Using Reinforcement Learning and Algorithmic Techniques to Develop Strategies for Tournament-Style Backgammon Matches

Student Name: Will Farmer
Supervisor Name: Robert Powell
Submitted as part of the degree of BSc Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—There exist countless agents and algorithms for playing backgammon, but they all focus on winning individual games, despite the fact that tournament backgammon is played up to a certain score and each game may be worth a different number of points. This project provides a collection of different agents for playing tournament-style backgammon that outperform intermediate-level human players and compete with current and classical backgammon programs. The deep learning model has been trained to output a move preference based on equities generated from a state-of-the-art program. The reinforcement learning model was trained through self-play and Temporal Difference Learning. The Deep agent's move selection statistically outperforms state-of-the-art agents trained using supervised learning, achieving intermediate-level play, and the Reinforcement agent outperforms all other agents presented in this report and attains advanced-level play. These agents demonstrate the flaw in building agents with the aim of winning individual games and express the promise in shifting the typical training paradigm to aim to maximise point per game (PPG) differential instead of win rate.

Index Terms—Artificial intelligence, Evolutionary computing and genetic algorithms, Games, Neural nets

1 Introduction

In backgammon's estimated 5000-year history it has been played in many formats with many variations of play. However, tournament-style backgammon has had a consistent ruleset since the first World Championships in 1967, with the only change being the score that the player must reach to win the match, which is typically up to the players' discretion but usually in the range of 17-25. Each game yields a certain number of points depending on the circumstances: 1 point for a regular win, 2 for a win by gammon and 3 for a win by backgammon. There also exists a device called the doubling cube, which can be played to double the stakes of the game. If a player offers the doubling cube, but is rejected by the opponent, then the player wins the game by the current value of the cube. A gammon occurs when one player has borne all of their pieces off while the other has borne no pieces off. A backgammon occurs when one player has been gammoned and still has a piece on the bar or in their opponent's home. Obviously it is better to win by a gammon or backgammon than otherwise, and conversely one should avoid losses by gammon or backgammon at all costs, and so it seems clear that one should include these circumstances in the program's decision-making. Once the doubling cube has been played, only the player who was offered the cube may use it. There is no limit to the number of times the doubling cube can be used in a single game, though the value and ownership of the cube reset at the start of each game. The doubling cube is an extremely powerful tool in backgammon, so much so that a player with world-class move-play but beginner-level cube strategies would most likely lose matches to intermediate-level opponents due to inappropriate handling of the cube.

Despite this first-to format of tournament-style backgammon, classical and modern backgammon programs have primarily focussed on winning individual games, with the idea being that if you win every game, you will win the match. In deterministic games with perfect information like chess and checkers, it is likely that the best player will win every game, especially if they start first. However, the random dice rolls in backgammon mean it is almost impossible to guarantee victory in every single game, unless there is a vast gulf in ability between the two players (an unlikely event when playing in tournaments). A player could be extremely far ahead of their opponent in a race and then suddenly receive terrible dice rolls while their opponent starts rolling exclusively 6-6. In this case, the better player will lose through no fault of their own. Factor in that one player could win by gammon or backgammon, or have a high cube value when winning, and it is clear to see that focusing on individual games is not a foolproof strategy for tournament-style matches, as simple bad luck can immediately ruin your chances of winning the match. This project is the first to tackle the problem of winning these tournament-style backgammon matches, with 25 chosen as the score to win the match. This value was selected to minimise the effect of ”luck” in the randomness of the dice rolls, but not to be so large that it would be impractical for humans to play in one sitting. Ultimately, the question this project answers is whether designing agents to maximize PPG differential rather than win rate improves agent performance in tournament-style

backgammon matches?

The agents produced in this project utilise different algorithmic and machine learning paradigms to tackle the problem. Each agent, the creation of a stable game engine, as well as a GUI, is considered a deliverable. These deliverables give insight into what techniques suit the problem of backgammon most, given its large branching factor (backgammon has an estimated branching factor of 250 compared to chess's estimated branching factor of 35 [1]). Analysis of the performance of each agent gives indications as to which methodology is best used in the different phases of backgammon. The two most notable phases are racing and contact. In racing positions, both players have moved all their pieces past their opponents and so cannot be hit off the board (where they have to re-enter from the start and no other piece can be moved until this has been done). The difference between these two phases is so vast that the best-performing agents had to have separate algorithms or models for selecting moves in racing positions and contact positions. Each agent is also used as a baseline against each other agent to assess performance and generalisability against different strategies. The agents are divided into two categories: intermediate and advanced. The intermediate agents are relatively basic and are not expected to achieve strong levels of play, but hope to give insight into what strategies and positions are successful and which are not. The advanced agents are designed to be able to compete with intermediate-level human opponents and consequently utilise more complicated techniques and structures such as reinforcement learning and neural networks. The advanced agents will be constructed using the information obtained by building and observing the intermediate agents.

As a result of how the agents were trained or developed, each adopts a different strategy. The Greedy agent has nested conditional statements that only compare the difference between the board state at the start of the move and possible board states at the end. The Genetic agent does the same, except none of the conditional statements are nested and so every circumstance is considered. The Expectimax agent looks ahead to the opponent's turn and attempts to minimise their expected payoff by using the Greedy agent's evaluation function. The Adaptive agent combines features from all 3 of the intermediate agents, using the genetic agent for opening moves and using the Expectimax agent's lookahead for tiebreaking equally-scored moves. On top of this, it scores each move by computing the probability that the agent wins from each possible board state, rather than comparing it to the board state at the start of the turn. The Deep agent uses its neural network to decide which of the possible board states are the best, while the Reinforcement agent uses its neural network to estimate the probabilities of each terminal game outcome and uses them to compute an expected value corresponding to the equity of a position. The agent then looks ahead to the opponent's move similar to how the Expectimax agent does.

As part of this project, this author had to craft datasets for choosing optimal moves and doubling decisions in backgammon as no free or digital datasets exist. The author intends to release these datasets publicly so that others may use them to train supervised learning models for use in backgammon. The agent trained on this data achieved statistical parity with Neurogammon [2], which handily won the 1989 Computer Olympiad. The core difference between the two models is that Neurogammon was given specific cases in training to avoid making certain mistakes, whereas the Deep agent was trained on a large volume of randomly generated boards. This method of data collection for supervised learning is certainly novel for backgammon and may prompt further investigation given its success.

Furthermore, the highest-performing agent in this project appears to be the first to utilize TD(λ) learning with 6 output units, as backgammons are not normally considered due to their rarity. This change in the model's structure yields a shift from the typical strategy resulting from TD(λ) learning, causing more aggressive plays against weaker opponents to maximise PPG differential. This change also causes the model to fear loss by gammon and backgammon and start rapidly moving pieces up the board when at risk of doing so or by playing technical backgames, which is a behaviour not shown by models just trained on cubeless equities, or with no indication of gammons and backgammons. These behaviours are natural for human players but are sometimes not observed by backgammon programs, showing the importance of considering backgammons and maximising PPG differential.

The developed agents are evaluated on their move-play and cube decisions. Each agent plays each other agent in multiple head-to-head cubeless matches to see which strategies are dominated by other strategies, which strategies generalise well against other opponents, and which boost the agent's points per game differential the most. Doubling decisions are tested against a large dataset of randomly generated boards and the resulting accuracies are compared against each other. Intermediate-level human trialists then played cubeful matches against each agent, with the match scores and ratings from the humans used to evaluate the agents' move-play and cube decisions to form an overall judgement of agent strength. Finally, the strongest agent was selected to play against various difficulty settings of the best openly available backgammon program [3] to estimate its level of skill (Table 3 contains the level of play for each difficulty).

2 Related Work

This project is the first to solve the problem of tournament-style backgammon. However, much research has been conducted on individual backgammon games, due to its large branching factor and overall complexity. Research conducted on other extensible games and probabilistic games are also relevant and may be applicable to backgammon.

2.1 Heuristic Approaches to Backgammon

Berliner [4] was the first to successfully attempt to build a program to play backgammon. The resulting AI, named BKG, was a hideously complicated heuristic algorithm, developed under immense scrutiny and supervision from the world's leading backgammon players. The algorithm would take in the state of the board as input, extract and compute features, and assign them weights. The sum of these weights would be the output of the algorithm and

would represent the strength of the position, known as the equity. The program would then select the move that yielded the greatest equity and would not look ahead to the opponent's turn. In 1979, BKG beat the world champion at the time in a first-to-7 exhibition match. This was the first glimpse of promise for man-made machines in games of skill, and in fact was the first time a man-made entity had beaten a world champion in a recognised intellectual activity [5]. BKG was unanimously the best backgammon-playing program for many years, and even still remains the best heuristic backgammon program.

2.2 Supervised Learning Approaches to Backgammon

Neurogammon [2] was the first application of neural networks, and indeed machine learning, in backgammon. In 1989 it handily won the First Computer Olympiad [6]. While Neurogammon's move selection was strong, Tesauro admitted it was likely not as strong as BKG's [7]. However, Neurogammon also used neural networks for doubling. Tesauro collected boards from 300 expert games and used a "crude 9-point scale" for the doubling strength of these positions [2]. The model would estimate the doubling strength of a position and consequently choose where to double, and whether or not to accept a double when offered. Despite being put together very soon before the tournament (all programs were required to be able to make cube actions in order to qualify), "*the consensus of expert opinion at the Olympiad was that Neurogammon's doubling algorithm was probably the strongest part of the program.*" [2]. Although the hastiness with which the strategy was built is a weakness of the study, it presents further opportunities for exploration. Another large difference between BKG and Neurogammon's doubling strategies is that BKG cannot double until in a racing position, whereas Neurogammon doubles whenever its network detects a position of suitable strength. Waiting too late in the game to double can be punishing, as an opponent may be more likely to pass the double with a greater chance of defeat, reducing the player's payoff.

2.3 Unsupervised Learning Approaches to Backgammon

In 1995, Tesauro drastically improved upon Neurogammon with the creation of TD-Gammon [1], a backgammon-playing agent that was trained via self-play and temporal-difference(λ) learning. This unsupervised learning approach to backgammon yielded significantly better performance than anything seen before, achieving master-level play. Comprising of a number of input units representing the board state, a hidden layer of between 40 and 160 units (depending on the iteration of TD-Gammon), and 4 output units representing a basic win, win by gammon, basic loss, and loss by gammon. TD-Gammon was able to learn important features from the board and use these learned features to make near faultless move decisions. However, by not considering backgammons in its reward vector, it is possible that TD-Gammon is not punishing enough against weaker opponents and thus does not maximise its PPG differential, which could cause it to suffer in tournament-style games.

Despite the success of Neurogammon's doubling algorithm, Tesauro decided to follow the theory, plugging the model's expected reward estimates into Zadeh and Kobliska's theoretical formula for optimal doubling [8]. Given that TD-Gammon's reward estimates are extremely accurate, it is likely that this is an improvement on Neurogammon's doubling strategy. However, no direct comparison tests were ever performed.

2.3.1 Training Methodologies

TD-Gammon's self-play training methodology ensured that it had a constantly competitive and evolving opponent, so it would not overfit to a single strategy or converge too early and suboptimally against an easy opponent. Another training methodology this author considered was curriculum learning [9]. In curriculum learning, the learning agent plays against an opponent until it is deemed to have overpowered it. Once this has occurred, the opponent is swapped for a slightly more difficult one. This is similar to the method of teaching in schools, where students are taught elementary concepts first to build a foundation of knowledge and understanding before being introduced to more complex topics. In order for curriculum learning to be applicable to a scenario, the skill gap between the adversaries must be very slight so that the agent does not get defeated too intensely or consistently, as this may inadvertently cause them to learn that good positions are, in fact, weak. In the scenario where the learning agent starts to outperform all presented adversaries, then the methodology must shift to self-play regardless in order to keep the learning agent from overfitting to the adversary's weaker strategy.

2.4 Game-Playing Tree-Search Algorithms

In extensive games like chess and backgammon, one can construct trees that represent each turn of the game. These trees can then be searched and combined with evaluation functions to play the games.

2.4.1 Minimax

The most common tree-search-based algorithm used in two-player games is the Minimax algorithm. Board states are represented as nodes and the moves (transitions between board states) as edges. The root node is the board state at the start of a player's turn. It looks ahead at all possible board states, using the moves as transitions between boards, to a pre-determined ply. The AI assumes the adversary will always choose the move that is the worst for the AI and so chooses a move that maximises its own score while minimising the score the opponent can get in return. This method is performed by two devices, a minimiser and a maximiser. The two devices work on alternating layers of the game tree, where the minimiser assumes the opponent chooses the move worst for the AI and the maximiser chooses the move best for the AI for when it would be the AI's turn. This ensures that the best move is chosen according to the evaluation function as each ply. It is only practical for use in deterministic games with low branching factors, though can be combined with pruning techniques [10] to expand its scope.

2.4.2 Expectimax

Expectimax, also known as Expectiminimax [11], is another algorithm that uses game trees to make decisions. It is based on Minimax and thus uses the minimiser and maximiser, but is designed for probabilistic games like backgammon and 2048. The root node is the current game (board) state. There is then an edge to each probabilistic outcome that determines the possible moves (dice rolls). From each probabilistic outcome node come edges representing the possible moves, which are then connected to the resultant game states. The game states are scored, and the minimiser or maximiser is applied depending on the tree layer. The probability of each outcome occurring is then applied to each individual score and then summed to give the score for the initial game state. This ensures that the probabilistically best move is chosen each time according to the evaluation function applied at each ply.

2.5 Approaches to Alternative Games

2.5.1 Chess

In 1997, IBM's specialist chess engine Deep Blue defeated the reigning World Champion of the time, Gary Kasparov [12]. Deep Blue used a standard minimax search with Alpha-Beta pruning [10] which would apply its Grandmaster-tuned evaluation function to various possible board states. Additionally, it had an opening and endgame database for dealing with more common and predictable scenarios. The same ideas of pruning and minimax were expanded upon further with the creation of Stockfish in 2010 [13], a more efficient brute-force engine with more advanced pruning techniques. In deterministic games like chess, minimax search is much more applicable than in probabilistic games with large branching factors such as backgammon.

2.5.2 Go

Go is a game more comparable to backgammon as they have similar branching factors. However, Go is a deterministic game and has significantly more plies per game than backgammon. Go was considered to be far too complex for computers, with most of the programs at the time only operating at amateur-level play. However, AlphaGo was able to beat the World Champion of the time decades earlier than thought possible [14]. AlphaGo used two networks: a policy network and a value network. The policy network was initially trained by supervised learning on expert moves, before moving onto unsupervised learning via policy gradient reinforcement learning. The policy network would play against randomly selected previous iterations of the network to prevent overfitting to a single strategy. The value network was trained solely via unsupervised learning and the two networks were used in conjunction with Monte-Carlo Tree Search (MCTS) to calculate fast rollouts. This initially seems highly promising in the context of backgammon given their equally large branching factor and Go's greater tree depth. However, Go is deterministic. In backgammon, you must expand a minimum of 21 nodes in the tree as there are 21 possible dice rolls. Pruning these moves become much more difficult, as even if a move is considered bad in a vacuum, it may be the best possible move for that dice roll, and so would have to be expanded out anyway. In Go, all obviously bad moves can be pruned immediately and reduce computation time.

2.6 Theoretical Doubling Strategies

Keeler and Spencer modelled backgammon as a continuous game with a player's dice having total value of 5 with probability $\frac{11}{24}$, 10 with probability $\frac{11}{24}$ and 20 with probability $\frac{2}{24}$ [15]. Using this abstraction of the game, they identified the optimal doubling point to be when the player on roll had an 80% chance of victory, with the optimal folding point to be if the player not on roll has less than a 20% chance of winning. This strategy was computed to have a higher expectation than any other strategy, though would not maximise payoff against a weaker opponent where doubling earlier is recommended. Two years later, in 1977, Zadeh and Gobliska [8] did not abstract features of backgammon, keeping the dice rolls untouched and treated the game non-continuously (which backgammon is). They agreed with Keeler and Spencer that the theoretical best doubling points were 80% and 20% respectively, but noted that in a non-continuous game one is highly unlikely to ever reach those exact probabilities of victory. They also noted that if the player on roll were to double with a probability of victory less than or equal to 70%, their expected payoff would be less than the previous value of the cube and so should never be done. Additionally, in race situations, they computed a large table comprising of the players' pip counts and the respective win probability of the player on roll, which can be interpolated to find accurate win probabilities for all racing positions.

2.6.1 Poker

6-player Texas Hold'em poker contains trillions of possible combinations of hands at any one time. Not only are the number of combinations grand, but the player has to deal with imperfect information, not knowing what cards each opponent has. This is different to chess and Go, where you can see not only the state of the board at all times, but all moves the opponent can make and similar to backgammon, where you can't accurately determine what moves the opponent is going to be able to make due to the randomness of the dice rolls. Pluribus is an AI application that attained superhuman-level play in poker via self-play and Deep Counterfactual Regret Minimisation (Deep CFR) [16] [17]. Pluribus also utilised game theory algorithms to always aim for a Nash equilibrium to ensure it could not be exploited by any of the adversaries, while also changing its strategy randomly to become less predictable and introduce bluffs. In poker there are two ways of winning: being the only remaining player when all others have folded, or having the best hand at the end of the game. In backgammon there are also two ways of winning: your opponent refusing a double and the player bearing off all their pieces. One can consider a player bearing off all their pieces as equivalent to having the best hand at the end of the game, but folding and passing a double are not equivalent. In backgammon, you have near-perfect information and thus cannot bluff an opponent into passing a double when they are in a winning position like you can in poker, though you can bait them into accepting a double when they are in a losing position.

3 METHODOLOGY

3.1 Robust Game Environment

In order for the algorithms to be applicable to backgammon, the environment in which they are executed must be completely representative of the backgammon ruleset. As such, the first task at hand was creating an engine that completely followed each rule of the game, from rolling a die each to determine who starts the game, to correctly checking the circumstance of victory. The game environment was confirmed to be a true replication of backgammon's ruleset by targeted testing, the playing and viewing of hundreds of games, and by inspection of the moves made by the random agent. The board is represented by a list of 28 integers. The black player (represented as Player -1) attempts to move their pieces to their home at points 18-23, while the white player (represented as Player 1) tries to move their pieces to their home at points 0-5. The black bar is represented by point 24 and the white bar is represented by point 25, with the borne-off black pieces being stored at point 26 and the borne-off white pieces at point 27. The number and colour of the pieces at each point are represented by the formula $x = p * n$, where x is the value of the integer at point i, p is player occupying the point and n is the number of checkers on the point. For example, if the black player had 5 checkers on a point, $x = -1 * 5 = -5$.

3.2 Random Agent

The random agent does not receive any information about the game. It does not know the roll of dice, the score, the state of the board, or even whether it is the black or white player. It simply generates 2 pairs of integers at a time (or 4 pairs of integers in the case of a double) representing start-end pairs. Each time these start-end pairs are generated, the agent attempts to play the move. If the generated move is illegal, it repeats the process until it is able to find a legal move, which it then plays. If the agent cannot locate a valid move after 200,000 iterations of this process, a move is chosen from the list of legal moves at random and played to speed up the rate of play. This agent was developed to test the robustness of the engine and was very useful in debugging the early stages of the program, identifying errors as a result of rolling doubles and incorrectly entering the board. Hence, while it may seem more logical to use the faster approach of just selecting a move at random, the slower method of randomly generating integer pairs more rigorously tests what moves the environment accepts.

3.3 Graphical User Interface

In order for human players to interact with the program in a familiar format, a Graphical User Interface (GUI) was constructed. In user testing, a familiar environment is paramount in ensuring participants operate at their expected level. Similar to many backgammon programs, there are also quality-of-life features. The pieces that can be moved are highlighted, and, when a piece is clicked, all available destination points are highlighted as well. All functionality on the command line was brought over the GUI. The images used in the GUI were found in a GitHub repository [18]. Furthermore, the GUI was vital when testing the game engine. It is much easier to view backgammon games on a GUI than on the command line, which in turn makes it much easier to spot errors or mistakes, such as a piece bearing off when it should not be able to do so, or two identical algorithms choosing to use different moves when given the same board state and roll.

3.4 Greedy Agent

Once the game engine was confirmed to be stable, development was started on the non-random agents. The first agent was to be a 1-ply greedy algorithm that would take in the initial board state as well as all the boards that could be made from each possible move. It would then pass the initial board state and each resulting board state into an evaluation function which would score the move. Initially, the evaluation function was extraordinarily simple, only considering 7 factors (as shown in Algorithm 3). It was so simple in fact that it would sometimes (though very rarely) lose to the random agent, which as stated previously has no knowledge of the game state nor rules.

Hence, the author decided to completely overhaul the evaluation function, using 18 factors (Table 13) and nested conditional statements to score moves. This improved evaluation function considers the same scenarios as Algorithm 3, but also considers trapping the opponent on the bar, anchoring the home board, whether or not the board is in a race or contact position, and many more scenarios. The improvement in performance was staggering. Armed with a new evaluator, the Greedy agent beats the Random agent by gammon or backgammon every time without exception, dropping no points at all.

However, when simulating the Greedy agent playing against itself, this author noticed that the black player was winning considerably more than the white player. After performing hypothesis testing on the available data, it was apparent that there was some inequality between when the Greedy agent acted as the black player and when it acted as the white player. After rigorous testing of the evaluation function and the game engine as a whole, the dice rolls were eventually hardcoded to be 5-1 every time. It was then observed via the GUI that the black player was more likely to push its far-back pieces up the board, while the white agent was more likely to push the pieces that had already progressed up the board even further. This led to scenarios where the white player had checkers trapped in the black player's home board - an undesirable position in backgammon. To solve this, a tiebreaker function was made that would take as input all the board states that had achieved the highest score from the evaluator and return only a single board. The tiebreaker function checks for the largest prime, the number of home walls, the number of blots, opponent and player pip count, and how far back the player's furthest piece from home is in that order. At each check, the number of eligible boards decreases. If there are multiple boards remaining then the selected board is chosen randomly from the list of eligible boards to remove any potential bias. After implementation of this tiebreaker, the Greedy agent played against itself for 10,000 games multiple times, with no significant results. The addition of the tiebreaker not only ensured symmetry, but also improved

the performance of the Greedy agent, allowing it to have a finer grasp of the intricate details of each position.

The Greedy agent was the first agent to be implemented for two reasons: we needed a baseline algorithm to compare the performance of the rest of the algorithms to, and the other algorithms we wanted to develop relied upon functions built into the Greedy agent.

3.5 Expectimax Agent

In my initial plan for this project, this author proposed using an implementation of the Ant System (AS) algorithm [19], modelling the transitions between points on the board as edges in a graph and the points themselves as vertices. However, this idea was dismissed after concluding that backgammon's branching factor and randomness is not suited to AS. The Expectimax algorithm [11] was chosen to be used instead. This algorithm was designed to accommodate large branching factors and probabilistic games and thus seemed appropriate for use as a backgammon agent.

My implementation of the algorithm is shown in Algorithm 1. The agent is given a list of all boards that it can end the turn with given the initial board state and roll of the dice. Each of the 36 possible dice rolls are simulated and each board that could be obtained from the agent's board and the dice roll is scored using the evaluation function from the Greedy agent. The scores of the best boards for each dice roll are then summed together. Once this has been done for all of the agent's boards, the agent then chooses to play the move that results in the board that yields the lowest payoff for the adversary. In racing positions, the Expectimax agent operates identically to the Greedy agent.

Algorithm 1 Backgammon Expectimax Implementation

Require: resulting board states T
Ensure: chosen board b
  $b \leftarrow \emptyset$
  best_score $\leftarrow \infty$
  for each $t \in T$ **do**
    board_score $\leftarrow 0$
    for $d_1 = 1$ to 6 **do**
      for $d_2 = 1$ to 6 **do**
        max_roll_score $\leftarrow 0$
        $V \leftarrow$ all possible board states given t and roll d_1-d_2
        for each $v \in V$ **do**
          $x =$ Evaluate(v)
          if $x >$ max_roll_score **then**
            max_roll_score $\leftarrow x$
          end if
        end for
        board_score $\leftarrow$ board_score + max_roll_score
      end for
    end for
    if board_score $<$ best_score **then**
      best_score $\leftarrow$ board_score
      $b \leftarrow t$
    end if
  end for
  return b

It is worth noting that while this agent has 2-ply lookahead (it checks all of the opponent's possible moves), it has no notion of self-interest until the board is in a racing position. While in contact positions, the agent's entire purpose is to reduce its opponent's payoff as much as possible. As backgammon is not a deterministic game, the agent can experience issues when the opponent gets "lucky" dice rolls, despite doing all it can to minimise large payoffs for the adversary.

3.6 Genetic Agent

The final "intermediate" agent is based on genetic and co-evolution algorithms. In principle, genetic algorithms generate an initial population, typically a list of random values, which are then given a fitness score. The higher the fitness score of an individual, the more likely it is to be chosen for reproduction. The result of the reproductions is the next generation, which continues the cycle a predefined number of times. The individual deemed to be the fittest is then chosen as the individual to be used for whatever task it was designed for.

This is true also for the Genetic agent. 18 random values are selected, each of which corresponds to a factor in the Genetic agent's evaluation function. The genetic evaluation function has the exact same parameters and conditions as the Greedy agent's evaluator, though all conditions have been flattened (there are no nested conditional statements) so all scenarios are always considered and the whole board is always observed.

The genetic algorithm was executed with a population size of 100 and a limit of 50 generations. Each individual was given a fitness value indicative of their performance against the Greedy agent in a First-to-Five match:

$$fitness = \begin{cases} Score_i \div Score_g & \text{if } Score_g > 0 \\ Score_i + 1 & otherwise \end{cases} \quad (1)$$

where $Score_i$ is the individual's score and $Score_g$ is the Greedy agent's score. As numerous individuals had fitness values greater than or equal to 5, the author decided to implement a co-evolution algorithm, where the fittest individuals would play against each other in a single-knockout tournament. Each individual in the tournament would play another in a First-to-25 match, and the winners would progress to the next stage of the tournament. This continued until there remained only one individual, who would then be used as the Genetic agent. The process of co-evolution was useful for multiple reasons. All individuals chosen for co-evolution were shown to have comprehensively beaten the Greedy agent. One could not be certain that the weights possessed by the individual were the best though, as they could simply be the best against just the Greedy agent, by potentially exploiting some weakness in its game to form a dominating strategy. By having each individual play against each other, we know that the final individual would not only be able to consistently beat the Greedy agent, but other backgammon-playing algorithms as well.

3.7 Adaptive Agent

The first "advanced" agent was designed to be an amalgamation of the best features of each of the "intermediate"

agents. The agent differs from the previous agents, having an equity calculator rather than a move evaluator. A higher equity represents a greater chance of winning the game, whereas a better evaluation represents the quality of a move in isolation. For example, in the scenario where the player has borne off no pieces but the opponent has borne off 14, there would be a very low equity score. However, the player's move would receive a high evaluation if they could hit off the opponent's piece, even though defeat was almost guaranteed. The move evaluator compares the strength of the board after the turn relative to its strength at the start, whereas the equity calculator determines the absolute strength of the board.

The Adaptive agent's equity calculator takes in 30 weights which are assigned to various board features (Table 12), such as pip count, blot volatility, probability of a gammon and existence of a prime. This equity calculator guides the Adaptive agent throughout the midgame; each possible board has its equity calculated, and the agent chooses the best one. Should multiple boards have the joint-highest equity, the Adaptive agent looks ahead to the possible responses from the opponent and chooses the move that gives the opponent the lowest equity. This was done as a result of analysis of the Expectimax agent, which performs well by reducing the opponent's payoff despite having no notion of self-interest. The Adaptive agent improves on this by not only using an equity calculator instead of an evaluation function, but also by first seeing what move benefits itself the most and only then does it try to limit the opponent's payoff. If there are any remaining boards, one is selected randomly and played in order to mitigate any bias.

The key difference between the Adaptive agent and the agents already discussed is that this agent follows different strategies at different phases of each game and states of the match. For example, if it is losing 23-18, the agent will change its weights to be much more likely to double or play for a gammon in order to catch up to the opponent. At the first turn of each game, the Adaptive agent uses the genetic evaluation function to make the opening move. This decision was made after comparing the textbook opening moves with the opening moves made by the "intermediate" AIs as well as the Adaptive agent's midgame move selection function. The Genetic agent's success rate at choosing optimal opening and counter-opening moves was superior to the other algorithms, and hence was chosen. After spectating countless games, this author observed that, while the Adaptive agent's play in contact positions was strong, it suffered in race positions. For race positions, the Genetic agent was selected to move the checkers into the agent's home board. Once the checkers are all in the agent's home, Matussek's regression formula [20] is used to calculate the game-winning chances (GWC) of each possible move, with the move that yields the greatest chance of victory being selected.

Despite all this, the Adaptive agent was still prone to loss by gammon. Upon inspection, it was shown that the Adaptive agent would waste valuable recovery time by repositioning pieces already in its home board, rather than pieces in the outer board or opponent's home. This prevented the Adaptive agent from bearing off and thus caused losses by gammon. To remedy this, if the opponent's checkers are all in their home board and the agent had not borne off any pieces, the agent now prioritises pushing its furthest-back pieces forward in order to lose by a smaller margin.

Once all of this was completed, the same genetic training process used in the Genetic agent was applied to the weights in the adaptive midgame function's equity calculator to improve its performance against the genetic, Greedy and Expectimax agents. The individuals deemed the fittest were then co-evolved to improve generalisation against other strategies.

3.8 Supervised Deep-Learning Agent

3.8.1 Initial Designs

The supervised deep learning agent was planned to use a deep neural network that accepted the 28 points on the board as input and output a score corresponding to the strength of the board position (equity). However, the development of the agent encountered a major issue: there existed no free-to-access datasets for backgammon moves. The plan for this agent was to simply install a dataset of board positions and associated scores to train a model on. The proposed solution to this problem was as follows: generate a random board position, set the game to start at this board position and then simulate the Adaptive agent playing itself for one hundred games. The mean score for each board position was saved along with the board itself. The theory being that a board state that wins 70% of the time is a more desirable board state than one which wins 30% of the time. The dataset was constructed of such board states and mean scores and the model was trained to predict the score from 4000 out of the 5196 random board states by minimising Mean Squared Error (MSE) loss:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2}$$

where $\hat{y}_i$ is the model's predicted value and y_i is the ground truth. Despite the MSE loss being extraordinarily low on both the training and test sets, the model performed catastrophically when faced against the other agents, losing by backgammon to all.

3.8.2 Restructuring model and dataset architecture

In an attempt to remedy this, the model's architecture was changed to instead accept 289 input units, pass the information into a hidden layer of 12 units, and then output a single number to predict the equity of the board. This method of predicting the equity of a single board is known as the Relative Score Paradigm (RSP). The 289 input units represent an encoding of the board as well as some precomputed features. Each point $0 \leq i < 24$ is split into a ten-dimensional vector, where every dimension is its own input unit. Each dimension in the vector contains a binary value indicative of the number of checkers occupying the point (Fig. 1).

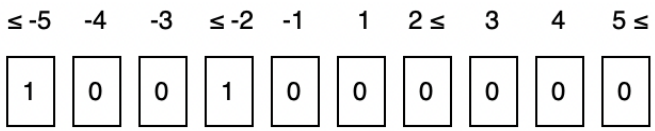

Fig. 1. Example encoding of a point on the board occupied by 5 black pieces.

The encoding is done in this way to emphasise when a point is occupied by an anchor or blot. The first and tenth dimensions cover a range of values, as once a point has over four pieces occupying it, the exact number of pieces ceases to matter. The probabilities that each of the 24 points on the board could be hit off are also precomputed and included as input units. The bars are encoded as three-dimensional vectors with binary values that represent whether there are no, 1, or greater than or equal to 2 pieces on the bar. Once the bar contains at least two checkers, the exact number becomes irrelevant. Precomputed features included as input units also include the agent's pieces in their home board, the number of points in the agent's home board and opponent's home board that are occupied by the agent, the existence of a prime, the agent and opponent's pip count, and the probability the agent's frontmost anchor can be passed by the opponent. The dataset was encoded in the same way so that the data would still be accepted by the new model. Each input unit has a value between 0 and 1 and many values are binary. This helps with weight initialisation at the start of training and prevents vastly different gradients, improving the speed of convergence.

The new model was then trained on the dataset but similarly faced heavy defeat, beating only the random agent. The data quality was not good enough to train a competent agent. Given backgammon's branching factor, one hundred games was far from the number needed to gain a reliable understanding of the strength of a position, with 10,000 games normally simulated for rollouts. Furthermore, even if every possible dice roll was simulated, the fact that the Adaptive agent still made consistent mistakes and suboptimal decisions meant that the collected data would be tainted regardless, even though it was my best agent at the time.

3.8.3 Crafting a reliable model

Having established that there were no freely accessible backgammon datasets, the author researched other deep learning models, finding the vast majority used unsupervised learning. Eventually, the author encountered Kolatacz's Medium article [21] about how he developed a program to find out what the best move in each situation was using GNUBG's[3] command-line interface (CLI). Upon further research into GNUBG, we found that it was a world-class backgammon player, with a FIBS [22] rating of 2000 and capable of 4-ply lookahead. Using Python's subprocess module, this author was able to interact with GNUBG's CLI and receive reliable equity values for any move from a given position. By generating random boards, passing them into GNUBG and extracting the moves and equities, we were able to generate the resulting boards and encode them into a new dataset that could be used to train the aforementioned model. While the dataset was being prepared, more research was conducted on the application of supervised learning in the setting of backgammon. Tesauro concluded that networks using the Comparison Paradigm (CP) tend to perform much better than networks using the RSP [23]. In the RSP, each board is given a score between -1 and 1 and is trained on MSE loss. In the CP, two boards are accepted as input, and the output is a binary representation of which board is preferred. This model is trained on Binary Cross-Entropy loss (BCE loss):

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \quad (3)$$

where N is the number of samples, y_i is the true label for the i-th value and $\hat{y}_i$ is the model's output for the sample. This not only minimises the impact of evaluation noise, but allows the network to understand the strength of positions and what makes one position more desirable than another. For a model to become proficient at backgammon, this is paramount [24].

The combination of the CP and a reliable dataset resulted in the best agent at the time of development (V1). However, there remained identifiable weaknesses to the agent's game, including exposing blots to make hits in the agent's home, but most notably its decision-making in disengaged racing positions. In these positions, both players have moved all of their checkers past the opponent's, and so there is no degree of contact. This renders anchors and primes obsolete as there is no opponent checker to hit one's pieces off, and vice versa. However, given that the bulk of the training data contained contact positions, the agent tended to prioritise building anchors and primes instead of moving pieces into the home board, or even bearing pieces off. To remedy this, approximately 8000 racing boards were collected and split into mid-board and bear-off positions. The bearoff positions were board states where all of the player's pieces were home, and the mid-board positions were all other race positions. Additionally, pre-generated contact positions were separated. Using the same paradigm and network, 3 new models were trained (one for each type of position). The agent would identify the board's position type and use the appropriate model to select the move to play (V3). This noticeably improved the agent's decision making in racing positions, but bizarrely the performance of the model in contact positions worsened, and the new agent lost consistently against V1. Considering that each of the new models were individually trained on a smaller dataset than the first model, this author considered the possibility that the new models may have overfitted to the training data. Consequently, we retrained the models with a reduced epoch count to some success (V3.5); outperforming V3 but not V1. Version 4 of the agent discards the contact-specialist model in favour of V1 and uses V3.5's racing specialists. While ultimately much closer in performance to V1, it was still inferior. As a result of this performance boost, V4.5 was created, using V1 for contact positions and V3's racing specialists for disengaged positions. This was the first model to outperform V1 and was thus selected to act as the Deep agent.

TABLE 1
Descriptions of the Iterations of the Deep Agent

Version	Description
1	A single general-purpose model
2	V1 with 200 training epochs rather than 100
3	3 specialist models for different types of position
3.5	V3 with 50 training epochs rather than 100
4	Uses V1 for contact positions, else V3.5
4.5	Uses V1 for contact positions, else V3

3.9 Deep Reinforcement Agent

Given Tesauro's overwhelming success [1] it seemed natural to implement Temporal Difference-Lambda "TD(λ)" learning with $\alpha = 0.1$ and $\lambda = 0.7$ [25]. The network took an input of 198 units and had a fully connected hidden layer of 80 units, with 6 output units. These output units represented a regular win, a regular loss, a gammon win, a gammon loss, a backgammon win, and a backgammon loss. In the training loop, the estimated value of each state was calculated according to Equation 4.

$$V(S) = \begin{pmatrix} \text{P(Regular Win)} \\ \text{P(Regular Loss)} \\ \text{P(Gammon Win)} \\ \text{P(Gammon Loss)} \\ \text{P(Backgammon Win)} \\ \text{P(Backgammon Loss)} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \\ 2 \\ -2 \\ 3 \\ -3 \end{pmatrix} \tag{4}$$

While backgammons are a rare occurrence, the decision was made to include them in the output vector to ensure that the agent capitalises on opportunities to win by backgammon, but to a greater extent to avoid loss by backgammon wherever possible. The Deep agent, for example, loses games by backgammon where it could have easily prevented the 3-point loss in favour of just a loss by gammon. The 198 input units were an encoding of the board similar to that used in the Deep agent, but simpler and with no precomputed features. Rather than each point being encoded into 10 units, they were instead encoded into 8 units using Algorithm 2 with the bars encoded as $bar = min(1, x/2)$, where x is the number of pieces on the player's bar. The borne-off pieces for each player were simply encoded as $borne\_off = \frac{x}{15}$ where x is the number of pieces the player had borne off. Whoever was on roll was also added as an input via a binary encoding. Handcrafted features were not included to improve the speed of each time step. To compensate for this, 80 hidden inputs were used instead of 12. The expectation was that eventually the model would be able to identify simple features, such as blots being bad, and use the hidden units to represent these features.

The first 200 training games (episodes) were played against the random agent. At this point the reinforcement agent was already consistently beating the random agent and had learnt some useful features of backgammon. It had learnt to push pieces towards the home board, to hit pieces and even to bear off and make a point rather than greedily bearing off twice and exposing a point while still in a contact position. However, when in a racing position it still chose to do this, so had not yet learnt the different between the two position types. While this is a very rudimentary understanding of the game, the amount learned in such a small number of games was staggering. Content that the model was indeed learning, the training cycle was restarted with the model now being trained by self-play. Both players, black and white, viewed the board as if they were the white player to ensure continuity between states and prevent confusion when updating the model's weights. The model was left to train overnight, and after 80,000 episodes there was no noticeable improvement. Checkpoints were made every 5000 episodes, with some being improvements on the initial 200-episode model, but most being worse. We reinitialised the model to now have $\alpha = 0.01$ and ran the training process again. To hedge my bets, this author uploaded the model and environment to the Durham University NVIDIA CUDA Centre (NCC) GPU system and set $\lambda = 0, \alpha = 0.1$ like Tesauro did in subsequent iterations of TD-Gammon [7]. After 124,000 episodes, the model trained on my local machine was tested against the intermediate agents with tremendous success, surpassing all. The success was so great that it also beat the Adaptive agent in head-to-head matches! The model had started to learn good bear-in positions and bear-off strategies, it would expose blots but when risk was minimised and would double-hit the opponent wherever possible. The agent was not tested against the Deep agent at this time to preserve processing power for further training of the model, but 100,000 episodes later it was given the opportunity to play against the Deep agent, once again to success. At 224,000 episodes (and potentially even earlier, though this could not be verified) the Reinforcement agent was already my strongest model. In comparison, the $\lambda = 0$ model performed horrifically, sometimes not even getting a point off the Greedy agent

Algorithm 2 Point Encoder

Require: point value p
Ensure: 8D point encoding p_e
  $p_e \leftarrow \vec{0}$
  if $p < 0$ **then**
    for $i = 0$ to 3 **do**
      if $p < -i$ **then**
        $p_e[i] = 1$
      end if
    end for
    if $p < -3$ **then**
      $p_e[3] = min(1, \frac{|p|-3}{2})$
    end if
  else if $p > 0$ **then**
    for $i = 4$ to 7 **do**
      if $p > i - 4$ **then**
        $p_e[i] = 1$
      end if
    end for
    if $p > 3$ **then**
      $p_e[7] = min(1, \frac{p-3}{2})$
    end if
  end if
  return p_e

in 100 games. However, the model's performance started to decrease after approximately 500,000 episodes. It had learned that backgammons were the best way to maximise PPG differential and so formulated a strategy to win by backgammon as often as possible. An error in the reward function was found which failed to properly check the criteria for a backgammon (it only checked that the loser had pieces on the bar or in the victor's home, failing to check it was also a gammon), meaning that the plays it kept making were incredibly risky. Consequently, it would lose consistently when playing against a relatively strong opponent, as they would be punished for their risky plays. The reward function was fixed and the model was retrained (V2) to astounding success, overpowering the Greedy agent in under 2000 episodes. The agent had no lookahead in the training loop to save computation time, but would have two-ply pruned lookahead when playing opponents (Fig. 6). The agent would calculate the estimated value (equity) of each board state it can make in its turn. It would then prune all boards with equity less than $max(\text{equities}) - 0.16$ to save computation time and then use a tweaked version of Algorithm 1 to calculate the mean equity after the opponent's turn (rather than using the Greedy agent's evaluation function the tweaked version uses the model to calculate equity). The board with the highest estimated equity after the opponent's move is selected.

Table 2 contains the main structural differences between the Reinforcement agent's model and TD-Gammon's model. The logic behind the difference in the number of output units and the value of α and λ have already been discussed. Rectified Linear Unit (ReLU) activation [26]: ReLU(x) = $max(0, x)$ was used instead of sigmoid activation: $\sigma(x) = \frac{1}{1+e^{-x}}$ to prevent exploding gradients. Additionally, the input range remains between 0 and 1 inclusive, whereas in TD-Gammon, *most* input units were between this range, but the bar encoding could reach a value of 7.5 should all of the player's pieces be on the bar. Also, rather than the fourth bit representing each point for a player (that is, the fourth bit for player -1, the eighth bit for player 1) being clipped at a maximum value of 1 for $x \geq 5$, the value of the bit monotonically increases with the numbers of pieces on the bar, using the formula $\frac{x-3}{2}$. For example, if there were 7 white pieces at a point it would be encoded as $(0, 0, 0, 0, 1, 1, 1, 1)$ using the Reinforcement agent's encoding scheme and as $(0, 0, 0, 0, 1, 1, 1, 2)$ using TD-Gammon's encoding scheme [1][25]. Once a point is occupied by at least 5 pieces, the specific number ceases to matter as the general principle is that stacking points is bad practice.

TABLE 2
Reinforcement Agent and TD-Gammon Structure Comparison

	Agent	TD-Gammon
Output Units	6	4
Activation	ReLU	Sigmoid
α	0.01	0.1
λ	0.7	0
Input Range	[0, 1]	[0, 7.5]

The model was saved every 500 episodes for testing against the Greedy agent. Every 1000 episodes it would be tested against the Genetic agent and then every 5000 episodes it would be tested against GNUBG on beginner difficulty (Table 3). These tests were done with no lookahead as to save computation time and allow for more training. The results obtained therefore showed the trend in the performance of the agent, but not its true performance (Fig. 7, Fig. 8, Fig. 9). The model did appear to converge rather early, but was allowed to train for 400,000 episodes in case any sporadic improvements occurred. The best performing episodes were saved and tested against each other with a 2-ply lookahead built-in to determine which would be selected to act as the agent.

TABLE 3
GNUBG Difficulty Settings.

Difficulty	Description
Beginner	Novice-level play
Casual Player	Strong novice/ weak intermediate-level play
Intermediate	Strong intermediate-level play
Advanced	Strong advanced-level play
Expert	Master-level play
World Class	Strong master-level, on par with World Champions
Supremo	Very strong master-level play
Grandmaster	Superhuman-level play
4-ply	Use only for move analysis. Perfect play.

3.10 Doubling

3.10.1 The Random Agent's Doubling Strategy

With the intention of preventing the random agent from excessively doubling opponents, the decision was made not to allow the random agent to offer a double. However, if doubled, the agent randomly decides whether to accept or reject the offer, without knowledge of the board state or match score.

3.10.2 Intermediate Doubling Strategies

Keeler and Spencer proved that the optimal time to double in backgammon was when the player had an 80% chance of victory [15]. The player should also decline a double if their chance of victory is below 80%. After writing a simple equity calculator (comprised of much fewer parameters than that used in the Adaptive agent), the author simulated hundreds of games in which the Genetic agent would play itself. The equities of all the boards possessed by the winner and the loser were recorded and separated. The resulting values were modelled as a normal distribution. The point for offering a double was chosen to be the 80^{th} percentile of winning equities, and the point for rejecting a double was chosen to be the 20^{th} percentile of losing equities. This strategy is used by all of the "intermediate" agents.

3.10.3 The Adaptive Agent's Doubling Strategy

The Adaptive agent initially had a very similar doubling strategy to the "intermediate" agents; the winning and losing equity values were modelled as a normal distribution and the 20^{th} and 80^{th} percentiles were taken as doubling points, though the equity calculator was the one used in its

move selection, rather than the simplistic implementation used by the "intermediate" agents. Eventually, the doubling strategy was modified. The genetic algorithm used in training the Genetic agent and Adaptive agent's advanced equity calculator was again utilised to determine the best doubling points. The best performing individuals were once again co-evolved against each other to ensure that the doubling strategy was not just exploiting the "intermediate" agents' doubling points. These doubling points are used until the game is in a racing bear-off position, where the Adaptive agent then uses the GWC calculated from Matussek's formula [20] and applies it to a classical formula [15] to ensure optimal doubling in these positions.

3.10.4 The Deep Agent's Doubling Strategy

Again there existed no freely accessible dataset for cube actions. However, when using GNUBG [3], one can also receive hints for cube actions. The program outputs the equity before doubling, the equity after doubling should the opponent accept, and the equity after doubling should the opponent reject the offer. The program then suggests what the correct decision is based on these outputs. For example, if the player's equity after the opponent accepts their offer is less than the equity before doubling, the player should not offer a double. The program also suggests what the opponent should do if the player doubles them. Using this, the author created a dataset of 19,000 encoded board states and the correct cube action with binary-encoded outputs. In the offer dataset, an expected output of 1 meant that the player should double and an expected output of 0 meant that they should not. In the accept dataset, an expected output of 1 meant that the player should accept the double, with an expected output of 0 meaning that the player should reject the double. Two models were then trained; one for accepting or rejecting a double and one for doubling or not doubling their opponent. The models' architecture included 289 input units, a hidden layer of 12 units, and a single output node. The input vectors were the result of the same encoding algorithm used to select moves. Although the initial results looked positive, there were some apparent and extreme flaws in the strategy. On numerous occasions where the author was playing against the Deep agent, the agent would double the author despite almost being guaranteed defeat. From observation and analysis of 10 woeful cube decisions, the author noted that there were always two things in common: the author had blots exposed and pieces borne off. Most of these boards were in race positions, so the exposed blots did not matter and in fact gave a better position for bearing pieces off. The author concluded that the model had realised that, in general, the opponent having exposed blots meant that their position was weak and hence the agent should double. Consequently, 2 additional networks were implemented, one for accepting or rejecting the cube in race or bearoff situations and one for doubling or not doubling the opponent in race or bearoff situations. New data was collected and approximately 10% of all board positions in the race/bearoff dataset contained pieces that had already been borne off to represent the significance borne-off pieces have in a race. Additionally, the structure of the networks were different for contact doubling positions. Instead of 289 input units, there were 265. 2 units were added showing the proportion of pieces borne off and the units to do with hitting, primes, and passing blockades were removed. The original 289-unit network for contact cube decisions was also changed to accept 291 input units to explicitly represent the number of borne off pieces.

TABLE 4
Descriptions of the Iterations of the Deep Agent's Doubling Strategy

Version	Description
1	289-12-1 structure. 1 model for accepting, 1 for doubling
2	291-12-1 with 1 model for accepting and 1 for doubling
3	V2 for contact decisions, 265-12-1 for race/bearoff decisions

3.10.5 Reinforcement Agent Doubling Strategy

There were 3 initial considerations for this agent's doubling strategy: train a doubling network using the same principles that trained the move selection model, use the Deep agent's doubling models, or utilise the move selection model and combine then with theoretical doubling formula [15][8]. Consideration 1 was the preferred choice at the start, but proved to be extremely exploitative of fixed doubling points and gained no generalisability. The Deep agent's doubling strategy had already undergone a lot of baseline testing and its degree of success had already been determined. The final consideration was evaluated against the combined train and test set used by the Deep agent's models to determine its accuracy and generalisability (Table 10) and thus the Deep agent's strategy was chosen to also be used by the Reinforcement agent. Note that the same formula was used for racing and contact positions, rather than interpolating the racing tables present in Zadeh and Gobliska's paper [8], though they would have been included should the model have shown promise in contact positions.

4 Results

To totally measure the performance of each agent, many games were simulated against each other both with and without the cube in action. This aimed to demonstrate the strength of the agents' move play and their cube actions.

4.1 Cubeless Performance

As shown in Table 5, the Genetic agent outperforms the other intermediate agents, winning 54.8% of games against the Greedy agent and 56.9% of games against the Expectimax agent, with a mean point per game differential of +0.133 and +0.331 respectively. The Greedy agent then outperformed the Expectimax agent with a narrow win rate of 51.5%, although interestingly with a large PPG differential of +0.148. This differential is larger than that of the Genetic agent against the Greedy agent, despite the 3.3% win rate disparity. Their head-to-head results do not extend onto their results against other agents, with the Expectimax agent outperforming both the Greedy and Genetic agent when facing the Deep agent (Fig. 2) and only falls 43 points behind the Genetic agent when facing the Adaptive agent (Fig. 3) while boasting a better win rate.

TABLE 5
Cubeless Head-to-Head Intermediate Agents Results. Game Wins are shown in the Upper Triangle, Cumulative Scores are shown in the Bottom Triangle.

Agent	Genetic	Expectimax	Greedy
Genetic	X	569-431	5478-4522
Expectimax	567-898	X	2436-2574
Greedy	6068-7401	3157-3898	X

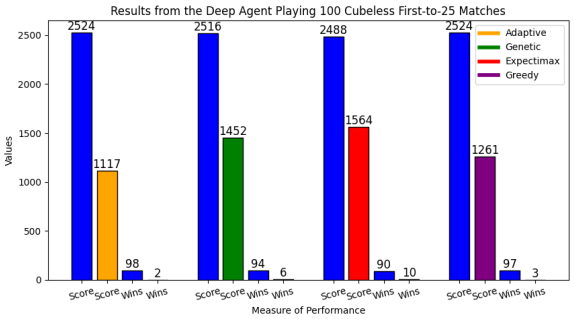

Fig. 2. Performance of the Deep agent (blue) vs the Adaptive and intermediate Agents in 100 cubeless matches.

The Deep agent convincingly outperformed all of the intermediate agents as well as the Adaptive agent in head-to-head matches. Additionally, the agent's move selection models performed well against unseen data (Table 6).

TABLE 6
Performance of Supervised Models on Unseen Data

Position	Deep Accuracy	Neurogammon Accuracy
Contact	66%	69%
Bear-in	80%	63%
Bear-off	95%	83%

A bear-in position relates to moving a piece into the player's home board. Given that the Deep agent was not trained on data representing this exact position, the value entered into the table is the accuracy of the mid-board race model, which is the most similar to the data Neurogammon's bear-in model was trained on [2], though it should be noted that it is likely some positions faced in the contact model by the Deep agent may also have been considered bear-in positions by Tesauro. The accuracy values have also been rounded to integer percentages in alignment with the presentation of the results in Tesauro's paper [2].

Combining the best aspects of each intermediate agent as well as adding a custom equity calculator, selective 2-ply lookahead, and a bear-off algorithm designed to maximise game-winning chances (GWC) led the adaptive agent to overpower the intermediate agents. It also boasted a win rate of 60%, 75% and 54% against the Genetic, Greedy, and Expectimax agents respectively (Fig. 3), showing that it not only has a strong PPG differential, but is also capable of winning the tournament-style matches. The Adaptive agent is no match for the Deep agent, however, having won only 2 matches against it. Peculiarly, the Deep agent had the most convincing win against the Adaptive agent, despite the Adaptive agent outperforming each of the intermediate agents in head-to-head matches.

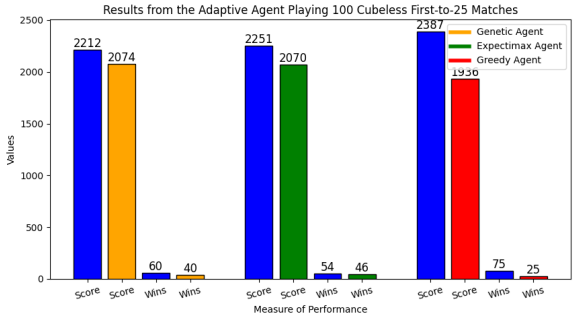

Fig. 3. Performance of the Adaptive agent (blue) vs the intermediate agents in 100 cubeless matches.

Official data about the random agent was never collected as, in the small samples of head-to-head matches, it would always lose by gammon or backgammon to any other opponent, even to the reinforcement agent after only 200 episodes.

The Reinforcement agent was able to beat the Greedy agent in under 2000 episodes and without lookahead. It would then go on to beat the Genetic agent and GNUBG beginner mode in under 5000 episodes, also without lookahead. With the introduction of the 2-ply selective lookahead, the agent became vastly superior to all other agents in this project (Fig. 5) and was thus primarily tested against various GNUBG difficulty settings (Table 3) [3] to identify its level of play in human terms. The Beginner and Casual Player difficulty settings were no match for the Reinforcement Agent, and the Intermediate setting also faced heavy defeat. Eventually, the Reinforcement agent met its match against the advanced difficulty setting (Fig. 4). Interpolating between the skill level of the Intermediate and Advanced difficulties of GNUBG suggests that the Reinforcement agent operates at weak advanced- or advanced-level move-play.

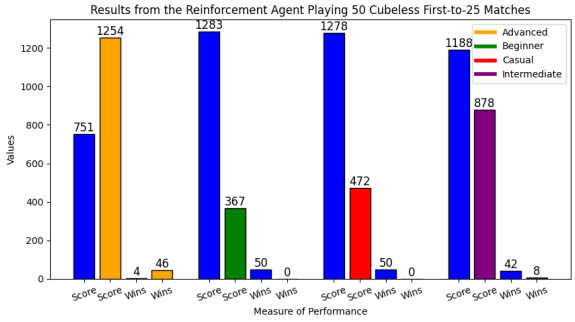

Fig. 4. Performance of the Reinforcement agent (blue) vs various difficulties of GNUBG in 50 cubeless matches.

4.2 Cubeful Performance

Performance of each agent's doubling strategy is graded on, where applicable, accuracy on unseen board states, ratings from human trialists, and results in matches against

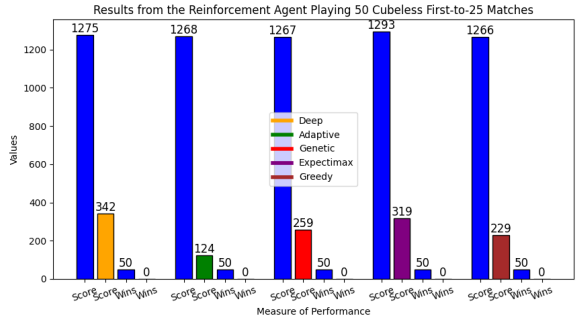

Fig. 5. Performance of the Reinforcement agent (blue) against the other agents in 50 cubeless matches.

human players. It is not good practice to compare doubling strategies by just head-to-head performance as one strategy may exploit another. The results of human trials are wildly

TABLE 7
Cubeful Results Against Intermediate Level Human Trialists.

	Deep	Adaptive	Genetic	Expectimax	Greedy
A	25-23	25-7	26-1	27-3	64-8
B	22-25	28-12	27-10	25-23	128-11
C	19-25	26-16	27-6	23-26	71-0

surprising. The Expectimax agent performed the worst against other agents in cubeless matches and possesses a very simple doubling strategy, but is one of the best against human players. The Deep agent outperforms the other agents and even some humans, while the Adaptive agent ranks similarly against humans as it does in head-to-head agent games. The trialists were very impressed with the Deep agent, with its move-play considered almost perfect and the cube play exceptionally strong (Table 8).

TABLE 8
Mean Ratings of Agent Performance Given by Human Trialists on a Scale From 1-10 Based on Their Experiences in the Trial.

Action	Deep	Adaptive	Genetic	Expectimax	Greedy
Moves	9.5	7	5	6	4
Cube	9	8	3	5	3

The Deep agent's various doubling models performed well against unseen board states, most notably when offering the cube (Table 9).

The other agents' strategies were tested against a joint dataset made up of the Deep agent's test and training sets. All agents perform well in race positions and generally offer the cube accurately, though the Adaptive and intermediate agents struggle most when deciding whether to pass or take a double. The intermediate agents' contact accepting strategy is awful, performing even worse than the Random agent's (which randomly chooses whether to take or drop a double, hence attaining an accuracy of 50%). The unused strategy for the Reinforcement agent performed reasonably well in contact positions but horribly in race decisions, especially when offered the cube (Table 10).

TABLE 9
Performance of Deep Agent's Doubling Models

Model	Test Accuracy
Contact Accept	74.59%
Contact Offer	93.38%
Race Accept	89.32%
Race Offer	92.31%

TABLE 10
Accuracy of Doubling Strategies on Randomly Generated Boards.

Position Type	Reinforce (unused)	Adaptive	Intermediate
Contact Accept	69.12%	63.51%	39.29%
Contact Offer	77.98%	85.06%	75.63%
Race Accept	37.53%	86.89%	84.06%
Race Offer	80.12%	86.80%	74.81%

4.3 Opening Move Decisions

Apart from some opening moves and moves in bear-off situations, it is unlikely the Deep agent has seen any of the boards in the training set again. Although, it has been able to generalise well to unseen boards, it most notably blunders on the opening move with 6-1 roll. The best move (and most obvious move, even for novice players) is to play 13/7, 8/7, which creates a 3-wall and immediately starts blockading the opponent's furthest-back pieces. However, the deep agent instead plays 13/7, 24/23. This moves one checker into the 7-point, where it is relatively protected and can be made into an anchor later, while also starting the progression of the far-back pieces to prevent a losing back-game. Given the strength of the 13/7, 8/7 move, however, this is considered a blunder, and throws away an immediate advantage. Table 11 compares the advanced and intermediate agents' error in making opening moves [3] (the Adaptive agent was excluded as it uses the Genetic agent's move-selection function for opening moves). These optimal opening moves were calculated by simulating 36x36x36 games in which the dice rolls for the opening move and the following 2 moves were enumerated to ensure all possibilities in the first 3 moves were considered. Opening moves that resulted in a rollout equity of less than -0.05 were excluded, hence some values in Table 11 are lower bounds of the error, which could actually be much greater. Save 2 anomalous blunders, the Deep agent is incredibly accurate with opening moves. On the 4-2 roll, the Deep agent decides to slot a piece on the 11-point, giving it a mere $\frac{1}{18}$ chance of being hit off and again advances its furthest back piece to try and prevent a losing back game later on. This is not an awful move, but is much worse than the best move of 8/4, 6/4 which creates an anchor in the home board and reduces the chance of the opponent's furthest back pieces escaping the blockade. The intermediate agents perform identically. They all build an additional anchor where possible, which sees them play the best moves in the same roll that the Deep agent blunders. Given that many of their chosen opening moves result in an equity of less than -0.05 (a losing position), their mistake count is much greater than the Deep agent's and it is possible that their total error is greater than the Deep agent's

as the values presented are just lower bounds. It is worth noting that the intermediate agents were not designed to give special consideration to opening moves, so it is very interesting that they choose identical opening moves despite their different move-selection algorithm. Their performance could be easily improved by hardcoding the best moves to make on the opening roll. The Reinforcement agent has near-flawless opening move selection, likely as a result of starting approximately half of the hundreds of thousands of games it played and identifying a trend with opening moves. Additionally, its 2-ply search minimises the chance of the opponent winning rather than the quality of the move in a vacuum (like the Expectimax agent) and is able to consider its own interest as well as the opponent's.

TABLE 11
Equity Lost Between Best Opening Move and Move Agents Choose. A Value of 0 Means the Agent Chose the Best Move.

Roll	Reinforce	Deep	Greedy	Genetic	Expectimax
1-2	0.0470	0.0043	0.0344	0.0344	0.0344
1-3	0	0	0	0	0
1-4	0.0372	0	0.0372	0.0372	0.0372
1-5	0	0	≥ 0.0556	≥ 0.0556	≥ 0.0556
1-6	0	≥ 0.1535	0	0	0
2-3	0.0401	0.0001	0.0401	0.0401	0.0401
2-4	0	≥ 0.1734	0	0	0
2-5	0	0	≥ 0.0532	≥ 0.0532	≥ 0.0532
2-6	0.0183	0	≥ 0.0566	≥ 0.0566	≥ 0.0566
3-4	≥ 0.0625	0.0065	≥ 0.0625	≥ 0.0625	≥ 0.0625
3-5	0	0.0442	0	0	0
3-6	0.0143	0.0404	0.0416	0.0416	0.0416
4-5	0.0010	0	0.0010	0.0010	0.001
4-6	0	0.0602	0	0	0
5-6	0	0.0387	≥ 0.1045	≥ 0.1045	≥ 0.1045
Total	≥ 0.2204	≥ 0.5213	≥ 0.4867	≥ 0.4867	≥ 0.4867

5 EVALUATION

From first glance at Table 6 it may seem that the Deep agent outclasses Neurogammon. One must consider, however, that the vast majority of positions in backgammon are contact and thus Neurogammon's better performance in these positions may cause it to beat the Deep agent more often than not, as by the time the Deep agent is in the positions it performs best in, it could already be too late. Additionally, given the hand-crafted nature of Neurogammon's dataset, it is possible that the unseen boards in the test sets were of a greater variety than those present in the Deep agent's test sets and potentially differed more from their respective training sets. Regardless, the 12% difference between the bear-off models is staggering. Having played multiple games against the deep agent, this author can attest to the quality of the agent's bear-off decision making; having lost every single equal-footed bear-off race against the agent. While it was initially surprising to see versions 2-4 of the Deep agent perform worse than V1, the author concluded that the initial model, having access to all available data, was able to plan ahead much better than the specialist networks. For example, the initial model would know which positions are optimal for bearing off pieces later on in the game and could move checkers into these positions accordingly. However, the contact-specialist model has little concept of bearing off (only in the somewhat unlikely scenario where all of the agent's checkers are home, but some of the opponent's pieces are on the bar or the agent's home still), so it is unable to move checkers into desirable late-game positions early enough. Therefore, when used in conjunction with the race specialist networks, it was able to achieve greater performance than previous models. To improve its accuracy when making an opening move (Table 11) a new specialist model could be trained so that the agent can better distinguish between a strong board state in the midgame and a strong opening board state. The Deep agent's doubling strategy is also fantastic. During play against the agent, this author was left confused when offered the cube, only to go on and lose the game. However, the doubling networks only look at the board, not the current score of the match. This led the agent to pass doubles even when 1 point from defeat, guaranteeing a loss. As a result of this observation, all agents were changed to accept a double if their opponent was 1 point from victory. Human trialists agreed that it was fantastic, with their only criticism that it did not recognise when it had better move play than the opponent and hence double more often, and conversely when the opponent had better move play than the Deep agent, it did not double more conservatively. It would seem that including the match score in the data could improve the Deep and Reinforcement agents' doubling strategy, though an alternative source to GNUBG would need to be found.

The Deep agent suffers when threatened with loss by gammon or backgammon. The data collected only used cubeless equities, and so it sees that it is guaranteed a loss regardless and thus makes poor move decisions, sometimes causing it to lose by backgammon when it could easily have been only by gammon. This could be solved by using cubeful equities, so it can better distinguish between a loss by gammon and a loss by backgammon. This further supports the theory that maximising point per game differential instead of win rate (the Deep agent had already lost, so effectively "gave up" rather than fighting to minimise point loss) is the better strategy for tournament-style matches. Indeed, the singular human trialist who was able to beat the Deep agent stated that this was largely only because the Deep agent did not stop itself from losing by gammon when it had the opportunity.

The Adaptive agent took a remarkable amount of hand-crafting and analysis to reach its current level of performance. Even when initially taking the best part of each agent and combining them, it still underperformed against the genetic agent. This was partially as a result of the weights chosen for each parameter in the equity calculation function and indeed the parameters selected - while all those selected are valid, there are so many factors in a game of backgammon that it is very possible this author did not include them all. Berliner [4] showed it was possible to build a strong backgammon program using only heuristic functions, but to do so he had to seek the advice of world-class backgammon players and perfectly tune and craft each detail. Hence, this author chose to genetically tune each weight given the success of the Genetic agent in comparison to the Greedy agent, which it was based on. This drastically improved the performance of the Adaptive agent, to the point where it now consistently beats the Genetic agent. It should be noted that in the 48 years since the creation of

Berliner's BKG program, no other heuristic algorithm has been able to come close in terms of performance despite the advancement in the level of backgammon since the release of TD-Gammon [1], though this is likely because it has been shown to be much more practical to use neural nets than to toil other fine-tuning heuristic evaluation functions. Combined with the strength of accurate equity estimations and genetically-tuned doubling points, the Adaptive agent similarly possesses a strong doubling strategy. Although the margin of victory against the Genetic agent in cubeful matches is lower than in cubeless matches, the human trialists rated the Adaptive agent's strategy as better than the intermediate agents' strategies, and it performs noticeably better on the set of randomly generated boards (Table 10). This suggests that the intermediate agents' strategies perform well against the Adaptive agent's doubling strategy but do not generalise well against other strategies.

The Genetic agent was always going to outperform the Greedy agent as that was quite simply what it was trained to do. Flattening the conditions in the evaluation function and then genetically choosing the parameter weights gave the individuals huge room to explore the state space and generalise the overall strategy, while also aiming to instil a deeper understanding of the board; what makes a position good or bad? Additionally, the co-evolution tournament after the genetic cycle further increases the generalisability of the Genetic agent's strategy, minimising overfitting to the Greedy agent. Given that the Expectimax agent either minimises the Greedy agent's expected payoff, or indeed plays the same way as the greedy agent, it is similarly expected that the genetic agent would beat it as well.

The Expectimax agent's results show great promise in the idea of minimising opponent payoff to win games, even in the stochastic environment of backgammon. The suggested cause for its bizarre win rate-to-PPG ratio is the randomness presented by the dice rolls. This randomness means that it is very difficult for the Expectimax agent to always prevent the opponent from making a good move, as if even 1 out of the 21 valid combinations occurs, the opponent agent may suddenly go from being blockaded to being on for a gammon. Additionally, the Expectimax agent minimises the Greedy agent's evaluation function of the next move. If it were to instead minimise the equity, for example, it may find itself in better positions as it would be minimising the chance of the opponent winning the game, not the chance of the opponent making a good move. The core issue is that the Expectimax agent is only as good as the function it is trying to minimise and so cannot outperform the Greedy agent, as the Greedy agent knows what moves it can make and makes the best ones, whereas the Expectimax agent sees what moves the Greedy agent *could* make with each dice roll. One could argue that the Expectimax agent is strictly dominated by the Greedy agent. It tends to perform comparatively better than other agents when facing stronger opponents at least, showing that it is able to generalise better than the other intermediate agents.

All 3 intermediate agents possess the same doubling strategy. While the theory behind the strategy is sound, the weak equity calculator used to determine the doubling points can cause the agents to reject a double 3 turns in. Given the stochastic nature of backgammon, one should never be passing a double this early. Their shared strategy offers doubles sensibly (Table 10). It is also interesting to note how the human trialists held the Expectimax's doubling strategy in higher regard than the other intermediate agents' strategy, despite them being one and the same. This shows the strength of the relationship between cube actions and move-play, where the Expectimax agent's cube actions were deemed to have made more sense than the Genetic and Greedy agents' cube actions simply because its move play was better.

5.1 Reinforcement Agent Strengths and Weaknesses

The Reinforcement agent is clearly the strongest agent presented in this project. Its move play is at an advanced level and uses 4 highly-accurate models to make doubling decisions. From observation and experimentation, this author has witnessed it play tactical backgames to success when it had appeared there was no hope of victory and play opening moves to perfection. Its bear-off strategy is very good, but not as strong as the Deep agent's. The main weakness of this agent is its *absolute* accuracy in predicting the actual outcome of the game. From the results of the matches against GNUBG's intermediate and advanced mode, we know that the model's expected payoff has a mean error between 0.015 and 0.03. This error seems to be consistent across all evaluations, so it tends to always select the best move in each scenario, but causes it to be an unreliable source of information for making doubling decisions. This is likely as a result of not having played enough games for the outcome predictors to be fully tuned, but enough for the performance and strategy to have converged.

5.1.1 Potential Improvements

The agent could be extended to a 3-ply search (a 4-ply search would be too slow to play against humans) to better evaluate the long-term rewards and perform better in bear-off races (a specialist model could also be implemented to improve performance in these positions). Additionally, increasing the number of hidden units could allow the model to identify and represent more features to indicate position strength. Perhaps it could be worthwhile to slow down the speed of computation and include precomputed features in the input units, like in the Deep agent. While episodes would take longer, the model could learn quicker and would likely grasp a finer understanding of the game, which could improve its accuracy and move play in the long term, especially when you consider the level of play it was able to obtain without any precomputed features. Furthermore, although the nature of dice rolls forces a degree of exploration, the model selects the action according to its policy every time during training. This could be what led ReinforcementV2 to converge so quickly, so perhaps implementing ϵ-greedy exploration [27] or a method that slowly reduces the chance of a random move being selected the more episodes the model plays would allow for more optimal strategies to be found at the cost of slower convergence. Although the doubling strategy is very strong, it could also be improved to consider the match score and the skill disparity between the opponent and the agent. Additionally, interpolating Zadeh and Gobliska's race GWC table [8] would likely

improve the agent's doubling decisions in race positions. Additionally, if the above changes to the model's structure and training process improve the model's absolute accuracy in predicting the outcomes of games, reimplementing Zadeh and Gobliska's formulas could ensure optimal doubling for contact positions.

6 Conclusion

Backgammon is a game that appears to be too complex to be played competitively by heuristic algorithms without expert oversight and rigorous fine-tuning. As evidenced by the results, the agents that utilise neural networks and were trained on thousands of board states are able to develop a knowledge of the game much more fine-grained than a heuristic algorithm can practically obtain. The findings of this project also demonstrate the importance of focussing on maximising the PPG differential instead of the win rate, with the Deep agent occasionally missing opportunities to win by gammon or backgammon and also missing opportunities to avoid loss by gammon or backgammon as a result of being trained on cubeless (rather than cubeful) equities. Although these decisions do not change whether or not the agent wins or loses the game, they do change the number of points won or lost in a game, which resultantly affects the overall outcome of the match. One could train a supervised learning model on cubeful equities of the same board states used to the train the Deep agent and play the resulting cubeful agent against the same opponents the Deep agent faced to see if these mistakes are indeed as a result of the use of cubeless equities or if they are instead a result of model failing to generalise well to guaranteed win/loss situations (the agent was not trained on these positions as all board equities were identical and thus there was no way to determine the better board). The Reinforcement agent demonstrates not only the strength of self-play and reinforcement learning in the space of two-player zero-sum games, but specifically in the context of maximising backgammon PPG differential. By including the 2 additional output units for backgammons, the agent learns to fear loss by backgammon and lust for victory by backgammon. Unsupervised learning appears to be much more powerful of a learning methodology than supervised learning when trying to solve complex games like backgammon as there is no built-in skill ceiling (a supervised learning model can only become as good as the data it is trained on). The fact that Version 2 of the Reinforcement agent was able to attain such strong levels of play with a rudimentary board encoding and no pre-computed features is a fantastic accomplishment, especially when compared to TD-Gammon 1.0 [1][25], which only achieved strong intermediate level-play with a similar board encoding, though did not use 2-ply lookahead. One can speculate that an agent trained using an encoding of the board that included precomputed features (blot exposure, pip count, et cetera) could obtain master-level play. This author has begun experimentations using the same board encoding for the Reinforcement agent as was used with the Deep agent, ϵ-greedy exploration, and 160 hidden weights. Preliminary results for ReinforceV3 are already very promising, beating the Genetic agent in only 1800 episodes (faster than ReinforceV2 beat the Greedy agent, which has been shown to perform worse than the Genetic agent) and after 4000 episodes only loses 0.28PPG against ReinforceV2 with no lookahead (note that ReinforceV3 also has no lookahead in these benchmark tests), though the computation time is much longer and more data is required before any judgement can be made. However, these initial observations are extremely promising, and one can assume ReinforceV3 will quickly overtake ReinforceV2 in terms of performance and accuracy when estimating the outcome of the game.

Genetic algorithms have been shown to have great success in giving agent A the ability to beat agent B, and indeed the other agents that agent B beats. However, once exposed to stronger agents that are able to also beat agent B, one can see that the ability of agent A to generalise is low (the Adaptive and Genetic agents tend to perform worse against stronger agents compared to agents that they beat in head-to-head matches). In this project, co-evolution was used to try and improve the ability of an agent to generalise as much as possible, to some success. The method of training a model by self-play is a form of co-evolution in itself and yielded great success with the Reinforcement agent, which has been shown to generalise exceptionally well to many different strategies. While overall generalisability across all strategies is the goal, it must be said that a strategy that can be strictly dominated by another strategy is very undesirable, regardless of how it performs against other agents. That is why the lookahead feature from the Expectimax agent is used in other agents (with the improvement in performance shown in Fig. 6), but is not the sole metric used to score moves.

Over the course of this project the doubling cube has proven to be even more valuable than initially expected. In experiments performed out of pure curiosity, it was observed that agents with superior move play would, on average, be outperformed by agents with inferior move play, so long as the agent with inferior move play had a better doubling algorithm. Matches were organised where the Deep agent would use the intermediate agents' doubling strategy and its opponents, the Adaptive and Genetic agents, would use the Deep agent's doubling strategy. The win rates would be relatively even between the two players, but the Deep agent's adversary would have a massive PPG advantage (Fig. 17). This was because when the adversary did win a game, it would be by a spectacular margin (on one occasion the Deep agent lost a match by over 4 million points against the Genetic agent!) which consequently undid all of the work the Deep agent had done up to this point. This shows just how important strong doubling strategies are when trying to maximise PPG differential, as one can exploit an opponent with better move-play if one's doubling strategy is superior.

To conclude, this project demonstrates the importance and promise of prioritising points won per game rather than just the raw win rate in tournament-style matches. Reinforcement learning has been shown to extract complex features relevant to the game extremely quickly, with self-play and co-evolution very beneficial to the training process. Improvements can be made to each agent, but the Reinforcement agent has the most promise, achieving advanced-level play without even possessing precomputed board features.

REFERENCES

[1] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, p. 58–68, 1995. [Online]. Available: https://doi.org/10.1145/203330.203343

[2] G. Tesauro, "Neurogammon: a neural-network backgammon program," in *1990 IJCNN International Joint Conference on Neural Networks*, 1990.

[3] G. Wong, "Gnu backgammon," 2001. [Online]. Available: https://www.gnu.org/software/gnubg/

[4] H. J. Berliner, "Bkg: A program that plays backgammon," 1977. [Online]. Available: https://bkgm.com/articles/Berliner/BKG-AProgramThatPlaysBackgammon/

[5] H. J. "Berliner, "Backgammon computer program beats world champion," *Artificial Intelligence*, vol. 20, no. 2, pp. 121–136, 1980.

[6] G. Tesauro, "Neurogammon wins computer olympiad," *Neural Computation*, vol. 1, no. 3, pp. 321–323, 1989. [Online]. Available: https://ieeexplore.ieee.org/document/6796123

[7] G. Tesauro, "Td-gammon," 2007. [Online]. Available: http://www.scholarpedia.org/article/User:Gerald_Tesauro/Proposed/Td-gammon

[8] N. Zadeh and G. Kobliska, "On optimal doubling in backgammon," *Management Science*, vol. 23, pp. 853–858, 1977. [Online]. Available: https://api.semanticscholar.org/CorpusID:61447889

[9] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*. Association for Computing Machinery, 2009, p. 41–48. [Online]. Available: https://doi.org/10.1145/1553374.1553380

[10] C. E. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 7, no. 314, 1950.

[11] D. Michie, "Chapter 8 - game-playing and game-learning automata," in *Advances in Programming and Non-Numerical Computation*. Pergamon, 1966.

[12] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001291

[13] D. Yang, "Stockfish," 2010. [Online]. Available: https://stockfishchess.org

[14] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. [Online]. Available: https://doi.org/10.1038/nature16961

[15] E. Keeler and J. Spencer, "Optimal doubling in backgammon," *Operations Research*, 1975. [Online]. Available: https://pubsonline.informs.org/doi/10.1287/opre.23.6.1063

[16] N. Brown and T. Sandholm, "Superhuman ai for multiplayer poker," *Science*, vol. 365, no. 6456, pp. 885–890, 2019. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aay2400

[17] N. Brown, A. Lerer, S. Gross, and T. Sandholm, "Deep counterfactual regret minimization," 2019. [Online]. Available: https://arxiv.org/abs/1811.00164

[18] A. Martens, "Backgammon ui with ai in python," 2022. [Online]. Available: https://github.com/alexandremartens/Backgammon

[19] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 1996.

[20] J. Matussek, "How to calculate exact game winning chances during bearoff," 2004. [Online]. Available: https://bkgm.com/articles/Matussek/BearoffGWC.pdf

[21] N. Kolatacz, "How i used machine learning to beat my friends at backgammon," 2025. [Online]. Available: https://medium.com/@nitzankolatacz/how-i-used-machine-learning-to-beat-my-friends-at-backgammon-fb541ec1c0e5

[22] A. Schneider, "Fibs, the first internet backgammon server," 1992. [Online]. Available: http://www.fibs.com

[23] G. Tesauro, "Connectionist learning of expert preferences by comparison training," in *Advances in Neural Information Processing Systems*, 1988.

[24] H. J. Berliner, "On the construction of evaluation functions for large domains," in *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, 1979, p. 53–55.

[25] G. Tesauro, "Practical issues in temporal difference learning," *Mach Learn*, 1992. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5726779

[26] K. Fukushima, "Cognitron: A self-organizing multilayered neural network," *Biol. Cybern.*, vol. 20, no. 3–4, p. 121–136, 1975. [Online]. Available: https://doi.org/10.1007/BF00342633

[27] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

APPENDIX A

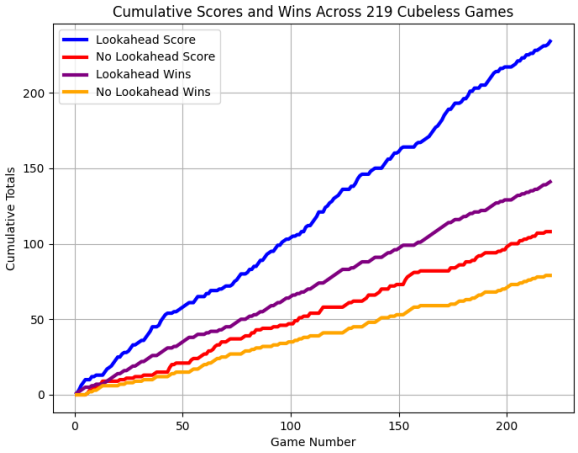

Fig. 6. ReinforcementV2 with 2-ply lookahead vs without 2-ply lookahead

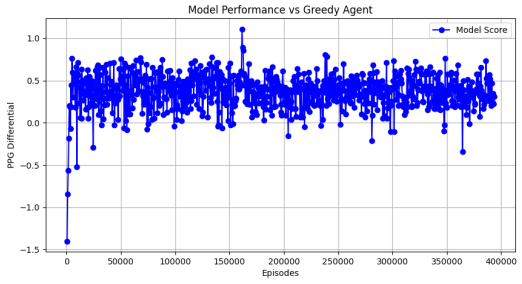

Fig. 7. ReinforcementV2 with no lookahead vs Greedy agent every 500 episodes

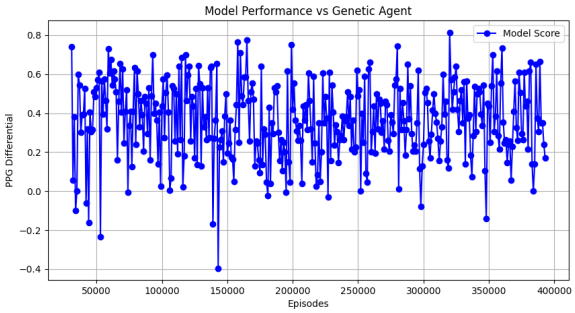

Fig. 8. ReinforcementV2 with no lookahead vs Genetic agent every 1000 episodes

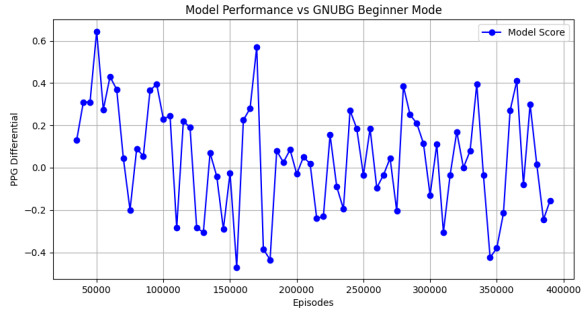

Fig. 9. ReinforcementV2 with no lookahead vs GNUBG Beginner mode every 5000 episodes

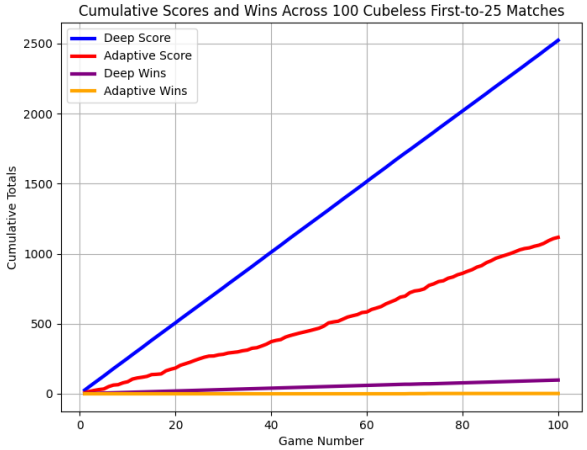

Fig. 10. 100 Cubeless matches between the Adaptive and Deep agents

Algorithm 3 Version 1 of the Greedy Evaluation Function

Require: starting board state s
Require: resulting board state t
Require: move m
Ensure: board evaluation $score$
  $score \leftarrow 0$
  for each x, y in m **do** ▷ x, y = start-end pair
    if $y = 26$ or $y = 27$ **then** ▷ Bearing off a piece
      $score \leftarrow score + 4$
    else if $s[y]$ is opponent blot **then**
      $score \leftarrow score + 3$
    else if $s[y]$ is player blot **then**
      if $s[x] > 2$ **then** ▷ Making another anchor
        $score \leftarrow score + 2.5$
      else if $s[x]$ is a player blot **then**
        $score \leftarrow score + 2$
      else
        $score \leftarrow score + 1.5$
      end if
    else if y in player's home board **then**
      $score \leftarrow score + 1$
    end if
    if $t[y]$ is player blot **then**
      $score \leftarrow score - 1$
    end if
  end for
  return $score$

TABLE 12
All factors Considered by the Adaptive agent's Equity Calculator. Factors Marked with an Asterisk are Computed using other Factors in the Table.

Factor
Pip advantage
Chance blot could be hit
Does player have a prime?
Position of prime
% of points agent occupies in their home board
% of point opponent occupies in their home board
Number of agent pieces on bar
Number of opponent pieces on bar
Number of agent's pieces borne off
Number of opponent's pieces borne off
Board volatility
Is the agent winning by a lot?
Is the match close?
Is the agent losing by a lot?
How many points until a victor is declared?
Number of blots exposed by agent
Number of blots exposed by opponent
Number of walls built by agent
Number of walls built by opponent
Number of player walls in agent's home
Number of player walls in opponent home
Number of opponent walls in agent's home
Number of opponent walls in opponent's home
Probability of opponent entering off bar
Probability of agent entering off bar
Does an exposed blot have a friendly anchor in consecutive points?
Cube volatility*
Gammon potential*
Gammon probability*

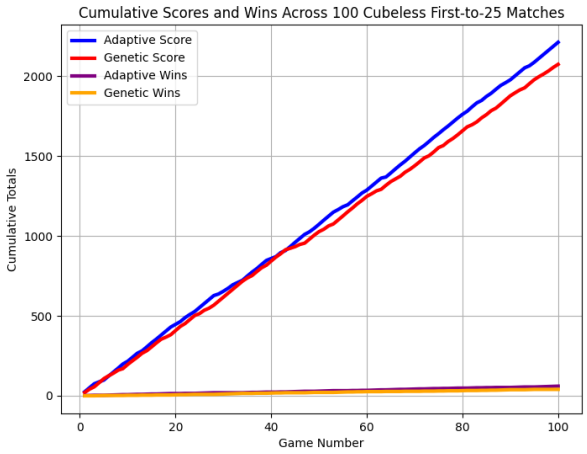

Fig. 11. 100 Cubeless matches between the Adaptive and Genetic agents

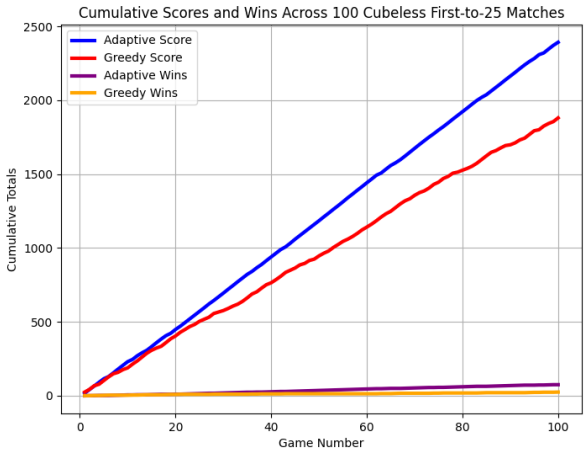

Fig. 12. 100 Cubeless matches between the Adaptive and Greedy agents

TABLE 13
All factors Considered by the Genetic and Greedy Evaluation Functions

Factor
Is the agent's home walled off with opponent piece on bar?
Did agent hit off piece and then completely wall off their home?
Number of pieces borne off
Number of pieces hit off
Did the agent remove any blots?
Did the agent hit a piece but expose blot in process?
Number of home walls
Number of walls
Number of blots exposed by agent
Is the board state a race position?
Is the board state a contact position?
Change in number of home walls
Change in number of walls
Did number of walls stay the same?
Did the number of blots stay the same?
Did the agent expose a blot in a contact position to bear off
Number of opponent blots in agent's home before move
Number of opponents blots in agent's home after move

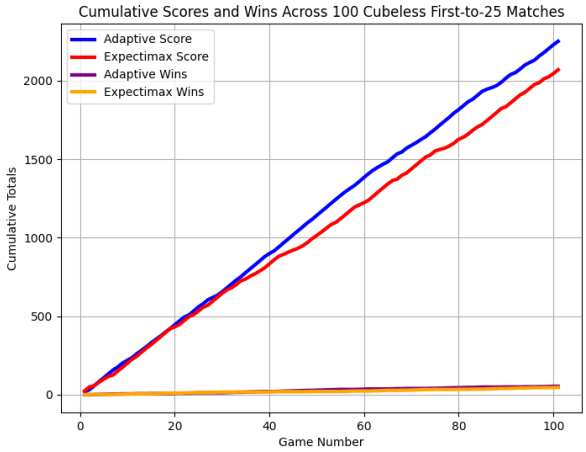

Fig. 13. 100 Cubeless matches between the Adaptive and Expectimax agents

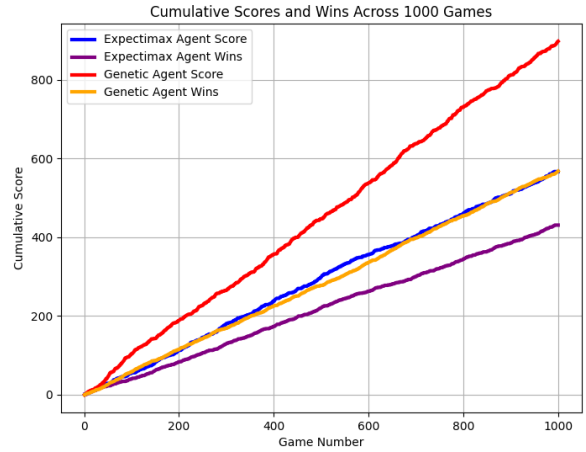

Fig. 14. 1000 Cubeless games between the Expectimax and Genetic agents

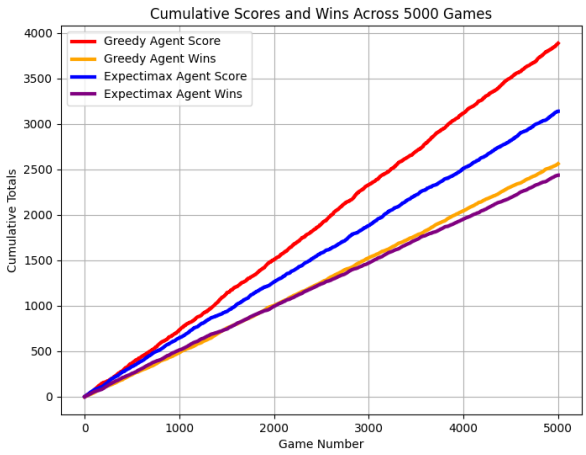

Fig. 15. 5000 Cubeless games between the Expectimax and Greedy agents

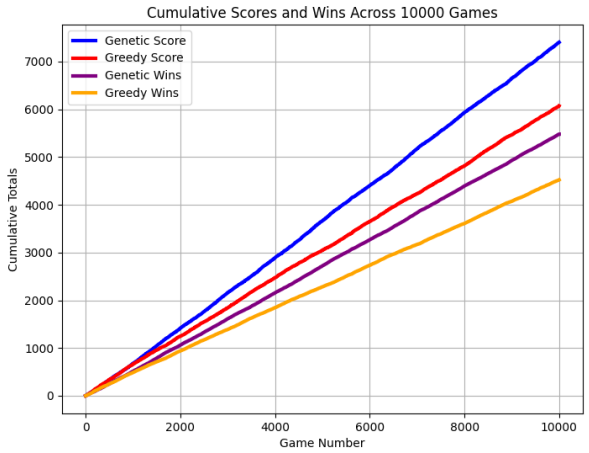

Fig. 16. 10000 Cubeless games between the Genetic and Greedy agents

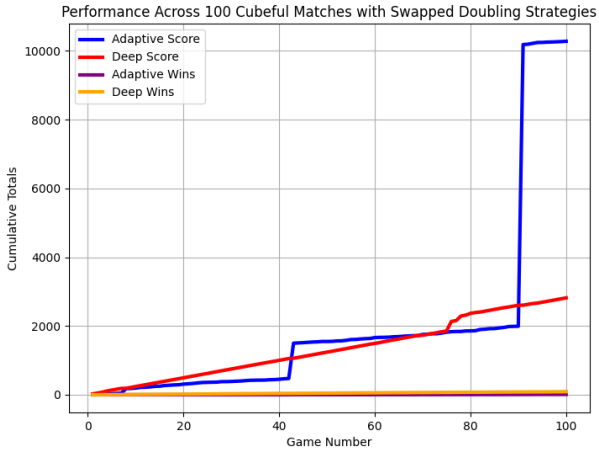

Fig. 17. 100 Cubeful matches between the Deep and Adaptive agents with their doubling strategies swapped