Date of acceptance Grade

Instructor

Policy Improvement in Cribbage

Sean R. Lang

Helsinki June 13, 2018

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Sean R. Lang

Työn nimi — Arbetets titel — Title

Policy Improvement in Cribbage

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	June 13, 2018	59 pages + 0 appendices

Tiivistelmä — Referat — Abstract

Cribbage is a card game involving multiple methods of scoring which each receive varying emphasis over the course of a typical game. Reinforcement learning is a machine learning strategy in which an agent learns to accomplish a task via direct experience by collecting rewards based on performance. In this thesis, reinforcement learning is applied to the game of cribbage, improving an agent's policy of combining multiple basic strategies, according to the needs of the dynamic state of the game. From inspection, a reasonable policy is learned by the agent over the course of a million games, but an increase in performance was not demonstrated.

ACM Computing Classification System (CCS):
I.2 [ARTIFICIAL INTELLIGENCE],
I.2.1 [Applications and Expert Systems],
I.2.6 [Learning]

Avainsanat — Nyckelord — Keywords
machine learning, reinforcement learning, cribbage

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Acknowledgments

I would first like to thank my parents, Kevin and Debra Lang, as well as the rest of my family back home for supporting me throughout the years and especially in the decision to pursue my studies in Finland. I would also like to thank Drs. Teemu Roos and Jukka Kohonen for their mentorship throughout the process of developing and writing this thesis. A gracious thank you also to Chelsea Dunham and Paul Saikko for their help in editing and proofreading this paper, without which, quality would surely have suffered. Additionally, a courteous thank you to Matt Jennings at `dailycribbagehand.org` for providing a data dump which could be used to compare my model against decisions humans made in a variety of situations. Finally, my gratitude is extended to the University of Helsinki for the opportunity to study here and to everybody in Finland who made my time here enjoyable and productive.

Contents

1 Introduction

This thesis will focus on the process of learning how to play a game of cribbage according to a set of predefined strategies via reinforcement learning. The remainder of this section will introduce the main features of the game and the topic of reinforcement learning for readers unfamiliar with either. Section 2 will provide an overview of related work in reinforcement learning with relevance to playing games as well as previous research done on the game of cribbage. The process used to develop the training framework will be described in Section 3. The results of the training and several experiments are explained in Section 4. Finally, a discussion of the potential applications for this thesis's findings and a summary of improvable areas are presented in Section 5.

1.1 Cribbage

Cribbage is a multi-phase card game, typically played between two opposing players. While variants exist for three or more players, this paper will focus on the two-player variant. A well-known characteristic of the game is its board: players will usually keep track of their points by moving pegs between holes drilled along the board in a process called *pegging*. The game presents an interesting research area because of its unique scoring methodology: each hand is counted in two different ways within each round and the first player to reach a score of 121 points or more is declared the winner. Because of its win condition, different strategies hold differing levels of importance throughout the game.

Rules of the Game

In order to be able to understand the temporally dependent nature of the strategies, the rules and flow of a game of cribbage must be fully understood. While a complete set of tournament rules can be found at [Ame18] or [Ame16], what follows is an overview complete enough such that a novice player, with the assistance of the scoring rules found in Table 1, could play a complete game.

The zeroth step, taken once per game, is to determine which player will be the dealer for the first round and who will be the pone. In order to determine these roles, each player cuts the deck in turn to get a single card: the player with the lower-ranked card[1] is the dealer; the other player is called the pone. In the case of a tie, this step is repeated until two unique cards are cut from the deck. From there, the usual round structure begins and proceeds in the following steps:

1. Each player is dealt 6 cards.
2. Each player selects 4 cards to keep for their own hand and 2 cards to discard, or toss, into a collective discard pile, called the crib.

[1] Ace $<$ 2 $<$ 3 $<$ 4 $<$ 5 $<$ 6 $<$ 7 $<$ 8 $<$ 9 $<$ 10 $<$ Jack $<$ Queen $<$ King; suits are irrelevant.

Cards or Combinations	**during play**	**in hand or crib**
Jack turned by dealer as cut card	2	—
Jack in hand or crib of same suit as cut card	—	1
two of a kind (pair)	2	2
three of a kind (3 pairs)	6	6
four of a kind (6 pairs)	12	12
straights of three or more cards (per card)	1	1
15-count (sum of any combination of cards)	—	2
four-card flush (only in the hand)	—	4
five-card flush	—	5
reaching a 15-count exactly	2	—
reaching a 31-count exactly	2	—
final card played (without reaching 31-count)	1	—

Table 1: Scoring Rules for a game of cribbage [Ame16]

3. The deck is cut at a random location by the pone and the top card from this cut is selected by the dealer and placed face-up on top of the deck. If this cut card is a Jack of any suit, the dealer is awarded 2 points and pegs the points accordingly.

4. Starting with the pone, each player alternates playing a single card by placing them face-up on the table, keeping track of the total value[2] of all cards played so far, until all cards have been played or neither player can play a card without exceeding a collective value of 31. If any of the situations or combination of cards mentioned in Table 1 is seen in the immediately preceding cards, the amount of points earned is immediately pegged on the board for the player who played the last card. In cribbage terms, this is called *the play* or, perhaps confusingly, in certain circles due to the rapid nature of the action, *pegging*. These terms will be used interchangeably throughout this paper, with a preference for *pegging*. This process is repeated until both players' cards have been exhausted.

5. After all cards have been played, the pone then counts his or her hand using the randomly cut card from Step 3 as a fifth card in the hand before pegging these points on the board.

6. The dealer proceeds to count his or her hand and peg the points in the same fashion, also considering the randomly cut card to be the fifth card in the hand.

7. The dealer then does the same for the crib.

8. The dealer and the pone swap roles and repeat from step 1.

[2]The value of all numbered cards is that number, aces have a value of 1, and all face cards have a value of 10.

The game is immediately over when either player achieves a score of 121.

The win condition for this game can occur at any moment of the game, even beyond either player's control (note Step 3). Because of this, it is crucial to play according to different strategies during different times of the game, based on the scores of both players and who is the dealer for that round. Typically, during early- and middle-game play, the pone will attempt to maximize their own hand, while avoiding giving too much opportunity for the dealer to score points from the crib. However, in later play, this may no longer be a concern. For example, should the player be playing as the pone with a score of 116 and the dealer has 117 points, then due to the counting precedence, the player needs not concern themselves with what points the dealer will obtain through the crib, if their own hand has at least 5 points guaranteed, since the pone will count first and win. This only works, however, if the pone does not allow the dealer to score 4 points from the play. As can be seen, the player must balance multiple, competing factors with varying emphasis over the course of the game.

1.2 Reinforcement Learning

Reinforcement learning is the machine learning equivalent of learning from one's failures rather than being coached to the correct answer.[3] In classical machine learning methodologies, the agent discovers an optimum model for a problem by approximation methods centered around minimizing a set loss function over a given set of data while assuming a known model for the solution. In reinforcement learning, however, the agent finds the optimal solution to the problem by repeatedly taking an action in an environment and gaining a reward or punishment for each action taken. It is the same principle used in teaching a pet or animal to do a trick: offer a full or partial reward for successful completion of the trick or for progress in the correct direction. As a comparison to teaching a human how to add, the strategy used in classical machine learning would be what is used in classrooms today: teach the method of adding digits and handling carry-over, giving some guidance and sample problems to ensure the technique is solidly replicable and generalizable to previously unseen problems. Meanwhile, teaching a human how to add by reinforcement learning would mean merely quizzing the subject by asking him to answer an addition problem while giving a vague hint as to how right or wrong they were. After enough rounds of this, the student will eventually figure out his own method for adding two numbers with the same accuracy as an established method.

While this may seem like a silly example where classical methods would clearly be the superior method, where reinforcement learning comes into its own is in situations in which no uniquely optimal answer exists for a problem. Take, for instance, the problem of learning chess. In humans, basic strategies for how to handle certain

[3]For the sake of clarity, all information presented in this section will not be cited in-line, but is referenced from Sutton and Barto's textbook [SB].

situations can be taught, but these are all from one's own or others' prior experience. Similarly, while strategies may bolster confidence in heretofore unseen situations, they cannot possibly cover all possible chessboard layouts. Arguably, the best way to learn, in humans and computers, is by doing. After a game has been played, the player can see what worked and failed during the game to cause the win or loss and extrapolate what to do, if a similar situation occurs, in order to improve play. Classical teaching methodologies would not be highly applicable in this situation because there is no known single optimal strategy that cannot itself be countered.

Agent, Environment, and Rewards

Reinforcement learning scenarios are modeled as Markov decision processes, with the three most important, constantly interacting components being the agent, the environment, and the rewards. The agent is the actor that makes decisions and learns the task at hand. The agent must learn to navigate the environment in order to maximize its rewards, much like a mouse navigating a maze to retrieve the cheese at the end.

The Environment In reinforcement learning scenarios, the agent interacts with and navigates what is known as the environment. The environment is a set of states in which an agent can find itself. What exactly constitutes the environment is problem-specific. An individual state can be any situation in which the agent finds itself and can be in either discrete or continuous space. For instance, in chess, a discrete state would be a specific board arrangement. In golf, an example of a state in continuous space would be the location of the ball along the course of play and the current wind velocity. An action is an interaction the agent can make with the environment to alter its current state. In the example of chess, an action is discrete and would be to move a piece X to position Y, e.g. moving the bishop to g4 ($Bg4$). In golf, the action is continuous and may be which club to use in which direction and with how much power.

State transitions within the environment may also be stochastic and the result of taking an action may not always be entirely predictable. For instance, in the example of golf, a sudden gust of wind may occur that pushes the ball off course. This means that reaching a desired outcome state by taking a given action is not necessarily a certainty, but proceeds according to the probability $P(s'|a, s)$ of reaching state s' from state s by taking the action step a.

Rewards and Goals Merely being able to navigate an environment does not satisfy the requirement for learning unless a given task is being completed. This desired task can be called the goal of the agent. For games scenarios, this is frequently simply the notion of winning.

A reward is a feedback event that encourages or affirms progress towards the goal and can be thought of as a way of enticing the agent to accomplish the task. As

with a dog learning to jump through a hoop, the reward of a treat is given after the dog has successfully jumped through the hoop, or perhaps a partial treat for first walking through a stationary hoop or other similar subtask. Rewards can also be negative and thought of as punishments for not completing the task, or doing so in an undesired manner. The goal of the agent is to maximize its cumulative received reward in the future.

Expressed mathematically, R_t is the reward at a given time t. An episode is a set of events that can be logically separated within the lifetime of the agent, e.g. an individual game of chess rather than a match, or a single hole of golf rather than a full round. The rewards sequence, $R_{t+1}, R_{t+2}, \ldots, R_T$, where T is the final time step of the episode, is a specific sequence of future rewards that may be collected by the agent. Due to the stochastic nature of the environment, there are, naturally, multiple possible rewards sequences. A return, G_t, is a function of a given rewards sequence, most simply its sum:

$$G_t = \sum_{k=t+1}^{T} R_k$$

This return formula can also incorporate a discounting factor γ to encourage actions conducive to reaching the terminal state in a speedy fashion:

$$\begin{aligned} G_t &= \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \\ &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

A rational agent will take an action to maximize its expected return.

Consider the scenario of trying to escape a maze. The reward for the agent may be $+1$ when it successfully exits the maze and 0 at all other times. Without a discounting factor in the returns, there would be no incentive to exit the maze in a timely fashion as the return remains the same no matter how long the agent takes. To handle this without a discount factor, the rewards would need to be negative for all non-exit spaces so that the return is worse the longer the agent remains in the maze.

Policies A policy is a mapping from states to probabilities of taking possible actions. A policy π describes a set of probabilities $P(a|s)$ of taking action $a \in \mathcal{A}$ when in state $s \in \mathcal{S}$ where $\mathcal{A}$ is the set of all possible actions and $\mathcal{S}$ is the set of all states in the environment. An optimal policy π_*, of which there may be several, is any policy which achieves a maximum expected reward over the course of taking its actions.

Learning an Optimal Policy

Although applicable to both discrete and continuous state representations, as long as it can be represented as a Markov decision process, it is useful for the sake of illustration to limit the scope of discussion to discrete representations. Heretofore, all discussion will assume a discrete representation as the domain of cribbage falls into this category and it streamlines notation.

Metrics A state can have a worth or *value* associated with it, for a given policy π. The value of a state $s \in \mathcal{S}$ under policy π is denoted $v_\pi(s)$ and is defined as the expected total return by following policy π from state s:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} P(s', r|s, a) \left[r + \gamma v_\pi(s') \right] \end{aligned}$$

where t is the arbitrary time within the episode when the agent is in state s, to maintain compatibility with previous definitions regarding rewards sequences. The optimum value of a state $v_*(s)$ is defined as:

$$v_*(s) = \max_\pi v_\pi(s)$$

Similarly, an action can have its own worth or *quality* assigned to it under a specific policy. The quality of an action $a \in \mathcal{A}$ at state $s \in \mathcal{S}$ under policy π is defined as:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \end{aligned}$$

and its optimum $q_*(s, a)$ is consequently defined as:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

As previously discussed, an optimal policy π_* is any policy which maximizes the expected reward received. If there are optimal and suboptimal policies, then it follows that there are ways in which policies can be compared. A policy π is greater than or equal to another policy π' if and only if the value of all states under policy π are greater than or equal to the value of all states under π'

$$\pi \geq \pi' \text{ iff } v_\pi(s) \geq v_{\pi'}(s) \ \forall s \in \mathcal{S}$$

Policy Evaluation Policy evaluation is the iterative process of approximating the state-value function v_π for some policy π. When the entire state space is observable, since $v_\pi(s)$ can be expressed recursively, the value of a state can be repeatedly updated to its cumulative expected reward when taking each possible action, converging to the true state-value function. However, as this convergence only occurs at the limit, the calculation process can be stopped when its magnitude of change indicates that it is sufficiently accurate, as shown in Algorithm 1. Due to its iterative nature, however, this can be a slow process as each state can affect others in a cascading fashion.

Algorithm 1 Policy Evaluation

Require: π (the policy), θ (some small number)
  Let $V[1...n]$ be an array of values for all states ($n = |\mathcal{S}|$)
  repeat
    $\Delta \leftarrow 0$
    for all $s \in \mathcal{S}$ **do**
      $v \leftarrow V[s]$
      $V[s] \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma V(s')]$
      $\Delta \leftarrow \max(\Delta, |v - V[s]|)$
    end for
  until $\Delta < \theta$
  return $V \approx v_\pi$

Policy Improvement and Iteration After the value of a policy can be accurately estimated, it is then possible to improve on that policy. This process is as simple as greedily choosing the action at each state such that the expected resulting reward is maximized. Intuitively, this is an optimal policy, given the current knowledge of the environment, since each action taken will always make the most rational decision in whichever state it finds itself in. The idea of policy iteration is formed from this simple greedy policy improvement mechanism. Alternating steps of policy evaluation and policy improvement are repeated, converging until a stable policy is reached that is within some precision bound of optimal. This process is given in Algorithm 2 and can be visualized as:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*}$$

where $\xrightarrow{E}$ shows policy evaluation and $\xrightarrow{I}$ shows policy improvement. Although Algorithm 2's conditional check in Line 5 of Step 3, written as is, can allow cycling between multiple optimal policies, the algorithm is illustrative and care can be taken in implementation to remove this possibility.

Value Iteration As mentioned, the iterative nature of policy evaluation leads to a slow convergence rate for the process of policy iteration. However, full convergence

Algorithm 2 Policy Iteration

1. Initialization:
   $V(s) \in \mathcal{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all s

2. Policy Evaluation (see Algorithm 1).

3. Policy Improvement:

```
policy-stable ← true
for all s ∈ S do
    old-action ← π(s)
    π(s) ← argmax_a Σ_{s',r} P(s', r|s, a) [r + γV(s')]
    if old-action ≠ π(s) then
        policy-stable ← false
    end if
end for
if policy-stable then
    return V ≈ v_* and π ≈ π_*
else
    Go to Step 2.
end if
```

does not need to occur for an optimal policy to be computed. Instead, a very truncated form of policy evaluation can be done to improve the speed of evaluation while still converging to an optimal policy. A special case of only performing one pass of policy evaluation and improvement during policy iteration is called value iteration and is shown in Algorithm 3. This algorithm still converges to optimal policy, just with less accurate steps made in improvement. Rather than make a slow, calculated step in the absolute best direction, value iteration takes a quick step in a generally good direction, always improving its situation and allowing later improvements in judgment to compensate for the potentially poor steps previously taken.

Algorithm 3 Value Iteration

Require: V initialized arbitrarily (e.g. $V(s) = 0$, $\forall s \in \mathcal{S}$), θ (some small number)

```
repeat
    Δ ← 0
    for all s ∈ S do
        v ← V(s)
        V(s) ← max_a Σ_{s',r} p(s', r|s, a) [r + γV(s')]
        Δ ← max(Δ, |v − V(s)|)
    end for
until Δ < θ
return Deterministic policy π ≈ π_*, such that
  π(s) = argmax_a Σ_{s',r} p(s', r|s, a) [r + γV(s')]
```

Monte Carlo Methods Policy iteration requires full knowledge of the environment to be computed. However, this is often not practical in real-world applications. Monte Carlo methods are a set of ways in which state-value functions, action-value functions, or policies can be estimated through experience without complete knowledge of the environment. These experiences can come from episodes in the real environment or through simulated encounters. By averaging observed returns from experienced episodes, the state-value function, and thus policy, can be learned.

The key problem in learning through experience is how to balance exploitation of the current value function or policy to increase immediate returns with exploration of the environment to discover potentially better returns. An agent which explores too much may not have very accurate knowledge of each action's eventual outcome and often sacrifices better returns in the attempt. Comparatively, an agent which explores too little may not discover that a better outcome is possible than those which it already knows. There are two mechanisms to combat this issue and ensure the necessary exploration of the environment while also exploiting current knowledge to maximize rewards. The first, using exploring starts, starts each episode randomly in the environment's state-space. An algorithm for this sort of Monte Carlo method is given in Algorithm 4. This mechanism forces the agent to explore a different region of the state space than it may likely reach on its own. The other, using an ϵ-greedy policy, follows the policy less strictly, taking an action at uniform random with chance ϵ. Taking an action at random puts the agent in situations just outside its normal operating area. Whereas random starts can be thought of as global exploration, random actions would be a local exploration step, widening the knowledge area around previously known states. Naturally, both of these methods can be combined to further ensure adequate exploration.

Algorithm 4 Monte Carlo with Exploring Starts

Initialize for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

- $Q(s, a) \leftarrow$ arbitrary
- $\pi(s) \leftarrow$ arbitrary
- $\mathit{Returns}(s, a) \leftarrow$ empty list

Repeat forever:

- Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ such that all pairs have probability > 0
- Generate an episode starting from S_0, A_0, following π
- For each pair s, a appearing in the episode:
    1. $G \leftarrow$ return following the first occurrence of s, a
    2. Append G to $\mathit{Returns}(s, a)$
    3. $Q(s, a) \leftarrow \text{average}(\mathit{Returns}(s, a))$
- For each s in the episode:
    - $\pi(s) \leftarrow \text{argmax}_a\, Q(s, a)$

2 Literature Review

In this section, an overview of major advancements in reinforcement learning as it has been applied to the domain of games is given. Following that, previous research into the topic of cribbage is presented.

2.1 Reinforcement Learning

From Samuel's seminal work in checkers [Sam59] to more recent accomplishments against Grandmasters of Go [SHM+16], the domain of games has been an area of great interest in machine learning research. Games provide an isolated environment with a set of distinct rules and clear objectives which make them well suited to being expressed as Markov decision processes [Sam59]. This, in turn, allows for modeling as a reinforcement learning problem and subsequent exploration.

History

No paper on reinforcement learning of sufficient size would be complete without mention of Arthur L. Samuel's foundational paper on machine learning applied to checkers [Sam59]. In this paper, despite the technological limitations of his time, Samuel develops and describes two main methodologies he used to make an agent learn how to play checkers.

The first method Samuel devised is what he called "rote learning." This strategy applied an arbitrarily decided polynomial value function to evaluate board positions. Future positions were then found by searching a minimax tree of limited ply depth in which each agent was assumed to play optimally given its own position. Although only a certain number of moves were allowed to be searched forward, a record of previously evaluated positions was stored. These kept records would then be used to virtually extend the depth of the search tree. For instance, if the ply depth of the search is three and a previously evaluated position was located as a leaf on the search tree of the current position, then the effective depth of the tree along this route is now six since the stored position has already been evaluated a further three plies. Samuel even recognized the idea of rewards being necessary for learning a task with what he called a "pushing mechanism," which would direct the agent towards winning the game and not just picking better positions.

Although the learning aspect of his program was limited to merely evaluating deeper and deeper plies and saving results, Samuel's program was trained through self-play, recorded games, and occasionally through human interaction. The recorded games, or "book games" as Samuel termed them, provide a mechanism for supervised learning in which the agent could track its own choices against those made by the humans in the actual game, providing another method to update the value of a board position. After training, and without imposing human knowledge such as a bank of openings, the playing program was evaluated to play on the same level as a

better-than-average novice.

The other learning method Samuel explained is what he called "generalized learning" and was a preliminary simulated annealing algorithm which was applied to improving the board position evaluation polynomial. In the paper, two agents would play against each other: the first (Alpha), in which the polynomial parameters could be tweaked, the other (Beta), using the same starting parameters, but never updating them. If Alpha won enough games in a tournament, Beta would inherit Alpha's parameters and the adjustments would begin again. This had the effect of incrementally improving the learned parameters. If Alpha was incapable of consistently beating Beta, then a local optimum was recognized and Alpha's polynomial would be drastically mutated to another starting position by eliminating the leading term. Although the reader is cautioned that the learning method is not guaranteed to find globally optimal parameters, Samuel's proposed hill-climbing technique created an agent capable of playing a better-than-average game of checkers.

Perfect Information Games

Perfect information games are those in which the entire game state is observable at any given time [Kho10]. Furthermore, an action has a definite and entirely predictable outcome: e.g. moving a chess piece on the board in a specific arrangement will always lead to the same resulting arrangement. Checkers, chess, and Go are all examples of perfect information games.

Jumping ahead by nearly 40 years, the basic ideas of Samuel's paper can be seen applied to the slightly different domain of chess with IBM's DeepBlue and its victory against Garry Kasparov. The DeepBlue team greatly expanded upon the idea of a minimax search to create a massively parallel search system for chess evaluation [CHJH02]. The final system was able to evaluate hundreds of millions of chess positions per second, millions of times faster than Samuel's capabilities at the time. However, the speed in evaluation was a result of hard-wiring an evaluation function by using chips custom-built for the purpose. As with Samuel's rote learning, very little in terms of actual learning how to play or adjustments to internal mechanisms occurred beyond an expanded search space made possible by more capable hardware. Despite this philosophical gripe, the IBM team was able to outplay a Grandmaster of chess, winning the match 3.5 games to 2.5.

Of notable attention recently is Google DeepMind's success in the game of Go with AlphaGo [SHM$^+$16]. More of a learning solution than DeepBlue's engineering solution, AlphaGo used multiple neural networks to evaluate value and policy functions. A first neural network was trained via supervised learning, using a database of human-played games to predict human play for a given board position. Using this network as a starting point, a policy network was trained via reinforcement learning to win more games than its previous renditions. From that policy network, a value network was trained to determine the likelihood of winning when in a given state. Both of these previous networks were trained through self-play. The AlphaGo

program would use both the value and policy networks in a lookahead search algorithm to determine the optimal move to make next. Potential game outcomes were searched in a similar fashion to minimax, but using the policy network to determine which move to make at any state, rather than evaluating all possible legal moves and pruning. Starting from the current state, possible games were simulated by using the policy network, expanding each potential outcome in the tree by using the value network and deeper simulations, also incorporating exploration mechanisms. After all searches reached a terminal state, the most commonly taken action from the starting position was selected as the agent's move, as it represents the choice which would optimize the expected outcome of simulations using all combined knowledge. The final system resulting from this training pipeline was able to beat European Go Champion Fan Hui in match play five games to zero [SHM+16]. A more trained version of AlphaGo would eventually beat Grandmaster Lee Sedol four games to one [Dee18].

DeepMind then took a step even further, removing all human-imposed knowledge and training a single neural network to play *tabula rasa*, i.e. from scratch. Using only this single neural network, the team trained AlphaGo Zero, the new program, using a simplified Monte Carlo tree search algorithm to evaluate positions during the game. To train the network to predict values more closely aligned with those of the simulated policy, after a self-play game had been played, states observed during the game were used to update the network's weights. After only 72 hours of training and using only a single machine, AlphaGo Zero was able to defeat the previous version of AlphaGo one hundred games to zero [SSS+17]. For context, AlphaGo was distributed across multiple machines and trained over a period of several months.

Stochastic Games

In contrast to the predictability of perfect information games, there exist games in which the entire game state is not fully observable or in which outcomes involve random chance. These latter games, in which state transitions proceed according to a set of transition probabilities, are called *stochastic* [Sha53]. Examples include games involving dice rolls (e.g. backgammon, craps), playing cards (e.g. poker, blackjack, cribbage), and other games of change (e.g. roulette, slot machines).

A pivotal example of machine learning applied to stochastic games, and a heavy influence in AlphaGo's creation, was TD-Gammon [Tes95]. In his paper, Tesauro designed and trained a simple feed-forward multilayer perceptron to play backgammon. Backgammon is an ancient board game in which moving abilities are determined through dice rolls. The inclusion of dice rolls provides a stochastic environment since a desired move may or may not be possible given a dice roll. As a result, the searching methodology of DeepBlue or Samuel's checkers is not applicable as there is no guarantee of reaching a given state. Instead, the neural network was trained to predict the likely outcome of a game from a given position through observing several recorded games. After a game had finished, the graph's weights were readjusted by using a final reward signal representing the actual outcome of the game.

Since predictions made later in the course of the game were expected to be more accurate than those made earlier, a system of diminished rewards was implemented, called TD(λ), based on the idea of temporal difference learning, a mechanism for apportioning credit assignments to states when the reward signal is delayed. Tesauro's implementation of TD(λ) updated the weight differences according to the formula:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

where α is a learning rate parameter, $\nabla_w Y_k$ is the gradient of the output with respect to the weights, and λ controls how far back credit is assigned to an outcome or more accurately how much that credit decays. For instance, $\lambda = 0$ means only the last evaluation is credited, while $\lambda = 1$ means all evaluations are credited equally and values in between provide smoothly decaying credit assignments.

Later experiments extended training through the use of self-play. TD-Gammon would play both sides of a single game and adjust its weights according to the observed result. This training method was proven to be highly successful when TD-Gammon was entered into the 1992 World Cup of Backgammon tournament and was shown to be a contender, losing by only 7 points total over the course of 38 games with highly respected players and even former world champions. In qualitative evaluations by Kit Woolsey, one of backgammon's most respected analysts, TD-Gammon was declared to play better than humans in situations which were considered more complex, as its evaluation function was able to precisely calculate the chances of winning without bias or emotion. TD-Gammon so impressed the backgammon community that strategies previously considered inferior due to human bias have been revisited and reanalyzed due to their favorability in TD-Gammon.

Tesauro credits this success to both the self-play method of training as well as phenomena directly inherited from the nature of the game of backgammon. The motion of the game is reliant on the outcomes of random dice rolls; the nature of the outcomes of these rolls naturally contributes to the exploration of a wide search space without the need for explicit exploration steps as would be needed in a perfect information game such as chess. Furthermore, since checkers can only be moved forward with the exception of being knocked back by an advancing opponent, the game cannot enter a non-terminating loop and must eventually terminate.

In addition to the backgammon community, TD-Gammon made leaps and bounds in the area of games AI research, especially with respect to neural networks. After the first few thousand training games, the system was capable of playing elementary backgammon, recognizing a few basic strategies for decent play. This was determined to be the result of simple combinations of features which could be quickly evaluated by a linear function of the raw board position. More complicated, context-sensitive strategies would develop later in training and are credited as being the result of the ability of neural networks to detect nonlinear features.

While Tesauro credited the neural network architecture and the game itself for the success of TD-Gammon, Drs. Pollack and Blair from Brandeis University demon-

strated that TD-Gammon's successes were not a result of temporal difference learning or even reinforcement learning necessarily, but due to the nature of backgammon itself and how self-play contributes to a constantly evolving environment [PB97]. In their efforts, Pollack and Blair applied basic hill-climbing techniques to a simple neural network setup similar to that described in Tesauro's paper. To this network, the authors applied a similar strategy to Samuel's generalized learning: a mutant was created by adding Gaussian noise to the "champion" weights. The normal weights played the mutant for a number of games. Each of these games was played in pairs so each agent got the chance to play as white, and started with the same random seed to avoid unfair bias in the dice rolls. With this setup, if two agents are identical, then the result of the two games would be one win each. If the mutant then won enough games, the champion's weights were shifted in the direction of the mutant. At no point was reinforcement or temporal difference credit used to adjust the neural network's weights.

The authors' system was able to perform similarly to an earlier version of TD-Gammon, based on comparative performance against other backgammon-playing systems. This led the authors to the conclusion that the success of TD-Gammon was more likely the result of the dynamics of backgammon and how they contribute to providing an excellent environment for self-play learning, not necessarily a result of the technique used in optimizing the neural network. In normal student-teacher learning, a teacher will attempt to point out edge cases of the student's knowledge and the student, in turn, learns by asking tougher and tougher questions. In self-play learning, the teacher and student are the same, so it is possible that a student and teacher can reach a draw situation in which both sides are placated and neither is challenged to further best the other. Backgammon, however, cannot end in a draw, so one of the players is always forced to improve.

2.2 Prior Cribbage Research

On the topic of cribbage, very little research has been done, perhaps as a result of its relative lack of popularity to such games as poker or its stochastic environment being too large to tackle. There are three main papers in which cribbage receives the main focus: a mathematical analysis of dealer advantage [Mar00], a genetic learner applied to the discard phase [KS02], and a multilayer perceptron attempt to apply a simpler version of TD(λ) to cribbage [O'C00].

For his senior thesis at Harvey Mudd College, Philip Martin set out to determine if the player which begins as the dealer had any statistical advantage in winning [Mar00]. The method used to accomplish this task was to enumerate and evaluate all possible combinations of cards which can be dealt to a player. From there, a matrix of potential average crib scores was created as a table indexed on each axis by one of $\binom{52}{2}$ combinations, introducing the minor inaccuracy of forgetting which cards have been seen by each player from their respective hands, but vastly decreasing the amount of computation needed. Using the bounds found, certain basic strategies were proven to be suboptimal for the player's own hand since their

expected outcomes were found to be below the lower bound for optimal strategies, e.g. avoiding throwing a pair or fifteen to the crib as the pone. Despite the slight inaccuracy and assumption of a uniform distribution, the paper concluded that being the first player to act as dealer gave the player an expected advantage of approximately 5 points on average as a result of starting with a guaranteed count of a crib. However, since different strategies needed for other portions of the game were not mentioned, by the author's own admission, this study barely scratched the surface of research into the game.

The next, and perhaps most useful, work performed on the game of cribbage is Robert O'Connor's undergraduate project at the University of California, Berkeley [O'C00] on adapting TD(λ) [Tes95] to the domain of cribbage. O'Connor trained a feedforward multilayer perceptron with only a single hidden layer to play a single hand of cribbage. He used TD(λ) to adjust the weights as training proceeded with the network playing full games against itself. No score context data was included in the input vector for training. After training for millions of games, the agent was able to choose hands reliably better than random, but just shy of an algorithm which would choose by maximum expected outcome. As there was no previous work done to produce an AI to play cribbage, no comparison could be made. Furthermore, as the results of games vary wildly due to luck of card games, an infeasible amount of games would need to be played against a human to determine if any advantage was present, and thus a comparison was not made to human play either. Regardless, the system was demonstrated to be learning as it played significantly better than random over the course of matches of one thousand games each.

It should be noted that the paper lacks significant details as to how the network was used, precisely. The output of the network was a single linear output representing the state utility function at any state. These states were represented by 209 binary values representing various cards in hand or in play. There is no mention of how this state space is explored or how this value function is applied to the agent making a decision; it can only be assumed that the agent made the greedy choice among potentially reachable states for each possible legal move.

The final paper worth addressing which attempts to learn cribbage is Kendall and Shaw's adaptive cribbage player [KS02]. The authors used an evolutionary strategy, similar to a genetic algorithm, to make an agent learn to play the discard phase of the game. In doing so, the limitation was imposed that the suit of a card would not be considered, sacrificing a small degree of accuracy in calculations to vastly simplify the search space. For each hand dealt, the evolutionary algorithm attempted to learn a set of weights for choosing which cards to keep and which to throw into the crib, in order to optimize its own points gained. The choices made were contrasted against optimal possible choices for that hand when including the cut card in order to determine fitness, and weights were updated if the resulting hand scored below a threshold percentage of optimal. While initial results were positive and learning occurred alongside an increase in points obtained by the agents, performance eventually returned to levels on par with the initial randomized weights. This was because values associated with each card converged until suboptimal behavior was

reintroduced.

Additionally, the authors attempted to use a co-evolutionary strategy in which two agents would learn to adjust their weights for each hand when playing each other by using the opponent's score as comparison for weights adjustment. This proved to be detrimental to agent performance when different hands were dealt to each competing agent because one agent must always update its weights as the loser and because losing may be unfair since the values of hands can vary widely. However, when the agents were trained by using the same set of cards, results were far more positive since the losing agent was always challenged to improve on its present choice in a fair manner.

The paper continued by prompting the evolutionary agent to learn a different set of weights for the hand when playing as the dealer than as the pone. As a demonstration, only a single set of dealt cards was used for the task and different sets of weights were learned for each position, showing that learning in different roles was possible.

As a final experiment, the trained agents were played against a commercially available cribbage-playing program on various difficulty setting levels. To supplement the developed card-choosing agent, a simple heuristic was included to play the pegging portion of the game. This conglomerate agent was able to easily outperform the opponent program on its 'easy' difficulty. The matches were more evenly matched between the agent and the program on its 'medium' difficulty, narrowly winning three of five games. The 'hard' and 'harder' difficulties proved too much for the simple pegging heuristic and the agent lost all five games.

Thus, the authors demonstrated that an agent can be made to learn how to choose a combination of cards well using evolutionary methods. Furthermore, they demonstrated that the agent could be trained to recognize differences in play present when playing as the dealer or as the pone.

3 Methods

In this section, an in-depth explanation of the experimental learner's setup is provided. Multiple basic strategies which a human player would use to play cribbage are presented and a mechanism for their combination is given. Finally, the intended training mechanism for the agent is covered as well as an overview of experimental changes made to determine the precise nature of learning.

3.1 Strategies

A few basic behavioral strategies were coded for the agent to learn over the course of training. These represented most of the basic factors which a player may consider when making their decision as to which cards to keep as well as some that are not quite as useful. These included:

- `hand_max_min`: The hand(s) with the highest minimum possible score for the kept cards will be more highly desired. This is equivalent to choosing the hand with the largest guaranteed points.
- `hand_max_avg`: The hand(s) with the maximum average points over all possible cut cards will be the most highly desired. This strategy is useful for trying to maximize the expected score of one's own hand.
- `hand_max_med`: The hand(s) with the maximum median points possible to score will be the most highly desired.
- `hand_max_poss`: The hand(s) with the maximum possible score will be the most highly desired. This strategy can be thought of as a Hail Mary choice for the player trying to score as many points as possible, not taking into account its potentially low likelihood to become reality.
- `crib_min_avg`: The hand(s) whose tossed cards led to cribs with the lowest average amount of points for the dealer will be the most highly desired. This strategy would be a very defensive strategy, typically used by the pone to avoid giving points to the dealer.
- `pegging_max_avg_gained`: The hand(s) with the maximum average points gained through pegging will be the most highly desired. This strategy would be useful for end-game play in a tight game. For instance, if both players are close to winning, the dealer may choose to forego placing points into their hand and instead try to "peg out" since it is unlikely that they will get a chance to count their hand at all.
- `pegging_max_med_gained`: The hand(s) with the maximum median points scored during the play will be the most highly desired.

- `pegging_min_avg_given`: The hand(s) with the minimum average points scored by the opposing player will be the most highly desired. This is a very defensive strategy also useful at the end of the game to prevent the opposing player from pegging out.

The above definitions refer to a hand's desirability. This can be thought of as an internal ranking of how likely a given strategy would be willing to choose a particular combination of cards. This desirability score is then scaled to lie in the range $[0, 1]$ with 1 representing the best possibilities and 0 the worst. The scaling is accomplished by creating a linear scale between the worst and best values for the retrieved statistic. For example, in the `hand_max_min` strategy, if the most any combination of cards was guaranteed to score was 12 points, while the least was 2 points, those guaranteed 2 points would be given a score of 0, those guaranteed 7 points would be given a score of 0.5, and those guaranteed 12 points would be given a score of 1. In the case of those strategies which advocate for minima (i.e. `pegging_min_avg_given` and `crib_min_avg`), these scores would be reversed since a smaller minimum is more desired by these strategies. Scaling in this manner allows for combinations which are almost as good to not be ignored in later weighting stages.

Statistics

Because of the massive amount of possible combinations—all of which were unique due to the cards' position affecting the scoring outcome—the calculation of some of these statistics involving the crib took a handful of seconds to evaluate on-demand during development. This was deemed much too slow as, over the course of hundreds of thousands of simulated games, this delay would very quickly accumulate to performance-affecting delays. As a result, an alternative strategy of pre-computing the required knowledge and storing the values of interest into a relational database was implemented instead.

There are $\binom{52}{6}$ possible combinations of cards which can be dealt to either player. Of these possibly dealt hands, there are then $\binom{6}{4} = 15$ possible combinations of cards that can be kept, with the remainder tossed to the crib. For each of these 15 possible combinations, there are a further 46 remaining cards that can be considered as possible cut cards for the kept hand, to keep computation straightforward. However, for the thrown set of cards, there are $\binom{46}{2} = 1035$ possible combinations of cards which can be thrown by the opposing player into the crib as well as the 44 remaining possible cut cards. These cut cards must be considered separately in such a manner of evaluating $\binom{46}{2} \times 44 = 45540$ possibilities rather than simply looking at each of $\binom{46}{3} = 15180$ possibilities since the presence of the card in the crib or as the cut card can affect the score.[4] Just as crucially, there are small differences in scoring the crib

[4]For example, as per the right-jack (his nobs) scoring rule, a hand of 5♣ 5♡ 5♠ J♢ with a cut card of 5♢ is a perfect hand with a score of 29, but moving those cards around so that the jack is no longer in the hand and is instead the cut (i.e. hand of 5♣ 5♡ 5♠ 5♢ with a cut of J♢) yields a score of 28.

as opposed to scoring the player's own hand that mean that previous evaluations' results are not reusable. For instance, the rule for gaining points from a flush are more lenient for one's own hand than for the crib: the crib's flush must be a five-card flush containing the cut card whereas the player's own hand needs only be a four-card flush of their own hand with bonus points gained from the crib also matching. This means that, altogether, there are

$$\binom{52}{6}\binom{6}{4}\left(\binom{46}{2}\times 44\right)\approx 1.391\times 10^{13}$$

possible combinations of cards that need to be evaluated in total to fully understand the statistics of a cribbage game for every set of cards that can possibly be dealt.

Although the majority of this project was coded in Python for its ease of use and speed of development, due to the performance-crucial, basic mathematical operations involved, the overheads of using a higher-level language were deemed too critical and this particular tool was developed in C instead. To put the performance gains into perspective, at its fastest, most parallelly processed state, the Python database populator was estimated to take approximately four months to simply list and evaluate all possible scores on a development machine, disregarding the file I/O operations required to write those results to the database. The same functionality, with the addition of file I/O, in the C program would take a relatively mere fourteen days on that same machine. Furthermore, access to a high-performance server allowed for further parallelization which cut the final run time down to approximately five and a half days.

Careful consideration and preparation needed to be taken for the retrieval of this information, however. The trillions of combinations could not be quickly searched by card value as doing so would require searching the entirety of the database on each lookup, decreasing the performance so much as to be worse than simply enumerating the combinations on-demand. A rather simple solution to this problem was to search by index instead of by card comparison. These indices could not simply be stored in the memory of the running program because the sheer size required to store all of the indices at all times would rival that of the database, and its population would take considerable enough time. Thus, a quick and reliable method for creating and recreating these keys needed to be used. As the cards were represented internally as an integer between 0 and 51 (inclusive) and there were only 6 cards for indexing, the concatenation of the cards' digits in (keep,toss) order would create a number with at most twelve digits, with an absolute maximum value of 484950514647 for a combination of (K♠K♣K♡K♢, Q♡Q♢), well within the range of numbers addressable by a 64-bit integer.[5] This index could be created by the sum

$$index=\sum_{i=0}^{5} hand[i]\times 10^{2(5-i)}$$

[5]Thanks to implementation, the order of cards was guaranteed to be sorted within each tuple, so each combination of cards was tracked and not permutation.

where *hand* includes all cards in (keep,toss) order, and still be guaranteed unique.

While the scores for combinations of chosen and tossed cards could be evaluated before any games had actually been played, the hands' usefulness during the pegging portion of the round needed to be evaluated in semi real-time. A single pegging agent was programmed with a simple one-ahead greedy heuristic: the card that gained the most points when played next was selected. Ties were broken by choosing the highest-valued card that reached that score in order to potentially outmaneuver the opponent from reaching a count of 31. This agent was then played against a copy of itself with randomly allocated cards and the results of that round were recorded into a pegging records database, separate from the previous hands' statistics database, to provide an initial knowledge base. This records database could then be queried by the agent during the choosing phase of the game and contributed to at the end of the pegging phase as training progressed.

3.2 Weighting

For the purposes of this thesis, how well certain combinations of cards lend themselves to being played with different strategies is not directly explored. Instead, only the player's position in score-space affects the decision as to which strategies to play by. Put another way, the agent is not concerned with what cards it is dealt as much as where it is located on the board. Each possible score-space location can be thought of as a discrete coordinate defined by the parameters *PlayerScore* $\in [0, 120]$, *OpponentScore* $\in [0, 120]$, and *Dealer?* $\in \{0, 1\}$. At each score-space location is a vector $\overline{w}_{p,o,d} = [w_1, w_2, \ldots, w_m]$ where m is the number of all possible strategies to be considered. At the beginning of each round, each of m strategies is evaluated for all $n = 15$ possible combinations of cards kept to produce an $m \times n$ matrix $\mathbf{S}$ where $\mathbf{S}_{i,j}$ is the desirability of the j^{th} keep/toss combination according to the i^{th} strategy, further constrained by $0 \leq \mathbf{S}_{i,j} \leq 1 \ \forall i, j$. A value vector $\overline{p}$ of length n, representing the total perceived value of every possible keep/toss combination, can then be computed by $\overline{p} = \overline{w}\mathbf{S}$ wherein $\text{argmax}_x \, \overline{p}_x$ can be thought of as the most desired combination of cards overall and $\text{argmin}_x \, \overline{p}_x$ as the least. These collective desirability metrics are later used to determine which combination of cards to choose and which to toss. The described method of combination, visualized in Figure 1, was used because it allowed for multiple strategies' suggestions to be combined into a collective common decision, in a manner reminiscent of the author's own internal thought process while playing cribbage.

3.3 Training

After a complete game has been played, the winning and losing agents need to modify their weights in order to determine a correct strategy at that time coordinate. The agent attempted to learn which subset of strategies make the best decision in tandem at a given point. Therefore, there is no single strategy adhered to at any

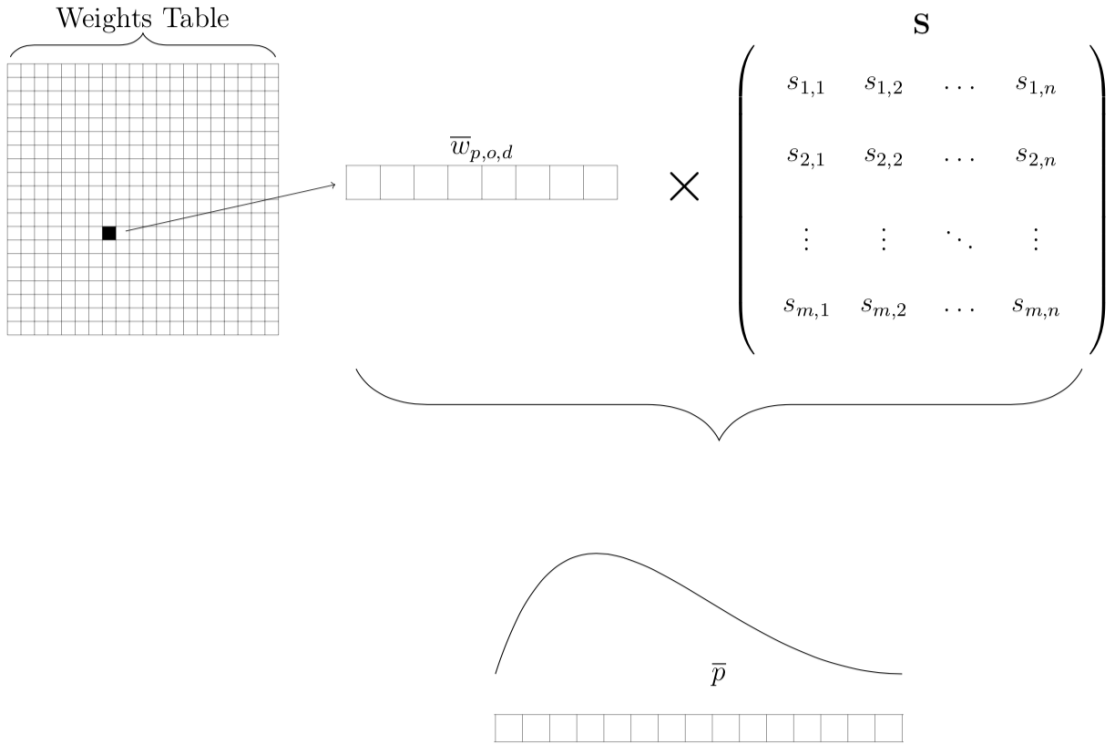


Figure 1: A visual depiction of the weighting operation.

given time. Instead, the strategies which advocated most for the chosen hand would be held responsible for the success or failure of the given hand and thought of as the action at the given step. These responsible strategies were determined as those whose values in $\mathbf{S}$ were the highest in its column. This subset of strategies, carefully chosen to include only the top contributors to avoid stagnant resulting weights, was then adjusted proportionally to itself according to the formula:

$$\overline{w}'_i = c\overline{w}_i$$

for all indices i which are in the responsible group, where c is a shared adjustment constant. All other weights would be left unaltered before the updated vector was re-normalized to the L^1-norm.

The resulting effect of this mechanism is to slightly reward or punish the weights corresponding to only the strategies which were most in favor of choosing the chosen cards. Note that this is not necessarily the same set of weights which were the highest in the weights vector. It may be the case that the weights for strategies x and y are the highest at a given score location, but that strategies u and v, with lower weight values, had higher advocacy for a given hand which summed up to a larger resulting desirability value. Were the aforementioned adjustment mechanism not used, and, instead, the highest weights were to be directly punished or rewarded, the resulting changes would merely reward or punish those strategies which were randomly allocated slightly higher weights initially.

Since locations earlier in the game have a higher number of potential resulting states

and are less likely to affect the final outcome of the game, they are modified less than their later counterparts. This is accomplished by decaying the adjustment amount for each step taken backward along the visited path of states in manner similar to TD(λ) [Tes95]:

$$c_j = C \cdot (1 - d)^{T-j}$$

where j is the index along the path starting at 1, C is a starting value of the adjustment constant, T is the final step index, and d is the rate of decay expressed as a ratio such that $0 \leq d \leq 1$. In order to allow for closer losses to be less severely punished than worse losses and vice versa for wins, the adjustment constant C, i.e. the reward, was defined as proportional to the difference between the players' scores according to the formula:

$$C = s \cdot (PlayerScore - OpponentScore)$$

where s is a scaling factor, analogous to a learning rate parameter. This scaling factor remains constant throughout training and does not itself decrease or decay. This is because all games are equally important, so a loss using strategy i during game x is just as important as it would be during game y.

The reinforcement training framework operated according to previously explained Monte Carlo methods. A file of starting weights was loaded to initialize an agent. Two agents were then placed into a game and played against each other. After the game, i.e. the episode, had been completed, the weights for each agent were adjusted accordingly along the path taken by the agent in order to improve its policy. After a set number of epochs, the agent saved its weights configuration to a checkpoint file. This allowed for the convenient benefit of being able to track how weights adjusted over the course of time.

To facilitate an adequate rate of exploration of the state space, exploring starts were implemented by using randomized score initializations during training games. A random score was chosen for each of the agents, keeping the spread of scores to within 60 points in order to limit the search space. This value was chosen since, with a maximum point total of 121, it is highly unlikely to get into a situation where one player is half the board behind the other. While there have been anecdotes of losses by more than that amount, it is a very rare occurrence and can be thought of as a failure of luck rather than of skill. As this is a matter of training skill and proficiency in the game of cribbage, situations of exceptional bad luck can be treated as outlier situations.

As a further measure to ensure adequate exploration, a modified version of an ϵ-greedy policy was implemented as well. Under normal conditions, cards were selected by choosing the combination which had the highest value in the produced $\overline{p}$ vector. At any given point in a training game, however, there was a chance of selecting which cards are played by random by using $\overline{p}$ as a probability distribution. This chance e of random choice was related to the variance of the weights according to the formula:

$$e = k - \mathrm{Var}(\overline{w}_{p,o,d})$$

where k is set at a constant 0.3, empirically chosen during the development process. This was implemented in order to ensure that situations in which there were more uniform weights had a higher chance of being explored, whereas those with a higher variance were deemed to be varied enough to have been previously trained, and thus should be further exploited.

The training was intended to take the form of a tournament of learning in which better and better agents would be paired off against each other until an ultimate agent was reached, allowing for more experience to be gained for agents from a variety of sources. After a set number of training games, one million in the case of this thesis, the two agents were played against each other in a tournament match fashion. The winner was determined as the agent with the most points at the end of a series of games, wherein two points would be awarded to the winner of a game or three if that win was by a margin of at least 31 points. Ties were broken by total point spread, according to ACC rules [Ame18]. After the match had completed, the winner advanced to the next round, training against another winner from the previous round. The process would be repeated until one agent is declared the ultimate winner.

Very quickly, the tournament structure was found to not provide any further benefits to the learning process. As a result, a set of additional experiments were performed to determine the limits of learning possible. While these will be covered in more detail in Section 4, these modifications included simple parameter adjustments, such as to the learning or decay rate, as well as more complicated alterations including using neighboring weights or punishing a loss less severely than a corresponding win is rewarded.

4 Findings

In this section, the results of the training rounds are presented. After the tournament was found to be reaching a plateau in performance, a series of experiments was performed in an attempt to improve learning. These experiments included parameter adjustments as well as alterations to the learning process itself.

4.1 Tournament

Round 1

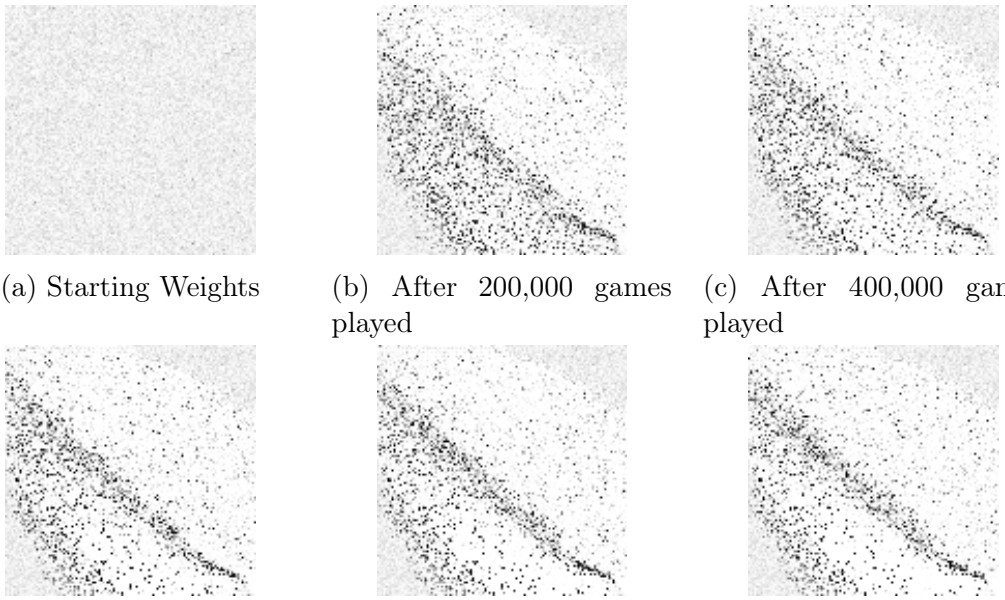
Round 1 consisted of 32 agents with randomly allocated starting weights paired off against each other. Each pair of agents played one million games against each other, each game starting at a random score location, learning and reinforcing their weight vectors after each game, according to Section 3.3.

Learning Process The progress of the first round's training on a sample agent can be seen in Figure 2. Each individual square within the image represents the strength of a single strategy, in this case `hand_max_avg`, where white means completely absent and black means completely dominant. From the origin at the top-left of each image, the agent's own score increases as one moves down along the y-axis and the opponent's score increases along the x-axis. Each image was taken at an intermediate stage to capture and show transitions.

There are two things to note from these images. The first, the stark contrast in colors in the majority of the image. The other, the area in which these stark contrasts are present.

In the starting phase, all weights are randomly assigned and relatively uniform with only slight variances, hence the blurry, dull, gray appearance. As time progresses, the image becomes crisper and filled with more contrast. This indicates not only stronger preference for the strategy at the given point, but an almost all-or-nothing attitude towards adhering to a single strategy. This means there is little to no nuance to which cards are chosen or chance for other strategies to collectively override the major strategy.

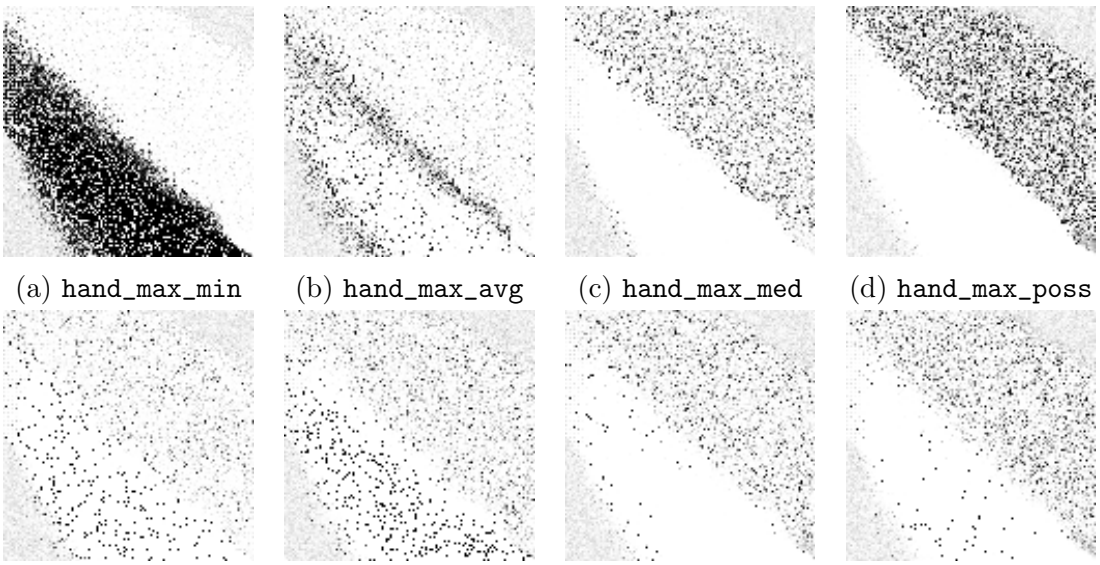
Also of note is where the previously mentioned stark contrast is present and, more precisely, where it is absent. Since only those states which have been visited can have their weights influenced, the remainder will continue to stay untouched. Note the top-right and bottom-left corners of the strategy graphs: regions in which one player has amassed a large lead. These states remain unvisited because, even with a potential spread of 60 points when initialized, these outlandish scores are outside the realm of potential visitation. Therefore, they have not been a part of any game, so they cannot have their weights adjusted.



(a) Starting Weights (b) After 200,000 games played (c) After 400,000 games played

(d) After 600,000 games played (e) After 800,000 games played (f) Final Weights

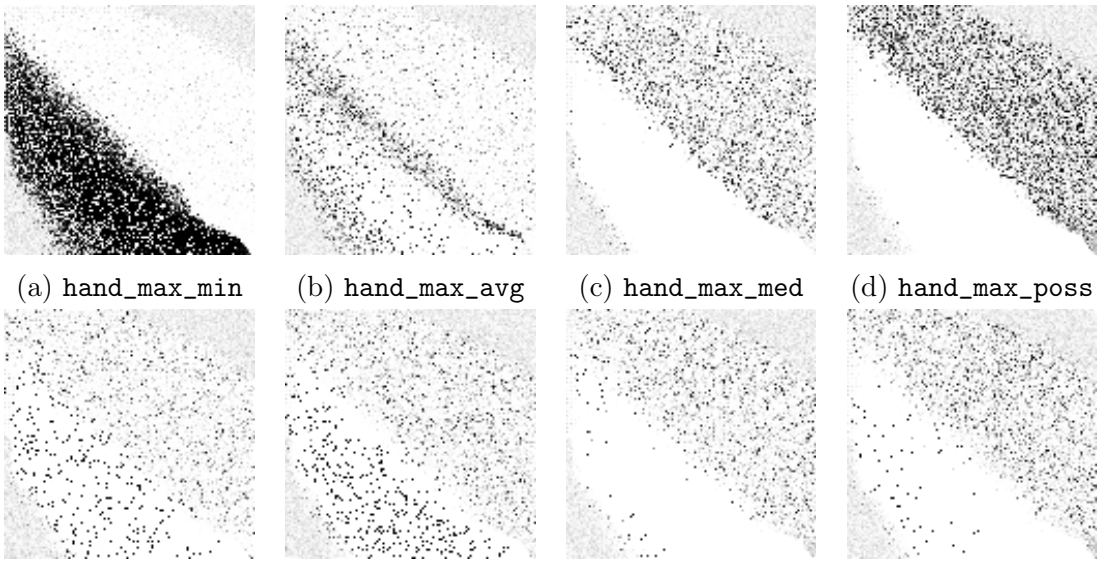
Figure 2: Training weights representation for an agent's `hand_max_avg` strategy when that agent is the pone over the course of the one million games of Round 1. In these images, the y-axis represents the player's own score, the x-axis the opponent's score, with the origin starting at the top-left of the image.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 3: All final strategy strengths for an agent when playing as the dealer after training for one million games during Round 1.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

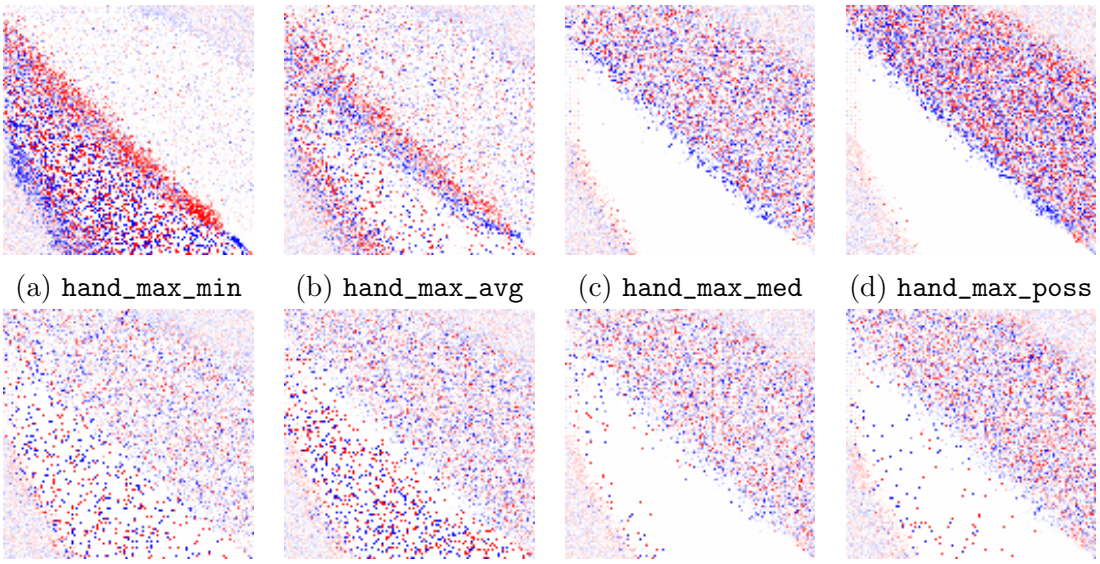
(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 4: All final strategy strengths for an agent when playing as the pone after training for one million games during Round 1.

Learning Results Despite the all-or-nothing nature of how a single strategy is potentially learned, it is still worth noting that the agent did, in fact, learn to play different strategies at different times. As can be seen in Figure 3, the strengths of each strategy's weight do vary across score-locations. For instance, when in the lead by a fair margin, the agent will prefer to choose the hand with the most guaranteed points in its own hand by following `hand_max_min`. However, when the game is either extremely close or when the agent is well in the lead, the agent will take a slight gamble and play for expected points. Occasionally, the agent will also attempt to pay attention to the points gained through the play phase of a round by playing a combination that pegs well. Ironically, the agent may play against its own best wishes by minimizing the average return of the crib. This is speculated to be a result of coincidental alignment of the results between `crib_min_avg` and more the reasonable `hand_max_min` or `hand_max_avg`.

Of further interest is how little the agent knows how to handle a losing position. As can be seen by looking in the upper-right half of each strategy's individual graph, there is little consensus or pattern as to which strategy should dominate in losing states. It is possible that agents which end up in these positions lose more often than they win. If this is the case, the resulting punishment will decrease the top two or three strategies that were most responsible for the hand choice at that state, effectively increasing all others. This, in turn, would lead to a cycle in which different strategies are cyclically placed in a role of strongest weight, generating the fuzz seen now.

By comparing the pone's strategy graphs (Figure 4) to those of the dealer's (Fig-



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 5: Difference graphs for each strategy. In these graphs, blue spaces represent those score locations in which the pone preferred to play by the given strategy while the dealer did not, and vice versa for red. White spaces show locations where the strategy's strength is mostly equivalent between the two roles.

ure 3), a few patterns emerge. At first glance, the graphs look highly similar and as if splitting up the search space into two merely increased the complexity of finding a solution without payoff. For instance, the majority of the winning positions favor the `hand_max_min` strategy with a "border" of `hand_max_avg` being used in positions where riskier decisions are needed or can be afforded. Additionally, the losing positions offer no definitive answers and seem to be a jumbled mess of suggestions, the only major suggestion being to play more riskily to regain the lead.

However, upon closer inspection, with a side-by-side comparison or using the difference graphs in Figure 5, there are subtle differences that can be seen. The most major of these differences is that all patterns are shifted lower and more to the left when the agent is the pone, correlating with scores which are more advantageous to the player. This means that the agent is more comfortable when it has a slightly larger lead as the pone than it necessarily needs as the dealer, most likely as a consequence of the additional points possibly accrued from the crib.

Perhaps the most interesting difference is the traceable shape of the weights during the final scores of a close game. Although difficult to discern in all but the `hand_max_min` and `hand_max_avg` strategies, the final weights trace a sinusoidal curve along the diagonal for approximately the last twenty points of the game. Additionally, these waves are opposing in nature. This presents the fascinating conclusion that the agent is learning that different states or checkpoints are preferred when playing as the dealer than when the pone. Not only is this fascinating since the agent is

learning a state-value function indirectly, but in the game of cribbage, there exist different checkpoints in which the player aims to get himself by the end of a round to be in favorable position for the next. Here the agent is seen learning these checkpoints through its play. These waves also show that the pone is more confident in its chances of winning in the final stages of a close game than the dealer is, as a consequence of the counting precedence. That confidence begins to transfer to the dealer, however, when the pone has too many points to score than can be expected from a single hand.

Performance It is clear from the strategy graphs that, while perhaps strategies are being over-trusted, basic game trends are clearly being learned by the agent. However, perhaps the only metric which matters from a learning perspective is the agent's performance. In this area, the trained agents failed miserably. The winning agent between the two learners was pitted against an agent using randomly allocated starting weights where both agents would only strictly follow the policy generated without exploration. As can be seen in Table 2, the trained agent lost easily to the randomly-weighted agent: with the exceptions of spectacular point differentials in Games 3 and 9, all other games were relatively close. This was speculated to be a result of the previously mentioned over-aggressive learning pattern and its all-or-nothing end result. Since the agent aligns so intensely to a single strategy, it is not able to overcome a local optimum and has essentially been overfit to the scenario. Another potential reason for the losses could be the lack of knowledge of how to handle losing situations. As previously discussed, in the event of a loss, most strategies will effectively be increased except those most responsible, contributing to a system of cycling weights. Furthermore, it is also likely that an agent which finds itself in a losing position does not end up recovering and winning the game, meaning that the agent effectively learns that it is mostly by chance that it will recover.

In order to more accurately diagnose if an overfitting situation was occurring, a winning agent was played against its own previous checkpoints. A sample set of scores can be found in Table 3. The point spreads from twenty separate 100-game tournaments are depicted in Figure 6a. While point spreads are from total points pegged to the board, not gained from wins, and therefore not a direct indicator of performance in tournament play, it is useful for determining general performance. Furthermore, as the rewards during the training phase, point spreads are direct indicators of learning progress. As can be seen, there is no discernible correlation or pattern between epoch checkpoint and performance. This indicates that, despite a policy being learned, quite a lot is still left to chance of the cards dealt. This has the potential to be exacerbated even further when policies are nearly deterministic and non-adaptive in their weight assignments. Additionally, a random agent was played against its future iterations in the same fashion, with similar results shown in Figure 6b.

Applications for Round 2 Because it was presumed that the over-aggressive learning of singular strategies was negatively impacting performance, some learning

Game	Random	Trained
1	104	121
2	121	114
3	75	121
4	121	118
5	111	121
6	121	99
7	121	87
8	110	121
9	121	64

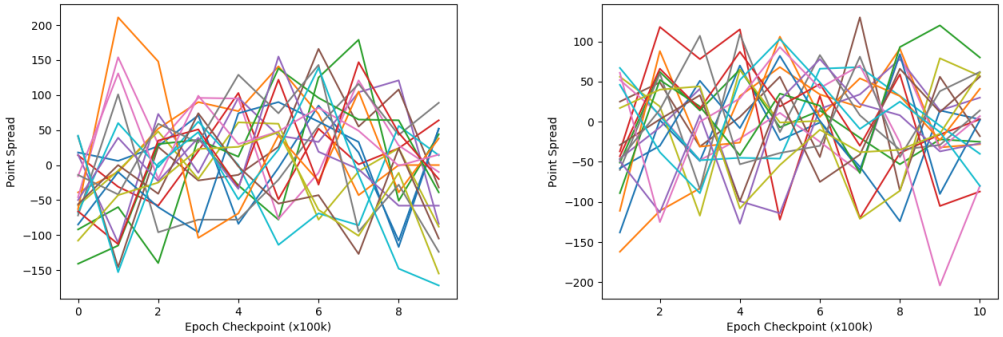
Agent	Score	Point Spread	Wins
Random	12	+39	5
Trained	9	—	4

Table 2: Results of a nine-game tournament played between a randomly-weighted agent and a trained agent after learning for one million games. The table on the left shows the final point amounts for each of the nine games. The table on the right shows the score of each player, as determined by the ACC scoring rules for match play; the total point spread for the top player, omitted for the second player for readability as it would just be the negative of the first player's; and the number of times each player won a game.

Epochs	Score	Spread	Wins
0	126	+523	59
1MM	86	—	41
Epochs	**Score**	**Spread**	**Wins**
200K	108	-27	47
1MM	118	—	53
Epochs	**Score**	**Spread**	**Wins**
400K	87	-510	41
1MM	133	—	59
Epochs	**Score**	**Spread**	**Wins**
600K	117	370	52
1MM	98	—	48
Epochs	**Score**	**Spread**	**Wins**
800K	124	239	56
1MM	96	—	44

Epochs	Score	Spread	Wins
100K	122	+200	55
1MM	104	—	45
Epochs	**Score**	**Spread**	**Wins**
300K	105	-180	48
1MM	116	—	52
Epochs	**Score**	**Spread**	**Wins**
500K	97	-386	43
1MM	129	—	57
Epochs	**Score**	**Spread**	**Wins**
700K	107	-174	48
1MM	115	—	52
Epochs	**Score**	**Spread**	**Wins**
900K	111	-82	51
1MM	111	—	49

Table 3: Results of multiple 100-game tournaments played between the fully trained agent and its various checkpoints across training during Round 1. **Score** indicates the number of points scored according to ACC rules for match play; **Spread** is the cumulative point spread for the first player, omitted for the opponent for readability; and **Wins** is the total number of games each player won in the match.



(a) An agent plays against previous iterations of itself.

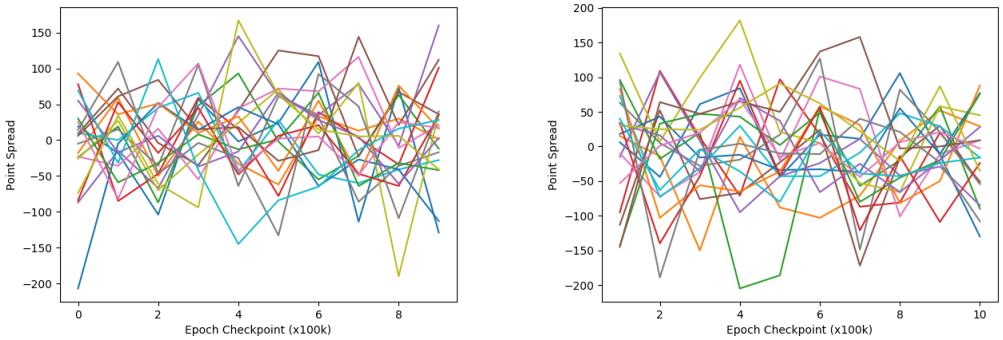
(b) A random agent plays against later, more learned agents.

Figure 6: Point spreads across twenty 100-game tournaments pitting a winning agent against its checkpoints. Here, a positive point spread indicates that the fully-trained agent has accumulated more points than its opponent, an agent created from a checkpoint generated after the number of training game epochs indicated on the x-axis.

parameters were altered for Round 2. Since it was estimated to be the primary reason for the learning behavior, the first parameter adjustment made was to decrease the scaling factor drastically to one fifth of what was used in Round 1. The other major parameter alteration was to make the exploration rate a constant and not dependent upon the variance of the weights. During Round 1, the exploration rate was defined as $e = 0.3 - \mathrm{Var}(\overline{w}_{p,o,d})$ in order to allow more trained states to be less affected by randomized exploration. This had the understandable but unintended effect of further exacerbating the all-or-nothing behavior by more quickly exploiting a single strategy. Although only producing a decrease from a 30% chance to 18%, this would still result in a smaller likelihood of exploring in the current state, potentially further cementing of the dominant strategy in its position.

In order to see if the heavily biased weights could be tempered down to more reasonable mixes which could outplay a random agent, the structure of the tournament was updated. In addition to having the winners of the previous pair of agents square off against one another in a "winners' bracket," there was also a "losers' bracket" created in which the losers would start over from the beginning. These two ways of playing were intended as a two-pronged approach to see if multiple agents could be trained at the same time which could outperform random. The winners' bracket would determine if a highly-biased set of agents could learn nuance while the losers' bracket would act as a reset with a different set of learning parameters.

Since the agents were assumed to be learning how to outmaneuver each other, but general game performance was not increasing, the next phase would attempt to determine the extent of this tailored learning. In addition to the previously mentioned alterations to the tournament structure, a new batch of agents would be trained against a static, randomly weighted agent. Rather than both agents in the game



(a) An agent plays against previous iterations of itself.

(b) A random agent plays against later, more learned agents.

Figure 7: Point spreads across twenty 9-game tournaments for an agent in the winners' bracket of Round 2. Note that since the winners' bracket uses an agent with prior training, the total epochs elapsed is one million more than displayed.

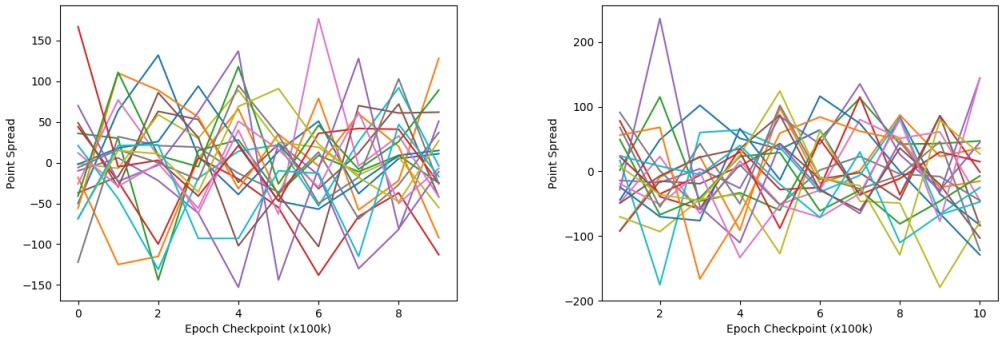
learning and altering their weights after each game, only one agent would train its weights. The prevailing logic behind this decision was that if each agent was indirectly affecting the other, the environment could be said to be slightly altered. This would, in turn, mean that the agents are no longer learning the original problem.

Round 2

Round 2 consisted of ten agents paired off against each other in both the winners' and losers' brackets. The winners' bracket used agents previously trained from Round 1 while the losers' bracket agents started with random initializations. These agents were each trained for a million games with mostly the same configuration as in Round 1, except the scaling factor was lowered to $s = 2.0$ and exploration rate was set to a constant $e = 0.30$.

Performance Judging by the lack of pattern found in Figures 7 and 8, there is no significant performance increase to be found from Round 2. If learning had occurred, Figures 7a and 8a would take on the form of a decreasing curve, gradually approaching zero, while Figures 7b and 8b would be their translation across the x-axis. On its own, Figure 7 would indicate that an agent trained for one million games plays on par with one trained for two million games. Alone, this would indicate that performance had reached a peak and saturated. However, the similarity to Figure 8 shows that neither bracket learned to perform better than previous iterations.

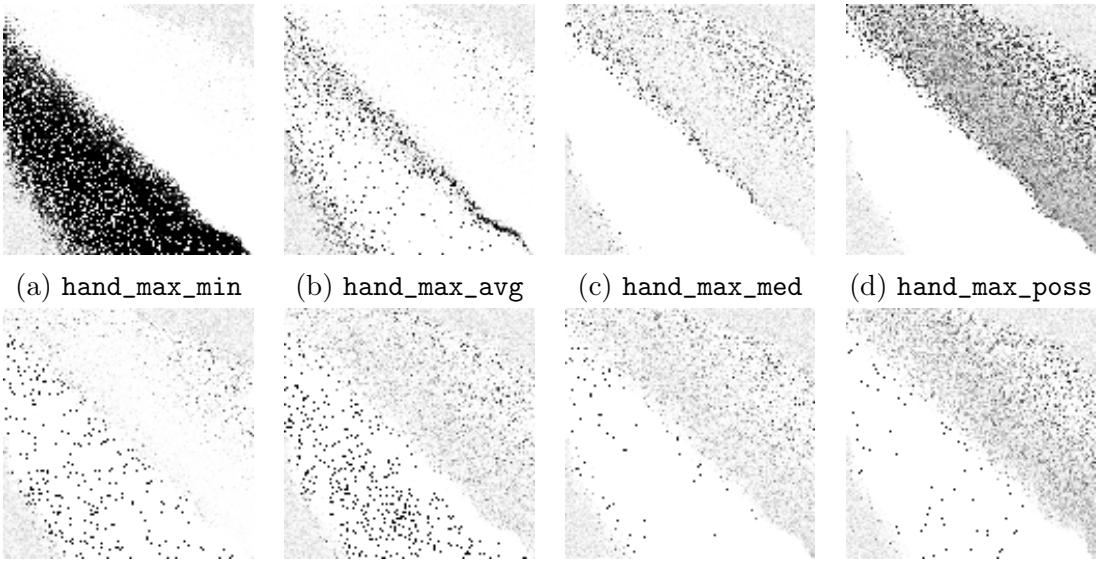
Learning Process and Results Despite the lack of performance increase after another million games played by both the winners' and losers' brackets, there are still interesting trends to be spotted between the different brackets of play. The



(a) An agent plays against previous iterations of itself.

(b) A random agent plays against later, more learned agents.

Figure 8: Point spreads across twenty 9-game tournamens of an agent in the losers' bracket of Round 2.



(a) `hand_max_min`

(b) `hand_max_avg`

(c) `hand_max_med`

(d) `hand_max_poss`

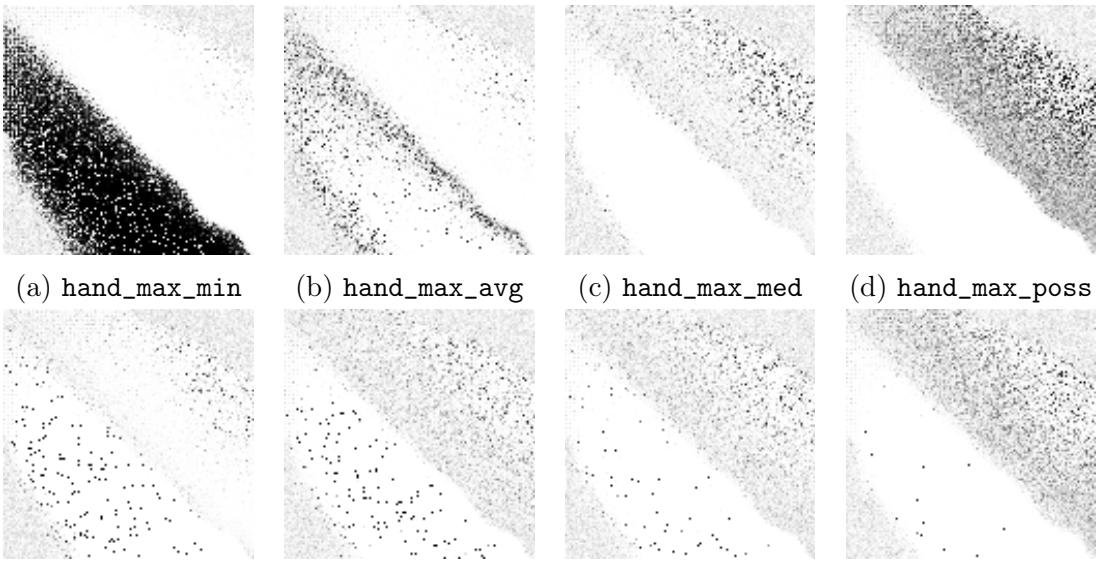
(e) `crib_min_avg`

(f) `pegging_max_avg_gained`

(g) `pegging_max_med_gained`

(h) `pegging_min_avg_given`

Figure 9: All final strategy strengths for an agent in the winners' bracket when playing as the pone after training for one million games during Round 1 and a further one million games during Round 2.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

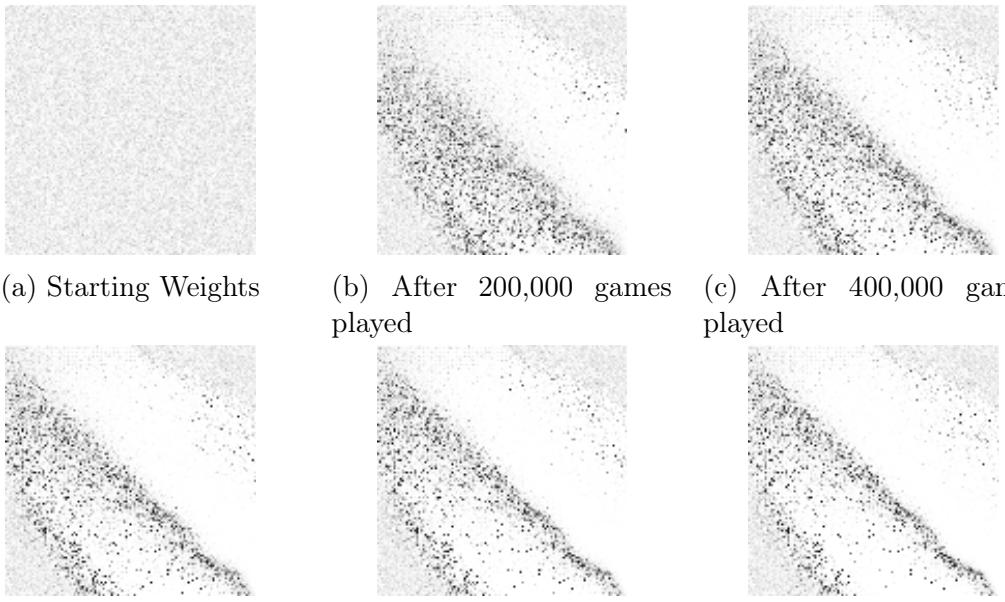
(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 10: All final strategy strengths for an agent in the losers' bracket when playing as the pone after training for one million games during Round 2.

most notable is how quickly policies converge to a similar state. While the strategy graphs for the less trained agents are less crisp in their appearance thanks to their slower learning rates, the patterns of which policy to mainly follow at which times are still quick to form.

Even more fascinating observations can be found from the strategy graphs from the winners' bracket (see Figure 9). By focusing on the `hand_max_min` and `hand_max_avg` strategies in particular, the sinusoidal wave along the diagonal can be observed extending further back along the diagonal to earlier game positions, albeit with smaller amplitude. While not being of much use to the agent directly, this is a useful observation from a cribbage player's perspective. Since it is possible to infer the state-value function, the implications of this wave are that earlier states are not crucial predictors for future success. Not only that, but the agent has learned that changes in lead are likely in later game states and not necessarily detrimental to the ability to win the game.

By comparing the winners' bracket strategy graphs (Figure 9) to those of the losers' bracket (Figure 10), a vast degree of similarity can be found. In both brackets, the same behavioral trends are learned. However, there does exist a small amount of difference between the two in how quickly and surely each trend is learned. The winners' bracket mostly further strengthens its current weight choices with the losing positions showing only a minor blending of strategies. Meanwhile, in the losers' bracket, the blending can be seen applied slightly more evenly to all strategies and across the winning-losing boundary. Of additional note, there are fewer spaces in which strategies such as `crib_min_avg`, `pegging_max_avg_gained`, and `pegging_`



(a) Starting Weights (b) After 200,000 games played (c) After 400,000 games played

(d) After 600,000 games played (e) After 800,000 games played (f) Final Weights

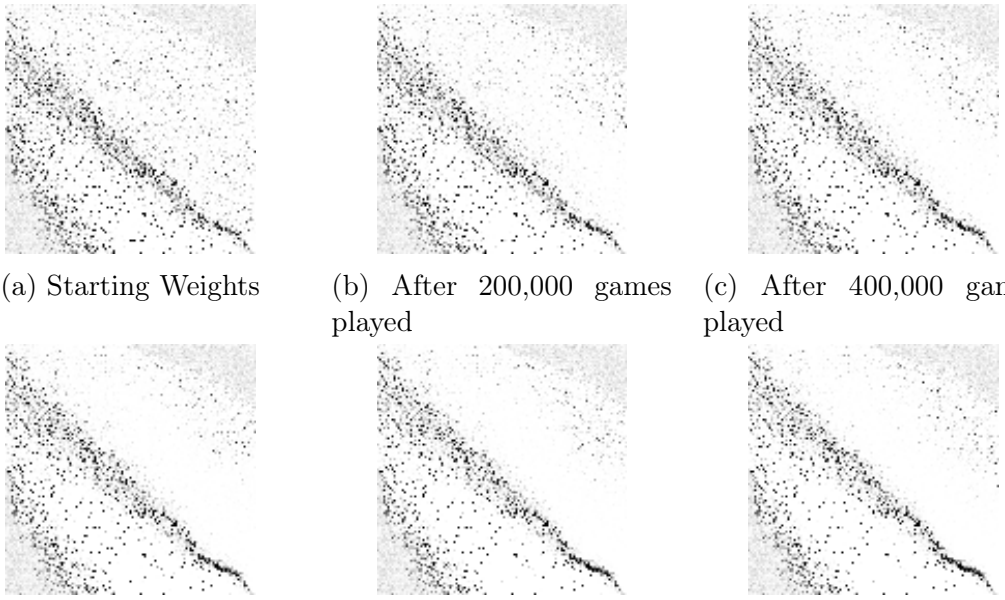
Figure 11: Training weights representation for a losers' bracket agent's `hand_max_avg` strategy when the agent is the pone over the course of the one million games of Round 2.

`min_avg_given` have been strengthened within the `hand_max_min` block. This is a good indicator that, although present, there may be less of a bias towards those strategies which are initially winners.

In analyzing the evolution of the `hand_max_avg` strategy in both the winners' (Figure 12) and losers' brackets (Figure 11), it can clearly be seen that the general trend of behavior forms relatively quickly, i.e. after approximately 500,000 games played. Beyond that amount of games, the agent learns to refine the strategy boundaries, but it can be seen that there is little further discovery being made.

The pairing of a learning agent with a non-learning, randomly weighted agent provided no additional benefit. The same behavioral trends seen in the losers' bracket are echoed in Figure 13. The lack of clarity found in the images is the result of fewer games being played because of batch job scheduling issues. From these images, it becomes clear that the learning mechanism, as is, is not conducive to playing a better game of cribbage. Thus there needs to be some form of adjustment made in the framework.

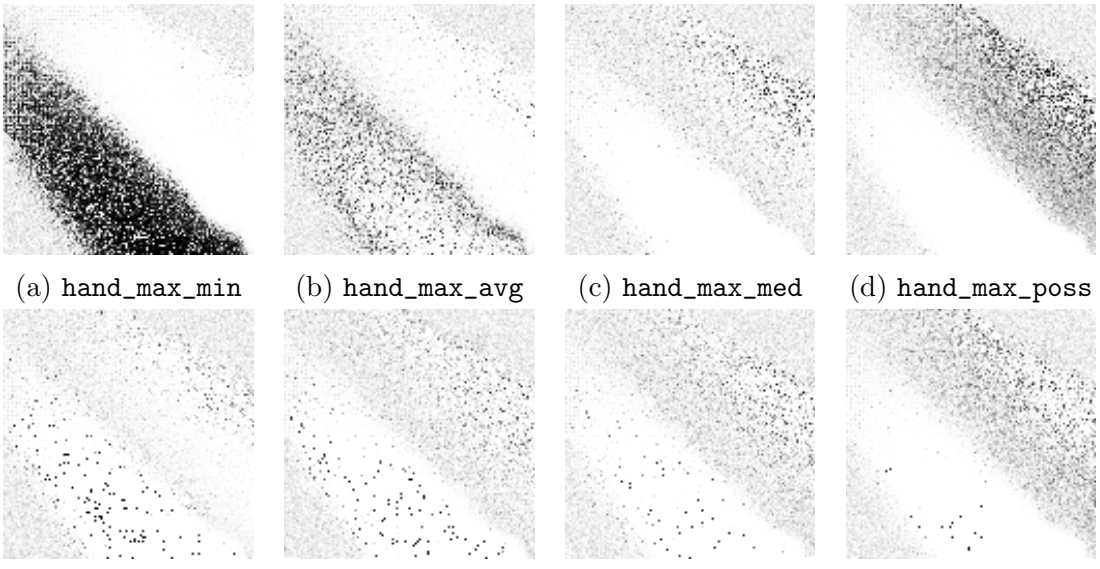
Potential Issues Before modifications can be made to the learning framework, the comparable performance to random weights must first be addressed as a potential error in implementation rather than in training. Since all methods of training so far have resulted in very similar policies being learned, it is fair to say that policy learned during training has been superior to random in some manner. There are



(a) Starting Weights (b) After 200,000 games played (c) After 400,000 games played

(d) After 600,000 games played (e) After 800,000 games played (f) Final Weights

Figure 12: Training weights representation for a winners' bracket agent's `hand_max_avg` strategy when the agent is the pone over the course of the one million games of Round 2. Note that the starting weights are carried over from Round 1, so the total training epochs to reach each position is actually one million higher than expressed.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 13: All final strategy strengths for a learning agent after playing a completely random agent when playing as the pone after training for 350,000 games during Round 2.

then a couple items to consider with regard to why such poor results are occurring during the review tournaments, such as the scale of play and potential overfitting of the model.

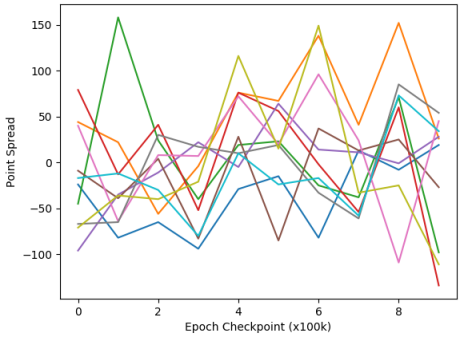
Scale of Play The prevailing theory as to the reason for which the agent learns a policy, but following that policy does not yield positive results in performance, is the issue of scale. The agent is trained on a million games, but tested on only a handful. Since the cards being dealt are not taken into account by the policy, the decisions made will likely be inaccurate on the scale of a single game, but not for thousands.

A regulation cribbage match between two human players consists of nine games. As can be seen in Figure 14b, in a series of 100-game matches between a fully trained agent and its previous checkpoints—using a Round 2-trained agent in the losers' bracket for demonstrative purposes—no pattern is discernible in total point spreads between the two playing agents. This demonstrates that, on this scale, the winner of the game is no more predictable than a truly random coin toss.

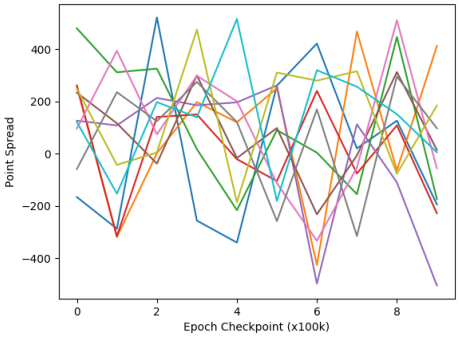
When the scale is increased to one thousand games, a slight pattern begins to emerge when visualized in Figure 14c. Whereas the majority of the graph remains highly varying and unpatterned, the first few games begin to trace a common curve. The match against the random agent is still unpredictable, but the matches against the 100,000 and 200,000 game trained checkpoints are consistently beaten by the final agent.

Although fewer matches are played to compensate for the additional time needed to play the increased number of games, the same pattern beginning to form in the thousand-game scale is visible in the ten thousand-game matches. With the exception of the purely untrained agent, the least learned agents perform the poorest against the final learned agent. Play is approximately evenly matched between the fully trained agent and its checkpoints which have been trained for 300,000 to 700,000 games. Following these matches, the agent begins to, again, win more consistently when more training is done. This confusingly indicates that more recent agents are getting worse than earlier iterations, yet the final agent is better than these. The reasons for this trend are unknown.

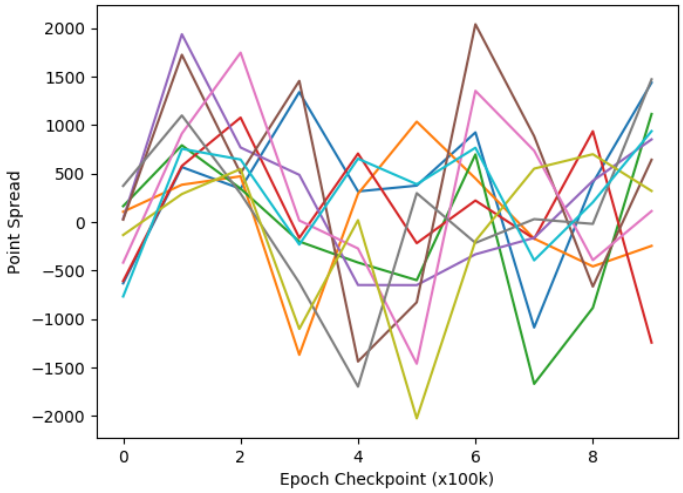
Overfitting If the agent is being trained correctly and no overfitting is occurring, then the point spread should be a positive number, gradually decreasing and approaching zero as more training is applied to its opponent, forming a similar curve to a loss metric used in classical machine learning. In the event of overfitting, the curve would dip below the zero before reapproaching zero. Neither of these shapes were seen in the resulting graph (Figure 14d). Instead, the shape of the curve implies that, since the random agent performs consistently better than the trained agent, the learning process is not learning the game so much as how to outplay or edge out its opponent in some experienced scenarios that do not generalize to the testing phase. This conclusion is supported by the observation that the agents with limited



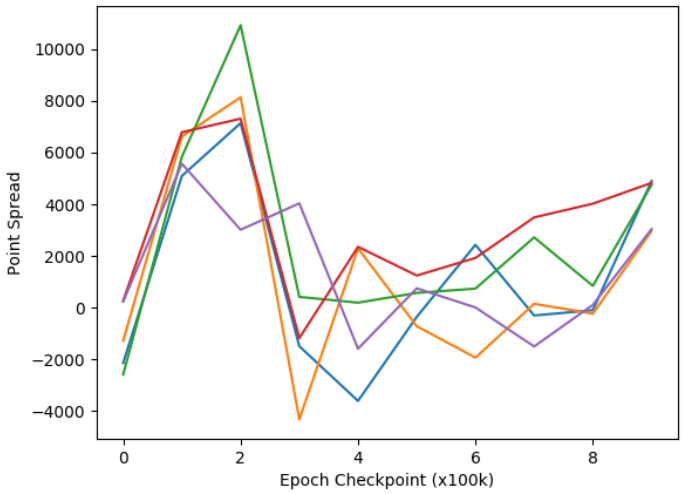
(a) Point spreads for 9-game matches



(b) Point spreads for 100-game matches.



(c) Point spreads for 1,000-game matches.



(d) Point spreads for 10,000-game matches.

Figure 14: Point spreads across matches of varying lengths. In each graph, the final trained agent is played against its previous checkpoint iterations.

training are steadfastly outplayed.

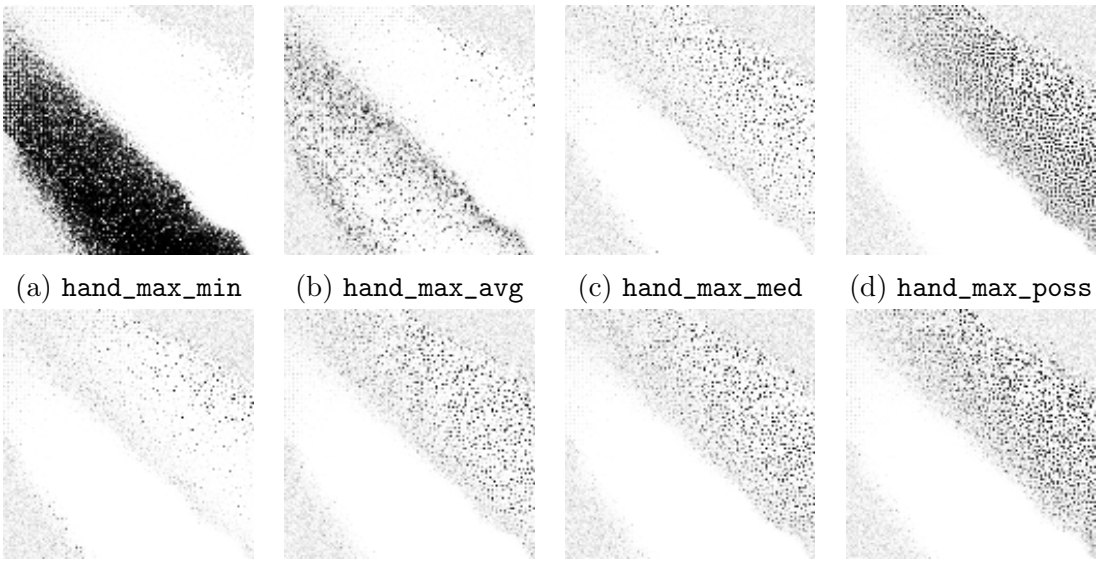
Also of further note, is the scale on the aforementioned graphs. In Figure 14d, the maximum point spread achieved is just a shade above 10,000 points over the course of 10,000 games. This means that the average point spread advantage is approximately one point per game at peak performance—and most often 0.2 points per game—in long-term play. The average spread increases to about 30 points per game when fewer games are played, but since performance is unpredictable on this scale, this can be explained as a result of the randomness of the cards given and cannot be considered a reliable measure of performance. In addition to the randomness of cards dealt, this massive point sway could also be the result of the agents' uncertainty on how to recover from a losing position. While the reasons for this inability cannot be said to be more than speculation, the learning of a policy directly, without taking into account cards dealt, is the likely culprit, as explained previously.

4.2 Further Experiments

As the results of Round 2 imitated those of Round 1, the tournament structure was abandoned and additional modifications were made to the learning process. These methods focused on directly affecting the weights applied to each decision made, both during learning and at choosing time. The intended goal of these modifications was to modify the learning process such that the resulting learning system might potentially be capable of developing a policy which plays the game consistently better than random. All modifications for these experiments started with the following common parameters:

- Scaling Factor: $s = 2.0$
- Decay: $d = 0.1$
- Random Exploration Chance: $e = 0.3$

Additionally, the pegging records database was standardized across all experiments as one arbitrarily selected from those resulting from the first round of training as it allowed for comparison of outputs the experiments to those of Round 2. As a final note, the mechanism for choosing a combination of cards using the $\overline{p}$ vector during the tournament was altered slightly. When making an exploration step, instead of using $\overline{p}$ as a probability distribution, as described in Section 3.2, the choice was made at uniform random, in a manner true to ϵ-greedy exploration [SB]. This was done to bring the implementation more into line with textbook definitions of the algorithm since the author's naïveté caused earlier implementations to deviate by introducing unnecessary complications. Furthermore, this difference in choosing mechanisms during exploratory steps made no demonstrable difference in the learning behavior, most likely as a result of $\overline{p}$ being similar to uniformly random for untrained states and, in essence, a choice between three or four most desirable choices when states had been visited.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 15: Final strategy graphs for an agent when playing as the pone after being trained for 500,000 games by following a policy which uses a weighted sum of neighboring weights.

Neighboring Weights

In order to smooth out the strategy graphs and prevent isolated states of separate weights, a blending of neighboring weights was developed. Rather than simply take the set of weights allocated to a single score location, the agent instead takes a weighted average of all surrounding weight vectors with its own location. In other words,

$$\overline{w}'_{p,o,d} = X\overline{w}_{p,o,d} + Y\, avg\left(\left\{ \overline{w}_{i,j,d} \;\middle|\; \begin{array}{l} i \in [p-1, p+1] \\ j \in [o-1, o+1] \\ o-1, o+1, p-1, p+1 \in [0, 120] \\ (i \neq p) \vee (j \neq o) \end{array} \right\}\right)$$

where X, Y are ratios of each vector's effect, $X + Y = 1$. The desired effect was to allow a score location to learn from its neighbors so that an area effect was present in the decision.

Results The agent was not able to develop blended strategy graphs that gradually shifted responsibility between strategies using the neighborhood technique. In fact, the `hand_max_avg` strategy, which one would most expect to meld with the `hand_max_min` strategy, was perhaps the most negatively affected. As seen in Figure 15, the `hand_max_avg` strategy graph has become more sparse in the winning states (lower left) with very little gray area surrounding the black. However, the presence of gray indicates that a melding of strategies did occur.

Interestingly, while the neighboring weights did not solidify any presence in the graphs, the new training method did make progress in negative space. This is to say that the weighted neighbors learning method eliminated the usually present islands of a single strategy within a swathe of another dominant strategy. A large amount of white space, indicating the absence of a strategy, was present in either the winning or losing states, depending on the strategy. This finding leads to the conclusion that weighted learning techniques do indeed allow for a sharing of knowledge between like states, but only insofar as to know what not to do.

Furthermore, there is still a vast degree of uncertainty present in the losing states of the strategy graphs. The theory that the certainty of the winning states might potentially bleed over into the losing side was not demonstrated over the course of the first half million training games. However, since the elimination of islands shows an improvement in preventing lucky happenstance to dictate a space's future, the neighboring weights training method has shown usefulness in training a cribbage agent in which states neighboring states are often similar in nature.

Regularization

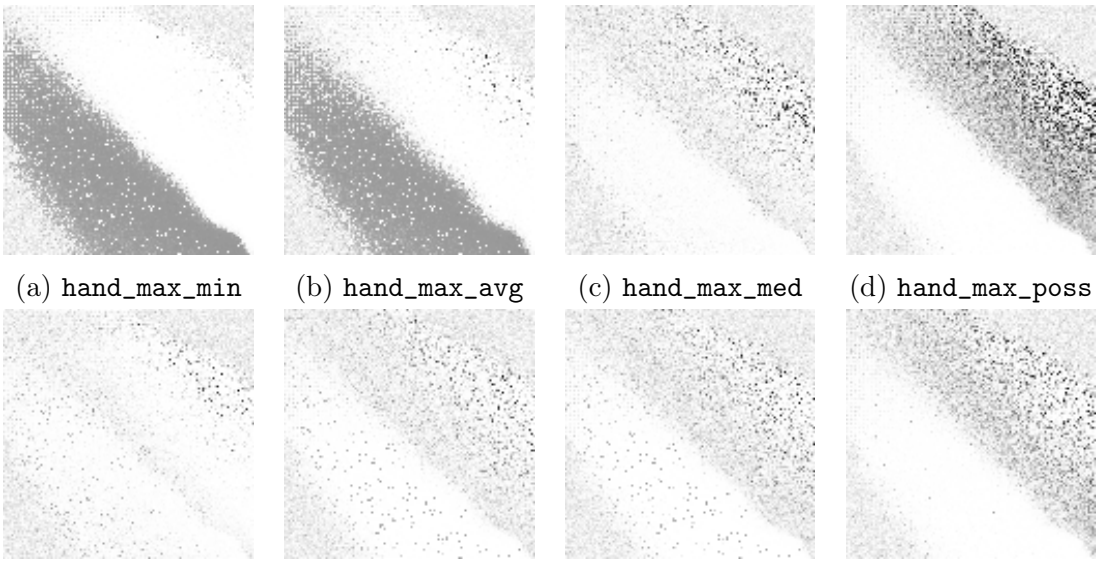
Since all previous attempts at developing a policy led to states that adhered strongly to an individual strategy, the idea that performance would increase under a policy consisting of mixed strategies was tested by regularizing the weights vector. In order to prevent a single strategy's weight from being so strongly preferred that a second strategy could not hope to possibly gain ground, a hard limit was placed on the pre-normalized update value. Expressed mathematically,

$$\overline{w}'_{p,o,d}[i] = \max\{r, c\overline{w}_{p,o,d}[i]\}$$

where r, the regulation rate, is some constant value throughout the training. While the value of $\overline{w}_{p,o,d}[i]$ could exceed r after re-normalization for a particularly strongly weighted strategy, the value could be seen converging to r within just a handful of iterations.

Results The limitation of maximum attainable value did indeed force the agent to learn multiple applicable strategies for each score location. As can be seen in Figure 16, the `hand_max_min` and `hand_max_avg` strategies are now more-or-less equally represented across the winning states. This is demonstrated by the monotone gray seen in these locations. Rather than each strategy specializing to its own dominant location, the territory is shared between the two most applicable strategies. Therefore, the regularization does indeed prevent the dominance of a single strategy unintentionally discarding all other potential strategies.

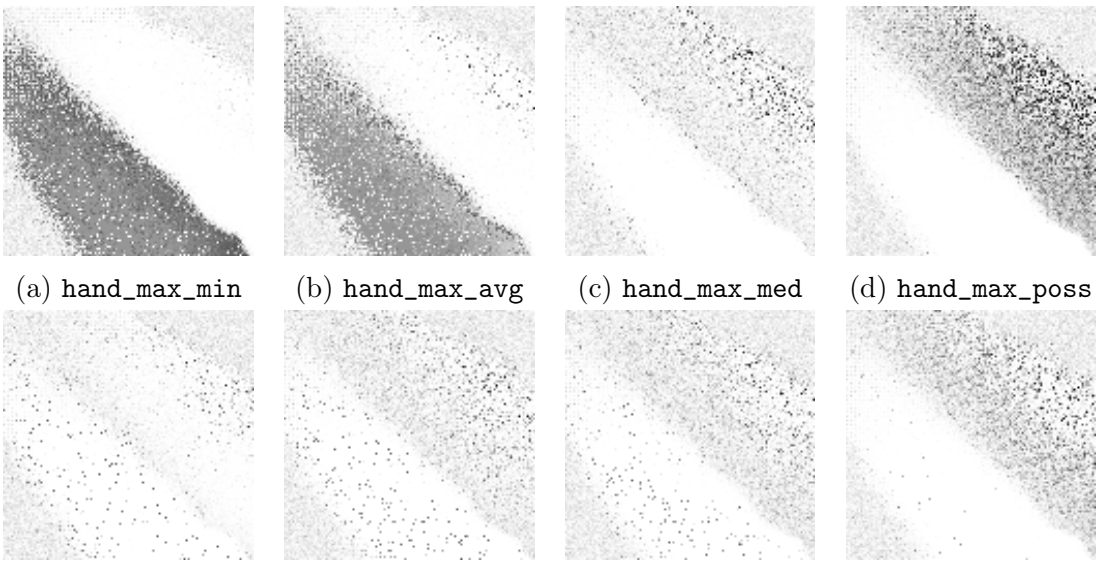
Additionally, the nature of this gray mass alters significantly as the regularization rate is altered. With a lower allowed maximum value, the strategy graphs take on the previously mentioned slightly amorphous, gray blob shape. As the maximum value approaches one, the gray shape begins to specialize more and begins to



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 16: Final strategies for an agent using regularized learning when playing as the pone and the maximum value weight allowed is 0.50 after training for 500,000 games.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 17: Final strategies for an agent using regularized learning when playing as the pone and the maximum value weight allowed is 0.80 after training for 500,000 games.

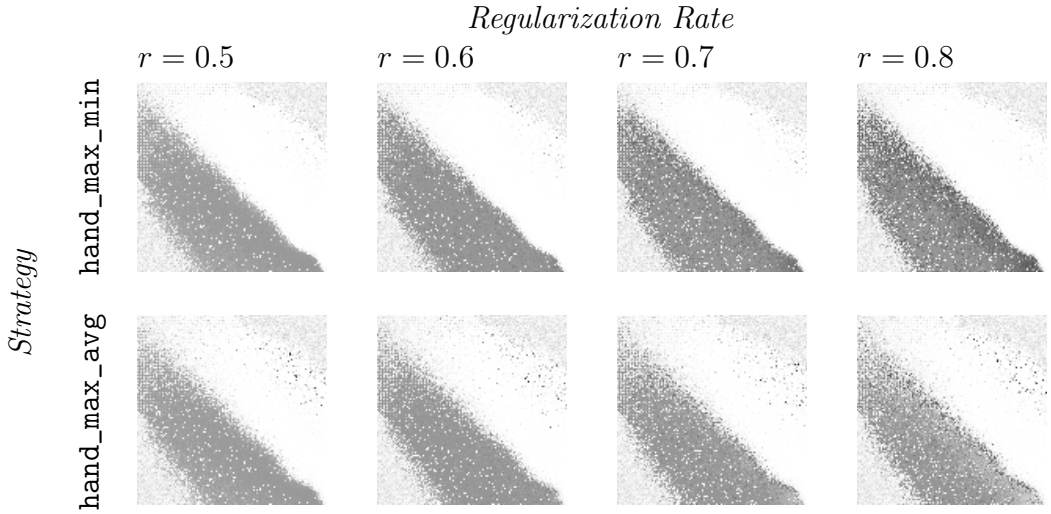


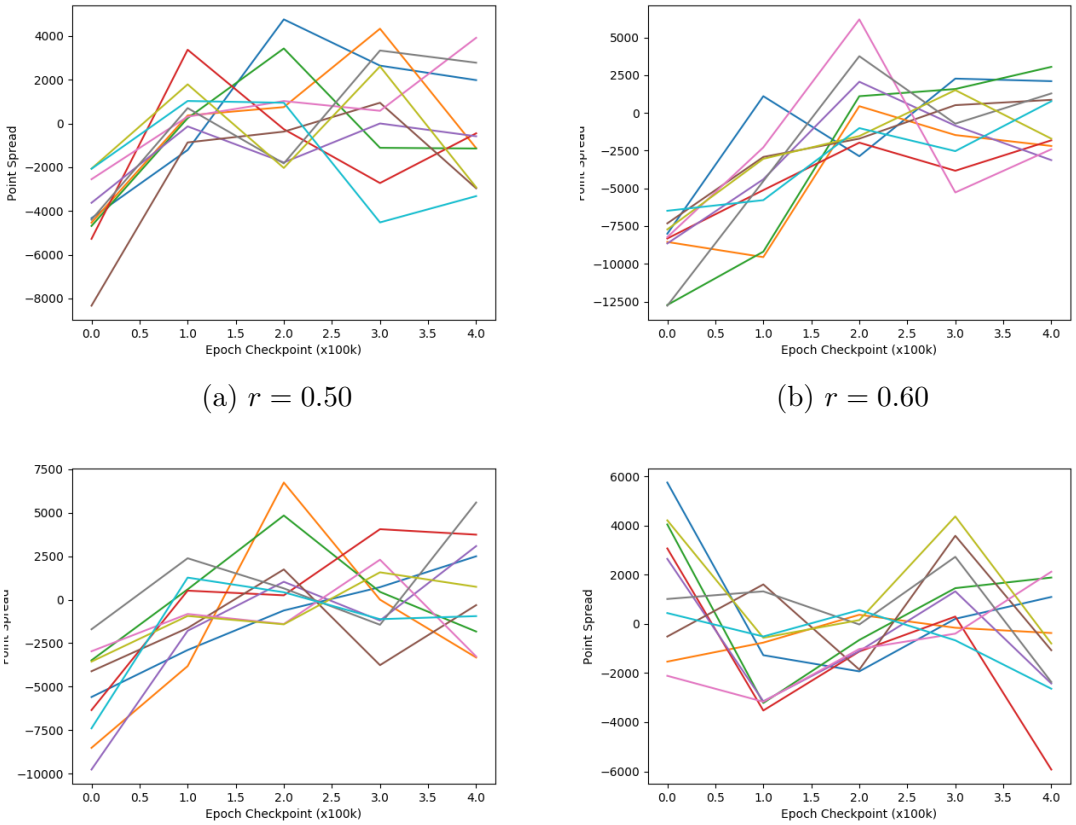
Figure 18: Comparison of the `hand_max_min` and `hand_max_avg` strategies for each of the regularization rates (r) used when the agent is playing as the pone.

show resemblance to the final strategy graphs obtained without regularization. A visualization of this process can be found in Figure 18.

Furthermore, as can be seen in Figure 19, only a regularization rate of $r = 0.80$ (Figure 19d) can be said to show an increase in performance as training progressed. With rates of $r = 0.50$ (Figure 19a), $r = 0.60$ (Figure 19b), and $r = 0.70$ (Figure 19c), the tournament point spread curves show that a trained agent plays on par with its recent checkpoints, but consistently worse than random weights. The endpoints of the point spread curves using $r = 0.80$ show a decrease for the trained agent from frequently better play against the random agent to more-or-less on-par performance with its later checkpoints, indicative of an increase in performance as training progresses. However, the increases and decreases of performance for intermediate checkpoints and the wide range of point spreads present prevent the conclusion that performance is progressively increasing.

Punishment Severity

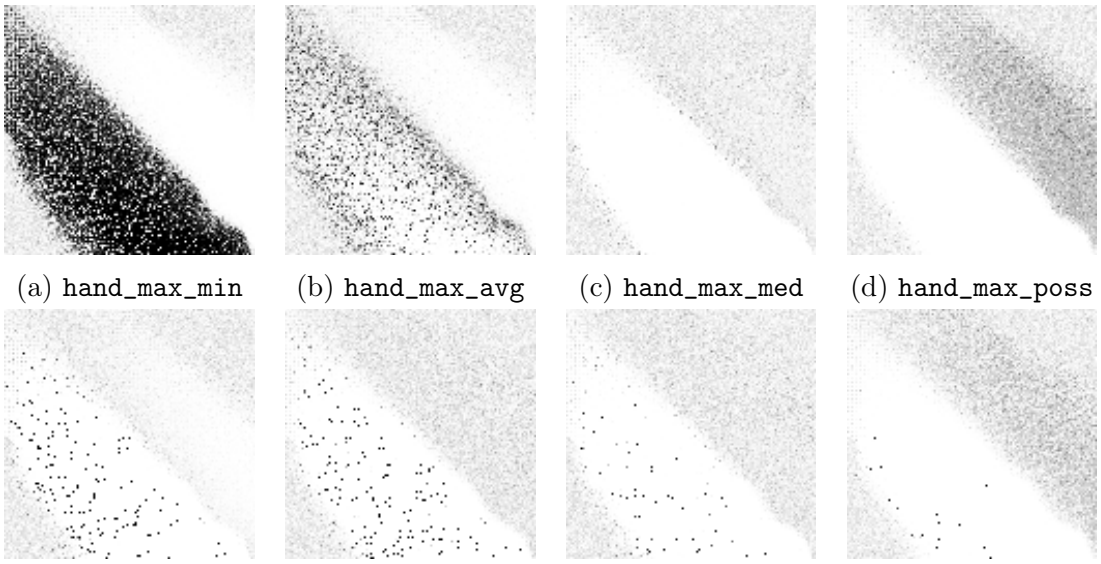
Under the assumption that a player in a losing position early in the game tended to never recover and thus lost as a result of unfortunate positioning, it followed that punishing losing states for something beyond the agent's control was potentially unfair. Furthermore, it was postulated that since the punishment mechanism, in effect, cycled strategies and that there was a possibility that an occasionally winning strategy was often outweighed by the tendency to lose, a less strict method of punishment could be used to ensure that the occasional win from a losing position remains visible. As such, the update step for modifying the weights was adjusted slightly so that the constant adjustment factor was significantly smaller for losing games than it was for winning games. Instead of using the adjustment constant of $C = s \cdot (\mathit{PlayerScore} - \mathit{OpponentScore})$ for both winning and losing agents, the losing



(a) $r = 0.50$ (b) $r = 0.60$

(c) $r = 0.70$ (d) $r = 0.80$

Figure 19: Point spreads of self-tournaments for 10,000 games carried out between agents trained using the regulation method mentioned in Section 4.1 after being trained for 500,000 games. Each tournament plot in this figure is referenced by the regularization rate r in use during training.



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 20: Final strategy graphs for an agent which has less severe punishment after training for one million games.

agent's adjustment factor was defined as $C = \frac{1}{4}s \cdot (\mathit{PlayerScore} - \mathit{OpponentScore})$. The reduction to $\frac{1}{4}$ was made arbitrarily for illustrative purposes.

Results The introduction of an amount of forgiveness did not lead to any worthwhile difference in learned policy. In contrast to the goal of allowing an occasionally good strategy to form in losing positions, those positions are even less sure as to which strategy to take, as seen in Figure 20. Therefore, it may be concluded that the increased likelihood of losing is likely not unfairly punishing potentially good recovery policies. It can be argued that $\frac{1}{4}$ was not a proper ratio to compensate for the likelihood of loss, but the similarity of patterns learned and decreased certainty indicate that the punishment mechanism is functioning adequately. Similarly, while reducing the amplitude of changes does lead to a more evenly mixed losing policy, this is effectively the result of a learning rate adjustment. As will be shown in the learning rate experiments, these forms of adjustments do not lead to differences in learned behaviors.

Policy Initialization

Since a desired secondary outcome of the learning process was to be able to use the generated strategy graphs to tell how a hand should be played in a certain score position, a comparison was made between the produced agent and pure strategies on a database of choices made by humans. The website Daily Cribbage Hand prompts its users with a set of dealt cards, their score, their opponent's score, and an indication

of whether or not the player is the dealer for the round [DCH]. With this prompt, the user then decides which cards they would keep in the given scenario.

With access to recorded answers from this website, the agent's choices could be compared to how humans ranked the choice. For each of the approximately 3600 usable records, the choice the agent made was compared against those made by the users of the website.

The results of this comparison, seen in Table 4, show that the agent trained in Round 2's losers' bracket chooses the same set of cards as the human users only marginally more often than an agent with randomly allocated weights. To approximately half of the prompts, the trained agent chooses the same answer as most humans; in almost 78 percent of the situations, the answer given by the agent is within the top three most common human answers. Additionally, most pure strategies, created by setting their weight to 1 while all other strategies are set to 0, performed worse than the trained agent. Notable exceptions to this trend are the `hand_max_poss` and `hand_max_avg` strategies, suggesting that in more situations than the agent, the typical human player will play according to what points can be expected to be gained from the cut card. Interestingly, the `hand_max_poss` strategy's presence as the second most common pure strategy used indicates a significant degree of risk-taking present in the users' responses. According to user comments, this increased riskiness in play may be a result of some users attempting to maximize points of the hand, without the responsibility to actually play the resulting game, and may not be entirely representative of actual in-game choices.

As a result of this finding, some of these strategies were used as initial weights to the learning process in order to determine if the agent could learn to fine-tune a policy starting from a reasonable assertion of good game-play as well as learn to discount demonstrably poor strategies. Since the update mechanism for weights relies upon renormalization of a vector which as been rewarded or punished, no modifications would occur in the case of punishment of a completely pure strategy since no other weights would have the chance to increase. Therefore, the pure strategies used before were slightly modified such that each other element of the $\overline{w}$ vector would have a small initial value which could be increased when this semi-pure strategy was punished. Two agents, each initialized with the same semi-pure strategy, were played against each other for a million games, with only a single agent updating its weights.

Results The comparison of different strategies' development after training (Figure 21) shows the agent learning applicable policies for both winning and losing positions. In the winning positions of the strategy graphs, the starting strategy is further strengthened to dominance. The purely mathematical operation of overcoming such an initially heavily weighted strategy requires many games to explore enough and perform better than the starting strategy to be rewarded; the explored combination of cards, itself, may coincide with the starting strategy already occupying these positions, further increasing the difficulty. Additionally, since the opponent

Strategy	Top 1	Top 2	Top 3	Percentage in Top 3 Human Choices
pegging_min_avg_given	160	303	458	12.64
pegging_max_med_gained	268	519	796	21.97
pegging_max_avg_gained	347	650	963	26.58
crib_min_avg	380	177	1081	29.84
hand_max_min	1576	2288	2666	73.59
Random	**1589**	**2321**	**2743**	**75.71**
hand_max_med	1649	2353	2768	76.40
Trained	**1706**	**2426**	**2821**	**77.86**
hand_max_poss	1677	2433	2847	78.58
hand_max_avg	2066	2828	3168	87.44

Table 4: Number of times the agent using a given strategy chose the same cards as the most common choice by human users according to 3623 total parsable records obtained from [DCH]. The columns labeled "Top X" display the number of times the given strategy's choice was within the top X choices of the user base. In this table, **Random** is the average of results from five agents which each used independently randomly allocated weights and **Trained** used an agent trained in Round 2's losers' bracket for one million games.

is a static agent always using the same starting semi-pure strategy and never training these weights, playing a similar strategy ensures a similar resulting position. When already in the lead, this is desirable as the agent will likely continue to be in a winning position and closer to the goal score of 121. An intriguing counter to this pattern of dominance in winning positions is the `hand_max_avg` strategy which yields some control of winning positions to `hand_max_med`. After normal training procedures, this space is occupied by `hand_max_min`, which, characteristically, would not allow a larger gap to develop.

In losing positions, the agent develops a reasonable counter policy to the given semi-pure opponent. For instance, when faced with `hand_max_min` which will always play safe, the agent learns to swing for the fences by playing according to `hand_max_poss` when it is losing since the opponent will not be taking any risks itself and risk is the only way to make up ground. Similarly, the agent learns that the best way to recover from a losing position against `hand_max_avg`, which plays to expectations and avoids unnecessary risk, is to mostly play safely according to `hand_max_min`. As a counter to the dominance of a single strategy in the losing positions, starting with `hand_max_poss` leads to a losing strategy shared between `hand_max_avg` and `hand_max_med`. This is because either of these two strategies is likely to recover ground against an agent which always tries to get the maximum amount of points with no regard for the likelihood of this outcome.

As this experiment showed promise in an agent potentially learning to out-play its opponent, an agent trained with a beginning pure strategy of `hand_max_avg` was played against its previous iterations in several 10,000-game tournaments. The

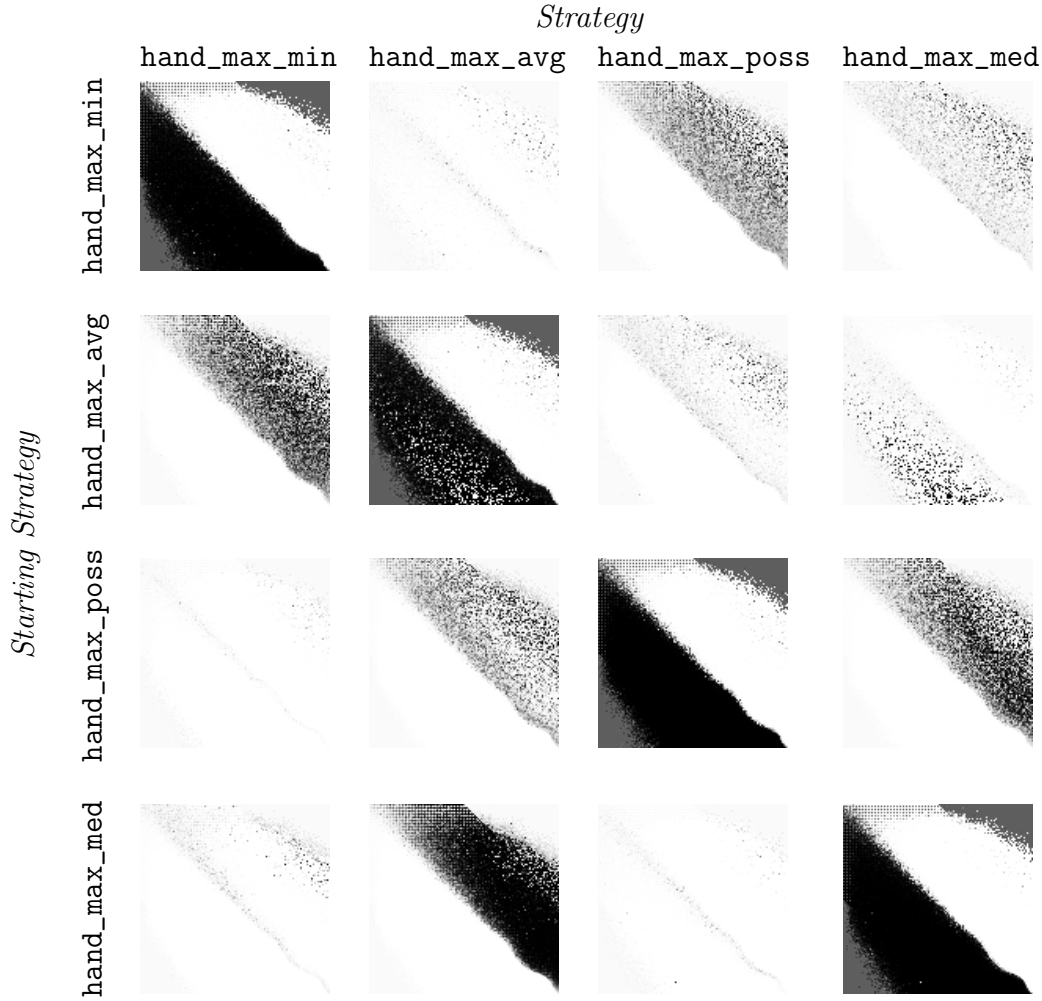
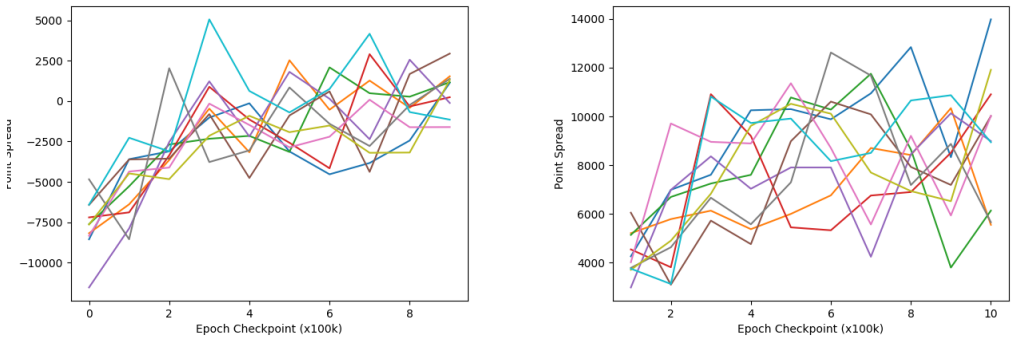


Figure 21: Comparison of how different strategies develop with different starting points. The strategy along the *Starting Strategy* axis is that strategy which started with 70% weighting and the strategy along the *Strategy* access is merely the strategy graph being presented. These strategy graphs are from an agent that has been trained for one million games against the same semi-pure strategy it started with, when playing as the pone.



(a) A trained agent starting from a semi-pure `hand_max_avg` strategy plays against previous iterations of itself.

(b) An agent using a semi-pure `hand_max_avg` strategy plays the epoch checkpoints of an agent which has been trained against this strategy.

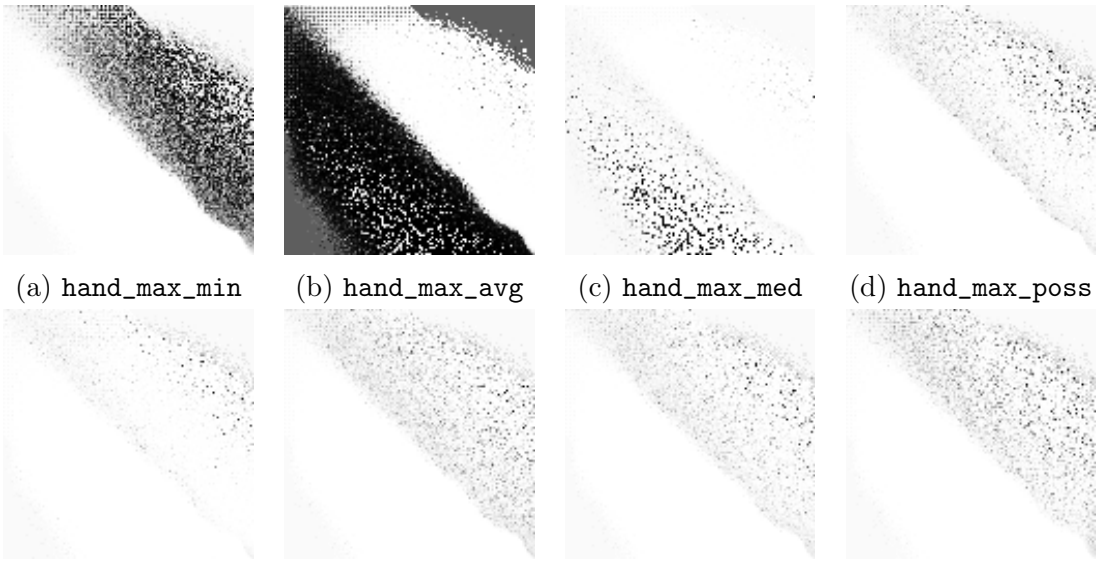
Figure 22: Point spreads of several 10,000-game tournaments between agents of varying training levels when started with set of semi-pure `hand_max_avg` strategy weights.

results, depicted in Figure 22, show a steady decrease in performance as training proceeds. Earlier iterations, closer to the semi-pure `hand_max_avg` strategy, perform better than the trained agents. Without further context, it would appear that `hand_max_avg` is simply a good strategy which is difficult enough to overcome, so the training framework is forcing an adaptation unnecessarily and undesirably. However, as the strategy graphs developed against a static semi-pure `hand_max_avg` opponent (Figure 21) match those developed against a random agent (Figure 23), this conclusion is unfounded. If the `hand_max_avg` strategy were clearly superior to others, then very little modification would be made by the agent. Instead, the losing states still drifted away from the starting strategy. The only conclusions that can be made from this data are that losing positions are fundamentally difficult to recover from and that cribbage involves much more than is encapsulated by the small set of strategies with which the agent has been programmed.

Learning Rate Adjustment

In order to determine if the learning rate was too high in Round 2, even though it had been significantly reduced from Round 1, a variety of learning rates were tried. These runs were intended to see if an optimal policy was being overstepped by making too large of an adjustment.

Results Even with very reduced scaling factors, no difference in behavioral trends learned was observed. However, it has been demonstrated in Figure 24 that a decrease in scaling factor leads to slower adjustments, showing that the scaling factor does indeed function as a learning rate. In fact, the images along each counter-



(a) `hand_max_min` (b) `hand_max_avg` (c) `hand_max_med` (d) `hand_max_poss`

(e) `crib_min_avg` (f) `pegging_max_avg_gained` (g) `pegging_max_med_gained` (h) `pegging_min_avg_given`

Figure 23: All final strategy strengths for an agent which started with a 70% semi-pure `hand_max_avg` strategy and was trained against an agent with unchanging random weights for one million games when playing as the pone.

diagonal are nearly identical because they roughly align in cumulative adjustment magnitude made to the weights. Therefore, it is safe to conclude that some optimum is not being stepped over, allowing a worse set of weights to be pursued instead.

Decay Rate

In addition to the learning rate, the decay parameter d was also adjusted to see what sort of effect it would have on learning. Instead of the default decay rate of 10%, rates ranging from 0% to 50% were tested to demonstrate the effect of temporal difference learning.

Results Adjustments in decay rate made no change in the behavioral trends learned. However, as can be seen in Figure 25, the rate of learning of these behaviors in earlier states is highly affected by this parameter. With lower decay rates, more of the responsibility for the final result of the game is transferred to earlier game states, meaning trends develop quickly. Contrarily, with higher decay rates, it is nearly impossible to learn earlier game states, so behavior remains random, as clearly seen when $d = 0.50$. For explanation, a typical cribbage game will take approximately ten hands to complete. At this rate, the first states will be adjusted by a factor of $(1 - d)^{10} = (0.5)^{10} \approx 0.00098$. Even over the course of one million games, this small magnitude of modification, combined with the rate of visitation of a single state, is not enough for any meaningful learning to occur in

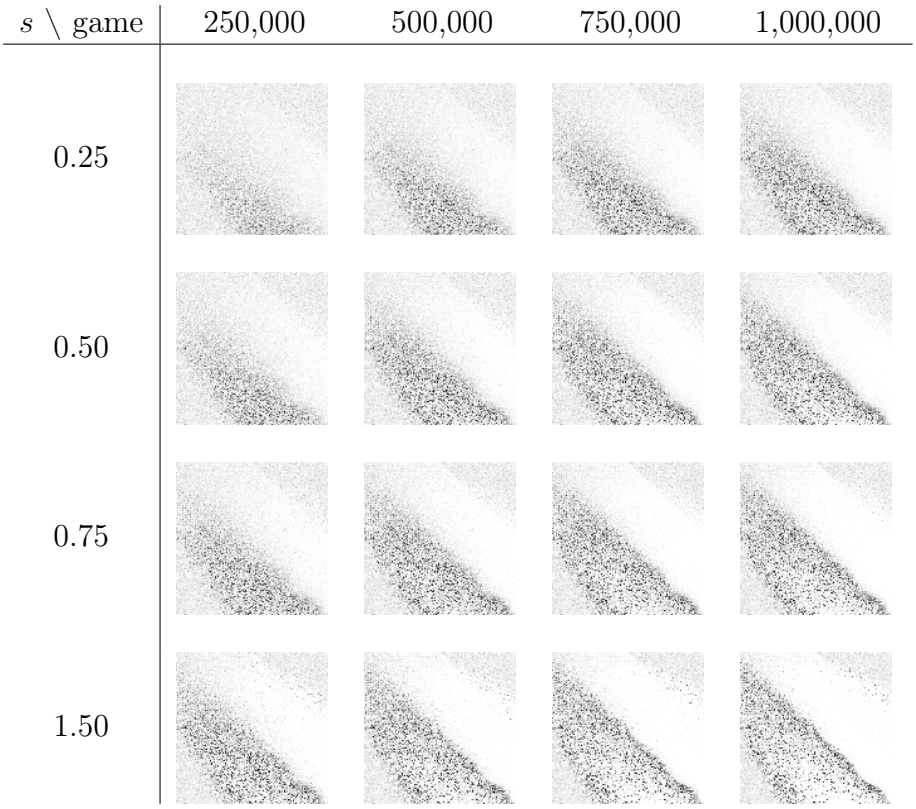


Figure 24: Comparison of different scaling factors (s), learning the `hand_max_avg` strategy when playing as the pone over the course of one million games. For comparison, $s = 10.0$ and $s = 2.0$ for Rounds 1 and 2, respectively (see Figures 2 and 11).

these earlier states.

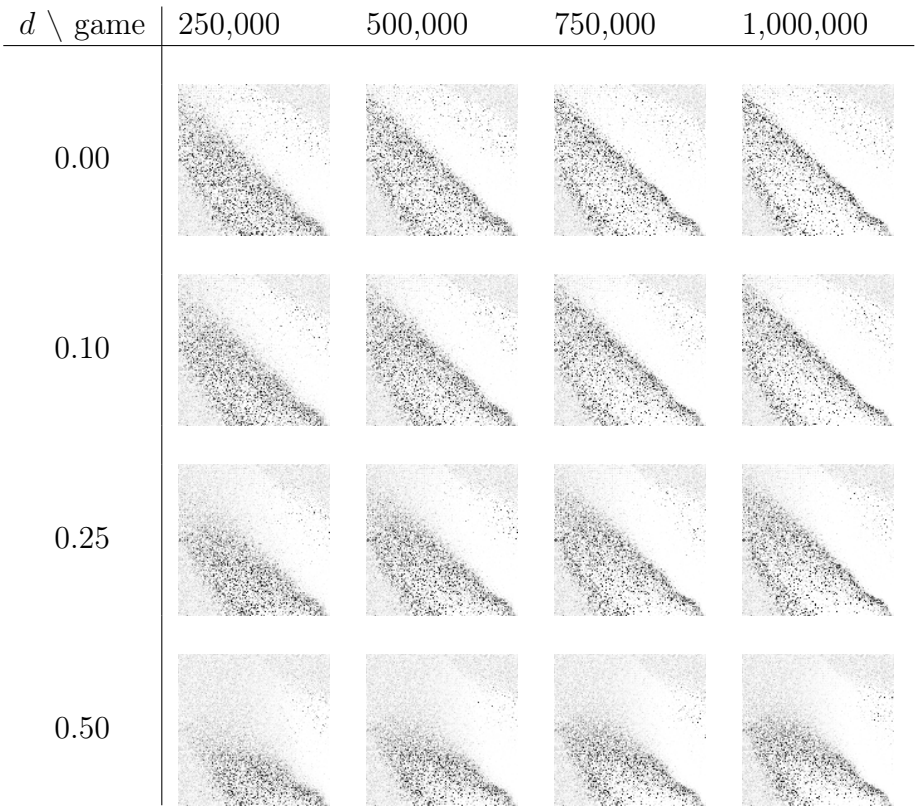


Figure 25: Comparison of different decay rates learning the `hand_max_avg` strategy when playing as the pone over the course of one million games.

5 Discussion

This thesis focused on an attempt to use ϵ-greedy Monte Carlo methods with exploring starts to develop a well-playing cribbage agent capable of weighing different strategies with the intention of using these learned weights to potentially improve the play of the author and readers. In the primary objective of evolving a cribbage agent to a good strategy, despite the setbacks of poor performance on a per-game level, the agent did remarkably well by removing irrelevant strategies such as `hand_max_med` and `pegging_max_med_gained` and emphasizing more useful strategies like `hand_max_avg` and `hand_max_min`.

This section will describe some reasons for successes and failures of the learning process as well as potential future applications or improvements to the system which can help make a successful cribbage-playing agent in the future.

5.1 Future Possibilities

Due to assumptions and simplifications made throughout the design and implementation process, several suboptimalities exist across this thesis which leave room for improvement in potential future endeavors attempting to apply machine learning to the game of cribbage.

Pegging

One area of the project that can be improved was the pegging system. Because the focus of the thesis was on whether or not a collection of strategies could be learned in combination when choosing which cards to keep, an ideally playing agent was not the outcome. A key reason for this was the lack of time dedicated to and nonchalant nature towards learning how to actually play the crucial pegging phase.

Playing Strategy The first shortcoming of the pegging portion of the agent was the simplicity of its playing strategy. In order to focus on the overall playing strategy and to have a consistent knowledge of how a set of cards have performed in the past, the agent was only capable of following a very simple immediately greedy heuristic: of the cards available which are legal to play, play the one with the highest immediate return. This has the potential flaw of opening oneself up to the possibility of allowing the opponent an opportunity to score more than the agent itself just gained, resulting in a net loss. As has been demonstrated by the rest of this thesis, the idea that one strategy can be applied at all times in the game of cribbage is laughable.

In future applications, a partial agent could be trained in just the pegging phase using reinforcement learning. In a similar way to this thesis, multiple basic strategies such as offensive or defensive play could be trained by rewarding each behavior and their combinations could in turn be learned through gameplay simulations.

Alternatively, a more ground-up approach can be taken to train an unbiased agent by simply having each player learn from trial and error with certain card combinations.

On the other hand, as cribbage is an imperfect-information game, a parallel can be drawn to the game of poker. Since pegging is the phase of cribbage which most depends on predicting and countering the opponent's strategy, recent successes by Dr. Tuomas Sandholm and Noam Brown in heads-up no-limits Texas hold'em may be applicable to this phase. Their AI, *Libratus*, defeated four top human players in a 120,000-hand tournament [BS17]. Furthermore, while *Libratus* utilized massive computational resources, Sandholm and Brown have also developed *Modicum*, a less powerful, but less computationally expensive, poker-playing agent which utilizes only hardware that can be found in moderately powerful desktop computers [BSA18]. Although not as well-performing as *Libratus*, *Modicum* was still capable of beating other previous top poker agents using fewer resources by orders of magnitude, allowing for its use in research by those without access to high-performance machines. In any case, a more suitable pegging system can be made, improving play.

Performance Evaluation Even though a better pegging system can be made, key to this thesis was the idea that performance can at least be anticipated with some amount of certainty. In this project, the performance of the pegging agent was tracked by card combination, recording the points scored and yielded during each occurrence. These records were used to anticipate how each combination of cards would play against an opponent.

Before the first round of training, however, there was relatively little pre-population of these records done: roughly one hundred thousand randomly dealt hands were played against each other. As a result of the small sample size, performance estimates were likely to be inaccurate. Furthermore, the small sample size did not cover all of the possible hand combinations. Because of this, data would be missing for multiple combinations of cards for the decision process throughout the first round. This means that the card combinations' true desirability would be misrepresented during the calculation, allowing for the possibility of an unfair punishment or reward for what would amount to a guess by the pegging-based strategies. It also means that the information received by the agent during the weighting operation would not be static, with respect to the cards given, as other strategies were and thus could not be as reliably trusted for accuracy. While this was counteracted by using records from a first round training session in all subsequent training sessions, the concern remains whether this was enough and how much the variability in the data provided by the pegging strategies affected learning.

Implementation Decisions

Due to the desire to get a training framework constructed and running as quickly as possible, runtime speed was deemed less important than development speed. As

a result, Python became the language of choice. While this is a great multipurpose language, the overhead of using a higher-level, interpreted language proved to be a mistake not only directly because of its slower speed involved with the lower-level calculations and manipulations needed for cribbage applications, but also because of resulting decisions that needed to be made to accommodate this reduced speed.

Database Dependency Because Python was the language of choice, the performance of making a decision for a single hand was sluggish: on a development machine, the decision for a single hand would take approximately ten seconds. As previously mentioned in Section 3, the decision made from that point was to create a database such that all calculations were done ahead of time and the statistics desired would be available upon request. This succeeded in its desired goal, bringing the time required for a single decision to approximately one twentieth of a second. At this point the project was able to play two games in about a second's time on the same development machine.

Despite the speed improvements in a development environment, the entire training process had become I/O-bound rather than CPU bound. This became a problem when an attempt was made to train multiple agents simultaneously. Because of concurrency issues which would have taken too long to fix or work around, no more than one process could successfully access the SQLite database with any consistency. This would mean that only two agents could be trained using a single database at any given time. Furthermore, since this database file was around twenty gigabytes in size and there was only limited local storage space available on each of the compute nodes in the Computer Science Department's Ukko2 high performance cluster, jobs had to be constructed carefully and spread across nodes to avoid accidentally disrupting other users' processes. This was mitigated slightly by using the Physics Department's Kale cluster which allowed for significantly more manageable locally mounted drive space, albeit in a slightly slower hard disk with a RAID0 configuration.

Were this project to be repeated, it would be recommended to use the Python code from this attempt as an outline for a rewrite in a lower-level language. Using a language such as C++, which could be optimized more for run-time efficiency, would likely increase speed of computation enough to remove the necessity of a statistics database entirely. While the data gathered from previously played pegging rounds would still need to be stored and retrieved between training rounds, the total size of this data would likely not exceed a few gigabytes and would fit easily in the program's RAM space. This is especially true if only minimums, maximums, and averages are considered, as their calculation does not require all values previously seen to be stored, meaning that far less storage space is needed. Also, like weight checkpoints are in the current system, these stored statistics could be exported for transfer between rounds.

With these changes made, the training program would be CPU-bound, which can be compensated for by parallelization. In contrast, the I/O-bound training program

used in this thesis used only half the total processing power of a single CPU core as the bottleneck was the database file on a hard drive.

Multi-agent Play

In all of the tournament training sessions and other experiments, two agents were paired against each other for a million games. It is rare for humans to only play a single opponent for an extended period of time. Only in tournaments, or in absence of other available opponents, would a human player play more than a couple of games against the same opponent. Although the reinforcement learning algorithm requires far more games to achieve similar progress in learning, there is no need to limit the agent to a single opponent.

As an alternative, the agent could be pitted against multiple agents, each with their own unique set of weights. This would force the agent to learn how to overcome a variety of opposing strategies, strengthening the overall performance of the agent. One potential solution to this is to have a pool of agents from which the agent will randomly select an opponent. Similarly, two agents could be chosen from a pool at random and pitted against each other, in a round-robin fashion.

After the promising results of the experiments using different starting points for the agents, the idea of learning how to overcome multiple, widely varying agents rather than a single co-adapting agent seems to be the most promising starting point for continued research into this topic. While learning from self-play was idealistic and even demonstrated possible by AlphaGo Zero [SSS+17], it was perhaps naïve as the need for some priming was a crucial step in earlier AlphaGo work [SHM+16] and in TD-Gammon [Tes95]. As an alternative to pure self-play, and in the absence of recorded games, an agent representing each of the most potent basic strategies (e.g. `hand_max_avg`, `hand_max_min`, and `hand_max_poss`) could similarly be played against and used to train the learning agent to basic levels before self-play is used for further improvement.

5.2 Shortcomings of the Model

In addition to the shortcomings of the implementation decisions made or necessitated by other decisions, the nature of the model itself should be considered and evaluated. Several assumptions were made to limit the scope and dimensionality of the problem, but it is likely the case that these limitations also affected the performance of the agent.

Linearity

One of the model's shortcomings that must be addressed is the linearity of the decision making apparatus. As described in Section 3.2, the mechanism for deciding which combination of cards to choose is, at its core, a simple linear operation. The

weights vector $\overline{w}_{p,o,d}$ is multiplied by a desirability matrix **S** to produce a probability vector $\overline{p}$, from which the maximum value is chosen when the policy is strictly followed. Although a human player would indeed evaluate which combinations are best to play with a set of strategies with varying importance over the course of the game, the nature of this relationship in the player's mind is unlikely to be a simple linear function. Additionally, a human player would also consider how two or more strategies might interact with each other at different points in the game, something the combination mechanism is unable to accomplish.

Strategies

Another shortcoming of the setup to the model to the problem of this thesis is the limitation of the strategies chosen. Since each strategy's contribution to the **S** matrix can be thought of as a feature, this means that the final model is no more than a simple linear combination of eight features used to determine the best choice of cards. Furthermore, the features selected, although sensibly selected by a fairly experienced cribbage player, may themselves not be ideally suited to the task at hand.

5.3 Usefulness of Results

Despite the trained agents' shortcomings in ability to consistently win a single game, this thesis can still be applied to expand the current knowledge of cribbage as well as serve as a guidance story. The generated strategy graphs can be used as a set of guidelines as to how a player *should* be playing a hand with a given score. Although a human player may not be able to calculate the statistics which the agent used as accurately as a computer, a fair amount of experience and intuition can be gained through repeated play which should approximate the expected and guaranteed returns well.

The agents' successes in the macro scale can also be of use to those applications which also operate on the scale of thousands to millions of games. Mainly, the trained agent could be used for the elementary first stages of training of a subsequent agent. Rather than learning from the self-play with no existing knowledge, a policy-following agent can be played against instead. Playing an agent of greater difficulty would allow the new agent to learn more quickly.

6 Conclusion

This thesis presented an attempt to learn how to play cribbage according to a set number of predefined strategies through reinforcement learning. Section 1 introduced the game and the topic of reinforcement learning for context. Section 2 provided an overview of related work in reinforcement learning within the domain of games as well as attempts to apply machine learning specifically to the game of cribbage. The methods for how the agents were trained was presented in Section 3. Section 4 detailed the results of the training and experiments to determine ways in which better results may be obtained. Finally, a discussion of potential applications of this thesis were presented in Section 5 as well as a summation of ways in which someone looking to continue this effort to apply reinforcement learning to the domain of cribbage might proceed.

References

Ame16 American Cribbage Congress, Official tournament rules, 2016. URL http://cribbage.org/rules/ACCRuleBook2016.pdf. Accessed April 9, 2018.

Ame18 American Cribbage Congress, Rules of cribbage, 2018. URL http://www.cribbage.org/rules. Accessed May 26, 2017.

BS17 Brown, N. and Sandholm, T., Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*. URL http://science.sciencemag.org/content/early/2017/12/15/science.aao1733. Accessed June 12, 2018.

BSA18 Brown, N., Sandholm, T. and Amos, B., Depth-limited solving for imperfect-information games. *arXiv preprint arXiv:1805.08195*.

CHJH02 Campbell, M., Hoane Jr, A. J. and Hsu, F.-h., Deep blue. *Artificial intelligence*, 134,1-2(2002), pages 57–83.

DCH Daily Cribbage Hand. URL http://www.dailycribbagehand.org. Accessed April 18, 2018.

Dee18 DeepMind, AlphaGo, 2018. URL https://deepmind.com/research/alphago/. Accessed May 18, 2018.

Kho10 Khomskii, Y., Infinite games. Technical Report, University of Sofia Bulgaria, Summer Course (July 2010), 2010.

KS02 Kendall, G. and Shaw, S., Investigation of an adaptive cribbage player. *International Conference on Computers and Games*. Springer, 2002, pages 29–41.

Mar00 Martin, P. L., Optimal expected values for cribbage hands. Bachelor's thesis, Harvey Mudd College, 2000. URL https://www.math.hmc.edu/seniorthesis/archives/2000/pmartin/pmartin-2000-thesis.pdf. Accessed May 24, 2017.

O'C00 O'Connor, R., Temporal difference reinforcement learning applied to cribbage. Bachelor's thesis, University of California, Berkeley, 2000. URL http://r6.ca/cs486/. Accessed September 5, 2017.

PB97 Pollack, J. B. and Blair, A. D., Why did TD-Gammon work? *Advances in Neural Information Processing Systems*, 1997, pages 10–16.

Sam59 Samuel, A. L., Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3,3(1959), pages 210–229.

SB Sutton, R. S. and Barto, A. G., Reinforcement learning: An introduction. Incomplete draft used available from author's site (`http://www.incompleteideas.net/book/the-book-2nd.html`). Accessed November 3, 2017.

Sha53 Shapley, L. S., Stochastic games. *Proceedings of the National Academy of Sciences*, 39,10(1953), pages 1095–1100. URL `http://www.pnas.org/content/39/10/1095`. Accessed May 16, 2018.

SHM+16 Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. et al., Mastering the game of go with deep neural networks and tree search. *Nature*, 529,7587(2016), pages 484–489.

SSS+17 Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. et al., Mastering the game of go without human knowledge. *Nature*, 550,7676(2017), page 354.

Tes95 Tesauro, G., Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38,3(1995), pages 58–68.