Mastering Duck Chess by Self-Play with a General Reinforcement Learning Algorithm

Doug de Jesus

New York University
drd8913@nyu.edu

Abstract

Reinforcement learning has been used to train programs that can play even the most complex games at superhuman levels. Previous approaches that had success with chess were unable to achieve strong performance for the game of Go, which has a much higher branching factor. Through a combination of supervised learning and reinforcement learning from self-play, *AlphaGo* was finally able to beat the best human Go players. Further iterations of the algorithm showed that the method generalizes to a wide variety of games, including chess and shogi, and can learn solely through self-play with no human knowledge. Duck Chess is a variant of chess that adds a new piece jointly controlled by the players; this increases the branching factor to significantly higher than even that of Go. In this paper, I present the implementation of an algorithm in the style of *AlphaZero* that attempts to learn the game of Duck Chess through self-play. Due to the significant complexity, combined with computing resources constraints, the model was unable to achieve even a novice level of play. I examine the challenges caused by the high-dimensional action space and discuss possible solutions. The full code is available at https://github.com/dougyd92/alpha-zero-general-duckchess.

Introduction

Building a computer program that can outperform humans in strategy games such as chess and Go has long been an elusive goal in artificial intelligence. Even after chess had been conquered, the game of Go presented a difficult challenge due to its higher complexity. Go has a branching factor of around 250, meaning on average there are 250 moves to consider in a given position; in comparison, chess has a branching factor of around 35 (Burmeister and Wiles 1995).

In 2016, *AlphaGo* became the first program to defeat a professional Go player (Silver et al. 2016). Soon after, new iterations of the algorithm showed even greater success, not just at Go but at chess, shogi, and even Atari video games (Schrittwieser et al. 2020). The first version of *AlphaGo* used both supervised learning on examples of games between human experts and reinforcement learning through repeatedly playing games against itself. Later iterations (*AlphaGo Zero* etc.) proved that self-play alone was sufficient, meaning the algorithm could generalize to many challenging domains without relying on human expertise (Silver et al. 2017b), hence the "Zero" moniker.

All of the variations share the same central idea, though they differ slightly in the details. A deep neural network is trained to output a policy, which predicts the probabilities of the available moves, and a value, which estimates the expected outcome of the game from a given position (Tomašev et al. 2020). The policy and value predictions are used to guide and narrow down a Monte Carlo Tree Search (MCTS), which simulates looking ahead and many possible move sequences and computing move probability and state value estimates in aggregate that are much stronger than the individual predictions of the neural network (Silver et al. 2017a). The network is in turn trained to more closely match these stronger predictions.

AlphaZero has previously been used to successfully study chess variants that do not have the centuries of theory that classical chess has (Tomašev et al. 2020). Duck Chess is a new variant that was invented in 2016 and grew in popularity in 2022 (Paulden 2022). The rules are similar to classical chess, but with an additional piece called the duck, which acts as a blocker. This new piece cannot be captured nor moved through, is shared by both players, and must be moved to a new square at the end of each player's turn. This simple change adds a huge degree of complexity and variety to the game. The standard "book" opening lines in classical chess are no longer applicable, as each player can block off the square that their opponent would like to move to. Check and checkmate work completely differently, as a player is allowed to leave their king under threat then use the duck as a shield. The number of possible moves is increased dramatically, as every standard chess move is then followed by one of 64 possible "duck moves". High level human chess players have not yet figured out established strategies for Duck Chess. Because expert domain knowledge is not available anyway, Duck Chess is a perfect candidate to test out the generalized *AlphaZero* algorithm.

Methods

As in (Silver et al. 2017b), a single neural network is used which outputs both the policy and the value. This consists of a convolutional block followed by 12 residual blocks, the output of which is passed to both the policy and value heads.

Listing 1: Model architecture

```
Input: 8(width) x 8(height) x 19(
    channels)
Convolution with 256 filters, 3x3 kernel
    , stride 1, padding 1: 8 x 8 x 256
Batch Normalization
ReLu

12 of the following residual blocks:
Convolution with 256 filters, 3x3 kernel
    , stride 1, padding 1: 8 x 8 x 256
Batch Normalization
ReLu
Convolution with 256 filters, 3x3 kernel
    , stride 1, padding 1: 8 x 8 x 256
Batch Normalization
Skip connection
ReLu

Policy head:
Convolution with 2 filters, 1x1 kernel,
    stride 1, padding 0: 8 x 8 x 2
Batch Normalization
ReLu
Fully connected layer with output shape
    8 x 8 x 73 x 8 x 8
Softmax
Output: 8 x 8 x 73 x 8 x 8

Value head (parallel to policy head):
Convolution with 1 filter, 1x1 kernel,
    stride 1, padding 0: 8 x 8 x 1
Batch Normalization
ReLu
Fully connected layer to a hidden layer:
     256
Relu
Fully connected layer to a hidden layer:
     1
Tanh
Output: 1
```

The detailed model architecture is shown in Listing 1.

The input is represented as an 8x8x19 image stack, with the location in each plane corresponding to one square on the chess board, similar to (Silver et al. 2017b). 6 binary features planes are used to denote the presence of each of white's pieces on the board (pawn, rook, knight, bishop, queen, king); 6 planes are similarly used for black's pieces. 1 binary feature plane indicates the location of the duck piece. The other 6 layers each have uniform value and represent some attribute of the game state: 1 for to indicate which is the current player, 4 to indicate whether each player is eligible to castle kingside or queenside, and 1 to count the number of moves taken thus far (games were terminated in a draw after 300 moves). The input is normalized so that the input images are similar for both white and black (the current player's own pieces are in planes 1-6, their own pieces start on the bottom two rows).

The policy output has shape 8x8x73x8x8, for a total size of 299,008. The first three dimensions encode chess moves in the same format as in (Silver et al. 2017b). First, the square with the piece to be move is indicated. Then, one of 73 possible encoded movements are indicated. The first 56 correspond to "queen-style" moves, i.e. moving in a straight line either vertically, horizontally, or diagonally by 1 to 7 squares. The next 8 encoded movements are "knight-style" moves, i.e. moving two squares vertically and one square horizontally, or vice versa. The final 9 encoded movements correspond to the 3 possible underpromotions when a pawn reaches the last rank by moving straight, capturing left, or capturing right.

The final additional 8x8 dimensions are unique to Duck Chess, and indicate the square onto which to move the duck after making the normal chess move. This greatly increased the number of possible actions from 4,672 to 299,008, which proved to be problematic. In a standard chess game, of the 4,672 possible actions that could be described in this way, only around 35 are valid in any given position (this is the branching factor of chess). However, as the duck has no restrictions in how it can move other than needing to be placed in a new, empty square, the number of possible duck moves is always at least 31, meaning the new branching factor is around 1,000, which is five times higher than Go.

The self-play training framework was based on prior work by (Thakoor, Nair, and Jhunjhunwala 2017), which is an open-source, single-threaded implementation of the *AlphaGo Zero* algorithm. Several modifications were made to this implementation for performance reasons and to bring the algorithm more in line with the later *AlphaZero* algorithm; namely, one model is continuously trained, rather than comparing each iteration and taking the current best, as this was shown in (Silver et al. 2017a) to be unnecessary.

The tree search has access to the game rules in order to traverse the possible move sequences. Namely, it can see what the valid moves are from a given state, and it can see what the next state would be after applying a given action. Recall also that the structure of the input and output implicitly match the grid structure of the board and the action space. Other than those pieces of information, the system does not have any domain knowledge, strategies, nor heuristics.

Starting from a new model with random initial weights, games of self-play were conducted using 30 MCTS simulations per move, roughly equivalent to 1.2s thinking time per move. After the tree search, probabilities π are returned proportional to the visit count of each move from the root state. Exploration is controlled by a temperature parameter τ; for the first 15 moves of each game, $\tau = 1$ so that moves will be selected proportionally to their visit count. After that, τ is set to 0 so that simply the most visited move is selected (Silver et al. 2017b).

The results of each tree search and eventual move selection are stored as training examples as (s_t, π_t, z_T); the value of z is filled in after the end of the game, and indicates whether the current player at that timestep went on to win ($z = 1$), lose ($z = -1$), or draw ($z = 0.1$).

Self-play was conducted in batches of 10 games between training iterations. In each training epoch, the model was trained to minimize the loss function

$$(p, v) = f_\theta(s), \quad l(\theta) = (z - v)^2 - \pi^T log(p) \qquad (1)$$

i.e. the sum of the mean-squared error over v and the cross-entropy loss over p (Silver et al. 2017b). The training was done using stochastic gradient descent, with a learning rate of 0.01 and momentum 0.9, and a batch size of 64. Each batch of data is sampled uniformly from the most recent 20 self-play games (*AlphaGo Zero* looks at the most recent 500,000 games, but there aren't anywhere near that many here).

The final model was trained over 18 epochs on a total of 18,000 training examples.

Results

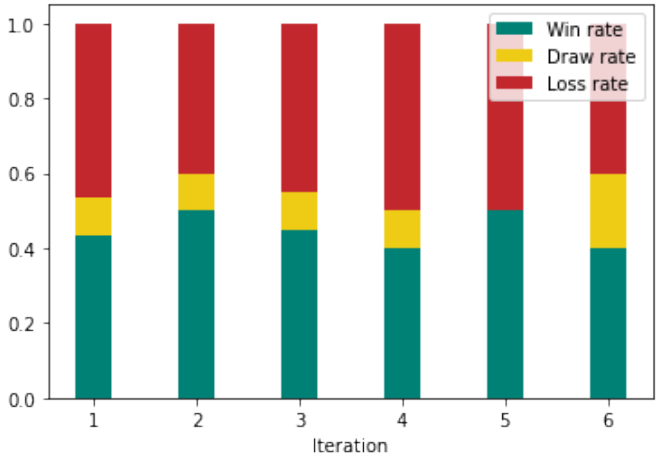

Figure 1: Comparison against random move selection as a baseline

With such a small number of training samples compared to the complexity of the problem space and the size of the model, no noticeable improvement was found in the model's ability to play Duck Chess.

Unlike standard chess, which has many established engines and tournaments in which they are compared, no baseline engine exists for Duck Chess. Instead, successive iterations were compared against an agent which makes purely random moves as a baseline. This random agent selects from among the valid actions for the current game state with equal probability, but does not use MCTS nor any policy predictions. Figure 1 shows the result of comparing the first 6 model iterations against this baseline. Each model played 20 games against the random agent, with each side playing as white 10 times and as black 10 times. τ was set to 0, so that the best move is chosen rather than exploring. One would hope to see the win rate increase as the model learns from more and more examples, but here each model seems to perform no better than random. In any case, the sample size is too small to make any inferences.

For qualitative analysis, two human volunteers, with Elo rating 1000 (beginner) and 2000 (expert), played against the final model in best-of-five matches. In each case, the model lost within the first ten moves without offering any resistance. It seemed to favor pawn moves (advancing by only a single square) and never made any captures. A sample of one of these games is shown in Listing 2.

Listing 2: Sample game, human (white) vs. best model(black). Duck moves denoted by @.

```
1  d4@e6  g6@g4
2  Bf4@g7 d6@c5
3  c3@d7  f6@c5
4  Qa4@d7 Na6@c5
5  Qxe8#
```

The large size of the action space and thus the policy output created problems that made it difficult to quickly generate training samples. With nearly 300,000 moves to represent, the policy prediction at each node in the MCTS tree took up 1.2mb, severely limiting the number of MCTS simulations that could be done for each move. Even 30 simulations took around 1.2 seconds, though this number increased as the number of turns in a game increased as the MCTS tree grew in size, and after a certain point the system become memory starved. In the early iterations, the model makes mostly random moves, so many games go on for hundreds of moves without a winner, resulting in individual games taking upwards of 20 minutes. For comparison, when training *AlphaZero* on highly specialized hardware, 44 million self-play games were completed in just 9 hours.

Further exacerbating this problem, the MCTS search is unable to explore a meaningful proportion of the action space, resulting in very shallow evaluations and failing to gain any benefit. In training *AlphaZero* for Shogi, (Silver et al. 2017a) note that the 11,259 possible moves there presented significant computational complexity, so trying to explore an action space twenty times that was perhaps overly ambitious on consumer hardware.

Conclusion

With the representation of the input and output as described here, the implementation of the *AlphaZero* algorithm was not enough to overcome the complexity of Duck Chess. While the algorithm is certainly powerful, not enough credit is given to the extremely powerful and specialized hardware that it was running on. The implementation described here may have fared better if it were running on a high-performance cluster rather than a consumer laptop, and if it were rewritten to be parallelized instead of single-threaded.

Even then, the high dimensionality of the action space would likely not be ideal, as it prevents taking full advantage of MCTS depth. Rather, it would be better to model the normal move and the duck move as two separate steps taken independently; this would require slight alterations to the training algorithm, which currently assumes that players alternate turns, but it would not be a significant change. Depending on how the moves are encoded, the model may lose some of the implicit knowledge of how pieces move, but if the principle of learning from zero domain knowledge holds,

that shouldn't matter in the end.

References

Burmeister, J.; and Wiles, J. 1995. The challenge of Go as a domain for AI research: a comparison between Go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, 181–186.

Paulden, T. 2022. Duck Chess. https://duckchess.com/. Accessed: 2022-11-30.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; Lillicrap, T.; and Silver, D. 2020. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, 588(7839): 604–609. ArXiv:1911.08265 [cs, stat].

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.

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

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(7676): 354–359.

Thakoor, S.; Nair, S.; and Jhunjhunwala, M. 2017. Alpha Zero General. https://github.com/suragnair/alpha-zero-general.

Tomašev, N.; Paquet, U.; Hassabis, D.; and Kramnik, V. 2020. Assessing Game Balance with AlphaZero: Exploring Alternative Rule Sets in Chess. ArXiv:2009.04374 [cs, stat].