AlphaGOLAD Zero: Mastering the Game of Life and Death with Self-Play

Elias Wang [*1] **Hana Lee** [*2] **Zhilin Jiang** [*2]

Abstract

Monte Carlo Tree Search, neural network-based policy and value evaluation, and self-play are three popular techniques widely used in reinforcement learning. Inspired by the recent AlphaGo Zero paper, we design, construct, train, and evaluate an agent to play the Game of Life and Death (GOLAD) using a combination of the aforementioned techniques. GOLAD is a two-player game based on Conway's Game of Life (GOL), where players can manipulate their cells after each simulation step. We obtain positive results on a small board versus a random agent, but challenges remain in transferring our player to larger boards and playing versus more sophisticated opponents.

1. Introduction

We use reinforcement learning (RL) techniques to create an agent that plays the Game of Life and Death (GOLAD). The game is governed by a few relatively simple rules and a single objective: to survive longer than your opponent. Despite this simplicity, it requires significant amounts of strategy due to its large state space (18x16 grid with 3 possible states for each cell) and action space, much like the game of Go.

GOLAD is a two-player game based on Conway's Game of Life (GOL) (Gardner, 1970), a cellular automaton initially proposed by J.H.Conway in 1970. The original GOL is a zero-player game where the transition and results are deterministic based on the initial state and a set of fixed transition rules. GOLAD converts GOL to a two-player game by restricting the game to a fixed-size (18×16) grid space, assigning cell ownerships to players, and allowing players to manipulate their cells after each simulation step.

In GOLAD, each player can take one of three actions on their turn:

[*]Equal contribution [1]Department of Electrical Engineering, [2]Department of Computer Science, Stanford University, Stanford, USA. Correspondence to: Elias Wang <elias.wang@stanford.edu>, Hana Lee <leehana@stanford.edu>, Zhilin Jiang <zjiang23@stanford.edu>.

Stanford CS 234: Reinforcement Learning Winter 2018 Copyright 2018 by the author(s).

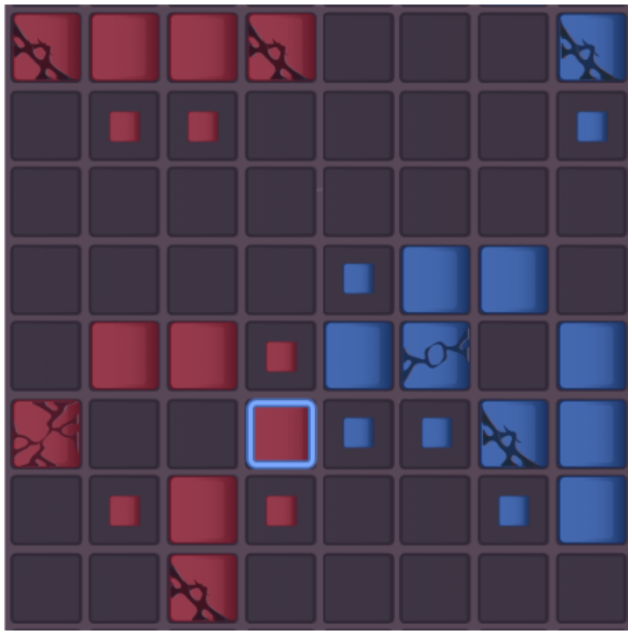


Figure 1. Partial board of an ongoing GOLAD game visualized by *Riddles.io*. Small squares represent cells that will become alive at the next time step and cracked large squares represent cells that will be dead at the next time step. The light blue highlighting represents the location of the current action.

1. Kill any living cell.
2. Birth a new living cell of their own in any empty (dead) cell, by sacrificing two of their own living cells.
3. Pass (do nothing).

After each player's turn, the game engine simulates one step of GOL. Cells with two or three neighbors live on to the next step, while cells with more than three neighbors die from overcrowding, and cells with fewer than two neighbors die from loneliness. Dead cells with exactly three neighbors are reborn into living cells, belonging to the player who owns the majority of the three neighbor cells. A visualization of a partial board is shown in Fig. 1.

The game ends when either of the players loses all its cells, or when the turn limit (100 by default) is reached. If both players lose their last cells at the same time or the turn limit is reached, the game ends as a tie. Otherwise, the player with surviving cells wins the game.

The discrete grid space and the turn-based nature of GOLAD allow us to model it as a Markov Decision Process (MDP). Each episode consists of a full game, terminating when all of a player's cells are eliminated or when the turn limit is reached. While various bots have been submitted to the Riddles.io challenge website, to the best of our knowledge, none have used Monte Carlo Tree Search (MCTS) self-play and neural network policy and value evaluation.

2. Related Work

While there has not been significant published work addressing learning how to play GOLAD, there have been numerous attempts to understand and address the complexity of GOL. For instance, it has been shown that GOL can simulate a universal Turing machine (Rendell, 2011), making predicting future states impossible in general without direct simulation (Alexiou et al., 2018).

In order for an agent to make informed decisions in GOLAD, it needs to accurately predict the value of the current state, which is largely determined by a future state (whose cells died first) of a GOL simulation with non-deterministic future manipulations (opponent's actions), without running full simulations to reach all possible future states. This complexity in GOL makes GOLAD an interesting problem to tackle from a RL perspective.

With the success of Deep Blue (Campbell et al., 2002) in chess, there has been additional interest in solving more complex games. The game of Go, with its astronomical state space, set itself as a major target, and even the "holy grail" to some. Recently, computers were able to defeat the world champion in Go using the innovations of AlphaGo (Silver et al., 2016). After AlphaGo's success, AlphaGo Zero was developed to learn entirely from self-play, removing the need for human input in the form of expert examples (Silver et al., 2017b). While impressive, both of these algorithms took weeks to train on 50 or more GPUs. With the release of Alpha Zero, the models can now be trained within hours or days, depending on the game, but still require thousands of TPUs (Silver et al., 2017a).

In addition to investigating whether AlphaGo Zero's approach can be applied to the more complex and novel GOLAD, we also attempt to address its feasibility with more reasonable computational constraints (a single GPU) and shorter training time (1-3 days).

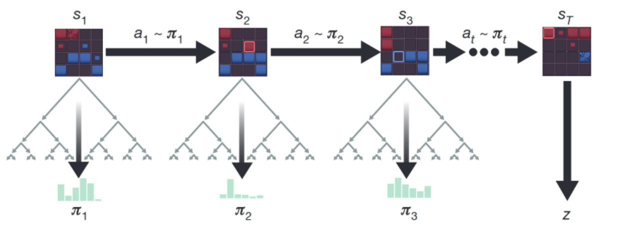


Figure 2. Overview of MCTS self-play. Adapted from (Silver et al., 2017b). At each time step, the agent performs a number of simulated rollouts using MCTS, evaluating each node's value using the neural network. Afterward, the agent samples its next action from a probability distribution.

3. Methods

3.1. Game Engine & Simulation

Although a GOLAD game engine written in Java is available from Riddle.io, we chose to implement our own GOLAD engine in Python for easier interfacing with the self-playing MCTS and neural network implementations. The engine keeps track of current game state, applies changes and simulates GOL for one step after each player's move. With our implementation, the MCTS agent is able to repeatedly perform self-play and simulate the game on its own without the need for the GOLAD engine provided by Riddle.io. A separate bot instance named MCTSBot is implemented for use in performance evaluation, invoking the same MCTS search function but interfacing with the original Riddles.io GOLAD engine instead.

3.2. Algorithm

The recent success of AlphaGo (Silver et al., 2016) in the game of Go provides promising techniques that can be applied to GOLAD. We will also use the concept of self-play, described by Silver et al. (Silver et al., 2017b), to train our agent without the need for "expert" games to learn from.

There are two key parts of the AlphaGoZero self-play algorithm: MCTS and neural network policy and value evaluation. To simplify the implementation, the training procedure is as follows. First, MCTS is run serially to fill a data buffer. Then random batches are sampled from this buffer and fed to the neural network, which is trained for a predefined number of steps. This new network is then used for the next phase of MCTS data generation and the procedure is repeated. Fig. 2 and Fig. 3 illustrate these two components.

3.2.1. Neural Network policy and value evaluation

The neural network takes in a game state representation at each time step and outputs v, a scalar between -1 and

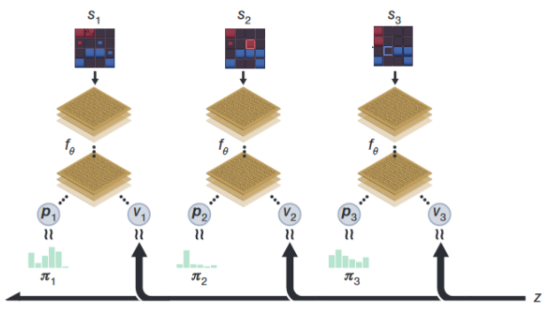


Figure 3. Overview of NN training. Adapted from (Silver et al., 2017b). π represents the "true" probability distribution over possible moves calculated by MCTS, and z represents the winner of the self-play game, encoded as a scalar in {-1,1}. p and v represent the probability distribution and the winner predicted by the neural network, respectively.

1 predicting the value of the current game state (where 1 represents a game victory for Player 0 and -1 represents a game victory for Player 1), and p, a vector representing the probability distribution over all possible moves from the given state. MCTS uses the neural network to evaluate each node in the tree (see next section for more details).

The neural network trains on each batch of self-play iterations in order to minimize the difference between the values it predicts and the values found during self play and MCTS. The loss calculation $L(p, v, \pi, z)$ is given below, where π and z represent the policy found by MCTS and the self-play winner, respectively:

$$L(p, v, \pi, z) = (z - v)^2 - \pi^T \log p$$

MCTS therefore fills the role of the "target network" in DQN, which is updated at intervals (here, after the neural network finishes training on a batch of self-play examples).

We based the network's architecture (visualized in Fig. 4) on the neural network architecture described in the appendices of (Silver et al., 2017b). To summarize, the neural network consists of a residual tower in which each residual block contains two convolutional layers, each followed by a batch normalization and rectified linearity, followed by a skip connection (He et al., 2016). The network is then split into a policy head and a value head. The policy head contains a single convolutional layer followed by a batch normalization and rectified linearity, followed by a fully connected probability output over each cell on the game board plus one (for the 'pass' move). The value head contains a convolutional layer followed by a batch normalization and rectified linearity, followed by two fully connected layers outputting a scalar between -1 and 1. We reduced the number of residual blocks from 19 to 3 to account for a smaller game board size and less compute power.

3.2.2. Monte-Carlo Tree Search Self-play

The MCTS procedure generates data (s_t, π_t, z) using self-play for the supervised neural network training. In a game, at each timestep t, MCTS is run from the current state s_t using the current neural network f_θ to generate output probabilities π_t for each possible move from that state. A move is then sampled from this distribution to be the action taken. We simplify the action space by choosing cells to sacrifice for birth moves randomly. This gives π a dimensionality of $(board\_height \times board\_width + 1)$, by also making use of the observation that each location on the board can only either be a kill or birth move, but not both. The additional dimension corresponds to the pass move.

The MCTS algorithm follows the implementation described in (Silver et al., 2017b) with some minor modifications and additions. Most notably, τ is set at 1 for the whole game, we do not use random symmetries, and we play against the most recent NN instead of using an evaluator. We also add a beam search which reduces the computational complexity. The evaluation of the effect of these simplifications to the final performance of the agent is left for future work.

For completeness, the MCTS will be briefly described here, but we refer the reader to (Silver et al., 2017b) for further details.

Select We define the root node s_0 to be the current state s_t. From the root node we iteratively select a child node until a leaf node s_L is reached using a PUCT algorithm variant (Rosin, 2011). At each step $t < L$, an action is chosen using $a_t = argmax_a(W(s_t, a)/N(s_t, a) + U(s_t, a))$,

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}, \quad (1)$$

where c_{puct} is a constant that determines the level of exploration and set to 1 for our experiments. $P(s, a)$ is the prior probability of selecting the node, $N(s, a)$ is the total visit count, and $W(s, a)$ is the total action value.

Expand and evaluate The leaf node s_L is then evaluated by the current NN f_θ which returns $p, v = f_\theta(s_L)$. If the leaf node is also the root node, it is expanded for all actions, otherwise the only the most probable actions are expanded. This is to ensure sufficient exploration. For each new node, $P(s, a)$ is initialized to p_a, and $N(s, a)$ and $W(s, a)$ are initialized to 0.

Backup We update the statistics of each node by incrementing $N(s, a) = N(s, a) + 1$ and $W(s, a) = W(s, a) + v$.

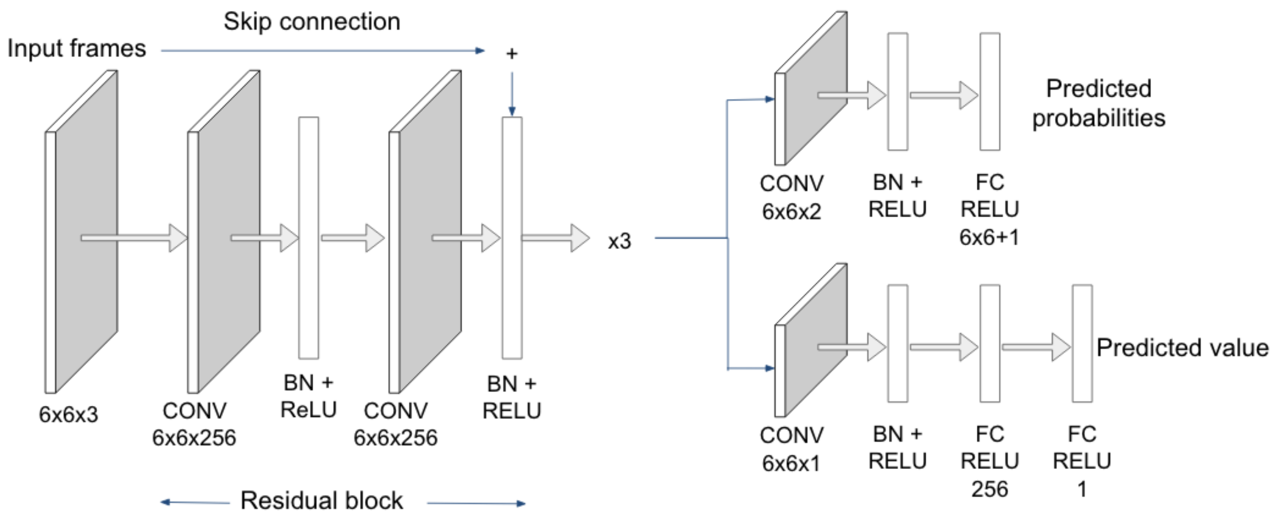


Figure 4. Neural network architecture, consisting of a residual tower (He et al., 2016), a policy head, and a value head. Adapted from (Silver et al., 2017b).

Play After the specified number of max MCTS iterations, an action a is selected to play from the root node s_0, proportional to the exponentiated visit count, $\pi(a|s_0) = \frac{N(s,a)^{\frac{1}{\tau}}}{\sum_b N(s,b)^{\frac{1}{\tau}}}$, where τ controls the level of exploration and is set to 1 for training and an infinitesimal value, $\tau \to 0$, for evaluation. Therefore, the agent select actions proportional to the visit counts when generating data, but greedily during evaluation.

3.2.3. Challenges

Compared to the AlphaGo Zero researchers, we had relatively little computational power at our disposal: a single machine with a single GPU, compared to AlphaGo Zero's 64 GPU workers and 19 CPU parameter servers. AlphaGo Zero also trained for 40 days to achieve its final performance, whereas we were constrained to about 2 weeks of training time - effectively much less, since we restarted several times to retune hyperparameters, fix bugs, and change board size. The makers of Leela Zero, an open source reimplementation of AlphaGo Zero, estimate that recomputing the weights trained by AlphaGo Zero on commodity hardware (similar to what we used for our training) would take 1,700 years.

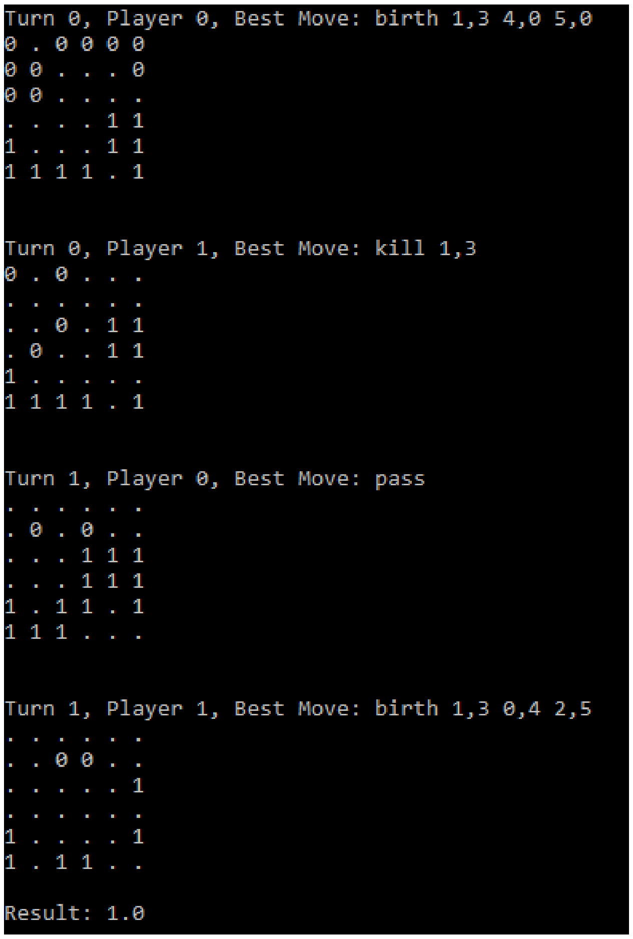
With these obstacles in mind, we decided to make some sacrifices to performance in an effort to reduce the training time required to obtain preliminary results. First, we reduced the board size to 6x6, with the intention of increasing the board size once we were able to obtain good results on the 6x6 board. Next we simplified the action space.

GOLAD's action space is enormous compared to Go's; there are three actions that can be taken, and the birth action involves selecting two living cells to sacrifice. If the MCTS player has 10 living cells, there are 45 possible sacrifice choices for each possible birth position. If the player has 100 living cells, there are 4950 possible choices. Even if we had access to the same compute resources as AlphaGoZero, it would take many times longer to compute each move in a game of GOLAD using MCTS than it would for a game of Go.

We chose to simplify the action space by randomizing the sacrifice choice. Every time the MCTS player takes a birth action, the two required sacrifices are chosen randomly from the player's living cells. The choice of sacrifices is made at the beginning of each turn and held constant for all MCTS simulations. This simplification reduces the action space and makes it possible to run MCTS in a feasible amount of time on a single machine.

4. Experiments & Results

We generate a buffer of data from multiple MCTS self-play games containing 2000 samples; each sample is a single turn (s_t, π_t, z) in GOLAD. We set the number of MCTS iterations per move to 15 and the beam width to 10. We train the NN using a batch size of 16 and learning rate $\alpha = 0.01$, and we implement gradient clipping using a maximum gradient norm of 5. Every 5000 training steps the buffer is updated by discarding 25% of the oldest samples, and replacing it with 500 samples from new games, using the updated neural network. 25% was arbitrarily chosen as a compromise between spending too much time gathering



```
Turn 0, Player 0, Best Move: birth 1,3 4,0 5,0
0 . 0 0 0 0
0 0 . . . 0
0 0 . . . .
. . . . 1 1
1 . . . 1 1
1 1 1 1 . 1


Turn 0, Player 1, Best Move: kill 1,3
0 . 0 . . .
. . . . . .
. . 0 . 1 1
. 0 . . 1 1
1 . . . . .
1 1 1 1 . 1


Turn 1, Player 0, Best Move: pass
. . . . . .
. 0 . 0 . .
. . . 1 1 1
. . . 1 1 1
1 . 1 1 . 1
1 1 1 . . .


Turn 1, Player 1, Best Move: birth 1,3 0,4 2,5
. . . . . .
. . 0 0 . .
. . . . . 1
. . . . . .
1 . . . . 1
1 . 1 1 . .

Result: 1.0
```

Figure 5. Example output from MCTS self-play. Cell coordinates are column-major and using 0-based indexing, following Riddles.io specifications.

new samples and training the network on too much outdated data.

We evaluate our training results by playing our MCTS agent (denoted as MCTSBot) against a baseline random-decision agent (denoted as RandomBot) using the original Riddles.io game engine for 100 independent games. To ensure fairness, MCTSBot and RandomBot take turns to make the first move (i.e. assigned as Player 0), and the randomly generated initial board is guaranteed to be symmetric. Both players begin the game with 10 living cells.

We count each game MCTSBot wins as “1 win”, and each game ending in a draw as “0.5 wins”, to calculate the winning rate as our performance metric:

$$\text{winning rate} = \frac{\text{number of wins}}{\text{number of games played}}$$

On the 6x6 game board, after training for 2.55 million training steps and about 260,000 game turns, the MCTSBot achieves a 70% win rate against RandomBot. For comparison, PassBot, an agent which simply chooses ”pass” on every turn, achieves a 60% win rate against RandomBot. This strongly suggests that passing is a superior strategy to playing random moves, likely because random moves are more likely to result in the death of the passing player’s own cells than their opponents (due to the sacrifice mechanic). MCTSBot only achieves a 50% win rate against PassBot, suggesting that MCTSBot is at least able to learn an active strategy that matches the performance of passing on every turn, but is unable to surpass it on the limited board size.

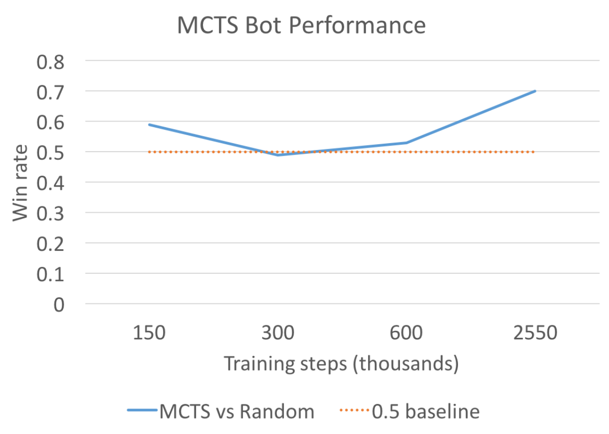
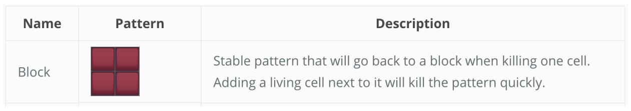


Figure 6. Results throughout training on the 6x6 board.

We hypothesized that MCTSBot’s relatively poor performance was partially due to the constraints of the 6x6 board. Good strategies in GOLAD rely on setting up stable cell structures and working to disrupt the opponent’s structures; however, on such a small board, setting up a stable structure is a difficult feat. During training, we observed that the only stable structure to appear with any regularity was the 2x2 block structure, which can survive the death of any one cell in the structure.



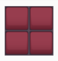
Name	Pattern	Description
Block		Stable pattern that will go back to a block when killing one cell. Adding a living cell next to it will kill the pattern quickly.

Figure 7. Block structure, a stable structure learned by MCTSBot on the 6x6 board.

We tested our hypothesis by training on an 8x8 game board for 2.9 million training steps. However, throughout training, MCTSBot was unable to perform better than RandomBot; its highest win rate was 46%.

We speculate that good strategies for the 8x8 board may be even harder to identify compared to the 6x6 board, and that MCTSBot may need to train several times longer on the 8x8 board due to the larger action and state space, or run more MCTS simulations for each step. Unfortunately, training on the 8x8 board already takes roughly twice as long for each step as training on the 6x6 board, and increasing the number of simulations has a detrimental effect on the runtime of MCTS. Training longer on the 8x8 board or increasing the board size to further test our small board hypothesis is therefore out of scope for this quarter.

5. Conclusion

We demonstrate the potential of the MCTS self-play framework for solving a novel complex game, GOLAD. While the MCTS self-play framework is able to achieve fairly good performance against a random agent on a 6x6 board, its performance on an 8x8 board is worse than random given limited training time and resources.

The following future work remains:

- Transfer results from the 6x6 board to a larger board (such as 18x16) using transfer learning.
- Use MCTS with neural network evaluation for training, and use neural network evaluation only for playing evaluation games to reduce execution time.
- Compare performance of agent using MCTS with neural network evaluation to an agent using only neural network evaluation.
- Encode each cell's future transition in the neural network input to speed up training. The large changes that the board undergoes between each time step are difficult for the neural network to learn, and encoding each cell's next state (dead, alive, or reborn) in the input could reduce the time required to learn the simulation patterns.
- Train for more iterations on larger batches from self-play, and tune hyperparameters such as the learning rate α, layer and filter sizes for the residual blocks, number of residual blocks, and regularization coefficient λ.
- Optimize MCTS so that more simulations can be performed at each time step with less impact to training time.

Acknowledgements

We would like to thank the CS 234 course staff for their excellent instruction and advice.

References

Alexiou, Katerina, Johnson, Jeffrey, and Zamenopoulos, Theodore. Embracing complexity in design. pp. 196, 03 2018.

Campbell, Murray, Hoane, A.Joseph, and hsiung Hsu, Feng. Deep blue. *Artificial Intelligence*, 134(1):57 – 83, 2002. ISSN 0004-3702.

Gardner, Martin. The fantastic combinations of john conways new solitaire game ”life”. *Scientific American*, 223: 120–123, 1970.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Rendell, Paul. A universal turing machine in conway's game of life. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pp. 764–772. IEEE, 2011.

Rosin, Christopher D. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, Mar 2011. ISSN 1573-7470. doi: 10.1007/s10472-011-9258-6.

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

Silver, David, Hubert, Thomas, Schrittwieser, Julian, Antonoglou, Ioannis, Lai, Matthew, Guez, Arthur, Lanctot, Marc, Sifre, Laurent, Kumaran, Dharshan, Graepel, Thore, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 12 2017a.

Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, Chen, Yutian, Lillicrap, Timothy, Hui, Fan, Sifre, Laurent, van den Driessche, George, Graepel, Thore, and Hassabis, Demis. Mastering the game of go without human knowledge. 550:354–359, 10 2017b.

A. Contributions

The contributions of each team member are summarized in this section.

Elias Wang

MCTS self-play code; training code for NN with MCTS data; training of MCTS player.

Hana Lee

Neural network code; training and evaluation of MCTS player on 6x6 and 8x8 game boards.

Zhilin Jiang

GOLAD engine re-implementation; RandomBot and Pass-Bot; MCTSBot interfacing with Self-Play and with the original Riddles.io GOLAD engine; performance evaluation automation script.

B. Code

The code for our project is available on Github.