CS 182 Final Project:
Building A Backgammon AI

Virgile Audi
vaudi@g.harvard.edu

December 11, 2016

1 Introduction

Backgammon is one of the oldest board game known to man. We can trace back its origins to Ancient Egypt, dating back 3500 BC, and most of the neighbouring empires had their own version of what resembles today Backgammon. Backgammon is a two player zero-sum game where the objective is to removing all of one's pawns before his opponent. Players move their pawns according to the roll of two dice. It is therefore both a game of strategy and luck (which we will see will impact the use of certain algorithms). It is said that learning to play backgammon is relatively easy, but learning to play backgammon well is extremely hard. The goal of this project was to build an AI capable of playing backgammon. The methodology was inspired by DeepMind's work on Alpha Go. To simplify, the methodology I decided to use, and for which I have not been able to find similar work in the backgammon literature, is to first used supervised learning in the form a multiclass logistic regression as the initial state of reinforcement learning using genetic algorithms and Q-learning.

Figure 1: Backgammon Board

2 Background and Related Work

The literature is mainly dominated by the work of Gerald Tesauro who developped TD-Gammon in 1992 [3]. His AI, which is still considered one of the best (if not the best) in the backgammon sphere, consists of a neural network trained using TD-(λ). I tried to stay away of Neural Networks as this was not the focus of this class and sticked to linear evaluation functions. As mentioned in the introduction, I was inspired by the work on "AlphaGo" [2]. I was interested to see how much of a head start would supervised learning could give reinforcement learning, and hopefully compensate for the use of linear functions instead of neural nets. For the reinforcement learning part of this project, I adapted methods from two papers. The first one [4] presented encouraging results of the use of TD learning where the AI would face an already trained opponent, contrary to TD-Gammon where the AI would face himself. The second paper [1] showed how we could use a certain type of genetic algorithm, which will be presented in the following sections, to update the weights of the evaluation function. Nevertheless, both of these methods were originally presented in the context of uninformed reinforcement learning i.e. where the initial weights of the evaluation function are initialised randomly. I, on the contrary, adapted these methods (some with success, some without unfortunately) by initialising these algorithms using the results of the supervised learning from expert moves.

3 Problem Specification

In order to explain formally how the problem was formulated mathematically, a few basic rules of backgammon should be explained.

3.1 Backgammon's Rules

In the introduction, I mentioned that the goal of backgammon was to remove all of ones pawns before his/her opponent according to the rolls of two dice but this is still pretty vague. Let's dig into more details. Each player needs to first bring every pawns of his into his camp which following the directions of the white and red arrows below on figure 2. Let's showcase a example. Consider the white player rolls (6,5), he can effectively move any pawn in the direction of the white arrow 6 or 5 position forward. A move is acceptable for the white player if the destination is either empty (no pawns white or black are present), already occupied by one or more white pawns or if only one black pawn is occupying the space. In this last case, the white player captures his opponents pawn, who will then need to free his pawn in his oppenent camp. Figure 3 shows for instance the possible moves for the white player when rolling (6,5).

Backgammon can therefore been seen as a succession of boards and rolls of dice. The problem that therefore needs to be solved is to which board should one player move to given the current board and the roll of the dice. How can we evaluate the boards so that the player can move to the board with the highest value?

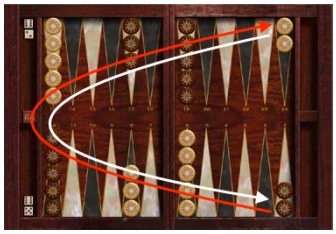

Figure 2: General Dynamics

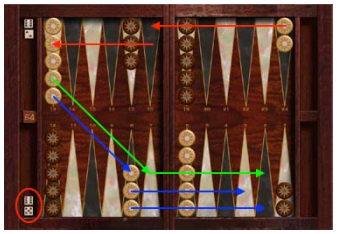

Figure 3: Possible Moves

3.2 Formal Description

If we define the states to be a tuple (board, roll) and action be the possible moves given the board and the roll, then Backgammon fits perfectly in the litterature of Markov Decision Process. Formally:

- Let $s = (b, r) \in \mathcal{S} = \mathcal{B} \times \mathcal{R}$ be the states
- Let $a \in \mathcal{A}$ with $a : s \to b \in \mathcal{B}$

Unfortunately, it is possible to show using combinatorics that the size of $\mathcal{S}$ is greater than 18×10^{18} which makes any type of exact algorithms intractable in reasonable time or with the computational power at my disposal. To resolve this issue, I decided to represent each board as a feature vector. If I experimented with different types of feature vectors, features included for instance:

- Number of left alone pawns in each quadrant
- Number of pairs of pawns in each quadrant
- Number of enemy pawns imprisoned in each quadrant using the last action
- Number of safe pawns (pawns which are after the last oppenent's pawn)

Using this feature vector $\mathbf{f}$ and a weight vector $\mathbf{w}$, we could therefore score each resulting board using a linear function: $Q(s, a) = \mathbf{f}_{s,a} \cdot \mathbf{w}$.

In the following section, we will detail how we estimated the weights w_i using both supervised and reinforcement methods.

4 Approach

The approach used is a two step process: first, using supervised learning and second, improve the results using reinforcement methods in the form of either genetic algorithms or Q-learning.

4.1 Training using expert moves

Once each state was reduced to a feature vector and scoring each resulting state was a linear function, using multiclass logistic regression seemed like the right approach. Indeed, the goal at each step is to use the action:

$$\text{argmax}_{a \in \mathcal{A}}(\mathbf{f}_{s,a} \cdot \mathbf{w})$$

which is equivalent to:

$$\text{argmax}_{a \in \mathcal{A}} \left(\frac{\exp(\mathbf{f}_{s,a} \cdot \mathbf{w})}{\sum_{a' \in \mathcal{A}} \exp(\mathbf{f}_{s,a'} \cdot \mathbf{w})} \right)$$

by representing the choice of next moves as a probability distribution. I then used stochastic gradient descent combined with a cross-entropy loss function and L2 regularisation:

$$\tilde{L}(y, \hat{y}) = \hat{y}_a \log y_a + \frac{\lambda}{2} \sum w_i^2 \equiv L(y, \hat{y}) + \frac{\lambda}{2} \sum w_i^2$$

where $\hat{y}$ is a one-hot vector where a corresponds to the index of the action taken by the "pro" and y is the distribution outputed given the current weights i.e. $y_a = \frac{\exp(\mathbf{f}_{s,a} \cdot \mathbf{w})}{\sum_{a' \in \mathcal{A}} \exp(\mathbf{f}_{s,a'} \cdot \mathbf{w})} = softmax(\mathbf{f}_{s,a} \cdot \mathbf{w})$

The algorithm used goes as follows:

Algorithm 1 Multiclass Logistic Regression

procedure TRAINING($\{f_i\}, \{\hat{y}_i\}, \eta, \lambda$)
  Initialise **w** randomly
  while training criterion not met **do**
    Sample a training feature vector f_i
    Compute the loss $L(\hat{y}_i, y_i; \mathbf{w})$
    Compute the gradient $\frac{\partial L}{\partial \mathbf{w}}$
    Update the weights: $\mathbf{w} \leftarrow \mathbf{w} - \eta(L(\hat{y}_i, y_i; \mathbf{w}) + \frac{1}{2}\lambda \mathbf{w})$
  end while
  return w
end procedure

4.2 Genetically updating the weights

As suggested in [1], we could find the optimal weights by following this algorithm:

Algorithm 2 Genetic Algorithm

procedure UPDATE($\mathbf{w}, \eta, M_{score}, M_{it}$)
  while $it < M_{it}$ **do**
    $\mathbf{w}^\star \leftarrow$ GENERATE_OPPONENT($\mathbf{w}$)
    $winner, diff \leftarrow$ PLAY_TOURNAMENT($\mathbf{w}, \mathbf{w}^\star, M_{score}$)
    if $winner =$ challenger **then**
      if $diff < M_{score}/2$ **then**
        $w_i \leftarrow (1 - 2 * \eta) * w_i + 2 * \eta * w_i^\star$
      else
        $w_i \leftarrow (1 - \eta) * w_i + \eta * w_i^\star$
      end if
    else Continue
    end if
  end while
end procedure

Note that in backgammon, if a player wins the game with his opponent not having removed one of his own pawns, then the winner wins a game with value multiplied by 2. The functions used in the pseudo code are presented below:

Algorithm 3 Auxiliary Functions

procedure GENERATE_OPPONENT(**w**)
  $\mathbf{w}^\star = []$
  for i in 1:|**w**| **do**
    $w_i^\star = w_i + Unif(-0.5, 0.5)$
  end for
  return $\mathbf{w}^\star$
end procedure

procedure PLAY_TOURNAMENT($\mathbf{w}, \mathbf{w}^\star, M_{score}$))
  $s_{champ} = 0$ and $s_{chal} = 0$
  while $s_{champ} < M_{score}$ and $s_{chal} < M_{score}$ **do**
    Play a game with **w** vs $\mathbf{w}^\star$
    if Champion wins **then**
      $s_{champ} +=$ value of the game
    else
      $s_{chal} +=$ value of the game
    end if
  end while
  return $winner, abs(s_{champ} - s_{chal})$
end procedure

In other words, if the challenger wins, move in the direction of the challenger. How fast you move in its direction depends on the magnitude of the win.

4.3 Approximate Q-learning

The second reinforcement learning algorithm implemented is Approximate Q-Learning as presented in lectures. Applied to the game of backgammon, the pseudo-code is presented in Algorithm 4 below.

5 Experiments and Results

5.1 Data

We processed data from games played by pros taken from the website http://paulspages.co.uk/bgvaults/. Processing the matches and invidual games composing these matches, this yielded a dataset of about 27000 moves, 90% of which were used for training and 10% were left for testing.

Algorithm 4 Q-Learning Game

procedure Q_LEARNING($\mathbf{w}, \mathbf{w}_{opp}, r, \gamma, M_{it}$)
Initialise $\mathbf{w}$
while it¡M_{it} **do**
$\mathbf{w} \leftarrow$UPDATE($\mathbf{w}, \mathbf{w}_{opp}, r, \gamma$)
end while
return w
end procedure

procedure UPDATE($\mathbf{w}, \mathbf{w}_{opp}, r, \gamma$)
while Game is not finished **do**
Roll Dice
Evaluate all possible boards given moves $Q(s, a, \mathbf{w}) \forall a$ given current board s
Move to best state
Opponent rolls and plays his move given $\mathbf{w}_{opp}$ to get to s'
Roll Dice
Evaluate all possible boards given s' and the possible actions
Update $w_i \leftarrow w_i + \alpha(r + \gamma \max Q(s', a') - Q(s, a)) f_i(s, a)$ where r=10 if win, r=-10 if loss, r=0 o/w
end while return w
end procedure

5.2 Experiments

I ran mutliple experiments with the same methodology. The main difference was in the size of the feature vector extracted from the board. In the following section, small will refer to a 21-entry feature vector where the features were only concerning the player's pawn. On the other hand, large will refer to a 41-entry feature vector which includes the 21 previous features as well as features about his opponent's positions. I trained two MLR based using the two feature vectors.To show that the success of the MLR, we present below the evolution of the accuracy on held-out test data over the number of epochs:

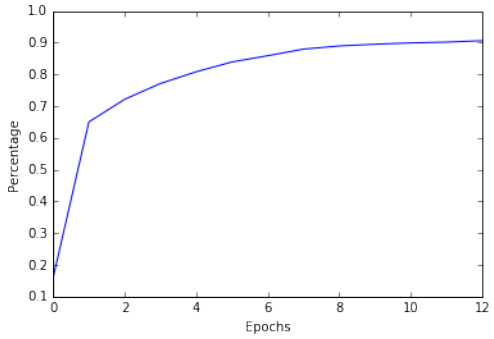

Figure 4: Evolution of the Loss on held-out test data for the large feature vector

We then switched to using the genetic algorithm to improve on these two MLR as well as using the genetic algorithm without prior informed initialisation, in order to have a fair comparision. We also ran Q-learning with the two feature vectors.

The method of comparing the accuracy on ”pro” moves will be used as the main method for testing the success of an algorithm. We will also present results on direct match-ups between the different AIs.

5.3 Accuracy on Pro Moves

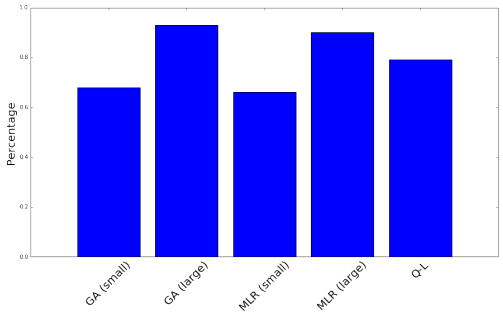

Figure 5: Comparing accuracies of the different AIs

The key points of this figure are:

- larger feature vector yields better results than the smaller one, which would be expected.
- Genetic Algorithms increased the performance for both feature vectors
- Q-learning reduced the performance

5.4 One VS The Other

	GA (L)	**MLR (S)**	**MLR (L)**	**Q-L**
GA (S)	(8-15)	(15-5)	(15-6)	(16-1)
GA (L)		(15-0)	(16-0)	(15-0)
MLR (S)			(1-15)	(8-15)
MLR (L)				(15-2)

Table 1: Match-up Results

These results seemed for most of them in accordance with the accuracy results ! Genetically enhanced with large features is the best AI trained during this project !

6 Discussion

6.1 Why did Q-Learning not work as expected?

I ran into multiple issues with the Q-Learning part of this project. The first major issue was divergence. This was fixed by lowering drastically the learning rate as well as the discounting factor. The second issue I faced was that the Q-learning made the AI actually weaker. I did limit the impact by increasing the magnitude of the rewards but still did not manage to make the AI stronger with Q-learning. This is in my opinion due to fact that I was using a TD(0) algorithm whereas TD-Gammon uses $TD(\lambda)$. By using a longer path to update the weights, the system becomes much less sensitive to the stochastic nature of backgammon and the rolls of dice that could potentially changed the state of the game.

6.2 Potential Improvements

The next step would be to definitely switch from TD(0) to TD(λ). Also I would also want to use a feed forward neural network as the evaluation function. Indeed, if the accuracy is extremely promising, I can still beat the AI myself (I am a decent player) because the trickiest moves, the AI still misses. A Neural Network should catch more of these !

A System Description

All the code is on github at: www.github.com/virgodi/pygammon.

You can play against the computer by typing: 'python backgammon.py' in a terminal window !

Training and algorithms are contained in ipython notebooks in the folder notebooks_and_data

References

[1] Jordan B. Pollack & Alan D. Blair. Why did td-gammon work?

[2] Chris J. Maddison Arthur Guez Laurent Sifre George van den Driessche Julian Schrittwieser Ioannis Antonoglou Veda Panneershelvam Marc Lanctot Sander Dieleman Dominik Grewe John Nham Nal Kalchbrenner Ilya Sutskever Timothy Lillicrap Madeleine Leach Koray Kavukcuoglu Thore Graepel & Demis Hassabis David Silver, Aja Huang. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[3] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3), 1995.

[4] Marco A. Wiering. Self-play and using an expert to learn to play backgammon with temporal difference learning. *J. Intelligent Learning Systems & Application*, 2:57–68, 2010.