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 School of Engineering and Computing Sciences, Durham University

Abstract —

A Context/Background

Backgammon and many other board games have been widely regarded as an ideal testing ground for exploring a variety of concepts and approaches in artificial intelligence and machine learning. Using Reinforcement Learning techniques to play backgammon has given insight to potential strategies that were overlooked in the past.

B Aims

The aim of this project is to find a new strategy for backgammon; a hybrid of known strategies will be used as the basis for the new strategy.

C Method

A modular neural network architecture will be used to incorporate the different backgammon strategies. The priming and back games will be used for this project. Two modular networks will be implemented and trained, one that will include the 2 strategies separately and another one that will include a hybrid of the 2 strategies. A single neural network based on TD Gammon will also be implemented and trained. The modular networks will be evaluated against the single network and against each other. Test games against an expert user will be included to validate the new strategy.

D Proposed Solution

A python package that will include modules to train and test the networks using self-play. It will also include a module for setting both a textual and a graphical user interfaces to play against the trained networks.

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 his checkers home and bear them off while preventing the opponent from doing the same thing (Keith 1995). Figure 1 illustrates the basic setup that will be used for this project.

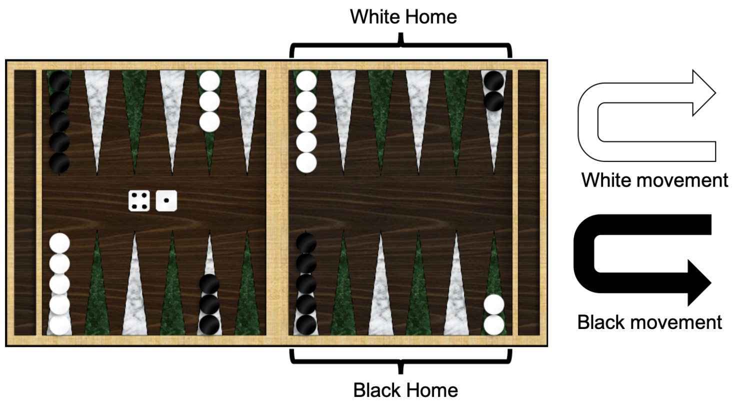

Figure 1: Backgammon board setup

B Reinforcement Learning

A reinforcement learning algorithm allows an agent to learn from its experience generated by its interactions with an environment (Sutton & Barto 1998). Temporal difference (TD) learning is a class of reinforcement learning which uses the difference between two successive positions for backpropagating the evaluations of the successive positions to the current position. There is a whole family of temporal difference algorithms known as TD(λ)-algorithms which are parametrised by the value λ which makes the agent looks further in the future for updating its value function (Sutton & Barto 1998).

C Early work

TD-Gammon of Tesauro (1992) used TD(λ) reinforcement learning methods with a single neural network that trains itself and learns from the outcomes to be an evaluation function for the game of backgammon. Extending on the work of Tesauro, Boyan (1992) showed that the use of different networks for different subtasks of the game can perform better than learning with a single neural network for the entire game. Many AI software for Backgammon use similar modular neural network architecture; GNU-Backgammon uses 3 different neural networks for their evaluation function (Silver 2006).

D Project Overview

This project will focus 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 will be evaluated by comparing it to another network that includes the strategies separately. Combinations of 2 strategies will be studied: **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).

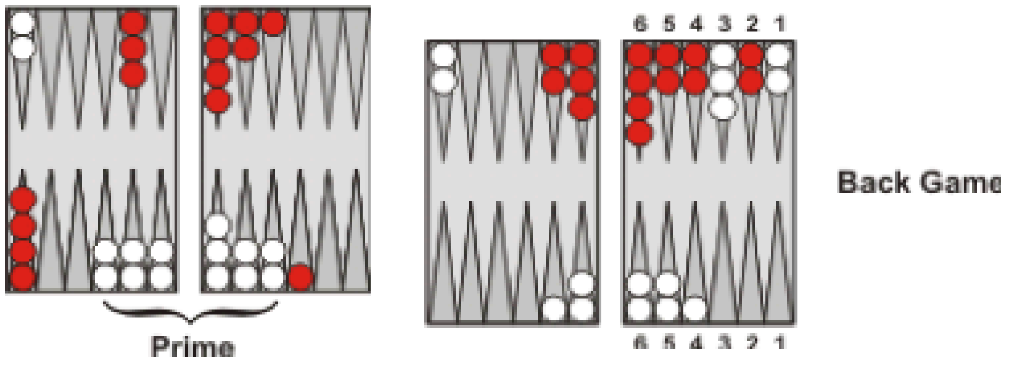

Figure 2: Illustration of Priming and Back games

E Research Questions

The following questions will be investigated in this project:

1. Would using hybrid strategies result in the agent learning a new strategy? or would one strategy be more dominant than the other?
2. How effective would the hybrid strategies be over the strategies being included separately?
3. How would be adding doubling cube to the network evaluation influence the learning outcome?

F 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 will be able to use the trained network to evaluate and make moves.
- **User interface and human agent**: This will be a simple command line interface which takes user inputs to make moves and to play against the trained networks. Another complex implementation will be a graphical user interface which captures the user clicks for making moves.
- **Testing suit**: This will be used to run all tests and evaluations to be done on the networks.
- **Project Report**: All tests and evaluations will be recorded and analysed in this report.

II DESIGN

A Requirements

Table 1 includes the functional requirements for this project.

Table 1: List of Functional Requirements

ID	Requirement	Priority
FR1	A module for Backgammon game must be implemented. This will include the actual board setup with the rules and constraints of the game e.g. legal moves and the dice role	High
FR2	An AI agent should be created such that it uses a neural network to evaluate legal moves/actions to play backgammon and uses 1-ply search algorithm to pick the best legal move/action	High
FR3	A module for the neural network should be created. It should support the functionalities required for the neural network e.g. updating weights through back-propagation, saving, and restoring the network's meta-data	High
FR4	A module for training the neural network should be created	High
FR5	Single Neural Network should be implemented and trained based on Tesauro's TD Gammon, extends FR4 AND FR3	High
FR6	Modular Neural Network that includes Holding and Priming Game strategies separately should be implemented and trained, extends FR4 AND FR3	High
FR7	Modular Neural Network that includes a hybrid of Holding and Priming Game strategies should be implemented and trained, extends FR4 AND FR3	High
FR8	A testing module should provide capabilities of evaluating and testing the neural networks, used for testing FR5 to FR7	High
FR9	2-ply search algorithm should be implemented and incorporated into AI agent in FR2	Medium
FR10	Textual User interface for a Human agent to play against the AI agent from FR2 should be implemented	Medium
FR11	Graphical User interface for a Human agent to play against the AI agent from FR2 should be implemented, extending on FR10	Low
FR12	Doubling cube evaluation. A heuristic function should be implemented to determine when to use the doubling cube or to accept or refuse the double. This should be included to FR4	Low

Table 2 includes the non-functional requirements for this project.

Table 2: List of Non-functional Requirements

ID	Requirement	Priority
NFR1	Trained networks should return the outcome from forward propagation within 40ms	Medium
NFR2	The AI agent should pick a legal move/action within 1s. In other word the search algorithm for the best move/action should return a value within 1s	Medium
NFR3	The AI agent should pick a legal move 100% of the time	High

B Algorithms

B.1 Temporal Difference, TD(λ)

In backgammon, the dice rolls guarantee sufficient variability in the games so that all regions of the feature space will be explored. This characteristic made it perfectly suited for learning from self-play as it overcomes the known Reinforcement Learning problem with the trade-off between exploration and exploitation (Frnkranz 2000). 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 $(0, 1)$. The ideal value of λ is between 0.7 and 0.6 based on Tesauro's (2002) results. α on the other hand, should ideally decay. $e_t(s)$ 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).

B.2 1-ply search algorithm

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). Initially, the AI agent will use a 1-ply search algorithm to pick the best legal action for the current turn. Each action will be evaluated in the neural network, forward feeding, and the action with the maximum outcome will be returned (Tesauro 1992).

C System Components

Python 3.6 will used as the language for this project. The neural networks will be implemented using TensorFlow package. TensorFlow will be used as it allows to generate training progress summary, to save and to restore a trained network. Figure 3 shows the expected structure and component dependencies of this project.

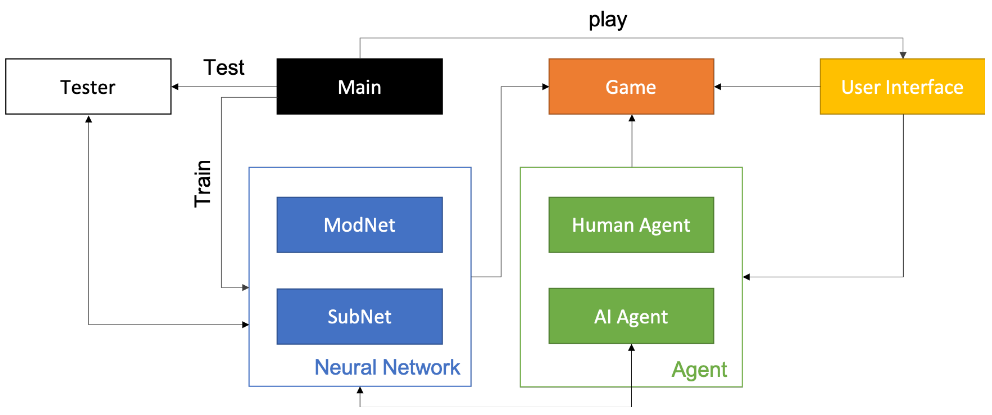

Figure 3: System components and dependencies

C.1 Main

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

C.2 Game

This module will hold the game setup and define the rules and constraints of the game e.g. take an action, find legal moves and game board setup. An open source implementation taken from a GitHub repository, backgammon (Hannun 2013), of this module will be refactored and modified for the use of this project.

C.3 User Interface

This module will be handle generating the command line interface (textual interface) and the graphical user interface. The use of either interface will depend on the availability of pygame, python package; pygame allows generating a graphical user interface in python. Similarly, to the Game module, an open source implementation taken from a GitHub repository, backgammon (Hannun 2013), of this module will be refactored and modified for the use of this project.

C.4 Agents

There are 2 types of agents that will be implemented for this project:

- **A human agent**, an interactive agent which will take user inputs either from the command line or by capturing the user clicks though a GUI to make a move/action.

- **AI agent** will use a modular neural network to determine the move/action for the current turn. A list of legal moves is obtained from the game module and an action will be picked based on the search algorithm.

C.5 Modnet

This module will define the operations for extracting features from the game board, testing and training neural network/s. This module will heavily depend on Subnet module. For modular networks, a game-specific gating program will be implemented in this module to determine which sub-network will be suitable to a given input, set of extracted features. For the different modular neural networks to be trained for this project, different instances of this module will be created as each modular network will require different gating program. The monolithic neural network won't need the gating program.

C.6 Subnet

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

C.7 Tester

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

D Neural Network Architecture

D.1 Monolithic Neural Network

For this network, it will be based on Tesauro's (1994) TD Gammon implementation; a fully-connected feed-forward neural networks with a single hidden layer. Initially, the architecture will consist of one input layer I with 298 units which will consist of 288 raw inputs representing the checkers configuration on the board, each field in the board is represented by 6 units as there are 24 fields so 144 total units and each player has their own configuration represented separately making the total 288. In addition, 8 input units will be included as expert features, table-3. Including expert features proved to provide better outcomes from the network (Tesauro 2002). Lastly, 2 input units to represent the current player's token. As part of the network architecture, there will be one hidden layer H with 50 units and one output layer O with 1 unit representing the winning probability. At later stages the output layer will be extended to include 4 units, 2 units for the player winning the game and winning by a gammon and 2 units for the player losing the game and losing by a gammon. The network will have weights w_{ih} for all input units I_i to hidden unit H_h and weights w_{ho} for all hidden units H_h to output unit O_o. The weights will be initialized to be random values; hence the initial strategy is a random strategy. Each hidden unit

Table 3: 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
pip_count_1	pip count for current player
pip_count_2	pip count 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
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

and output unit will have a bias b_h and b_o with sigmoid activation. Each bias will be initialized to an array of constant values of 0.1. Figure 4 includes the layout of the neural network.

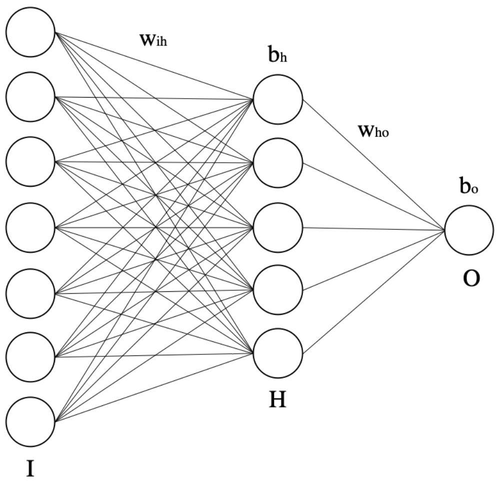

Figure 4: Neural Network architecture

A lot of implementations of this network are available in the open source community and will be used as a reference for this project; two code bases from GitHub, backgammon (Hannun 2013) and td-gammon (Fleming 2016), will be used and referred throughout the life cycle of this project. The main challenge with using open source code will be debugging the code and validating it.

D.2 Modular Neural Network

To incorporate backgammon strategies, modular neural architecture will be implemented. The strategies will be represented by different monolithic neural networks that will be activated when certain board configurations are reached. This approach has been implemented by Boyan (1992) and what most recent software such as GNU-Backgammon (Silver 2006) follow. The modular networks that will be implemented for this project will consist of a combination of the following networks:

1. One network for back game positions; the player is behind in the race (pip count) but has two or more anchors (two checkers at one field) in the opponent's home board. This network will also be used when there are many checkers on the bar.
2. One network for priming games; if the player has a prime of 4-5 fields (a long wall of checkers)
3. One network for a hybrid priming and back game; combines the conditions for both games
4. One default network for all other positions

Initially each network will have the same layout/architecture as the monolithic neural network, however the networks don't necessarily need to have the same layout.

There are two types of Modular Neural Network architectures that will be implemented in this project:

- **Designer Domain Decomposition Network (DDD) -**This architecture will be used in the first stages of the project. The DDD network consists of n monolithic neural networks and a hard-coded gating program, figure 5. The gating program, based on the work of Boyan (1992), partitions the domain space into n classes. For this project the n classes are represented by the different backgammon strategies. In both forward feeding and backward

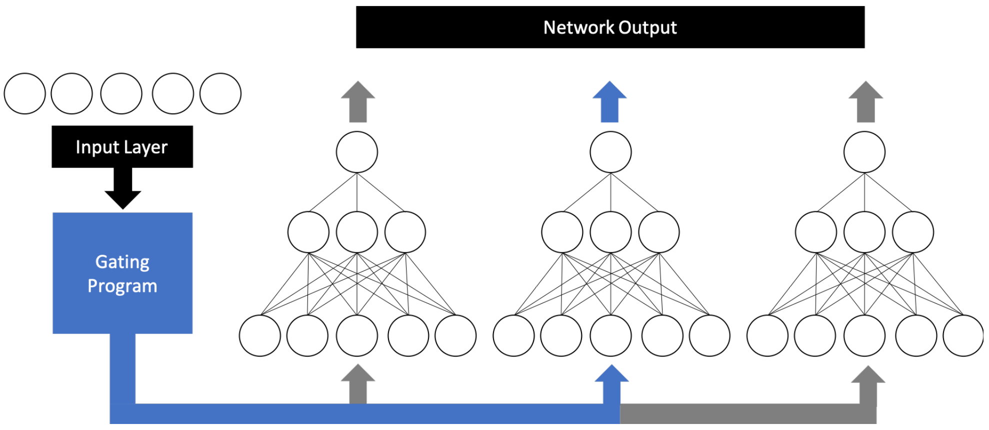

Figure 5: The DDD Network

  propagation, the gating program will be called to select an appropriate network for the current board extracted features/inputs. Exactly one network will be activated at any time.

- **Meta-Pi Networks -** The gating program in the DDD network will suffer from a blemish effect; the stiffness introduced by hard coding the triggers for the networks results in a non-smooth evaluation as noted by Boyan (1992). The Meta-Pi network is a trainable gating network, figure 6. This network will be used to determine the most suited network to be triggered based on a given input. The benefit of this approach is that it will allow the agent to develop a smoother evaluation function. This network will be introduced in later stages of the project once all networks have been trained.

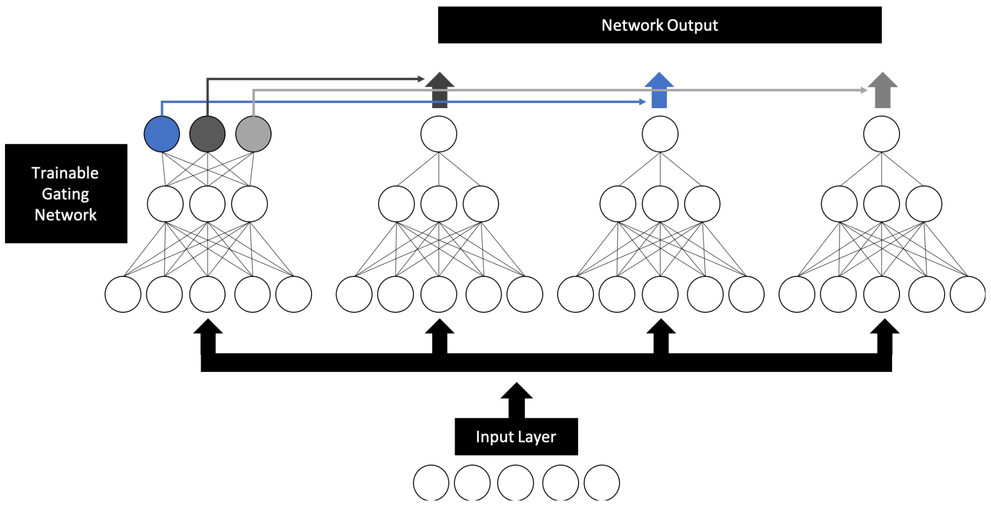

Figure 6: Meta-Pi gating network

E Training

A total of 3 networks will be trained; 1 monolithic network and 2 modular networks. The training strategy will be based on the work of Tesauro (1994), Boyan (1992) and Wiering (2010). The monolithic network and the modular networks with the DDD architecture will be trained by self-play using TD(λ) learning with a decaying learning rate α starting from 0.1 until 0.01 and a decaying value for λ starting from 0.9 until 0.7; exponential decay will be used for both learning rate α and λ. Each network will be trained on 500,000 games. After each 5000 games, the network will be tested for 1000 games against the previously stored version of the network itself. This will allow to monitor the progress of the network's training.

Before running the full training with 500,000 games, different configurations for each network will be tested e.g. the addition of expert features as part of the input layer. The number of training games will be reduced to 100,000 and the resultant network will then be tested against the other configurations of that network for 2000 games. The best combination of configurations will be used to fully train each network.

Once DDD networks are finished training, meta-pi gating network will be trained following the same strategy.

F Evaluation

Both modular networks will be tested for 5000 games against the monolithic network. The result obtained will give an indication of the general performance of the networks and effectiveness of the strategies. A random sample of those games will be recorded and analysed to evaluate the strategies followed by both modular networks. In addition, both modular networks will be tested for 5000 against each other to measure the performance of the hybrid strategy.

To further evaluate the networks, few test games against an expert-level backgammon human player will be conducted. The expert will be asked to provide feedback about the AI agent's actions and strategy of each modular network. The expert won't be told any details regarding the network that the AI agent will be using at first. After the feedback is obtained from the expert player, they will be made aware of the AI agent that used the network with the new strategy and will be asked to do few more test games and the validate if the network performs a hybrid strategy successfully and it is indeed a new strategy.

G Extensions

G.1 Doubling Cube

As an extension of this project, doubling cube will be included as part of evaluating the actions taken by the AI agent. This will be implemented by including a heuristic function as part of the action evaluation process. The implementation will be initially based on Tesauro's (2002) work, then an implementation using Crawford rule will be tested. The best trained network will be retrained with the doubling cube taken into consideration. The newly trained network will be tested for 5000 games against the older version of the network.

G.2 2-ply search algorithm

Following Tesauro's (2002) work and recent backgammon software, the search algorithm used to determine the current turn's move will affect the general training outcome; it is evident from results collected by Depreli (2012) that better results are expected from 2-ply and 3-ply search. The 2-ply algorithm will be implemented as follows:

Algorithm 1 2-ply search

```
 1: $legalActions \leftarrow$ getLegalActions()
 2: $actions \leftarrow []$
 3: for $action$ in $legalActions$ do
 4:     takeAction($action$)
 5:     $features \leftarrow$ extractFeatures()
 6:     $v \leftarrow$ getModelOutput($features$)        ▷ get the outcome of the taken action, forward propagation
 7:     $actions$.append(($action, v$))
 8:     undoAction($action$)
 9: sortActions($actions$)                              ▷ sort actions in descending order
10: $topActions \leftarrow$ getSubList($actions, 5$)    ▷ get first five actions
11: $actionsOutcome \leftarrow []$
12: for $action$ in $topActions$ do
13:     takeAction($action$)
14:     $outcomes \leftarrow$ runAllPossibleDiceRollForOpp()   ▷ 20 legal moves considered for each roll, the best outcome of the move of each roll will be returned
15:     $avgOutcome \leftarrow$ avg($outcomes$)
16:     $actionsOutcome$.append(($action, avgOutcome$))
17:     undoAction($action$)
    return getBestRankAction($actionsOutcome$)          ▷ best rank action
```

It is important to note that this computation could take some time and in timed games this could be unfavourable.

References

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.

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

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

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. (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. (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*