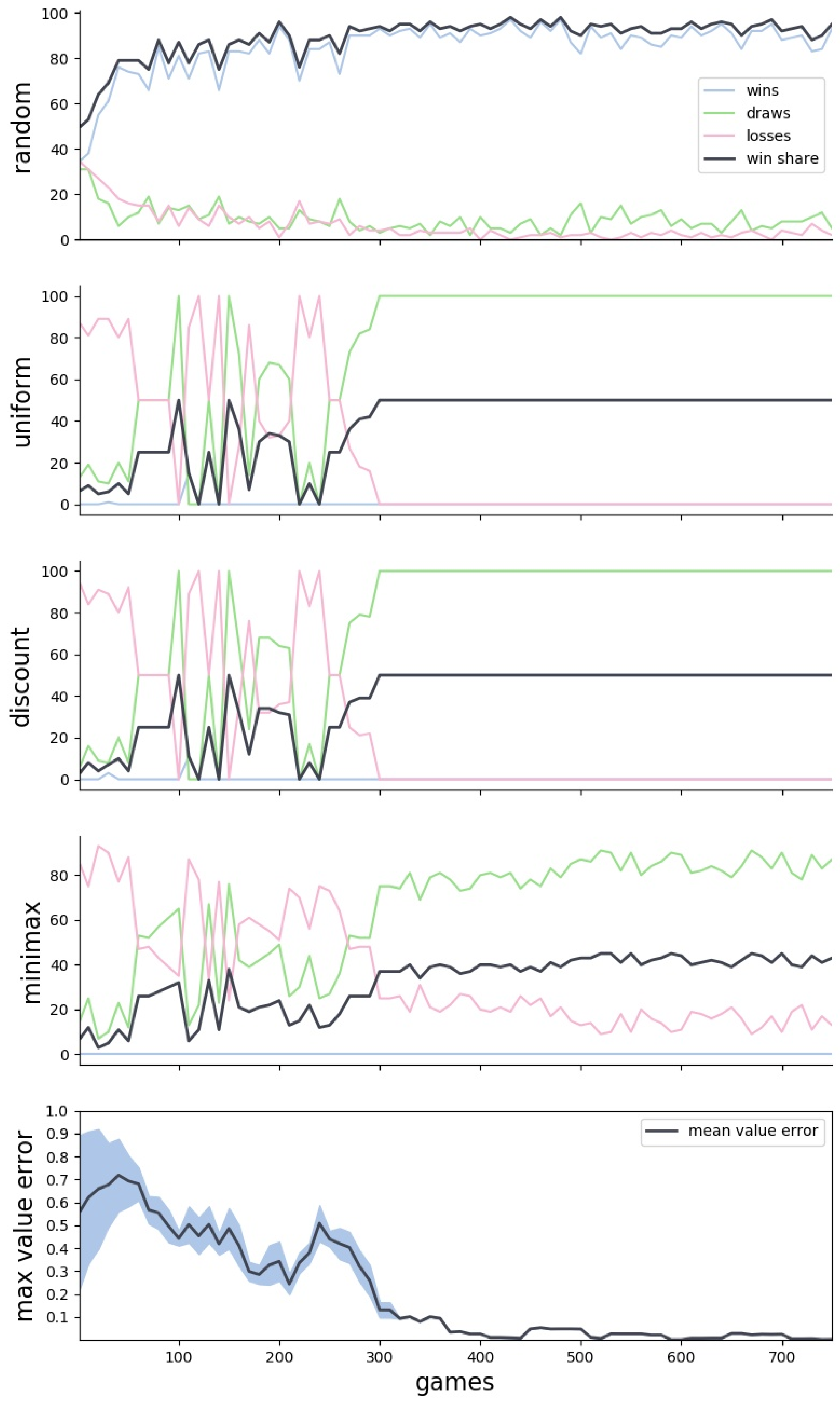
Dynammic Programming vs Self-Play Reinforcement Learning

TicTacToe Learning Curves

MonteCarlo

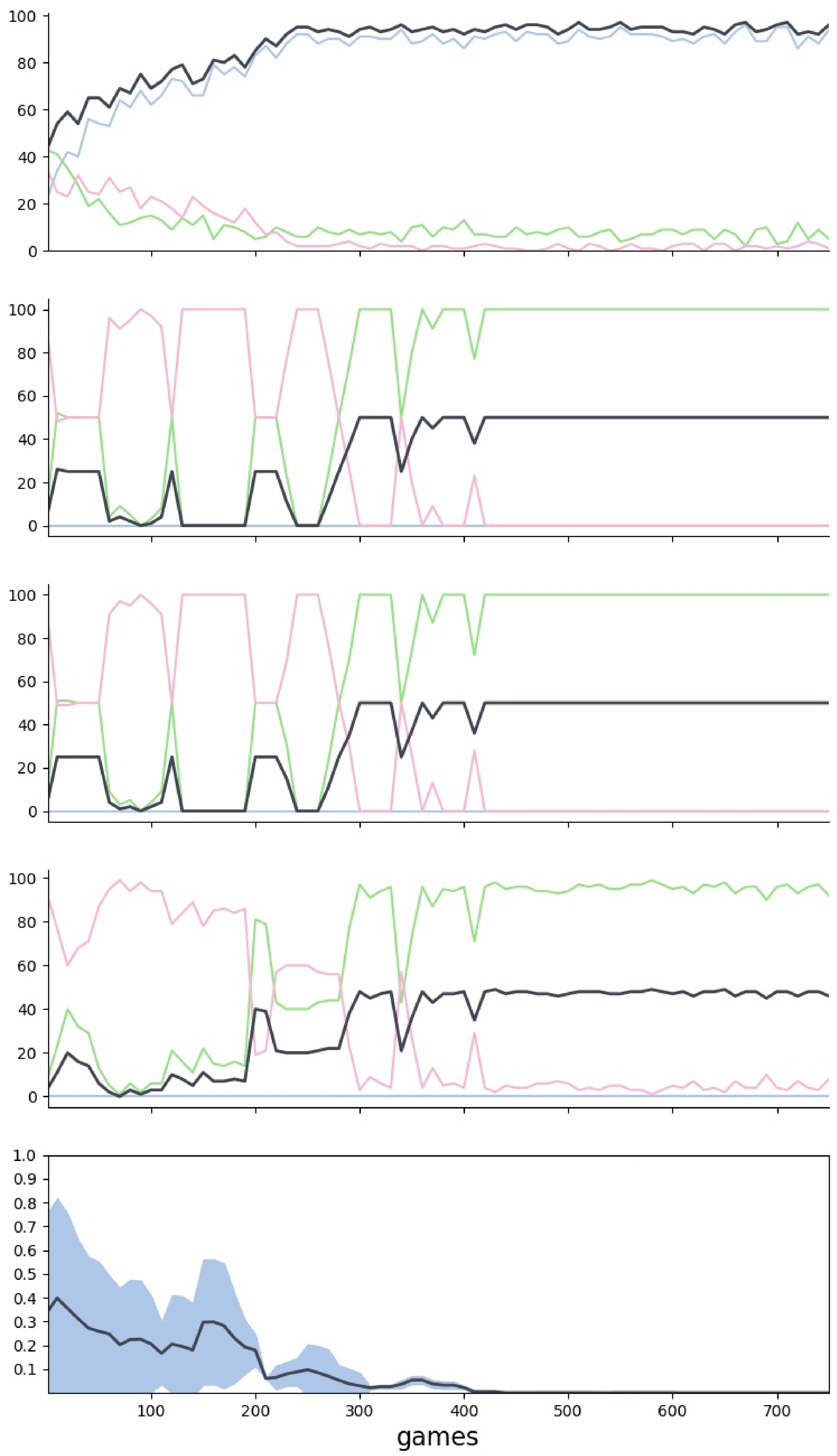
Parameters: gamma=1, alpha=0.5, epsilon=5



Converged with final win share [92 50 50 42]

TD Lambda

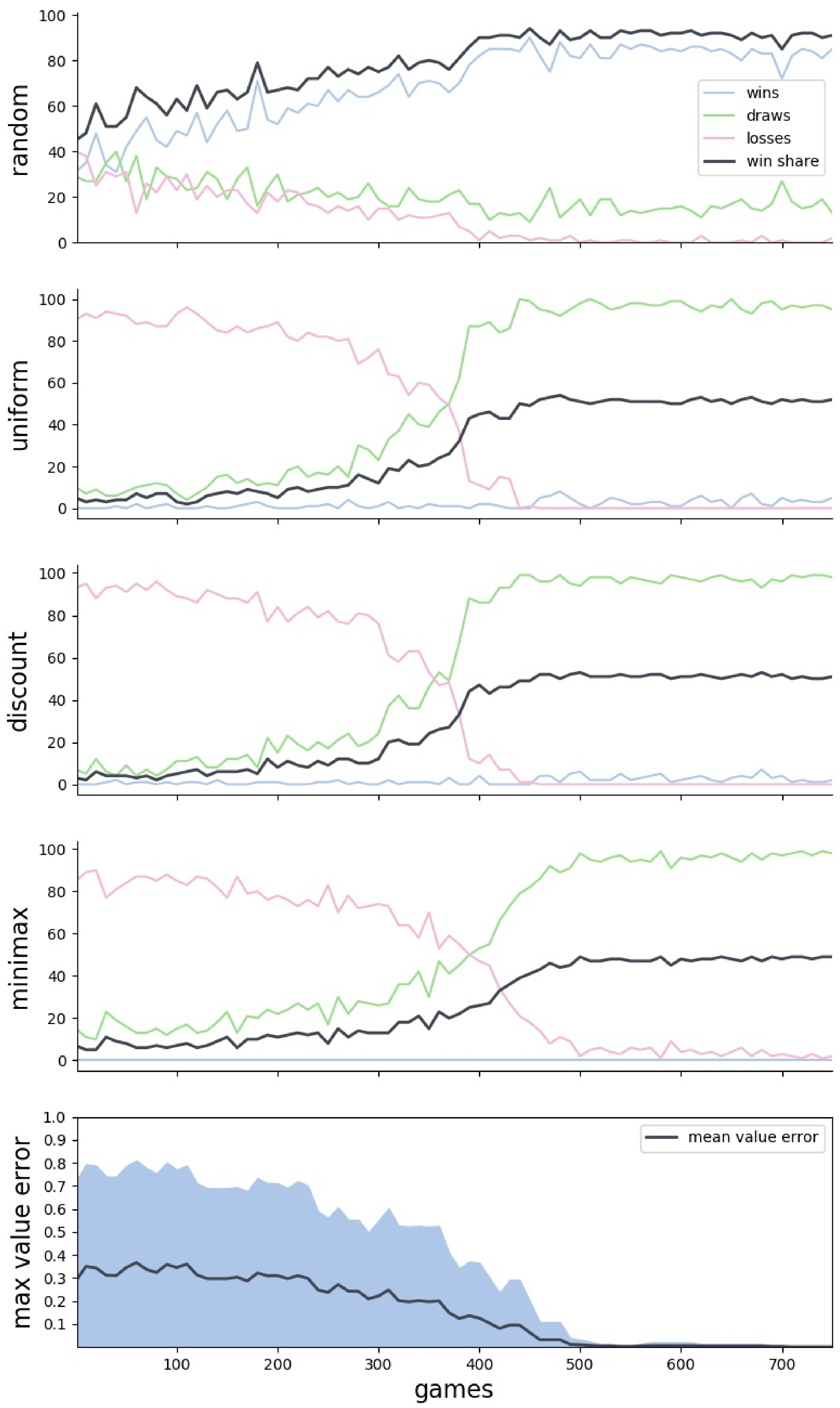
Parameters: gamma=1, alpha=0.1, epsilon=1, lambda=0.0



Converged with final win share [94 50 50 47]

Q Search

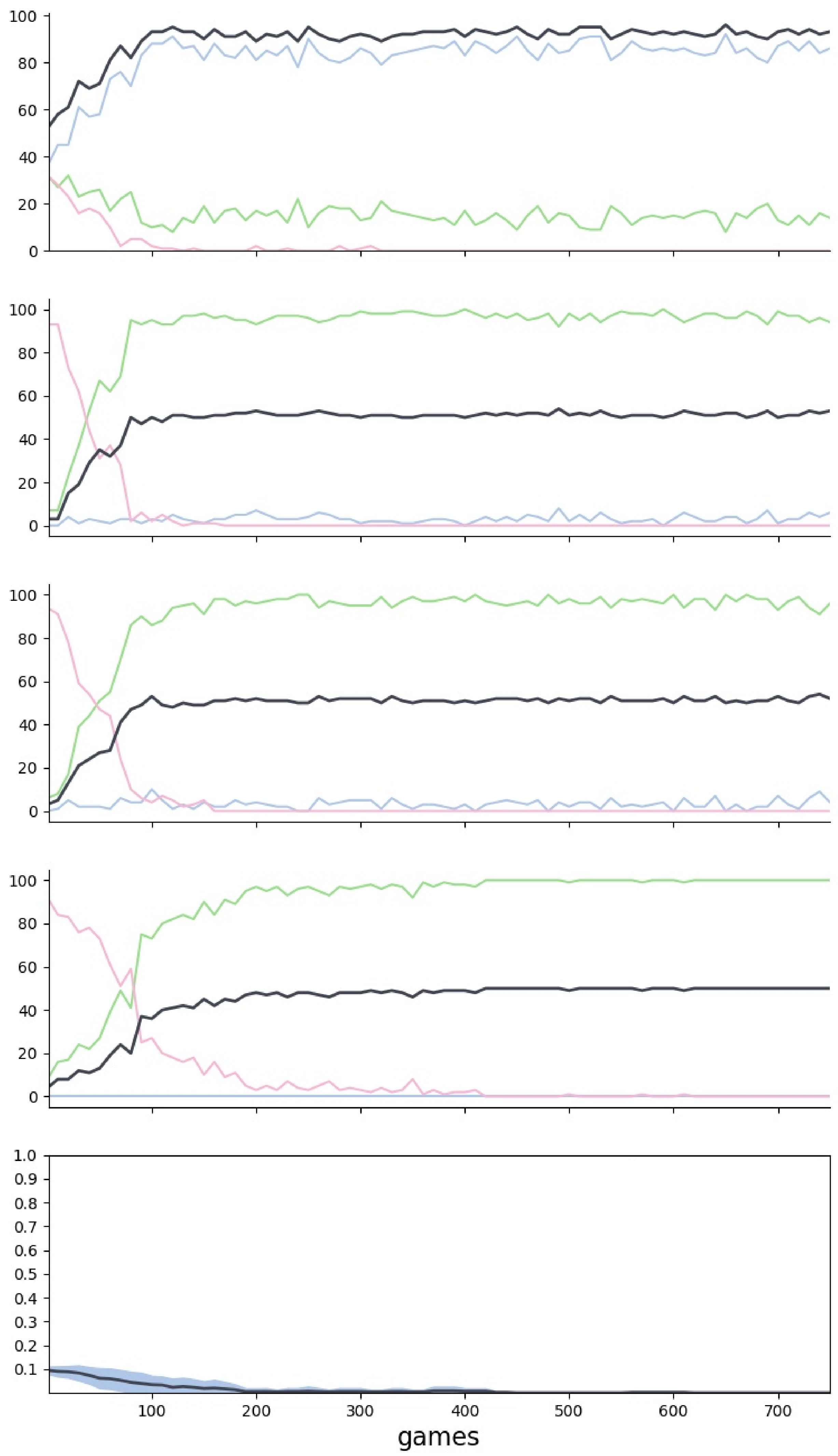
Parameters: gamma=1, alpha=0.1, epsilon=1, depth=2



Converged with final win share [90 52 52 50]

TreeStrap

Parameters: gamma=1, alpha=0.1, epsilon=1, depth=2



Converged with final win share [96 51 51 50]

TicTacToe or Noughts & Crosses is a 2-player deterministic game played on a 3x3 board. On each turn a player chooses an empty cell. A player wins if they select three in a row (or column, diagonal). The game ends in a draw if all cells are filled without a winner.

The game can be viewed as a Markov Decision Process where each board is a state and each player can take an action to advance to an afterstate. As a graph, afterstates are children. The root node is the empty board, and the leaves are the terminal boards (win, lose, or draw). An upper bound for the number of states is 3^9 = 19,683. The actual number of states is 5478. Accounting for symmetry and transpositions, we can narrow it down 765 states.

To develop different policies, we create a table mapping each state to a value. Terminal states are valued at (1, -1, 0) for (agent1 win, agent2 win, draw). During game play, the agent will choose an action with the best afterstate. So agent1 finds the maximal child value, agent2 the minimal.

To find an optimal policy we take two approaches. Dynammic Programming (DP) recursively generates the complete game tree, then backs up the values from the leaves to the root. The Uniform DP agent assumes actions are chosen randomly. So parent values equal the average of its children. The Discount agent is similar, except values decay for each ply of the backup. The Minimax agent assumes the opponent plays optimally as well. Therefore, each parent value will equal the optimal child value.

The second approach is Reinforcement Learning (RL). These agents learn iteratively through experience. After each game, the agent is given a reward based on the outcome. Each state along the game path is updated toward the outcome by gradient descent. The self-play variant has one agent playing both sides.

MonteCarlo RL updates toward the actual reward. Temporal DIfference (TD 0) uses estimated rewards stored in the table. Q-Search is similar to Q-Learning by updating toward the best estimated reward. This is the result of a minimax search up to a certain depth. TreeStrap is similar to Q-Search, except the estimated reward is backed up to all interior nodes of the Minimax tree search.

The Minimax agent is a perfect player. It never loses. Each RL agent converges to the optimal policy eventually. A sample of the first 750 games is shown. Note that MonteCarlo values have little variance since there are no intermediate rewards. TD-Lambda relies of estimated rewards, which compounds variance with each action. Q-Search is off-policy and has the most variance. Estimated returns are a result a multi-ply search. The variance trades off with a steady learning rate.

TreeStrap has minimal variance and outstanding learning rate. It converges weakly within 200 games. When experimenting with random values initalization, it often converges within 100 games. Where as Q-Search only updates the root of each minimax search tree toward the result of the search, TreeStrap adjusts all interior nodes as well, thus fully utilizing each search. Self-play RL shows promise for larger games as well. With feature extraction, function approximation, and a deeper Alpha-Beta search, TreeStrap can be extended to more complex games.