Flounder: an RL Chess Agent

Andy Bartolo, Travis Geis and Varun Vijay
Stanford University

We implement Flounder, a chess agent using MTD(bi) search and evaluation functions trained with reinforcement learning. We explore both linear and nonlinear evaluation functions. We discuss differences between Flounder and other chess engines, including Stockfish and Sunfish.

■

1. INTRODUCTION

In 1950, Claude Shannon's seminal *Programming a Computer for Playing Chess* introduced the world to the concept of the modern chess-playing agent. Shannon presented minimax search as a natural means of making competitive moves, coupled with an evaluation function to guide the search algorithm. In the decades since, computer chess has evolved into a major field of research, with both more efficient search algorithms and more powerful evaluation functions. Today, chess serves as a classic benchmark for new search, evaluation, and learning techniques.

2. PROBLEM MODEL

We model chess as a two-player game with perfect information: the entire board is visible to both players so opponents can assess each other's possible moves according to minimax search when deciding an optimal policy. Chess's high branching factor means that search algorithms must use optimization techniques like alpha-beta pruning to avoid searching the game tree exhaustively. Players receive a reward only upon winning a game. The values of non-terminal states may be used as heuristics for the minimax evaluation function, but they do not contribute directly to players scores.

Each state in the game tree represents a board position combined with the color of the current player. The board position also includes information about which players can castle and whether en-passant capture is legal. The current player seeks to maximize his own reward while minimizing that of the opponent. Given a board position and player, possible actions are the legal chess moves available to that player. The value of a given board position is determined by the combination of a feature extractor and corresponding feature weights. We employ reinforcement learning to discover the weight of each feature.

2.1 Baseline Implementation

To establish a performance baseline for our agent, we implement a baseline agent that makes random legal moves.

2.2 Oracle Implementation

To understand the upper bound of performance we expect from our agent, we consider two existing chess engines, Stockfish and Sunfish.

Sunfish is a simple chess engine written in Python and designed for readability and brevity [Ahle 2016]. It uses a scoring function based on lookup tables. For each piece type, Sunfish has an 8×8 table of integers mapping board positions to the value of the piece in each position. The simplicity of the piece-square tables contributes to Sunfish's speed, particularly because the tables allow incremental adjustments to the value estimate of a board position during search. As the engine tries moves and backtracks, it can add and subtract the value of the moved piece from the tables rather than recomputing the value in closed form [Wiki 2016]. Sunfish uses the MTD(bi) search algorithm.

Stockfish is currently the most powerful open-source chess engine. It excels at deep game tree search and includes advanced domain knowledge like endgame move tables. It is implemented in C++ and uses a large dynamic programming cache and aggressive search-tree pruning to achieve its high speed [Romstad and Kiiski 2016].

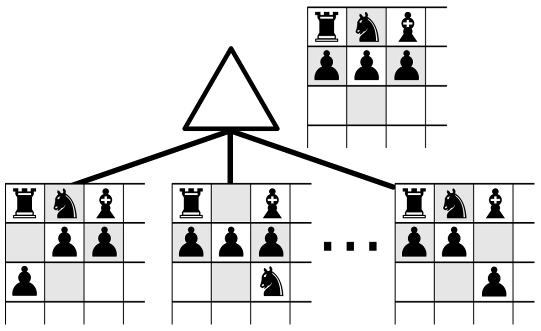

Fig. 1: An illustration of minimax search for the optimal move. Actions are legal moves, and states are board positions. The agent seeks the next board position with the highest expected utility.

2.3 Evaluation of Baseline and Oracle

To evaluate the performance of the baseline and oracle agents, we employ the Strategic Test Suites, thirteen sets of 100 board positions labeled with the optimal move from each position [Corbit and Swaminathan 2010]. We give each of the oracle engines 500ms to decide each move. Table I shows the number of moves each agent chooses correctly for each of the thirteen test suites. For the random agent, we average the number of correct moves over 100 runs of each test suite. We discuss these results in more detail in Section 5.3.

The test suites allow fine-grained evaluation of the strength of each engine in different scenarios. To understand more broadly the disparity in power between the baseline and the strongest oracle, we simulate games wherein Stockfish plays against the random agent. As expected, Stockfish defeats the random agent in every game. We also measure the length of the game in moves, in order to gauge how much stronger Stockfish is than the random agent. Over 50 games, the average game length is 25.66 moves. Because Sunfish does not use the python-chess API, we had difficulty using it in games against our other agents; instead, we use it only as a reference for the Strategic Test Suites.

Test Suite	Random	Stockfish (500ms)	Sunfish (500ms)
STS1 (Undermining)	2.69	89	13
STS2 (Open files and diagonals)	3.04	81	16
STS3 (Knight outposts)	3.07	77	23
STS4 (Square vacancy)	2.72	77	6
STS5 (Bishop vs. knight)	2.52	81	34
STS6 (Re-capturing)	2.83	78	21
STS7 (Offer of simplification)	2.89	75	9
STS8 (Advancement of f/g/h pawns)	2.52	77	9
STS9 (Advancement of a/b/c pawns)	2.71	75	7
STS10 (Simplification)	2.62	82	42
STS11 (Activity of the king)	3.23	74	7
STS12 (Center control)	2.49	79	13
STS13 (Pawn play in the center)	2.49	80	15

Table I. : Results of running the baseline and oracles on the Strategic Test Suites. Each cell shows the count of moves chosen correctly. For the random agent, the count of correct moves is an average over 100 runs of each test suite.

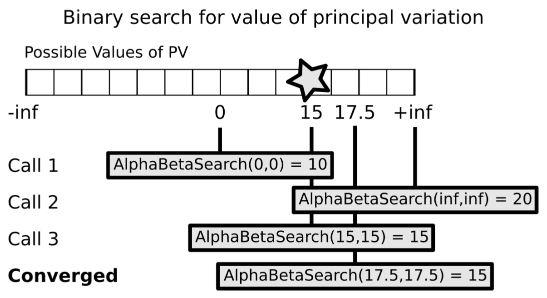
Binary search for value of principal variation
Possible Values of PV
-inf
0
15
17.5
+inf
Call 1
AlphaBetaSearch(0,0) = 10
Call 2
AlphaBetaSearch(inf,inf) = 20
Call 3
AlphaBetaSearch(15,15) = 15
Converged
AlphaBetaSearch(17.5,17.5) = 15

Fig. 2: An illustration of the operation of the MTD(bi) algorithm searching for a principle variation with value 15. Through repeated calls to null-window alpha-beta search, MTD(bi) performs binary search and converges on the value of the principal variation on the fourth iteration.

3. SEARCH TECHNIQUES

The main work of the chess engine is to search for the optimal move to make from the current board position. The optimal move is defined to be the move that maximizes the expected utility of future game states until the end of the game. Equivalently, it is the move most likely to lead to victory.

Chess engines have used many different search algorithms to navigate the large search space required for determining a move. Because the branching factor is so large, chess engines must prune the search tree for reasonable search runtime.

We began by implementing standard minimax search with alpha-beta pruning, which we found to be too slow. We settled on an algorithm called MTD(bi), which builds upon alpha-beta search and can achieve faster runtime by exploiting its properties.

3.1 Alpha-Beta Search

Alpha-beta pruning improves on standard backtracking search by avoiding certain parts of the search space. This search algorithm explores the game tree depth-first, visiting children of the root node in left-to-right order. It maintains upper and lower bounds for the minimax value of the principal variation in the game tree, and prunes subtrees outside those bounds as the bounds tighten. In the context of chess, alpha-beta search allows the chess engine to consider only a subset of all legal moves while still guaranteeing correctness.

For example, suppose the white player performs minimax search to a depth of 2 plies (white's move followed by black's countermove). Consider white's decision between two possible moves, A and B. Evaluating move A, white concludes that black will not be able to capture any piece in his countermove. Evaluating move B, white discovers that one of black's countermoves is to capture a piece. White can immediately stop considering other possible scenarios after taking move B, since move A is already guaranteed to be better than move B. The white player prunes the boards that follow move B from the search tree.

3.2 Caching with Transposition Tables

We employ transposition tables to avoid duplicating calls to our evaluation function. A transposition table is a hash table mapping game states to their values. During minimax search, we first consult the transposition table to determine the value of a board position. If the value is in the table, we fetch it in constant time; otherwise, we compute the value according to standard minimax search.

3.3 MTD(bi)

Alpha-beta pruning alone reduces the size of search tree, but further reduction is possible. The MTD(bi) algorithm is part of the Memory-Enhanced Test-Driver family of algorithms introduced by Plaat et al. in 1995 [Plaat 1995]. It is equivalent to Coplan's C* algorithm, introduced in 1982 [Coplan 1982]. The MTD algorithms exploit properties of alpha-beta search and transposition tables to reduce the work needed to find the principal variation (PV).

The key innovation of the MTD algorithms is in calling alpha-beta search with a null window (where alpha and beta are equal, and represent a guess at the value of the PV). The authors call the combination of null-window alpha-beta search and transposition tables "memory-enhanced test" (MT). The outcome of MT can be to find the PV with precisely the value of the guess, or to discover that the value of the PV is higher or lower than the guess.

By repeatedly performing null-window alpha-beta search with different guesses, MTD algorithms converge on the true value of the principal variation. The distinction among algorithms in the family arises in the choice of the next guess. For example, MTD(f) begins with an arbitrary guess. If alpha-beta search at the guess discovers that the guess is too high and the real upper bound is b, then the next guess is $b-1$. If the guess was too low and the lower bound is actually a, then the next guess is at $a+1$.

Many chess engines use MTD(f), but one key assumption of the algorithm is that the evaluation function takes on only integer values. If the value of the PV can lie between a and $a+1$, then taking a step of size 1 might overshoot the target.

Because we employ reinforcement learning to determine feature weights, our evaluation function can take on non-integer values. Therefore, we use the MTD(bi) algorithm instead of MTD(f). In MTD(bi), each guess at the value of the PV establishes a new upper or lower bound on its true value, as in MTD(f). However, unlike in MTD(f), the value of the next guess is the midpoint between the new search bounds. MTD(bi) thus performs binary search over the possible values of the PV, meaning the step size can be arbitrarily small. When the upper and lower bounds are equal, the algorithm has converged on the principal variation.

3.4 Iterative Deepening and the Killer Heuristic

The number of game subtrees pruned by algorithms like MTD(bi) depends on the ordering of the search over the possible next moves. Searching a subtree that is far from optimal will result in mild or no pruning, after which another search over a more optimal move is required. In contrast, searching near the principal variation first

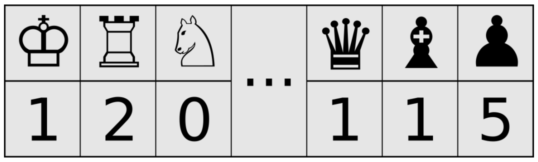

Fig. 3: An illustration of the material configuration feature, which includes the count of each type of piece on the board.

will result in tighter bounds on its value, leading to more extensive pruning and thus faster search.

To maximize the number of pruned subtrees, we wish to approximate best-first move ordering during search. We use iterative deepening combined with a "killer heuristic" to facilitate more rapid tree pruning.

Iterative deepening is a form of depth-limited depth-first search. The algorithm performs depth-first search at depth 1, then depth 2, and so on until the depth limit. On each round of iterative deepening, we observe which moves lead to the tightening of the alpha-beta bounds. These moves are called "killer moves" and are likely to be among the best in the next round of iterative deepening, a heuristic called the "killer heuristic" [Huberman 1968]. We store killer moves in a cache based on their depth in the game tree, and try them first on the next round of search when considering moves at the same depth.

Korf has shown that iterative deepening is asymptotically optimal in time and space requirements, as well as in the cost of found solutions, for exponential search trees [Korf 1985]. Its runtime complexity is $O(b^d)$, where b is the branching factor and d is the maximum search depth.

4. EVALUATION FUNCTIONS

We explore both a linear evaluation function and a neural network for state evaluation, and use TD learning to train each. Linear evaluation functions are easier to train and faster to compute, but cannot encode complex relationships between features. Neural networks can express nonlinear feature relationships but require more training data to learn those relationships. We use function approximation to learn generalizable patterns from board positions.

4.1 Feature Extractor and Linear Evaluation Function

To create a linear evaluation function, we apply a feature extractor Φ to the current board position x, resulting in a vector of feature values. We take the dot product of the feature vector with some weight vector w to get the approximate value $\hat{V}$ of the board position:

$$V_x \approx \hat{V}_x = \text{Eval}(x) = \boldsymbol{\Phi}(x) \cdot \boldsymbol{w} \tag{1}$$

Linear evaluation functions are efficient in time and memory use, but they can only encode a simple relationship between features. We must use many features to compensate if we wish to allow nuanced evaluation of the board position.

We borrowed ideas for our linear feature extractor from the experimental Giraffe chess engine [Lai 2015]. Giraffe uses a neural network to construct an evaluation function from its features, but our hope was that these features would be expressive enough for a linear feature extractor as well. We implemented the following features:

—Side to move, either white or black
—Castling rights for each player on either side of the king
—Material configuration, the number of each type of piece
—Piece lists, which record the existence and coordinates of each piece, and the value of the least-valued attacker of the piece
—A constant of 1 to allow for an intercept term

In total, these feature templates result in 146 features for each board position.

4.2 Neural Network

Neural networks allow us to encode more complex relationships between features, in theory allowing for a more nuanced estimate of the value of each board position. Prior chess engines including Deep Pink [Berhardsson 2014] and Giraffe use neural networks to construct their evaluation functions. Because neural networks can learn nonlinear relationships between features, many chess engines using neural networks can achieve good performance using simpler input features, allowing the neural network to infer higher-level implications of those features.

To try to overcome the limitations of a linear evaluation function, we constructed a neural network using a simple representation of the board, in addition to information about which player is to move, and castling rights for each player. We based our implementation on a similar system for playing Backgammon, described by Fleming. The most significant feature is the board vector, B, which is essentially a map of all the pieces on the board. B has dimensions $2 \times 6 \times 8 \times 8$, where $B[c][p][r][f] = 1$ if and only if the player with color c has a piece of type p at the board position with rank r and file f.

We feed the feature vector into a two-layer Multilayer Perceptron consisting of a first hidden layer with 1,024 units, followed by a Rectified Linear Unit non-linearity and a softmax classifier. The softmax output is interpreted as the probability of a white victory given the input board features.

4.3 TD Learning

The difficulty of approximating the value of board positions with function approximation is in choosing weights for each feature. We use temporal difference (TD) learning to discover the approximate weight of each feature.

TD learning blends concepts of Monte-Carlo methods and dynamic programming. Like Monte-Carlo methods, TD learning uses experience gleaned by exploring the state space to refine its estimate of the value of each state. However, TD learning differs in that it does not wait until the end of a session to incorporate feedback: it updates its estimate of state values on every state transition. To make these online value updates, it incrementally adjusts previous state values fetched from lookup tables, in the style of dynamic programming [Sutton and Barto 1998].

Chess's large game tree means that even after the observation of many games, the learning algorithm will not have observed most possible board configurations. To decide on optimal moves from states it has not observed, the learning algorithm must use function approximation for the evaluation function, rather than simply looking up the value of a board position in a table.

We use a specific form of TD learning called TD(λ). The idea of TD(λ) is to weight more heavily the contributions of states that contribute more directly to a reward. With function approximation, instead of weighting states more heavily, we increase the weight of the features more responsible for a reward.

To track which features participate more heavily in a reward, TD(λ) employs an eligibility trace vector $\boldsymbol{e_t}$ of the same dimensions as the weight vector $\boldsymbol{w}$. The elements in the trace vector decay on each state transition according to a rate governed by the trace-decay parameter λ. When the algorithm observes a reward, elements in the feature vector corresponding to nonzero traces are said to "participate" in the reward and their weights are updated.

The TD error for a state transition from S_t to S_{t+1} is given by

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w_t}) - \hat{v}(S_t, \boldsymbol{w_t}) \tag{2}$$

R_{t+1} is the reward for transitioning from state S_t to S_{t+1}. In the case of a linear evaluation function, $\hat{v}(S_t) \doteq \boldsymbol{\Phi}(S_t) \cdot \boldsymbol{w_t}$. The TD error δ_t contributes to the weight vector update according to the eligibility of each weight and the learning rate α:

$$\boldsymbol{w_{t+1}} \doteq \boldsymbol{w_t} + \alpha \delta_t \boldsymbol{e_t} \tag{3}$$

The eligibility trace begins at 0 at the start of a learning session, and is incremented on each state transition by the value gradient. It decays at a rate given by $\gamma\lambda$, where γ is the discount factor:

$$\boldsymbol{e_t} \doteq \nabla \hat{v}(S_t, \boldsymbol{w_t}) + \gamma \lambda \boldsymbol{e_{t-1}} \tag{4}$$

TD(λ) is a hybrid between pure Monte Carlo methods and the simple 1-step TD learning algorithm. When $\lambda = 1$, the eligibility of each weight falls by γ per transition, so the update will be the Monte-Carlo update. When $\lambda = 0$, only the features of the previous state participate in a reward. Increasing λ from 0 assigns more credit for a reward to states earlier in the session.

5. EXPERIMENTAL METHODS

5.1 Training the linear evaluation function

Because we use TD learning to find the optimal weight of each feature, we must train our system before it can play a game on its own. Starting from a completely unknown weight vector, we must "bootstrap" the weights to some reasonable values. After bootstrapping, we can train the agent further if desired by playing it against itself. There are two possible training methods.

We can train the system "offline," by allowing it to play many games and compiling a record of its experience, then applying in a single batch all of the updates to the weight vector. Offline learning offers the possibility of playing many simultaneous training games across multiple computers, because the weight vector does not change during each game.

Alternatively, we can train the system "online," using the reward observed after each state transition to update the weight vector immediately. Online learning incorporates feedback during each game, so the system can learn more quickly. However, it does not allow easy parallelization, because the weight vector could change after each move.

We began by attempting offline training, but it proved difficult, so we moved to an online learning approach. We attempted to implement training by self-play, but our agent did not play quickly enough to experience a meaningful number of games.

5.1.1 *Offline learning.* We began by attempting to implement offline learning, because of its possibilities of parallelization. We employed bootstrapping to obtain reasonable values for the feature weights.

There are multiple possibilities for bootstrapping. For example, David-Tabibi et al. bootstrap their genetic evaluation function using a stronger chess engine as a mentor: they take the value estimates of the stronger engine to be the ground-truth values of each board position [David-Tabibi et al. 2008]. Lai, the author of the Giraffe chess engine, uses a simplified feature extractor with knowledge only of material counts to initialize Giraffe's neural network before self-play [Lai 2015].

Initially, we sought to avoid using a stronger chess engine as a mentor for initializing our weight vector. Instead, we decided to use positions from recorded chess matches as training examples, with the winner of each match as ground truth for the value of each position.

Given many board positions $x_i \in X$, each labeled with winner $y_i \in \{-1, 1\}$, we can run stochastic gradient descent to find weights w which correctly predict the winner of a match given each board position. This stochastic gradient descent minimizes the logistic-regression training loss:

$$\text{Loss} = \frac{1}{|D_{train}|} \sum_{(x,y) \in D_{train}} \log\left(1 + \mathrm{e}^{-(\mathrm{w} \cdot \phi(x))y}\right) \tag{5}$$

In theory, finding some weight vector that labels many board positions with the correct winner implies that the weight vector contains generalizable knowledge of the advantages and disadvantages of each board position.

Unfortunately, using SGD to bootstrap our weight vector was less effective than we anticipated. While the loss function initially decreased over the first 1,000 training example games, the regression ultimately did not converge. We hypothesize that predicting the outcome of an entire match based only on one board position is too noisy for gradient descent to converge. In fact, it might be a harder problem than playing chess.

5.1.2 *Online Learning.* Instead of using gradient descent to train the weight vector offline, we can use a stronger chess engine as a "mentor" to train our agent online. We used our oracle engine, Stockfish, as a mentor. We simulated 4,400 games in which Stockfish played against itself, and our TD learning algorithm observed the board positions and rewards.

To expedite the training, we must show the learning algorithm decisive games with few ties, because only decisive games receive nonzero reward. We configure one of the Stockfish agents with a move-time limit of 100ms, and the other with a limit of 10ms. Using different move time limits ensures that one of the agents will generally be able to search the game tree more thoroughly, making it stronger.

Figure 4 shows the total number of moves our agent chooses correctly on STS for varying levels of training. After only a few training games, the agent outperforms the random baseline's average total score of 35.82. Subsequent increases in strength require many more training games, and the agent's performance on STS does not strictly increase. During training, we use a learning rate $\alpha = 0.001$ and a trace-decay parameter $\lambda = 0.75$.

Although Stockfish does not play deterministically, it is possible that using only Stockfish as a mentor provides example games with too much repetition, leading the learning algorithm to overfit to Stockfish's playing style. Overfitting would reduce the generality of learned board values and could result in decreases in general test suite scores like the one seen at around 1,500 training games.

As mentioned previously, we expect that a linear evaluation function will only generalize so far. At some point, the complexity of the knowledge of relationships between the pieces will exceed the expressive power of a linear combination. Such an upper limit could contribute to the sharp drop in test suite score at around 4,000 training games.

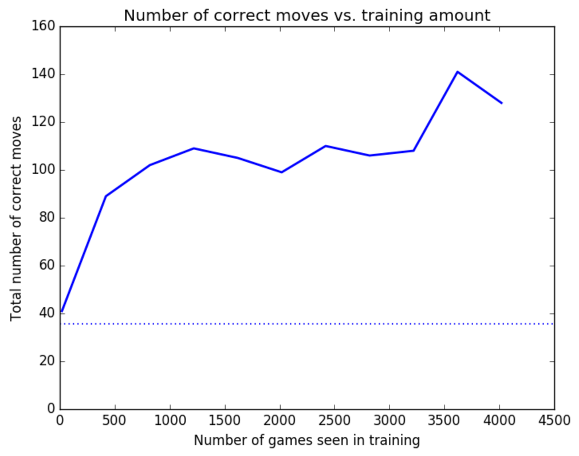

Fig. 4: Plot of the total number of correct moves our agent chooses on STS versus its level of training. The agent uses the linear evaluation function. The dotted line shows the average performance of the baseline random agent.

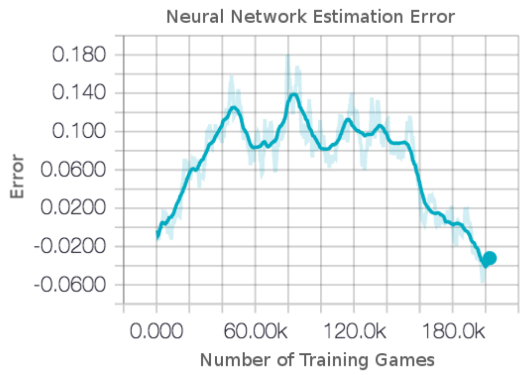

Fig. 5: Plot of an exponential moving average of the estimation error of the neural network model during training. The model is trained on a database of recorded human games using the TD(λ) update rule. After 120,000 training examples, the training error begins to decrease, suggesting that the network has learned to evaluate board positions more consistently. A positive estimation error indicates that the network underestimates the value of a position, while a negtive error indicates an overestimate.

Due to time constraints, we did not train the agent further than 4,400 example games.

5.2 Training the neural-network evaluation function

To bootstrap our neural-network evaluation function, we ran online TD(λ) on a database of recorded human chess games. We use TD(λ) only for value iteration, and rely on the sequence of moves made by the human players for control. Neural networks benefit from well-behaved cost functions during training, and we hypothesized that online training would be more effective than offline training for the neural network because the inclusion of the model's own evaluation in the TD(λ) update target makes the cost function smoother.

Figure 5 shows the estimation error of the neural network during training. We see an initial rise in estimation error, followed by a steady decline after a large number of training example games. The decrease in estimation error indicates the the neural network learned to evaluate board positions more consistently as it trained.

We decided to use our linear evaluation function instead of the neural network, because our agent could not finish games using the neural network. We believe that the neural network did not distinguish the value of board positions clearly enough, so our search function was unable to prune the search tree far enough for reasonable search times. The neural network's poor performance could be due to limitations of the input feature vector, or perhaps due to insufficient training examples.

5.3 Evaluating the Engine

Evaluating a chess engine is difficult, because there is no ground truth for the values of board positions in chess. The most evident measurable property of the game is the outcome. However, playing entire games is time-consuming and looking only at their outcomes masks the strengths and weaknesses of the engine.

5.3.1 *STS.* To achieve better resolution during evaluation, we use the Strategic Test Suites [Corbit and Swaminathan 2010]. The test suites consist of thirteen sets of 100 board positions each, with each board position annotated with the optimal move from that position. Stronger chess engines should be able to find the optimal move more often than weaker engines.

We use STS to evaluate our own agent using the linear evaluation function. We expect the oracle engines to score much higher than our own, partly because they incorporate more domain-specific knowledge of chess, and partly because they are mature projects. See Table I for the results of running STS on the oracles. Table II shows the results of testing Flounder against STS. As expected, Stockfish performs the best on most suites, followed by Sunfish and then our agent.

However, our agent scores near or above Sunfish on a few of the test suites. For example, consider STS 10, which tests board positions involving offers of simplification. Simplification is a strategy of trading pieces of equal value with the opponent, perhaps to reduce the size of an attacking force or to prepare a less complex endgame [Corbit and Swaminathan 2010]. Using search depth $d = 2$ plies, our agent outperforms Sunfish 51-to-42 on STS 10. At such a low search depth, our advantage likely indicates that our evaluation function estimates more accurately the value of board positions.

Also notable is the fact that our agent performs worse on this same test suite when we increase the search depth to $d = 3$ plies. It is difficult to identify exactly why the performance decreases so markedly. At depth 2, the agent only considers a single move and countermove, so a fair piece trade could appear trivially to be the most optimal situation. With an extra ply of search depth, it is possible to consider the agent's next move but not that of the opponent, so our agent might erroneously attempt to gain some further positional advantage without considering possible retaliation. We were unable to increase the search depth further, because doing so made our search function too slow.

5.3.2 *Moves until checkmate.* In addition to testing our agent with STS, we simulate games against Stockfish with our agent at various levels of training. If our agent learns generalizable knowledge as it trains, it should beat Stockfish, draw, or at least play longer games before Stockfish wins.

Table III shows the results of playing our agent against Stockfish. After 4,020 games of training, it outperforms the random baseline

Test Suite	Sunfish (500 ms)	Flounder d=2	Flounder d=3
STS1 (Undermining)	13	0	0
STS2 (Open files and diagonals)	16	0	2
STS3 (Knight outposts)	23	3	8
STS4 (Square vacancy)	6	4	5
STS5 (Bishop vs. knight)	34	32	32
STS6 (Re-capturing)	21	17	27
STS7 (Offer of simplification)	9	1	2
STS8 (Advancement of f/g/h pawns)	9	0	1
STS9 (Advancement of a/b/c pawns)	7	0	2
STS10 (Simplification)	42	51	35
STS11 (Activity of the king)	7	4	9
STS12 (Center control)	13	2	4
STS13 (Pawn play in the center)	15	0	2

Table II. : Results of running our engine at search depths 2 and 3 on the Strategic Test Suite. The engine uses the linear scoring function described in Section 4.1, trained by observing 4400 games of Stockfish with move time 100ms versus Stockfish with move time 10ms. For reference, we include the results for Sunfish from Table I.

agent, but only slightly. One possible explanation is that Stockfish is so much stronger than our agent that, in comparison, the progress our agent made in training is not enough to differentiate it from an agent without knowledge of strategy.

Another possibility arises from the nature of TD(λ) with a linear evaluation function. Recall that in TD(λ) with $\lambda \neq 1$, features observed later in the training session participate more in the reward. Thus, we might expect the learning algorithm to favor features it observes in the endgame. For example, in the endgame, it is often advantageous to move the king towards the center of the board, but exposing the king is not a good opening policy. With a linear evaluation function, it is difficult or impossible to learn both policies. If our agent has learned valuable knowledge of endgames but opens poorly, it will not likely have the opportunity to demonstrate its knowledge before Stockfish wins.

Training Level (games)	Moves until Stockfish wins by checkmate
20	28.1
420	26.1
820	29.5
1220	16.1
1620	24.9
2020	23.7
2420	26.3
2820	29.1
3220	27.3
3620	26.3
4020	30.5

Table III. : Number of moves until our agent loses by checkmate against Stockfish. The training level of the agent refers to the number of games it observed to train its weight vector in online TD(λ). The number of moves until loss is an average across 10 games. Recall from Section 2.3 that the random agent lasts an average of 25.7 moves until checkmate by Stockfish.

6. FUTURE WORK

The biggest limitation of our chess agent's strength is its slow search function. Profiling and optimizing MTD(bi), possibly by rewriting it in a more performant language would save a significant amount of time in the searches. Faster search would allow our agent to search to higher depths, meaning it could explore more future scenarios and call the board evaluation function closer to the leaf nodes of the search tree, which could improve its value estimates.

To improve the accuracy of our evaluation function, we could incorporate attack and defend maps. For every coordinate on the chess board, these maps encode the lowest-valued attacker (LVA) and highest-valued defender (HVD) with respect to the current player. High LVA values indicate the opponents reluctance to sacrifice his high-valued piece in an attack, and are thus better for the current player. Likewise, the current player should prefer not to sacrifice his highest-valued defenders. These maps are computationally expensive to produce, but could improve evaluation accuracy.

Several variants of TD-learning might offer faster, more accurate weight convergence. One such algorithm is TDLeaf(λ) [Baxter et al. 2000]. While TD(λ) only updates the value of the root node of a search, TDLeaf(λ) updates the values of all states on the principal variation. Another TD-learning algorithm that could offer improvement is TreeStrap(minimax) [Veness et al. 2009], which is similar to TDLeaf(λ), but performs updates on the principal variation nodes within one timestep when performing backups, instead of across timesteps.

The minimax search itself might be made faster by replacing deterministic alpha-beta cutoffs with probabilistic cutoffs [Knuth and Moore 1976]. Probabilistic cutoffs allow selectively searching more promising areas of the search tree to greater depth, while limiting search depth in less optimal parts of the search tree.

REFERENCES

Thomas Ahle. 2016. Sunfish. (aug 2016). https://github.com/thomasahle/sunfish

Jonathan Baxter, Andrew Tridgell, and Lex Weaver. 2000. Learning to play chess using temporal differences. *Machine Learning* 40, 3 (2000), 243–263.

Erik Berhardsson. 2014. Deep learning for... chess. (nov 2014). https://erikbern.com/2014/11/29/deep-learning-for-chess/

Kevin Coplan. 1982. A Special-Purpose Machine for an Improved Search Algorithm for Deep Chess Combinations. In *Advances in Computer Chess: 3*, M. R. B. Clarke (Ed.).

Dann Corbit and Swaminathan. 2010. Strategic Test Suites. (jun 2010). https://sites.google.com/site/strategictestsuite/

Omid David-Tabibi, Moshe Koppel, and Nathan S. Netanyahu. 2008. Genetic Algorithms for Mentor-assisted Evaluation Function Optimization. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO '08)*. ACM, New York, NY, USA, 1469–1476. DOI:http://dx.doi.org/10.1145/1389095.1389382

Niklas Fiekas. 2016. Python-chess. (nov 2016). https://github.com/niklasf/python-chess

Jim Fleming. 2016. Before AlphaGo there was TD-Gammon. (apr 2016). https://medium.com/jim-fleming/before-alphago-there-was-td-gammon-13deff866197

Barbara J. Huberman. 1968. *A program to play chess end games*. Ph.D. Dissertation. Stanford University.

Donald E Knuth and Ronald W Moore. 1976. An analysis of alpha-beta pruning. *Artificial intelligence* 6, 4 (1976), 293–326.

Richard E. Korf. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27, 1 (1985), 97–109.

Matthew Lai. 2015. *Giraffe: Using Deep Reinforcement Learning to Play Chess*. Master's thesis. Imperial College London, https://arxiv.org/abs/1509.01549.

Bruce Moreland. 2002. Zobrist Keys: A means of enabling position comparison. (nov 2002). https://web.archive.org/web/20070822204038/http://www.seanet.com/~brucemo/topics/zobrist.htm

Aske Plaat. 1995. Best-First Fixed-Depth Minimax Algorithms. (dec 1995). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.8838

Aske Plaat. 1997. MTD(f): A Minimax Algorithm faster than NegaScout. (dec 1997). https://people.csail.mit.edu/plaat/mtdf.html

Marco Costalba Romstad, Tord and Joona Kiiski. 2016. Stockfish. (nov 2016). https://stockfishchess.org/

Claude E. Shannon. 1950. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 314 (1950), 256–275.

Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning: An introduction* (first ed.). Vol. 1. Cambridge: MIT Press.

Joel Veness, David Silver, Alan Blair, and William Uther. 2009. Bootstrapping from game tree search. In *Advances in neural information processing systems*. 1937–1945.

Jean-Christophe Weill. 1991. Experiments With The NegaC* Search - An Alternative for Othello Endgame Search. (1991). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.3189

Chess Programming Wiki. 2016. Piece Square Tables. (nov 2016). https://chessprogramming.wikispaces.com/Piece-Square+Tables