Using Reinforcement Learning to Optimize the Rules of a Board Game

Gwanggyu Sun, Ryan Spangler
Stanford University
Stanford, CA
{ggsun,spanglry}@stanford.edu

Abstract

Reinforcement learning using deep convolutional neural networks has recently been shown to be exceptionally powerful in teaching artificial agents how to play complex board games. We have applied the learning framework used by Google Deepmind's AlphaZero to train an agent that plays "Organism", a board game that is undergoing active development. The hexagonal geometry of the game board and the possibility of multiple moves per turn required major modifications from the existing framework. The resulting agent was capable of playing the game at a competitive level, and provided massive amounts of valuable play data that could be used to optimize the rules of the board game before its release.

1. Introduction

In recent years, reinforcement learning using deep convolutional neural networks has proven to be exceptionally powerful in teaching artificial agents how to play complex board games and achieving superhuman performance. Most famously in 2016, Google DeepMind's AlphaGo defeated a top-tier professional player in the game of Go, which was once thought by many to be an impossible feat. AlphaGo used a combination of expert human moves and self-play results as its training data for its "value" and "policy" deep networks [2], which were iteratively improved through continuous rounds of self-play. A year later, DeepMind released a new version of the agent named AlphaGo Zero [4], which was trained by using only self-play data and a single, combined value-policy network. Alphago Zero outperformed the original version of AlphaGo by a significant margin. Since then, AlphaGo Zero's reinforcement learning framework has been generalized to train agents that can play numerous other board games, including chess and shogi, at human or super-human levels [3].

However, most, if not all, of these applications have focused on training neural networks to play an established board game with a fixed set of rules. *Organism* is a board game that is actively being developed by Ryan Spangler, who was previously one of the developers of the board game *Sol: Last Days of a Star*. Like many other board games that have been successfully conquered by reinforcement learning, *Organism* is a two-player, "perfect information" game, where both players have full knowledge of the state of the board at all times during the game. Although the core frameworks of the rules of the game have already been established, many details are still left undecided, including the size of the board, the initial state of the board, and a player's winning conditions. In this work, we aimed to train a neural network agent that can play this game at a competitive level, and use the self-play data generated from this model to discover a set of rules that will lead to a dynamic and fair game. The generation of this self-play data will ideally replace the "play test" phase of board game development, which can be costly and imperfect due to limitations on how fast human players can learn to play the game at an expert level. To our knowledge, this work is the first application of a reinforcement learning algorithm in the development of a novel board game.

2. Methods

Our approach closely followed the methods used by AlphaZero [3], which successfully generalized AlphaGo Zero's learning framework to other board games of a similar style to Go. An open-source repository[1] that provides the basic framework to apply AlphaZero's framework to an arbitrary game of choice was used as the starting point for our project.

AlphaZero uses a combination of deep convolutional neural networks (CNN) and Monte Carlo Tree Search (MCTS) to determine the next move that an agent will make. The CNN takes the current game state (represented as a 3D tensor) and produces two outputs - a scalar $v(s)$ that predicts the probability that a given player will win the game (value network), and a vector that predicts the probabilities $p(s, a)$ of the next moves that a player can make (policy network). After a game is completed, the result of

[1] https://github.com/suragnair/alpha-zero-general

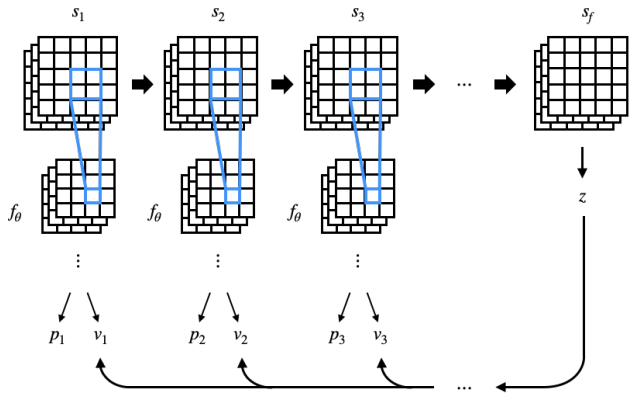

Figure 1. The basic reinforcement learning framework of AlphaZero. The CNN takes each state s_i from a self-play game as inputs, and outputs a scalar (v_i) that predicts the winner of the game, and a vector (p_i) that predicts the next move. Once the round of self-play is over, the actual result of the game (z) and the probabilities of moves selected by the MCTS (not shown) are used to update the parameters θ of the CNN.

each game, z, and the probabilities of each move chosen by MCTS, π, are used to update the parameters of the CNN, such that the values given by the value network and the policy network lie closer to actual observed values z and π (See Figure 1). When selecting a move, both the probabilities $p(s, a)$ given by the policy network and the values $v(s')$ given by the value network for the resulting states are considered. More details on this approach can be found in Silver et al., 2018 [3]. Due to some unique characteristics of *Organism*, however, two important modifications needed to be made to the original framework used by AlphaZero.

Unlike games like Go and chess where the positions of the board can be easily represented as coordinates on a Cartesian grid, *Organism* uses a hexagonal game board where each position on the board is adjacent to at most six other positions. Thus, the square-shaped kernels (filters) used by conventional CNNs and the CNN used in AlphaZero cannot accurately reflect the adjacencies of different positions on the hexagonal board. To ensure that this geometry of the board is taken into account in our convolution operations, we needed to adopt an unconventional convolution operation known as "HexaConv" [1], which uses a square kernel that is masked off at the two opposite corners such that the resulting convolution correctly covers adjacent positions on a hexagonal board (See Figure 2). To use this method, we also had to ensure that positions on the hexagonal grid are translated correctly into positions on a square grid such that the masked square filters always cover a continuous hexagonal area on the board.

Another unique feature of *Organism* is that a single player can make multiple, discrete moves on one's turn, whereas in chess or Go the number of moves is strictly limited to one per turn. This feature of the game drastically

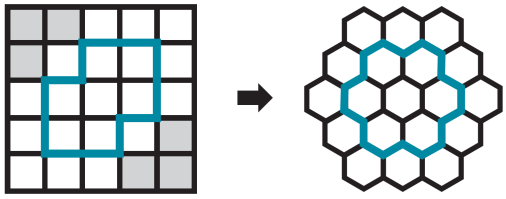

Figure 2. Representation of hexagonal grids and filters with masked 2-D matrices. The hexagonal board consisting of 19 positions can be represented by a 5×5 matrix where three positions each from two of the corners (left, gray) are masked out. A 3×3 kernel where two corner positions are masked out (left, blue) translates into a kernel that covers seven positions on a hexagonal grid (right, blue).

increases the number of possible actions that can be made in one turn, which makes it extremely difficult to represent $p(s, a)$, the probability that a policy network will take action a given state s, as a fixed-length vector. To avoid this issue with the policy network, we chose to rely solely on the values of v during MCTS - instead of using $p(s, a)$ as the initial probability for taking action a, we used the expected value of the resulting state $V(s')$ to determine the probabilities for each action a (See Figure 5 for more details). A drawback of this approach is that the width of the search tree will be significantly expanded, since we will need to compute the value of $V(s')$ for every state s' that is reachable from state s with a single action. Since the number of legal actions that can be made given a state is roughly in the scale of hundreds for *Organism*, we did not expect this would be prohibitively expensive in terms of computation time - indeed, for the board sizes tested so far, computation time has not been a serious issue.

With these considerations in mind, we started by constructing an $11 \times N \times N$ array-form representation of the board state, where each position on the hexagonal grid is mapped to a specific position on a square array (See Figure 3) such that HexaConv layers would cover a correct range of positions. These array representations of board states were fed in as inputs to a 7-layer neural network composed of 4 HexaConv layers and 3 fully connected layers (See Figure 4). Each layer was immediately followed by batch normalization and ReLU layers. Similar to AlphaZero, this neural network was trained to return the predicted winner of the game given the current board state, with a value closer to 1 meaning player 1 will win the game, and a value closer to -1 meaning player 2 will win.

Using the trained CNN, our agents chose the action that will be taken given a board state by performing MCTS (See Figure 5). At each point in the game, an agent performs 25 rollouts from the current state, and either selects an action with a probability that is proportional to the number

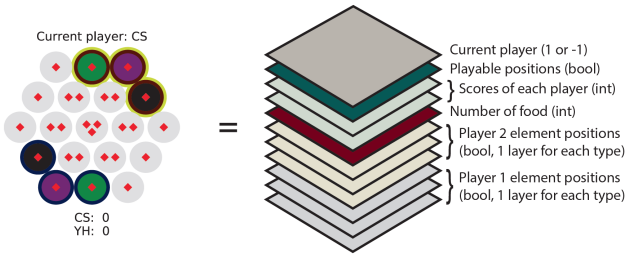

Figure 3. Representation of a board state as a 3D tensor. Each position on the hexagonal board with m rings was translated into a position on a square grid of size $N = 2m-1$. Each layer of the 3D tensor was designed to hold information on either (i) the locations of particular game elements on the board, or (ii) the general state of the board including the current player and the number of scores scored by each player. The layers were designed such that each HexaConv filter would have full information of the game inside the range of positions that it covers, plus all information about the state of the game that is required to estimate the winner.

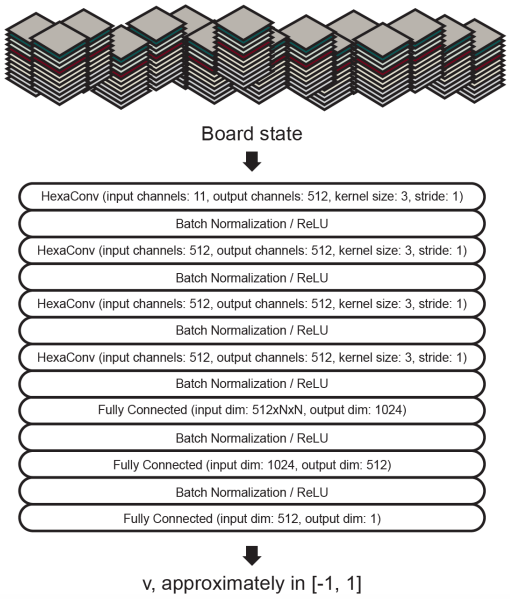

Figure 4. The architecture of the convolutional neural network used by each of our agents. The CNN takes the array representation of board states as inputs, and uses four HexaConv layers and three fully connected layers to return a single value v that gives a prediction for the winner of the game.

of times the action was chosen (when the agents are generating self-play data), or selects an action that was most frequently chosen (when the agents are playing an actual game). The purpose of using different selection policies for self-play and actual games is to promote more exploratory behavior during self-play, while enforcing agents to make "safe" moves during an actual game. On each rollout, the probability of selecting move a_j is initially determined by $P(s, a_j)$, which is proportional to $e^{-V(s'_j)/T}$, where $V(s'_j)$ is the value of the next state s'_j that action a_j will lead to, and T is a temperature parameter that controls the degree of exploration. Thus, the probability of action a_j being chosen will be higher if the next state s'_j is predicted to be more favorable to the current player. In later rollouts, the probability of selecting action a_j converges to $Q(s, a_j)$, which is equal the average values of $V(s)$'s that are observed for the states that follow after choosing action a_j from the current state. In essence, MCTS was designed to help the agent choose actions that will be beneficial to the current player in the short-term (by choosing actions with high $P(s, a_j)$) and in the long term (by choosing actions with high $Q(s, a_j)$).

Using these individual components, agents were trained with the following general learning protocol: First, we generated training data by making the current best agent play against itself 100 times, and logging the board state (plus all of its symmetric forms) at each stage of the game and the final result. This data was used to train the neural network such that it can correctly predict the winner of the game given a board state that had occurred during the game. Then, the agent with the newly trained network was pitted against the the current best agent 40 times. If the updated agent won more than 60% of the games, we chose the updated agent as the new best agent. This process was repeated until convergence.

Python 3.6.6 was used for all computations, with PyTorch 1.3.1 as the library for training CNNs. Initial development was performed on a local machine, but all final results were generated on a Google Cloud virtual machine. All of the code used for this project can be found in our Github repository[2].

3. Results

We used our learning framework to train agents that can play a simplified version of the game, with a smaller board (3 rings instead of 5 rings) and a simplified win condition (eliminate the other player), to demonstrate that the model is capable of learning this game. Figure 6 shows the sample games that were played by the agents at different stages of the training cycle.

Without any training (Figure 6A), agents would not have any prior information on which board state is desirable - the values of the initial probabilities $P(s, a_j)$ would come from a distribution that is nearly uniform. Repeated iterations of MCTS rollouts would tend to steer the agents away from making "suicidal" moves that will immediately cost them the game, as the values accumulated for $Q(s, a)$ will decrease when the agent observes a lost game further down the branch. Nevertheless, untrained agents seem to be unable to avoid suicidal moves efficiently, and games between un-

[2]https://github.com/prismofeverything/organism/tree/alphazero

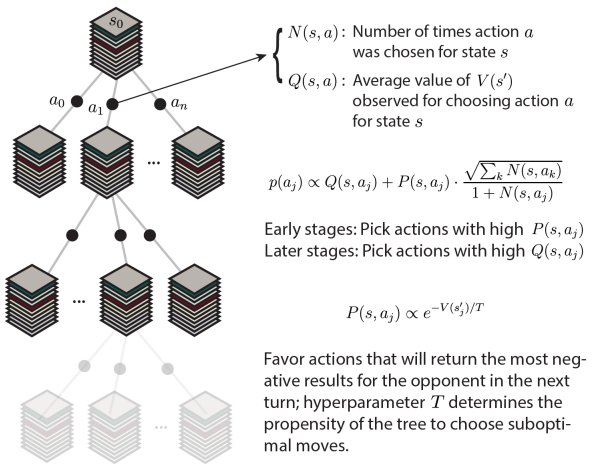

Figure 5. The Monte Carlo Tree Search algorithm used by each agent to select the next move. Each branch in the tree holds two values $N(s,a)$ and $Q(s,a)$ that hold the total number of times the branch was visited and the average board state values that were observed by following that branch, respectively. At each rollout, the probability of choosing action a_j is determined by a linear combination of $Q(s,a_j)$ and $P(s,a_j)$. At earlier stages of the rollouts where N is generally low, $P(s,a_j)$ is more heavily weighted. At later stages when N is high, $Q(s,a_j)$ becomes more heavily weighted. The temperature hyperparameter T used in calculating $P(s,a_j)$ controls the level of exploratory behavior encouraged by the values of $P(s,a_j)$.

trained agents usually end by one agent making a move that disintegrates its own organism, and thus losing the game. This shows that while the MCTS should be able to assist agents to avoid bad decisions, correctly evaluating different states through a trained CNN is critical for the agents to play the game effectively.

After just a single iteration of self-play (Figure 6B), we observed that the agents were significantly better at avoiding suicidal moves - the values returned by the CNN for "lost" game states were close to -1, meaning that the values of $P(s,a_j)$ for actions that will lead to these states were effectively close to zero. The games played by these agents, thus, lasted for longer turns, as both agents were able to choose moves that would not disintegrate their own organism. The agents, however, were still unable to make any coordinated attacks on the opposing player's organism.

After five iterations of self-play (Figure 6C), we began to observe more sophisticated plays generated by our agents. Most agents chose to "grow" an "EAT" element as their first move, which has a higher importance on a smaller board than the other two elements, "GROW" or "MOVE", because of the limited number of open spaces on the board. Some agents also chose to move their single "MOVE" elements closer to the center of the board, which is a requirement for the agents to perform attacks on the opposing player's organism. After such actions are taken, the agents were quickly able to identify weak positions in the opposing player's organism, and initiate attacks with the correct type of element that would effectively eliminate the other player from the board.

From the self-play games generated by this agent, we were also able to discover that the agent that plays the first move of the game had a win percentage of approximately 70%, which implies that the first player may have an unfair advantage over the other player on this small board. Indeed, it is likely that with a smaller board size, making the first move becomes a more significant advantage, which is illustrated well by how the komi (the extra points that a player receives for playing second) changes in Go depending on the size of the board. Thus, if the game does get played at this board size, it seems reasonable to give the second player a certain handicap to alleviate the disadvantage. The appropriate level of handicap could also be determined by generating self-play data with these agents, and observing which settings lead to win percentages that are close to 50%.

Trained agents were able to perform the set number of rollouts and return the next move in approximately 30 seconds, which led to a typical self-play game consisting of 8 moves per player taking about 5 minutes. Further iterations of self-play rounds were unable to yield agents that are significantly better at the game - no agent trained so far was able to defeat the agent in Figure 6C with a win rate above 60%. To train stronger agents, we may need to use deeper convolutional networks or more rollouts on the MCTS, which likely will increase the time required for agents to compute the next move.

4. Discussion

In this work, we were able to successfully apply AlphaZero's reinforcement learning framework to teach an artificial agent to play *Organism* at a competitive level, and generate self-play data that allowed us to learn more about the dynamics of the game and evaluate its fairness. The modified form of MCTS that uses only the values of v returned for states was still capable of guiding the agents to make the correct moves. Two key drawbacks of this approach is that it vastly increases the width of the search on the tree by requiring the evaluation of all possible next states, and that it requires the addition of another temperature hyperparameter which controls the level of exploration performed by the agent. The number of iterations required to train the agents and the final play level of the agents both depended heavily on how this hyperparameter was set - if the temperature was too high, agents made too many risky moves and was not able to generate self-play data of a reasonable length; if the temperature was too low, agents ended up making repeated moves over different games which led

Figure 6. (A) A sample game played by agents with an untrained CNN. Untrained agents make seemingly random moves, and are unable to avoid moves that will immediately result in a loss. (B) After just one iteration of self-play, agents become significantly better at avoiding suicidal moves, but is still generally unable to "attack" the opposing player in a coordinated way. (C) After five iterations of self-play, agents learn how to "grow" their organism, and find vulnerabilities in the opposing player's organism that could be easily attacked.

to insufficient training data. The final values that were used for this hyperparameter was $T = 0.7$ for generating self-play data, and $T = 0.5$ for the actual gameplay, but it is possible that better choices exist.

5. Future Work

Prior to the release of the game, we plan to use this framework to train agents that play games with different versions of the rules. The self-play games generated by the agents will be evaluated using three criteria: (i) whether the player that goes first has an unfair advantage/disadvantage over the other player, (ii) there exists a "perfect" strategy that cannot be beaten under no circumstances, and (iii) the moves made by each agent are dynamic and not unduly repetitive over different games. These three criteria will point us towards a set of rules that will make this game most challenging and enjoyable. Also as a fun but meaningful analysis, we are planning to challenge our agent against a game of *Organism* against the creator of the game himself, Ryan Spangler.

6. Contributions

Ryan Spangler is the developer of the board game *Organism*, and is not a student of CS229. Ryan Spangler and Gwanggyu Sun both contributed to the encoding of the game rules in Python format. Gwanggyu Sun was solely responsible for generating self-play data, implementing MCTS, training the CNN, and writing the final report.

References

[1] E. Hoogeboom, J. W. Peters, T. S. Cohen, and M. Welling. Hexaconv. *arXiv preprint arXiv:1803.02108*, 2018.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[3] 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.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.