Reinforcement Learning, Looking for New Backgammon Strategies

Student Name: Fatema Alkhanaizi

Supervisor Name: Rob 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 —

A Context/Background

Neural networks are used as the evaluation function for the reinforcement learning agent for backgammon. Combinations of different backgammon strategies has been incorporated by using modular network architecture, surpassing the performance of agents that used monolithic network architecture and resulting in new game strategies.

B Aims

This project aimed to find a new strategy for backgammon, making use of the modular network architecture. A hybrid of known strategies mainly priming and back game strategies was used as the basis for the new strategy. It also aimed to study the influence of including game knowledge, mainly pip count and hit probability, as part of the network's input units.

C Method

Two modular networks were implemented and trained, one that included the priming and back game strategies separately and another that included a hybrid of the 2 strategies. A monolithic neural network based on TD Gammon was implemented and trained. These networks were used as a reference to the hybrid network performance. All networks' general performance was evaluated by playing test games against a benchmark agent, pubeval. To evaluate the strategies, GNU-Backgammon software was used to provide an analysis of a complete match.

D Results

Overall, best performance was recorded for monolithic network, 0.612 against pubeval. The best hybrid strategy network and the best separate strategy network recorded 0.36 and 0.282 respectively. Including features had resulted in the divergence of modular networks results in some cases. Priming strategy was the dominant strategy used by all networks. The hybrid strategy did not use any form of back game strategy.

E Conclusions

The hybrid network did not achieve the desired results but showed some potential. Incorporating better training strategy and more training time could result in better performance.

Keywords — Backgammon; Reinforcement Learning; Modular Neural Network; Priming Game; Back Game; Strategy

I INTRODUCTION

A Backgammon

Backgammon is a game played with dice and checkers on a board consisting of 24 fields, in which each player tries to move their checkers home and bear them off while preventing the opponent from doing the same (Keith 1995). Figure 1 illustrates the setup that was be used for this project.

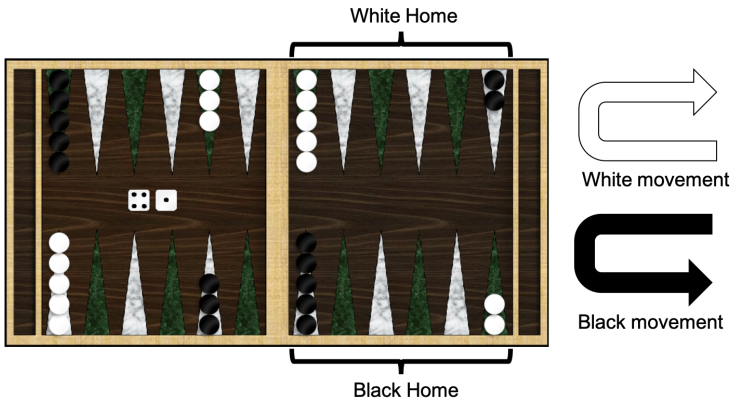


Figure 1: Backgammon board setup

Like many board games, the player with the best strategy usually wins. There are many basic backgammon strategies that are often used: **The running game**, the objective is to bring all checkers into inner board/home and bear them off as quickly as possible, similar to a competitive race (Keith 1995). **The back game**, the player tries to hold two or more anchors, points occupied by two or more checkers in the opponent's home board, as long as possible and force the opponent to bear in or bear off awkwardly (Keith 1995). **The priming game**, a particular type of holding game that involves building a prime a long wall of the player's pieces, ideally 6 points in a row in order to block the movement of the opponents pieces that are behind the wall (Keith 1995).

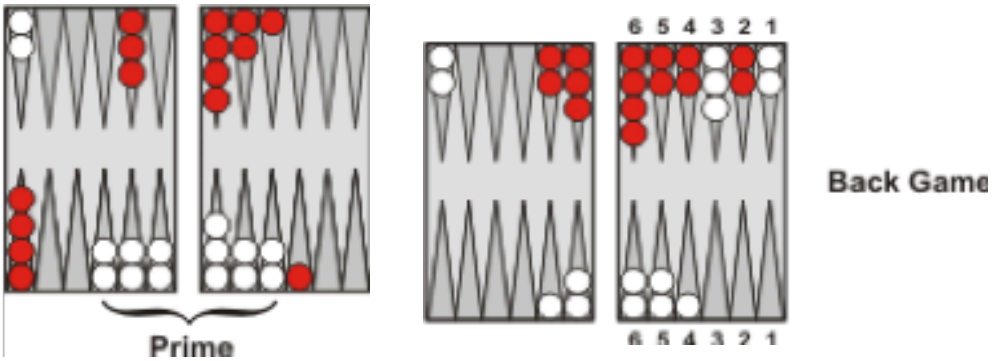


Figure 2: Illustration of Priming and Back games

B Using Reinforcement Learning for Backgammon

Backgammon and many other board games have been widely regarded as an ideal testing ground for exploring variety of concepts and approaches in artificial intelligence and machine learning. Reinforcement learning is an often-used approach. A reinforcement learning algorithm allows an agent to learn from its experience generated by its interactions with an environment. TD-Gammon of Tesauro (1992) used TD(λ) reinforcement learning methods with a single neural network that trained itself and learned from the outcomes to be an evaluation function for the game of backgammon. Due to the branching factor for backgammon, which is about 400, the conventional heuristic search methods that were effective for games such as chess and checkers couldn't be applied to backgammon (Sutton & Barto 1998). Making backgammon more suited for Temporal difference learning methods. Temporal difference is a class of reinforcement learning that uses the difference between two successive positions for backpropagating the evaluations of the successive positions to the current position. TD(λ) algorithms are part of Temporal difference algorithms that are parametrised by the value λ which makes the agent looks further in the future for updating its value function, strongly influencing the agent learning (Sutton & Barto 1998). In backgammon, the dice rolls guaranteed sufficient variability in the games so that all regions of the feature space would be explored. This characteristic made it also perfectly suited for learning from self-play as it overcame a known reinforcement Learning problem with the trade-off between exploration and exploitation (Frnkranz 2000).

C Project Overview

TD-Gammon lacked in some aspects of the game like the racing game (Frnkranz 2000), this was improved by using modular neural network architecture illustrated by the work of Boyan (1992). Modular network acrchitecture allowed the inclusion of backgammon strategies as part of the network structure. This project focused on using modular neural network architecture to incorporate a hybrid of backgammon strategies to find a new game strategy. In addition, the performance of the new strategy was evaluated by comparing it to another network that included the strategies separately. To evaluate the strategy used by both networks, GNU-Backgammon software was used to provide a complete match analysis. Combinations of 2 strategies was studied: **the back game** and **the priming game**. A strategy for **the racing game** was included as an additional support network for the overall strategy as it overcame the weakness of TD-Gammon's network. This addition did not affect the strategy during game play and only was used toward the end of the game.

D Project Objectives and Achievements

The objectives for this project were divided into 3 categories: basic, intermediate, advanced. Each category related to the research questions studied in this project.

The basic objectives for this project were related to studying the following research question: *How effective would the hybrid strategies be over the strategies being included separately?* Three neural networks were trained: a monolithic neural network based on TD Gammon architecture, a modular neural network that included priming and back game strategies and a modular neural network that included a hybrid of priming and back game strategies. The monolithic network was used as a reference for the performance of the implemented modular networks. Overall performance of the networks was evaluated by playing 1,000 games against pubeval, an intermediate

level benchmark agent provided by Tesauro (1993). The goal was to achieve a result close to 0.596 which was the recorded value for TD Gammon.

The intermediate objectives were related to studying the following research questions: *Would including more features to the networks result in better strategies? Would using the hybrid strategy result in the agent learning a new strategy? or would one strategy be more dominant than the other?* Pip count and hit probability were the additional game features studied. Pip count is the total number of points that a player must move his checkers to bring them home and bear them off; the total at the start was 167 points for the setup used. The pip count was scaled when included to the network's basic features to keep all features around the same range [0,1]. Hit probability measured the likelihood of the opponent to take single pieces within their reach on the board. For every addition, all networks were retrained. To get an insight into the strategies followed by the networks, GNU-Backgammon software was used to generate a complete analysis of the matches played by the networks. Each position made would be evaluated by giving it a score and a rank.

The advanced objectives were related to the inclusion of doubling cube and extending the action search algorithm to use more complex algorithms. The influence of these additional algorithms would be tested, and the networks would be retrained and reevaluated with the new additions.

E Deliverables

- **Trained neural networks**: A single neural network based on Tesauro's TD Gammon, a modular neural network that uses a new strategy, a hybrid priming and back strategy, and a modular neural network that uses two separate known strategies, priming and back strategies.
- **AI agent**: This agent was able to use the trained network to evaluate and make moves. It also included a greedy search algorithm.
- **User interface and human agent**: This was a simple command line interface which took user inputs to make moves and to play against the trained networks. Another complex implementation was a graphical user interface which captured the user clicks for making moves.
- **Testing suit**: This was used to run all tests and evaluations for the networks.
- **Project Report**: All tests and evaluations were recorded and analysed in this report.

II RELATED WORK

Many researches had been conducted on how to develop a strong backgammon AI agent. Early research had used heuristic approaches and supervised learning methods, however temporal difference learning methods proved to be superior and resulted in stronger agents surpassing the world's strongest Backgammon players (Tesauro 2002). Approaches that used temporal difference methods, also used neural networks for the value evaluation function. The architecture of the networks influenced the strength observed for the agent. Complexity and depth of action selection algorithm also played a part in the overall performance. In this section, an overview of

the current state of art implementations, the neural network architectures used, action selection algorithms, training techniques and other learning approaches is given.

A State of the art implementations

TD-Gammon of Tesauro (1992, 2002) had demonstrated the impressive ability of machine learning techniques to learn to play games. TD-Gammon used reinforcement learning techniques with a Neural Network that trained itself to be an evaluation function for the game of backgammon, by playing against itself and learning from the outcome (Tesauro 2002). This made TD-Gammon a point of interest to many following researches. Many had tried to replicate the success of TD-Gammon by applying Temporal difference learning to other game such as chess, Othello and Go, but non reached the same level of success as TD-Gammon (Frnkranz 2000). The stochastic nature of backgammon made it suited for temporal difference learning. TD-Gammon's network required little backgammon knowledge, but still achieved a near level of the world's strongest grandmaster. This outcome had shown that using reinforcement learning methods to play stochastic games like backgammon was promising. In addition, TD-Gammon's strength had given insight to potential backgammon strategies that were not considered by experts during that time (Sutton & Barto 1998). Extending the work of Tesauro, Boyan (1992) showed that the use of different networks for different subtasks of the game could outperform learning with a single neural network. Many AI software for Backgammon use similar modular neural network architecture such as GNU-Backgammon, a top performing open-source software, uses 3 different neural networks for their evaluation function (Silver 2006). Temporal difference with Modular network architecture is the state of art approach.

B Neural Networks Architecture

The architecture of the neural network used for the backgammon agent plays a big role in the networks strength and performance. A large neural network with more hidden nodes provided better results compared to a smaller network as shown in the work of researchers like Tesauro (2002) and Wiering (2010), however the bigger the network the more computations are required thus the slower the learning. The learning parameters used also influences the learning. Results from the work of Tesauro (1998) indicated that with λ being between 0.6 and 0.7 best performance was observed; Tesauro (1993) reported a 0.596 performance against a benchmark agent pubeval which he made available for the community to use. Tesauro developed pubeval himself and had remarked that it plays at intermediate player level. In another research looking at different variance of backgammon, $\lambda = 0$ gave much better performance (Papahristou & Refanidis 2012). TD-Gammon was rather strong agent, but it ignored important aspects of backgammon like the running game (Frnkranz 2000). Boyan's (1992) modular neural network architecture had proven to be superior than the monolithic neural network architecture used in TD-Gammon, overcoming its observed weakness. He used 2 types of modular architectures: Designer Domain Decomposition (DDD) architecture and Meta-Pi architecture. His DDD architecture made use of a gating program that partitioned the space of backgammon positions. Each partition had its own neural network. He noted that better partitioning of backgammon positions in the gating program could improve the performance. This architecture seems reasonable to be used to include different backgammon strategies, making the strategies act as partitions for the positions space. The Meta-Pi architecture dealt with replacing the gating program with a neural

network to deal with activating the partition networks. However, it did not improve performance on his DDD network. As for one the best-known AI agents, GNU-Backgammon is made of 3 neural nets: the contact net which is the main net for middlegame positions, the crashed net, and the race net (Silver 2006). This further proves the suitability of modular architecture for including backgammon strategies to the learning agent.

C Action Selection Algorithm

In early stage of research, a single lookahead search algorithm was used, 1-ply search. A ply is one turn taken by one user; n-ply refers to how far the player will look ahead when evaluating a move/action (Keith 1995). eXtreme Gammon (2017), currently the supreme software (Depreli 2012), uses 2-ply and 3-ply search to select an action/move. The best GNU-Backgammon (Silver 2006) agents also use up to 4-ply search algorithm, however, although the agent's performance improves, the agent starts to take longer time from 3-ply search level which is not ideal for timely matches. Doubling cube (cubeful) decisions are included in the action selection algorithm. TD-Gammon used a heuristic function that was based on a generalisation of prior theoretical work on doubling strategies by Zadeh and Kobliska (Tesauro 2002). GNU-Backgammon estimated the cubeful equity from the cubeless equity by using a generalised transformation as outlined by Rick Janowski (Silver 2006). Both approaches to computing the cubeful equity made use of a heuristic function instead of a neural network. The neural network approach is more complex, so the former approach is mostly used. Influence of doubling cube to the agent's strategy is not present in many recent researches.

D Training Techniques

Tesauro (1992) used self-learning to train the neural network for TD-gammon and it was able to reach master-level. In his research, Wiering (2010) demonstrated that by utilising an expert program, the speed of learning slightly increased in comparison to self-learning. However, the end outcome from both learnings was the same and learning from watching a match against two experts was the worst technique. Training neural networks by self-learning remains the most popular training approach that is used.

E Other learning approaches

Other approaches to learning included genetic programming (Azaria & Sipper 2005) and Hill Climbing (Pollack & Blair 1998). Pollack and Blair claimed that their Hill Climbing algorithm had 0.40 winning factor against Pubeval. The genetic programming approach had achieved better results against Pubeval of 0.58 and had shown good results from automatically obtained strategies, but it required more computational efforts and was more complex (Azaria & Sipper 2005). These are good results, but, TD-Gammon reached 0.596 with a less complex approach using temporal difference learning. Temporal difference remains the most suitable approach for backgammon.

III SOLUTION

A Algorithms

A.1 Temporal Difference, TD(λ)

TD Gammon used the gradient-decent form of the TD(λ) algorithm with the gradients computed by the error backpropagation algorithm (Sutton & Barto 1998). The update rule is as follows

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t$$

where δ_t is the TD error,

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

and $\vec{e}_t$ is a column vector of eligibility traces, one for each component of $\vec{\theta}_t$, updated by

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t}V_t(s_t)$$

The expression $\nabla_{\vec{\theta}_t}V_t(s_t)$ refers to the gradient. For backgammon, $\gamma = 1$ and the reward is always zero except when winning, reducing the update rule to

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[V_t(s_{t+1}) - V_t(s_t)][\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t}V_t(s_t)]$$

α, the learning rate, and λ are constraint by the range $(0, 1)$. $e_t(s)$ is the eligibility trace for state s, it marks s as eligible for undergoing learning changes when a TD error, δ_t occurs (Sutton & Barto 1998). This exact algorithm was used in this project.

A.2 Action selection algorithm

The implemented AI agent used a 1-ply search algorithm, (Tesauro 1992), to pick the best legal action for the current turn; no lookahead. Each action was evaluated in the neural network, forward feeding, and the action with the maximum outcome was picked.

B Neural Network Architecture

B.1 Monolithic Neural Network

This network was based on Tesauro's (1994) TD Gammon implementation; a fully-connected feed-forward neural networks with a single hidden layer. The basic architecture consisted of one input layer I with 288 units for raw inputs representing the checkers configuration on the board, each field in the board was represented by 6 units as there are 24 fields so 144 total units and each player had their own configuration represented separately making the total 288. For basic network architecture, the first 4 features from table 1 were used, the other 4 were included when experementing with extra knowledge features. Lastly, 2 input units were included to represent if it was the player's turn or the opponent's turn.

As part of the network architecture, there was one hidden layer H with 50 units and one output layer O with 1 unit representing the winning probability. The number of units for each layer was based on previous implementations, taking into account the complexity introduced by the increase of the units for each layer. The network had weights w_{ih} for all input units I_i to

Table 1: Possible expert features for input layer

Feature name	Description
bar_pieces_1	number of checkers held on the bar for current player
bar_pieces_2	number of checkers held on the bar for opponent
off_pieces_1	percentage of pieces that current player has borne off
off_pieces_2	percentage of pieces that opponent has borne off
pip_count_1	scaled pip count for current player
pip_count_2	scaled pip count for opponent
hit_pieces_1	percentage of pieces that are at risk of being hit (single checker in a position which the opponent can hit) for current player
hit_pieces_2	percentage of pieces that are at risk of being hit (single checker in a position which the player can hit) for opponent

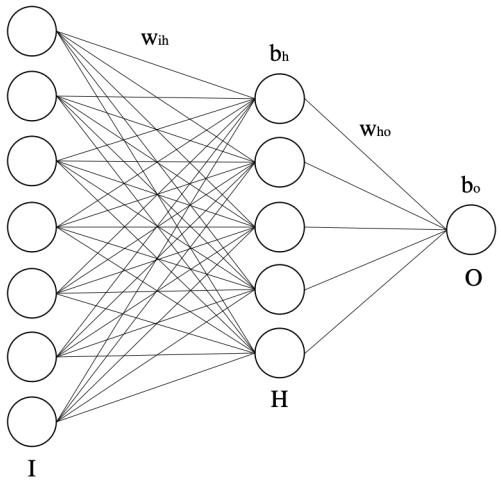


Figure 3: Neural Network architecture

hidden unit H_h and weights w_{ho} for all hidden units H_h to output unit O_o. The weights were initialized to random values; hence the initial strategy was a random strategy. Each hidden unit and output unit had a bias b_h and b_o. Sigmoid function was used for the activation of the layers. Each bias was initialized to an array of constant values of 0.1. Figure 3 includes the layout of the neural network.

A lot of implementations of this network were available in the open source community and were referred throughout the implementation of this project; two code bases from GitHub, backgammon (Hannun 2013) and td-gammon (Fleming 2016), were used as the starting point for this project. The main challenge with using open source code was debugging the code and validating it.

B.2 Modular Neural Network

Based on Boyan's (1992) work, Designer Domain Decomposition Network (DDD) was the Modular Neural Network architectures that was followed in this project. The DDD network consists of n monolithic neural networks and a hard-coded gating program, figure 4. For his gating pro-

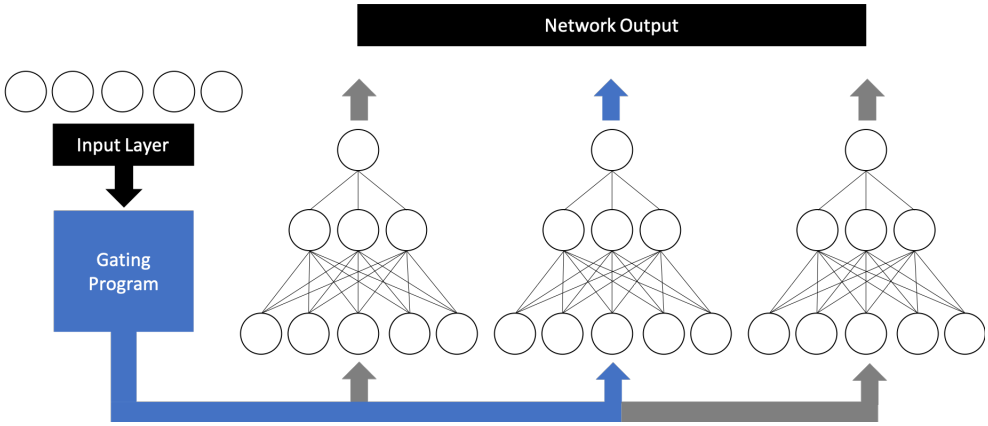


Figure 4: The DDD Network

gram, Boyan's (1992) partitioned the domain space into n classes. For this project the n classes were represented by the different backgammon strategies. each strategy was activated when certain board configurations were reached. Both modular networks, separate strategy and hybrid, implemented in this project consisted of the following basic networks:

- One network for racing game. This network addressed a known weakness of the monolithic network implementation and was only activated when both players' checkers were past each other.
- One default network for all other positions

The separate strategy modular network included the following additional networks:

- One network for back game positions: the player is behind in the race, the pip count difference is more than 90 points, but the player has two or more anchors, checkers at one field, in the opponent's home board. This network took into account the number of checkers on the bar.
- One network for priming games: if the player has a prime of 4-5 fields, a long wall of checkers, with at least 2 checkers on each field.

In comparison, the hybrid modular network included this network:

- One network for a hybrid priming and back game. This network combined the conditions of both back and priming games.

In both forward feeding and backpropagation, the gating program was called to select an appropriate network based on the current board configuration. Exactly one network was activated at any time. Each network had the same structure as the monolithic network including the parameters.

C Implementation

Python 3 was used as the language for this project. The neural networks were implemented using TensorFlow package. TensorFlow was used as it allowed the generation of training progress summary and easily save and restore the trained networks through checkpoints. Figure 5 shows the structure and the component dependencies.

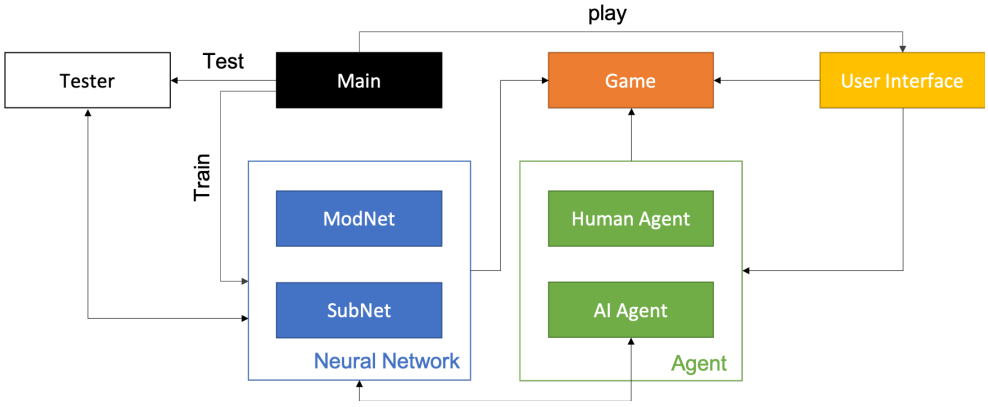


Figure 5: System components and dependencies

C.1 Main

This module was used to invoke other components in the system and to handle the general actions required from the system: play, train and test.

C.2 Game

This module held the game setup, the rules and constraints of the game e.g. taking an action, finding legal moves and game board setup. An open source implementation taken from a GitHub repository, backgammon (Hannun 2013), of this module was refactored and modified for the use in this project.

C.3 User Interface

This module handled generating the command line interface (textual interface) and the graphical user interface. The use of either interfaces depended on the availability of pygame, a graphical user interface package for python. Similarly, to the Game module, an open source implementation taken from a GitHub repository, backgammon (Hannun 2013), of this module was refactored and modified.

C.4 Agents

There are 4 types of agents that were implemented:

- **Random Agent**, an agent that would randomly select a legal move.
- **Human agent**, an interactive agent which took user inputs either from the command line or by capturing the user clicks though a GUI to make a move/action.
- **AI agent** that used a neural network to determine the move/action for the current turn. A list of legal moves was obtained from the game module and the best action was selected based on the search algorithm.

- **Pubeval Agent**, an intermediate level benchmark agent provided by Tesauro (1993). Pubeval had two set of weights that it used to select a move: weights for racing game and weights for the remaining positions.

C.5 Modnet

This module defined the operations for extracting features from the game board and training neural networks. This module heavily depended on Subnet module. An instance of this module was used for the monolithic network. For modular networks, a game-specific gating program was implemented in this module to determine which sub-network was best suited for a given board configuration represented as a set of extracted features. For the different modular neural networks to be trained for this project, different instances of this module were created as each modular network required different gating program.

C.6 Subnet

This module included the Neural Network implementation using TensorFlow. It provided routines for storing and accessing the network model, checkpoints and summaries generated by TensorFlow. In addition, it included the forward feeding and backpropagation algorithms. All networks created for this project used an instance of this module; networks used in modular neural network and monolithic neural network. An open source implementation taken from a GitHub repository, td-gammon (Fleming 2016), was used as the basis for this module.

C.7 Tester

This module included all evaluations and test routines for the neural networks.

D Training and Testing

Each network had 3 types of checkpoints that were used throughout the evaluation process of the project: latest, previous, test. The latest checkpoint stored the most recent trained network. The previous checkpoint stored the latest 1,000s game trained network. The test checkpoint stored all the trained networks after every 1,000s game.

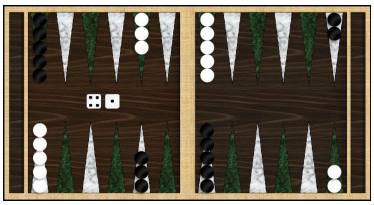
D.1 Monolithic Neural Network

The number of training games was set to 500,000. Based on previous work of Tesauro (2002), and Wiering (2010), the best weights could be reached at a game far before the 500,000 game. After every 1,000 games, the network was tested by playing 100 games against Pubeval agent. This test was set as an indicator of the training progress.

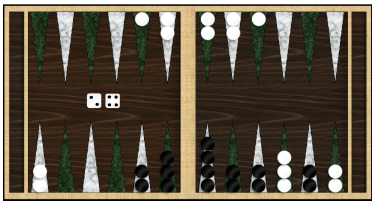
D.2 Modular Neural Network

Similarly to the monolithic network training, the number of training games was set to 500,000 and after every 1,000 games, the current network would be tested by playing 100 games against Pubeval agent. However, for modular networks, after every 2,000 games the layout of the game was changed, figure 6. Due to the gating program and the conditions implied for the activation of

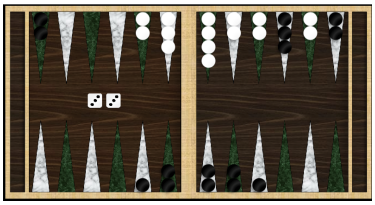
each sub-network, 4 extra layouts were included: racing layout, prime against prime, back game from the player perspective and back game from the opponent perspective. With these layouts, each sub-network was trained for relatively the same number of games.



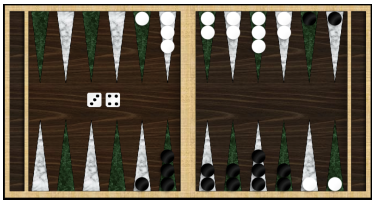
(a) Basic layout



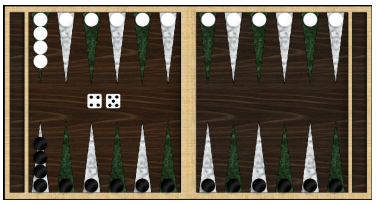
(b) Player back game layout



(c) Opponent back game layout



(d) Prime against prime layout

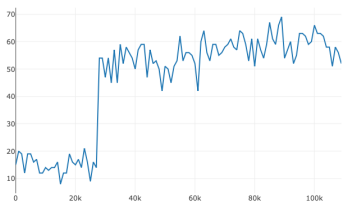


(e) Racing layout

Figure 6: Modular Network training layouts

E Strategy Validation

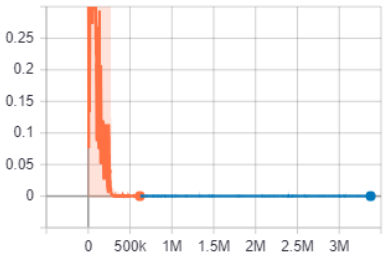
The main aim of this project was to find a new backgammon strategy. To validate the strategy followed by the networks, a measurement for the general performance of the strategy against a benchmark agent was taken and few complete matches for the best networks were analysis. Before making any measurements or analysis, it was important to determine at which stage should the training stop i.e. convergence of the network. There were 3 indicators that determined the convergence of a network: reasonably consistent test games results, low total number of turns taken and convergence of the loss plot, figure 7. The loss plot was for the mean squared of TD error, δ_t. A low total for number of turns per games indicated that the training was going in the right direction (Tesauro 2002); very long games, over 500 turns, were a red flag.



(a) Consistent test games results

```
[Train 56367/1000000] (Winner: x) in 37 turns
[Train 56368/1000000] (Winner: x) in 45 turns
[Train 56369/1000000] (Winner: o) in 40 turns
[Train 56370/1000000] (Winner: x) in 54 turns
[Train 56371/1000000] (Winner: x) in 35 turns
[Train 56372/1000000] (Winner: o) in 42 turns
[Train 56373/1000000] (Winner: x) in 38 turns
[Train 56374/1000000] (Winner: o) in 42 turns
[Train 56375/1000000] (Winner: o) in 48 turns
```

(b) Low number of turns for each game



(c) Converged loss plot

Figure 7: Convergence indicators

To measure the performance of the networks, the random agent was used in early stages of this project, but it was replaced by pubeval agent as it was too weak. To analyse the strategies of the networks, GNU-Backgammon software was used. GNU-Backgammon software provided a

complete analysis highlighting bad moves in red and marking them with question marks, figure 8. Further analysis was provided by clicking on the move with a breakdown of the points gained or lost by making that move along with its ranking. There were many agents with varied strengths provided by the software. The world class agent was picked to provide a good challenge for the trained networks; it played a really strong game close to the best human players in the world by using 2-ply lookahead, no noise, and a normal move filter (Silver 2006).

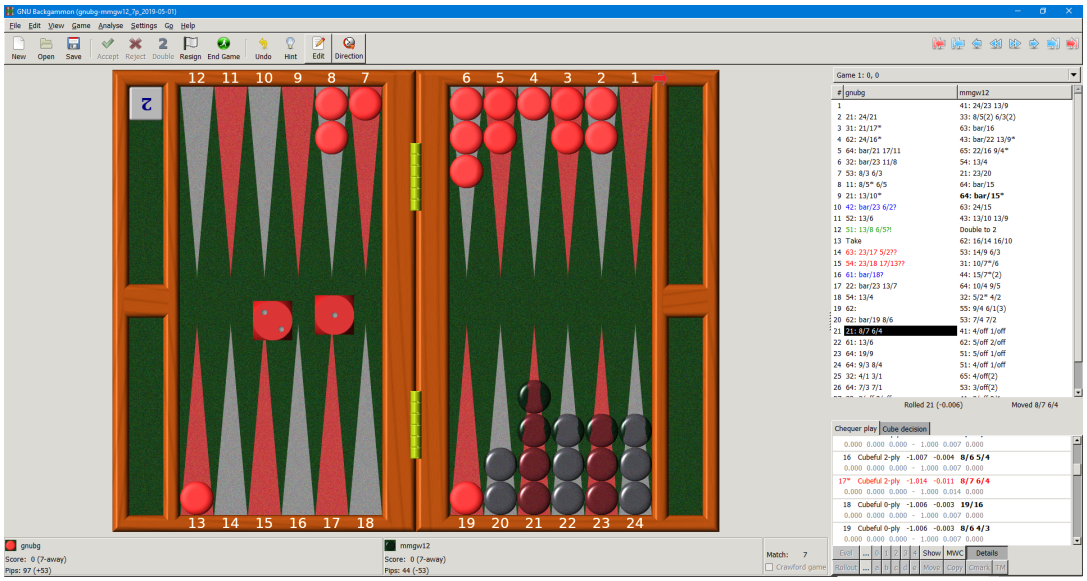


Figure 8: GNU Backgammon software

IV RESULTS

In this section, the results of all networks' training are presented. All Networks were trained on machines with the specifications outlined in table 2. For every parameter and feature experimented, the networks were retrained for relatively the same time duration to reach the goal of 500,000 trained games. Monolithic networks were estimated to take 1 week while modular networks were estimated to take 2 weeks; these estimates were made by considering the number of games ran in an hour, it was around 3,100 games per hour for monolithic network and around 2,000 games per hour for modular networks. Each network used approximately 1 core of the machine, this allowed for 4 instances of the networks to run at the same time on a single machine. Instances were run multiple times in case of divergence or test results were lower than 10% and no progress was observed within a weeks time.

Table 2: environment system specification

Component	Description
Processor	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
Ram	32.00GB
Operating system	Windows 10

A General performance

In the first stage of experiments, parameters for λ and α were varied. This was due to low performance and divergence issues with modular networks. The values used were based on previous related works. The basic input features were used in all experiments. Initially, the random agent was used as the benchmark player. However after experimenting with different parameters, the random agent proved to be too weak, so it was disregarded as a testing agent, table 3. Pubeval agent was used for the remainder of the tests. When $\lambda = 0.7$ and $\alpha = 0.01$,

Table 3: Best trained parameters after 1000 test games against pubeval (P) and random (R) agents

	$\lambda = 0.7, \alpha = 0.01$			$\lambda = 0, \alpha = 1, 0.1$		
Architecture	Max at	Win rate(P)	Win rate(R)	Max at	Win rate(R)	Win rate(P)
Monolithic	100,000	66.0%	3.0%	159,000	98.0%	61.20%
Separate Modular	-	-	-	111,000	90.1%	28.20%
Hybrid Modular	-	-	-	46,000	87.0%	17.30%

for monolithic network the performance did not improve even after 100,000s games and only resulted in a faster convergence for the network; the training wasn't improving the performance even after 500,000 games. For both modular networks the results quickly diverged for the number of turns per game reaching over 1,000, hence the missing results. When λ was closer to 0 and $\alpha = 1$, performance significantly improved after the first 1000 games and continued to improve after 100,000 games, figure 9a; after 500 games α was modified to 0.1, this was in line to the work of Papahristou and Refanidis (2012) as it gave good results. Changing α did make any noticeable improvements.

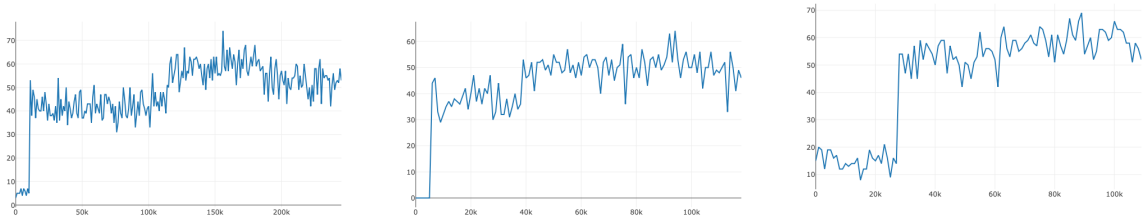
For the second stage of experiments, the input features were varied. Table 4 shows the performance of the networks with basic input features, with pip count included and with both pip count and hit probability included against pubeval agent after 1000 test games. λ and α were not changed. The networks were left to run for as long as possible. The results in the table were for the latest training game reached by each network.

Table 4: Performance after 1000 test games against Pubeval agent

	Basic		Pip count		Pip count + hit probability	
Architecture	Max at	Win rate	Max at	Win rate	Max at	Win rate
Monolithic	159,000	61.20%	94,000	54.00%	100,000	60.00%
Separate Modular	111,000	28.20%	34,000	12.20%	12,000	20.20%
Hybrid Modular	46,000	17.30%	78,000	36.00%	29,000	16.80%

Based on the results of table 4, for the monolithic network, the inclusion of pip count and hit probability features did not improve the performance of the network. The basic features resulted in the best performance against pubeval agent and converged at a much earlier stage, figure 9. Continuing the training however, resulted in making the performance slightly worst as shown in figure 9a after the 190,000-game mark.

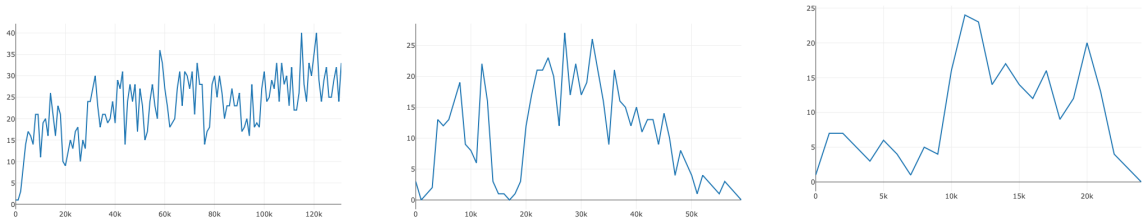
For separate strategy modular network, the inclusion of the extra features resulted in the divergence of the network. A single game would take over 1,000 turns, and the performance was



(a) Basic features (b) Pip count included (c) Pip count and hit probability included

Figure 9: Monolithic network against pubeval agent

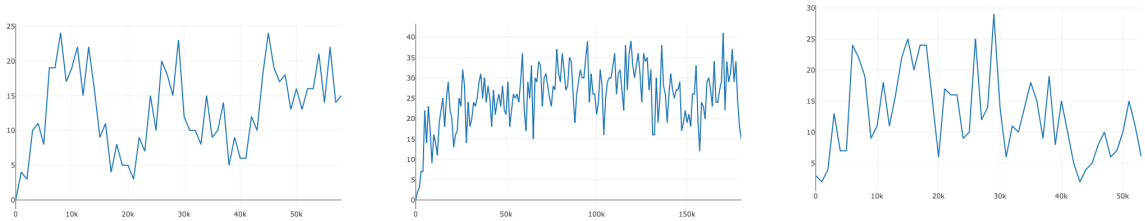
declining, figure 10. The basic input features gave the best results for this modular network with the trendline increasing at a slow rate reaching a limit.



(a) Basic features (b) Pip count included (c) Pip count and hit probability included

Figure 10: Separate strategy modular network against pubeval agent

The hybrid strategy modular network benefited from the inclusion of the pip count feature, but it diverged for when both pip count and hit probability were included, figure 11. For the case with basic features the results were very unstable. No trends for either the basic features or all features instances plots were observed but the instance with pip count showed an increasing trendline during early training games to reach a limit after the 60,000-game mark.



(a) Basic features (b) Pip count included (c) Pip count and hit probability included

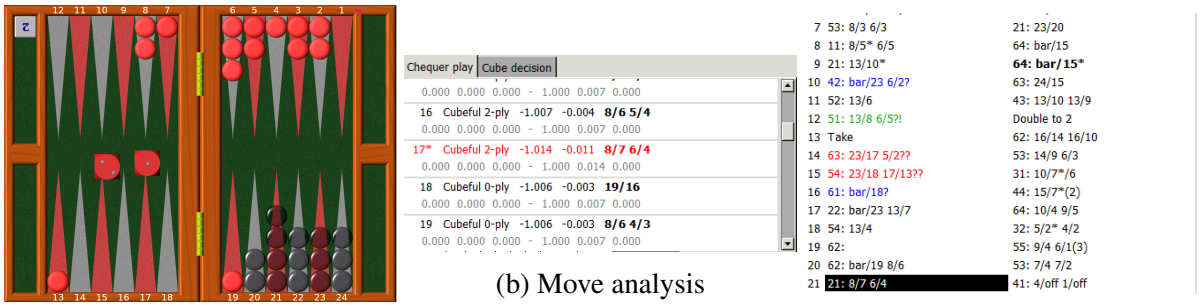
Figure 11: Hybrid strategy modular network against pubeval agent

B Match analysis

The best performing networks were tested against GNU-Backgammon's world class level agent for 10 games. The best performing networks were the monolithic network with basic input features, the separate strategy modular network with basic input features and the hybrid strategy modular network with pip count included.

B.1 Monolithic Network

Monolithic network made bad decisions in some positions but generally picked one of the 3 top moves during the game. The monolithic network tended to use primes as part of its strategy; this was observed by watching the network play against Pubeval and against GNU-Backgammon agent. As the game reached the racing stage, the network made average decisions by picking moves that were in the middle range, 17th move picked out of 25 possible moves, based on GNU-Backgammon analysis, figure 12. The monolithic network won 5 out 10 games against GNU-Backgammon agent.



(a) Game Board

(b) Move analysis

(c) Full game analysis

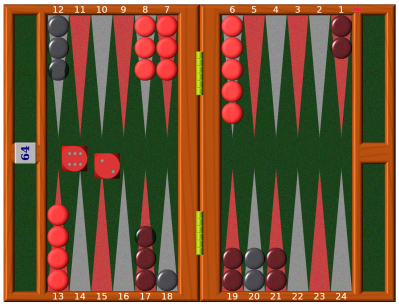
Figure 12: Monolithic network (Red) against GNU-Backgammon world class level agent (Black) - racing stage

B.2 Separate Strategy Modular Network

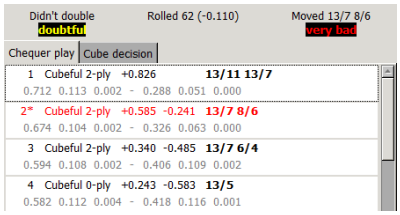
Separate strategy modular network played well first roll positions in some games and made many top decisions at the racing stage. It occasionally picked one of 3 top moves during the game but made more bad decisions than the monolithic, figure 14. In terms of strategy, the seperate strategy network tended to use some back game moves when it was about to lose and short primes, wall of 3 fields, otherwise. The hybrid modular network won 2 out 10 games against GNU-Backgammon agent.

B.3 Hybrid Strategy Modular Network

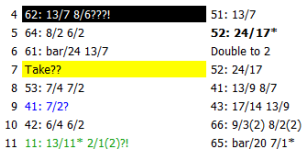
Hybrid strategy modular network played well first roll positions, made many top decisions during the racing stage of the game. It occasionally picked one of 3 top moves during the game but made more bad decisions than the monolithic network but less than the seperate strategy modular network, figure 14. The hybrid network tended to pick moves that reduced the pip count which proved the influence of including the pip count to the features. In terms of strategy, the hybrid



(a) Game Board



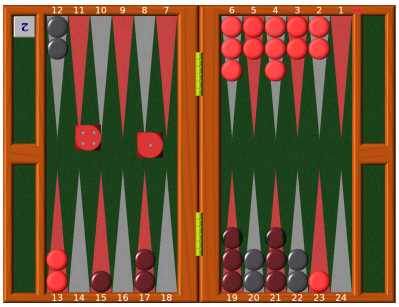
(b) Move analysis



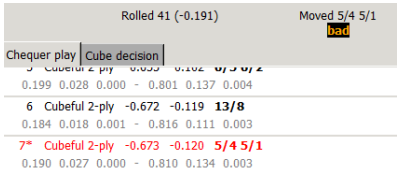
(c) Full game analysis

Figure 13: Seperate strategy modular network (Red) against GNU-Backgammon world class level agent (Black)

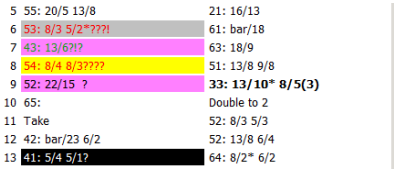
network tended to build primes but also kept 2 checkers at the 13th field for the majority of the game. It didn't follow any resemblance to a back-game strategy. The hybrid modular network won 3 out 10 games against GNU-Backgammon agent.



(a) Game Board



(b) Move analysis



(c) Full game analysis

Figure 14: Hybrid modular network (Red) against GNU-Backgammon world class level agent (Black)

V EVALUATION

In this section, an evaluation of the solution is given based on the results obtained. The solution strengths and limitations along with potential improvements are discussed.

A Solution Strength and limitations

Changing the training parameters, λ in particular, strongly influenced the general performance of the networks as evident from the results in table 3. Reducing λ greatly improved the performance of the networks. The inclusion of more input features did not provide any noticeable improvement to the monolithic network performance against the benchmark agent, pubeval. However, it caused instability to the separate modular network's training. Convergences was not guaranteed and different runs for the same configurations could either diverge or converge. This made training incredibly unstable with various results obtained in the various runs. A better training strategy would be promising in solving this problem. Similar issue could be observed with

the hybrid modular network results, figure 11. Although, the inclusion of pip count improved the performance and gave decent results for the basic input features, It diverged when all features were included.

The general performance and game analysis of the monolithic network was within the results obtained by Tesauro (2002) and Papahristou and Refanidis (2012). On the other hand, for modular networks, as the architecture of the networks was based on the architecture's description in Boyan's (1992) and Wiering's (2010) work along with notes included in GNU-Backgammon software (Silver 2006), the actual implementation did not match their results. Overall, the results proved that the monolithic network outperformed both modular networks. This result contradicted the expected state-of-art results that the modular networks would perform better than the monolithic network; GNU-Backgammon agents are ranked as one of the best AI agents for backgammon (GameSite 2017). In terms of the best modular architecture, the best hybrid modular network performed better than the best separate modular network against the benchmark agent, pubeval.

The results obtained from GNU-Backgammon software provided some insight into the strategy used by all networks. The priming strategy seemed to be the dominant strategy followed by all networks. The analysis emphasised the weakness of the monolithic network during the racing stage as noted in Frnkranz 's survey (2000). The hybrid network provided a better strategy for the racing stage than the monolithic network, but it did not provide a better strategy for the other positions. The seperate strategy modular networks tended to build short primes and use some sort of back game when at losing its about to lose.

Returning to the main aim of this project, whether the hybrid modular network had resulted in a new strategy remains a subjective matter. The hybrid network did learn a different strategy, however as evident from the results in table 4 and the analysis in figure 14, it was clear that the strategy was decent but not great.

B Improvements

If given a chance to repeat this project, the advance objectives would have been explored and the training strategy for the modular networks would be re-evaluated to solve the observed divergence issue. Doubling cube would be included as part of the actions selection algorithm. In addition, the action selection algorithm would be extended to include more lookahead, performing a 2-ply or a 3-ply search which had been proven to provide better performances as evident from results collected by Depreli (2012). As most bugs in the code has been fixed and a better performance measure had been included by using pubeval and GNU-Backgammon software, more parameters could be experimented. Various online resources claimed short training time even for 200,000 games, so different packages that would be more optimized to use the CPU would be considered to reimplement this project and reduce the training time.

VI CONCLUSIONS

The implemented solution had it flows but showed some promise. Minor tweaks to the networks parameters resulted in better performance against the benchmark agent, pubeval, as indicated by the experiments with the learning parameters λ and α along with extracting additional features from the game state and including them to the network's input units. Long training time limited the number of parameters experimented, but decent results were obtained. Modular net-

work architecture had improved play at some stages of the game and resulted in different strategies than the monolithic network architecture, however, it did not perform better in terms of the overall game strategy. This was highlighted by the match analysis for the networks against GNU-Backgammon world class level agent. Further improvements to the modular network could be made especially to the activation algorithm for the sub-networks, it could be either by redefining the conditions or incorporating the Meta-pi architecture; the addition of a network to handle the networks activations. Other combinations of backgammon strategies could result in interesting new strategies that would be better than the combination of the back and priming games studied in this project. The priming game strategy was generally more dominant in the hybrid modular network as observed from the test games and match analysis. But, the hybrid modular network performed better than the modular network with separate strategies. Carrying this project forward, more complex search algorithms for action selection could be incorporated along with the decision algorithm for using doubling cube during the game. This algorithm could produce better strategies at the expense of increasing the complexity of the implemented solution.

References

Azaria, Y. & Sipper, M. (2005), 'Gp-gammon: Genetically programming backgammon players', *Genetic Programming and Evolvable Machines* **6**(3), 283–300.
URL: *https://doi.org/10.1007/s10710-005-2990-0*

Boyan, J. A. (1992), Modular neural networks for learning context-dependent game strategies, Technical report, Masters thesis, Computer Speech and Language Processing.

Depreli, M. (2012), 'Bot comparison final table'.
URL: *http://www.extremegammon.com/studies.aspx#D2012*

Fleming, J. (2016), 'td-gammon'.
URL: *https://github.com/fomorians/td-gammon*

Frnkranz, J. (2000), Machine learning in games: A survey, *in* 'MACHINES THAT LEARN TO PLAY GAMES, CHAPTER 2', Nova Science Publishers, pp. 11–59.

GameSite (2017), 'extreme gammon'.
URL: *http://www.extremegammon.com/default.aspx*

Hannun, A. (2013), 'backgammon'.
URL: *https://github.com/awni/backgammon*

Keith, T. (1995), 'Backgammon galore glossary'.
URL: *http://www.bkgm.com/glossary.html*

Papahristou, N. & Refanidis, I. (2012), On the design and training of bots to play backgammon variants, *in* L. S. Iliadis, I. Maglogiannis & H. Papadopoulos, eds, 'Artificial Intelligence Applications and Innovations - 8th IFIP WG 12.5 International Conference, AIAI 2012, Halkidiki, Greece, September 27-30, 2012, Proceedings, Part I', Vol. 381 of *IFIP Advances in Information and Communication Technology*, Springer, pp. 78–87.
URL: *https://doi.org/10.1007/978-3-642-33409-2_9*

Pollack, J. B. & Blair, A. D. (1998), 'Co-evolution in the successful learning of backgammon strategy', *Machine Learning* **32**(3), 225–240.
URL: *https://doi.org/10.1023/A:1007417214905*

Silver, A. (2006), 'All about gnu'.
URL: *http://www.bkgm.com/gnu/AllAboutGNU.html*

Sutton, R. S. & Barto, A. G. (1998), *Introduction to Reinforcement Learning*, 1st edn, MIT Press, Cambridge, MA, USA.

Tesauro, G. (1992), Temporal difference learning of backgammon strategy, *in* D. H. Sleeman & P. Edwards, eds, 'Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992', Morgan Kaufmann, pp. 451–457.

Tesauro, G. (1993), 'Ftpable benchmark evaluation function'.
URL: *https://www.zabkat.com/blog/tesauro-backgammon-reinvented.html*

Tesauro, G. (1994), 'Td-gammon, a self-teaching backgammon program, achieves master-level play', *Neural Computation* **6**(2), 215–219.
URL: *https://doi.org/10.1162/neco.1994.6.2.215*

Tesauro, G. (1998), 'Comments on "co-evolution in the successful learning of backgammon strategy"', *Machine Learning* **32**(3), 241–243.
URL: *https://doi.org/10.1023/A:1007469231743*

Tesauro, G. (2002), 'Programming backgammon using self-teaching neural nets', *Artif. Intell.* **134**(1-2), 181–199.
URL: *https://doi.org/10.1016/S0004-3702(01)00110-2*

Wiering, M. A. (2010), 'Self-play and using an expert to learn to play backgammon with temporal difference learning', *JILSA* **2**(2), 57–68.
URL: *https://doi.org/10.4236/jilsa.2010.22009*