# Augmenting Self-Learning In Chess Through Expert Imitation

**Michael Xie**
Department of Computer Science
Stanford University
Stanford, CA 94305
xie@cs.stanford.edu

**Gene Lewis**
Department of Computer Science
Stanford University
Stanford, CA 94305
glewis17@cs.stanford.edu

## Abstract

Strong chess engines are generally based on a depth-limited lookahead search process and an evaluation function at the leaves of the search tree to determine which move to make. These evaluation functions have predominantly relied upon extensive human engineering and heuristics to achieve good performance. A recent result using a neural network evaluation function with a minimal amount of human engineering and trained via purely unsupervised self-play was able to achieve performance on a level similar to that of a FIDE Grandmaster [1], [2]. In this paper, we present a hybrid supervised-unsupervised approach to learning a neural network state evaluation function for chess, in which we first conduct supervised pre-training of the evaluation function using human expert data and then improve upon this using unsupervised self-play. Intuitively, the supervised step provides guided initialization of the evaluation function by imitating expert tactics, and the unsupervised step improves upon the initialization through self-play, allowing for discovery of novel tactics and exploitation of expert tactics.

## 1 Introduction

Chess is a well-studied problem and the one of the standard problems for game playing computer agents, due to its large but manageable state space. While neural network based end-to-end systems that learn feature representations from data have recently revolutionized fields such as computer vision [3], chess and game-playing agents in general still rely largely upon extensive human engineering. Being able to learn feature representations from data is valuable to find important and novel features and is an idea that can generalize to new problems. Thus we aim to learn a neural network evaluation function that facilitates the decision making process.

However, most advances in deep learning systems have stemmed from a deluge of new data for supervised learning. In the case of computer vision, it is natural to think about allowing a small child to see many examples of objects and teach them the labels as they see them. In game playing, it is more natural to learn through experience playing the game, past the initial step of learning the rules. Furthermore, it is also natural to envision learning through imitation of an expert or teacher and then later developing original tactics through experience. This motivates learning through imitation followed by self-play. In self-play, the game-playing agent plays against itself and incrementally improves by exploiting the tactics that the agent has learned thus far. For our self-play step, we use TD-Leaf, a temporal difference learning algorithm for game trees.

In this paper, we present a hybrid supervised-unsupervised approach to learning a neural network state evaluation function for chess, in which we first conduct supervised pre-training of the evaluation function using human expert data and then improve upon this using unsupervised self-play. The supervised step can be viewed as a good initialization based on imitating the tactics of experts, while the unsupervised step allows the agent to discover new tactics and exploit expert tactics. We evaluate and compare the performance of an agent that is trained using various methods in the supervised step, followed by unsupervised self-play using TD-Leaf learning.

## 2 Background and Related Work

Most chess engines rely upon a search through the game from the current board state, using computational power to search as exhaustively as possible within time constraints and improving efficiency by pruning the tree wherever possible. At the leaves of this search tree are evaluation functions that score the board states at each particular leaf, and the action that leads to the leaf with the largest score is taken. Most strong chess engines, including the current state-of-the-art Stockfish chess engine, use complex, hand-engineered evaluation functions that give bonuses and penalties to hard-coded situations such as

having both rooks or the structure of the pawn placement, as well as polynomial regression for complex interactions between pieces [4]. We aim to learn an evaluation function using limited domain knowledge, and thus we use a neural network evaluation function.

Temporal difference(TD) learning is a reinforcement learning method of training game-playing agents in an unsupervised manner through exploration and exploitation of the game's state space. TD learning is often paired with a neural network evaluation function. A classic example of TD learning applied to games is in backgammon, where the state-of-the-art TD-Gammon algorithm learns a neural network evaluation function through self-play [5]. In recent work, the Giraffe chess engine, which uses a neural network evaluation function with minimal human engineering and is trained via TD-Leaf, a variant of TD learning for game trees, was able to achieve performance on a FIDE Grandmaster level [1], [2].

The work in this project builds upon the recent work on the open source Giraffe chess engine [1]. In particular, we use the chess engine implementation, feature representation, and adapt the TD-Leaf algorithm.

### 2.1 Features

The features used in our model are curated from the literature; following Lai [1], we convert raw chess board representations into 363-dimensional vectors that attempt to smooth the representation space by placing features that lead to similar outcomes close together.

Our feature vector utilizes a tripartite representation, where the state of the game is encoded in three different modalities: global-centric, piece-centric, and square-centric.

Global-centric features are those that are generic to the state of the game as a whole; example features include which side to move, presence of castling rights, how many of each piece is present on the board, etc.

Piece-centric features encode specific details about each game piece on the board. These features are represented using a slot system by which all of the relevant information for a particular chess piece is encoded at a positionally-invariant location in the feature vector. These features include the presence or absence of each piece, location of each piece on the board, lowest valued attacker and defender of each piece, current mobility of each piece, etc.

Square-centric features encode positional awareness and strategy in an effort to help the model learn concepts of regional control. These features are primarily encoded attack and defend maps; though these features can be learned from other sources, providing these maps as an explicit part of the feature representation helps prompt the network to learn high-level control strategy.

Though a neural network could potentially learn these features from the raw chess board data, this representation doesn't always lend itself easily to the goal of data disentanglement; indeed, experiments with TD-Gammon have shown that extracting a hand-engineered feature vector from raw game data instead of passing raw game data itself can lead to very large increases in evaluation performance [5]. Thus, the goal of using our hand-crafted feature representation is to encapsulate enough chess knowledge to relieve the burden of learning basic chess features while still allowing enough freedom for the model to pick up on diverse game dynamics.

### 2.2 Network Architecture

Artificial Neural Networks (ANNs) are a class of nonlinear models that have been successfully utilized in the control and reinforcement learning literature to learn optimal policies by mapping states to values [6, 5, 1].

Our neural evaluation model utilizes two fully connected layers and an output layer. Each fully connected layer utilizes the ReLU activation function [7] to avoid saturation of gradients. The first, second, and output layers have weights $w_1 \in \mathbb{R}^{363\times37}$, $w_2 \in \mathbb{R}^{37\times64}$, $w_o \in \mathbb{R}^{64\times1}$ respectively. For the sake of experimental consistency, we have the output layer constrain the output scalar to lie between $[-1, 1]$ by passing it through a $tanh$ activation.

Our neural evaluation model reflects the multi-modal structure of our feature representation by delaying the mixing of variables from different modalities until further along in the model. This delay is achieved by creating three masking layers that we insert between the feature vector and the first-layer weight. These masking layers consist of 1's where we have features from the desired modality, and 0's where we have features from other modalities; when we then calculate the full-connection, we achieve the desired separation.

### 2.3 TD-Leaf

TD-Leaf is a variant of temporal-difference reinforcement learning which aims to make the evaluation function predict the value of the evaluation function at a later time step during self-play. TD-Leaf generates its error signal through the objective of achieving temporal consistency. Temporal consistency is especially desirable for games where the reward is concentrated at the end state: for example, if we can accurately predict the value of the state in the next time step, then we should be

able to accurately predict the winner in the time step before the checkmate. An optimally accurate predictor with temporal consistency, therefore, should be able to predict whether the player is on a winning path. Assuming that chess is a zero-sum game, we can consider the TD error for a state with respect to the white side as the negation of the TD error of this state with respect to the black side in our calculations.

Let $s_t$ be the state at time $t$ and $f(s)$ be the evaluation function. In TD-Leaf, we take length $N$ paths from many given starting points $s_0$ and at time $t$, we calculate the TD error

$$\delta_t = f(s_{t+1}) - f(s_t),$$

which is the difference between the value of the evaluation function at the states in time $t$ and $t+1$. This is used to update the evaluation function at $s_t$. In the case of a neural network, the TD error scales the gradient during backpropagation, so that large TD errors cause a larger gradient descent update. As in the implementation of the Giraffe chess engine, we use TD-Leaf($\lambda$), which is a version of TD-Leaf with eligibility traces. This propagates TD errors through the entire state trajectory taken during self-play, decayed exponentially through time by $\lambda$. This is in contrast to TD-Leaf, which only updates the time step before. The gradient descent update rule for TD-Leaf($\lambda$) with a neural network evaluation function is

$$\theta = \theta + \alpha \sum_{t=1}^{N-1} \left( \nabla f(s_t) \sum_{j=t}^{N-1} \lambda^{j-t} \delta_t \right)$$

where $\theta$ be the parameters of the evaluation function, $\alpha$ is the learning rate, and $\nabla f(s_t)$ is the gradient of the neural network at time $t$.

## 3 Experiments

The expert training data is a set of 334699 chess games played by Grandmaster level human chess players [8]. The data is a sequence of board states annotated with which side is making a move from that state. The expert data can be seen as sample trajectories corresponding to expert policies. In the supervised step, we aim to learn from this data to provide a good initialization for unsupervised self-play. Additionally, the states in the expert data are randomly sampled as starting positions for TD-Leaf($\lambda$) in the unsupervised self-play step.

For consistency among results, we follow Lai [1] and utilize the Strategic Test Suite (STS) [9] as our evaluation metric. The Strategic Test Suite is a set of 1500 chess board configurations designed to exercise immediate short-term strategy, split between 15 different concepts ranging from tests of regional control to optimal trade of pieces. Each board position in STS comes with a list of moves accompanied by point values; the optimal move to make in a situation gives 10 points, with up to 3 listed (sub-optimal) moves earning fewer, and non-listed moves earning 0; there are 15000 points to be earned in total. The STS dataset is an independent test set that was not seen during training time.

In our experimentation, we examine the results of four different hybrid training strategies: a model that is bootstrapped from a static evaluator, a model pretrained on expert data using supervised state-scoring, a model pretrained on expert data using TD-Learning, and a model both bootstrapped and pretrained in a dual-phase manner on both a static evaluator and expert data. All models were trained on self-play using unsupervised TD-Learning after the pretraining phase, leading to a comparison of supervised-unsupervised approaches that differ in their supervised pretraining step, where the largest amount of prior knowledge can be encoded in the supervision algorithm.

### 3.1 Static Evaluation Bootstrapping

In the original implementation of the state evaluation model from Lai [1], the model is bootstrapped using a supervised training scheme where the evaluation model attempts to match the output of a hand-coded static evaluation function. This function incorporates some domain knowledge, including information about which pieces can be captured, a weak estimation of the value of each piece, if a promotion is possible, etc. Very broadly, the score is positive if we can capture more higher valued pieces in a given board state than the enemy can capture of ours. It should be noted that this scheme does not train the state evaluation model on expert data directly. This model is our basis for comparison.

When we bootstrap with the static evaluator, the model is placed in a space much closer to a local optimum than if we start from random initialization; in terms of the STS score (see section 3.5), our model starts from around 6000/15000.

### 3.2 State Scoring

Taking inspiration from the Static Evaluation model, we wish to pretrain the state evaluation model in a similar manner but directly on expert chess data instead of deriving the expert knowledge from a hand-engineered static evaluation function. We label each state in our expert dataset with either a 1, -1, or 0 corresponding to if the state was part of a win, loss, or draw by the side currently making a turn. Intuitively, given a state, we aim to predict whether this state was in a winning trajectory. Then,

given a state, our state evaluation function predicts a score $p \in [-1, 1]$; our loss and gradient for a particular state evaluation function $f$ is then given by

$$L_f(s, y) = \frac{1}{2}(f(s) - y)^2$$
$$\nabla L_f(s, y) = (f(s) - y)\nabla f(s)$$

where our gradient error signal is incorporated back into our predictor $f$ via the standard backpropagation algorithm.

### 3.3 Supervised TD-Leaf

In supervised TD-Leaf, we treat the expert data state trajectories as trajectories of self-play and aim to fit the evaluation function so that it achieves temporal consistency in the expert data. Intuitively, we aim to learn a evaluation function that can explain the expert state trajectory. The formulation is the same as that of TD-Leaf($\lambda$), except that moves are not sampled but taken directly from data.

A drawback of this procedure is that since the evaluation function is initialized randomly in the beginning of the supervised step, the TD errors that are returned from the expert state trajectories are similar to random noise. It is then possible for the evaluation function to be taking gradient descent steps in random directions, which may degrade learning.

### 3.4 Bootstrapped TD-Leaf

To alleviate the drawback of Supervised TD-Leaf, we combine the static evaluation process described in Section 3.1 and the supervised TD-Learning process described in Section 3.3 in a model called Bootstrapped TD-Leaf, with the hypothesis that the TD error signals from expert data state trajectories are meaningful after the bootstrapping procedure. If the state evaluation process can learn some of the low-level dynamics of chess from the bootstrapping process first, then the model can derive higher-level strategies and techniques from the expert data. When the model then performs self-play, it will have a broader set of strategies from which it can sample moves, leading to more innovative processes and more fruitful self-play.

During training, we lower the learning rate to .0001, since we are intuitively fine-tuning from the static evaluation model.

### 3.5 Results

For the unsupervised TD-Leaf($\lambda$) step, we used $\alpha = 1$ as the learning rate, $\lambda = 0.7$ as the decay parameter, and $N = 12$ length trajectories from starting positions, as in the original Giraffe implementation [1]. Each supervised model is run for 2000 iterations, with the exception of supervised TD-Leaf, which is run for 9000 iterations.

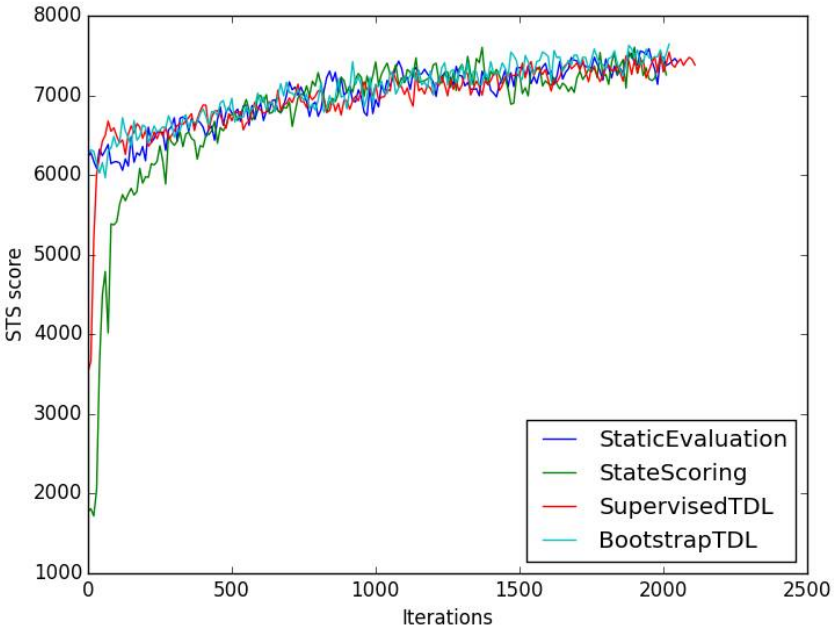


Figure 1: STS score per iteration of unsupervised TD-Leaf($\lambda$) for the 3 supervised expert data based models in comparison with static evaluation as pre-training.

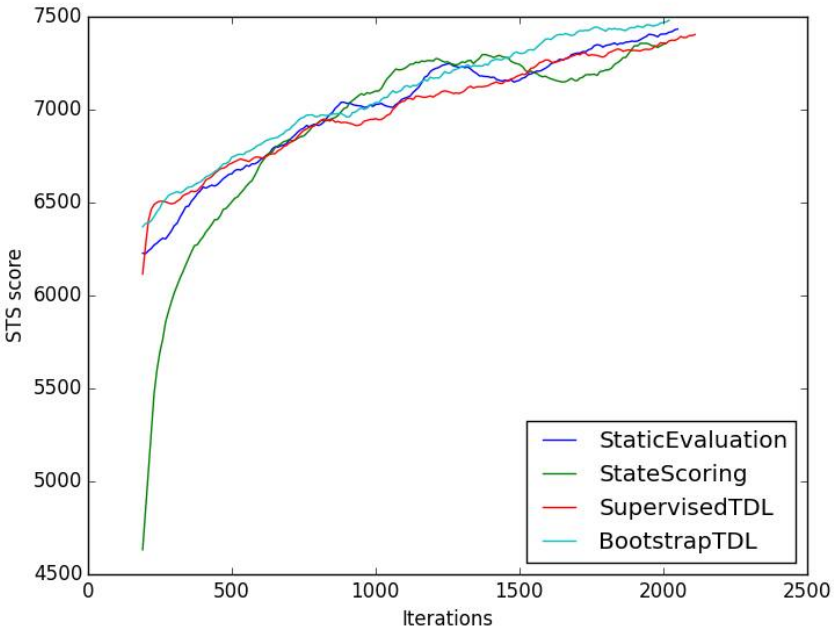


Figure 2: Smoothed version of the plot above. Here, we can see that the Bootstrapped TD-Leaf($\lambda$) generally outperformed the other models, including static evaluation.

The unsupervised self-play step is run for 2000 iterations for each model. Due to time constraints, we stop the training before convergence and analyze the results in an intermediate state. We find that the state scoring supervised step results in the lowest scoring initialization for self-play at around 1700. Supervised TD-Leaf results in a better initialization, with STS score around 3500. However, both of these are lower scoring initializations than the static evaluation method. Affirming our hypothesis, we find that Bootstrapped TD-Leaf, which uses a supervised TD-Leaf step on top of static evaluation, results in a better scoring model throughout most of the training process.

| Model | Initial STS Score | Avg STS Score | Avg # Optimal Moves | Avg # Scoring Moves |
|---|---|---|---|---|
| Static Evaluation | 6232 | 7278 | 535 | 939 |
| State Scoring | 1775 | 7258 | 532 | 932 |
| Supervised TD-Leaf | 3529 | 7242 | 532 | 933 |
| Bootstrapped TD-Leaf | 6257 | 7343 | 537 | 951 |

Table 1: Models, Avg STS Scores, and Avg Move Statistics after 1000 iterations

Numerical results for each of our models are presented in Table 1. The Bootstrapped TD-Leaf model displays superior performance to the other models, out-scoring the Static Evaluation model by around 65 points. We note that though the Bootstrapped TD-Leaf model and the Static Evaluation model have a similar number of optimal moves, the Bootstrapped TD-Leaf model significantly outperforms in the average number of scoring moves, which suggests that it is able to make better decisions in more diverse situations. This supports our hypothesis that running TD-Leaf pretraining on expert data enables our model to pick up on higher-level strategies in variable board states than with self-play alone.

Similarly, we note the vast divide in performance between the Bootstrapped TD-Leaf model and the non-bootstrapped models; both bootstrapped models strongly outperform the non-bootstrapped models, supporting our hypothesis that the Static Bootstrapping process provides valuable learning about the low-level game dynamics that are a prerequisite to useful learning from expert data. This suggests that a rudimentary supervised bootstrapping process could be an important step in the deep reinforcement learning pipeline; indeed, this step has precedence in the deep supervised learning literature, where a similar dual procedure involving an unsupervised feature-extraction step has been shown to boost the results of supervised learning in deep models [10].

## 4 Further Work

Behavioral cloning is a logical next supervised initialization to experiment with. In behavioral cloning, the evaluation function tries to predict the next action made by the expert. While this is a classification problem over the space of possible moves, we can use the layers before the last classifier layer as initialization for unsupervised self-play.

An interesting tradeoff exists between the extremes of a chess engine based on behavioral cloning(move classification with no search) and one exhaustively searching the game tree. In exhaustive search, the chess engine requires a large amount of computation to play the game to the finish in every turn. However, since the engine plays to the finish, no evaluation function is needed to approximate the value of states. In behavioral cloning, the chess engine does not use computational power to search the game tree. However, the evaluation function would need to be very powerful to accurately approximate searching the game tree. For a neural network evaluation function, this represents a tradeoff between the size of the neural network and the depth of the search. In our experiments, using 62 units in the hidden layer of the evaluation function, in contrast to 37 units in the original paper [1], resulted in faster training and better performance, even when doing unsupervised self-play from random initialization and no bootstrapping or supervised step; this achieves a STS score of almost 8000 in 2000 iterations. This implies that the neural network defined in the original Giraffe implementation may require a search of more than 12 steps to be fully effective, or that a larger neural network is needed if 12 steps of search is used. Note that the original Giraffe engine implementation takes into account time limits enforced on chess players, and larger networks with the same amount of search results in a much longer computation time. The tradeoff between size and depth of the neural network and the amount of search necessary should be further explored.

## 5 Conclusion

In some ways, we find that hand-engineered features are a very efficient way to encode domain knowledge and are hard to replace with expert data. However, we would still like to incorporate the information from data; from our experimentation, we find that it is possible to combine expert data with the hand-engineered features to improve performance. An effective way of combining expert data with domain knowledge can potentially have large improvements on the result of learning to play games through self-play.

## References


[1] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *CoRR*, abs/1509.01549, 2015.

[2] Fide rating list. `http://ratings.fide.com/download.phtml`.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. pages 1097–1105, 2012.

[4] Stockfish - open source chess engine. `http://stockfishchess.org`.

[5] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[8] Pgn mentor. `http://www.pgnmentor.com/files.html`.

[9] Strategic test suite. `https://sites.google.com/site/strategictestsuite/about-1`.

[10] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, Tao Wang, D.J. Wu, and A.Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 440–445, Sept 2011.