AlphaGoStop: Applying Deep Reinforcement Learning to Go-Stop

20170058 Keonwoo Kim
keonwoo@kaist.ac.kr / https://github.com/cs470-2020f-team36

Abstract

In this report, we apply the deep reinforcement learning architecture used in DeepMind's AlphaZero, which was a breakthrough in the field of artificial intelligence for Go, Chess, and Shogi, to Go-Stop, a famous card game in South Korea. Existent artificial intelligences for Go-Stop is mostly written using hard-coded logic, which is complicated to cope with every different case, so using a neural network instead is worthwhile. However, different from the games that AlphaZero dealt, Go-Stop is a game with hidden information, which means that each player might not see the entire details of the state of a game. We replace the ResNet part in the architecture of AlphaZero by fully connected layers To overcome this issue, we use PIMC as a workaround for hidden states. We also discuss about limitations of this approach and propose would-be better solutions to this problem.

1. Introduction

Go-Stop is a famous card game in South Korea, which is turn-based and playing with a deck of flower playing cards *(hanafuda.)* The deck consists of 12 sets of 4 cards, each assigned with a month, and additional two bonus cards. Basically, a player starts the turn by throwing a card from their hand and ends the turn by flipping the top card of the drawing pile. When there are matching cards on the field at the center, then the player captures those cards. When a player gets an enough amount of card so that the score of the player exceeds a deterministic number, the player can decide whether the game keeps going (*Go*,) or stops with gaining scores (*Stop.*) While there is a variant of it played by more than 2 players, we will focus on Go-Stop played by 2 people.

Go-Stop is a game of hidden information, meaning that each player cannot access to the entire game state. For instance, each player cannot see the opponent's hand (except some cards to be public), and the list of cards in the drawing pile.

Note that it is impossible to search every possible move, because there are about 20 turns in a game, and each turn has about 4–5 legal actions on average. Moreover, for a Go-Stop game instance of hidden information, the number of games of public information possible from the given game instance is huge to search all of them. Therefore, playing Go-Stop is not a trivial task, and we need an efficient way to search a good move given a game instance.

We employed DeepMind's Alpha(Go) Zero [13] to search the game tree efficiently in this project. And in order to deal with hidden information in Go-Stop, we used PIMC approach. For more information, refer to Section 2.

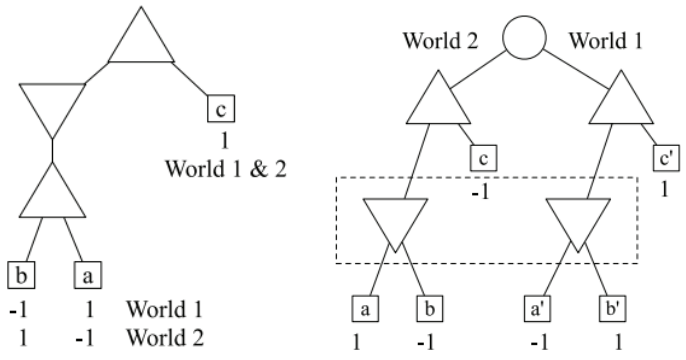
The demonstrative implementation is publicly available at `https://gostop.kanu.kim/`. The frontend is written with React and served with gh-pages, and the backend is written in Python with Flask and Flask-SocketIO and served on a Heroku instance.

- `https://github.com/cs470-2020f-team36/go-stop-python`: repository for the game logic, training process, and the backend
- `https://github.com/cs470-2020f-team36/go-stop-visualizer`: repository for the frontend
- `https://github.com/cs470-2020f-team36/project-proposal`: repository for the project proposal
- `https://github.com/cs470-2020f-team36/presentation`: repository for the final presentation
- `https://github.com/cs470-2020f-team36/report`: repository for the final report

2. Background

2.1. MCTS and PUCT

MCTS (Monte–Carlo Tree Search) is a best-first tree search algorithm consisting of the iterative process of the following three steps: 1) to *select* a leaf node according to a specified selection algorithm, 2) to *evaluate* the leaf node, and 3) to *update* the tree accordingly with the evaluation result. Then MCTS averages the evaluation results in the subtree rooted at state s to obtain the expected value $V(s)$ of the state. By means of evaluation, Alpha(Go) Zero and



(a) Strategy fusion (b) Non-locality

Figure 1: Diagrams describing two issues of PIMC approach: strategy fusion and non-locality [10].

this project use neural networks as state value approximators instead of rollout—a totally random search on the tree. Most selection algorithms are designed to minimise the expected regret of sampling the state s by balancing exploration and exploitation. Examples of selection algorithms include UCB1 [9], PUCT [12], and a variant of PUCT used by AlphaGo, as shown below:

$$a_t = \arg\max_a \left(Q(s,a) + U(s,a)\right),$$

$$U(s,a) = c_{\text{PUCT}}\, P(a \mid s)\, \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}.$$

Here $Q(s,a)$ is the value of taking the action a at the state s, and $U(s,a)$ is a bonus value to regulate exploration, given by the variant of PUCT. $P(a \mid s)$ is the prior probability of taking the action a at the state s, and $N(s,a)$ is the simulation counts of taking a at s. c_{PUCT} is a constant multiplicative factor to be chosen manually.

The resulting probability distribution $P(a \mid s)$ is proportional to $N(s,a)^{1/\tau}$, where τ is a hyperparameter deciding how deterministic the choice would be. It is because as $\tau \searrow 0$, $N(s,a)^{1/\tau}$ tends to 0 if $N(s,a) < \max_b N(s,b)$.

2.2. PV-MCTS

Policy Value MCTS (PV-MCTS) uses neural networks to provide a policy $P(a \mid s)$ and a value $V(s)$ for given state s. The policy and the value provided by the neural networks are used to estimate the policy and the value of the unvisited leaf node during each step of MCTS. While AlphaGo used two separate networks to approximate P and V, AlphaGo Zero used one shared neural network that outputs the prior probabilities and the state value simultaneously.

Experiments from [13] show that the output policy of PV-MCTS is about 2000 Elo ratings stronger than simply using policy value neural networks to learn the strategy without MCTS.

2.3. Imperfect Information

Go-Stop is a game of imperfect (incomplete, or hidden) information. In this section, we introduce the notations used within the report. Let G denotes the set of full states of all possible (Go-Stop) games, and $i \in \{0,1\}$ be the (index of) player. Now, Let H_i denote the set of all hidden states for the player i, and $f_i \colon G \to H_i$ be the *observation function* of the player i, i.e., $f_i(g)$ is parts of the state of the game g which observable to the player i. Note that f_i is surjective, $H_i = f_i(G)$. But usually f_i is not injective, and $f_i^{-1}(f_i(g)) = \{\tilde{g} \in G \colon f_i(\tilde{g}) = f_i(g)\}$ denotes the set of game states which are seemingly identical to the player i.

2.4. PIMC

One popular way of dealing with imperfect information is a method called *Perfect Information Monte–Carlo* (PIMC), also known as *determinization.* With this approach, given a game state with hidden information, a game state of perfect information is sampled in which the current state is chosen from the agent's current information set. For instance, in the game of Rock-Paper-Scissors, the decision of 'me' is visible but the decision of opponent is hidden. So PIMC chooses one deterministic game state, where the decision of 'me' is the same with the hidden one and the decision of the opponent is one of rock, paper, and scissors. PIMC has shown expert-level performances on games like Bridge [8] and Skat [2], and it has produced strong play on Hearts [14].

In 1998, Frank and Basin published an extensive critique of PIMC approach to imperfect information games [7]. They suggest two problems residing in the nature of PIMC: strategy fusion and non-locality. In Fig. 1, an upward pointing triangle represents a player (*player I*) who wants to maximize the payoff, and a downward pointing triangle represents a player (*player II*) who wants to minimize

the payoff.

In Fig. 1a, at the first step, the player I has an option to select the right edge, which is always better than selecting the left node. However, by a determinization, even if the player I selects the left edge at the first step, the player I can obtain the payoff 1 no matter which action the player II did, so the player I will be confused between the left edge and the right edge at the first choice. This problem is called *strategy fusion.*

The second error identified is termed *non-locality,* and it is a result of the fact that the value of a node may depend on other regions of the game tree not contained within its subtree, primarily due to the opponent's ability to direct the play towards regions of the tree that they know are favorable for them, using private information. In the game represented by Fig. 1b, the best action of player II will be selecting an edge randomly. But, assuming the player I is reasonable, the player II can always know the correct move, because the player I would take the right edge to win the payoff 1 if the player I were in World 1. This means that if there is a chance to play an action for the player II, the game is in World 2, so the player II will take the right edge to take the payoff -1.

Despite those critiques of PIMC, in practice it has often produced strong results in a variety of domains, as indicated by [10].

3. Methods

3.1. Self-Play Training Pipeline

We basically followed the steps of PV-MCTS as described in Section 2, but there are miscellaneous differences. First, execute Algorithm 1 $n_{\text{epi/evol}}$ times to gather the training examples. After every call, append the result of it to the replay buffer, and take its recent $n_{\text{rep buf}}$ examples. Split them into minibatches of size n_{mb} and train the neural network with those minibatches.

$\mathcal{S}_g$ is a sample from $f_i^{-1}(f_i(g))$, so PIMC approach is used here. When π_{noise} is assigned, $\bar{e}[1]$ represents the average policy produced by MCTS in the game instances in $\mathcal{S}_g$. By adding a Dirichlet noise E, we get an π_{noise}, the average policy with a Dirichlet noise. Finally, when an example is appended to $\tilde{\ell}$, we put $w_{\text{reward}}\, v + (1 - w_{\text{reward}})\, \bar{q}$ as a value for the example. $\bar{q}$ is the average of expected rewards of the games in $\mathcal{S}_g$ from the previous neural network, and v is the expected reward from the MCTS simulation. By considering $\bar{q}$ together, we may reduce the bias from the determinization.

3.2. Reward Function

After a game is terminated, the agent receives a reward measuring how good the agent is doing. In this project, we modeled the reward function considering both scores and whether the agent won:

$$v(g) = (1 \textbf{ if the agent won else } -1) \cdot (w_{\text{win}} \cdot 40 + (1 - w_{\text{win}})\, g.\texttt{winners\_score})$$

For instance, if w_{win} is set to 0.5 and the score for the agent was -16, then the reward will be $-(0.5 \cdot 40 + 0.5 \cdot 16) = -28$.

Note that for the evaluation, we chose $w_{\text{win}} = 0$, that is, $v(g)$ is the raw score from the game.

Function `execute_episode(`*NN*`):`
- $\ell \longleftarrow []$; $\tilde{\ell} \longleftarrow []$; $g \longleftarrow$ `Game()`;
- **while** ***not*** g.*ended*() **do**
  - $e \longleftarrow []$;
  - $i \longleftarrow g$.`current_player`;
  - $n_{\text{hand}} \longleftarrow |g.\texttt{hand}(0)| + |g.\texttt{hand}(1)|$;
  - $\tau \longleftarrow 1$ **if** $n_{\text{hand}} < n_\tau$ **else** $\tau_{\ll 1}$;
  - $\mathcal{S}_g \longleftarrow$ sample $n_{\text{sim games}}(n_{\text{hand}})$ games from $f_i^{-1}(f_i(g))$ uniformly at random;
  - **foreach** $\tilde{g} \in \mathcal{S}_g$ **do**
    - Call `MCTS`$(g, \texttt{NN})$ $n_{\text{MCTS/simul}}$ times;
    - $a \longleftarrow$ sample an action according to the distribution $\pi(g, \cdot; \tau_{\ll 1})$;
    - e.`append`$((f_i(g), \pi(\tilde{g}, \cdot; \tau), Q(\tilde{g}, a), i))$;
  - **end**
  - $\bar{e} \longleftarrow e$.`mean(axis` $= 0)$;
  - ℓ.`append`$(\bar{e})$;
  - $E \sim \text{Dir}(\alpha)$; $\quad \pi_{\text{noise}} \longleftarrow (1 - \epsilon) \cdot \bar{e}[1] + \epsilon \cdot E$;
  - $a_{\text{next}} \longleftarrow$ sample an action according to the distribution π_{noise};
  - $g \longleftarrow g$.`next_state`(a_{next});
- **end**
- **foreach** $(h_i, \pi, \bar{q}, i)$ ***in*** ℓ **do**
  - $v \longleftarrow$ `reward`(g, i);
  - $\tilde{\ell}$.`append`$((h_i, \pi, w_{\text{reward}}\, v + (1 - w_{\text{reward}})\, \bar{q}))$;
- **end**
- **return** $\tilde{\ell}$

Algorithm 1: `execute_episode`

3.3. Network Architecture

While AlphaZero uses a ResNet as a base network architecture, we use a series of fully connected layers as shared network components. Go-Stop board has no geometric properties in it, as opposed to Go, Chess, or Shogi, which AlphaZero deals with. In such games, game pieces placed near to each other and far to each other should be treated differently due to the nature of those games. However, in Go-Stop, everything is either a number or a set of cards having no specific orders. Therefore, it makes no sense to use convolutional layers, where the weights are nonzero only if the row index and the column index are close enough, when we see the convolution operator as a matrix. So we use fully connected layers. The entire network layers are shown below.

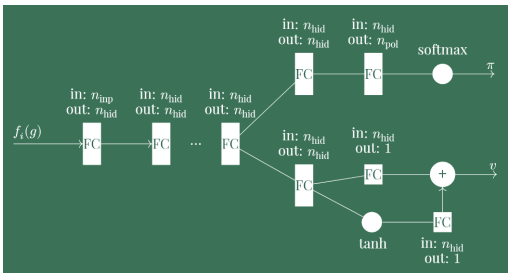


Figure 2: Network architecture used in the project.

3.4. Optimizer

The chosen optimizer is a stochastic gradient descent optimizer, with the initial learning rate 10^{-3} and the exponential decrease rate 0.96. The loss function is as follows:

$$L(\hat{p}, \hat{v}; p, v) = L_{\text{policy}}(\hat{p}; p) + L_{\text{value}}(\hat{v}; v),$$
$$L_{\text{policy}}(\hat{p}; p) = -10 \sum_a p(a \mid s) \log(\hat{p}(a \mid s)),$$
$$L_{\text{value}}(\hat{v}; v) = (v - \hat{v})^2.$$

Note that the constant 10 is multiplied in order to equalize the (empirical) magnitudes of two losses.

3.5. Evaluation

For both self-play and a match with a random agent, to reduce the fluctuation of results, we use a fixed set of games, in the file `games.pickle`. However, the fluctuation still exists due to the randomness of the random agent. Further, to limit the effects of luck, for each game, the same game is played just with switching the starting player (refer to `match_agents` method.)

3.6. Implementation Details

3.6.1 Encoding Game State

The encoded game has the following information: my hand, opponent's hand (visible cards only), my and opponent's capture fields, my and opponent's 'Go' histories (records the scores of a player when they declared 'Go,') the numbers of shakings for me and opponent, my and opponent's stacking histories (who had shaken cards of which month,) my and opponent's scores.

A set of card is encoded to a sum of one-hot vectors representing each element card. For a 'Go' history, it is just an array (of maximum length 9) of scores. A stacking history is a sum of one-hot vectors representing months.

3.6.2 Hyperparameter Choices

The batch size B is chosen to be 256, which is believed to be in a standard range.

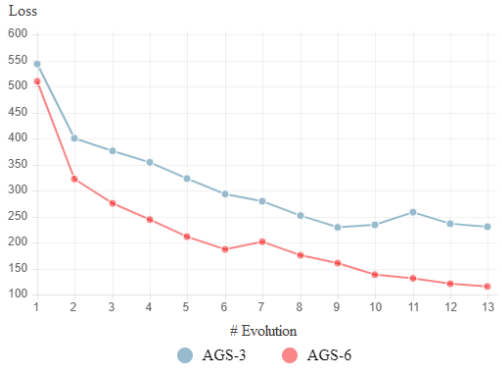


Figure 3: The loss plots for two networks, AGS-3 and AGS-6.

$c_{\text{PUCT}} = 4$, $\epsilon = 0.25$, $\alpha = 1$, $n_\tau = 20$ are chosen as suggested in [6].

$n_{\text{sim games}}(t) = \max(\lfloor 2^{(34-t)/7} \rfloor, 3)$, $n_{\text{MCTS/simul}} = 50$, $n_{epi/evol} = 35$, $w_{\text{reward}} = 0.7$ are chosen without any prior knowledge. In particular, first two values are chosen to avoid too-long training process.

As mentioned above, the neural network is trained using $n_{\text{rep buf}}$ most recent examples from the replay buffer. We decided $n_{\text{rep buf}} = 500 + 400c$, where c is the generation count of the network. It is beneficial to phase out early training examples as these contain less valuable information, according to [6] and [3].

4. Results

4.1. Quantitative Results

We tested two network structures, AGS-3 and AGS-6, where the numbers of shared hidden fully connected layers are 3 and 6, *resp.* The loss plots (Fig. 3) are included.

After the training until 13-th evolutions, AGS-3 showed a performance of gaining $\approx$ 2 points on average in the matches with a random agent. And with AGS-6 the average score is ≥ 10 most of the time in the matches with a random agent. To reproduce it, run `test_agents.py` at the root of the backend repository.

We also tried to get Elo ratings of the agents, but the results are too unstable to draw any meaningful conclusion, due to the randomness of the evaluation method.

4.2. Qualitative Drawbacks

Both agents seem to be declare 'Stop' even when it is fairly able to declare 'Go,' as in Fig. 4. It seems that the MCTS did not provide an enough amount of training examples where it is worthwhile to declare 'Go' under that situation.

Furthermore, sometimes the agent tries to discard a card even when there are plenty of cards that can be matched,

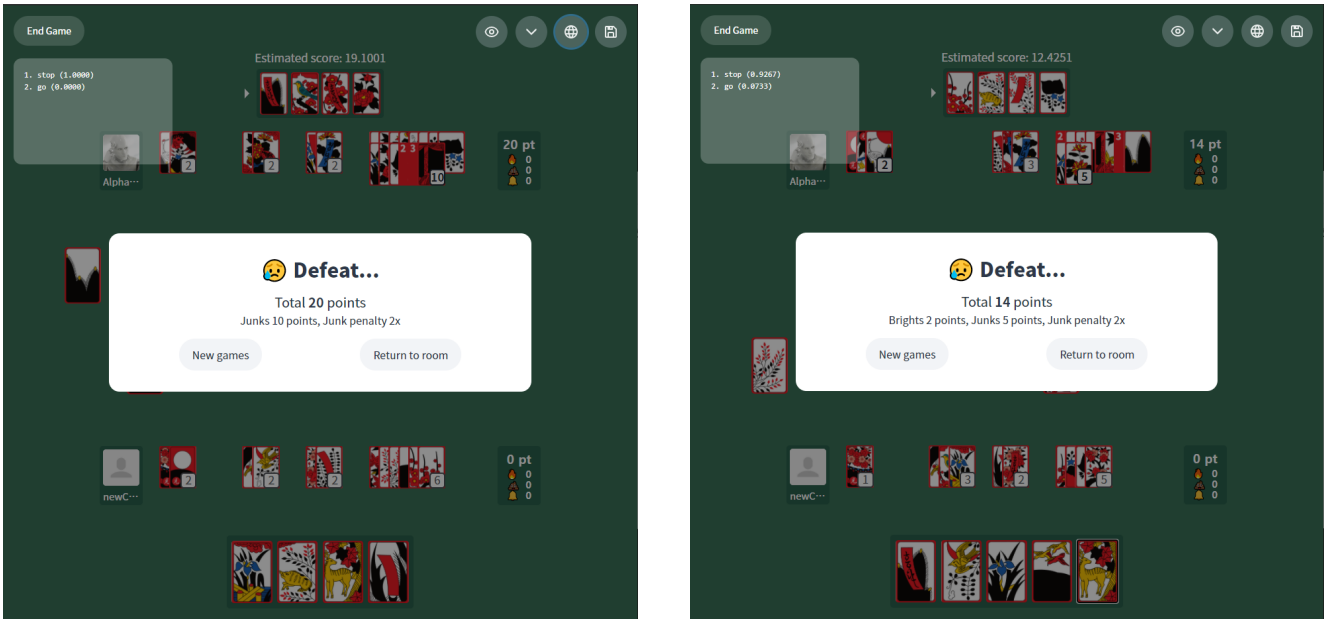


Figure 4: The first qualitative drawback. The agent declares 'Stop' most of the time. 'Go' makes the score multiplied by 2 (when the player had declared 3 or more Go's,) so, declaring 'Go' might be better on some occasions.

as in Fig. 5. We guess it is because the agent is aiming for *discard-and-match.*

As predicted in Section 2, it seems that both agents also suffer from strategy fusion and non-locality.

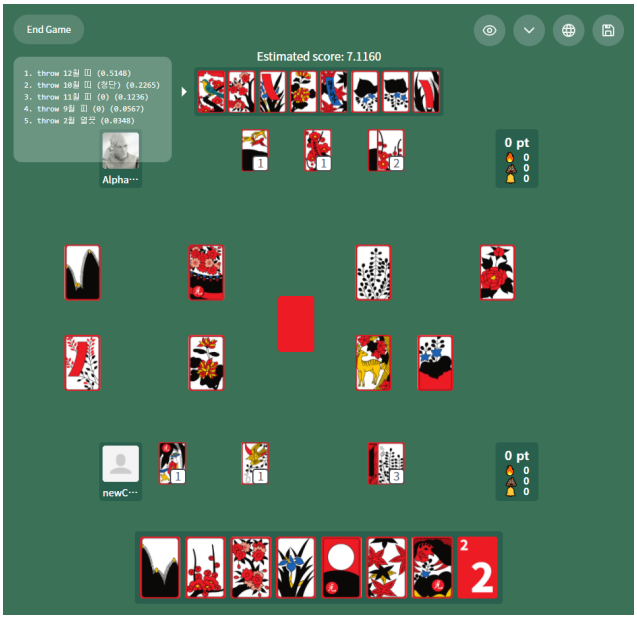


Figure 5: The second qualitative drawback. The agent is about to throw a card of December, while there are several cards that can be captured by it, such as the double junk of November, the bright of March, or a junk of September.

5. Limitations and Future Works

From the result of Section 4, we observed that Go-Stop is quite vulnerable to the negative effects of determinization. This would be because the scores in Go-Stop is changing very rapidly from some stages, and every single action is important where the luck is involved more significantly in it.

Also, we introduced some heuristics (such as considering average of policies of games in $\mathcal{S}_g$ from the neural network) when we implement PIMC with multiple game instances into Go-Stop, as described in Algorithm 1. It may cause another issue related to the nature of those heuristics, which are not investigated in this project yet.

To resolve those issues, we think the major solution to those is changing the hidden information mechanism from PIMC to another methods, such as ISMCTS [5], $\alpha\mu$ search [4], or deep counterfactual value networks [11] to avoid strategy fusion and non-locality problems.

Also, as the scores in Go-Stop *explode* after '3 Go,' so it would be better if the reward function is being concave on the positive x axis (and *vice versa*), to penalize exponentially growing scores.

Recently, ReBeL [1] showed a good performance on the game of Poker, and ReBeL is proved to converges to a Nash equilibrium in *any* two players game. If we employ this method, we might obtain a better result.

6. Conclusion

AlphaZero is a deep reinforcement learning model aiming to solve Go, Chess, or Shogi, which are games of perfect information and having similar geometric structures. By applying the general strategy of AlphaZero to Go-Stop, we obtained an agent getting ≥ 10-points versus a random agent. There are some drawbacks and inconsistencies found from the agent, as predicted by [7]. So, we can conclude that non-randomness and the publicity of information of the game is important to the soundness and convergence of PV-MCTS process used in AlphaZero. To make the agent better, we proposed to use other methods than the naive MCTS to deal with hidden information and to employ other modifications, as written in Section 5.

Contributions

The project proposal is written by Keonwoo Kim, Yongwook Lee, and Junwon Jo. Everything else is implemented by Keonwoo Kim.

References

[1] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *arXiv:2007.13544*, 2020.

[2] Michael Buro, Jeffrey Long, Timothy Furtak, and Nathan Sturtevant. Improving state evaluation, inference, and search in trick-based card games. pages 1407–1413, 01 2009.

[3] Fredrik Carlsson and Joey Öhman. *AlphaZero to Alpha Hero: A pre-study on Additional Tree Sampling within Self-Play Reinforcement Learning*, 2018 (visited December 13, 2020).

[4] Tristan Cazenave and Véronique Ventos. The $\alpha\mu$ search algorithm for the game of bridge. 2019.

[5] P. I. Cowling, E. J. Powley, and D. Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

[6] Oracle Developers. *Lessons From Implementing AlphaZero*, 2018 (visited December 13, 2020).

[7] Ian Frank and David Basin. Search in games with incomplete information: a case study using bridge card play. *Artificial Intelligence*, 100(1):87 – 123, 1998.

[8] M. L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 14:303–358, Jun 2001.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. page 282–293, 2006.

[10] Jeffrey Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. page 134–140, 2010.

[11] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, Mar 2017.

[12] Christopher Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 09 2010.

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

[14] Nathan R. Sturtevant. An analysis of uct in multi-player games. In *Proceedings of the 6th International Conference on Computers and Games*, CG '08, page 37–49, Berlin, Heidelberg, 2008. Springer-Verlag.