Using Reinforcement Learning to Optimize the Rules of a Board Game

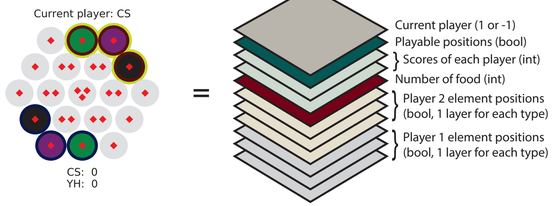
CS 229 (Machine Learning) Final Project, Fall 2019
Gwanggyu Sun (ggsun@stanford.edu), Ryan Spangler
Department of Bioengineering, Stanford University

Abstract

We used the deep reinforcement learning framework used by AlphaGo Zero[1] to train an agent that plays *Organism*, a 2-player board game that is actively undergoing development. The resulting agent was able to play a simplified version of the game at a competitive level, and was able to rapidly generate self-play data that would normally require input from human beta-testers. We plan to use this data generated by the agents to find the optimal rules of this game that generates the most fair and fun gameplay.

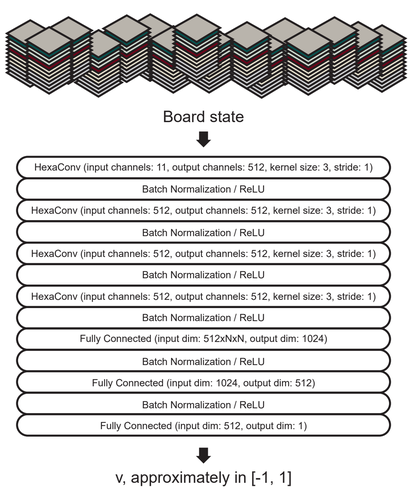
Board Representation

At each point in the game, the state of the board was represented as an array of size 11xNxN, where N is dependent on the size of the game board. This array was used as the input to the convolutional neural network.



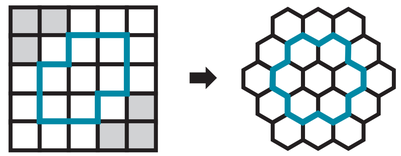
Convolutional Network

Each agent uses a single convolutional neural network that takes the current board state as the input, and outputs a single value v, which is closer to 1 if the player 1 is likely to win the game, and closer to -1 if the player 1 is likely to lose. The network consists of four "HexaConv" layers followed by three fully connected layers.



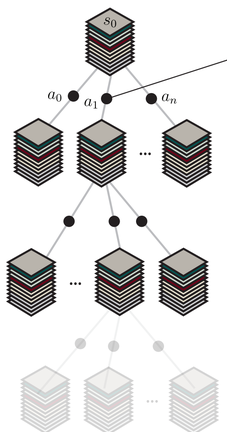
HexaConv Layers

Since the geometry of the board is hexagonal, conventional convolutional layers with rectangular kernels cannot accurately capture the adjacencies of neighboring board positions. Thus, we used square kernels that are masked off at the two corners, which correctly captures the adjacencies of a "skewed" hexagonal grid[2].



Monte Carlo Search Tree

At each point in the game, an agent chooses the next move by executing 25 rounds of Monte Carlo rollouts on a search tree. The probabilities of choosing each move are determined by the average values of v's returned by the neural network for the series of board states that follow each move.



$N(s,a)$: Number of times action a was chosen for state s

$Q(s,a)$: Average value of $V(s')$ observed for choosing action a for state s

$$p(a_j) \propto Q(s,a_j) + P(s,a_j) \cdot \frac{\sqrt{\sum_k N(s,a_k)}}{1+N(s,a_j)}$$

Early stages: Pick actions with high $P(s,a_j)$
Later stages: Pick actions with high $Q(s,a_j)$

$$P(s,a_j) \propto e^{-V(s'_j)/T}$$

Favor actions that will return the most negative results for the opponent in the next turn; hyperparameter T determines the propensity of the tree to choose suboptimal moves.

Learning Protocol

1. Using the current best agent, execute 250 (early stages) or 100 (later stages) games of self-play. Log the board state (plus all of its symmetric forms) at each turn and the final results of the game.
2. Retrain the neural network using the additional self-play data gathered from this round.
3. Pit the agent with the newly trained network against the current best agent 50 times. Log the winner of each game.
4. Choose new agent as the best agent if it wins more than 60% of the games. Repeat until convergence.

Results

An untrained agent makes seemingly random moves, and is unable to avoid moves that will immediately result in a loss. Games are short (1-3 moves per player), and usually ends when either player makes a suicidal move.

After just one iteration of self-play, the agent becomes significantly better at avoiding suicidal moves, but is generally unable to "attack" the opposing player in a coordinated way. Games last longer (3-8 moves per player) and is often won by the player that avoids suicidal moves for a longer time.

After further iterations of self-play, the agent learns how to "grow" its organism, and find vulnerabilities in the opposing player's organism that could be easily attacked. Games last much longer (5-20 moves per player), and an increasing fraction of games are won by coordinated attacks on the other player's organism, rather than bad moves made by the opponent.

Lessons Learned

- Representing the board state as an array of numbers is one of the most difficult parts in applying this framework to other games. You will have to think of the most concise, yet complete, representation of the board that is suitable for training neural networks.
- Hyperparameters matter! Temperature parameters that determine the degree of "exploration" are critical in determining the behavior of the agent. Temperatures that are too high will lead to frequent bad moves. Temperatures that are too low will lead to repetitive rounds of games that are not suitable to be used as training data.
- A constant-length action space is desirable, but it is not required to train these networks. When the number of possible actions increase, however, evaluating the value of all the next possible states through the trained network becomes computationally expensive.

Future Work

- Train agents that play on bigger boards, up to the full-size version of the game (5 rings).
- Experiment with different versions of the game with different rules, including the initial state of the board, win conditions, and tiebreaking conditions. Observe how the behaviors of trained agents change with modified rules.
- Devise a metric to evaluate the fairness and the degree of dynamic gameplay generated by the game.
- Pit the trained agent against a human player!

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140-1144 (2018)
[2] E. Hoogeboom, J. W. Peters, T.S. Cohen, and M. Welling. Hexaconv. *arXiv:1803.02108* (2018)
[3] https://github.com/suragnair/alpha-zero-general