

# 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 has been the subject of many studies involving artificial intelligence and machine learning. TD-Gammon of Tesauro () had demonstrated the impressive ability of machine learning techniques to learn to play games. TD-Gammon used reinforcement learning techniques with a Neural Network (NN) that trains itself to be an evaluation function for the game of backgammon, by playing against itself and learning from the outcome (). However, the monolithic neural network soon reached its limitation an outcome studied by Boyan () and a modular neural network becomes more suitable to overcome this limitation. The newest software for Backgammon build on top of the modular architecture such as eXtreme Gammon () and GNUBG ().

## ***B Aims***

The aim of this project is to find 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. Test games against an expert user will be included to validate the new strategy.

## ***D Proposed Solution***

A python script that allows a user

***Keywords —*** Backgammon; Reinforcement Learning; Modular Neural Network;

## **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 (). Figure 4 illustrates the basic setup that will be used for this project.

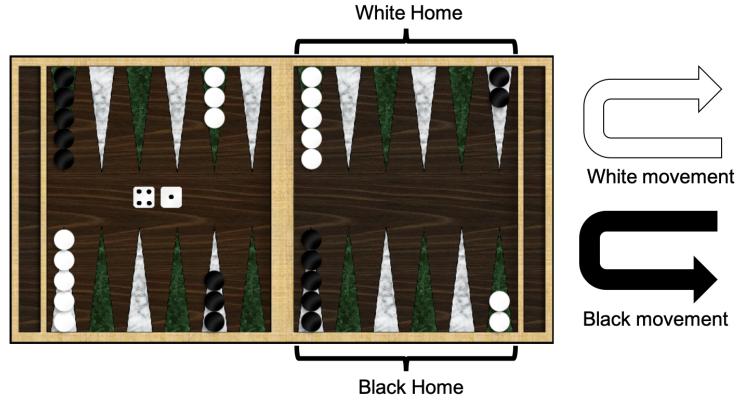


Figure 1: Backgammon board setup

## B Early work

Backgammon and many other boardgames have been widely regarded as an ideal testing ground for exploring a variety of concepts and approaches in artificial intelligence and machine learning. TD-Gammon of Tesauro () had demonstrated the impressive ability of machine learning techniques to learn to play games. TD-Gammon used reinforcement learning techniques with a single neural network that trains itself to be an evaluation function for the game of backgammon, by playing against itself and learning from the outcome (). Extending on the work of Tesauro, Boyan () showed that the use of different networks for different subtasks of the game can improve over learning with a single neural network for the entire game. Many AI softwares for Backgammon use modular neural network architecture; GNU-Backgammon () uses 3 different neural networks.

## C Project Overview

This project will focus on modular neural network architecture in incorporating a hybrid of backgammon strategies to find a new 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 both anchors as long as possible and force his opponent to bear in or bear off awkwardly (). **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 ().

## D Research Questions

The following questions will be investigated in this project:

1. How will combining game strategies for backgammon influence learning outcome?

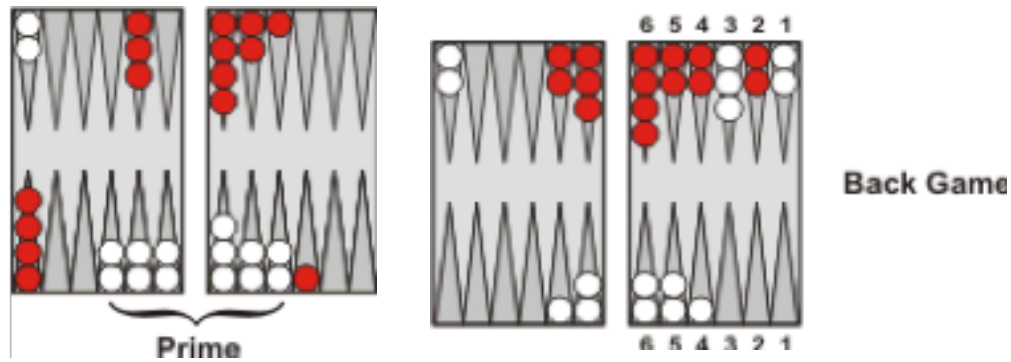


Figure 2: Illustration of Prime and Back games

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

### ***E Deliverables***

- minimum objectives
  1. create a basic backgammon player to be used as a bench player
  2. create a player that uses the Priming Game
  3. create a player that uses the Back Game
  4. create a player that combines the Priming and Back Games
  5. evaluate the performance of the players with strategies against the basic player
- intermediate objectives
  1. create a basic player with a bigger hidden network
  2. evaluate the performance of the best player with a strategy against the new basic player
- advanced objectives
  1. include doubling cube to the players
  2. perform a rollout for one of the games
  3. test the effect of including other Backgammon Game strategies

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 nueral network to evaulate legal moves/actions to play backgammon and a greedy search algorithm to pick the best legal move/action                               | High     |
| FR3  | A module for the nueral network should be created. It should support the funtionalities required for the nueral network e.g. updating weights through back-propagation, saving and restoring the network meta-data | High     |
| FR3  | A module for the training and testing the nueral network should be created   | High     |
| FR3  | Monolithic Nueral Network should be implemented and trained based on Tesauro's TD Gammon   | High     |
| FR7  | Modular Nueral Network that includes Holding and Priming Game strategies seperately should be implemented and trained  | High     |
| FR7  | Modular Nueral Network that includes a hybrid of Holding and Priming Game strategies should be implemented and trained   | High     |
| FR7  | A testing module should provide capabilities of evaluating and testing the nueral networks   | High     |
| FR8  | depth (n-ply) search algorithm should be implemented and incoporated into AI agent in FR2  | Medium   |
| FR9  | Textual User interface for a Human agent to play against the AI agent from FR2 should be implemented   | Medium   |
| FR10 | Graphical User interface for a Human agent to play against the AI agent from FR2 should be implemented   | Low      |
| FR10 | Doubling cude evaluation. A hueristic function should be implemented to determine when to use the doubling cube or to accept or refuse the double  | Low      |

## II DESIGN

### A Requirements

### B Algorithms

#### B.1 Temporal Difference, $TD(\lambda)$

Temporal difference will be used

#### 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 (). For the first stages of the project, the AI agent will use a 1-ply

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     |

search algorithm to pick the best legal action for the current turn. Each action will be evaluated in the neural network, forward propagation, and the action with the maximum outcome will be returned.

## C System Components

Python 3.6 will be used as the language for this project. The neural networks will be implemented using tensorflow package. Tensorflow will be used as it is easy to use, to generate training progress summary, to save and to restore the trained network. Figure 3 shows the expected structure and component dependence of this project.

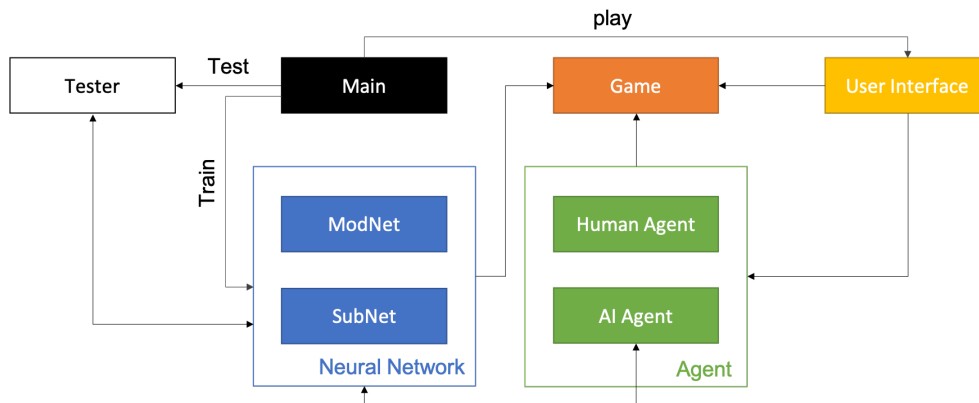


Figure 3: System components and dependencies

### C.1 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 Awni github repository of this module will be refactored and modified for the use of this project.

## C.2 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 through a GUI to make a move/action.

- **AI agent** will use a modular nueral 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.3 Modnet

This module will define the operations for extracting features from the game board, testing and training nueral 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 nueral 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 nueral network won't need the gating program.

### C.4 Subnet

This module will include the Nueral 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 propagation and backpropagation algorithms. All networks created for this project will use an instance of this module; networks used in modular nueral network and monolithic nueral network. The architecture of these networks will be explained in the next section.

## D *Nueral Network Architecture*

### D.1 Monolithic Nueral Network

For this network, it will be based on Tesauro's TD Gammon implementation (Tesauro 1992, 2002); a fully-connected feed-forward nueral 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 thier own configuration represented seperately making the total 288. In addition, 8 input units will be included as expert features, table-3. Those expert features proved to provide better outcome from the network (). 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. 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 intialized to be random values, hence the initial strategy is a random strategy. Each hidden unit and output unit will have a bias  $b_h$  and  $b_o$  with sigmoid activation. Each bias will be intialized to an array of constant values of 0.1.

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 basis from github \_ and \_ will be used and referred through out the life cycle of this project. The main challenge with using open source code will be debugging the code and validating it.

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 beared off   |
| off_pieces_2 | percentage of pieces that opponent has beared 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         |

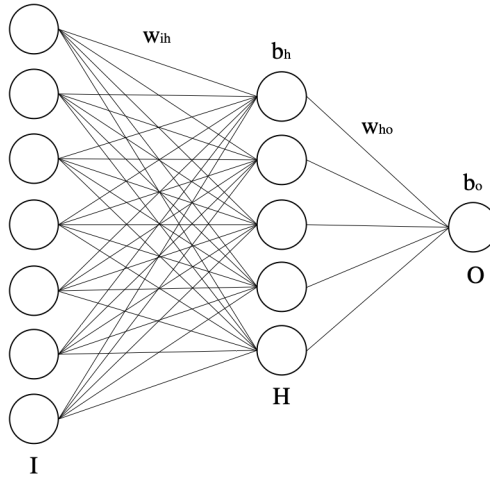


Figure 4: Nueral Network architecture

## D.2 Modular Nueral Network

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

1. One network for racing game; the checkers cannot be hit anymore by another checker or the checkers layout is close to this outcome.
2. One network for back game positions; the player is behind in the race (pipcount) 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.
3. One network for priming games; if the player has a prime of 4-5 fields (a long wall of checkers)

4. One network for a hybrid priming and back game; combines the conditions for both games
5. One default network for all other positions

Those networks will be used to allow the learning agent to better evaluate the positions and reach a strategy that is close to the strategy followed by professional players from each network. 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. At later stages, different layouts will be tested to configure each neural network e.g. racing game network does not need inputs for all fields since most checkers will be beared off or close to being beared off. This could help reduce computing time and thus increase the efficiency of the training.

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 (year), partition the domain space into n classes. For this project the n classes are represented by the different backgammon strategies; there are other possible decompositions for the backgammon strategies but Racing, Back and Prime games will be the focus for this project. In both forward feeding and backward propagation, the gating program will

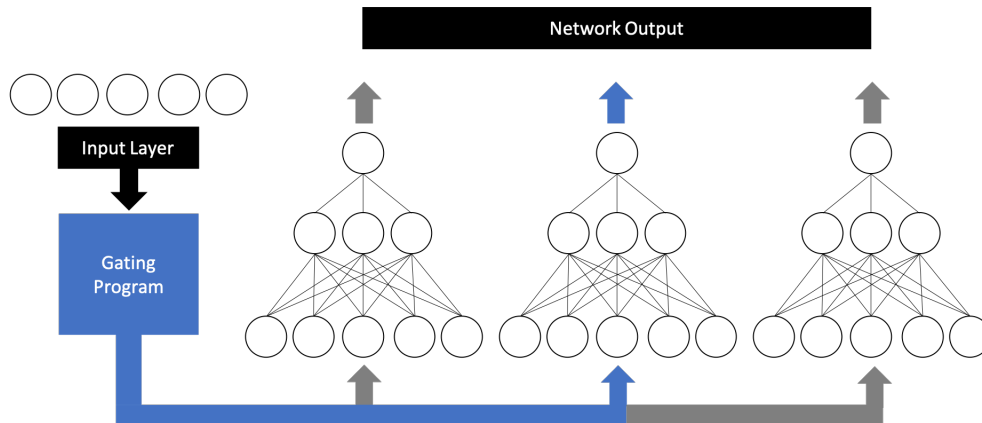


Figure 5: The DDD Network

be called to select an appropriate network for the current board extracted features (inputs). Exactly one network will be activate at any time.

- **Meta-Pi Networks** - The gating program in the DDD network will suffer from a blemish effect as noted by Boyan (year). 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 give input. The benefit of this approach is that it will reduce the stiffness introduced by hard coding the triggers for the networks and will allow the agent to develop a smoother evaluation function. This network will require the other networks to be fully trained and only the meta-pi network would be updated in the training process. Thus, this network will be introduced in later stages of the project once all networks have been trained.



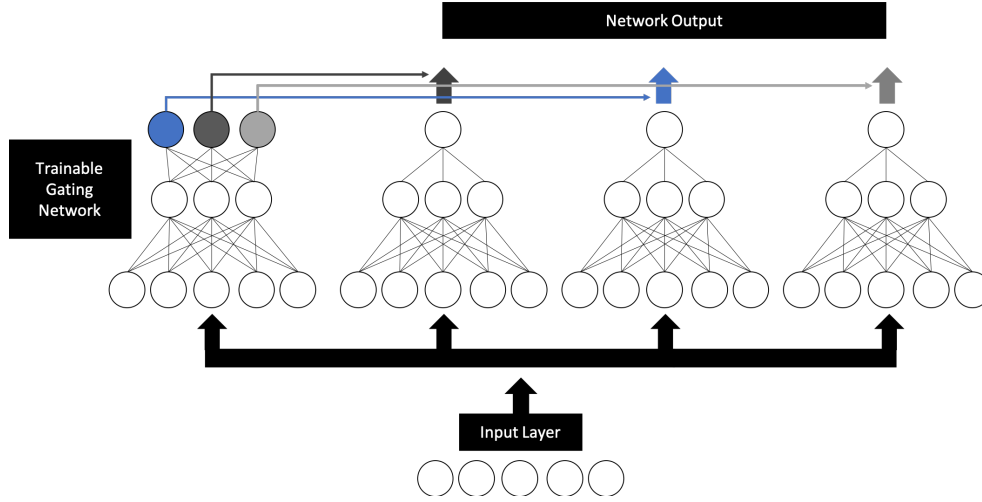


Figure 6: Meta-Pi gating network

## E Training

A total of 5 networks will be trained; 1 monolithic network and 4 modular network with different strategy combinations.

The training strategy will be based on the work of Tesauro, Boyan and Weiring. The monolithic network and the modular networks with the DDD architecture will be trained by self-play using  $TD(\lambda)$  learning with a decaying learning rate  $\alpha$  strating from 0.1 until 0.01, a discount factor  $\gamma$  of 1 and a decaying value for  $\lambda$  starting from 0.9 until 0.7; exponetial decay will be used for both learning rate  $\alpha$  and  $\lambda$ . Each network will be trained on 500,000 games. After each 5000 games, the network will be tested for 2500 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 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, but it will be continuously tested against the DDD network .

## F Evaluation

All 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 modular networks and effectiveness of the implemented architecture.

Test games with certain starting configurations, 5 different configurations for each racing, back and prime games, will also be used to evaluate the strategies followed by each modular network, answering the question if the modular networks have indeed learned the strategy imposed for the specific network. The dice roll for each game will be set such that all networks will have the same dice roll each turn. The opponent will be set to be an ai agent that use the monolithic network. 1000 games will be ran for each configuration. For a random sample of games, the

actions taken by the AI agent will be recorded and manually traced for each network. Other statistical values will also be collected and averaged such as the number of turns taken and the end outcome (win/lose).

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 for each modular network. The expert won't be told any details regarding the model that the AI agent will be using.

The last evaluation will be for the best trained network in which 5000 games for each modular network against the other neural networks will be tested.

## ***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 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 research and recent backgammon softwares, the search algorithm used to determine the current turn's move will effect the general training outcome; it is evident from (backgammon league table) that better results are expected from 2-ply and 3-ply search. The 2-ply algorithm will be implemented as follows:

It is important to note that this computation could take some time and in timed games this could be unfavourable. The only difference between 1-ply and 2-ply is that the 1-ply won't check all the possible rolls of the opponents and stops after evaluating the immediate best action.

## **References**

---

**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
```

---