Playing Notakto

Caleb Kisby

October 2021

Background

Consider the following variant on the classic game tic-tac-toe. As usual, players take turns placing their tokens on a board. But instead of 'X's and 'O's, both players place 'X's on the board — this is known as *impartial* tic-tac-toe. Additionally, the player that completes a row, column, or diagonal *loses* the game (known as *misère-play*).

This game on an $n \times n$ board was proposed by Timothy Chow in a mathoverflow.net thread (Chow 2010), and was later coined *Notakto* by (Plambeck and Whitehead 2013). Tic-tac-toe has a well-known winning strategy, so the natural question arises: Assuming perfect play, which player should win at Notakto?

(Plambeck and Whitehead 2013) answers this question for *disjunctive* play, i.e. the game on a number of disjoint 3×3 boards. But for a single $n \times n$ board, the question is harder to answer. First, the easy cases: On a 1×1 board the second player always wins, and on a 2×2 board the first player always wins. For $n = 3$, the first player can secure a win by placing an 'X' in the center of the board on their first move. The question has also been answered for $n = 4, 5, 6$, and also for $n = 4k, k \in \mathbb{Z}^+$:

n	**Winner under perfect play**	**Reference**
1	2nd Player	
2	1st Player	
3	1st Player	
4	2nd Player	(Chow 2010)
5	1st Player	(Chow 2010)
6	1st Player	(Chen et al. 2021)
$n = 4k$	2nd Player	(Chen et al. 2021)

Unfortunately, brute-force solving of the game is intractable and impractical even for boards as small as $n = 6$.

(Chen et al. 2021) had the major insight to leverage a trained neural network to glean a strategy for Notakto. They used AlphaGo Zero (Silver et al. 2017), which intends to be a common ANN-based framework for mastering board games beyond Chess and Go. The hope was that AlphaGo Zero's success with tic-tac-toe would generalize to the impartial misère setting of Notakto. Unfortunately, they found that AlphaGo Zero failed to learn competent strategies even for small board sizes. The authors hypothesize that its failure is due to the statistically rare nature of key Notakto board states. They account for this by biasing AlphaGo Zero's sampling towards these key states, but they lament that this fix is specific to the game of Notakto.

In this report, I explore the hypothesis that AlphaGo Zero's failure is due to special knowledge involved in Notakto that the system is not able to easily represent. This might be surprising, given AlphaGo Zero's claim to learn *tabula rasa*. But some critics, e.g. (Marcus 2018), make a strong case that AlphaGo Zero implementations (and successive versions) make use of a significant amount of knowledge about the particular game at hand. Marcus points out for example that the Go implementation made use of a sampling mechanism that the Chess implementation did not — presumably because this choice resulted in poorer performance at Chess.

To test this, I supplemented an AlphaZero tic-tac-toe ANN with additional features. These features were determined via a rough analysis of the knowledge involved in Notakto. In the following sections, I will describe this system and then assess its performance against a variety of players, including the original tic-tac-toe ANN. Afterwards, just for fun, I will illustrate my ANN's Notakto play style by analyzing heat maps of its behavior.

My Notakto Player

Basic Approach

In order to explore the above hypothesis, I conducted some preliminary tests. The goal of these tests was to determine how much we can improve an AlphaGo Zero ANN by supplementing its input with knowledge-rich features.

I opted not use the AlphaGo Zero framework for these tests. My primary reason for this is that AlphaGo Zero's competence is a result of many factors besides the ANN architecture, e.g. its use of Monte Carlo tree search and its sampling technique. Instead, I need a controlled setting where the competence of the system depends heavily on the choice of ANN architecture.

The following is a high-level overview of this environment[1]; I will go into detail in the sections that follow. Rather than learn via self-play, my ANN learns all at once from a set of training boards. Given a board, the ANN aims to predict a score (via regression) representing how likely the ANN is to win from this board. We can label the training boards with these scores via greedy play.

Once trained, this system plays as follows. Given a board, we only consider possible positions on the board that do not end the game prematurely. Of these positions, we use the

[1] If you're interested, I've uploaded my code to github. Mind the mess! https://github.iu.edu/cckisby/qualifying-exam/tree/master/1-NN_architecture_and_Notakto

ANN to evaluate the board that results from taking that position. During evaluation, we consider the board from our *opponent's* point of view; ideally, we should take the move whose board has the worst score for the opponent. But this policy would result in deterministic game-play. That is, the system would only play out a single game when pitted, e.g., against itself. So instead we introduce an element of randomness: We choose the second best move with a probability of $\frac{1}{10}$.

Generating Training Boards

When generating a set of boards to train on, (Lai 2015) stresses that we must strike a balance between these competing requirements:

1. **Distribution**: The boards should be representative of typical play, i.e. the set should have the correct distribution.
2. **Variety**: The set should contain some unusual boards, e.g. boards that are unfairly stacked against the player.
3. **Volume**: The set should be large enough for the ANN to generalize.

In order to obtain boards that (1) represent typical play, I run a certain number of games involving a greedy player against itself. This player is slightly more competent than a random player, since it will never select a square that prematurely ends the game. We use all of the board states resulting from these games as a base for training set generation.

We then introduce (2) variety and imbalance in this initial base of game boards. Following (Lai 2015), we apply up to two random legal moves to each board (so long as these do not prematurely end the game). We then take all of the resulting boards and *flip* the player whose turn it is — a clever trick introduced by the Falcon Chess system (Lai 2015).

This generation process is completely automated and has a natural exponential blowup, so in principle we have (3) having enough boards to train on. But the process is also brute-force, and so generation can take an impractical amount of time. I compromised by running enough games to have a large enough training set, but not too many to run my patience thin.

Evaluating Training Boards

Our goal is to have the ANN use regression to predict *how likely* a given board state is. To do this, we have to tag our training set with these likelihood scores.

This is straightforward: We just perform a search of the game tree, stop at a certain level, and then determine the winner by having playing each leaf board according to the greedy strategy. We score the root board with the average of the wins of these leaf boards. I decided to keep the tree search simple, i.e. a deterministic 2-move depth search rather than something like Monte Carlo tree search. Again, I choose to impair the system in this way so that the competence of the system is not too dependent on the tree search used. (Although in a future implementation I plan on replacing this with a much more competent search.)

Feature Representation

How we represent our game boards influences to a great extent our ANN's ability to learn Notakto. Since AlphaGo Zero intends to learn without any domain knowledge about the game at hand, it uses a 'raw' representation of the board. And while this is always possible in principle, AlphaGo Zero's poor performance on Notakto suggests that certain information about Notakto may be difficult to learn without assistance. We can supplement AlphaGo Zero's 'raw' representation with additional features deriving from our knowledge of the game. With these extra features, the system will not learn information it *couldn't* before, but rather its search space will be guided towards more relevant information.

To come up with features, I played lots of Notakto games. I arrived at the following list through a rough analysis of the considerations I made during play:

Turn Flag. One thing we might notice about Notakto play is that, unlike tic-tac-toe and Chess, one cannot quickly determine who won a game by looking at the final game board. In partisan games the winner is easily determined by checking, e.g. whether the 3 in a row was formed by 'O' rather than 'X'. But for impartial games, we have to count the parity of moves made. Counting the number of 'X's on the board takes some effort, for both humans and an ANN. And so we provide the ANN with a *turn flag* indicating whose turn it is on a given board.

Dead Squares. Next, in Notakto the space of possible moves contrains as the game progresses. This contrasts with Chess, where the board opens up late-game. In addition, the misère nature of Notakto means that the space of moves that don't prematurely lose the game rapidly vanishes. We call moves that *do* prematurely lose the game *dead squares*. By providing the ANN with the number of dead squares, we allow it to more easily learn how late the game is. Also, the parity of the dead squares on a board might suggest which player has the advantage.

Row Parity. Finally, when one plays enough Notakto they might begin to develop a certain foresight about whether rows, columns, or diagonals are worth pursuing. For instance, say you're making your next move on a 4x4 board, and suppose a row already has an even number of 'X's. Then there are an odd number of empty spaces left in this row. Assuming you and your opponent both took turns placing 'X's on this row alone, you would lose. We can codify this foresight via the *parity* of X's on that row, column, or diagonal.

I tried other configurations of these features, e.g. providing a map of dead squares rather than the count. But these variations did not help the ANN learn. The reason why *these* features do well is a complicated issue. (Lai 2015) suggests this guiding principle for evaluating features:

> For neural networks to work well, the feature representation needs to be relatively smooth in how the input space is mapped to the output space. Positions that are close together in the feature space should, as much as possible, have similar evaluations.

But this is difficult to apply. For example, it's not true that all boards with the same turn flag are desirable boards, but the turn flag along with other features *might* result in a smoother feature space.

Neural Network Architecture

My own ANN (the 'Featured Net') builds on the net used by (Nair 2018) for training AlphaZero to play tic-tac-toe (the 'AlphaZero Net'). This AlphaZero Net consists of four 2D convolutional layers, followed by a layer to flatten the dimension, followed by two dropout layers (to help prevent overfitting). Batch normalization (Ioffe and Szegedy 2015) is performed between each layer as an extra measure against overfitting. All nodes within the hidden layers use Rectified Linear activation (ReLU) (Glorot, Bordes, and Bengio 2011). In the original net, softmax and hyperbolic tangent activation functions were used for the output nodes. However, since the task at hand is regression I have replaced these with a linear activation function.

My Featured Net begins similarly with these four 2D convolutional layers and a flatten layer. But after the flatten layer, I feed this flattened layer along with all of the features — the turn flag, the number of dead squares, as well as the parity for each row, column, and diagonal — into a merged layer. From here, I employ an abstraction layer of size n (the board dimension) in the hope that this will allow the net to learn higher-level concepts (Lai 2015). These additional hidden layers all use ReLU activation, and the output layer uses linear activation. As for preventing overfitting, we perform batch normalization between these new layers, but remove the dropout layers from before.

Other parameters for the nets (e.g. the dropout rate, batch size) can be found in the source code (see the footnote on the first page).

Assessment

Tests and Results

My intent here is to explore the hypothesis that AlphaGo Zero cannot easily learn Notakto due to its lack of special knowledge about the game. To this end, I set up these preliminary tests to see whether my Featured Net (with Notakto-specific features) performs better than the AlphaZero Net in this simple controlled environment.

These tests consisted of games of Notakto played on 3×3, 4×4, 5×5, and 6×6 boards. For each n, I generated a number of boards for training: 155 for $n = 3$; 2378 for $n = 4$; 9958 for $n = 5$; and 79055 for $n = 6$. Both the Featured Net and the AlphaZero Net were trained on the same set of boards. Then, they were each pit against four opponents: Themselves, the other, a random player, and a greedy player. To highlight their ability to learn winning strategies, I tested their ability to play as both first and second player. For each configuration of board size, opponent, and player, I ran 100 trials.

The results of these tests are shown in Figure 1. I should warn that although this table is formatted similarly to the results from (Chen et al. 2021), it should not be read the same way. In (Chen et al. 2021), they use slashes to indicate differences between their system (before the slash) and the AlphaGo Zero system (after the slash). But I found it more helpful to use them to compare performance as first player (before the slash) and performance as second player (after the slash). To highlight each net's ability to learn the winning strategy, the results corresponding to the known winning player (assuming perfect play) are shown in blue.

n	**Opponent**	**Featured Net**	**AlphaZero Net**
$n = 3$	Itself	100% / 0%	92% / 8%
	Featured Net		100% / 1%
	AlphaZero Net	99% / 0%	
	Random	94% / 82%	90% / 84%
	Greedy	79% / 26%	88% / 20%
$n = 4$	Itself	8% / 92%	92% / 8%
	Featured Net		18% / 82%
	AlphaZero Net	18% / 82%	
	Random	93% / 94%	91% / 96%
	Greedy	37% / 55%	49% / 54%
$n = 5$	Itself	59% / 41%	41% / 59%
	Featured Net		47% / 49%
	AlphaZero Net	51% / 53%	
	Random	99% / 99%	99% / 96%
	Greedy	54% / 48%	60% / 39%
$n = 6$	Itself	23% / 77%	69% / 31%
	Featured Net		33% / 53%
	AlphaZero Net	47% / 67%	
	Random	100% / 99%	99% / 98%
	Greedy	52% / 47%	55% / 59%

Figure 1: The Featured Net and AlphaZero Net (top) were each pit in 100 games against various opponents (left) for board sizes $n = 3, 4, 5, 6$. The win rates of the nets are shown. Before the slash indicates performance as first player, and after the slash indicates performance as second player. Performances as the known winning player are highlighted in blue.

Discussion

First, I should mention that the results in this table are generally worse than the results obtained in (Chen et al. 2021). But this is expected, since our testbed environment is much less competent than the AlphaGo Zero environment. The significance of these results is not the systems' raw performance, but instead is how the Featured Net and the AlphaZero Net compare.

In addition, our system has an explicit bias to pick a suboptimal move $\frac{1}{10}$th of the time. This nondeterminism is necessary for the ANNs to vary their play. But during testing I found that this bias significantly affects the raw results. Unfortunately we are only performing 100 trials per scenario, so reducing the bias any further results in the two nets playing redundant games against themselves.

How do the two ANNs compare? The issue at stake is whether these ANNs learn the winning strategy for each n — and which ANN learns it more easily. For instance, consider $n = 3$. As first player, both the Featured Net and the AlphaZero Net play nearly perfectly against all but the greedy opponent, where they both fare alright. Both are clearly learning a near-optimal strategy for $n = 3$. But we see some major discrepancies: The Featured Net is slightly better at self-play, whereas the AlphaZero Net is slightly better against the greedy opponent.

One might argue that the systems' performance against greedy represents "ground-truth" competence. Perhaps the Featured Net is worse overall, and only beats itself so often because it doesn't put up a fight as the second player. This suspicion appears to be confirmed by the fact that across all n, the Featured Net's performance against greedy (taking the known winning player's turn) is only ever the same or worse as the AlphaZero Net.

But this argument breaks down when we examine the AlphaZero Net's self-play for other n. For instance, when $n = 4$ this net (as player 2) loses *92%* of the time! Similarly, when $n = 5$ this net (as player 1) loses more often than it wins. This seems very unlikely if the AlphaZero Net learns the winning strategies more easily than the Featured Net.

My own working hypothesis is this: The Featured Net isn't worse per se. It occasionally plays more competently than the AlphaZero Net when it is the player with a winning strategy. When its performance is worse *as this player*, it is never by a significant degree. But the Featured Net does seem to have a fixation on the strategy that works *as this player*. So while it seems to me that the Featured Net learns the winning strategies more easily, I should qualify that its self-play results are skewed due to its incompetence as the other player.

Mapping The Featured Net's Notakto Strategy

Our initial motivation for training an ANN to play Notakto competently was to derive winning strategies for $n > 6$. Unfortunately, my Featured Net in its impaired environment fails to converge on a strong winning strategy even for $n = 5$ and $n = 6$. But it does seem to play competently for $n = 3$ and $n = 4$. Just for fun, I decided to analyze the behavior of this ANN on these small boards to figure out what strategy it has arrived at.

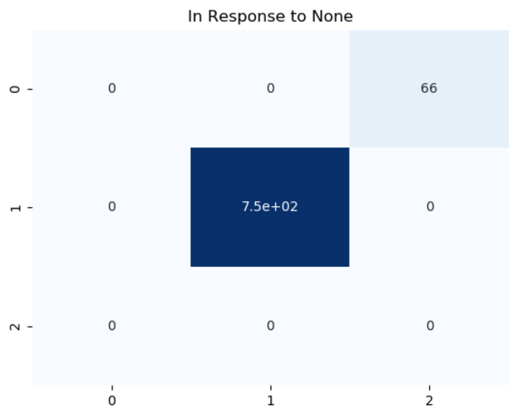

Figure 2: A heat map of the Featured Net's opening moves. These were selected from games where the Featured Net was player 1 and won. The darker the color of a position, the more frequently the net used that position to open.

We can visualize the Featured Net's play style by looking at heat maps of the its responses to the opponent. For each board size, we pit the net in 1000 games on a against a greedy opponent. The Featured Net was player 1 in all of the 3×3 games, and it was player 2 in all of the 4×4 games. Of these games, we only consider those in which the Feature Net won. We collect the Feature Net's responses to each of its opponent's moves, along with the number of times this response was selected. Figure 2 shows the Featured Net's responses to no move on a 3×3 board, i.e. its opening move.

For $n = 3$, there is a unique winning strategy: Player 1 opens by placing an 'X' in the center of the board. This forces a loss for player 2. This heat map reveals that in 750 of the 816 games where the Featured Net won, it did exactly this. This confirms that the Featured Net figured out the winning strategy for $n = 3$. (In fact, the 66 games where the ANN chose a different square appear to be due to the ANN's bias to occasionally choose its second-best option.)

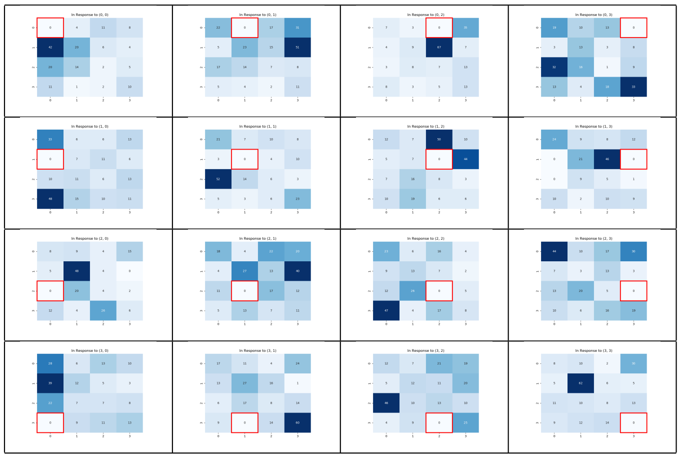

Figure 3: An array of heat maps revealing the Featured Net's responses to its opponent. These were selected from games where the Featured Net was player 2 and won. For each possible move made by the opponent (i.e. each position on the larger board), we display the heat map of the net's responses to this move. For convenience, the opponent's moves are also shown by a red square on each inner board.

For $n = 4$, the known winning strategy (for player 2) to mirror the opponent's moves according to the following table (Chow 2010):

a	b	c	d
e	f	g	h
b	a	d	c
f	e	h	g

That is, whenever player 1 places an 'X' on a square, player 2 responds by placing an 'X' on the square with the corresponding letter. The crux of this strategy is that the rows, columns and main diagonals are mirrored such that if player 2 completes one, then player 1 must have completed one first. Additionally, player 2 never places an 'X' on the same row, column, or main diagonal as the previous move made by its opponent. This guarantees that player 2 does not inadvertently complete a row while mirroring player 1.

Does the Featured Net learn this strategy? It appears that it does not. Consider the array of maps in Figure 3. These maps are arranged on a 4×4 board. Each position on this larger board represents the opponent's move. This is also indicated by a red square on the smaller board. Each smaller board is a heat map of the Feature Net's responses. Darker colors designate the net's most common responses to a given board position. In 9 of these 16 positions, the net somewhat coincides with the known strategy: It places an 'X' on a square that does not share the same row, column, or main diagonal as its opponent's previous choice. But if we look at the other 8 positions, the net has a strong preference to place an 'X' on the same row, column, or main diagonal. It seems to me that the Featured Net's strategy is much more aggressive than the known one: It might be advancing on the same row, column, or main diagonal to severely restrict its opponent's available options.

We could use this same approach to study the behavior of much more competent ANN systems and much larger boards. But the mixed results in this section demonstrate that we may not find a clear description of a winning strategy within the ANN's behavior.

Further Work and Improvements

As I discussed, the Featured Net seems to converge on a winning strategy more easily (although its overall competence seems to decrease). I personally do not think this is enough improvement to consider this a success. In other words, this new net *still* plays larger Notakto boards poorly, despite being souped up with features encoding game knowledge. So I should reconsider the features that we use with the principle from (Lai 2015) in mind: The representation needs to map similar boards to similar scores.

Of course, the next step is to test the Featured Net within a more holistically competent Notakto player. This will reveal whether or not any gains these features give us generalize to larger boards and a more capable system. The obvious choice is to inject this ANN within the AlphaZero environment. But I could alternatively mend the deficiencies of my own system and use that instead. One advantage of doing so is that we have much more control over the choice of training boards than in AlphaZero's self-play model. Another is that my system is less computationally intensive, since AlphaZero takes much longer to converge than traditional training over a corpus.

While designing features, I noticed that the the key difference between Notakto and games AlphaZero plays (Go, Chess, tic-tac-toe, etc.) is that Notakto is an *impartial misère* game. That is, each player places the same token 'X', and the player to complete the requirements *loses*. We might hypothesize that AlphaZero's domain knowledge does not generalize to impartial or misère games. This is a reasonable conjecture from a mathematics point of view, since misère games tend to be much more intrinsically complicated than their normal-play variants (Plambeck and Siegel 2008). We can test this idea by training AlphaZero on impartial games ("normal-play" Nim), misère games (the losing variant of checkers), and games that are both (Nim, Dawson's chess).

References

[Chen et al. 2021] Chen, Z.; Wang, C.; Laturia, P.; Crandall, D.; and Blanco, S. 2021. How to play notakto: Can reinforcement learning achieve optimal play on combinatorial games?

[Chow 2010] Chow, T. 2010. Neutral tic tac toe. MathOverflow. URL:https://mathoverflow.net/q/24693 (version: 2021-03-15).

[Glorot, Bordes, and Bengio 2011] Glorot, X.; Bordes, A.; and Bengio, Y. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 315–323. JMLR Workshop and Conference Proceedings.

[Ioffe and Szegedy 2015] Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, 448–456. PMLR.

[Lai 2015] Lai, M. 2015. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*.

[Marcus 2018] Marcus, G. 2018. Innateness, alphazero, and artificial intelligence. *arXiv preprint arXiv:1801.05667*.

[Nair 2018] Nair, S. 2018. Alpha zero general (any game, any framework!). https://github.com/suragnair/alpha-zero-general.

[Plambeck and Siegel 2008] Plambeck, T. E., and Siegel, A. N. 2008. Misere quotients for impartial games. *Journal of Combinatorial Theory, Series A* 115(4):593–622.

[Plambeck and Whitehead 2013] Plambeck, T. E., and Whitehead, G. 2013. The secrets of notakto: Winning at x-only tic-tac-toe. *arXiv preprint arXiv:1301.1672*.

[Silver et al. 2017] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *nature* 550(7676):354–359.