How to play Notakto: Can reinforcement learning achieve optimal play on combinatorial games?

Zhenhua Chen, Chuhua Wang, Parth Laturia, David Crandall, Saúl Blanco
School of Informatics, Computing, and Engineering Indiana University, Bloomington, IN
chen478@iu.edu, cw234@iu.edu, platuria@iu.edu, djcran@indiana.edu, sblancor@indiana.edu

Abstract

Reinforcement learning has achieved great success in learning strategies for games with minimal supervision. Here we consider a specific simple combinatorial game called notakto, in which players cover squares on a tic-tac-toe board until one of them loses by finishing a whole row, column, or diagonal. We introduce optimal strategies for this game played on small boards, but larger boards have proven difficult to analyze mathematically. We set out to use reinforcement learning to help give insight into optimal strategies for larger boards. Surprisingly, we found that, despite the simplicity of the game, AlphaGo Zero struggled to learn an optimal strategy, even though we know that they exist. We developed two ways to accelerate AlphaGo Zero's learning on this problem: one is a targeted sampling strategy that biases towards states that are likely to appear in competent play, and the other is to set the threshold for updating a model based on statistics in the training data. Both techniques make AlphaGo Zero converge faster on notakto, but they are specific to this game and still do not seem to find a near-optimal strategy. We conjecture that this difficulty may show some fundamental limitations of reinforcement learning on combinatorial games that need to be further investigated.

Introduction

Skill in adversarial games has long been considered a measure of intelligence in both humans and machines, and much work in AI has focused on popular human games like Chess, Checkers, and Go. While early techniques relied on fast search, look-up tables, and hand-crafted heuristics (Breuker, Uiterwijk, and van den Herik 2000; Kawano 1996; Schaeffer et al. 2005, 2007; Yoshizoe, Kishimoto, and Müller 2007), more recent approaches learn automatically. A major achievement of this line of work is AlphaGo Zero (Silver et al. 2017a,b), which showed that a common framework based on artificial neural networks can learn to master these and other games without human knowledge or intervention.

But while Chess and Go are popular among both human players and AI researchers, their complicated rules make it difficult to understand what the AI systems are actually learning. Games with simpler rules that are easier to analyze theoretically and analytically can nevertheless be quite challenging. Investigating how well (or poorly) modern algorithms learn these games could yield insight into the strengths and weaknesses of modern game-playing AI approaches.

For example, consider the simple childhood game of tic-tac-toe, in which players alternately place pieces on a 3×3 grid until a row, column, or diagonal is complete. This simple game can be made more difficult through three simple modifications: (1) allow an arbitrarily-sized $n \times n$ board, (2) give both players the same symbol (say, "X"), and (3) invert the winning condition so that the first player to complete a row, column, or main diagonal *loses*. Combinatorial game theorists would call this the *misère* version of one-symbol tic-tac-toe; we call it *notakto* for short.

Assuming perfect play, which player wins notakto for different values of n? It is obvious that the second player always wins with $n = 1$, while the first player always wins for $n = 2$. The first player can always force a win for $n = 3$, by placing their first piece in the middle of the board. It was known that the second player wins for $n = 4$ and the first player wins for $n = 5$ (Chow 2010), but solving the game beyond this point is difficult: the large branching factor makes it impractical to determine a winner through brute force search for any but the smallest of boards.

Meanwhile, analytic solutions help but not as much as we might hope. Algebraic analysis for this game has been carried out for the 3×3 board using the so-called *misère quotient* (Plambeck and Whitehead 2014). Unfortunately, while the misère quotient has been proven to be useful in solving several other misère games (Plambeck and Siegel 2008), for notakto it is not known if the misère quotient of even a 4×4 board is finite (Plambeck and Whitehead 2014), which indicates that this simple game is hard to fully analyze. Misère variants of combinatorial games, like notakto, are often harder than their normal play counterparts (see (Siegel 2013, Chapter V) and (Plambeck and Siegel 2008)).

In this paper, we report preliminary results towards our goal of understanding notakto's properties using reinforcement learning, and make several steps towards advancing what is known. We first attack the problem analytically, and prove that the second player always wins for game boards

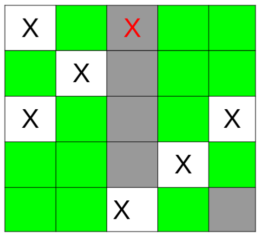


Figure 1: AlphaGoZero's undesirable option for the first player, "X". "X"s are current pieces on the 5 by 5 board. Green areas are the options that can force a win for the first player while the grey areas are not.

of size $n \times n$ for any n that is a multiple of 4. Unfortunately, this analysis does not yield insight into other values of n. We then try a traditional search-based technique with hand-crafted optimizations to deal with the huge branching factor, letting us solve the game for $n = 6$, for the first time. However, $n > 7$ remains well beyond our computational resources.

We thus tried a third approach: applying reinforcement learning (in the form of AlphaGo Zero) to learn to play this game automatically. Our hope was that if AlphaGo Zero could learn to play notakto well, especially for board sizes of $0n > 7$, we could use the patterns of its play to derive a winning strategy, and then prove (either analytically or through more targeted search) that this strategy was optimal. Unfortunately, and surprisingly, we find that AlphaGo Zero struggles to learn notakto, even for modest board sizes, despite its simple rules. One example is shown in Figure 1, a 5 by 5 board is occupied by six "X"s, the first player can force a win (there exists a winning route no matter what the second player plays) if it plays at any of the green area. On the contrary, the first player cannot force a win in any of the grey areas. We trained an AlphaGo Zero model that fails to play optimally, as "X" shows. We hypothesize that this is because the structure of notakto game play can cause the learning algorithm to spend much time considering highly improbable board states. We present a technique to help mitigate this property, and present experimental results that suggest a solution for $n = 7, 8$. However, the techniques are unsatisfyingly very specific to notakto. We hope the paper inspires interest in notakto and other simple games that seem difficult for reinforcement learning, which may help identify limitations and possible directions for future work.

Who wins notakto?

Despite the simplicity of notakto's rules, only limited results are known about optimal strategies for winning. The game is trivially solvable for $n = 1$ and $n = 2$, resulting in a second player and first player win, respectively. For $n = 3$, it is easy to verify by hand that the winner is the first player if she places her first "X" in the center square; if she chooses *any* other square as her first move, the second player can force a win (Chow 2010). Because the number of board states grows exponentially with n, it becomes very difficult to analyze larger boards by hand or through intuition.

Insight from analysis

For $4k \times 4k$ boards, with $k \in \mathbf{Z}^+$, it was known that the second player could always win (Chow 2010), though no formal proof was given. We take the opportunity to introduce both a formal proof and a winning strategy here. We use the following notation. If $\mathbf{Z}^+$ denotes the set of positive integers and $n \in \mathbf{Z}^+$, then we define $[n] := \{1, 2, \ldots, n\}$.

Theorem 1. *The second player has a winning strategy for notakto on any board of size $n \times n$, where $n = 4k$ and $k \in \mathbf{Z}^+$.*

Proof. If we label the tiles in the $n \times n$ board in a coordinate system where $(1,1)$ is the top-left corner and (n,n) is the bottom-right, then the second player is guaranteed to win using the following strategy described by the function $f : [n]^2 \to [n]^2$ given by

$$(x,y) \overset{f}{\mapsto} ((n+1) - x + (-1)^x, y + (-1)^{y+1}).$$

It is easy to see that f is its own inverse function; that is, $(f \circ f)(x,y) = (x,y)$, where $\circ$ denotes the composition of two functions. Furthermore, row i is mapped to row $(n + 1) - i + (-1)^i$, column j is mapped to column $j + (-1)^{j+1}$, and the main diagonal $\{(i,i) \mid i \in [n]\}$ is mapped to the main anti-diagonal $\{(x,y) \mid x,y \in [n] \text{ and } x + y = n + 1\}$. Furthermore, since $n = 4k$ with $k \in \mathbf{Z}^+$, it holds that $i \neq (n+1) - i + (-1)^i$, and so each row is mapped to a *different* row by f. Indeed, there are two possibilities: (i) if i is even, then $i = (n+1) - i + (-1)^i$ would imply $i = 2k + 1$, which is impossible, or (ii) if i is odd then $i = (n+1) - i + (-1)^i$ would imply $i = 2k$, which is impossible. Furthermore, since $y \neq y + (-1)^{y+1}$, rows get mapped to different rows as well. Similarly, the main diagonal and main anti-diagonal are swapped under the action of f. Therefore, if the second player completes a row, column, or diagonal by following the pairing of cells in the $n \times n$ grid given by f, then the first player must have completed a row, a column, or a diagonal earlier in the game. Hence, the second player is guaranteed to win. □

We illustrate the strategy and the corresponding map f in Table 1. For example, the square labeled "11" in position $(2,3)$ in Table 1 is mapped to the position indexed as 11 which is position $(8,4)$ (using row-major indexing starting with 1 at the upper left).

Interestingly, but unfortunately, this pairing strategy does not seem to exist for boards where n is not a multiple of four — even for other even numbers (i.e., for $n = 4k + 2$ and $k \in \mathbf{Z}^+$). However, the non-existence of such a strategy does *not* prove that the second player will lose (although, as we discuss below, we do know that the second player loses for $n = 6$ assuming perfect play).

Insight from search

Since we could not find analytical solutions for $n > 4$, we next turned to using explicit search: to show that the first player can always win for a given n, for example, we simply check recursively that there exists a move that the first player can make such that *any* move made by the second

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
18	17	20	19	22	21	24	23
26	25	28	27	30	29	32	31
2	1	4	3	6	5	8	7
10	9	12	11	14	13	16	15

Table 1: A strategy for the second player to always win on the 8×8 board, in the form of a look-up table with exactly two tiles labeled i for $1 \leq i \leq 32$. If the first player chooses tile i, the second player should place their next tile on the other tile indexed by i.

player yields a position in which the first player can eventually force a win. The number of possible configurations for an $n \times n$ board is 2^{n^2} and the number of possible games is bounded by $(n^2)!$ (although not all of those configurations and games are reachable in play).

For $n = 5$, explicit search easily verifies that the first player can always force a win. In fact, the first player can choose *any* tile in the first move and still win later on. If they choose the center tile on the first move, then they can choose *any* tile on their second move and still force a win, regardless of which tile the second player chooses on their first move.

For $n = 6$, we implemented several straightforward optimizations to allow explicitly searching all possible games (about 40 quadrillion). Because many paths through the game tree intersect in any given state, we build a look-up table of all possible states, indexed by a 36-bit integer that encodes the configuration of the board (where 1s are Xs and 0s are empty squares). Each entry in the look-up table needs 2 bits, one indicating if the first or second player wins from this state, and one indicating whether the winner has been computed yet. The look-up table thus requires about 20GB of memory to store all 2^{36} possible states. We also use "bit twiddling" to efficiently detect losing configurations (completed rows, columns, and diagonals) using bitwise binary arithmetic.

Our search found that the first player can guarantee a win for $n = 6$, which (to our knowledge) is a new result. The search took only about 15 minutes on a single CPU. Unfortunately, this approach would require nearly 150 TB of RAM for $n = 7$. We tried various alternative implementations, but were not able to find one practical for our computational resources.

A general winning strategy is not known

A general winning strategy for all boards remains elusive for any board $n \times n$ board with any $n \geq 5$ that is not multiple of 4. Indeed, if $n = 1$ there is only one move available and the first player will complete a row — i.e., lose — in that move. If $n = 2$, the first player will win since the second player is compelled to complete a row, column, or diagonal in its first move. The case $n = 3$ has been addressed in (Chow 2010) with a "knight move strategy" where the first player should cover the center tile and then counter each subsequent move by placing a tile in a square that is a knight move away from the previous move (either two rows and one column, or two columns and one row apart). If $n = 4$, Theorem 1 describes a perfect strategy for the second player. Despite that no general strategy is known, the fact that there exists a simple, provably-optimal strategy for $4k \times 4k$ for any k should mean that a powerful reinforcement learning-based game playing system, such as AlphaGo Zero, should be able to easily learn it. Moreover, it suggests that simple strategies may exist for other board sizes as well — if only we could find them.

Enter AlphaGo Zero

Of course, large branching factors are nothing new in adversarial AI research. Most recently, AlphaGo Zero (Silver et al. 2017b) has demonstrated that deep artificial neural networks are capable of learning to play games like Chess and Go at a level that quickly surpasses that of human champions. The original version, AlphaGo Zero (Silver et al. 2016), was specifically designed for Go and was trained using human supervision and hand-crafted features. AlphaGo Zero is much more general, avoiding human supervision through self-play, which simplifies the training process significantly.

Given that AlphaGo Zero performs so well on Go (where the board is 19×19), one would assume that it would easily learn to play notakto on our relatively modest 7×7 board, especially given that notakto is conceptually much simpler. Surprisingly, we found that it failed to play well even on small notakto boards. To explain why it fails and to suggest a fix, we now discuss key parts of AlphaGo Zero in more detail.

Background on AlphaGo Zero

AlphaGo Zero uses Monte-Carlo Tree Search (MCTS) to play against itself from random states. MCTS records some statistics, such as the times of each state-action pair ($N(s, a)$) appears as well as the corresponding results (-1 or 1), etc., to calculate expected reward $Q(s, a)$ during self-playing. The loss function is designed (1) to minimize the difference between predicted outcome v and the actual outcome z, and (2) to minimize the difference between the policy vector which comes from random playing (search probability π) and the predicted policy vector $\mathbf{p}$. That is, AlphaGo Zero tries to minimize a loss,

$$l(\theta) = (z - v)^2 - \pi^T \log \mathbf{p} + c\|\theta\|^2 , \tag{1}$$

where third term $c\|\theta\|^2$ is a regularizer to avoid overfitting, with c the weight of the regularizer and θ the parameters of the neural network.

AlphaGo Zero's training is the normal policy iteration of reinforcement learning: it computes Q values and learns policies based on these Q values. It uses a neural network to simulate the statistics of $Q(s, a)$. Please refer to (Silver et al. 2016) for details.

AlphaGo Zero for Othello and Dawson's chess

We based our implementation of AlphaGo Zero on publicly-available code (Nair 2017). Before applying it to notakto,

	Othello win rate		Dawson win rate	
Rival	as Black	as White	as first	as second
Random	100%	100%	99%	75%
Greedy	100%	100%	—	—
Itself	2%	98%	100%	0%

Table 2: AlphaGo Zero's performance on 6×6 Othello and 3×9 Dawson's chess, as the first and second player, when played against a random player, a greedy player (that plays randomly but avoids immediately losing moves), and itself. An optimal player would always win Othello as white (second), and always win Dawson's chess as the first player.

we first tested it on two games which are better understood to verify that our implementation was working correctly. We first trained a 6×6 Othello game, which is strongly solved (the white player can always force a win) (Feinstein 2004). We found that our trained model almost always forces a win (as second player) when it plays against a random player (100%), a greedy player (100%), and itself (98%), as shown in Table 2.

To test on a game more similar to notakto, we implemented Dawson's Chess (Conway 2001; Dawson 1934), which is a misére game played on a $3 \times n$ chessboard with pawns of opposite colors on the first and third row. White moves first and if a capture is possible, then it must be performed. The last player to move loses. On a 3×9 board, it is known that the first player can force a win by moving its center piece on the first move, and making their second move not adjacent to the opponent's first move. We trained our implementation of AlphaGo Zero on a board of this size a Dirichlet α of 1 and a c_{puct} of 6. The loss stopped decreasing at about 50 iterations each with 100 simulations. As with Othello, Table 2 indicates that AlphaGo Zero has learned a near-optimal strategy, suggesting that our implementation is working correctly.

AlphaGo Zero for notakto

We expected AlphaGo Zero to be competitive in playing notakto given its prowess in much more complicated games like Chess and Go, and our results for Othello and Dawson's Chess above. Surprisingly, we found that even for our modest board sizes, it did not perform much better than random play.

Our hypothesis is that this is due to a unique property of notakto. For most games, random play repeated enough times can simulate all cases (states) of a game. However, in notakto's case, there are certain very important states that prevent the game from ending early, and it is important that these states be sampled frequently since they normally arise in competent play. We call these *key states*. In Figure 3, for example, random sampling would place pieces at positions $1, 2, 3, 4$, or 5 with equal probability, but only position 4 makes sense; the other positions would immediately end the game, which means that play involving the piece at 4 is not sampled enough times. As a result, an optimal model is not easily learned, because the game tends to end prematurely.

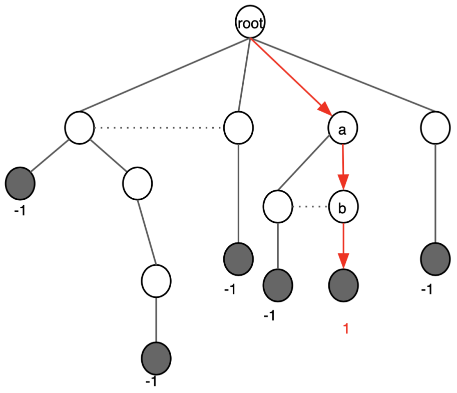


Figure 2: An example showing a case where only one path through the game tree results in a win, preventing AlphaGo Zero from learning a reasonable policy. See text for details.

In more detail, say we have a Markov decision process to find solutions that can maximize the reward (-1, or 1), as Figure 2 shows. There are many branches starting from the root, but only one path (colored by red) is optimal. During search, we use $\sum N(s,b)/N(s,a)$ to determine the searching direction. Obviously, it will take a long time for AlphaGo Zero to begin to search the optimal route ($root \rightarrow a \rightarrow b \rightarrow leaf$). This is not a big issue since we can adjust other parameters in AlphaGo Zero like c_{puct}, Dirichlet α, and τ to encourage the exploration. A flaw of the algorithm here is that AlphaGo Zero trains a neural network to simulate the statistics of explorations $(N(s,b), N(s,a))$ to get the policy $\pi(s,a)$, and the searching probability is proportional to its exponentiated visit count ($\pi_a \propto N(s,a)^{1/\tau}$). In other words, although most of the exploration statistics should not be part of policy, the neural network still tries to simulate them, so AlphaGo Zero just uses a neural network to simulate explorations rather than learn from them. This issue has also been pointed out in more detail in (Selsam 2018).

Key states sampling for notakto

We tried to address this sampling issue by applying a smart way of exploration in the specific context of notakto, and in particular by biasing the sampling towards the "key states." Key states are those crucial states that determine a win or loss of a game. The core idea of key state sampling (see Algorithm 1) is to avoid "bad" samples that could end the game early. From the viewpoint of machine learning, key state sampling re-balances the training data, effectively giving these crucial samples a higher weight — e.g., samples that involve Position 4 in the example of Figure 3. We still adopt random sampling during training, but deliberately prevent some states (like Positions 1, 2, 3, 5) from being taken into the calculation of Q values ($Q(s,a)$).

Since notakto boards are much smaller than Go, we used a neural network with 4 convolutional layers (kernel size 3), each with 512 nodes, and three full connected layers (with number outputs for 1024, 512, and the size of the board

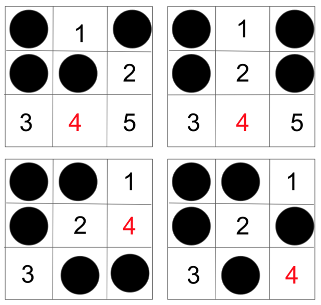


Figure 3: Illustration of key states sampling; see text.

plus one (to indicate no available moves), respectively). The learning rate was set to 0.001. We use exactly the same network architecture for our tests of both the original AlphaGo Zero, and AlphaGo Zero with key state sampling.

```
Sampling initialization $N(s, a)$;
while not converge do
    Get all valid candidate states given current state;
    Extract key states (remove states that result in loss);
    Update visit count $N(s, a)$ by key states;
    Update mean action-value $Q(s, a)$ ;
end
```

Algorithm 1: Key state sampling

Testing for convergence AlphaGo Zero has two phases. The first phase is sampling $(Q(s, a), N(s, q))$, and the second phase trains a neural network to simulate the statistics of sampled records. To make our model converge to the optimal point, we need to make sure the samples we extract are representative enough, and the loss of the neural networks is small enough. For the first phase, if we find that $Q(s, a)$ stops changing (for small boards) or changes very slowly (for large boards), then we know that the first phase (sampling) converges. For the second phase, we can monitor the loss curve for both policy vectors as well as the outcome (our model is fundamentally a neural network). Apart from the above method, for notakto, we can also take advantage of its unique property to decide whether we should end the training. For example, for a particular board of size n, if our model converges, the winning rate should be close to 50% when our model plays against the previous best model (since each player takes the first turn in half the rounds). Thus, a winning percentage of self-playing close to 50% is also an indicator of converging. In this paper, we check the winning rate (current best model versus previous best model) for an indication of convergence.

Key state sampling for small boards ($n \leq 6$) We compared the performance of AlphaGo Zero with our key state sampling to the original AlphaGo Zero implementation, when playing against three rivals: themselves (**case #1**), a greedy player that plays randomly but avoids placing a piece that will complete a row, column, or diagonal, unless there is no choice (**case #2**), and each other (**case #3**). We trained

			Ours /AlphaGo Zero	
n	Iterations	Rival	First	Second
$n = 3$	3 / 3	Itself	100% / 100%	0% / 0%
		Greedy	100% / 100%	100% / 100%
		AlphaGo Zero	100% / 100%	0% / 0%
$n = 4$	83 / 102	Itself	0% / 2%	100% / 98%
		Greedy	100% / 100%	100% / 100%
		AlphaGo Zero	6% / 0%	100% / 94%
$n = 5$	58 / 72	Itself	78% / 58%	22% / 42%
		Greedy	97% / 93%	70% / 68%
		AlphaGo Zero	90% / 57%	43% / 10%
$n = 6$	33 / 33	Itself	67% / 50%	33% / 50%
		Greedy	85% / 79%	79% / 69%
		AlphaGo Zero	66% / 54%	46% / 34%

Table 3: Comparison of the performance of our model (before the slash) and AlphaGo Zero (after the slash) on notakto boards of size $n = 3, 4, 5, 6$, as first and second player, and against three different opponents. An optimal player would win if playing first for $n = 3, 5, 6$, and would win if playing second for $n = 4$. Iterations is the number of training iterations for our model and AlphaGo Zero, respectively.

on a machine with one Tesla V100-SXM2-16GB GPU. For each board size, we trained our models until they converged or made reasonable predictions. The total number of iterations for training is shown in the second column of Table 3. For each case, we test its corresponding models 100 times.

For **case #1**, as Table 3 shows, both our model and the original AlphaGo Zero model learn the optimal or close-to-optimal policy for $n = 3, 4$, although our model is a little better. For $n = 5, 6$, the first player appears to have an advantage, although it appears an optimal strategy has not been learned. Nevertheless, we find that no matter what n is, the tendency is clear and our model always beats the raw AlphaGo Zero model. One exception is for the AlphaGo Zero model for a 6×6 board, where there is no difference between taking the first move or letting the adversary take the first move, at least for this case.

For **case #2**, both our model and the raw AlphaGo Zero model achieve better performance than the greedy player even when they have the disadvantage of taking the first or second move, and our model outperforms the original AlphaGo Zero.

For **case #3**, as Table 3 shows, our model achieves better performance than the AlphaGo Zero model. We can conclude that our model based on key state sampling is more efficient, requiring fewer iterations to achieve a higher winning rate.

Key-state sampling for large boards ($n > 6$) For a large board, we evaluate our sampling by having it play against itself. We use the same Tesla V100 GPU as above. For each board size, we train our model for about 20 iterations, with each iteration containing 100 episodes that are generated by key state sampling. Note that for each episode, we sample all the states (the initial state and all the immediate states)

		Our model's win rate	
n	Rival	as first player	as second player
$n = 7$	Itself	59%	41%
$n = 8$	Itself	30%	70%
$n = 9$	Itself	60%	40%
$n = 10$	Itself	52.5%	47.5%
$n = 11$	Itself	57.5%	42.5%
$n = 12$	Itself	80%	20%

Table 4: Our model's performance on $n = 7, 8, 9, 10, 11, 12$ Notakto, as the first and second player, when it plays against itself.

	Dynamic threshold versus fixed threshold	
n	as first player	as second player
$n = 7$	55% / 55%	45% / 45%
$n = 8$	60% / 45%	55% / 40%
$n = 9$	25% / 50%	50% / 75%
$n = 10$	60% / 45%	55% / 40%
$n = 11$	70% / 65%	35% / 30%
$n = 12$	60% / 60%	40% / 40%

Table 5: Updating with dynamic threshold (before the "/") versus fixed threshold (after the "/").

all along to the end state.

We let our model play against itself 100 (n=7, 8) or 40 (n=9, 10, 11, 12) times. Ideally, the winning rate for self-play should be close to 100% for the first or second player, which would indicate that the model has learn an optimal strategy. As shown in Table 4, the winning rates of moving first for $n = 7$ and second for $n = 8$ are 59% and 70%, respectively. This suggests that the first player for $n = 7$ and the second player for $n = 8$ have an advantage. Similarly, for ($n = 9, 10, 11, 12$), the first player has an advantage. However, according to Theorem 1, for ($n = 12$), the second player should have an advantage. We conjecture that our model for ($n = 12$) is still far away from being optimized.

AlphaGo Zero with Dynamic threshold for updating

In AlphaGo Zero (Silver et al. 2017b), a new model will be updated if it can win against the previous optimal model by a margin of at least 0.55. In particular, after each iteration, the newly acquired model p_1 will play against the last best model p_2. Say n_1, n_2 are the number of games won/lost by p_1 as the first player, and n_3, n_4 are the number of games won/lost by p_2 as the first player. Then AlphaGo Zero will decide whether to update the best model if,

$$\frac{n_1 + n_4}{n_1 + n_2 + n_3 + n_4} > 0.55. \qquad (2)$$

During training, we found that Equation (2) often results in "bumpy" loss curves, as the red curves in Figure 4 show. We modified this threshold to reflect the winning rate of r in the training dataset,

$$\left(\frac{n_1}{n_1 + n_2} > r\right) \vee \left(\frac{n_3}{n_3 + n_4} > 1 - r\right). \qquad (3)$$

This new dynamic threshold gives smoother loss curve (as Figure 4 shows) as well as better performance (as Table 5 shows).

Note that we did not discuss how the meta-parameters in AlphaGoZero (such as c_{puct}) impact the performance because we believe that these parameters are independent of the key states sampling and the dynamic threshold.

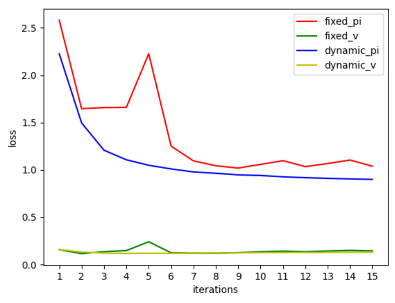
Discussion and Conclusion

In this paper we present preliminary work that makes several steps towards a better understanding of a combinatorial game that has proven difficult to analyze, notakto, but also suggests that additional work is needed. First, we provide a formal proof that the second player can always force a win when playing on a $4k \times 4k$ board with $k \in \mathbf{Z}^+$, and we give a strategy for optimal play. We also determine, for the first time, that the first player wins the 6×6 board. These both formalize and extend the results found in (Chow 2010). Unfortunately, these analytic and search techniques could not be applied to larger boards.

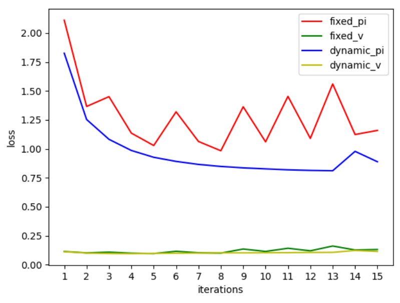
So we instead tried an alternative approach: trying to train an AI to play notakto well, and then using the learned strategy as a tool to prove properties of the larger boards. Surprisingly, we tested a standard AlphaGo Zero implementation and found that it does not perform very well, even though we *did* find that it worked well for Othello (in normal play) and Dawson's chess (in misére play), two games that seem intuitively similar.

We hypothesized that this is due, at least in part, to the fact that AlphaGo Zero trains the policy neural network ($\pi(s, t)$) from statistics of explorations and does not learn from the exploration itself. Since the exploration statistics reflect only the exploration history rather than the optimal policy that should have been taken, many exploration statistics should have been removed. We proposed and tested a technique we call "key state sampling" as a way of addressing this weakness of AlphaGo Zero in notakto. The idea is to bias sampling towards more important states, which re-balances the distribution of different state samples and forces key states to be sampled more frequently. For the neural network we used, any violations or disagreement with the key states' statistics result in a larger loss (as key state samples have larger weights), which in turn forces the neural network to focus on key state samples to reduce the training loss.

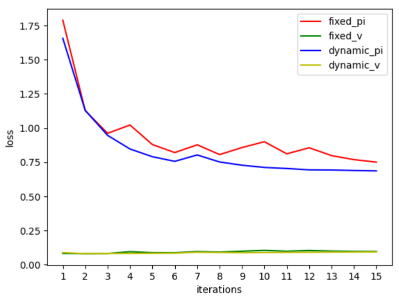
Our experimental results show that key state sampling improves the performance of AlphaGo Zero on notakto, playing better in fewer training iterations than the original algorithm. However, our results show that neither the original AlphaGo Zero nor our version has learned an optimal strategy for boards of $n = 5$ and greater, suggesting that notakto is significantly more challenging to learn than Othello, and perhaps beyond the compute capabilities of even our extremely high-end Tesla V100 GPU. However, our trained



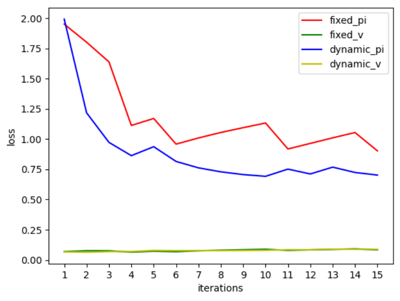
(a) $n = 7$



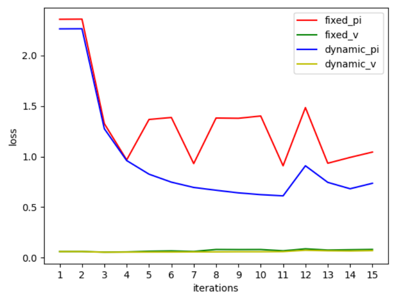
(b) $n = 8$



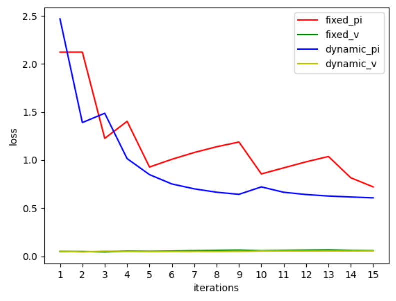
(c) $n = 9$



(d) $n = 10$



(e) $n = 11$



(f) $n = 12$

Figure 4: Fixed threshold versus dynamic threshold. "fixed_pi" or "dynamic_pi" is the loss that comes from the prediction for next state while "fixed_v" or "dynamic_v" is the loss that comes from the prediction of "who's winning". Note that we only list the cases of $n > 6$ to emphasize the generalization for large boards.

model wins significantly more often as the first player than the second player when $n = 6$, and significantly more often as the second player than the first player when $n = 8$, both of which agree with the expected winner that we have derived analytically or through brute force search. The fact that our model wins significantly more often as the first player with $n = 7$ leads us to conjecture that there exists an optimal strategy for the first player to always win, but proving this will require future work.

References

Breuker, D. M.; Uiterwijk, J. W. H. M.; and van den Herik, H. J. 2000. Solving 8×8 Domineering. *Theoret. Comput. Sci.* 230(1-2): 195–206. ISSN 0304-3975. doi: 10.1016/S0304-3975(99)00082-1. URL https://doi.org/10.1016/S0304-3975(99)00082-1.

Chow, T. 2010. Neutral tic tac toe. https://mathoverflow.net/questions/24693/neutral-tic-tac-toe/. Accessed: 09/02/2018.

Conway, J. H. 2001. *On numbers and games*. A K Peters, Ltd., Natick, MA, second edition. ISBN 1-56881-127-6.

Dawson, T. 1934. Fairy chess supplement. *The Problemist: British Chess Problem Society* 2(9): 94.

Feinstein, J. 2004. Perfect play in 6×6 Othello from two alternative starting positions. *http://www.feinst.demon.co.uk/Othello/6x6sol.html* URL https://ci.nii.ac.jp/naid/10013322109/en/.

Kawano, Y. 1996. Using similar positions to search game trees. In *Games of no chance (Berkeley, CA, 1994)*, volume 29 of *Math. Sci. Res. Inst. Publ.*, 193–202. Cambridge Univ. Press, Cambridge.

Nair, S. 2017. A Simple Alpha(Go) Zero Tutorial. https://web.stanford.edu/~surag/posts/alphazero.html.

Plambeck, T. E.; and Siegel, A. N. 2008. Misère quotients for impartial games. *J. Combin. Theory Ser. A* 115(4): 593–622. ISSN 0097-3165. doi:10.1016/j.jcta.2007.07.008. URL https://doi.org/10.1016/j.jcta.2007.07.008.

Plambeck, T. E.; and Whitehead, G. 2014. The secrets of Notakto: winning at X-only tic-tac-toe. *Recreat. Math. Mag.* (1): 49–54. ISSN 2182-1976.

Schaeffer, J.; Björnsson, Y.; Burch, N.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2005. Solving Checkers. In *IJCAI*, 292–297. Professional Book Center.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers Is Solved. *Science* 317(5844): 1518–1522. ISSN 0036-8075. doi:10.1126/science.1144079. URL http://science.sciencemag.org/content/317/5844/1518.

Selsam, D. 2018. Issues with AlphaZero. https://dselsam.github.io/issues-with-alpha-zero/.

Siegel, A. N. 2013. *Combinatorial Game Theory*, volume 146 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI. ISBN 978-0-8218-5190-6.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529(7587): 484–489. doi: 10.1038/nature16961.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017a. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv e-prints* arXiv:1712.01815v1.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017b. Mastering the game of Go without human knowledge. *Nature* 550: 354 EP –. URL http://dx.doi.org/10.1038/nature24270.

Yoshizoe, K.; Kishimoto, A.; and Müller, M. 2007. Lambda Depth-First Proof Number Search and Its Application to Go. In *IJCAI*, 2404–2409.