a point on the map which enables access to a port. One form of port is a better version of the Bank trade. Instead of trading four of the same resource for one resource, a player only needs to trade three of the same resource. The other port is more strategy specific. It allows the trade of any two of a specific resource for one resource. A player would only build to this port if the player has a surplus of the specific resource that the port takes advantage of.

Finally, the player trade mechanic is the most complex of all. In typical human Catan, the active player requests a resource and all the other players can help make a market for said resource. But in a simulated environment, implementing this mechanic is very expensive, each player proposes a trade and then other players must reject/accept the proposed trade. But this process can be expensive when players make nonsensical proposals leading to a significant increase in training time. Further, the downstream implications are significant. Consider the following example where the player has two ore resources and a hay resource. Immediately trading the ore for a sheep can enable the construction of a development card but waiting for the acquisition of another ore and hay can enable the construction of a city. The benefit if the trades can differ depending on the strategy, the game state, and more. Due to the projected increases in training time and complexities of trading, we elected to simplify the overall game by disabling inter-player trading.

4.5.2 Victory Point Limit

Typically Catan has a 10 point victory point limit. The first player to obtain 10 points is declared the winner of the game. But, in a simulated environment, we found that obtaining 10 victory points would take an inordinate amount of time. Further, the overall game would be subject to higher variance and the reward signal would be diluted. To amend this, we lowered the victory point limit to 6. Doing so decreases the length of the game and improves the strength of reward/loss signals.

4.5.3 Single Opponent

Catan has a max of four players and a minimum of two players. To improve simulation efficiency, we decreased the number of opponents to just 1. This improved simulation efficiency, decreased the variance of the game, and improve algorithmic performance.

4.6 Model Training

Before we can use MCTS, we wanted to first establish a strong baseline, as written in the AlphaGo paper [8]. But, as we don't have any human expert data to work with, we instead decided to use training data from execution of the Alpha Beta Pruning algorithm, the best performing player within the catanatron repository. Therefore, each of our models was trained by playing 10000 games against

the Alpha Beta player in an epsilon greedy training strategy. The first 1000 games were subjected to a linear decrease in epsilon probability with an initial probability of 0.9 to a final probability of 0.003.

4.7 Monte Carlo Tree Search

After training a strong baseline model, we tested our models in an MCTS implementation to test our model's value estimates. Our MCTS algorithm executes n rollouts (n = 25) on each available action. We administer an epsilon greedy exploration strategy, where for some percentage of actions, the model selects randomly from the available actions. For all other scenarios, the model selects the highest value action. The rollout executes for 200 moves or until the game is completed. From the 25 rollouts, the model gains an understanding of the value of each of the actions. One thing to note is that the opponents' actions are also determined by our model's value estimates rather than the opponent's strategy, which can be a negative if our model greatly mispredicts the opponents strategy.

In order to further improve our model, we continued with online training while using MCTS. Since we ran n rollouts, we used the win/loss rates as a reward of the action. We then utilized a replay buffer to store information about the current game state, next game state, action taken, reward, and terminal state details. We selected 200 samples from this buffer as a batch and ran stochastic gradient descent with exponential weight updating for the target model. This training greatly improved the performance when utilizing MCTS. Two considerations are that the reward function differed from training. Additionally the amount of data that was trained on was significantly less than before, as it was limited by the number of online simulations.

5 Experiments

5.1 RL Algorithms

From our results, we found that Dueling Networks was the algorithm that performed the best, with DQN and DDQN performing relatively similar.

5.2 Reward Functions

Due to the lack of literature on crafting reward functions as noted by Knox et al. [5], we wanted to inspect the impact of different reward functions on both training and performance. Therefore, we used the five different reward functions specified in the Methods section and in Table 2.

Inspecting Table 4, we can see that the effectiveness of a reward function greatly depends on the algorithm it is applied to. For small DQN, policies 2, 3, and 5 seem decent but for small DDQN, policies 1, 3, and 5 are decent, and for small DN, policies 2 and 5 seem most effective each boasting 45% win rates. This trend is turned on its head when it comes to the Medium size models where DQN Medium rarely wins at all and DDQN Medium only works with reward functions 4 and 5.

An overarching trend is that policy 5, the simple policy with only a terminal reward is pretty effective despite the additional effort placed in crafting reward functions 1-4.

5.3 Model Size

Through our experiments, we found that model size had a counter intuitive impact on model performance. For the DQN and DDQN models in Table 4, the Medium sized models are unable to generate a meaningful number of wins with most medium models expressing 0% win rates. This may have been due to the medium sized models not having enough data to fully utilize their size.

Curiously, for Dueling Networks, we see Medium model having the best performance across all models and reward functions versus all three players. This may have been due to some form of randomness in training, but also can be attested to the DN algorithm's ability to process and apply reward feedback from the environment.

5.4 MCTS

To test MCST + DQN, we played directly with other players using the Catanatron Simulation. This was setup by having the fully trained model running along with MCTS with $\epsilon = 0$, meaning no

random moves would be made for exploration purposes. This was ran against three types of players of greatly varying strength: Weighted Random, Greedy Player, and Alpha Beta. Our results for MCTS-DQN greatly surpassed any RL approach taken by the previous author who was unable to beat the performance of weighted random. For 25 games, our algorithm won 23 and only lost 2. This likely would be a flawless win if Victory Points was set to 10 however. With the other ones, it was much less successful though. Versus Value Function, our algorithm won 4/25 times, and 0 times versus Alpha Beta. Another indicator of how well an algorithm is performing is the average number of moves it takes to lose. As a strong baseline, AlphaBeta takes around 75 moves on average to win a 1v1 game. Our algorithm takes around 120-130 moves on average to win. Weighted Random takes 200+ moves on average for a 10 Victory Point Game. Overall the results were surprisingly strong considering the large branching factor and our limited compute power. Our success over the author's previous approach likely lies in the architecture that we used. We believe that our CNN based models outperformed the Fully Connected Neural Network models because of image-based features that are easily learnable via CNN. Further, initial training against the Alpha Beta player jump started the models performance (as opposed to a randomly initialized model).

5.5 Limitations

AlphaZero has proven to be an extremely successful strategy when dealing with full information board games such as Go and Chess which are both complete information and deterministic games. However, Catan is an incomplete information and non-deterministic game. These two factors introduce notions of unknown and variance to the game, increasing the noisiness of reward functions. Due to the high variance, even when the model selects the optimal action throughout the game, the model may not win the game.

The board and environment that we based our strategies on were also partially flawed for our usecase. A CNN shines when trying to form patterns based on proximity of elements in things like images or boards. However, our board tensor contained extra information such as Development Cards, Resources and Victory Points of each player. This creates more noise for the CNN and makes it harder to learn the underlying patterns for winning strategies. In addition, in a real game of Catan, there is much more information than what is encoded. For example, the opponents resources are encoded as a single number, with no distinction between each type of resource that they may have. However, a player in a normal game could easily keep track of what rolls have come and track exactly which resources the opponent has based on their settlements. Having this information is key to decision making and our models didn't have access to this through our environment.

Another interesting consideration is the fact that Deep Q-Learning is an Off-Policy algorithm. This means we can be very data efficient and store data in our replay buffer even if it was collected from a different policy. This is due to Deep Q-Learning being based off the Bellman Optimal Equations, so regardless of what occurs after, at a single time step the equation remains Equation 1. However, this is true if the reward was based on some deterministic reward function. In our case, in MCTS, our reward is determined by winrate from the current policy playing as both players. This means that for a specific policy or Q Function, our reward for the same state and action can be completely different. In other words, our training should actually be on-policy and we should be flushing our data each time, as the old rewards don't correlate to our new model. One way to fix this would be to have some sort of deterministic policy that our DQN plays against in the rollouts, but this doesn't make the most sense and would only teach our model to learn against this specific policy.

As previously mentioned, we had a lack of compute power. For a conservative estimate, we ran MCTS on around 400 games (approximately 400 hours of training). The main computation cost in this algorithm were the game rollouts, which don't contribute to the training of the model. Both increasing compute and implementing multithreading would greatly improve performance and also help with the amount of training done with the model.

5.6 Tuning

The MCTS algorithm had vast array of hyperparameters available, each influencing the model's performance and accuracy in distinct ways. In the training process, we controlled the γ for discounted reward, τ for updating the target network, and batch size. For MCTS, we needed to balance the number of rollouts with time taken, the value of ϵ in our epsilon-greedy agent, max depth of a rollout,

and scale of reward from rollout victories. Each of these required significant experimentation to find good values for, and when a single game takes one hour to simulate, it was difficult to tune these optimally. If possible, a grid search would have proven useful to find optimal hyperparameters but we would need much more compute to run this.

6 Conclusion

From our study, we have the following four key takeaways:

1. Algorithm matters most, a more powerful algorithm improves performance
2. Reward functions are difficult, simple reward functions are better
3. Larger models are not necessarily better
4. MCTS is a powerful tool, despite the computational cost

The change in algorithm produced the largest, most consistent difference in performance. Dueling Networks was able to consistently achieve a 30% win rate across model architectures and policies indicating that a powerful algorithm is required can take advantage of nuanced feedback and larger, more comprehensive models.

In terms of reward functions, notably, Policy 5, which utilizes a simple terminal reward, demonstrated surprising effectiveness across various algorithms and model sizes. This finding suggests that simpler, more straightforward reward functions can be as effective, if not more so, than more complex, granular strategies.

The difference in performance across model sizes also highlights the role of model capacity in learning dynamics. Smaller models generally performed better with certain policies where medium models did not, suggesting that the interplay between model architecture and reward processing is crucial and warrants further investigation.

The high posted win rates for MCTS indicate that despite the computational cost, MCTS is still a powerful tool in settings where the state can be reasonably simulated. Therefore, seeking more efficient modes of executing MCTS can greatly improve existing methods of model training and even model performance.

6.1 Future Work

Although an exhaustive list is presented in the appendix, we want to discuss key areas of future work that could elevate and build on the existing work, low hanging fruit if you will.

Self-play is a powerful strategy that was leveraged extensively in the AlphaGo paper. It has the combined benefit of doubling training efficiency as well as providing a gradual increase in opponent difficulty. This gradual increase in difficulty enables models to leverage their prior experience against opponents of increasing difficulty, lowering training time.

We only used Value-Based algorithms from the DQN class of algorithms. But further research could including Policy Gradient algorithms such as REINFORCE and REINFORCE-wb as well as Actor-Critic methods such as DDPG and PPO.

Finally, we see multi-headed agents as a key optimizer of model performance. Within our implementation, we needed to mask model probabilities for invalid actions, removing much of the selection power of the model. But by using a mult-headed agent where each model is responsible for selecting among a specific class of actions, it would be more easy to directly communicate rewards and inspect model choices.

References

[1] M. Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey, 2022.

[2] B. Collazo. bcollazo/catanatron. `https://github.com/bcollazo/catanatron`, 2024. Original work published 2020.

[3] H. Cuayáhuitl, S. Keizer, and O. Lemon. Strategic dialogue management via deep reinforcement learning. *CoRR*, 2015.

[4] Q. Gendre and T. Kaneko. Playing catan with cross-dimensional neural network. `http://arxiv.org/abs/2008.07079`, 2020.

[5] W. Bradley Knox, Alessandro Allievi, Holger Banzhaf, Felix Schmitt, and Peter Stone. Reward (mis)design for autonomous driving, 2022.

[6] P. McAughan, A. Krishnakumar, J. Hahn, and S. Kulkarni. Qsettlers: Deep reinforcement learning for settlers of catan. `https://akrishna77.github.io/QSettlers/`.

[7] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] D. Silver et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017.

[9] D. Silver et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[10] I. Szita, G. Chaslot, and P. Spronck. Monte-carlo tree search in settlers of catan. In H.J. van den Herik and P. Spronck, editors, *Advances in Computer Games*, pages 21–32, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[11] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[12] Z. Wang et al. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[13] C. J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[14] K. Xenou, G. Chalkiadakis, and S. Afantenos. Deep reinforcement learning in strategic board game environments. In M. Slavkovik, editor, *Multi-Agent Systems*, pages 233–248. Springer International Publishing, Cham, 2019.

7 Appendix

Future works list

- Enable self-play
- Enable gradual opponent difficulty increase such as training against Random for 1000 iterations, Weighted Random for another 1000, etc.
- Explore other policies such as Policy Gradient and Actor Critic
- Implement UCT rather than using epsilon-greedy
- Use a simplified trading mechanic that computes valid actions after all trades
- Create multi-headed agents

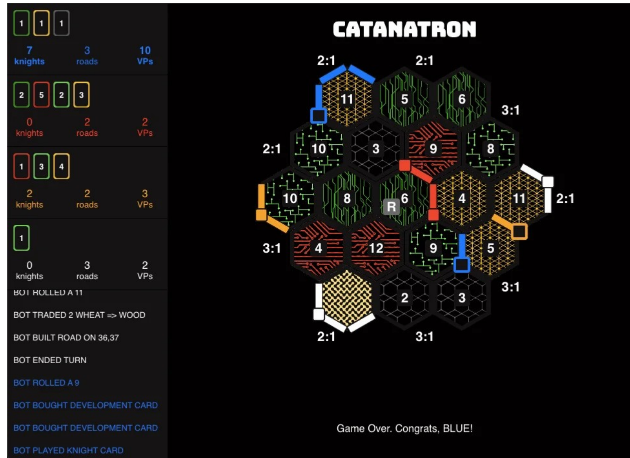

Figure 1: Catanatron Board Example.

Table 1: Player Performance in 1v1 Games

Player	% of Wins	Number of Games
AlphaBeta(n=2)	80% vs ValueFunction	25
ValueFunction	90% vs GreedyPlayouts(n=25)	25
GreedyPlayouts(n=25)	100% vs MCTS(n=100)	25
MCTS(n=100)	60% vs WeightedRandom	15
WeightedRandom	53% vs WeightedRandom	1000
VictoryPoint	60% vs Random	1000
Random	-	-

Table 2: Reward Functions Overview

Reward Function	Win Reward	Loss Reward
Reward Function 1	+100	-100
Reward Function 2	+1000	-100
Reward Function 3	+10	-10
Reward Function 4	+100	-10
Reward Function 5	+1	-1

Action Type	Output Size
Action Type 1	19
Action Type 3	72
Action Type 4	54
Action Type 5	54
Action Type 8	20
Action Type 10	5
Action Type 11	60

Table 3: Action Types and Their Output Sizes

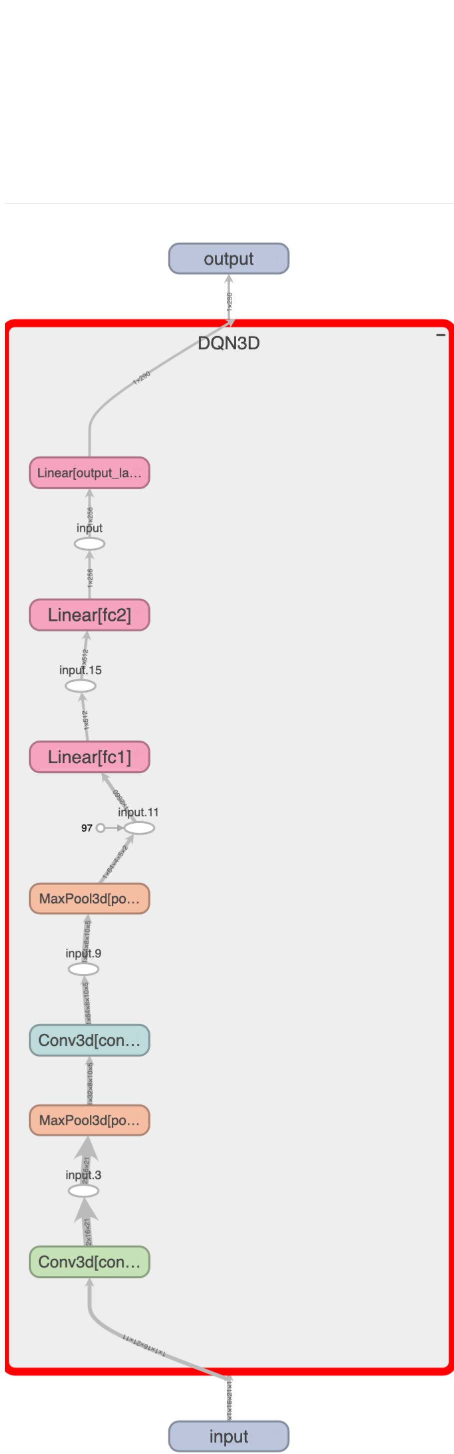

Figure 2: Architecture of Small DQN Network

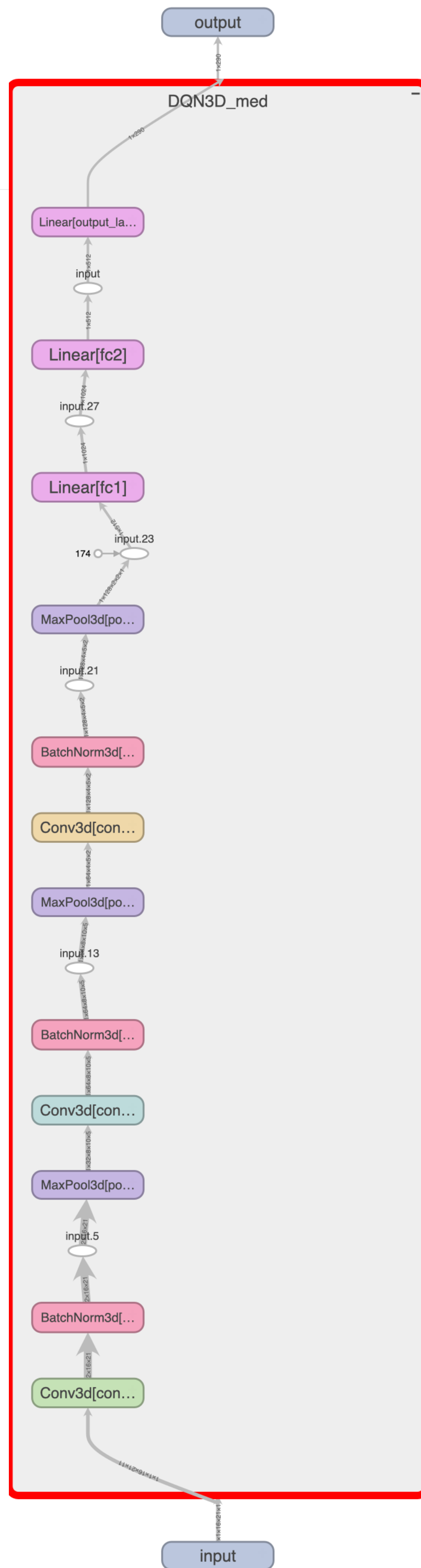

Figure 3: Architecture of Medium DQN Network

Figure 4: Architecture of Small Dueling Network

Figure 5: Architecture of Medium Dueling Network

Table 4: % Win Rates by Model and Size

Model	Model Size	Policy	Player		
			Weighted Rand	Search	AlphaBeta
DQN	Small	1	9	8	0
DQN	Small	2	36	32	1
DQN	Small	3	25	22	0
DQN	Small	4	18	16	0
DQN	Small	5	41	32	1
DQN	Medium	1	0	3	0
DQN	Medium	2	0	0	0
DQN	Medium	3	0	0	0
DQN	Medium	4	0	0	0
DQN	Medium	5	0	0	0
DDQN	Small	1	14	15	0
DDQN	Small	2	4	7	0
DDQN	Small	3	27	16	1
DDQN	Small	4	11	5	0
DDQN	Small	5	40	40	0
DDQN	Medium	1	0	0	0
DDQN	Medium	2	0	0	0
DDQN	Medium	3	0	0	0
DDQN	Medium	4	19	10	1
DDQN	Medium	5	19	18	1
DN	Small	1	29	36	1
DN	Small	2	46	49	0
DN	Small	3	32	27	0
DN	Small	4	14	7	0
DN	Small	5	43	39	0
DN	Medium	1	33	31	3
DN	Medium	2	31	30	1
DN	Medium	3	38	39	1
DN	Medium	4	47	32	0
DN	Medium	5	30	32	0

Table 5: Policy Visualizations Across Models

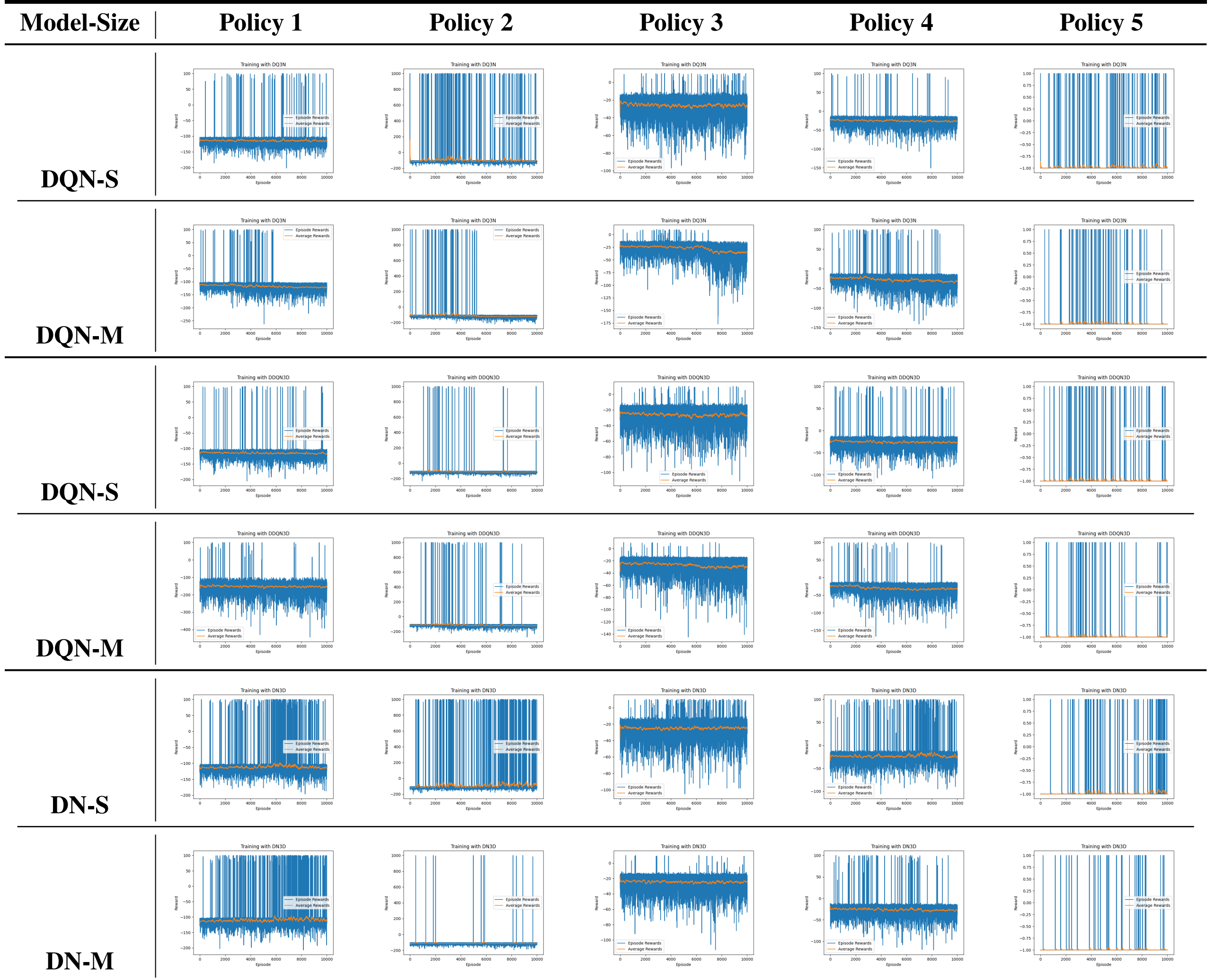

Table 6: Performance comparison of different strategies with DQN + MCTS. (out of 25 games)

	Weighted Random	**Search**	**ValueFunction**	**Alpha Beta**
MCTS + DQN	23	24	4	0