# TorusZero, a Torus playing Engine

Jules Johnson

May 19, 2025

## 1 Torus, the game

Torus is a board game inspired by Go, invented by mathematician Emma Joe Anderson and physicist Erin Ewart in 2018. In the time since its invention, Torus has gathered a small community of enthusiasts, and many aspects of its strategy and theory have been explored and written about. However, despite this enthusiasm, there have so far been no completed efforts to build a computer program that plays Torus. With this in mind, the aim of this project is to create a Torus engine that can compete with human players. To establish the motivations and goals of this project, we first explain the exact rules of Torus.

Torus is played on a square board, on which alternating black and white stones are placed on the intersections of a grid by each player, starting with black. The player who is currently placing a stone is called the *active* player, and the other player is called the *passive* player. Similarly, the active player's stones are called active stones, and likewise for the passive player. Although any size of board is technically possible, Torus is almost always played on a 9x9 board. Each point on the board is labeled as in Go, with the columns named after capital letters, and the rows named after numbers, so that each point can be expressed with a letter and a number, such as B3 or I7.

There are two objectives that may be accomplished by placing a stone in Torus. First and foremost, The active player wins the game by completely surrounding any one of the passive players stones in all four cardinal directions.

Second, the active player may remove the passive players stones by flanking them diagonally on two opposite sides. Any number of stones may be captured like this, so long as they all lie on the same diagonal line as the stones capturing them.

Figure 1: A game being won by black

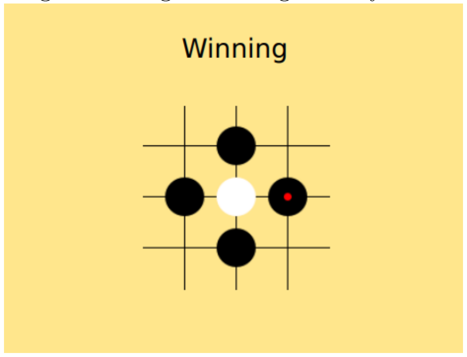


Figure 2: If black places in the location pictured here, the white stone is removed.

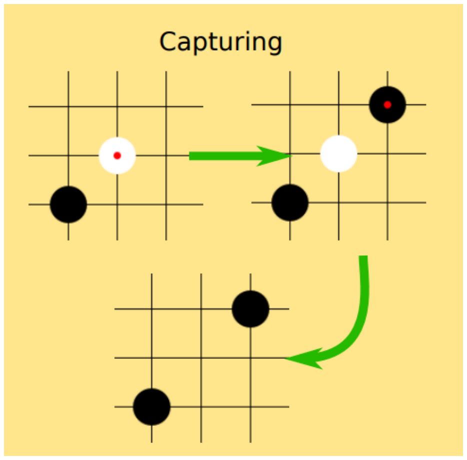
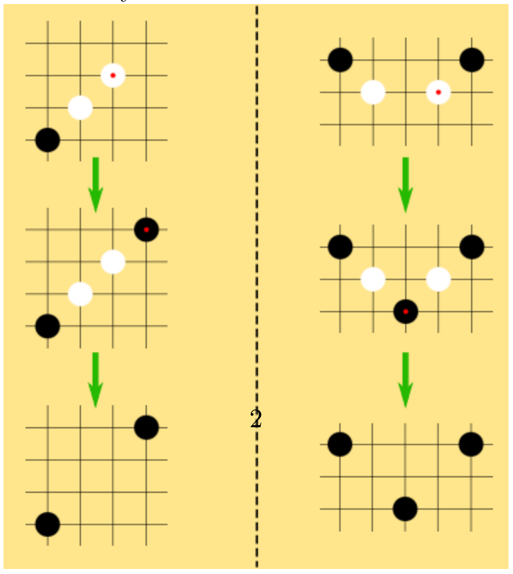
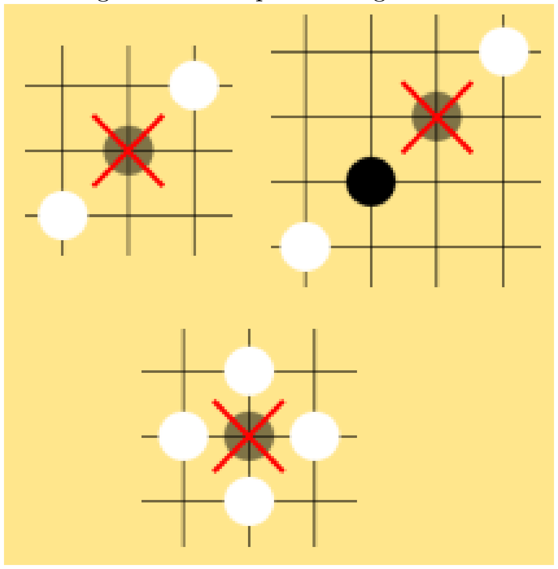


Figure 3: It is possible to remove multiple stones in a single row, or even in multiple rows simultaneously.



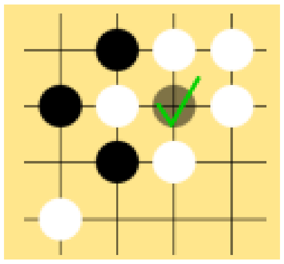
Many moves in Torus are illegal. Of course, it is illegal to place a stone where one is already present. Additionally, The active player may not place a stone where it would be immediately removed by the passive player, or were it would cause the passive player to win.

Figure 4: Examples of illegal moves



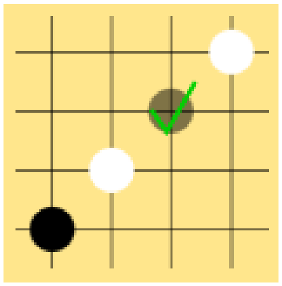
There are two exceptions to this rule. First, it is always legal to play a move that wins the game, even in situations where the placed stone might be captured, or that cause an apparent victory for the passive player.

Figure 5: This move causes black to win, because black is the active player, even though it appears illegal.



Second, the active player may play a stone into a location where it would be captured, if that move also removes the stone or stones that threaten the new move.

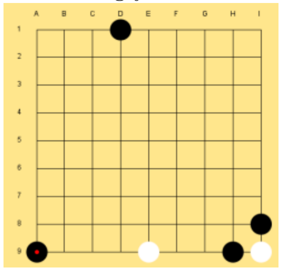
Figure 6: This move is legal, because the white stone that threatens it is removed first.



Finally, a move is never legal if it causes the board to enter a state that is identical to any state it has had previously. This is similar to the “ko” rule in Go, but it is more restrictive. In Go, it is illegal to return the board to the position it had in the previous turn [2], but in Torus, it is illegal to return the board to any previously held position. Therefore, this rule is referred to as *super-ko*.

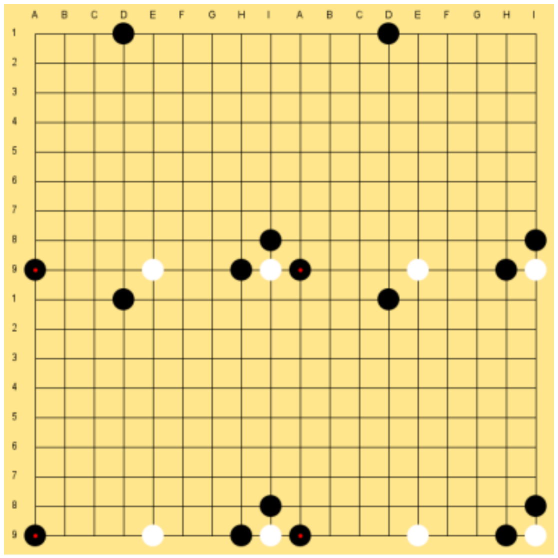
Finally, as the name of the game implies, Torus is played on a torroidal board, in which the edges connect to their opposite sides. This means that stones in the 9th row are adjacent to stones in the first row if they share a column, and stones in the 9th column are adjacent to stones in the first column if they share a row. For example, a board that looks like this:

Figure 7: A seemingly non-torroidal board.



Actually looks like this:

Figure 8: A "wrapped around" version of the previous board. Here, the stone placed on I9 is adjacent to A9, and the game would be over if black were to play at I1



The two remaining rules are obscure and almost never applied. They are:

- The active player cannot capture a continuous line of 8 passive stones by playing a single stone in the gap between them. A capture requires an active stone currently being placed, and a distinct active stone that had been placed previously.
- If there are no available legal moves for the active player, the passive player wins.

The complete set of rules, as well of a starting exploration of the theory and strategy is available on Anderson's website at *https://endless.ersoft.org/a-beginners-guide-to-torus-the-board-game/*.

## 2 Goals of the Project

Here, we will exact goals for this project that can inform future design decisions. First, we hope to build an effective and powerful program that can outperform human players. At present, the most experienced players in Torus are its creators, who nonetheless have only 5 years of experience. Torus is much newer than Chess or Go, and so there has been very little time to develop strategy or theory that can be compiled in literature. Because of this, even top human players are likely playing at a level far bellow what is possible, and it should be possible to create a machine that can outperform them.

**Goal 1** *To be able to outperform human players*

Although Torus's community has observed many strategies to be effective in human play, it is not known with certainty with of these are found in optimal play. Therefore, our second goal is to build this program without any pre-coded sense of strategy. This will be similar to the approach used by AlphaGo Zero, a Go playing program that is notable for not using any training data from human games. This will allow for games played by our Torus Engine to be useful tools for humans to analyze for new and ideal strategies.

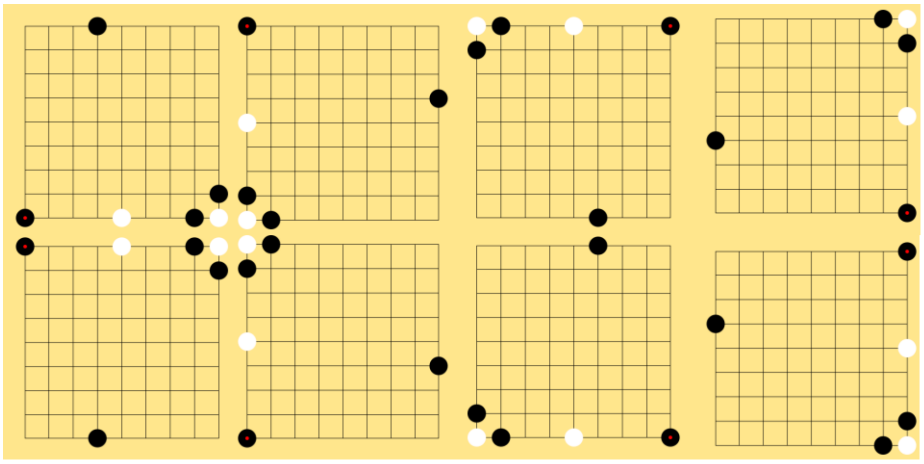
**Goal 2** *To be built without training data from human games, and without specific hard-coded pieces of human-discovered strategy*

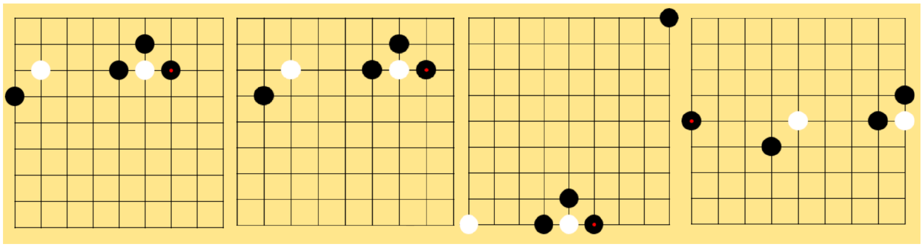
Many Chess and Go engines are able to tell users the predicted effectiveness of any given move, or the advantage to one player of any given board state, or both. To give an example, chess.com uses the popular Chess engine Stockfish to analyze their games [3], and can give move-by-move feedback on games played through their website [4]. This is an invaluable tool for human players wishing to improve their performance. Rather than simply outputting the expected best move for a given position, we would like our program to be able to rank the expected effectiveness of a move, for the benefit of human players who wish to learn.

**Goal 3** *To be able to give insight on the effectiveness of a human player's moves*

Torus is a game with many symmetries. In a game of Go, each board state is strategically equivalent to 8 board states: 4 that are rotations of the original board ($0°$, $90°$, $180°$, and $270°$), and an additional 4 that are rotations of the original board, plus a mirroring over the x or y axis. These 8 symmetries can also be thought of as the 8 possible combinations of mirroring over the x-axis, the y-axis, and one of the boards diagonals. By contrast, each board state in a game of Torus is equivalent to any board state made by shifting the entire board up, down, left, or right any amount. Each boardstate in Torus is therefore strategically equivalent not only to the 8 boardstates made by rotation and reflection, but 81 board states made by translation, for a total of $8 \cdot 81 = 648$ board states that are strategically equivalent. Some of these equivalencies are illustrated bellow.

Figure 9: Above: A board on the upper left, as well as all of the boards strategically equivalent to it by rotation and reflection.
Below: Four of the boards strategically equivalent by translation. A red dot is added for ease of reading





We wish for our program to give equivalent values to each of these equivalent board states. Human players will often adjust their view of a board to an equivalent transformation to get a better view of the boards structure, and it would be unacceptable for our engine to view these differently.

**Goal 4** *To play equivalent moves in strategically equivalent boardstates*

Of all the broad techniques for building a game playing engine, machine learning and neural networks are perhaps the most exciting. They are not the only tools available; Stockfish, the highest ranking chess engine, does not use neural networks at all. Instead, it searches the game tree for moves that lead to advantageous positions, according to a hand-crafted function that gives the probability of each player winning based on the boardstate [5]. AlphaGo, the Go engine developed by Google's subsidiary DeepMind, does use a neural network, but also uses a Monte-Carlo tree search, like Stockfish [6].

At present, The goal will be set to only use neural nets as opposed to game tree searches, as this narrows the range of tasks to be performed immediately to one that is useful to learn and practice. World chess champion José Raúl Capablanca is reported to have once said, “I see only one move ahead, but it is always the correct one” [1].

**Goal 5** *To avoid searching the game tree in lieu of rating moves by neural net*

In total, our goals for our Torus program are :

1. To be able to outperform human players
2. To be built without training data from human games, and without specific hard-coded pieces of human-discovered strategy
3. To be able to give insight on the effectiveness of a human player’s moves
4. To play equivalent moves in strategically equivalent boardstates
5. To avoid searching the game tree in lieu of rating moves by neural net (this goal may be removed in future versions)

## 3 Plans for Construction

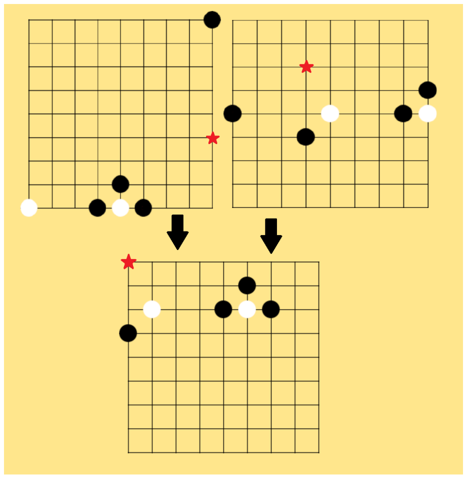
Due to requirement 3, it is necessary that this AI gives a rating to each move and selects the best one, rather than giving a single move it predicts to be the best. There are two apparent approaches to this goal:

1. Given any given space on the board, the program gives that spot a score. The program then plays a move at the highest ranking spot.
2. Given any board state, the program outputs a score. The program then selects the move for which the resulting board scores the highest.

Of these two, the first is likely easier to implement. The second option would more easily allow for searching a game tree, so it might be more effective in future projects or versions that drop that goal. However, scoring board states in a way that is completely equivalent across all that boards symmetries is very difficult, so for now, the AI will rank moves.

To make eliminate any possible differences in rankings of moves on boards that are equivalent under translation, we may find a way of standardizing boards. Given any board, and a space on that board to be ranked, we might first shift the board so that the spot of interest is in the top left. Therefore, a move can be scored equivalently regardless of what specific isomorphic boardstate is used.

Figure 10: Here, two equivalent moves on different boards are moved to a single move on a standardized board



We choose to create a program that scores specific moves, rather than board states, as well as a program that can shift boards so that the move in question is always located at the top left. Our first two classes, shown in pseudocode with some of their functions are:

```
Board()

    move_point_to_top_left(point):

        for i in point's x coordinate
            shift board left
        for i in point's y coordinate
            shift board up

MoveRater()
    score_point(point)
```

The actual number of methods in each object shall be significantly greater than those shown here. However, for brevity and ease of understanding, only the most important are shown in the pseudocode in this paper, and any code inside each method will be collapsed after its first appearance.

Our move rater also needs to be able to rank points equivalently across boards that are flipped or rotated versions of each other. Rather than attempt to find a single version of the board that is rated in all cases, MoveRater can simply sum up its scores for all 8 equivalent forms of the board.

```
Board()
```

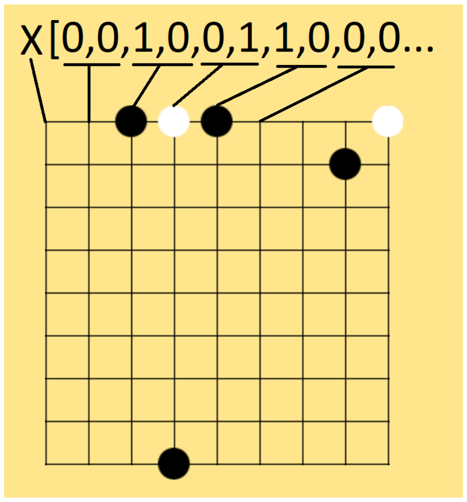
```
        move_point_to_top_left(point)

        rotate_board()

        flip_board()

    MoveRater()
        score_point(board, point)

        sum_scores(board, point):

            total = 0

            for i in {0,1,2,3}
                total = total + score_point(board, point)
                board.rotate_board()

            board.flip_board

            for i in {0,1,2,3}
                total = total + score_point(board, point)
                board.rotate_board()

            return total
```

Because the MoveRater will rank moves using a neural net, it will need to be able to intake a board as a one-dimentional array of numbers. The natural idea is to create an array of numbers with length 81, in which an empty space is represented by 0, a white piece is represented by 1, and a black space is represented by -1. However, a "good" or "bad" move depends wildly on if the person playing the move is placing a white or black stone, meaning that a neural net that recieves this data would likely only be accurate half the time. The obvious solution is to normalize the colors of the board, so that that active player can always consider the black stones as their own. This is an improvement, but there are still some problems with this approach.

First, there is no reason to include the top left space in this array, because it can safely be assumed to be empty if the program wishes to play there. Therefore, our array can be shrunk down to only 80 elements. More importantly, it is not accurate to model a passive stone as the opposite of an active one. As an example, in a board without any other stones in play, it is a very bad idea to place a stone next to an opponents stone, as this leads to an inevitable loss. However, it does not follow that it is a very good idea to place stones immediately next to your own. Active and passive stones have to be considered separately, and not merely as opposites of each other.

Because of this, we will represent the board with an array of 160 numbers, consisting of 80 pairs representing each available spot on the board. Every point on the board can be given a unique number $n$, ranging from 0 to 79. If a point has a white stone, the array will have a 1 at position $2n$, and a 0 at position $2n+1$. If a point has a black stone, the array will have a 0 at position $2n$, and a 1 at position $2n+1$. If a point is empty, both positions on the array will have a 0.

Figure 11: The method by which a board is outputted to an array. The X represents the top left point, which is assumed empty, and therefore is not represented in the array.



```
Board()
    move_point_to_top_left(point)

    rotate_board()

    flip_board()

    Reverse_colors():

        for spot in board:
            if spot has a black stone:
                give spot a white stone
            if spot has a white stone:
                give spot a black stone

    output_board_as_string():

        final_string = []
        for spot in board:
            if spot is not at the top left:
                if spot has a black stone:
                    final_string.append([1, 0]
                if spot has a white stone:
                    final_string.append([0, 1]
                else:
                    final_string.append([0, 0]

MoveRater()
    score_point(board, point)

    sum_scores(board, point)
```

Of course, the most important function in this program is the method that actually provides a score for a given point. In order for a point to be rated, the board is first "normalized" by shifting that point to A1. Each layer of neurons will be multiplied by a matrix of numbers between -1 and 1, and each value in the resulting neuron layer will have a bias added to it. To increase the flexibility of our system, we will let the biases for each neuron be freely varied within a certain range, rather than the usual approach of giving identical biases to each neuron.

Although the standard neural net has multiple output neurons, ours will have just one: the final score for a move. Because of this, in addition to the weights and biases that are used ubiquitously in neural nets, our neural net will also have a "power" associated with each neuron. Each neurons value will be multiplied by its power, before it is fed into the matrix of weights that activate the next layer of neurons. The justification for this is that we want a neural net that is able to set priorities. For example, if there exists a neuron that activates when a move is likely to cause a loss, this should have much higher priority than a neuron that activates when a move is likely to allow a capture. Powers also will be limited to fall within a pre-determined boundary, as with weights

Because of the decision to give each neuron a power, we will choose to use the Sigmoid as our activation function, rather than ReLU, leaky ReLu, or any other activation function. Using an activation function is required to prevent linearization. However, using ReLU or any variation could cause neuron values to grow out of control. Because of this, the use of powers will act as a replacement for ReLU.

The previous three paragraphs have established a handful of plans for our project that lie outside the standard strategies for developing Neural Nets. Justifications were given for these changes, but it's completely possible that they'll prove to be unideal as the project is better understood. Thankfully, each of these decisions can be easily reversed or turned off and on. This will allow for experimentation later on.

In total, each non-entry neuron in our program will be created by multiplying the previous layer of neurons by an array of weights, adding a bias, taking the sigmoid of the result, and then multiplying by the power.

This construction allows for

```
CONVOLUTION_SETS = [....]
BIAS_RANGE = [min,max]
POWER_RANGE = [min,max]

Board()
    move_point_to_top_left(point)
    rotate_board()
    flip_board()
    Reverse_colors()
    output_board_as_string()

MoveRater()
    score_point(board, point):

        board.move_point_to_top_left(point)
        if active color is white:
            board.flip_colors()
        current_neurons = board.output_as_sting()
        for size in convolution sets
            current_neurons = matrix_multiplication(current_neurons,
                                appropriate weights)
            for i in current_neurons:
                i = i + appropriate bias
                i = sigmoid(i)
                i = appropriate_power * (i)
        total = matrix_multiplication(current_neurons,
                    appropriate weights)

    sum_scores(board, point)
```

The other half of this project is to create a method by which to train a MoveRater that completes our first goal. There are a few obstacles that prevent the normal method of neural net training. First, we do not have an appropriate set of training data. Although Anderson and Ewart have kept a remarkably well curated set of played games, this set is not nearly large enough to use as the mountain of training data needed for machine learning. Additionally, to use this data would go against goal 2, which states that we want to avoid using human made strategies.

The second obstacle is that back propagation is impossible. Not only does back propagation require training data, but it also requires that the network has multiple outputs.

One solution is the one used by AlphaGo Zero and many other machine learning projects. The program can learn by playing against itself. In particular, there will be a single-elimination tournament of 32 MoveRaters, and the victors will seed a follow-up tournament. After many repetitions of this tournament, the victors will be much more powerful than they were when they started.

The top four from any tournament will be used as seeds for the next starting bracket. Although there is no randomness built into the MoveRaters themselves, the starting bracket itself ought to be scrambled. Because of this, the outcome of a tournament is somewhat random, so allowing MoveRaters who have come close to first to return in future tournaments seems like a good idea.

Of course, in order for MoveRaters to improve, we have to allow them to mutate and change. Each MoveRater will then have a method that will "salt" itself by randomly adjusting its values a small amount. This necessitates an extra global variable that can be adjusted, which we will call SALTCRAZINESS, which can control how severely salting will affect a MoveRater.

It would also be a good idea to allow successful MoveRaters to repeat their previous changes, in the hopes that a previous successful change could lead to more success when repeated. This requires then that each MoveRater also contains data on their parent.

```
CONVOLUTION_SETS = [....]
BIAS_RANGE = [min,max]
POWER_RANGE = [min,max]
SALT_CRAZINESS = 0.003

Board()
    move_point_to_top_left(point)
    rotate_board()
    flip_board()
    Reverse_colors()
    output_board_as_string()

MoveRater()
    score_point(board, point)

    sum_scores(board, point)

    normal_distribution(SALT_CRAZINESS):
        return a normal distribution with mean 0 and standard deviation SALT_CRAZINESS

    salt_self(severity):

        for i in self.weights:
            i = i + severity * normal_distribution()

        for i in self.biases:
            i = i + severity * size of BIAS_RANGE * normal_distribution()
        for i in self.powers:
            i = i + severity * size of POWER_RANGE * normal_distribution()

    repeat_change(severity):

        for i in self.weights:
            i = i + severity*(i - i’s previous value)

        for i in self.biases:
            i = i + severity * size of BIAS_RANGE * (i - i’s previous value)

        for i in self.powers:
            i = i + severity * size of POWER_RANGE * (i - i’s previous value)
```

In order for a tournament to determine a winner between two move raters, It must be able to keep track of a game. This requires a Gamestate object that is able to map new moves onto a board, and update that board accordingly. Although we already have a board object, we need an object that can keep track of many boards in order to check for if a move violates ko. The board object exists primarily to allow a board to be fed to a MoveRater.

Our Gamestate object will be set to only keep track of a game. This is useful, because the board object shifts around often as it’s normalized. By contrast, a Gamestate will be kept consistent. The functionality to find which MoveRater would beat another will be part of the MoveRater class itself.

The gamestate class will have a method to check if moves are legal, as well as one to apply moves to the board. This is straightforward, and does not need to be explained in any great detail. The already explained rules of torus are precise enough to determine for the reader what moves are legal, and what effects playing a move has on the board.

```
CONVOLUTION_SETS = [...]
BIAS_RANGE = [min,max]
POWER_RANGE = [min,max]
SALT_CRAZINESS = 0.003

Board()
    move_point_to_top_left(point)
    rotate_board()
    flip_board()
    Reverse_colors()
    output_board_as_string()

MoveRater()
```

```
    score_point(board, point)
    sum_scores(board, point)
    normal_distribution(SALT_CRAZINESS)
    salt_self(severity)
    repeat_change(severity)

    choose_move(gamestate):

        best_move = [0,0]
        record = -9999999999
        for spot on gamestate.board:
            if spot is a legal move:
                if sum_scores(spot) > record
                    record = sum_scores(spot)
                    best_move = spot
        return best_move

    would_beat(otherMR):

        Game = normal starting game
        while True
            if winning_move_exists():
                if active color is black:
                    return True
                else:
                    return False
            else: if active color is black:
                point = self.best_move(game’s current board)
                game.apply_move(point)
            else: if active color is white:
                point = otherMR.best_move(game’s current board)
                game.apply_move(point)

Gamestate()

    legal_move(point)

    apply_move(point)

    winning_move_exists()

        for point in the board:
            if point would win the game:
                return True
        return False
```

Finally, we wish to build a Tournament object that can determine the most effective of the MoveRaters put inside it. As mentioned, this will be done this via a single elimination tournament, in which the top four performing MoveRaters are chosen to seed the next tournament. Each successful MoveRater will be allowed a certain number of child MoveRaters to appear in the next tournament, which will be formed from a combination of salting the parent, and repeating the parents previous changes.

In the produced bracket for a following tournament, it seems reasonable to allow 4 children from each semi-finalist, 4 additional children for each finalist, and 4 additional children for the winner. Because salting a MoveRater is totally random, it is completely reasonable to include multiple children from a single parent who have each been salted the same number of times.

This totals to exactly 28 children. One of the big pitfalls involved in machine learning is that the process of optimizing a neural net might fall into a local minimum, rather than an absolute maximum. Out of fear that we might fall into a pattern of seeking a sub-optimal MoveRater, we will also include 4 randomly generated move raters as well. This will also help to ensure that MoveRaters are able to play effective games in a variety of situations, rather than just against themselves.

```
CONVOLUTION_SETS = [...]
BIAS_RANGE = [min,max]
POWER_RANGE = [min,max]
SALT_CRAZINESS = 0.003

Board()
    move_point_to_top_left(point)
    rotate_board()
    flip_board()
    Reverse_colors()
    output_board_as_string()

MoveRater()
    score_point(board, point)
    sum_scores(board, point)
    normal_distribution(SALT_CRAZINESS)
    salt_self(severity)
    repeat_change(severity)
    choose_move(gamestate)
    would_beat(otherMR)

Gamestate()
    legal_move(point)
    apply_move(point)
    winning_move_exists()
```

```
Tournament()

    fill_brackets():

        starting_bracket = current_bracket
        While winner is undecided:
            next_bracket = []
            players = []
            for MR in current_bracket:
                add MR to players
                if players has two MoveRaters in it:
                    Those two Move Raters play each other, and the
                    winner is added to next_bracket.
                    players = []
            current_bracket = next_bracket

    create_child_bracket():

        child_bracket = []
        for MR in semifinalists:
            add MR to child_bracket
            add MR.salt_self(1) to child_bracket
            add MR.repeat_change(1) to child_bracket
            add MR.repeat_change(1).salt_self(1) to child_bracket
            add random MR to child_bracket
        for MR in finalists:
            add MR.salt_self(1) to child_bracket
            add MR.salt_self(2) to child_bracket
            add MR.repeat_change(1).salt_self(1) to child_bracket
            add MR.repeat_change(1).salt_self(2) to child_bracket
        for MR in winners:
            add MR.repeat_change(1).salt_self(1) to child_bracket
            add MR.repeat_change(2).salt_self(1) to child_bracket
            add MR.repeat_change(3).salt_self(1) to child_bracket
            add MR.salt_self(2) to child_bracket
```

Our main training program runs 100 tournaments like this, periodically outputting the final game to a txt file.

Of course, the pseudocode shown here is wildly simplified from the actual implementation of code in the project. The code for the project is kept well commented, so any confusion can be resolved there.

## 4 Results to the First Draft of Code

Unfortunately, the results of the first draft of this project have been mixed. To start with the good news: There is now a Torus playing engine that plays equivalent moves in strategically equivalent positions without using training data or game trees. However, successes with the first and third goal are much more mixed.

First, The rankings of moves is not especially useful. Many MoveRaters are set up so that every value they produce is negative. It might be nice to normalize a raters scores, so that the scores given line up between 0 and 100

However, the larger issue is that the Torus engine developed is not especially effective. Although the engine as exists does seem to have a solid grasp of how to capture, it doesn't seem to understand much else. Regrettably, this means that a lot more work will be required before this project can be used to inform human play.

## 5 Possible Improvements for a Future Draft

It might be useful to switch from rating moves to rating boards. First, this can still be used to rate the effectiveness of moves, by measuring how much it improves the board state. But second, we have a suspicion that having more stones on the board is better most of the time. This would be a good way to make it easier for a neural net to realize this quicker.

Second, it might be nice to begin by training MoveRaters against programs that, for example, only try to capture stones. Training MoveRaters against a simple but broadly effective strategy might be a good way to quickly raise their effectiveness to a good base level.

Third, it would likely be a good idea to relax our goal of avoiding game tree searches. This goal is non-standard for Go and Chess engines, and dropping could streamline the process greatly.

## 6 Version 0.1.2

### 6.1 changes

I've decided to start naming the specific versions of this program. With version 0.1.2, first and foremost, a small bug that sometimes viewed an illegal move as legal has been removed. There was an edge case when a move might be played that would be captured immediately, but that's been removed.

More significantly, there is now a small amount of tree searching. MoveRaters will now play moves that will prevent a loss next turn, as well as automatically play moves that guarantee a win the turn after next. This is done by having the gamestate object give each MoveRater a list of moves to consider.

For the purposes of this project, a stone is in "peril" if it has 3 stones of the opposite color surrounding it, and no stones of the same color around it. A stone is ”in danger” if it has 2 stones of the opposite color and at least 1 empty space touching it, and a stone is ”threatened” if it has 1 stone of the opposite color and at least 1 empty space touching it. These terms are not in common use among players of Torus, but will be useful for this project in particular.

If the active player has a stone $s$ that is in peril, there are only two ways to avoid a loss on the opponents next turn. One is to place a stone at the empty spot adjacent to $s$, although this can usually be undone by the opponent next turn, and so is not ideal. The other option is to capture any of the stones surrounding $s$, if possible. Because of this, checking to see if a stone in peril is saveable is quick and easy.

Similarly, it is easy to find which moves lead to inescapable peril for the opponent. The only possible moves that create peril in this way are the ones that are played adjacent to passive stones that are in danger. As mentioned, it is easy to confirm which of these moves lead to peril that cannot be escaped. Therefore, games can be massively sped up by automating the game ending process.

## 6.2 results

One of the biggest growing concerns for the project moving forward is speed. Although the time required to make an individual move isn’t especially high, a single game can easily contain dozens of moves, and each tournament consists of 31 games. Given that training includes hundreds of games of training, this can take a very long time. This is exacerbated to a great degree in the cases where the Torus engines in training play incredibly defensively. Human players rarely play a game with more than 50 moves, and almost never reach 70. However, around Tournament 50 in the training for this version, the length of games ballooned to be nearly 300 moves long. Thankfully, by tournament 100, they’ve shrunk back down to reasonable levels. However, it still would be worth it to look into methods that can quicken the program in the future.

It is very nice to see move raters rush towards winning the game in situations where victory is obvious to a human player. It would probably be worth it to have the program completely enter a game-tree searching move once two stones are directly adjacent and threatening each other. As before, understanding game trees would benefit board-rating to a greater degree than move rating. This is also likely to restrict the number of moves used in each game, which would allow for quicker training.

From a strategy standpoint, the engine does seem to understand some of the techniques used by humans. But moreover, it seems to have certain preferences that humans don’t use. Human players usually follow the common wisdom that the best moves are the ones placed two spaces to the side and one space above or below (or vice versa) to an already existing stone. However, the MoveRaters here seem to prefer playing their moves in small boxes, clustering moves together along diagonal lines.

# 7 Version 0.1.3

## 7.1 changes

After the previous version, there was a strong desire to shorten the lengths of games. Here are the changes made for the newest version

- First, the goal of avoiding gametrees has been abandoned. The program now performs a depth first search for any move that provides a victory within 5 turns.
- The MoveRaters no longer bother to rank every possible move in the first two turns. Because every spot is identical, the engine now simply selects A1 as the first move. For the second move, a smaller list of possible moves is given to the engine, to account for the existence of symmetry.
- The lengths of convolutional neurons within each neural net is now included when that neural net is printed for records. This is for convenience, in case future versions want to upload a neural net of a different size.

## 7.2 results

Finally, the games between these engines are starting to resemble human play! Stones are definitely placed more clustered together than they are in human play. In order to test if this is better or worse than human strategies, it would be necessary to create a function that plays an engine against a human player. This will be the necessary next addition in future versions.

# 8 Version 0.2.1

Enough changes have been made to the previous version, it makes sense to call this a minor update rather than just a patch! Here's what's new:

- MoveRaters are now able to play games with humans. The human can enter a move, and if that move is valid (i.e. is a properly formatted location on the board) and legal, that move is played. The human and machine take turns, with the first turn being decided randomly. If the human ever inputs an invalid/illegal move, they are prompted to enter another one.
- Ratings of any move can be normalized, placing the minimum possible rating at 0, and the maximum possible rating at 100, and adjusting all ratings in between to keep their original order. It does not bother doing this while deciding on a move to play, but does do this to inform the human player how effective a move is.
- The previous goal of avoiding game trees searches has been abandoned. The engine is now able to perform a breadth-first search of game trees to search for a move that can guarantee victory. However, this is incredibly slow, and so is currently turned off.

- It is now possible to reuse a MoveRater developed in a previous training session. This can be useful as a seed for further training sessions, to start off with a partially effective neural net for further training.
- Methods have been put in place to re-size Neural nets, to have different lengths of convolution layers, either by removing neurons (and their connections), or by adding new ones, where appropriate. New neurons can either have all values set to 0, or to a random value within the allowed range.

While this update is exciting for all the big changes made, it does not represent any large increase in the engine's effectiveness. There has been no further training, and the tree searching is currently unusable. The next version will hopefully optimize some of these problems away. In general, the current objective is to minimize the time used for every single function present. Some of the ideas for this are:

- Having boards, rather than games, alter themselves. This will make ko checking simpler in execution, and hopefully quicker as well.
- Giving the board object counters that keep track of how many black and white stones are present. This also will likely speed up ko checking radically.
- Checking only every second board in a games history for ko violations. Additionally, checking the first, second, third, and last boards is not required. This is due to the precise rules around ko, where all of these boards are guaranteed not to cause ko problems. This is likely to be a very small improvement if the previous change is made, but a small change may accumulate to a significant one over time.
- Compiling is currently done with the stock Python interpreter, CPython. Switching to another compiler, such as Pypy, might speed up the program.
- Tree searches currently check every legal move for a path that leads to inevitable victory. It might be more effective to only consider certain moves. For example, tree searches might only consider moves that are horizontally, vertically, or diagonally adjacent to existing stones.
- As mentioned, this project uses the old-fashioned sigmoid function rather than the newer, more popular, and faster ReLU. This was given justification, but it might be reasonable to swap from one to the other in light of everything that has been learned. This will create problems for the function that places all move scores between 0 and 100, due to the fact that the value inside any neuron is no longer limited to between -1 and 1.
- If the previous change is made, it would make sense to reconsider the use of powers as well. As mentioned, the decision to use a multiplier for each neuron was partially associated with the use of the sigmoid function.

Previously, it was argued that the min and max value can simply be set to the same number to have no power in effect. It's worth noting that this number would likely be 1, as any other power value would have the same effect, and cause complications to certain calculations. However, this would not be maximally efficient in terms of calculation time! It would be ideal not to waste time multiplying each neurons value by 1 while calculating an output for the neural net. Time would also be wasted assigning a value of 1 to every element in the arrays of neuron powers in each MoveRater. Therefore, it might be best to remove powers entirely, or at least add a global variable that can act as a flag to turn them on or off. Incidentally, turning off powers would also allow the function that places all move scores between 0 and 100 to be simpler again.

- Although Python is famous for its automatic garbage cleanup, it might be worth considering looking at garbage cleanup precisely. The tree searching function has a tendency to sputter and stop at random intervals, which might imply some sort of memory problem. Even if this is not a memory problem in particular, this clearly indicates some sort of issue in need of being resolved.

## 9 Version 0.2.2

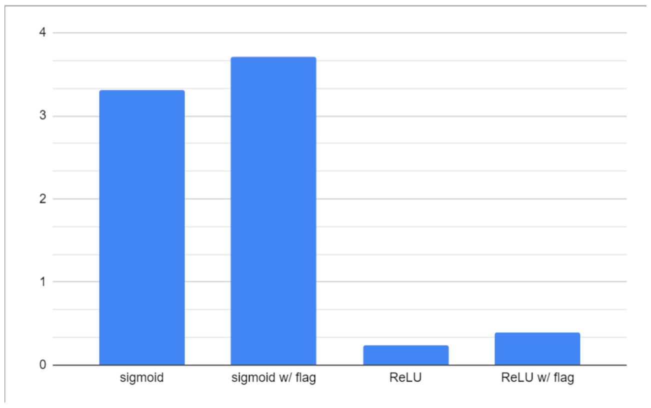
The primary goal set out after the last version was to speed the program up wherever possible. Because of this, this section will focus mostly on comparing alternate methods of optimization against the previous implementations.

For example, we wish to compare the sigmoid function to ReLU in terms of speed of execution. In addition to comparing sigmoid directly to ReLU, we can also measure the speed of a linearization function that uses either the sigmoid or ReLU based on an if-statement that accesses a global variable that acts as a flag.

This test works by creating a list of 1000000 random numbers between -5 and 5, and then applying the noted linearization function to each of these. Below are the observed times for 10 trials with each of the possible linearization functions.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| sigmoid | 3.24 | 3.35 | 3.31 | 3.32 | 3.48 | 3.38 | 3.36 | 3.23 | 3.24 | 3.21 |
| ReLU | 0.24 | 0.27 | 0.24 | 0.23 | 0.24 | 0.25 | 0.25 | 0.24 | 0.24 | 0.24 |
| sigmoid w/ flag | 3.64 | 3.55 | 3.88 | 3.58 | 4.09 | 3.59 | 3.70 | 3.65 | 3.83 | 3.60 |
| ReLU w/ flag | 0.35 | 0.35 | 0.35 | 0.35 | 0.35 | 0.39 | 0.34 | 0.37 | 0.44 | 0.59 |

Figure 12: A comparison of execution times for different linearization functions



ReLU is clearly much faster than the sigmoid function. But is this a worthwile change? Each time a move is rated, every non-input neuron within a MoveRater has to be normalized. MoveRaters currently have 100 of such neurons. MoveRaters have to consider at most 79 moves. Most games are over within 100 moves, and games played by humans are usually over within 50 movs. If tree searching for game ends can be made to work better, games will be shorter, but 100 is an okay ballpark estimation for the upper bound of the length of a game for now. Each tournament includes 31 games. Therefore, the number of times a neuron has to be linearized within a tournament is roughly

$$100 \cdot 79 \cdot 100 \cdot 31 = 24,490,000$$

Assuming the average times found earlier, the sigmoid function takes about 81.1 seconds over the course of a tournament. By contrast, ReLU requires only 5.9 seconds to accomplish the same task. On one hand, this is a significant improvement from the old version. On the other hand, this is only a fraction of the time required for a tournament at present. A tournament presently takes between 15 and 30 minutes to complete. Switching over to ReLU would be worthwhile, but would not fix the problem.

How much time is saved by skipping the power calculations? It would most likely be minor, but what are the exact time frames? Here is the structure of this test:

- 100 games are created. For each game, a random number $n$ between 0 and 40 is chosen, and $n$ random moves are applied to that game.
- 100 random MoveRaters are generated.
- A timer is started.
- For each number $i$ between 1 and 100, the $i$th MoveRater finds its highest ranking move on the $i$th game. This is done first with the powers on using sigmoid.

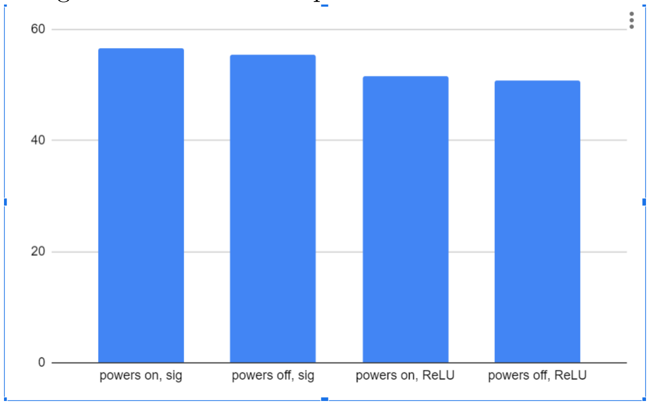
- The timer is stopped.
- The previous three steps are repeated with powers on using ReLU, powers off using sigmoid, and powers off using ReLU

This test is performed 10 times, and the resulting times, in seconds, are shown below:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| powers on, sig | 114.48 | 56.76 | 55.96 | 53.59 | 58.25 | 49.25 | 57.83 | 51.68 | 63.64 | 51.82 |
| powers on, ReLU | 51.79 | 52.31 | 51.52 | 49.91 | 53.90 | 46.20 | 53.78 | 47.80 | 61.09 | 47.75 |
| powers off, sig | 55.97 | 56.35 | 55.56 | 53.35 | 57.77 | 48.80 | 59.07 | 51.21 | 63.22 | 51.49 |
| powers off, ReLU | 51.60 | 51.92 | 50.98 | 49.40 | 53.38 | 44.89 | 53.07 | 47.20 | 58.04 | 47.40 |

As before, the averages are shown in this bar graph:

Figure 13: Another comparison of execution times



As expected, it seems that leaving powers off is faster than having them on. Of course, one could run an analysis to find the statistical significance of these numbers. However, this is only meant to be a quick check to confirm suspicions about which option is faster. According to this data, removing powers and switching to ReLU may result in a 10% increase in speed. As mentioned, games of Torus last roughly 40 moves, and tournaments contain exactly 31 games. This means that a tournament requires roughly 1240 moves, and so a 12 minute tournament may be reduced to a 10.5 minute tournament.

Next, the overall structure of the Gamestate object is going to be rethought. Presently, the ”board” in each Gamestate is simply a 2D array. However, for some of the plans we have, I think it makes more sense to create a Boardstate class, of which the Gamestate is a subclass. Then the history within a gamestate can be an array of these boardstates, and can store a lot more information in order to speed up Ko checks.

While this is being done, a general effort is being made to trim the fat from the rest of the functions of this project. For example, the function that checks if a move is suicidal was cut from an average runtime of 8.33 seconds per million runs to 6.49 seconds. This is a remarkably small change in the grand scheme of a full tounament, but its at least a little faster.

$$A \sqsupseteq B$$

## References

[1] Ross, Philip E. "THE EXPERT MIND." Scientific American, vol. 295, no. 2, 2006, pp. 64–71. JSTOR, //www.jstor.org/stable/26068925. Accessed 2 June 2023.

[2] The Brittish Go Association "How to Play" www.britgo.org/intro/intro2.html. Accessed 7 July 2023.

[3] chess.com "Chess Terms: Chess Engine" www.chess.com/terms/chess-engine. Accessed 7 July 2023.

[4] chess.com "How are moves classified? What is a 'Blunder' or 'Brilliant' and etc?" support.chess.com/article/2965-how-are-moves-classified-what-is-a-blunder-or-brilliant-and-etc. Accessed 7 July 2023.

[5] Mohammad Hamzah "Game Theory: How Stockfish Mastered Chess" https://blogs.cornell.edu/info2040/2022/09/30/game-theory-how-stockfish-mastered-chess/. Accessed 7 July 2023.

[6] Deepmind "AlphaGo" https://www.deepmind.com/research/highlighted-research/alphago Accessed 7 July 2023.