Prior State Voting (PSV): Predicting unknown move states from prior states in Q-table reinforcement learning

Oliver Benning

May 2021

Contents

1 Introduction

1.1 Background

This project was completed as a final year honours project in mathematics at the University of Ottawa Faculty of Science. It was supervised by Dr. Patrick Boily from the Faculty of Science. Additionally,

Rachel Ostic is thanked for proofreading the project and assisting in the graphical diagrams.

All source code for this project is available publicly at https://github.com/obenn/PSV.

1.2 Motivation

This project investigates the use of Q-learning in the context of a two-player board game with full information, i.e. both players have the same information in regards to the game's state at any given time. Examples of such games include chess, checkers and Thud!. In Q-learning we maintain a table of the states of a given game along with the moves available, and train by adjusting the table's weights of which moves are best based on the results of training iterations, usually positively or negatively dependent on a reinforcement factor, such as a win or a loss.

In a two player context we can train a model for each player or just use the same model to contain states for both sides of the board. Either way, two significant challenges remain. First, unless we have access to a skilled adversary, often the only way to train a model is to play against itself, which requires many iterations in order to bootstrap to a certain level of proficiency. The second issue is in Q-table size, where a game like checkers has 500 quintillion possible board states [1], more states than any amount of computer memory available can handle.

Our contribution in this research is first to apply a straightforward Q-learning algorithm to checkers and initially evaluate the quality of moves trained as well as breadth of states that are covered in training. The next, more significant, contribution is the proposal and initial development of an algorithm called "Prior State Voting" (PSV) which can efficiently determine a good move to make in an unknown board state by analyzing prior states in a Q-table and deciding on the next move by seeing which move scored best in similar past states. We implement this for checkers and provide empirical results against a random player when compared to a random approach in Q-table move substitution as well as a naive "keep training" approach, with promising results. We also highlight how this algorithm can generalize to other games of a based on similar grid-like form, such as Chess and Thud!, and ways in which the algorithm can be made more performant.

1.3 Why checkers?

Checkers is an interesting game because it has already been weakly solved to guarantee a stalemate if both players play perfectly [2]. This gives a clear performance benchmark to aim for. However, the authors also state that with their technique, it is unlikely that checkers will be strongly solved since they rely in part on depth searches reaching 100-step look-ahead and this is computationally expensive. Other techniques, like reinforcement learning may have advantages [3, 4, 5].

It is also a good game to tackle because it is not as complex as chess but still has a large number of possible states to explore. This is the main reason it was used for this research, in order to provide the lowest technical barrier to entry to implement and study the proposed algorithm, with the aim to expand to more complex games later on.

1.4 Rules of checkers

Checkers is a classic two-player board game. Though there are many variants, these are rules for a basic and more or less standard version of the game.

Each player begins with 12 pieces on the dark squares in the three rows nearest to them. Black plays first and then players alternate. The pieces can advance diagonally, one square at a time. Pieces are initially only allowed to move toward the opponent, they cannot retreat until they have reached the opposite side of the board.

If an opponent's piece is on the next square with an open space behind it, it can be captured by a piece "jumping" over it. A sequence of jumps can capture multiple opponent pieces in a single turn. If

a capture is possible, the player must capture instead of doing a single step move. If there are multiple possible captures, the player is free to choose which to perform. The capture chain has to be completed once it is started, meaning that if the selected jumper can capture more pieces on its path, it must do so.

If a piece ends its turn in the row of the board closest to the opponent, it becomes a "king" and is able to travel both backward and forward, including alternating forward and back in a capture chain.

The goal is to prevent the opponent from making any more moves, either by trapping their pieces so they are stuck or by capturing all their pieces. Stalemates are also possible if the same board state appears three times, or if there have been 100 turns taken with no captures.

2 Development

2.1 Game encoding for checkers

First the game must be represented as a Python class in order to be manipulated with code. The key here is not only to model checkers, but to model it in such a fashion that the interface with the game can be generalized with all similar board games in its class, such as chess. We therefore detail the checkers implementation here, but note the aspects which are done in a generalizable manner.

2.1.1 `Piece` class

The first class used is `Piece` to refer to the player's tokens. This is a fairly simple class in this case, but would be more complicated in a game like chess where there are many different kinds of pieces. The class is only use internally to facilitate tracking the game's state.

Table 1 lists the properties of this class.

Attribute	Value	Description
`colour`	`'black'` or `'white'`	Token colour, player with black plays first.
`king`	`True` or `False`	Indicates whether or not piece has reached the opponent's side of the board to become a king.
`dead`	`True` or `False`	Indicates whether or not a piece has been jumped over and removed from the board. Pieces are marked as dead when removed.
Representation	`'b'`, `'B'`, `'w'` or `'W'`	Pieces are visually represented with the first letter of their colour. Capital letters indicate kings.

Table 1: `Piece` attributes

2.1.2 `Cell` class

Next, the `Cell` class refers to the board squares, which are indexed as shown in Figure 1 based on a similar scheme seen in a checkers web game [6]. In checkers, only the black squares are playable since the pieces always move along diagonals, so these are the only ones that need to be numbered to concisely identify a board state. When initializing a game, the black pieces are placed on indices 0 through 11 and white pieces occupy indices 20 through 31.

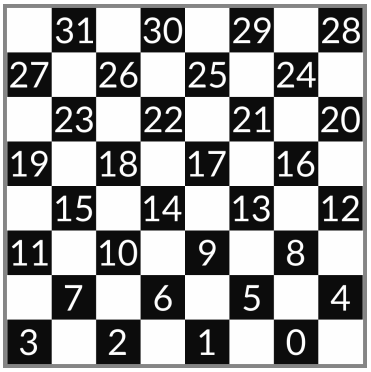

Figure 1: Indices for board cells.

The `Cell` class also has a set of attributes that are assigned upon initialization. These are given in Table 2. Many of the properties of the cells depend on mathematical properties of their indices. Taking the index n modulo 8 gives the the position relative to the edges: $n \equiv 3 \pmod 8$ is far left while $n \equiv 4 \pmod 8$ is far right. If $n \in \{0, 1, 2, 3\} \pmod 8$, then the `above` cells are $(n + 5, n + 4)$ and the `below` cells are $(n - 3, n - 4)$. If $n \in \{4, 5, 6, 7\} \pmod 8$, these become $(n + 4, n + 3)$ and $(n - 4, n - 5)$. These conditions as well as upper and lower bounds to define end rows provide the basic structure of the board.

Attribute	**Value**	**Description**
`index`	Integer, range 0 through 31 inclusively	Position of cell on board according to Figure 1.
`piece`	Instance of `Piece` class or `None`	Gives the piece found on the selected cell.
`game`	Instance of `Game` class	Puts the cell in the context of the full game board status.
`above` and `below`	Pair of arrays of instances of `Cell` class	Lists the cells diagonally adjacent to the one selected, since these are the cells that pieces can potentially travel to in a regular move. These arrays each contain 2 entries. Above means in the row of greater indices, below is the row of lower indices. When assigned, the first element is to the left of the current cell and the second is to the right. If no such cell exists (e.g. below cell 2 or left above cell 11), the associated entry is `None`.

Table 2: `Cell` attributes

There are class methods for cells to determine what moves are possible for the different pieces on the board. For reference, we distinguish between "jumps" and "moves" in the code, where "jumps" pass over opponent pieces to capture them, and "moves" (or "regular moves" if the context is unclear) are the single-step shifts to diagonally adjacent cells.

Finding valid moves for a piece on a given cell is quite simple: check if adjacent cells in its `above` and `below` lists are unoccupied. Black pieces can only move in the above direction until they become kings, while white pieces can only move below until they are kings. The method `possible_moves` returns the list of allowed destination indices for a piece given its starting point. To make the options clearer, the method `valid_moves` returns pairs of start and end indices in the form `((start,end`$_1$`),(start,end`$_2$`),...)`.

Finding valid jumps requires a recursive algorithm to follow all possible capture sequences. The player can choose which jump to do, but once they start, they must continue to capture with their piece until they can't anymore. The first step is to determine if the selected cell's piece has the potential to start a jump sequence by checking that the adjacent cell contains an opponent piece, and that the next adjacent cell in the same direction is vacant. This method is called `possible_jumps`. Next, `possible_jumps` is

called recursively on the landing cells, adding captures until there are no more possible. When adding to the sequence, the already-captured opponent pieces are excluded to ensure they cannot be captured twice. This recursive method is called `valid_jumps`, and it returns an array of index tuples of the form `(start,((captured`$_1$`,landing`$_1$`),(captured`$_2$`,landing`$_2$`),...))` to indicate the possible capture paths to follow.

The `Cell` class also has a `do_move` method to perform a chosen move, whether regular or jump.

The `Cell` class would have analogous implementations in other games, but, like `Piece`, it is only used internally and not a necessary part of implementation to be used with the Q-learning algorithm.

2.1.3 `Game` class

The final class is the `Game` class which encompasses the whole board. All the methods and attributes here are part of the interface to operate with the Q-table algorithm, so while they are implemented specifically for checkers here, to train against another game it suffices to implement the game using the interface described in a manner that makes sense for that game. As such, the method and attribute names are suitably general so as to be more broadly applicable, assuming that the rules and mechanics of the chosen game are supported by lower-level classes like `Piece` and `Cell`.

The `Game` attributes are presented in Table 3.

Attribute	Value	Description
`board`	Array of 32 `Cell` instances	Representation of board cells ordered by indices shown in Figure 1.
`initial_turn`	`'black'`	Colour that plays first, black by default.
`turn`	`'black'` or `'white'`	Which player's turn it is. This alternates every move.
`moves_played`	Positive integer	Initialized to zero, this is the combined number of turns that have been taken by both players.
`over`	`True` or `False`	Indicates whether or not the game is finished.
`winner`	`'black'` or `'white'`	Indicates which player won the game.

Table 3: `Game` attributes

The `Game` methods are presented in Table 4.

Method	Parameters	Description
`clone`	None	Returns a new copy of the current game state.
`outcome`	None	Returns a Boolean representing True if the first player that moved won.
`get_playable_moves`	None	Returns a list of playable moves given the current game state.
`do_move`	move	Performs the move given and changes the game state appropriately.
`random_move`	None	Returns a random move.
`do_random_move`	None	Performs a random move.
`__str__`	None	Returns a representation of the game in string form.
`approximator`	Previous state, New state	Required when using PSV, must update coefficients in new state depending on similarity to previous state.

Table 4: `Game` methods

2.2 Q-learning

The algorithm is a back-propagation implementation of Q-learning where the moves are reinforced in reverse playing order after reaching an end state which determines what reward to attribute. Q-learning is a temporal-difference learning technique which assigns a numerical "Q-value" to each potential action that can be performed based on previous training returns. The values for different alternatives can be compared to determine the optimal choice given information from the board state or, more generally, the environment. To update Q-values, the following Bellman equation is used to calculate a weighted average between the current estimate and new reward information:

$$q_{\text{updated}}(S_t, a_t) \;=\; q(S_t, a_t) + \alpha \left(r_t + \gamma \max_{a \in \{a_{t+1}\}} q(S_{t+1}, a) - q(S_t, a_t) \right) \tag{1}$$

In other words, the Q-value associated to state-action (S_t, a_t) pair is updated by using a linear combination of the old value:

$$q(S_t, a_t) \tag{2}$$

and a new estimate:

$$r_t + \gamma \max_{a \in \{a_{t+1}\}} q(S_{t+1}, a) \tag{3}$$

The new estimate contains the reward r_t received for taking action a_t, and the maximum over the set of possible Q-values for actions that can be taken from new state S_{t+1}. These two terms are weighted with coefficient α known as the learning rate. The next step contribution is weighted with coefficient γ, a discount factor.

This implementation uses a zero-initialization condition for new Q-values. The Q-table entries contain a flattened form of the board state based on Fig. 1 concatenated with which player's turn it is. It's possible that the same state may be entered twice, once for each player, but this makes it easier to get the best moves for the relevant player's turn. During training, the system plays through a full ϵ-greedy game until one colour wins. Both colours refer to the generated Q-table, i.e. the model is playing against itself. The winner gets a $+1$ reward for all the moves made while the loser gets -1. The player's reward is applied to each of its moves from game end to start in the Bellman equation update. There are other potential strategies for this credit assignment that could be tested in future experiments including boosting the reward for captures or discounting the reward when back-propagating toward the start of the game to reflect a confidence level increasing toward known endgame strategies.

The algorithm is implemented in Python but described in pseudo-code in algorithm 1 for clarity.

Algorithm 1: Q-learning algorithm

Finding a game solution:
while *game not won* **do**
  if *current state in Q-table* **then**
    do move ϵ-greedy;
  else
    do random move;
  end
end
return *path_to_root, move_list*

Backtrack to update Q-table:
n = length(path_to_root);
if *win* **then**
  reward = 1;
else
  reward = -1;
end
max_next = 0;
for $i = n-1, n-2, \ldots, 2, 1$ **do**
  move = move_list(i);
  if S_i *not in Q-table* **then**
    add S_i and its valid moves $\{a_j\}$ to Q-table;
    for a *in* $\{a_j\}$ **do**
      $q(S_i, a) = 0$;
    end
  end
  $q(S_i, \text{move}) = q(S_i, \text{move}) + \alpha\left(\text{reward} + \gamma\,\text{max\_next} - q(S_i, \text{move})\right)$;
  max_next = $\max_{a \in \{a_j\}} q(S_i, a)$;
end

2.3 Prior State Voting (PSV)

In order to make best use of the limited number of states in the Q-table, we define a strategy in algorithm 2 to find rankings for available moves based on similar board positions seen in training.

For an unknown state, take the list of possible moves and define a score for each that will ultimately determine their rankings. The unknown state is compared with all previous states in the Q-table. When performing the comparison, if there is an "equivalent" move (i.e. move is same relative direction), then calculate the "matching fraction" between board states. The score for the associated move is incremented by the Q-value for the equivalent move multiplied by the matching fraction. This is illustrated in Figure 2. Finally, take the average score added over all the previous equivalent moves to compensate for the case in which one move may have more Q-table equivalences than another. This becomes the value to

use in the ranking and selection.

Algorithm 2: State approximation algorithm

Set up:
Get all valid moves;
Examine previous similar states:
for ***move*** *in valid moves* **do**
    score(**move**) = 0;
    for ***state*** *in Q-table* **do**
        n = 0;
        if ***state*** *has equivalent move* **then**
            matching_fraction = (# matching cells in overlap area)/(total # board cells);
            score(**move**) = score(**move**) + (matching_fraction)×q(**state**, equivalent move);
            n = n+1;
        end
    end
    score(**move**) = score(**move**)/n;
end
Pick move to do:
max_score = max_{move}score(move);
do move associated with max_score;

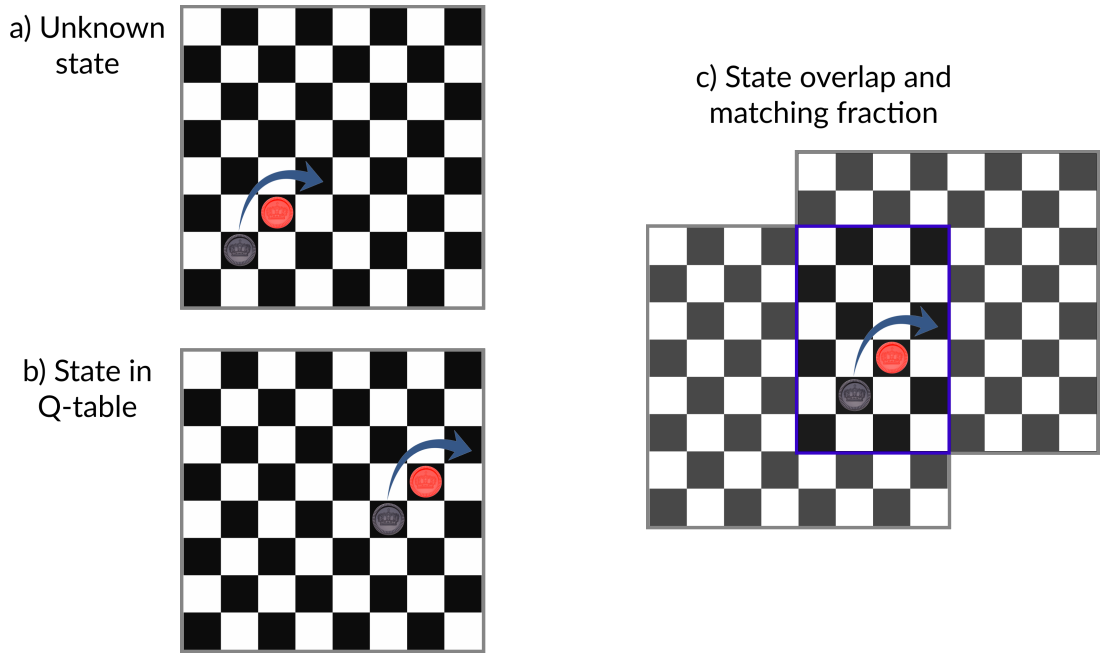

Figure 2: a) It is black's turn with a possible move as indicated, the black piece jumping over the red piece located above and to the right. The board state is encountered during gameplay, but is not in the Q-table, so the agent cannot accurately assess the quality of the move. b) A state found in the Q-table with an "equivalent" move. c) Overlapping the two board states to calculate the "matching fraction", which corresponds to the number of matching cells in the overlap zone divided by the total number of cells. In this example, the fraction is $\frac{4\times6}{64} = 0.375$ since all the cells in the overlap are the same.

2.3.1 Code implementation

For this strategy, it's easier to picture the board as a plane with a two-dimensional coordinate system. Then, the shift between overlapping boards illustrated in Figure 2c can be expressed as a vector $(\Delta x, \Delta y)$.

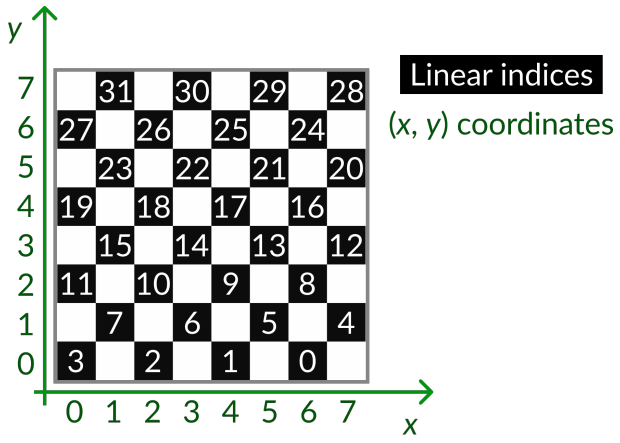

Figure 3: Comparison of linear or 1-dimensional index system for black cells with (x, y) coordinates.

To convert from one system to the other, we use the following mappings where n represents the usual index:

$$x = 2\,(3 - n(\text{mod } 4)) + \left\lfloor \frac{n}{4} \right\rfloor (\text{mod } 4) \tag{4}$$

$$y = \left\lfloor \frac{n}{4} \right\rfloor \tag{5}$$

$$n = 4y - \left\lfloor \frac{x}{2} \right\rfloor + 3 \quad \text{if } (x + y) \text{ even (else represents non-indexed white square)} \tag{6}$$

With this notation, if S represents the unknown board state with a cell of interest at (x, y) and the Q-table state S' is visually shifted by $(\Delta x, \Delta y)$ with respect to S, then the equivalent location in the overlap in the frame of reference of S' is $(x', y') = (x - \Delta x, y - \Delta y)$.

To perform Prior State Voting, we get the list of possible moves in order to assess and rank them. Then, we iterate through the Q-table, checking each stored state for possible equivalent moves: same player's turn, same move direction and distance.

To find these equivalent moves, the first step is to check that the move types are the same by comparing move lengths and types of pieces captured (e.g. both are capture sequences which jump over a king then a regular piece). Then, apply the mappings in Eq. 4 and 5 to both the considered move starting position and the Q-table move starting position in order to place them in the two-dimensional grid representation for easier comparison. Let S_Q be the frame of reference for the Q-table state while S_N is the new unrecognized state, with potentially equivalent moves starting from coordinates given by (x_Q, y_Q) and (x_N, y_N) respectively. The translation vector is calculated as $(\Delta x, \Delta y) = (x_Q, y_Q) - (x_N, y_N)$. Then, to directly compare the two in the same frame of reference, the vector $(x_N + \Delta x, y_N + \Delta y)$ brings the new state move into the Q-table state frame of reference S_Q.

For example, Fig. 2 has a move starting from cell number $(7)_N \mapsto (x_N, y_N) = (1, 1)$ which needs to be compared with $(13)_Q \mapsto (x_Q, y_Q) = (5, 3)$ leading to a translation vector $(\Delta x, \Delta y) = (4, 2)$.

From this point, the last check to ensure the moves are fully equivalent is to see that the lists of piece landing points are equal once expressed in the same frame of reference. In Fig. 2, this is the case because when expressed in S_Q, the black piece travels from $(5, 3)$ to $(7, 5)$ on both boards.

If true, the final comparison is calculating the matching fraction between boards. The range of squares

to check in frame of reference S_Q depends on the sign of the translation.

$$x_{min} = \begin{cases} 0 & \text{if } \Delta x < 0 \\ \Delta x & \text{otherwise} \end{cases} \tag{7}$$

$$x_{max} = \begin{cases} 7 + \Delta x & \text{if } \Delta x < 0 \\ 7 & \text{otherwise} \end{cases} \tag{8}$$

$$y_{min} = \begin{cases} 0 & \text{if } \Delta y < 0 \\ \Delta y & \text{otherwise} \end{cases} \tag{9}$$

$$y_{max} = \begin{cases} 7 + \Delta y & \text{if } \Delta y < 0 \\ 7 & \text{otherwise} \end{cases} \tag{10}$$

For each cell delimited by the range $[x_{min}, x_{max}]_Q \times [y_{min}, y_{max}]_Q$, compare the Q-table and new states to count how many cells contain the same pieces. This value divided by 64 total board cells is the matching fraction. In Fig. 2, this works out to the range $[4, 7]_Q \times [2, 7]_Q$. The equivalent range to check in the new state within its own frame of reference is $[x_{min} - \Delta x, x_{max} - \Delta x]_N \times [y_{min} - \Delta y, y_{max} - \Delta y]_N = [0, 3]_N \times [0, 5]_N$.

3 Analysis

3.1 Random and Keep training comparator

In order to evaluate results we train our model with varying levels of training repetitions and analyze how often the model is able to beat an adversary who is playing random moves, in order to determine if our model really is capable of winning. We also compare it to two other strategies as comparison. The first is to use the Q-table state if available, and a random move otherwise, effectively giving a benchmark for how well the prediction algorithm enhances the play. The other is another experimental algorithm where when an unknown move is seen, the model goes back and retrains for n repetitions from that unknown state, similar to a Monte Carlo graph traversal style of approach. In the analysis we set the keep training comparator to 100 repetitions.

3.2 Big O remarks

No matter the comparator algorithm used the training time is identical. It is in linear proportion to the number of training repetitions, so $O(r)$ where r is the number of repetitions.

When playing the game, the time to make a move varies between the three models, and is shown empirically in Figure 5. Random and keep training models are constant time, i.e. their runtime is unrelated to the number of training iterations, since both choosing a random move and training 100 iterations both take the same amount of time invariant to the size of the Q-table.

For PSV, the runtime for finding each move is linear in proportion to the size of the Q-table, which in turn is proportional to the number of training repetitions. This is because each time the model encounters an unknown state it must compare this state with all prior trained states in the Q-table. We can therefore consider the runtime for finding an unknown move $O(r)$ where r is the number of training repetitions.

3.3 Results

For all three models we also collect the mean and standard deviation for time taken to play each test game, in order to measure the time tradeoffs between the different approaches.

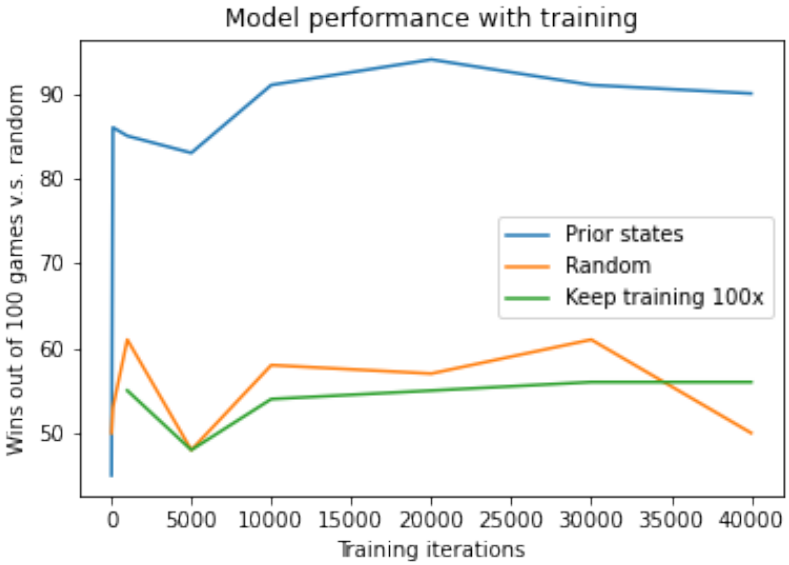

Figure 4: Model performance with training

In Fig. 4 we see some very interesting results. The most significant is the high performance of PSV which consistently outperforms the reference models, and performs extremely well in general. Other observations include the keep training model under-performing until the end, which implies it could possibly have regressive effects. Additionally the random model performance is consistently above 50%, meaning the Q-table states do give the model an advantage in the early game.

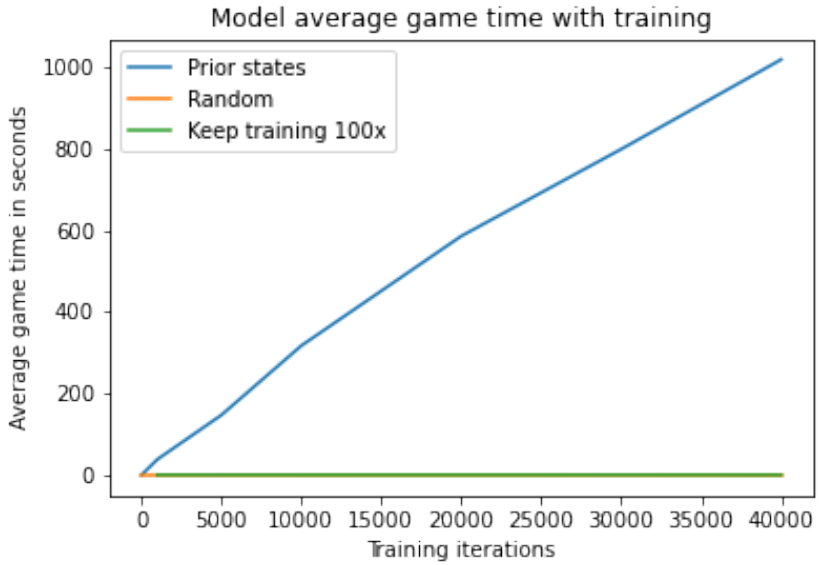

Figure 5: Model average game time with training

In Fig. 5 we see confirmations of our analysis from section 3.2, with PSV showing a continuous linear growth and the other two constant. We note that Random is constant at a value of 0.002 seconds, and keep training has a mean value of about 1.5 seconds.

3.4 Broader applicability and further work

The results of this algorithm are extremely promising, but there is more to investigate. A key issue here is the linear time scaling to Q-table size when making moves, which grew too large to test higher values of repetitions in this study. In order to feasibly test higher values, computation parallelization could be implemented on PSV in a MapReduce style of approach, effectively dividing the computation time by the number of CPU cores used, which coupled with a cloud server could enable much faster analysis.

Another candidate improvement to Q-learning is deep Q-learning, which is used in many next-generation chess engines. This technique replaces the large Q-table with a neural network that takes the state as an input and the output is predicted Q-values for moves. One reason this could be advantageous is that when using a state not seen in training, the output may still be in a realistic range based on how much it resembles training cases. As seen in projects like alpha-zero-general [7], this contains the drawback of requiring GPU training hardware as well as much more time to train, but using less storage space. Q-learning coupled with PSV could bring faster training and better explainability, as long as there is enough storage available. Performance however is the largest differentiator, and a great candidate for future work is to investigate both models and their performance in checkers.

An important point in this work is that the algorithm is not at all restricted to checkers. It can be expanded to similar games like chess and Thud! by encoding those games in Python and writing the comparator logic. This is not a trivial task, but this project was made to facilitate future work on top of the checkers game, which can lead to exciting results.

Finally, Q-learning and PSV are highly interpretable algorithms. The inspiration for PSV is how a human figures out a new move in chess or checkers, by looking at their own past experience and finding similarities to what worked, including noticing patterns in relative piece positions. There is high potential for augmenting the PSV algorithm to not only predict the next move, but provide rich information as to how it got said result, and conveying that to a human actor looking to improve in the game in question. To extract even more information from an existing Q-table, it would be possible to expand the equivalent move search based on symmetry properties of the game board such as 90° or 180° rotations about its center, reflections, or player inversions in addition to the translation system described here.

4 Conclusion

In this work we introduce a novel algorithm for the prediction of unknown move states when playing two-player, full-information, grid-structured games called Prior State Voting. We detail the development and implementation of the algorithm for the game of checkers and how it can be generalized. We then test the performance of the algorithm against two reference algorithms with extremely promising results. Finally, we detail the many expansion possibilities of this work into future games and deeper, more optimal implementation of the algorithm as well as opportunities for use in explainable AI.

References

[1] Fraser Cain. The game of checkers: Solved. https://www.wired.com/2007/07/the-game-of-che/. Accessed: 2021-04-19.

[2] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[3] K. Chellapilla and D. B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428, 2001.

[4] Henrique C. Neto, Rita M. Julia, Silva, Gutierrez S. Caexeta, Ayres R. Barcelos, and Araujo. Ls-visiondraughts: improving the performance of an agent for checkers by integrating computational intelligence, reinforcement learning and a powerful search method. *Applied Intelligence*, 41(2):525–550, 09 2014. Copyright - Springer Science+Business Media New York 2014; Last updated - 2020-11-18.

[5] Castro N. Henrique and Rita M. Silva Julia. Ace-rl-checkers: decision-making adaptability through integration of automatic case elicitation, reinforcement learning, and sequential pattern mining. *Knowledge and Information Systems*, 57(3):603–634, 12 2018. Copyright - Knowledge and Information Systems is a copyright of Springer, (2018). All Rights Reserved; Last updated - 2018-09-07.

[6] Einar Egilsson. Checkers. https://cardgames.io/checkers/. Accessed: 2021-03-30.

[7] Alpha zero general. https://github.com/suragnair/alpha-zero-general. Accessed: 2021-04-21.