From Resources to Victory: Heuristic-Driven Reinforcement Learning in Catan

Shayan Daijavad, Samuel Kaplan, Jacob Kelleran, Aiden Smith and Tymon Vu
Cal Poly,
San Luis Obispo, United States
{sdaijava, sfkaplan, jckeller, asmit332, tvu38}@calpoly.edu

***Abstract*—For our project, we worked on a Catan simulation that allows for different types of agents to compete against each other. We extended work done by one of our team members (Shayan) in CSC 480. We completely redid the codebase, added new game mechanics like Longest Road, and added the ability to choose different types of agents to play the game. We added a way to vectorize the game state for deep learning purposes. We redid the original codebase's heuristic agent and added a random action agent and a reinforcement learning agent. We also added the ability for humans to play the game.**

I. INTRODUCTION

The problem we are trying to solve is coming up with an agent that is capable of effectively playing Settlers of Catan.

Catan is a complex game with many rules, but it lends itself nicely to programmatic representations, as its board can be represented using a graph data structure, and game and player state can be nicely represented using object oriented design.

There are many effective strategies in Catan, and it is ripe for testing different kinds of artificial intelligence techniques. Moreover, thanks to Catan's unique multiplayer setting, we can gain insight into the performance of these techniques by comparing them against each other and against human players. Developing dynamic systems to adapt to changes in environment based on multiple agents actions maintains a high value as more people adopt and use AI models in day to day life.

This project relates to the class objectives because it pushed us to look into prior work done in the field (there is a lot), and come up with a novel way to approach this problem and refine our artificial intelligence knowledge and capabilities. We learned a lot about encoding game state, and using those same encodings for deep learning.

Our contribution is providing a robust codebase with a fully featured representation of Catan. Our codebase allows developers to easily add new types of agents and compare them to previously created ones, including a random action agent, a heuristic agent, and a reinforcement learning agent. We also provide a way to easily encode Catan's game state into a vector format for agents that use deep learning techniques.

II. GAME SPECIFICATION

Catan, also known as The Settlers of Catan, is a strategy-based resource-management board game that was designed by Klaus Teuber in 1995. The game is designed to be played by 2-4 players, where the game board is composed of hexagonal tiles, each with a number from 2-12, that represent different terrains, with each producing a different resource. There are 5 main resources/terrains in the game: wood, brick, sheep, wheat, and ore. These resources are used to help players build settlements, cities, and roads to further expand to more terrain tiles and produce more resources, which is fundamental to the game's economy. The board representation we used is shown in Figure 1.

Each player starts the game with two settlements and two roads and strategically chooses where to place each settlement along with a connected road at the intersections of terrain tiles. Once all players have placed their starting settlements and roads, the game begins with the first player rolling two six-sided dice. Any settlement adjacent to a tile with the matching rolled number grants its owner the resource produced by that terrain type. If a player rolls a 7 (which can never be on a tile), they get the chance to move the "robber", an extra game piece, to any tile on the board and steal from any player settling on that tile. Any tile that has the "robber" on it can not produce any resources from it. Play proceeds with the player that rolled building based on their resources or requesting to trade with the bank, harbors they settled on, or other players for any resources they might need. The first player that gets to 10 victory points will be crowned as the winner of the game, where settlements (initial houses) are worth 1 point and cities (upgraded houses) are worth 2 points.

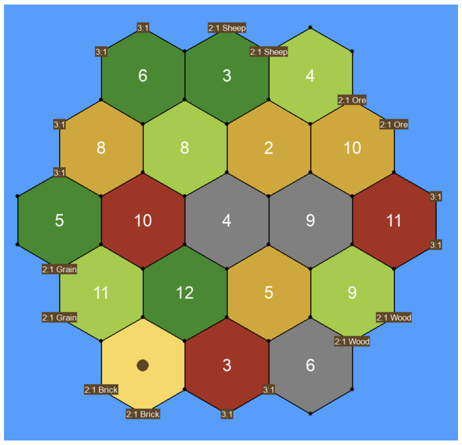


Fig. 1. Sample Catan Board (Our implementation)

Outside of the main expansion and building aspect of the game, there are other outside factors where players can win extra victory points, which include development cards and win conditions (longest road and largest army). Longest road and largest army are both conditions that are met and rewards a player an additional 2 victory points if they have the longest connected road network on the board (starting at 5) or if they have the most "knight" development cards played (starting at 3). These win conditions can be overtaken and stolen by any player. Development cards, on the other hand, act similar to "Chance cards" in Monopoly with only positive benefits, where players draw from a face down deck randomly and get an extra action they can use in any future turn to help them out. This card is hidden to all players, except the one that drew, and revealed when they want to play the action. There are five unique development cards in vanilla Catan, which include victory points, road building, knights, monopoly, and year of plenty.

III. RELATED WORK

There have been numerous forays into developing artificially intelligent Catan bots over the years, with varying levels of success. In Michael Pfeiffer's 2004 paper [1], he explored using a reinforcement learning approach to this topic, training a game with a hierarchical learning structure. This worked by having an overarching policy network provide rewards for winning the game, and alternative networks for lower level actions that made up the vast majority of the gameplay. Although this approach served to create a model that made reasonable moves, the results ended up not being able to best human players, obtaining 3-7 reward points based on different learning models. The heuristic model they developed was actually the most successful, which showed that there was not a lot of progress with the reinforcement learning approach.

Another paper that explored a reinforcement learning method in Catan was done by Brahim D. and Tristan C. in "Deep Catan" [2]. Here, they used a combination of Monte Carlo Tree Search (MCTS) and reinforcement learning to gain better results. This process is reminiscent of the AlphaGo paper, in which they had their bot able to explore multiple branching pathways, while still having the benefit of thousands of iterations of games and turns. The specific MCTS algorithm they used was Upper Confidence bounds applied to Trees (UCT). They labelled this as the UCTnet model - which signifies a combination of network training and MCTS, and found that it outperformed the pure UCT model. However, they did not publish any results against human opponents, and completely sidestepped any trading as part of the game.

An inspiration for our game playing environment was a Catan Simulator called "Catanatron" [3]. This simulation had a variety of different agents - plus the ability for humans to play and interact with the environment. We took inspiration from this in our game simulation, including multiple different agent types as well as the capabilities for humans to play alongside the bots. We initially discussed the idea of using this environment instead, but found that it was ill-tailored to our specific needs and opted to create our own.

Finally, for the development of our heuristic we looked into past research of optimal moves and strategies. One paper we found that went quite in depth with this was "Settlers of Catan Analysis" by Peter Keep [4], which discussed optimal resource combinations, settlement placements, and win conditions like longest road and largest army. We tried to take these metrics and strategies into account for our heuristic agent.

IV. IMPLEMENTATION

One of our team members (Shayan) implemented a Catan simulation in his CSC 480 group project, and had heuristic-based agents play the game. We extended his implementation for 570,adding additional game features, reworking the code-base, adding human players, encoding the game state into a vector format, and finally attempting a reinforcement learning agent.

A. Additional Game Features

In the original implementation, the simulation lacked the longest road and largest army special cards, which get awarded to players who have the longest continuous sequence of roads and the largest number of knight cards, respectively. Largest army was relatively simple to implement as we already kept track of the number of cards a player had, but longest road required a new algorithmic solution that runs every turn. To compute the longest road for a given player, we run a DFS from every road vertex the player owns, and find the maximum length path the DFS returns. We then compare the length of the longest road of each player to determine who to give the award to.

We also added visuals that allow us to keep track of player resource counts, victory points, dev cards, and longest road and largest army. Moreover, we made the board visuals scale to screen size, so that the visuals remain consistent across devices.

B. The Remuwork

To make it more adaptable to different agents all interacting within the same board, the codebase was largely reworked to directly integrate abstract agents within the gameplay loop. With this addition, for every player's turn, it repeatedly asks the agent associated with that player what action it would like the player to take. This happens until the agent runs out of options or directly ends its turn. Through providing an interface for methods that consume information like every possible action that could be taken and output the agent's desired course of action, this laid the groundwork for adding new agents that could interact with and be swapped with each other. The rework also exposed certain structures and information that helped with serialization and encoding, and segmented different parts of the application into consolidated packages. Finally, with 100% type-hinting coverage, the code could be verified for correctness more easily and type-related errors became less common. This also increased development speed as intellisense systems could pick up on the structure of the project and provide more informed suggestions.

C. *Human Players*

To provide another way to evaluate the effectiveness of our agents, we added the ability for humans to also play the game. This was tricky, as it involved a lot of Pygame manipulation to handle player inputs and effectively identify the intended action. For instance, we created functions for roads and road vertices (the points between roads where settlements can be placed) that computed the distance between a player's cursor to the road/road vertex visual representation.

Human players work as follows: they have a specific set of actions they can take during the setup phase of the game (placing their initial settlement and then an initial road). After the setup phase, they can do the general game playing actions like placing settlements, roads, and cities, buying dev cards, and using dev cards, although the Monopoly and Knight dev cards do not work as of now. Human players can also trade with the harbor, but not with other agents, as the simulation as a whole does not support trading with other players.

D. *Encoding Game State*

To record the current board state of the game, we had to translate the hexagonal board in a standard game to fit into a data type that can be easily read by our training model. This was done following a brick representation layout, which was established in Deep Catan [2]. Just as each hexagonal cell makes contact with six others (assuming it is not on an edge), each brick in the brick representation makes equal length contact with six other bricks surrounding it. It is effectively a vertically squashed representation, and its image-like structure makes it perfect for passing into a convolutional network. This is shown in Figure 2.

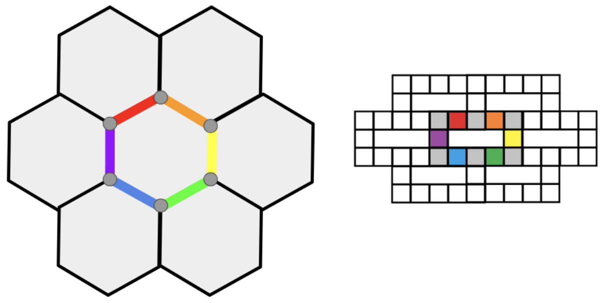


Fig. 2. Brick Representation

With this standard BrickRepresentation recorded, our complete board state that is passed into the reinforcement learning agent would have n+1 channels, where n is equal to the number of players, and each channel contains a main copy of the BrickRepresentation model. These channels are shwon on the left side of Figure 3. The reason why we need a separate channel for each player is to record that individual player's current settlement, road, and city spots on the board, where 0 represents nothing is owned from that player and any positive number correlates to the player owning that spot. Stacking all of this data together would easily allow the total board state with n+1 channels, where 1 channel is dedicated to the starting board information, to be passed into a convolutional neural network that can process all the brick-based layers.

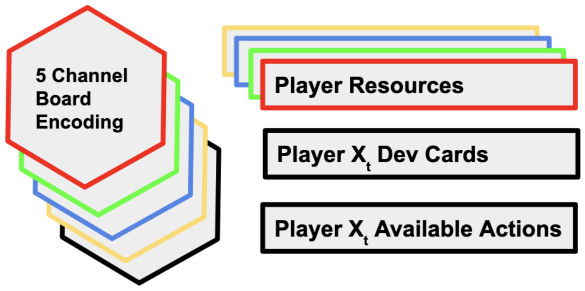


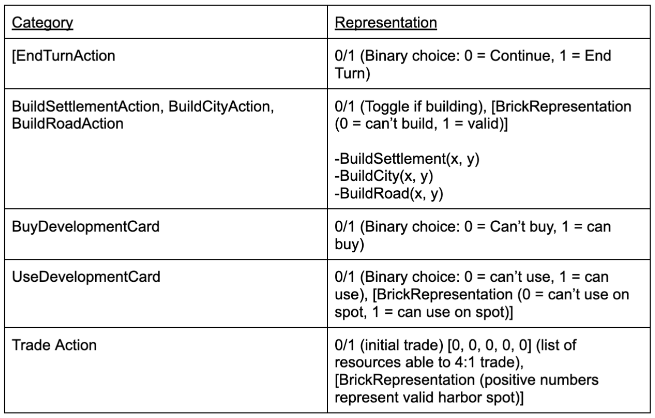
Fig. 3. Board and Player State Encoding

When it comes to figuring out the best action to take and how our model should be able to return that information, we have to pass in a unique static action encoding scheme every turn for all the available different action categories that the agent can perform. In a standard normal game of Catan, every player is allowed to perform any number of actions (given the conditions are met) from the listed 7: end turn, build a settlement, build a city, build a road, buy a development card, use a development card, trade for resources. To pass this in our model, we can use a simple array of size 7 with simple 1s and 0s to represent which actions are valid at the start of every turn for the reinforcement learning agent.

However, for our given model and in any standard game of Catan, this action encoding alone is not sufficient to fully define the action space for a player's turn or for our reinforcement learning (RL) agent to learn effectively. When considering building actions, such as constructing a road, simply marking the action as available with a 1 does not convey the full extent of choices available to the player. Instead, we must also encode the specific locations on the board where these actions can be performed. This is where all of these special actions have an additional copy of the BrickRepresentation (which is our method of encoding the board state) filled with 1s and 0s that show all the valid locations for these actions to be taken.

All building options have their own copy of BrickRepresentation based on the aforementioned reason above. Trading also requires their own board channel that records all the special harbors that the player can trade on (i.e. 2:1 brick, 3:1) as well as a separate array for the available resources that could trade if they have the sufficient amount.

Development card usage also requires additional encoding beyond a simple 1 or 0 in our action array. Cards such as ”Knight” (which moves the robber) and ”Road Building” (which allows placing two roads) requires further spatial information rather than categorical inputs. For example, if a player wishes to use the ”Knight” card, the model must receive a representation of all valid hexes where the robber can be placed. This would be done as before by using the same BrickRepresentation and setting all valid hexes to be 1s for all tiles that do not currently have a robber on it.



Category	Representation
[EndTurnAction	0/1 (Binary choice: 0 = Continue, 1 = End Turn)
BuildSettlementAction, BuildCityAction, BuildRoadAction	0/1 (Toggle if building), [BrickRepresentation (0 = can't build, 1 = valid)] -BuildSettlement(x, y) -BuildCity(x, y) -BuildRoad(x, y)
BuyDevelopmentCard	0/1 (Binary choice: 0 = Can't buy, 1 = can buy)
UseDevelopmentCard	0/1 (Binary choice: 0 = can't use, 1 = can use), [BrickRepresentation (0 = can't use on spot, 1 = can use on spot)]
Trade Action	0/1 (initial trade) [0, 0, 0, 0, 0] (list of resources able to 4:1 trade), [BrickRepresentation (positive numbers represent valid harbor spot)]

Fig. 4. Action Encoding

In addition to this action space channel, we also pass an additional n+1 channels for other static game information on the board. N channels are dedicated to the current remaining resources count of all the players in the game, including the bot. These include the individual resources that the player has (wood, brick, sheep, wheat ore); the remaining amount of roads, settlements, and cities they can build; the amount of victory points they currently have; if they have largest road and army; the number of knights played; and the current length of their longest road. The last channel is solely dedicated to the bot's currently held development cards (if any). Since we are passing in a binary flag of 0/1 in UseDevelopmentCardAction, we need to keep track of which development cards the bot has and adjust for if they decide to play a development card in that process. This is because a development card like "Year of Plenty" would not need reference to its valid BrickRepresentation rather than "Road Building" which needs reference to this. That means for the entire player states encoding, we pass a total of n+2 channels. These channels are visible on the right side of Figure 3.

E. Reinforcement Learning Agent

Using these encodings our model processes the data in a two part Deep Q-Neural Network as shown in Figure 5. The first part, a convolutional neural network, performs convolutions on the various brick based board encoding channels. A convolutional model was selected for its ability to preserve special hierarchies when processing data. The convolutional neural network processes the board channel encoding in 3 convolutional layers outputting the data into a fully connected layer. These layers are shown in Figure 6 This fully connected layer is integrated with the other core part of our model, a multilayer perceptron which processes the vector encoding of the RL agent's action space and resource information. This MLP is connected to the fully connected layer of the Convolutional Neural Networks forming one final fully connected deep neural network. With this the model is able to combine the processed and weighted special hierarchies of the board state with the vectorized encodings of the action spaces and resource values. This final neural network outputs to seven neurons corresponding to the seven types of potential actions the model could take, being the ones mentioned in Figure 4.

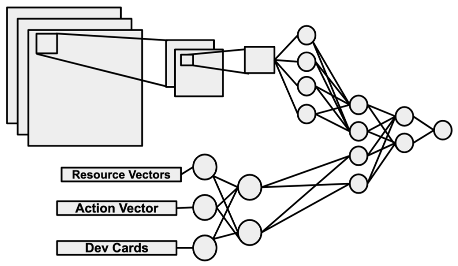


Fig. 5. Neural Network Structure (not to scale)

The structure of the network is dependent on the number of players, the size of the player state, and the size of the action space. This analysis will assume a player count of 4 (used for all our tests), a player state size of 1224, and an action space size of 7.

With 4 players, there are 5 board channels (one for each player plus the neutral information of the board). Each of these channels has a height of 11 and a width of 21 (see the brick representation described in Figure 2), for a total of 11 * 21 = 231 values per channel. These five channels get passed into a convolutional neural network, each layer having a kernel size of three, a stride of one, and a padding of one. The layout is shown in Figure 6:

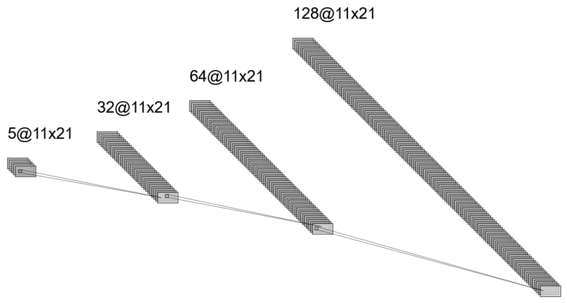


Fig. 6. CNN Layers

The player state-related values follow a different pipeline, as they do not have the same spatial relations as the board does. This consists of two fully-connected layers with the structure shown in Figure 7.

After both of these aspects have been processed, they are concatenated together and passed through one final fully-connected layer and argmax is used to determine the bot's final action, as shown in Figure 8.

Through all these layers, the model is able to separate the spatial components of the board and the independent components of the player state, combining them together for a final pass at the end to evaluate the agent's action.

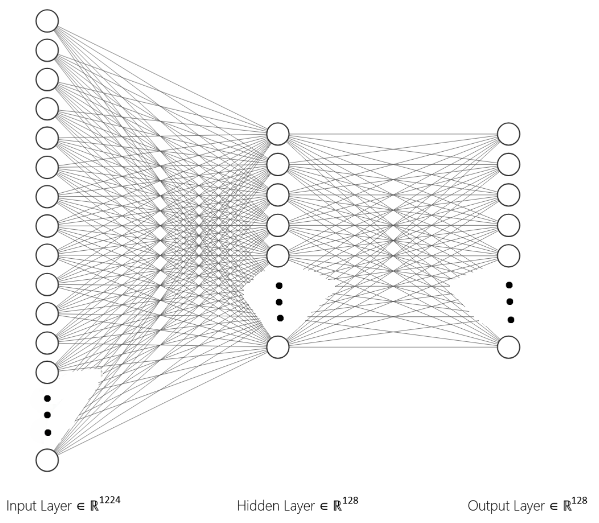


Fig. 7. Fully Connected NN Layers

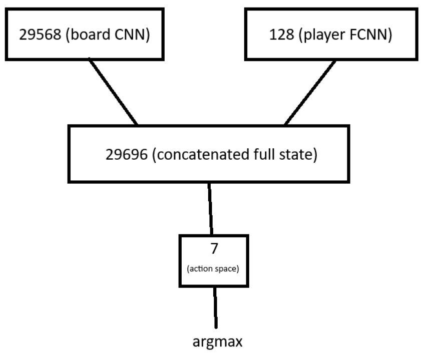


Fig. 8. NN Concatenation

On any given turn when the RL agent is called to take its turn, if first records a snapshot of the board states full encoding. Then the model evaluates if it will be using heuristic or its Deep Q network to decide its action based on its epsilon weight. If the heuristic is not selected then the encoding is then passed through the Deep Q Neural Network. This then outputs logits or q values, corresponding to the 7 possible actions. The max q value is then selected and the action said q value corresponds to is then performed. The resulting board state is then recorded and rewards are calculated based on the resulting outcome of the action. These rewards are used to update the weights of the Deep Q Network in back propagation.

We tuned the reward function multiple times but saw the most success rewarding both victory points as well as resources. Additionally we found that hyperparameter tuning the reward values of specific resources produced better results. In specific we found that increasing the reward for Ore and Grain, two resources critical for creating towns, increased our models' win rate.

V. Evaluation / Results

For our evaluation, we performed multiple tests across 100 sample games. Each test used different distributions of agents, those being the Random, Heuristic, and RL agents. Simulations were done with 100 games unless otherwise noted.

For the first test, we found a benchmark by comparing a full room of random bots against each other as a baseline.

Player Type	Wins
Random 1	27
Random 2	22
Random 3	23
Random 4	28

TABLE I
4 Random Wins

As seen in Table I, the bots performed roughly the same, and the average turn number in this experiment was 259.6 turns, showing that the games lasted much longer than the previous number of 71 that was found from a group of average players playing Catan in our proposal. This makes sense, as the bots make completely nonsensical moves and do not take into account anything in the position.

For our second test, we ran the heuristic bot against the random bots to validate its capabilities. Through testing, we found that it won the vast majority of games. When adding two heuristic bots they both won much more than the random bots.

Player Type	Wins
Random 1	1
Random 2	0
Random 3	0
Heuristic	99

TABLE II
3 Random and 1 Heuristic Wins

Player Type	Wins
Heuristic 1	74
Random 1	2
Random 2	1
Heuristic 2	23

TABLE III
2 Random and 2 Heuristic Wins

As seen above in Table II and Table III, the heuristic agents consistently outperformed their opposition. An interesting fact to note is that in the second table, we see that the first heuristic bot won 3x more than the second. We believe this is due to the first player starting advantage allowing them to get an early lead that usually converted into a win.

The first simulation had an average turn number of 90.17, and the second one had an average turn number of 113.73. Although it may seem counter-intuitive for the game with two heuristic bots to be longer, the competition between the two likely dragged the game out as they moved to oppose each other. When there was only one intelligent bot it is able to obtain largest army and longest road uncontested, resulting in faster victories.

Next, we tested the RL agent against the random bots to provide a benchmark over the heuristic bot in terms of beating opponents with random actions. Our main model we used for testing had 75% usage of QNN, with the remaining 25% being based on a heuristic. We found that it also performed quite well against the random bot, showing that the training did not have random results.

Player Type	Wins
Random 1	1
Random 2	0
Random 3	0
RL Agent	99

TABLE IV
3 Random and 1 RL Agent Wins

Player Type	Wins
RL Agent 1	71
Random 1	1
Random 2	2
RL Agent 2	26

TABLE V
2 Random and 2 RL Agent Wins

As seen above in Table IV, the RL agent performed similarly to the heuristic bot, winning 99 out of the 100 games. This simulation's average number of turns as 100.41, which was also much lower than the 4 random simulation, showing that the RL agent quickly found a strategy to victory.

In Table V we also saw an emulation of the 2 Random/2 Heuristic experiment, where the first player won the majority, with the second RL agent also performing admirably while the randoms barely won any games. The average turn number was 120.35, showing that competition also increased the turn number.

Finally, we tested an RL agent against 3 opposing heuristic bots. This was our main test to check its performance, as it was against players that made reasonable choices.

As seen above, in the first experiment of 100 games in Table VI the RL Agent performed admirably winning the majority of the games when in the first position. The average turn number was 88.38, showing that games progressed quickly despite the constant back and forth of win conditions like largest army and longest road.

Player Type	Wins
RL Agent 1	73
Heuristic 1	5
Heuristic 2	17
Heuristic 3	5

TABLE VI
3 Heuristic and 1 RL Agent Wins

Player Type	Wins
RL Agent 1	124
Heuristic 1	787
Heuristic 2	73
Heuristic 3	16

TABLE VII
3 Heuristic and 1 RL Agent Wins

However, in the second experiment of 1000 games in Table VII we got significantly different results. In this one the RL agent still won over 100 games, but the first heuristic bot swept the competition with 787 wins, with an average turn number of 124.50. Further testing must be done to find out details of why this occurs, but it is safe to say that both the heuristic and RL agents performed much better than our baseline of the random bot, and both obtained significant wins in simulations against the other, showing their competitiveness.

VI. Conclusions

Our efforts to create a bot that can play Settlers of Catan at a level competitive with human players was mostly successful. In the end, we managed to extend the original heuristic agent and implement a reinforcement learning model, both of which showed significant improvement over random choice. On top of this, the structural requirements of having a functional Catan simulator that is flexible enough to support human players and multiple agent types was realized. This includes a game state that automatically tracks and verifies the possible actions each player can take, eliminating the class of illegal moves from agents. Through all of these combined elements, we were able to gather reasonable data that suggests considerable strength in both models we developed.

Unfortunately, there were also a few features that we were not able to implement. For example, trading between players is not supported in any of the agents, as the social aspect of proposing and confirming a trade proved too difficult to feasibly implement into any of our structures. At a higher level, this social aspect becomes more prominent, and as a result our bot would likely struggle in such a setting. But in more amateur scenarios, where trading between players can be considered as more of an afterthought than a strategic force, our approach would likely yield reasonable results.

As far as the reinforcement learning approach goes, we were not able to beat the heuristic agent with the amount of training we were doing and/or the model structure. Although it does consistently outperform the random agent, showing that it has learned a significant amount of material, its win rate drops significantly when pitted up against a stronger opponent. More training and hyper parameter tuning would likely improve these results, but we were unfortunately not able to explore these areas as thoroughly as we would have liked. Future extensions of this project would likely prioritize those areas of improvement.

REFERENCES

[1] M. Pfeiffer, "Reinforcement learning of strategies for settlers of catan," in *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[2] B. Driss and T. Cazenave, "Deep catan," in *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*. Springer, 2022, pp. 503–513.

[3] B. Collazo, "Catanatron: A settlers of catan bot," 2025, accessed: 2025-01-27. [Online]. Available: https://github.com/bcollazo/catanatron

[4] P. Keep, "Settlers of catan analysis," 2011, accessed: 2025-01-27. [Online]. Available: https://developingcatan.wordpress.com/wp-content/uploads/2011/02/settlers-of-catan-analysis.pdf