COMP30024 Artificial Intelligence Project 2 Report

Alexander Westcott 994344 and Cameron Chandler 993990

Abstract

This report describes the research behind the construction of a self-learning agent to play the game of expendibots. Chess is a game with centuries of human trial and error from which to build agents; Expendibots is no such game. Because there is no expert knowledge or historical database of games in the domain of expendibots, a self-playing algorithm is necessary to produce a competitive agent. A variant of treestrap was implemented as the core algorithm to determine an appropriate evaluation function, and many modifications are discussed in this report. The resulting agent is able to beat humans with no experience in expendibots, but is unable to triumph against the authors. Further work should focus on the application of neural networks to evaluate states.

1. Introduction

This report discusses how a constructed expendibots agent plays games, methodologies of the machine learning algorithms used to train it, and the effectiveness of this agent. This is supplemented by discussion of further research, optimisations and trialed techniques relevant to the development of the final agent.

2. Choosing Actions

As it is in chess, a minimax search to terminal depth is completely infeasible in expendibots due to the large branching factor and depth of the search tree. This is even with $\alpha\beta$ pruning and other search enhancements. Instead some sort of evaluation of state must be made at a cut-off depth. The expendibots agent typically uses an alpha-beta search to a depth of two ply, utilising a heuristic function detailed in later sections. The formalisation of the search, and ultimately how actions come to be chosen is detailed in this section.

State s can be represented by numerical feature vector $\phi(s)$. Using weight vector θ, a simple linear evaluation function $H_\theta(s)$ can be used to output a single number representing the desirability of a state. Given that the utility of win and loss was assigned to 100 and -100 respectively, the evaluation function was truncated between $(-100, 100)$ – strictly between the utility of a loss and a win. The precise formula used is

$$H_\theta(s) = \begin{cases} -100 \cdot \tanh\left(\dfrac{\phi(s)^T\theta}{35}\right), & \phi(s)^T\theta \leq -99 \\ \phi(s)^T\theta, & -100 < \phi(s)^T\theta < 100 \\ 100 \cdot \tanh\left(\dfrac{\phi(s)^T\theta}{35}\right), & \phi(s)^T\theta \geq 99 \end{cases}$$

It appeared that the $\alpha\beta$ pruning search slowed down when large stacks were on the board. This is likely because of the large number of moves (and therefore larger branching factor) that a large stack can make since it can move anywhere from one to n pieces where n is the height of the stack. To counteract this, the agent was limited to selecting from moving $\{1, n-1, n\}$ pieces at a time from a given stack. These were chosen as they cover the majority of standard moves. In practice, this feature was not found to significantly speed up search so was removed. It is hypothesised that not enough states were encountered with large stacks to save significant time.

2.1. Move Ordering

The performance of $\alpha\beta$ search is highly dependant on move ordering.[1] If branches with better moves are explored sooner, it will be easier to prune off subsequently explored branches. Since

in expendibots the best moves possible moves are booms that increase the piece advantage, moves were sorted with booms first. Furthermore, booms on the opponent's side of the board are tried first, as there is likely a higher chance of finding favourable booms there. Further optimisations were made in only generating certain booms, as discussed in the next section.

Move actions were also sorted, prioritising moving towards the opponent's side of the board as these moves are more likely to lead to favourable booms. Speed improvements in the $\alpha\beta$ search greater than 100% were noted on account of the move ordering.

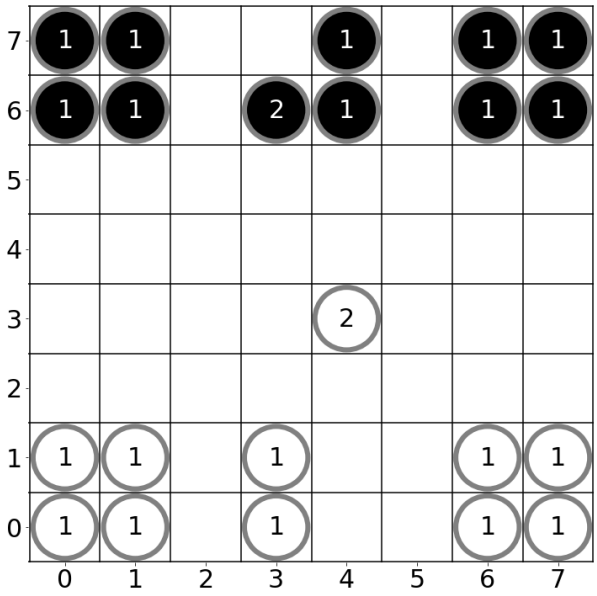
2.2. Action Generation

In a game like chess, there is a bijection between a set of actions for a given state and the set of resulting states from those actions, however, this is not always the case in expendibots; The carnality of the set of actions can be greater than the carnality of the set of resulting states. This is easily seen in the opening state of the game; The player to move has twelve boom actions available (one for each piece) but only four different board states result from these booms. It is therefore unnecessary to consider all booms in this state as presenting three is sufficient. This principle was motivation to remove all redundant boom actions from the action generation, reducing the branching factor of states where this overlap occurs, and allowing for faster search. Booms were also not considered if they did not improve the ratio of the number of pieces to opponent pieces on the board. There is very rarely a strategic reason to make such a boom, so these were removed from the action generation entirely to both allow for better move ordering and to reduce the branching factor, ultimately speeding up search.

2.3. Opening Book

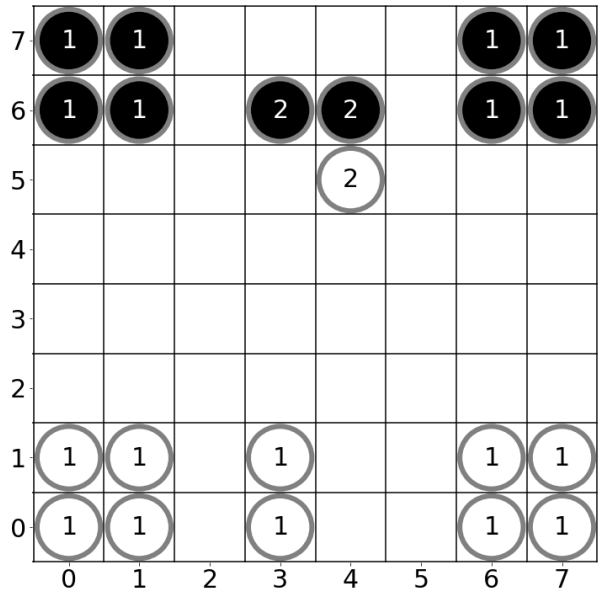
It is believed that an optimal opening has been found for expendibots, however, this is yet to be formally proven. Playing white in the following examples, the first moves are $(\text{"MOVE"}, 1, (0,4), (1,4))$ $(\text{"MOVE"}, 2, (1,4), (3,4))$ regardless of what the opponent (second to move) does (Fig 2.1).

Figure 2.1 Game state after three ply



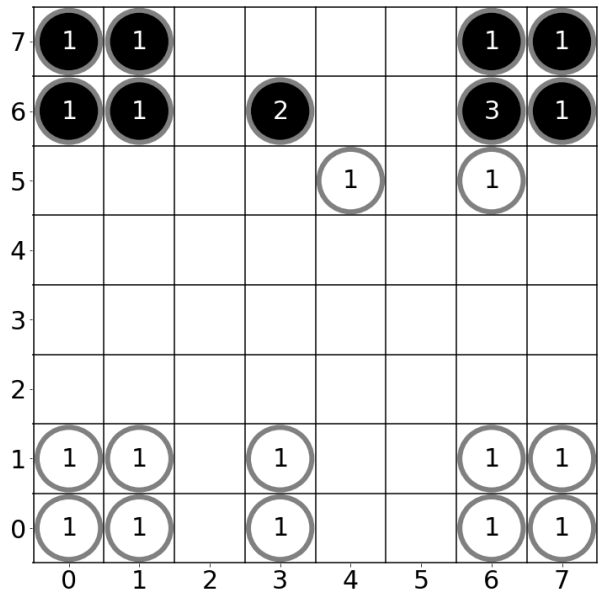
After only three ply, black's pieces are already susceptible no matter how they moved. If black focuses on building the left or right clusters, then white trades one for four in the centre stack. If black moves the centre stacks away, then white makes an equivalent trade on the right. In extending play, no method for black to take advantage of the remaining white clusters has been found, since white is able to move away before black can get there. In fact, it is believed that the only response that does not end in a loss for black is employing the same tactic and mirroring white. In expendibots, a draw can always be forced if the black perfectly mirrors white, and black is only able to win if white makes sub-optimal moves to its detriment.

Figure 2.2 Game state after five ply



In this case, black can save three pieces with $(\text{"MOVE"}, 3, (6,6), (3,6))$, but will be unable to trade advantageously enough to catch up with white's piece advantage (Fig 2.3).

Figure 2.3 Game state after seven ply



An opening book including these presumably optimal moves was constructed as a hash table for every possible state in the first eight ply. Because it was not necessary to consider turn number or state history when making an opening move, the keys were derived from only the array representation of the board. It was found that hashing the string representation an array was the most time efficient method, since mutable arrays cannot themselves be hashed. White's opening book consists of mappings from $8,690$ unique states (excluding symmetry) to actions, and black's opening book consisted of $75,514$ unique states.

The book was constructed using a hand-crafted decision tree, where a given state was allocated an action according to a set of rules. To cut the branching factor and save computing time, all symmetry was removed at each ply such that moves that are symmetrically equivalent were not calculated multiple times. After computation, the symmetrical moves were added back into the book by mirroring each calculated state and action pair. The final opening book was around 40MB in size, well within the 100MB module size constraint.

2.4. End Game

From observations of human-played games, it was noted that piece advantage when either player had three or less pieces guaranteed a reasonably simple win with perfect play (assuming they aren't in a position to immediately lose upon arriving in a state with the aforementioned piece balance). Playing against one piece, it simply needs to be 'chased' by the opponent, as it will eventually hit the edge of the board and be forced to make a turn, where it will not be able to escape the boom radius if an opposing piece is right behind. Playing against two pieces, the other player can form three stacks on one column/row with two squares space between stacks, and advance the line one square at a time towards the opponent - the opponent will be unable to move on to the other side of the line.

Since there exist relatively simple strategies for winning from these positions, it was decided to make an algorithm to follow the above steps, rather than rely on a time consuming search. A large disadvantage of searching in this sort of situation is that the opponent may be on the other side of the board, and any winning moves could be beyond the search horizon of the player with piece advantage. Searching also takes significantly more time, which may be in limited supply when playing games.

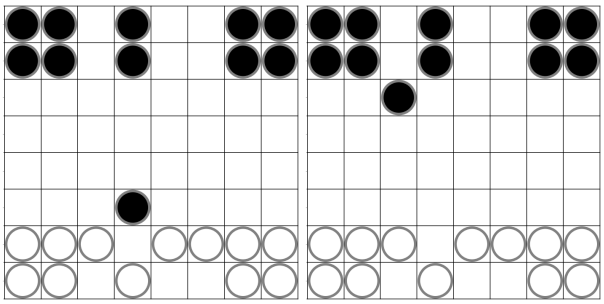
The endgame strategy was also noted to be effective against an opponent with all of their pieces in a single stack, and so was extended to be used instead of a regular search in these situations, no matter what stage of the game.

3. Features

Creating a vector of features for a given state, the previously defined $\phi(s)$ is necessary to evaluate a heuristic for the utility at that state for use in search by the agent. A discussion of considerations as well as features used in the evaluation follows.

The board state was internally represented by a two dimensional array, where each element was an integer $\in [-12, 12]$ to represent that height of the stack at that position, with sign representing colour. For example, the first column of the initial state would be represented as $[1, 1, 0, 0, 0, 0, -1, -1]$. Many chess engines[2] add a flattened one dimensional representation of the board as additional input into the evaluation function as part of the feature vector $\phi(s)$. Lai[2] suggested that a bitboard representation alone is insufficient as input to an evaluation function because states with similar bitboards can have extremely different real values. To see why, consider the Fig 3.1 where the left state is about to be won by black, whilst the right state is fairly even despite almost identical bitboards.

Figure 3.1 Similar bitboards



3.1. Chosen Features

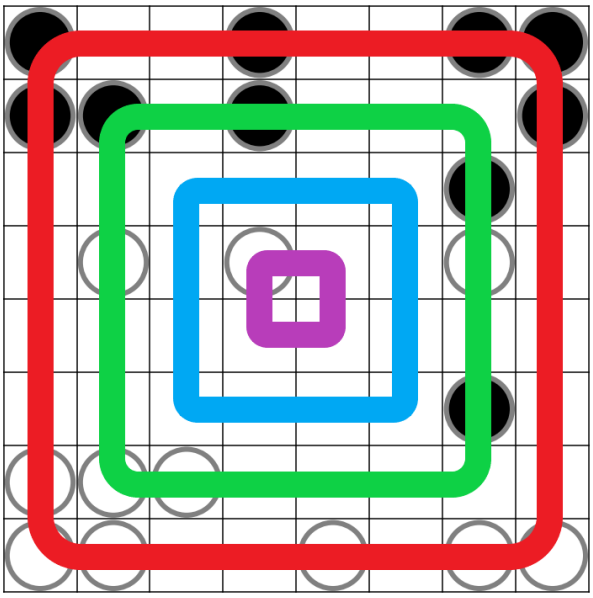
In chess, features are typically categorised into five groups: Position, control, mobility, protection and threat. These categories were the inspiration for many expendibots features. Simple features used as part of the evaluation include the piece counts, stack counts, number of actions and average stack size. Whilst the optimal stack count/size isn't immediately apparent, it seems clear that having all pieces in one stack is strategically different to having as many stacks as possible, so these stack related features will likely be useful to evaluate. Row counts (rows measured as distance from starting row) of both pieces and stacks were also considered as some indication of how desirable it is to have pieces and stacks away from the starting row, but it was found that they were not utilised by the agent after training, so they were removed to allow for faster computation.

The largest connected cluster feature finds the largest number of stacks a player has that would be lost if one stack were to boom, ignoring opposing pieces. The motivation behind this feature is that the bot should be discouraged form having largely connected clusters of stacks, as they are vulnerable to being pinned by a good opposing move. The reason stacks are counted and not pieces is that there is a large strategic difference between a single stack of four, and four single adjacent stacks (such as in the starting position). The former can move away from danger much more readily, and thus should not be counted equivalently in this context.

The closest opposing pieces feature finds the minimum Manhattan distance between any two opposing pieces. This feature is inspired by the fact that with a shallow search, if both players are on opposite sides of the board, they can be oblivious to actions that interact with one another. In an aggressive game, this value would be minimised by a player seeking out the opposition, though when down, it would be best to minimise and hope to force the opponent into a draw. The average closeness feature draws on similar ideas, but it was decided to let the machine learning algorithm determine the relative utility of these features, rather than choose one or the other.

Centrality features were made with the strategic thinking that the middle of the board is a desirable location, as a stack in not pinned by the side of the board. Centrality was defined by rings, where the board was divided into four rings (Fig 3.2).

Figure 3.2 Ring Centrality



Counts of both pieces and stacks in each of the four rings were decided to be used as features, with the hope that it could be learned what level of centrality is desirable.

Mobility was calculated as the number of unique squares a player can move onto. It would be thought that in general having a greater number of potential places to move is more desirable as it allows for more options. It is suspected that generally greater mobility and number of actions will be desirable, but is likely to be important strategic information in any case.

Threat was calculated as the number of opposing stacks adjacent to the agent's pieces. The strategic value here is fairly clear: it is desirable to have this number higher for the opponent than for us, by threatening more opposing pieces than the agent has pieces threatened. Connectivity is a closely related feature, but ignores opposing pieces, and instead measures the number of pieces adjacent to others of the same colour. This is likely desirable to be minimised, as adjacent pieces are vulnerable to recursive booms.

3.2. Feature Vector Additions

It was found that some features such as number of pieces that a player had were being weighted negatively. This is because in order to win the game and boom the opponent's pieces, one must sacrifice their own pieces. However, if an agent puts a negative weight on the number of pieces that it has, it will explode its own pieces without trading. To account for this kind of behaviour, additional features derived from a difference between features for each player were created.

$$f_{\text{diff}} = f_{\text{white player}} - f_{\text{black player}}$$

Note that the difference features are very inexpensive since the features that they are derived from have already been calculated. In the case of describing behaviour where losing pieces is desired, but only in sacrifice for an advantageous trade, an agent can now learn negative weights for both its piece and its opponent's and have a very positive weight for the difference between these two values. This encourages an agent to find advantageous trades that increase its lead.

Some features such as closest opponents piece have a different real utility depending on which player is winning. For example, if white has one piece, and black has two, white should be trying to maintain as much distance as possible from the black pieces to try and force a draw. Meanwhile, black should be trying to close the distance to white and boom the remaining piece. This is not only a change in the magnitude of the weight's real utility, but also a change in sign. To account for this behaviour, an additional set of features was constructed depending on the sign of the piece advantage. The winning player was first calculated as:

$$\text{winning} = (\text{sign}(\text{n white} - \text{n black}) + 1)/2$$

such that winning $\in \{0, 1\}$. Then for each standard feature, the following two features were added.

$$f_1 = \text{winning} \times f$$

$$f_2 = (1 - \text{winning}) \times f$$

Much like the difference features above, these new features are inexpensive because the base features have already been calculated. In the case of f being closest opponent piece, an agent can now learn an associated positive weight when losing (i.e. a general desire to increase the distance to the closest opponent piece) vice versa when winning.

The difference in features between players, and the winning/losing features were created for all of the features detailed in the previous section.

4. Training

4.1. Training Algorithm

With many features, it is difficult to choose a θ that accurately evaluates a state. Thus, learning algorithms are turned to. Preliminary research involved investigating TDLeaf(λ), first introduced by Baxter *et al*[3] in the domain of chess. The algorithm updates θ such that the evaluation of a principal leaf state of a given minimax search moves towards the evaluation at the principal leaf of the next minimax search - the temporal difference that makes up TDLeaf.

Veness *et al*[4] compared several training algorithms for a chess evaluation function, and found superior performance with less training using a learning algorithm known as treestrap. A more primitive form of this algorithm known as rootstrap was also implemented, which shares some similarities with TDLeaf(λ). In rootstrap, θ is updated by having the evaluation of the root state move towards the evaluation of the leaf node in a given minimax search. An advantage of this over TDLeaf(λ) is that treestrap guarantees that the evaluation is on the same variation. As noted by Veness there will at times be a change of variation with the deepened search at the next search, and so the two leaf nodes may represent very different states, and it does not make sense to update the leaf of a given search with the lead of the next in this situation.

Treestrap builds upon the rootstrap algorithm, using the fact that when searching to a depth of greater than two ply, information is being thrown away that could be used for updates. A search to depth four (two turns) with rootstrap will only have one update being the root towards the principal leaf. However, every state at depth two has a principal leaf for what is a effectively a depth two minimax search from that state as a root node. This state can also be updated towards the value of its principal leaf. Thus every state matching the player of the root state is updated towards the principal leaf from that state. For this to work, the search depth must be even. This idea can be extended to updating the states of the opponent's plays. Updates can also be made for evaluations at the opponent's turn in the search tree.

Veness'[4] work suggests that online play is crucial when training a high performing agent, however, this is infeasible for a widely unknown game such as expendibots. Instead, they recommend giving an agent good initial weights rather than random weights to begin training. To address this, hand-picked weights were allocated to all agents before training began.

Unlike TDLeaf(λ) which updates all state evaluations to reduce the temporal difference to future state evaluations (with behaviour depending on λ), treestrap can update θ at each move. During the minimax search, the heuristic value of states s and their principal leaves s' are saved. After the search, $\Delta\theta$ is set to zero, then updated for each of these pairs according to the following formula:

$$\Delta\theta \leftarrow \Delta\theta + \alpha(H_\theta(s') - H_\theta(s))\phi(s)$$

Note that α is simply the learning rate. Following this, θ is set to the value $\theta + \Delta\theta$.

This formula is derived as the stochastic gradient descent of the sum of squared errors of the evaluation at a state and the evaluation at its principal leaf.

$$\Delta\theta = -\frac{\alpha}{2}\nabla_\theta \sum_{s\in T} (H_\theta(s') - H_\theta(s))^2$$

$$= \alpha \sum_{s\in T} (H_\theta(s') - H_\theta(s))\phi(s)$$

Of course this formula is slightly inaccurate since H_θ is not a continuous function (see definition on pg.1). Taking suggestions from Real and Blair,[5] some additions were made to this update formula. A scaling constant $\lambda \in (0, 1)$ is taken to the power of the depth d of the state. This essentially assigns an exponentially lower learning rate to positions further away from the root. This is based on the assumption that nodes closer to the root are both more realistic, and have a more extensive subtree underneath them, therefore providing better quality information. The formula becomes:

$$\Delta\theta \leftarrow \Delta\theta + \alpha(H_\theta(s') - H_\theta(s))\phi(s)\lambda^d$$

Additionally, the amount that each element of θ was allowed to change by was limited by constant $M > 0$.

$$\Delta\theta_i = \max(\min(\Delta\theta_i, M), -M)$$

A minimax search of depth four is incredibly time consuming considering the branching factor of expendibots, and so the next extension of the algorithm takes the form of using an alpha beta search instead of minimax. This adds some complexity in that not all evaluations are exact, rather some give an upper or a lower bound. This information can still be used however, if the bounds fall the right way. For example, take a state s with principal leaf s'. If $H_\theta(s) = 4$ and $H_\theta(s') \geq 6$, the root evaluation could be updated towards the leaf. However, if $H_\theta(s') \leq 6$, $H_\theta(s)$ could be accurate, and therefore it should not be updated. This form of treestrap – treestrap($\alpha\beta$) – is what was used for training the weights of the evaluation function, searching to a depth of four (ply). See Figure A.1 for further description of this algorithm's behaviour.

Because an alpha beta search relies upon good move ordering to prune effectively and speed up search, starting training with random weights was unlikely to be efficient. Rather, rootstrap training was used to a depth of two on random weights with tweaks to key features (positive weight for piece ratio etc.) to train some weights to a reasonable level of play, before allowing the alpha beta treestrap to take over, once the search was able to make use of pruning.

Another key element to training a successful agent is ensuring that training instances are both diverse and representative of tournament experience. Diversity encourages learning agents to generalise, avoiding overfitting, and experiences should be representative rather than simply randomly generated boards such that the agent is learning weights that will be useful in tournament.

To ensure diversity, each training game had a 0.5 chance of starting with 32 random ply, 0.25 chance of 16 random ply, 0.125 chance of 8 random ply etc. This weighting led to more training games having more randomness, but still including games that start from the beginning such that agents have a chance to learn how to open.

Utilising self-play was the best way to ensure that training instances were representative. Furthering this, each player had an independent one in four chance to use the opening book. Since the player is using the opening book, this allowed the agent to train on more realistic scenarios that it was likely to encounter in tournament.

4.2. Stages

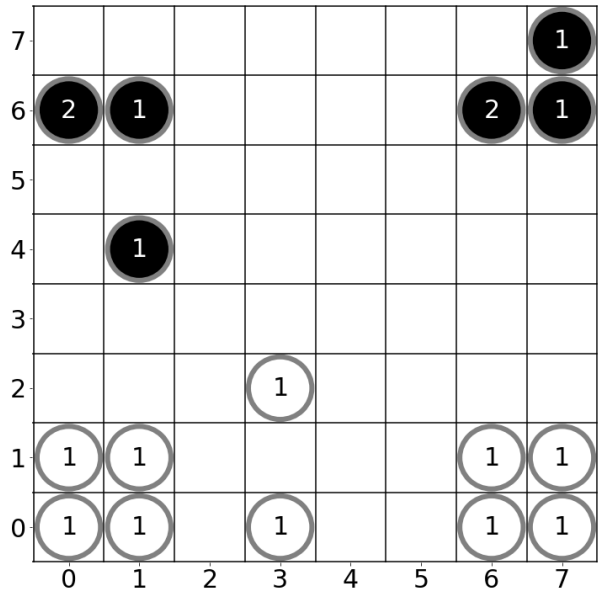
The agent's poor initial weights resulted in most training games ending in a draw, encouraging the evaluation function to predict 0. This led to the inspiration to split a game of expendibots into smaller stages, each with their own rewards and terminal states. Intermediate rewards are a commonly regarded as a poor idea in the wider artificial intelligence community as they can lead an agent to make undesirable actions to receive a high reward. Despite this, the intermediate rewards allowed the agent to optimally make non-zero evaluations, which was an improvement.

The game was split into four stages as follows: Opening (the initial stage), Developing (once the second boom had occurred), Mid-Game (once the fourth boom had occurred) and Ending (overrides all other stages, occurs when both players have four or less pieces). Moving from one stage to another is represented as a terminal state for a searching agent, and the corresponding utility is:

$$\text{sign}(\text{piece advantage}) \times 100 + \text{piece advantage}$$

It was decided that the developing stage should begin after the second boom instead of the first to counteract the horizon effect. If the the terminal utility was given after the first boom, then an agent could make a favourable trade (one for two) and perceive this as being very good action despite the opponent making a devastating retaliation (one for eight). Changing the terminal state to be after the second boom isn't perfect though. In Fig 4.1, white has made the first boom favourably trading two for four. White can now reach the opening terminal state and receive a large positive reward for the action $(\text{"BOOM"}, (2, 3))$ since this will still maintain the piece advantage.

Figure 4.1 Intermediate reward entices boom



4.3. Transposition Tables

A technique used successfully to speed up training was the use of transposition tables. The feature vector, $\phi(s)$, of previously seen states was memoised so that it would not have to be recomputed during the same game when it was seen again. This is particularly useful in cases where the players move back and forth, repeating positions. Further benefits are seen in deeper search trees, as the same states can be reached from different sequences of moves. The implementation of these tables during training allowed for more games to be played in training in a shorter span of time, thus training faster.

Making a hash of a game state is a potentially expensive process, so Zobrist hashing[6] was investigated as a potentially faster hashing of game states, as opposed to python's inbuilt hash function operating on a string of the board array. Zobrist hashing works by generating a random number (e.g. an unsigned 64 bit integer) corresponding to every type of piece for every position of the board. For expendibots, there are 12 possible pieces one can have on a given board state (a stack of size 1 to 12 inclusive), two players worth of pieces, and 64 board positions. Thus 1536 random numbers are generated, and the game state is also represented as an unsigned 64 bit integer, initially zero. The hashing works due to the self inverse property of the logical XOR (exclusive or) operation. When a stack of size one moves from (0, 0) to (0, 1), an XOR is performed with the number associated with that players stack of size one at (0,0), and the game hash. Then an XOR is performed with the number associated with that players stack of size one at (0,1), and the game hash. Thus a new hash is calculated with few operations, given the previous hash.

Zobrist hashing has been used effectively in chess and chess like games,[5] however there are a few reasons why it isn't as quick to calculate a hash of a game state in expendibots. In chess, if a piece moves no piece is left behind, unlike in expendibots where a stack can split, requiring more XOR operations to calculate a new hash. Furthermore, boom moves require potentially many XOR operations to get to the new state. It was found that calculating the Zobrist hash to use as a key for memoised board states was slower than simply using pythons inbuilt hash function for lookup, so Zobrist hashing was ultimately not used in training for expendibots.

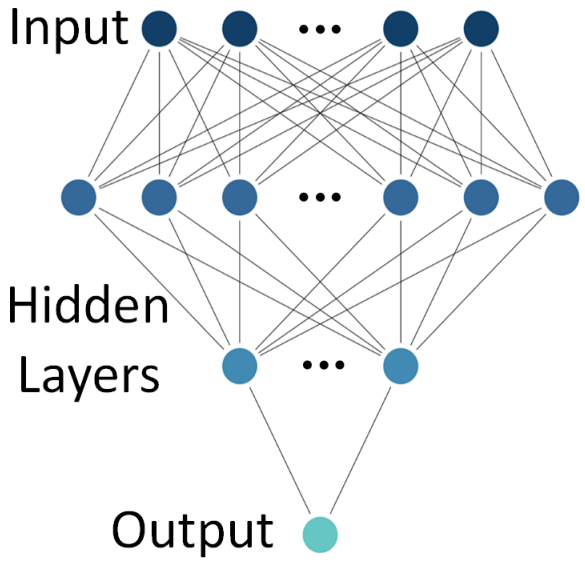
4.4. Multiprocessing

A technique used to gain a substantial performance boost when training with a minimax tree was the use of multiprocessing. Python is only able access a single CPU thread, and so when training on a machine with four cores, each with two threads, training can use at most one eighth of the CPU capacity. Pythons default multiprocessing library was used to delegate expanding different branches of the search tree from the root node to different processes, allowing the full capacity of the CPU to be used. The eight times speed improvement was not observed (which would have been considered the best case scenario). Performance was likely hampered to some extent by the necessity of sharing the large transposition table between the processes, and it is also likely that the processor was unable to overclock to the same extent with a larger portion of the CPU engaged. Regardless, the two times speed improvement allowed for twice as many games to be played in the same amount of time when training with a minimax tree.

4.5. Neural Networks

Inspired by the self-learning chess engine *DeepFish*,[7] neural networks were implemented using PyTorch. A dense network with two hidden layers (1000 and 100 nodes respectively – Fig 3.2) was initially chosen to mirror the architecture used in both DeepFish and Giraffe[2] chess engines. It was found that this architecture beat both networks with smaller hidden layers and networks with only one hidden layer. Although one hidden layer is sufficient for capturing linear relationships, the reduction in performance may imply that there are non-linearities between features and utilities. ReLU was chosen as the activation function for the hidden layers because it both simple to implement and effective at capturing linear relationships. Hyperbolic tan was chosen for the output activation function as it could predict a win, draw or loss as $\{1, 0, -1\}$ respectively. The utility of terminal states was linearly transformed from $[-100, 100]$ to $[-1, 1]$ to better align with this range of prediction.

Figure 3.2 Neural Network Architecture



In addition to the standard features, the bitboard representation was passed into the network. Two bitboard inputs were trialled. Many chess engines pass a different bitboard for each type of piece into their evaluation functions. Mirroring this, one bitboard representation was 12 separate bitboards, one for each possible stack size. The second bitboard representation was simply the size of each stack at each board position as an integer, where sign represents colour. This was then divided by 12 to normalise the values between $[-1, 1]$. Both representations gave similar results. Given a bitboard, treestrap was found to overfit with a linear evaluation function. Neural networks are likely prone to this same problem as Real and Blair found.[5] It is difficult to determine whether or not the expendibots neural network overfit as it did not appear to have any preference for specific board positions like the overfit treestrap agent did. Many chess engines that use neural networks train on the plethora of games played by human experts, however, this was not an option for expendibots. Instead, the network's evaluation at search tree nodes and corresponding principal leaf nodes was stored and training was completed after each game. The training pairs were shuffled, and training was performed in batches of 32. Loss was calculated as the mean squared difference between the heuristic evaluations, and this was back propagated through the network using the ADAM optimiser.

Because the network was trying to minimise the loss, it learnt to predict 0 most of the time since this would result in a small mean squared error. The network would often be correct since its poor evaluation resulted in draws, further reinforcing the bad evaluation. It is possible that this issue could be averted by assigning negative values to draws, such that the network's near-zero predictions are not reinforced by the terminal utilities. In addition to this, training instances from states near the end of the game could be duplicated such that these instances have greater effect of training.

It is hypothesised that further work would have found that the neural network is superior to the linear evaluation function, as the limited work yielded human-like play. However, the attained results failed to beat the standard treestrap agent, so the neural networks were not implemented into the final agent.

5. Conclusion

A typical graph of a day's worth of training is displayed in Figure A.2, with final feature weights for each stage in Figure A.3. It can be seen that convergence was not reached, suggesting that more training was required. It was assumed that more training always led to better agents, and no evidence of overfitting was found.

Whenever a new modification was implemented, the new agent played 100 games against its predecessor, each agent starting 50 games. Because most agents chose actions deterministically, a random number of initially random turns was chosen each game for both agents. If the new agent's winning proportion was statistically significantly greater than 0.5 it would be deemed superior and the modification was kept. If the modification led to an agent losing statistically significantly less than 0.5 the modification was either altered or discarded. To pick a final agent, several training instances were performed from human picked initial weights to find several different local minima.

In 100 games against a random player, the agent was able to win every time, suggesting that it has the capacity to avoid bad situations and hunt down opposing pieces to secure victory. However, the agent was not effective against the authors, being unable to defeat human play. Although the opening book secures a fantastic initial position, the agent seems to be unable to make moves that will have long-term benefits. This is only to be expected with a two ply search. It is expected that the expendibots agent will be similar in strength to other $\alpha\beta$-pruning agents searching to a depth of two.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. 2010.

[2] M. Lai, "Giraffe: Using Deep Reinforcement Learning to Play Chess," *arxiv.org/abs/1509.01549v2*, 2015.

[3] J. Baxter, A. Tridgell, and L. Weaver, "TDLeaf(λ): Combining Temporal Difference Learning with Game-Tree Search," *Australian Journal of Intelligent Information Processing Systems*, vol. 5, no. 1, pp. 39–43, 1999.

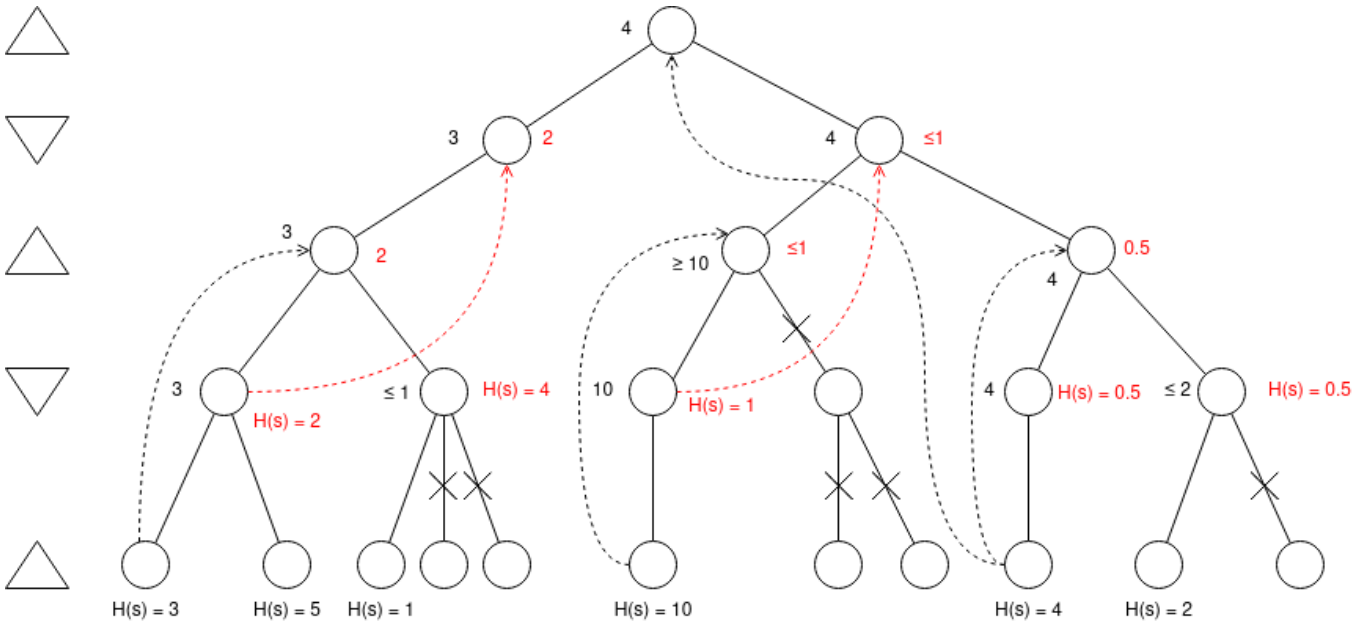
[4] J. Veness, A. B. David Silver and, and W. Uther, "Bootstrapping from game tree search," *Advances in Neural Information Processing Systems*, vol. 19, pp. 1937–1945, 2009.

[5] D. Real and A. Blair, "Learning a Multi-Player Chess Game with TreeStrap," *IEEE Congress on Evolutionary Computation*, 2016.

[6] "Zobrist hashing," *Chess Programming Wiki*, 2020.

[7] Dyth, "DeepFish GitHub," *https://github.com/dyth/deepfish*, 2018.

Figure A.1 – Alpha-Beta Treestrap Weight Updates



The alpha beta search tree above is intended to illustrate which states are used in weight updates in the implementation of alpha-beta treestrap. This tree is significantly oversimplified compared to one that would be created in expendibots training; it serves to demonstrate the updating procedure. The player is indicated with the triangles to the left: max is selecting a move in the root state. On the last layer, the value of the evaluation function from the max player's perspective of the board is shown in black. The alpha-beta search values that propagate up from this layer are shown in black to the left of the states. On the second to last layer, the value of the evaluation function from the min player's perspective is shown in red. These values also propagate upwards, accompanied by an upper or a lower bound where necessary – including potentially on a principal variation, unlike the alpah-beta values for max in black.

Weight updates occur for every state that is not in the bottom two layers of the search tree. Each state is updated towards the value of the principal leaf at even depth for that state, with the principal leaf being evaluated from the same perspective of the player who is to move at a given updating state. These update relationships are shown with dotted arrows in the diagram; black arrows for updates for states from max's perspective, red arrows for updates for states from min's perspective.

An important detail to note is that some updates are conditional. Considering min's state one ply down from the root on the right, the search on max's alpha-beta tree has returned a bounded value, rather than an exact one. The principal leaf will take on a value of less than or equal to one, but no more. If the heuristic value of this state were to be three, it could be updated towards the bound of one since it is the correct direction, even though it is not the correct value. However, if the heuristic value of this state were to be zero, it cannot be known if the principal leaf would be less than or greater than zero, so the update direction is also unknown. In this case, no update occurs. This generalises to bounds in both direction, for either player.

Figure A.2 – Feature Weights for each Stage During Training

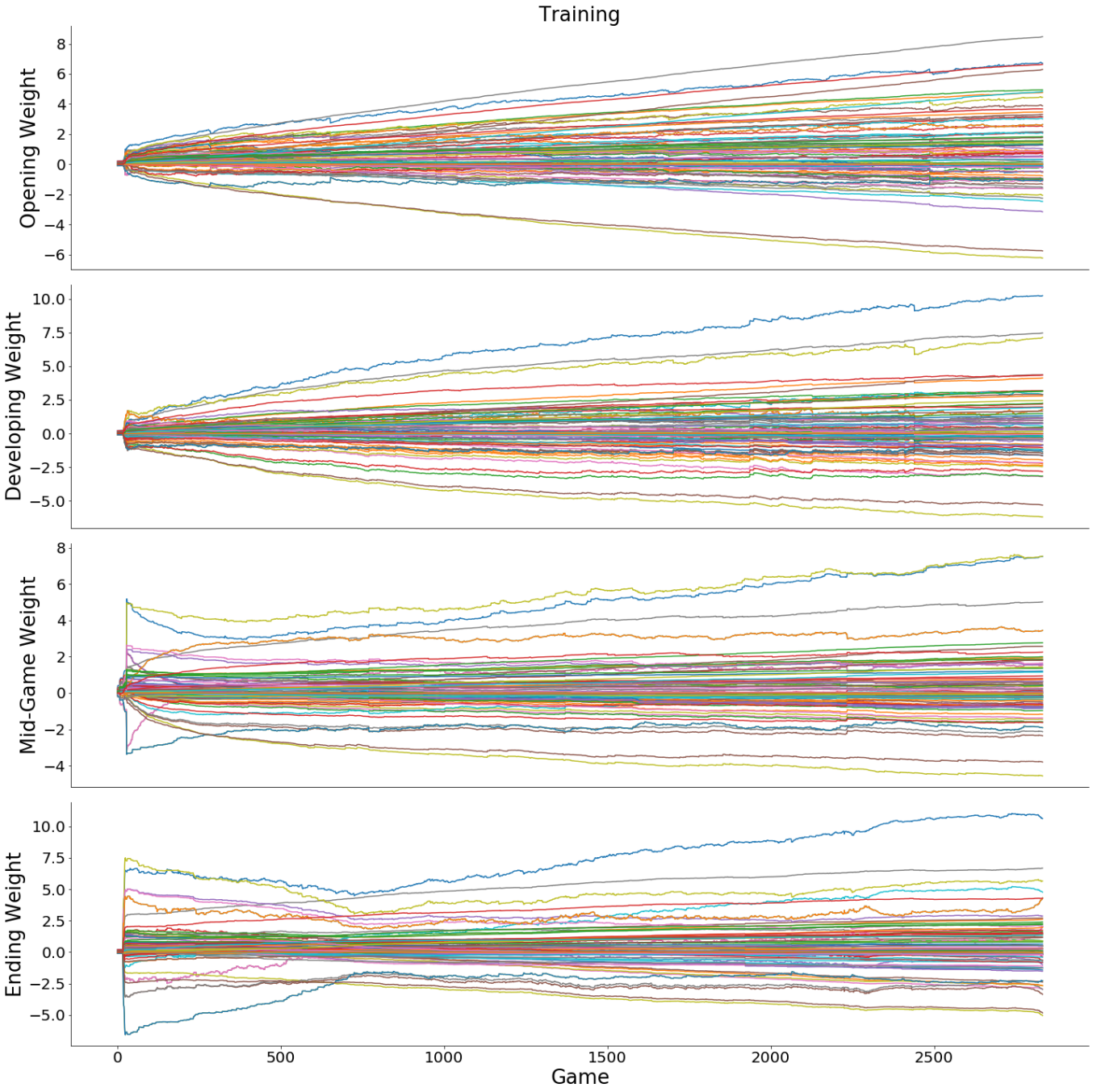


Figure A.3 – Feature Weights for each Stage after Training