CATANBOT: ANNOYING YOUR FRIENDS WITH DEEP NEURAL NETWORKS AND TREE SEARCH

Samuel Triest, Ivy Nuo Chen
Robotics Institute, Mechanical Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{striest, nuoc}@andrew.cmu.edu

ABSTRACT

Reinforcement learning and tree search have proven to be effective techniques on a wide range of problems in robotics and games. We use reinforcement learning and Monte-Carlo tree search to deign an agent that performs the initial placements phase of the board game Catan. We find that our method outperforms other techniques, and compares favorably to human preferences.

Our code is publicly available at https://github.com/striest/catan_bot.

1 INTRODUCTION

Settlers of Catan is a four-player strategy game in which the four players race to develop a civilization on a hexagonal island. Players develop the island by using the island's resources to build settlements and cities, which in turn allow players to collect additional resources. A player wins the game when they build enough settlements and cities (as well as potentially achieve other side objectives) to cross a 10-point victory threshold. A full explanation of the rules can be found here. An important aspect of Catan is that there are two distinct gameplay phases. The first phase (henceforth referred to as the initial placements phase) requires each player to pick two locations to on the island to build their initial settlements. In the second phase of game play, players go around rolling die and developing the hex from their initial positions to be the first to get 10 victory points.

In general, initial placements is regarded as a major component of the overall strategy of Catan, as the initial placements dictate many factors of the main game, such as where each player is able to expand their settlements to, what resources they are able to gather, etc. Players often take several minutes to deliberate where they should place their initial settlements (as compared to a number of seconds for many Catan turns), and much discussion of Catan strategy centers around this phase.

In this paper, we outline a method for using recent advances in reinforcement learning to design an agent that is able to play the initial placements phase of Catan. Our method relies on using a combination of tree search and reinforcement learning in a manner similar to AlphaGo (1).

Our work makes the following contributions:

1. We design an agent that plays the initial placement phase of Catan.
2. We combine tree search with deep neural networks to outperform either method alone.
3. We use a graph neural network to perform value estimation directly from board state.

2 RELATED WORK

TD-Gammon (2) is one of the first applications of reinforcement learning to a strategic board game. In this work, the authors use neural networks to perform value estimation on states in Backgammon to achieve near expert-level play. Important to the success of this work are the use of deep neural networks and self-play; concepts that remain essential to the success of many current game-playing AI systems.

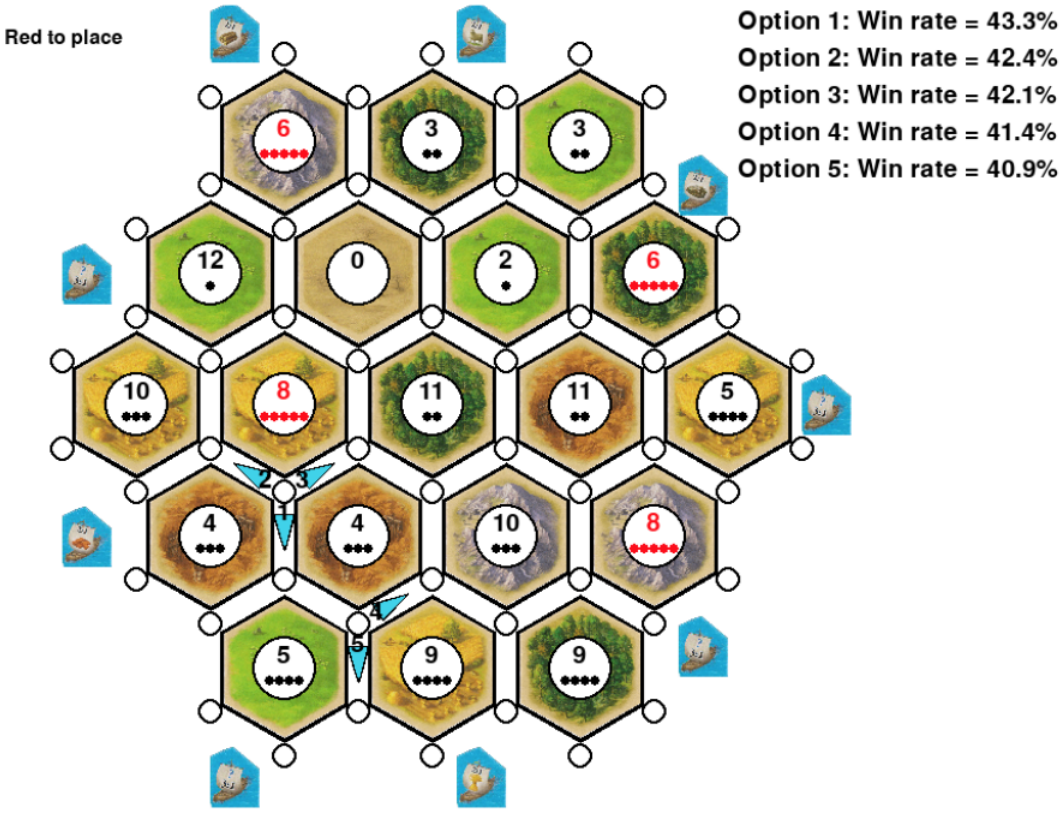

Figure 1: Our system uses a combination of Monte-Carlo tree search and deep reinforcement learning to choose good initial settlement locations for the game Catan. Shown here are initial placement suggestions for a particular board.

Deep Blue (3) is a very famous instance of tree-search being applied to board games (in this case, chess). Deep Blue relies on standard minimax tree-search techniques, but is able to play at a master-level because of the amount of board states it is able to evaluate (including the use of custom hardware to increase search speed). Additionally, hand-deisgned heuristics and domain knowledge are used to improve the search, as well.

AlphaGo (1) and AlphaZero (4) represent a significant leap forward by combining tree-search and reinforcement learning with deep neural networks to create an agent that exceeds human performance in the game of Go. AlphaGo relies on recent advances in tree search (5), as well as improvements in computational power to train deep neural networks on massive amounts of data. AlphaGo relies on data collected both from self-play and human demonstrations, while AlphaZero trains entirely from self-play. Both methods combine neural network-based value estimation with tree search to achieve superhuman performance on the game of Go.

Ideas from AI have been applied to Catan in the past. One of the earliest instances is Jsettlers (6), an implementation of Monte-Carlo Tree Search for Catan. (7) and (8) both employ reinforcement learning techniques to the main game of Catan. All previous works in this area focus on playing the game of Catan after the initial placements phase. (6) use Monte-Carlo tree search to select actions and thus do not require learned functions for value estimation. (7) and (8) both train function approximators on self-play in order to select optimal actions. (7) use hand-crafted features, while (8) transform the hexagonal board into an image-like grid in order to use convolutional neural networks for value estimation.

3 METHODOLOGY

Catan differs from traditional AI benchmarks in several major ways:

1. The game contains a high degree of stochasticity, as resource collection is intrinsically tied to a roll of the dice.
2. The game contains an initial placement phase, with rules distinct from those of the main game.
3. The game is played with four players instead of two.

We use deep RL to train an agent to play the initial placements phase of Catan. We chose to focus on this area of the game because initial placements significantly affect the outcome of the game. We conducted a survey to verify this. We also argue that it is also more interesting strategically. Compared to 3 to 4 possible actions during the main game, there are a total of up to 114 distinct actions a player can take in the initial placements phase. When making decisions in this phase, skilled Catan players have to take into account many factors, such as the relative scarcity of resources, whether they can use ports, as well as consider all of these factors for each of their opponents.

3.1 FORMULATING THE MARKOV DECISION PROCESS

Since we decide to focus on the initial placements phase of the game, we need to formulate this phase as a Markov Decision Process. A Markov Decision Process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma)$, which correspond to a state space, action space, reward function, transition function and discount factor. These items will be explaned in further depth in this section.

3.1.1 STATE SPACE ($\mathcal{S}$)

The state space for this MDP is straightforward - it is simply the collection of all the information available about the board state, such as current settlement locations, the location of resources on the board, etc. Additionally, there is no hidden information in the initial placements phase. Thus there is no need to model the problem as a partially observable MDP (POMDP).

3.1.2 ACTION SPACE ($\mathcal{A}$)

The action space for this MDP is quite simple; it is simply 114 discrete actions corresponding to every possible initial placement (a combination of a settlement location and road direction).

3.1.3 REWARD ($\mathcal{R}$)

Defining a reward function for this MDP is non-trivial as our goal is to maximize our agent's win rate; a quantity not readily obtainable from initial placements alone. In order to construct a reward that reflects the overall win rate from the initial placements phase, we designed a heuristic agent using common-sense Catan rules (like the random rollout policy in JSettlers) and had the four copies of this agent play themselves from the initial placements until the game ended. From there, we can use the simulated win rates as reward. Since Catan has a high degree of stochasticity, we average our win rates over ten games.

Thus, we are able to design a reward function for our initial placements MDP as follows: If the game is not in a terminal state (a state is terminal if and only if all 8 initial settlements are placed), we say the reward is 0 for all agents. Otherwise, use the heuristic agents to simulate a game and use the simulated win rates as reward.

3.1.4 TRANSITION FUNCTION ($\mathcal{T}$)

This is relatively straightforward. Only the current player is able to transition the MDP. The transition function itself only requires placing a settlement according to where the action says to place.

3.2 Augmenting the Markov Decision Process for Multiple Players

Since there are four players in a game of Catan, we need to augment the Markov Decision Process framework to handle multiple agents. Traditionally, a multi-agent Markov Decision Process augments the transition function to admit actions from all agents simultaneously. However, since actions in a Catan game happen sequentially, and there is no hidden information in initial placements, we can simplify the multi-agent MDP formulation somewhat to:

1. $\mathcal{S}$ The current information of the board, available to all players, as well as the current player's turn.
2. $\mathcal{A}$ One of the available settlement locations from the current state
3. $\mathcal{R} : (\mathcal{S}, \mathcal{A}) \to \mathbb{R}^4$ as each player gets their own reward.
4. $\mathcal{T} : \mathcal{A} \to \mathcal{S}$ which will only transition using the current player's action.

3.3 Value Estimation with Multi-agent Q-learning

In order to construct a policy to do initial placements, we use a variant of Deep Q Networks (9) for multiple agents.

Given that there are four agents at any given time, we use four Q functions to perform value estimation. Notationally, we will refer to the Q value of a state-action pair s, a for player i as $Q(s, a, i)$. As in traditional Q-learning, we train our neural network to minimize the Bellman loss:

$$J(\theta) = (Q_\theta(s, a, i) - (r_t + \gamma \max_{a'} Q_{\theta'}(s', a', i)))^2 \tag{1}$$

where θ' are the parameters of a target network that is updated with Polyak averaging per (10). ($\theta'_{t+1} = \beta\theta'_t + (1 - \beta)\theta_t$)

For agents that don't act in a given state transition tuple (s_t, a_t, r_t, s_{t+1}), we can still recover additional training signal by enforcing a Bellman loss on that agent's V function:

$$J(\theta) = (V_\theta(s, i) - (r_t + \gamma V_{\theta'}(s', i)))^2 \tag{2}$$

where we can compute $V(s)$ as $\max_a Q_{\theta'}(s, a)$

3.4 Planning via Monte-Carlo Tree Search

We use Monte-Carlo tree search (5) with our learned Q-function to implement our Catan-playing policy. Monte-Carlo tree search is a variant of tree search that uses random game rollouts to estimate the value of game states explored in the tree. At any given time, the agent explores the tree according to $\arg\max_a \{Q(s, a) + u(s, a)\}$. When the agent reaches a leaf node s, it will compute the value $v(s)$ and propagate its information up to its parents as $W(s)$. This process is repeated for some amount of time. Afterwards, the best action can be selected from the root node r as $\arg\max_a W(t(r, a))$, where t is the MDP's transition function. (i.e. pick the child of the root with highest win rate).

Like in (1), we use a combination of random rollouts and Q-function evaluations to compute the value of various board states, where the value of a given board state is given as:

$$v(s) = \lambda V_\theta(s) + (1 - \lambda) r(s) \tag{3}$$

where $V_\theta(s)$ is obtained using the trained Q-function (i.e. $V_\theta(s) = \max_a Q_\theta(s, a)$) and $r(s)$ is obtained by using random actions until the state is terminal, then computing the win rates at that state. A hyperparameter λ is used to control the relative importance of the network prediction and random rollout.

Like in (1), we use the PUCT exploration described in (11) to encourage the agent to balance exploitation and exploration of promising nodes:

$$u(s,a) = c_{puct} p(s,a) \frac{\sqrt{\Sigma_b N(s,b)}}{1 + N(s,a)} \tag{4}$$

This exploration bonus $u(s,a)$ is added to the current win rates $Q(s,a)$. c_{puct} is a hyperparameter that controls the strength of the exploration bonus. $p(s,a)$ is a pre-computed probability of choosing action a in state s. Unlike (1), we did not train a policy π that computes action probabilities, given state. Instead, we follow the approach from (12; 13; 14) and use the Boltzmann distribution with energy $Q(s,a)$ to compute $p(s,a) = \frac{exp(Q_{\theta(s,a)})}{\Sigma_{a'} exp(Q(s,a'))}$. The final term is a function of the current node's visit count $\Sigma_b N(s,b)$ and its child's visit count $1 + N(s,a)$, which decreases as the child node s,a is visited more.

3.5 SELF-PLAY

In order to collect transition tuples to train our Q-functions on, we use self-play in a similar style to AlphaGo. We accomplish this by keeping track of a set of parameters θ^* that correspond to our current best-performing agent. After every epoch, we compare our newly trained parameters θ to θ^* by having two agents parameterized by θ and θ^* play against each other. Since there are four players in any Catan game, we randomly select two players for each agent to control (two players controlled by the same agent are not allowed to cooperate). If the agent parameterized by θ achieves a win rate of at least α against θ^*, we replace θ^* with θ. The full algorithm is presented as Algorithm 3.

Given our agent θ^*, we generate actions from a state s by computing $Q(s,a,i)$. Given our Q-values, we use an ϵ-greedy exploration strategy and take $a = \arg\max_a Q(s,a,i)$ with probability $1 - \epsilon$ and take $a \sim unif(\mathcal{A})$ with probability ϵ.

4 IMPLEMENTATION

4.1 CONSTRUCTING FEATURES FROM BOARD STATE

In order to use neural networks to perform board placements for Catan, we first need to construct a feature vector from a given Catan board. In general, we consider there to be four main entities in a game of Catan:

1. Settlement locations: The spots on the board where players place their settlements. Features include which player has a settlement built there, whether that settlement is a city or not, or what kind pf port is there (if any). We also compute a feature that we call production, or expected number of resources that settlement will produce per dice roll.
2. Road locations: The spots in the board where players place roads. The only feature to keep track of is the player that has a road built there.
3. Tiles: The resource-producing spots on the board. We use the resource type, the dice value that the tile produces on, and whether the tile is being blocked by the robber as features for tiles.
4. Players: For each player, we keep track of their resources, development cards, and the rates at which they can trade resources for other resources.

We concatenate the features of all entities into a single feature vector to represent the state of our Catan board.

4.2 Q-LEARNING WITH GRAPH NEURAL NETWORKS

In practice, we observed that using a simple feed-forward neural network yielded poor performance. Given the high dimensionality of the neural network input (over 1400 features), we decided it was appropriate to use a graph neural network to impose additional structure on the learning problem.

We chose to construct our Catan graph using settlement locations as vertices and roads as edges. Two settlement vertices were connected by an edge e if and only if the edge e connected the two

Table 1: State Space of the Catan MDP

Entity	Number per Board	Features	Total Features
Road	72	Player (5)	360
Settlement	54	Player (5), Port (6), City (3), Production (1)	810
Tile	19	Resource Type (6), Dice Value (1), Is Blocked (1)	152
Player	4	Player Number (4), Resources (5), Dev Cards (5), Trade Ratios (5)	76
Other	1	VP (4), Longest Road (4), Largest Army (4)	12
Total	1	-	1410

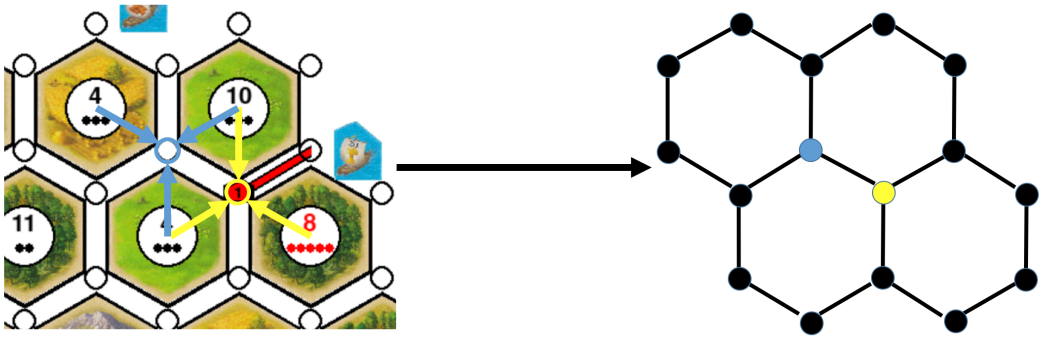

Figure 2: A visualization of a subsection of the graph we perform inference on. Settlement locations are represented as vertices, and roads as edges. Settlement features include the tile features of bordering tiles (as denoted by the colored arrows and vertices). Tile associations for the black vertices omitted for clarity.

settlement locations. Each edge and vertex contained features corresponding to the entity that they represented. Settlement features also included the tile features of the tiles that that settlement spot bordered. We present this visually as Figure 2.

We chose to implement a modified version of GraphSAGE (15) as our Q-function.

Algorithm 1: GraphSAGE with edge features

Input: Vertices V, Edges E, Vertex features x_0, global features z_0, Embedding functions $F_{[0-N]}$, Aggregator functions $A_{[0-N]}$, Linear Layer L
Output: Q values q
for $i \in N$ **do**
  for $v \in V$ **do**
    $k_i = A_i(v, neighbors(v), edges(v))$
    $x_{i+1} = F_i(k_i)$
    $x_{i+1} = x_{i+1}/||x_{i+1}||_2$
  end
end
$q = L(v_N \cup z_0)$;

GraphSAGE uses neural networks to compute vertex features by constructing per-vertex embeddings of each vertex and its one-hop neighbors. This is accomplished via an aggregation step followed by an embedding step. implement a slightly modified version of the pooling operation described in the paper.

Essentially, aggregation is performed by using a neural network to embed the out-neighbors of v and their corresponding edges. From there, a neighborhood embedding is extracted by taking the element-wise max over each neighbor u. This enforces a constant amount of features per vertex, regardless of the underlying graph structure at that vertex.

Algorithm 2: Pooling Aggregator with Edge Information

Input: Vertex v, Vertex features x_v, Edge features $y_{u,v}$, Aggregator MLP f_θ
for $u \in$ *neighbors(v)* **do**
  $z_u = f_\theta(x_u | y_{u,v})$
end
$z = \max((z_u) \forall u \in \text{neighbors}(v))$

The receptive field of GraphSAGE is equal to its depth. We use fewer layers in our network than the diameter of the graph induced from the Catan board (we use five layers in our GNN, while the diameter of the Catan board is 11). Thus using this formulation, it is not guaranteed that vertex features can depend on distant vertices. Since the value of a given settlement spot can depend on distant settlement spots in addition to local ones, we allow for long-range dependencies by using a fully connected layer on the final graph embeddings (which we flatten into a vector) and global features (such as the Player and Other features in Table 4) to compute our Q values.

Algorithm 3: DQN with Alphago-style Self-Play

Input: Untrained Q-function parameters θ, Replay Buffer $\mathcal{D}$, win threshold α, discount factor γ, Learning rate η, Polyak term β
Output: Trained Q-function parameters θ
$\theta' \leftarrow \theta$
$\theta^* \leftarrow \theta$
while *not done* **do**
  $\tau = \text{rollout}(\theta^*)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \tau$
  for *train iterations* **do**
    $(s, a, r, s', t, p) = \text{sample}(\mathcal{D})$
    $J(\theta) = \begin{cases} -(Q_\theta(s,a,i) - (r + \gamma \max_{a'} Q_{\theta'}(s', a', i)))^2 & \text{if } i = p \\ -(V_\theta(s,i) - (r + \gamma V_{\theta'}(s', i)))^2 & \text{otherwise} \end{cases}, \forall i \in [0, 3]$
    $\theta \leftarrow \theta + \eta \frac{\partial J}{\partial \theta}$
  end
  $z \leftarrow \text{compare}(\theta, \theta^*)$
  if $z > \alpha$ **then**
    $\theta^* \leftarrow \theta$
  end
  $\theta' \leftarrow \beta\theta' + (1 - \beta)\theta$
end
return θ^*

5 Experimental Setup

5.1 Baselines

To evaluate the performance of our method, we perform several ablation studies, comparing our algorithm (referred to as MCTS + QF) to:

1. MCTS without a Q function (MCTS)
2. Q function without MCTS (QF)
3. A heuristic policy that chooses the settlement spot with the highest production (and randomly chooses if there are multiple spots with the highest production) (Heuristic)
4. A policy that chooses settlement locations at random (Random)

We compare to MCTS and Q-learning baselines to determine whether the combination of learning and planning yields better results than either alone, as well as the relative importance of each in our

Method	Win Rate Against MCTS + QF	Win Rate of MCTS + QF (ours)
MCTS	47.5 %	52.5 %
QF	41.2 %	58.8 %
Heuristic	34.4 %	65.6 %
Random	8.8 %	91.2 %

Table 2: The performance of our method compared to several ablations.

final system. We compare to the heuristic and random policies as sanity checks; we expect our system to outperform both methods.

For each ablation study, we compared our method to the others by playing 50 games. For each game, each algorithm controlled two players, chosen at random for each game. We report the win rates of each agent (where a win was given to a given algorithm if either of its players won, and ten games were played from each position) in the following tables. The MCTS-based algorithms were given three minutes thinking time (this equated to around 2000 simulations using 12 threads on a single laptop via Ray (16)). The results are presented in Table 2.

Overall, we find that our method (MCTS + QF) performs the strongest, as evidenced by all other methods having a win rate of under 50 % against it. While our method is able to outperform others, we note that the effect of the learned Q function appears to be quite small, as evidenced by the relatively good performance of MCTS. Another point is illustrated in our relatively strong performance against a heuristic baseline. This highlights the importance of strategic decision-making in the initial placements phase of the game, as opposed to relying on simple heuristics.

5.2 Comparison to Human Preferences

In addition, data on human preferences were obtained survey link. A copy of the survey questions can be found here. Sixteen game state boards, each with varying placement positions, were randomly generated with colonist.io, an online Catan board game simulator. For each game state, participants were asked where they would place their initial settlement and the direction of their road. This dataset consists of 18 responses, and should not be considered statistically significant. In the two cases detailed in the tables below, Humans preferences from the set are compared to Catanbot. Human preferences were first filtered, they had to have reported winning 25% or more of games and had to have played more than 5 games. This effectively dimishes our dataset to 11 'expert' Catan player survey responses. Responses with only 1 player were not included in consideration, hence the 'N/A' in many lower ranks. Catanbot was then given the same board, running on 12 threads with an exploration factor of 12, and a search time of 3 minutes. The results are compared below in Tables 3 and 4. It is observed than humans tend to pick at least one of the spots that Catanbot picks, but with an overwhelming emphasis on the settlement placement, with a large variation in road placement.

Table 3: MCTS vs Human Preferences, Case 10 in survey

Rank	Catanbot RL	Humans
1	44,2	44,2
2	8,2	44,1
3	44,1	44,0
4	33,1	N/A
5	8,1	N/A

Table 4: MCTS vs Human Preferences, Case 14 in survey

Rank	Catanbot RL	Humans
1	37,2	21,0
2	37,0	21,2
3	21,0	21,1
4	37,1	N/A
5	44,2	N/A

6 Conclusion and Future Work

We presented a system that plays the initial placements phase of Catan using a combination of deep reinforcement learning and tree search. While this task is not intrinsically useful to robotics, the techniques applied here to address aspects of long-term planning, and reasoning with uncertainty are key topics in current robotics research. Furthermore, the techniques described in this paper are very general. Aspects of the algorithm presented here (like tree search and reinforcement learning) have already been used on robotic systems.

While evaluating the experiments, we disagreed with some of the moves that our placement agent made. We believe that this is due to the playing strength of the heuristic agent we designed for the main game. Additionally, we believe that performance improvements to the underlying MCTS would improve results as well (we were unable to reach search depths of more than 4 or 5 moves).

Overall, we believe that this project was an interesting look at modern AI techniques that combine planning and learning. We are confident that the skills we learned here can translate to robotics domains in our future research.

7 Acknowledgements

The authors would like to thank Benjamin Jensen, Andrew Galassi, Sophia Yoo, Steve Kovacic, Ally Mangine, Michael Vinciguerra, Sinead Kim, Jose Loli, Nataliya Rokhmanova, Erik Rundlett, Ian Gonzalez Afanador, Ryan Henne, Elizabeth Shaloka, Aravind Vadali, Adam Brock, Benjamin Stankovic, Justin Cai, Nathan Cai, Julia Triest and Emma Krampe for helping us collect Catan data.

References

[1] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[2] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[6] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Advances in Computer Games*, pages 21–32. Springer, 2009.

[7] Michael Pfeiffer. Reinforcement learning of strategies for settlers of catan. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[8] Quentin Gendre and Tomoyuki Kaneko. Playing catan with cross-dimensional neural network. In *International Conference on Neural Information Processing*, pages 580–592. Springer, 2020.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[10] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

Hyperparameter	Value
# Layers	5
Embedding Dimension	[64] * 5
Aggregation Dimension	[64] * 5
Nonlinearity	Tanh
Output Dimension	16

Table 5: Hyperparameters for GraphSAGE

Hyperparameter	Value
Learning Rate	1e-4
Optimizer	Adam (17)
Buffer Size	500000
Polyak Averaging	0.995
Batch Size	256
Discount	0.99
ϵ	0.05
Rollouts Per Epoch	200
Comparison Rollouts Per Epoch	200
Win Threshold	0.55

Table 6: Hyperparameters for AlphaGo-style Q-learning

[11] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[12] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

[13] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. *arXiv preprint arXiv:1702.08165*, 2017.

[14] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[16] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

A HYPERPARAMETERS

This section includes the hyperparameters that we used for various parts of our experiments. The hyperparameters for GraphSAGE, Q-learning and MCTS are provided in tables 5, 6, and 7 respectively.

Hyperparameter	Value
c_{puct}	2.5
λ	0.5

Table 7: Hyperparameters for MCTS