Training Connect 4 Agents Against an Optimal Agent as an Alternative to Self-Play

Darrel Ma, Sophia Pietsch, Ethan Zhang
{d32ma , stpietsc, y2788zha}@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Abstract

We explore an alternative to the use of self-play in reinforcement learning for the game of Connect 4 by training against an optimal agent. Our work was motivated by previous researchers demonstrating the feasibility of Temporal Difference Learning (TDL) when applied to Connect 4. We evaluate models trained through self-play and models trained against an optimal agent by having them play against said optimal agent and recording their winrate. To optimize our self-play models, we also experimented with various combinations of hyperparameters.

We found that while models trained against the optimal agent had a higher winrate in the first one million training games, they took many more hours to train. We suspect that this is due to the optimal agent having a much higher time complexity than the TDL agent when making a move. Additionally, our optimal values of the hyperparameters for self-play differ from those found by previous researchers. We conclude that the long training times required to train against the optimal agent prevent it from being a viable option for Connect 4 TDL agents.

Introduction

The goal of our project is to develop an agent using machine learning capable of playing the game Connect 4 optimally. In Connect 4, 2 players alternate placing pieces of their colour into a rectangular grid 7 spaces wide and 6 spaces high. Any pieces placed in a column land on top of pieces previously placed in said column, forming a stack. A player wins when they form a horizontal, vertical, or diagonal sequence with 4 pieces of their colour.

As a solved game, it is known what the optimal move is for a player at any given board state. Already, there exist models that play Connect 4 'perfectly', such that they always win if they make the first move. Our goal is to come as close as possible to the optimal strategy, using a machine learning approach.

While there are no obvious immediate benefits to developing board game playing AIs, their success serves as a proof-of-concept for the feasibility of certain machine learning techniques. Since all the information is available to the players from the beginning, board games serve as a natural starting point for artificial intelligence research. For example, AlphaZero, the newest program from AI company DeepMind, demonstrates the capabilities of a general-purpose algorithm that has allowed it to master chess, go, and shogi (also known as "Japanese chess") (Stetka 2018). In addition, unlike previous versions of AlphaGo (also developed by DeepMind) that used a mix of brute force, tree search, and successful moves made by human players, AlphaZero used only a neural network to guide its play (Purves 2019). With a 61% winrate against AlphaGo in the game go, AlphaZero highlights neural networks as a promising area for future AI research (Stetka 2018).

For our paper, we compare two different methods of training an agent to play Connect 4, playing against an optimal agent and self-play. For both training methods, we use Temporal Difference Learning (TDL) along with an n-tuple system, which generates features from the raw board state. TDL uses rewards from the environment to compute the value function, which is then used to estimate the value of a given board state. The agent then makes the next move based on the estimated value. To further optimize TDL, the n-tuple system is introduced into the training process. Based on a given board state, the system assigns values to pre-generated sequences of board positions. The TDL algorithm uses these values to calculate the expected outcome for the given state.

We evaluate both agents by letting them play against the optimal agent. Since Connect 4 is a solved game where the player going first can always force a win, we let our trained agents make the first move against the optimal agent. A perfectly trained agent should achieve a winrate of 100% but for this experiment, our goal is for the trained agents to achieve a high winrate (>80%). We will compare not only their winrate against the optimal agent, but also the number of training games required to achieve the target winrate.

Our experiments revealed that self-play is a better method for training TDL agents to play Connect 4. Our self-play agents reached a winrate of 80% after an average of 4.5 million training games for several different combinations of hyperparameters. While training against the optimal agent led to a higher winrate in the first one million games, the higher time complexity significantly increased the amount of time required for each game. Due to time constraints, we were unable to continue training against the optimal agent until our agents reached a winrate of 80%.

Contributions

- We are the first to train a TDL agent for Connect 4 against an optimal agent.
- We modified the implementation of the TDL algorithm in order to train against an optimal agent.
- We find a positive correlation between an increase in the learning rate α and the winrate of self-play TDL agents, up to a certain point.

Related Work

Since Connect 4 was introduced in 1974, there have been many attempts to find an optimal strategy. James Dow Allen was the first to solve the game followed closely by Victor Allis, both in 1988. Allis proposed a knowledge-based approach, which focuses on nine strategic rules for optimal play. Along with these nine strategies, Allis presented the VICTOR program, which compiles the nine strategies to determine the value of a position. The name VICTOR was chosen for the program's ability to always win given it makes the first move. (Allis 1988).

More recent research into Connect 4 has used the Minimax algorithm along with alpha beta pruning to solve the game. The Minimax algorithm is a type of backtracking algorithm that is mainly used in two player games. The algorithm finds the optimal move for the current player assuming that opponent is also playing optimally. However, it is not feasible to search through all 4 trillion possible states in Connect 4 using the Minimax algorithm. Instead, the alpha beta pruning method was introduced. Alpha beta pruning is considered an optimization of the Minimax algorithm, as it uses additional parameters to improve both search time and search efficiency (Nasa et al. 2018).

Recently, the rising popularity of reinforcement learning has led many researchers to begin applying this method on traditional board games. In particular, the TDL (Temporal Difference Learning) reinforcement learning strategy has allowed agents to play traditional board games such as checkers, chess, and backgammon at a high level. The use of TDL grew more popular after Tesauro's TD-Gammon backgammon AI was published. TD-Gammon was developed using TDL, specifically the TD(λ) algorithm, and it was able to play at a high level through self-play training (Tesauro 1995). This opened up further research into the use of TDL in training AI to play board games such as Tic-Tac-Toe, Othello, and Connect 4. In this paper, we will focus on the use of TDL for Connect 4.

Early attempts at using TDL for Connect 4 were unsuccessful. Sommerlund trained two neural networks to play Connect 4, one using the raw state of the board and another using precomputed features, such as the longest sequence of pieces crossing each board space. In both cases, the training did not improve the winrate of the neural networks against an agent using a predefined evaluation function (Sommerlund 1996). In contrast, Browne and Scott trained two agents using Q-Learning against existing agents. Q-Learning is a TDL algorithm that uses dynamic programming to calculate the value at each state using state-action pairs. One agent was trained against increasingly advanced agents, whereas the other agent was trained exclusively against the most advanced agent. The results were poor, with both agents achieving a winrate of only 11% against the most advanced agent. (Browne and Scott 2005).

Recently, TDL agents for Connect 4 have been improved by using n-tuples instead of a raw representation of the board state. This was inspired by the successful use of n-tuples for the game Othello, where agents trained using n-tuples learned much more quickly and outperformed all other agents (Lucas 2008; Krawiec and Szubert 2011). When using n-tuples, random sequences of n adjacent board spaces are generated. Given some board state, a number is calculated for each sequence based on the pieces present in the corresponding board spaces. Given these numbers for all sequences, TDL is used to estimate the value of the board state (Thill, Koch, and Konen 2012).

One such experiment was conducted by Thill et al., who trained a neural network with n-tuple inputs using self-play and TDL. Initially, their agent achieved a winrate of 90% when playing first against an optimal Minimax agent (Thill, Koch, and Konen 2012). After fine-tuning the parameters, Bagheri et al. reduced the number of training games needed to achieve this winrate from 3 million to 1.2 million games (Bagheri et al. 2014). Eventually, Thill et al. created an agent requiring only 300 thousand games to reach a winrate of 90% by increasing the number of n-tuples used and implementing the TD(λ) algorithm. While TDL updates the estimated value of a state based only on the next state visited, the TD(λ) algorithm updates the value of a state multiple times based on the next several board states visited (Thill et al. 2014).

Besides TDL, there have been other approaches taken to teach an agent to play Connect 4. Curran and O'Riordan implemented cultural learning using neural networks, letting multiple agents play against each other and using the best agents of each generation to train the next generation of agents. The agents were able to consistently outperform a random agent (Curran and O'Riordan 2004). Similarly, Browne and Scott trained agents that generated rules to determine their actions and then used a genetic algorithm to search for the best combination of rules. Their best agent had a winrate of over 50% against an existing advanced Connect 4 agent (Browne and Scott 2005). Finally, several researchers have investigated training an agent based on computer generated game trajectories. The trained agents performed well against the agents used to generate the game trajectories (Runarsson and Lucas 2015; Schneider and Rosa 2002).

Methodology

In reinforcement learning, agents learn based only on periodic rewards from the environment. The goal of many agents is to find an optimal policy based on such rewards. An optimal policy determines the best action to take in each state to maximize the reward received from the environment. To find an optimal policy, many reinforcement learning algorithms estimate a value function $V(s)$, which gives each state s a value that represents the expected future reward of being in that state. The agent can follow an optimal policy given an

optimal value function by choosing the successor with the highest value at each point. Thus, an algorithm can approximate an optimal policy by approximating the optimal value function (Sutton and Barto 2018).

Temporal Difference Learning is a reinforcement learning algorithm that allows the agent to train by learning from past experiences and attempting to find the best action at all given states. Our description of Temporal Difference Learning is based off of Sutton and Barto (Sutton and Barto 2018). Unlike tree search algorithms, TDL does not need to search through millions of possible states, and thus does not require long calculation times. This is advantageous as many board games consist of millions of states, and any tree search algorithm will not be able to find the best solution in a reasonable amount of time. TDL also outperforms the Monte Carlo method of reinforcement learning; it updates the estimated value function at each step, whereas the Monte Carlo method only updates said function at the end of each training game. This allows the TDL algorithm to adjust the value function multiple times during one training game, allowing the model to learn at a faster rate.

Temporal Difference Learning combines aspects of several other reinforcement learning algorithms, including the Monte Carlo and Dynamic Programming methods. Like Dynamic Programming, TDL updates the value of the current state using the estimated value of the best successor state, without waiting for some reward from the environment. If an agent starts in state s_t and moves to state s_{t+1} next, then TDL updates the value function using the following rule:

$$V(s_t) = V(s_t) + \alpha(r(s_{t+1}) + \gamma V(s_{t+1}) - V(s_t))$$

Here, the learning rate α controls the rate at which the value function changes. The learning rate is decreased between training games according to an exponential decay scheme. The discount rate γ is a value between 0 and 1 that controls how much the agent prioritizes immediate rewards over future rewards. When $\gamma = 1$, all rewards are treated equally. $V(s_{t+1})$ is the estimated value of the new state s_{t+1} and $r(s_{t+1})$ is the reward received from the environment for being in the state s_{t+1}. Since terminal states have no future rewards, we have $V(s_{t+1}) = 0$ if s_{t+1} is terminal. In the update rule, the expression in parentheses is called the TD error (denoted δ_t) and represents the difference between the current estimated value $V(s_t)$ and the better estimate $r(s_{t+1}) + \gamma V(s_{t+1})$:

$$\delta_t = r(s_{t+1}) + \gamma V(s_{t+1}) - V(s_t)$$

TDL chooses successor states using some fixed strategy, and we will use an ϵ-greedy strategy for our agents. When using this strategy, the algorithm will choose the best successor state with probability $1 - \epsilon$, and a random successor state with probability ϵ. By having the algorithm sometimes take a random action, more states are explored. Exploration ensures that the value function is updated for all states, so that new, better policies can be discovered. The value function is not updated after a random move unless the random move results in a win, since the move will likely not reflect the best possible move of the agent in the given state.

The ϵ-greedy strategy is a straightforward approach to the exploration-exploitation dilemma. Despite its simplicity, the ϵ-greedy strategy is often very successful. Unlike strategies that distinguish between the exploration and exploitation process, ϵ-greedy will continue to collect new data and optimize over the entire training period. Another advantage is that ϵ-greedy is simple to calculate and no extra parameters are needed besides the value of ϵ. Given the large state space of Connect 4, strategies that require storing counters or extra values will require a large space overhead during the exploration process. Compared to ϵ-first, ϵ-decreasing and other, more complex strategies, in most cases ϵ-greedy has similar or even better performance. (Vermorel and Mohri 2005)

To speed up learning, we approximate the value function using an n-tuple method. When learning a value for each state with TDL, the agent needs to visit each state many times to approximate an optimal value function. Since Connect 4 has over 4 trillion possible board states, this method would require very many training games. Instead, the n-tuple method approximates a value function by considering predetermined sequences of board positions, called n-tuples. With this approach, the algorithm can take advantage of the similarities between board states that have the same pieces in certain spaces. Thus, the agent is able to generalize to unseen states and learn more quickly. We chose this particular method of value function approximation since it has been applied to Connect 4 successfully in the past. Our description of n-tuples is based off of previous work in this area (Thill, Koch, and Konen 2012; Thill et al. 2014; Bagheri et al. 2014; Runarsson and Lucas 2015).

Each n-tuple $T_i = [s_{i0}, \ldots, s_{i(n-1)}]$ is a predetermined sequence of n board positions $s_{ij} \in \{0, 1, \ldots, 41\}$, where the board positions are labeled row by row, starting from the bottom left. For a given board state, we assign each single board position a number $N(s_{ij})$ based on what piece is in that position. In particular, we use

$$N(s_{ij}) = \begin{cases} 0 & \text{if } s_{ij} \text{ is empty and unreachable} \\ 1 & \text{if } s_{ij} \text{ contains a yellow piece} \\ 2 & \text{if } s_{ij} \text{ contains a red piece} \\ 3 & \text{if } s_{ij} \text{ is empty and reachable} \end{cases}$$

A position is treated as reachable if it is the lowest-valued empty position in its column. We treat reachable and unreachable empty positions differently since Thill et al. showed this reduced the number of training games needed and improved the agent's winrate by 6% (Thill, Koch, and Konen 2012). Since each $N(s_{ij}) \in \{0, 1, 2, 3\}$, we can associate with each possible configuration of pieces in the positions of T_i a unique index k_i as follows:

$$k_i = \sum_{j=0}^{n-1} N(s_{ij}) \cdot 4^j$$

Observe that for each state there is an equivalent state that can be obtained by reflecting the board horizontally. By taking advantage of this, we gain more information and reduce the number of unique states. Since we expect the values of these two states to be identical, for each n-tuple T_i we

consider its reflection. The configuration of pieces in the reflected tuple give a second index $\overline{k}_i$ for the same tuple T_i. For example, Figure 1 shows one possible 4-tuple and its reflection, taken from Thill et al. If the original tuple is the one on the left and the elements of the tuple are ordered starting on the bottom left, we have $k_i = 1{\cdot}4^0{+}2{\cdot}4^1{+}1{\cdot}4^2{+}1{\cdot}4^3 = 89$. Similarly, we have $\overline{k}_i = 147$ (Thill et al. 2014).

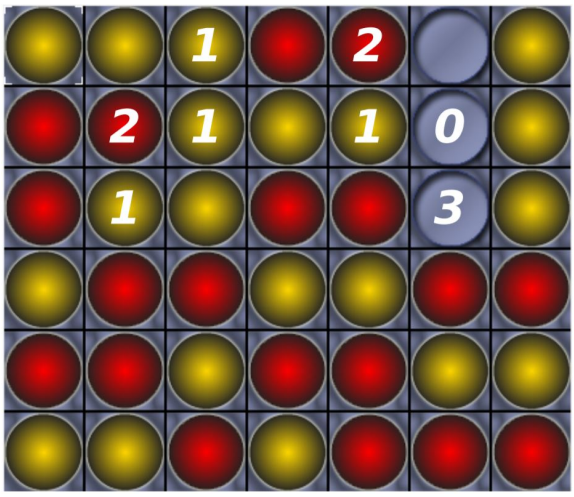


Figure 1: A 4-tuple state and its equivalent mirrored state (Thill et al. 2014)

For each tuple T_i, the indices k_i and $\overline{k}_i$ are used to retrieve weights w_{ik_i} and $w_{i\overline{k}_i}$ from a weight vector $\mathbf{w}$. w_{ik_i} and $w_{i\overline{k}_i}$ represent the estimated value of the current configuration of pieces in T_i and the reflection of T_i, respectively. All weights are initially set to zero. Given the current weight vector $\mathbf{w}_t$ for tuples T_1 to T_m, the approximate value of the current state s_t is:

$$V(s_t, \mathbf{w}_t) = \tanh\left(\sum_{i=1}^{m} w_{ik_i} + w_{i\overline{k}_i}\right)$$

The use of the tanh function ensures that the estimated values of the states are between -1 and 1. This limitation is appropriate since the value $V(s_t, \mathbf{w}_t)$ of s_t should be between -1 and 1 to accurately represent the expected future reward of that state. The input to the tanh function can be written as a dot product of the weight vector $\mathbf{w}_t$ with a vector $\mathbf{x}_t$, where

$$x_{ij} = \begin{cases} 1 & \text{if } j = k_i \text{ or } j = \overline{k}_i \\ 0 & \text{otherwise} \end{cases}$$

Instead of updating the value of each state directly, the TDL algorithm updates the weight vector. Since the weight vector only approximates the optimal value function, we cannot achieve the optimal value for all states. Instead, if $V(s)$ is the optimal value for state s and $\mathcal{S}$ denotes all possible states, we want to find the weight vector $\mathbf{w}$ that minimizes the mean squared error between the approximate value function and the optimal value function (Sutton and Barto 2018):

$$\overline{VE}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \left(V(s) - V(s, \mathbf{w})\right)^2$$

Since $V(s)$ is unknown, TDL approximates $V(s_t)$ with $r(s_{t+1}) + \gamma V(s_{t+1}, \mathbf{w}_t)$. The following update rule for the weights will converge to a minimum of $\overline{VE}(\mathbf{w})$ in most cases, given sufficiently many training games (Sutton and Barto 2018):

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \delta_t \nabla_{\mathbf{w}} V(s_t, \mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha \delta_t (1 - V(s_t, \mathbf{w}_t)^2) \mathbf{x}_t \end{aligned}$$

Here, the update to the weights depends on the learning rate α, the TD error δ_t as well as the gradient of $V(s_t, \mathbf{w}_t)$. While convergence is not guaranteed, this method is what allows the TDL algorithm to update weights continually throughout a training game when an approximate value function is used. As explained above, continually updating the value function leads to better performance earlier on compared to other reinforcement learning algorithms.

Algorithm 1 shows the pseudo-code for one iteration of TDL. Given some state s_t and the current weights $\mathbf{w}_t$, it will choose some successor s_{t+1} and calculate the new weights $\mathbf{w}_{t+1}$ as described in the previous paragraphs.

Algorithm 1: One iteration of TDL with n-tuples

Input : A state s_t and weights $\mathbf{w}_t$
Output: A state s_{t+1} and weights $\mathbf{w}_{t+1}$
calculate $V(s_t, \mathbf{w}_t)$
determine all successors $\mathcal{S}$ of s_t
generate uniformly at random $p \in [0, 1]$
if $p < \epsilon$ **then**
  randomly select $s_{t+1} \in \mathcal{S}$
  determine $V(s_{t+1}, \mathbf{w}_t)$, $\mathbf{x}_t$ and $r(s_{t+1})$
else
  calculate $V(s, \mathbf{w}_t)$ for all $s \in \mathcal{S}$
  determine $r(s)$ for all $s \in \mathcal{S}$
  $s_{t+1} = \text{argmax}\{r(s) + V(s, \mathbf{w}_t) : s \in \mathcal{S}\}$
end
if $p \geq \epsilon$ **or** $r(s_{t+1}) = 1$ **then**
  $\delta_t = r(s_{t+1}) + \gamma V(s_{t+1}, \mathbf{w}_t) - V(s_t, \mathbf{w}_t)$
  $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t (1 - V(s_t, \mathbf{w}_t)^2) \mathbf{x}_t$
else
  $\mathbf{w}_{t+1} = \mathbf{w}_t$
end

Similar to Thill et al., we will use a Minimax algorithm for the optimal agent against which we will evaluate our models. This optimal agent will have access to a pre-computed database of moves and board states. On each turn, the agent will try to find a move that allows it to force a win, or at least a draw. If no such move exists, the agent will make the move that leads to the most distant loss. This prolongs the game and makes it more likely that the opponent will make a sub-optimal move that allows the Minimax agent to force a win/draw.

If at any point there are multiple optimal moves, the Minimax agent selects one at random. Additionally, during the first 12 moves if it cannot force a win/draw it will select a move at random from those that prolong the game for at least 12 more turns (as opposed to the most distant loss). This keeps our Minimax agent from being completely deterministic, which is important for evaluation (Thill, Koch, and Konen 2012).

Results [1]

Experimental setup

We train agents using both self-play and against a Minimax agent, as we aim to compare the effectiveness of training against opponents of different skill. Since the success of training depends on the random moves made by the agent, we train multiple agents with each configuration of parameters. We initialize all weights of our agents to 0.

When training agents using self-play, we use separate weight vectors for the agent making the first move and the agent going second. Thill et al. showed that separate weights improve the winrate of an agent by 35% in comparison to using the same weights (Thill, Koch, and Konen 2012). To determine the value of a successor state, the algorithm uses the weights of the respective agent. We give the agent a reward of 1 for a win, a reward of -1 for a loss and no reward for all other states.

When training agents against the Minimax agent, the TDL algorithm makes the first move against the Minimax agent, as otherwise the Minimax agent forces a win. Here, we use only one set of weights. To determine the value of a successor state, the algorithm examines all possible states that could result from the next move of the Minimax agent. The algorithm uses its current weights to assign each such state a value and sets the value of the successor state equal to the lowest of these values. We reward the TDL agent based on the length of each training game, such that the agent can learn despite always losing initially. If there are k pieces on the board when the game ends, the agent receives a reward of $1 - \frac{k}{100}$ for winning and $-1 + \frac{k}{100}$ for losing the game. (As before, the agent receives no reward for all other states.)

For agents trained both through self-play and against the Minimax agent, we initially use the parameters of the TDL algorithm that were found to be optimal for learning Connect 4 by Bagheri et al (Bagheri et al. 2014). Since there is only one reward at the end of each game, it does not make sense to prioritize earlier rewards. Thus, we use $\gamma = 1$ for the discount rate. Furthermore, we use $\epsilon = 0.1$ throughout the whole training process. We initialize $\alpha = 0.004$ and decrease α according to an exponential decay scheme until $\alpha = 0.001$. To achieve a similar rate of decay as used by Thill et al., we set α to the following value after the nth training game:

$$\alpha(n) = 0.001 + 0.003e^{-0.000005n}$$

With these parameters, Bagheri et al. required 670 000 training games for their agent to reach a winrate of 80% against a Minimax agent (Bagheri et al. 2014). We compare agents trained using these parameters with agents trained with similar values of ϵ and α.

For the n-tuples, Thill et al. determined that using 70 8-tuples is most effective. Among all such n-tuples, some n-tuples lead to better results than others. Therefore, we use the same n-tuples that Thill et al. used, shown in Figure 2 (Thill, Koch, and Konen 2012).

```
 0  6  7 12 13 14 19 21 : 13 18 19 20 21 26 27 33
 1  3  4  6  7  8  9 10 :  7  8  9 12 15 19 25 30
 4  5  9 10 11 15 16 17 :  1  2  3  8  9 10 16 17
 3  8  9 10 11 14 15 16 :  0  1  2  6  8 12 13 18
25 26 27 32 33 37 38 39 :  3  4  8  9 11 14 15 21
 2  3  4  8  9 14 15 20 : 18 19 24 30 31 32 36 37
 3  4  8  9 10 14 15 16 :  5 10 11 16 17 21 22 27
 4 10 15 20 21 22 27 28 : 18 24 25 30 31 32 37 38
11 17 21 23 27 28 33 39 : 21 25 26 27 32 34 35 41
22 25 26 27 30 32 33 37 :  4 10 11 16 20 21 22 23
 0  6  7  8 12 13 14 15 : 17 23 28 29 32 33 34 35
 0  6  7 12 18 25 32 38 :  2  3  4  5  8  9 10 11
27 32 33 34 37 38 39 40 :  4 10 16 21 26 32 33 38
 0  6  7 12 13 20 27 28 :  0  6 12 19 25 31 32 33
 1  2  6  7 13 14 15 20 :  1  2  5  8 11 15 16 17
13 14 16 18 21 22 23 24 :  2  3  9 10 11 16 17 22
15 16 17 20 22 23 25 31 : 15 16 17 21 22 23 28 29
24 26 30 31 32 33 36 37 : 12 13 18 19 20 26 27 33
 1  2  3  8  9 13 14 21 : 13 14 18 20 24 25 31 37
14 15 16 21 26 31 38 39 :  1  2  6  7 12 13 14 20
 4  5 10 11 17 22 23 29 :  2  4  5  7  9 10 14 19
 5  9 10 11 15 16 21 27 :  1  2  3  7  8 13 14 20
 1  2  8  9 14 15 21 26 : 22 23 29 33 34 35 38 41
13 18 19 24 25 26 31 32 : 27 28 29 31 32 33 37 38
10 14 15 16 17 20 21 23 :  4  5  9 10 15 20 21 22
13 20 25 26 27 32 34 41 : 30 31 33 34 36 37 38 39
11 16 23 28 34 35 40 41 :  3  4 10 11 14 15 16 17
15 20 21 22 26 32 33 39 : 18 19 25 26 31 32 34 39
 4  9 11 15 16 22 23 29 : 26 27 31 32 33 37 38 39
20 27 28 33 34 35 40 41 :  1  2  7 14 20 27 28 29
 8  9 10 15 16 17 22 23 :  9 14 15 20 21 22 27 32
 1  2  3  6  7  8  9 13 : 10 14 15 16 20 23 25 26
 0  1  2  6  7  8 13 14 :  1  6  7 12 13 20 26 27
 8 14 20 25 26 31 33 38 : 20 21 26 27 28 33 35 40
```

Figure 2: The n-tuples used by Thill et al. (Thill et al. 2014)

Since the agents have probability ϵ of making a move at random each turn during training, they will not be playing optimally. As such, we will explicitly evaluate the models every 20 000 games, as Thill et al. did (Thill, Koch, and Konen 2012). During each round of evaluation, the TDL agent plays 100 games against an optimal agent. Since the optimal agent takes a random optimal move, we assume that our agent can beat any Connect 4 agent if it can beat the optimal agent. As the optimal agent can force a win if it goes first, we will always have our model go first and the optimal agent go second.

Note that in some cases, we use the same optimal agent for training and for evaluation, which raises the issue of potential overfitting. However, our end goal is to train a model to play Connect 4, for which the rules and board never change. The optimal agent is representative of a talented human player, and the situations faced by the agent during training accurately reflect those it would encounter in the real world. Thus, it makes sense for us to use the same conditions for evaluation and training.

For each evaluation game, we award the model 1 point for a win, 0.5 points for a draw, and 0 points for a loss. An

[1] The code is available in `https://github.com/cherrykit/Connect-Four/`

optimal agent evaluated under this system should win every game it plays and achieve a score of 100. We make note of how many training games each agent requires to achieve a score of 80. This reflects the initial 80% winrate achieved by Thill et al (Thill, Koch, and Konen 2012). We continue to train our agents until they show little improvement. When the score of an agent remains the same across 3 consecutive rounds of evaluation, we stop training.

Implementation details

We used an existing Connect 4-playing framework containing a board representation, a Minimax agent, and methods for modifying a board state and generating successor states (Thill and Konen 2014). We implemented the framework required for n-tuples, the TDL algorithm and evaluation. All our code is written in Java and was executed on an eight-core virtual machine (3.1 GHz on each of the cores).

Our application trains eight independent agents at a time, using a separate thread for each processor core. For each agent, the application starts a loop which alternates between simulating 20 000 training games and 100 evaluation games until the agent meets the stopping criteria explained above. The initial values of α and ϵ can be set through command-line arguments. Based on preliminary results, we created an alternate version of the application to explore a slower α decay scheme. For training against the Minimax agent, we created a modified version with the alternative reward system described above.

Each training game starts by passing the TDL algorithm the initial state and the partially trained weights vector from the previous training game. The output state of each TDL iteration is given to the opposing agent, which generates the next state and passes it back to the TDL agent. The weights of the TDL agents are carried over between training games. During evaluation games, the TDL agent does not update its weights and does not make random moves. After each round of evaluation, we write the current number of training games and the score achieved by the agent to a results file.

Agent performance

The comparison of the training methods was limited by the long training times we observed when training against the Minimax agent. As the Minimax agent uses tree search to determine its next move, the Minimax agent requires more time than the TDL agent to compute each move. When training against the Minimax agent, we only managed to simulate 1 million games despite training for over 40 hours. Figure 3 shows the average score of eight agents trained against the Minimax algorithm and eight using self-play. The agents trained against the Minimax agent were able to beat the Minimax agent very early in the training process, with the best agent reaching a score of 9 after only 20 000 training games. These agents quickly learn to avoid making mistakes, as they would soon lose and receive a negative reward for doing so. In comparison, the first self-play agent to reach a score of 9 required 250 000 training games to do so. Initially, the self-play agents are unlikely to receive a negative reward for every mistake, since they cannot yet take advantage of their opponents mistakes. Because of this early advantage of agents trained against the Minimax algorithm, these agents had better performance during the first 1 million training games, with an average score difference of 5-10.

We continued to train the self-play agents until they reached an average score of 80. This required approximately 4.5 million training games, or about 5 hours. The shorter training time made training through self-play a more viable strategy than training against the Minimax agent. However, our agents required significantly longer to meet the stopping criteria than the agents trained by Bagheri et al., which reached a winrate of 80% after 670 000 training games (Bagheri et al. 2014). Instead, our results are comparable with the initial results from Thill et al., whose agents

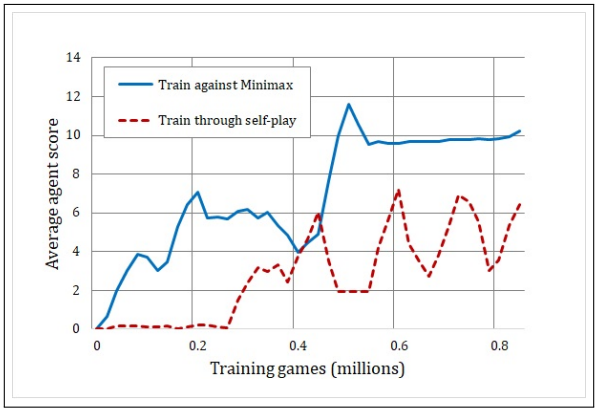


Figure 3: Performance comparison of agents training against a Minimax agent and agents trained through self-play. Comparison is limited to the first 1 million training games as use of the Minimax agent significantly slowed down the training process.

required 4 million training games to reach a winrate of 80% (Thill, Koch, and Konen 2012). We attempted to reduce the number of training games required by tuning the hyperparameters α and ϵ. We will refer only to agents trained through self-play from now on.

We first experimented with several initial α values while using $\epsilon = 0.1$ and keeping the final α value fixed at 0.001. We trained eight agents with each configuration of parameters and graphed their average scores in Figure 4(a). In general, we observed that larger α values lead to better initial performance. The agents using an initial value of $\alpha = 0.004$ learned at a steady rate, while agents using $\alpha = 0.001$ throughout the whole training process rarely won against the Minimax agent during evaluation until they exceeded 4 million training games. After only 1.8 million training games, the agents using initial values of $\alpha = 0.01$ and $\alpha = 0.03$ achieved an average score of 50, compared to an average score of 20 achieved by the agents using $\alpha = 0.004$. A further increase of the α value to 0.05 was detrimental, with such agents only achieving a score of 30 after 4.5 million training games. The weights of these agents may fluctuate too often, such that the values of states do not converge. The benefit of larger α values is inconsistent with the results of Thill et al., who noted that agents would not learn until the

value of α dropped below 0.004 (Thill, Koch, and Konen 2012).

Despite the early variations caused by the initial choice of α, almost all agents required 4.5 million training games to reach a score of 80. One potential cause for this may be the rate of decay for the alpha value. Using the rate of decay we specified previously, we observed that the α values were very close to the lower bound of 0.001 after only 1 million games, regardless of the initial value. This implies that changing the initial value of α only has an noticeable impact during the first 1 million games. To further examine this problem, we implemented a new decay scheme using the following slower rate of decay:

$$\alpha(n) = 0.001 + 0.003e^{-0.0000005n}$$

With this decay scheme, the α value approaches the lower bound of 0.001 only after 3.5 million training games. We tested both decay schemes using agents with an initial α value of 0.004. As shown in Figure 5, the agents using a slower α decay scheme had a significantly better initial performance. As these agents have a larger value of α early on, this is consistent with our previous observations about the initial value of α. However, the agents using the slow scheme did not improve after the first 1.5 million training games. The slower decay in α results in more fluctuation in the values of states, preventing them from converging to the optimal value. In contrast, the fast-decaying scheme allows the agent to learn quickly with a large initial α, while decaying rapidly enough for the value of states to converge later on.

Next, we investigated the impact of ϵ on the training process by fixing the $\alpha = 0.004$ and using three different initial ϵ values for training. Here, we decreased ϵ according to the following exponential decay scheme:

$$\epsilon(n) = 0.1 + (\epsilon_{init} - 0.1)e^{-0.000005n}$$

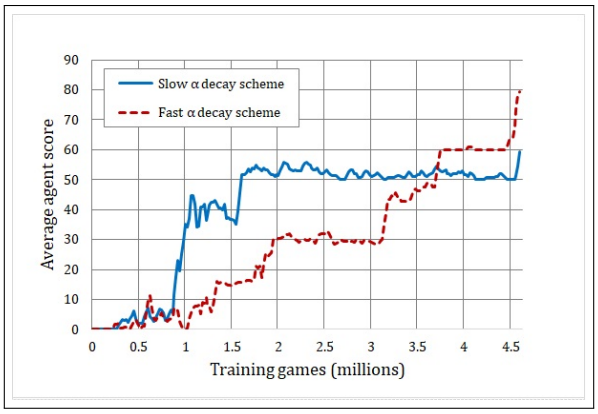
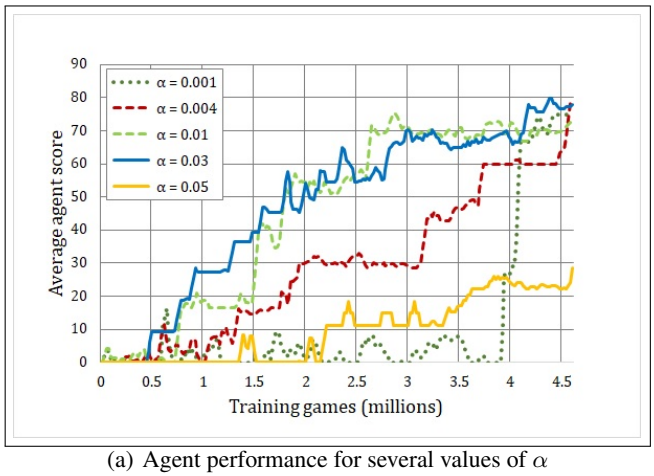


Figure 5: Performance comparison of agents trained through self-play using different α decay schemes. All agents use an initial α value of 0.004 and $\epsilon = 0.1$.

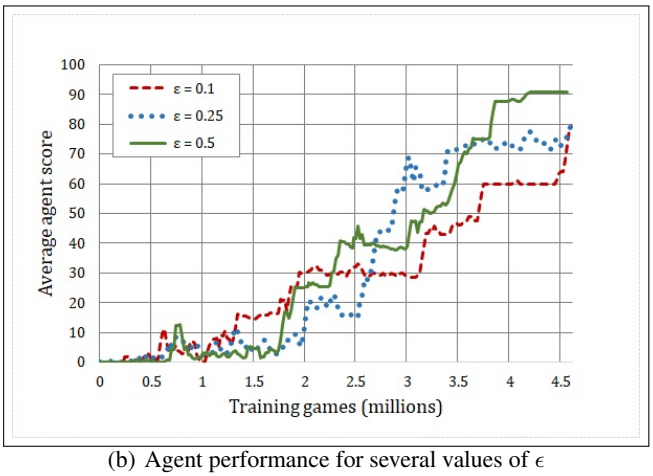
Here, ϵ_{init} is the initial value of ϵ. As shown in Figure 4(b), the initial performance across all three agents is fairly consistent, which agrees with the results of Thill et al. (Thill, Koch, and Konen 2012). However, increasing the ϵ value increases the final performance of the agents, with the agents using $\epsilon = 0.5$ achieving an average final score 10 points higher than that of the agents using $\epsilon = 0.1$. This is inconsistent with the results of Bagheri et al., who concluded that $\epsilon = 0.1$ provided the best performance (Bagheri et al. 2014).

Lessons Learned

Over the course of working on this project, we gained insight into reinforcement learning methods, particularly Temporal Difference Learning. None of us had heard of TDL beforehand, so we had to familiarize ourselves with the topic before we could begin our implementation. In addition, through our research we learned the value of separate



(a) Agent performance for several values of α



(b) Agent performance for several values of ϵ

Figure 4: Performance comparison of agents trained through self-play for different values of α and ϵ. Agents on the left used varying initial values of α, while using $\epsilon = 0.1$ and letting α decay to 0.001. Agents on the right used varying initial values of ϵ, while letting α decay from 0.004 to 0.001 and letting ϵ decay to 0.1.

training, validation, and testing sets for reinforcement learning.

Outside of reinforcement learning, all of us gained valuable hands-on experience with machine learning in general. For example, we saw firsthand how our choice of hyperparameters can dramatically affect our final result. We also found out that training a model can take several hours, and that we needed to prioritize configurations of interest for the hyperparameters in the limited amount of time we had to work with. When assessing the models, we realized that training time was an important factor in deciding the quality of a model, alongside performance - training against the Minimax agent produced better results, but took significantly longer.

Finally, we came to appreciate the value of thoroughly reading and understanding other people's research, which is especially applicable in cutting-edge fields like machine learning. It took us several attempts to correctly implement the algorithm described in the original paper, and only then did we obtain any significant results. We also found that we could not always replicate the results of previous researchers, and would sometimes draw different conclusions.

Discussion

Interpretation

Our original research question was to compare the performance of Connect 4 agents trained through self-play and agents trained against an optimal agent. We compare the two training methods through the agents' winrate against the Minimax agent as well as the number of games required for the agents to achieve a winrate of 80%. As we expected, the TDL agents achieved a higher winrate in the first one million games when training against the Minimax agent (see Figure 3). However, we failed to take into account the higher time complexity of training against the Minimax agent when formulating our research question. After 40 hours of training against the Minimax agent on a 3.1 GHz PC, we were only able to obtain data for the first one million games. In comparison, we required only 5 hours to train the self-play agents for 4.5 million training games using the same hardware. The slow training speed made it infeasible for us to continue training agents against the Minimax agent until they achieved a winrate of 80% with the given time constraints. Therefore, we conclude that using self-play is a better method for training TDL agents to play Connect 4.

We investigated training through self-play further by comparing several configurations of hyperparameters (see Figure 4). None of our tests reduced the average number of training games agents needed to reach a winrate of 80%. Furthermore, we found that varying ϵ had little impact on the winrates of the agents. However, we did find that increasing the initial values of the learning rate α to 0.01 or 0.03 improved the winrate of the agent early in the training process. We hypothesized that our learning rate α may decay too quickly for smaller α values, so we trained some agents using a slower decay scheme. Using this slower decay scheme was detrimental, as the agents were unable to improve their winrates after 2.5 million training games and did not reach a winrate of 80% within 4.5 million training games (see Figure 5). We believe the slower decay in α causes greater fluctuation in the values of states, preventing them from converging to the optimal values. Thus, we conclude that a relatively large initial value of α improves the TDL agents' early performance.

Implications

Our results illustrate the importance of using a simple algorithm for training to minimize the time required for the agents to learn. The advantages of simpler algorithms explains why no previous papers train their reinforcement learning agents against a Minimax agent.

Using an estimated value function significantly speeds up learning, as the estimation allows the agent to generalize to unseen states. The use of the n-tuple framework produced results within hours on a 3.1 GHz PC, while Browne and Scott required several weeks to train their traditional Q-Learning agents using similar hardware (Browne and Scott 2005). The n-tuple system allows the agent to form relationships between groups of board spaces, making the framework more flexible than limiting the agent to predetermined features. Thus, our agents were able to outperform the TDL agents trained by Summerlund, which were only given the raw state of the board or precomputed features (Sommerlund 1996).

Our overall results are most similar to the initial results obtained by Thill et al., whose n-tuple approach we used for our agents (Thill, Koch, and Konen 2012). For self-play, we required a similar number of training games to reach an accuracy of 80% against a Minimax agent. However, we observed that larger initial values of the learning rate α were beneficial, while the agents trained by Thill et al. would not learn until the value of α dropped below 0.004. As Thill et al. did not explicitly describe their α decay scheme in their paper (giving only the initial and final values), we may have used a slightly different decay scheme for our agents. This difference may have caused the inconsistency between our results and the results of Thill et al. Furthermore, the difference implies that our algorithm has different optimal parameters than the algorithm used by Bagheri et al. (Bagheri et al. 2014). Thus, our agents required a greater number of training games than expected when using the tuned hyperparameters found by Bagheri et al.

In accordance with the results of Thill et al., we found that the initial value of ϵ did not strongly affect the number of training games the self-play agents needed to reach a winrate of 80% (Thill, Koch, and Konen 2012). In contrast, after testing 60 different combinations of hyperparameters Bagheri et al. find that certain values of ϵ reduce the number of training games required (Bagheri et al. 2014). Here, simultaneously varying both α and ϵ may produce more accurate results than varying ϵ for a fixed value of α. This would explain why we did not observe a correlation between performance and the initial value of ϵ.

Limitations

One of the largest limitations in our research was that of time. As previously mentioned, we were unable to reach

the target winrate of 80% when training against the Minimax agent within the time constraints. The Minimax agent performs a tree search before making each move and has a much higher time complexity compared to the TDL agent. As a result, each training game takes longer to complete. Thus, we are only able to extrapolate based on patterns in the preliminary data.

Another limitation of our research was that we only evaluated 7 different combinations of initial values for α and ϵ. Using the optimal configuration determined by Bagheri et al. (Bagheri et al. 2014) as a baseline, we selected additional combinations with the intent of producing variations in our final results. A more thorough investigation into initial values for α and ϵ would require a larger search space and likely some form of cross-validation.

To reduce variability in our final results, we averaged the performance of each initial configuration across multiple instances. For each initial configuration we trained 8 instances in parallel. If we had access to greater computing power, we could train a greater number of instances for each configuration at once. Should their average performance be roughly similar to our current results, then we would have more evidence to support our conclusions.

Finally, a major assumption we made near the start of our research was that if our models could beat the Minimax agent, then they could beat any Connect 4 agent. However, a human player could make sub-optimal moves and create a board state that the model has not previously encountered. The model may not know the optimal move in this situation, and make sub-optimal moves as well upon which the human player could then capitalize. This is particularly true for the model trained against the Minimax agent, as it will only have experience playing against an optimal player. As such, more research is required before we can draw conclusions about our models' performance against human players.

Conclusion

Our research problem was to compare the performance of Connect 4 models trained through self-play and Connect 4 models trained against an optimal agent. The motivation behind our research was to explore the capabilities of Temporal Difference Learning as revealed by previous researchers. Connect 4 is a solved game for which an optimal agent exists, which provides us with a benchmark to evaluate our models against. The presence of an optimal agent is what led us to experiment with training against the optimal agent as an alternative to training through self-play.

From our research, we conclude that training through self-play is more effective than training against the optimal agent in the context of Connect 4. Training against the optimal agent produced models with a higher winrate in the first one million training games. However, the optimal agent required a higher time complexity to make each move, which made training against it take significantly longer - it took 40 hours to reach one million training games. In comparison, it took the agents trained through self-play 5 hours to reach 4.5 million games. With respect to the training time, training through self-play is much more effective.

Further investigation into self-play revealed that it was very feasible to train agents that consistently achieve high winrates of over 80%. However, we found different optimal values for the hyperparameters than previous researchers. In addition, our agents required a greater number of training games to reach an 80% winrate.

An important factor that was not fully explored in our research was the choice of α decay scheme. We used an exponential α decay scheme that roughly followed the one used by Thill et al. (Thill, Koch, and Konen 2012). However, it is not known whether this α decay scheme is optimal or if better ones exist. Regardless, it would interesting to further investigate how the choice of α decay scheme influences model performance.

While training against an optimal agent is unfeasible for Connect 4, our findings may not extend to other board games. Our particular issue was that the optimal agent took extremely long to play, which made training against it inefficient with respect to time. If applied to a game where an efficient optimal agent exists, then training against said optimal agent may produce more promising results.

As previously mentioned, one of the limitations of our research was that we only evaluated our models against the optimal agent. A human player could make sub-optimal moves that produce unfamiliar board states, in which our models may not know how to act. To truly understand the capabilities of TDL when applied to Connect 4, further research into the models' performance against human players is required.

Finally, Connect 4 is a solved game for which an optimal agent is known. Most board games are not solved, and the best models use some form of machine learning to guide their play. Even then, they are still sub-optimal agents and can lose against a skilled human player. In this context, one could explore how training against these sub-optimal agents compares to training through self-play. It would be interesting to see how quickly models can be trained to play at the level of the best models, and how well they can imitate their play.

References

Allis, L. V. 1988. A knowledge-based approach of connect-four. *J. Int. Comput. Games Assoc.* 11(4):165.

Bagheri, S.; Thill, M.; Koch, P.; and Konen, W. 2014. Online adaptable learning rates for the game connect-4. *IEEE Transactions on Computational Intelligence and AI in Games* 8(1):33–42.

Browne, W., and Scott, D. 2005. An abstraction agorithm for genetics-based reinforcement learning. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, 1875–1882. New York, NY, USA: Association for Computing Machinery.

Curran, D., and O'Riordan, C. 2004. Evolving connect-four playing neural networks using cultural learning. Technical report, Technical Report NUIG-IT-081204, National University of Ireland, Galway, Ireland.

Krawiec, K., and Szubert, M. G. 2011. Learning n-tuple networks for othello by coevolutionary gradient search. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 355–362.

Lucas, S. M. 2008. Learning to play othello with n-tuple systems. *Australian Journal of Intelligent Information Processing* 4:1–20.

Nasa, R.; Didwania, R.; Maji, S.; and Kumar, V. 2018. Alpha-beta pruning in mini-max algorithm–an optimized approach for a connect-4 game. *Int. Res. J. Eng. Technol* 1637–1641.

Purves, D. 2019. Opinion: What does ai's success playing complex board games tell brain scientists? *Proceedings of the National Academy of Sciences*.

Runarsson, T. P., and Lucas, S. M. 2015. On imitating connect-4 game trajectories using an approximate n-tuple evaluation function. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 208–213. IEEE.

Schneider, M. O., and Rosa, J. G. 2002. Neural connect 4-a connectionist approach to the game. In *VII Brazilian Symposium on Neural Networks, 2002. SBRN 2002. Proceedings.*, 236–241. IEEE.

Sommerlund, P. 1996. Artificial neural nets applied to strategic games. *Unpublished, last access* 5:12.

Stetka, B. 2018. "superhuman" ai triumphs playing the toughest board games. *Scientific American*.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58–68.

Thill, M., and Konen, W. 2014. Connect-four game playing and learning framework. `https://github.com/MarkusThill/Connect-Four`.

Thill, M.; Bagheri, S.; Koch, P.; and Konen, W. 2014. Temporal difference learning with eligibility traces for the game connect four. In *2014 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Thill, M.; Koch, P.; and Konen, W. 2012. Reinforcement learning with n-tuples on the game connect-4. In *International Conference on Parallel Problem Solving from Nature*, 184–194. Springer.

Vermorel, J., and Mohri, M. 2005. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*, 437–448. Springer.