# On the Design and Training of Bots to play Backgammon Variants

Nikolaos Papahristou and Ioannis Refanidis

University of Macedonia, Department of Applied Informatics,
Egnatia 156, Thessaloniki, 54006, Greece
nikpapa@uom.gr, yrefanid@uom.gr

**Abstract.** Recently, a backgammon bot named *Palamedes* won the first prize in backgammon at the 16th Computer Olympiad. *Palamedes* is an ongoing work aimed at developing intelligent bots to play a variety of popular backgammon variants. Currently, the Greek variants *Portes*, *Plakoto* and *Fevga* are supported. A different neural network has been designed, trained and evaluated for each one of these variants. This paper presents the details of the architecture and the training procedure for each case. New expert features as inputs to the networks are also introduced, whereas experimental results demonstrate improvement over previous versions of *Palamedes*.

**Keywords:** TD(λ), Neural Networks, Self-Play, Backgammon, Plakoto, Fevga

## 1 Introduction

Backgammon is one of the oldest board game of chance and skill that is very popular throughout the world with numerous tournaments and many popular variants. Variants of any game usually aren't as interesting as the standard version, but often offer a break in the monotony of playing the same game over and over again. In this paper we examine three popular variants of backgammon in Greece, namely *Portes*, *Plakoto* and *Fevga*, collectively called *Tavli.* In a traditional Tavli match these three games are played in turn, one after the other, until one of the players reaches a predefined number of points (usually seven). *Palamedes* [6] (Fig. 1) is an ongoing project dedicated to offer expert-level playing programs for Tavli and other backgammon variants.

The objective for each player of virtually all variants is to move all his checkers to the last quadrant (called the *home board*), so he can start removing them; a process called *bearing off*. The player that removes all his checkers first is the winner of the game. Players may also win a double game (2 points) when no checker of the opponent has been beared-off. Portes is essentially the same with standard backgammon; the main differences are: (1) the absence of the doubling cube and (2) the absence of triple wins (also called backgammons). The complete set of rules for standard backgammon, Plakoto, Fevga and other variants can be found in [2].

In previous work [7,8], following the successful example of TD-Gammon [12,13,14,15] and other top playing backgammon programs, we trained neural networks (NN) using temporal difference learning to play *Portes*, *Plakoto* and *Fevga*, three variants very popular in Greece and neighboring countries. In this paper we

present in detail the complete algorithm for the training of our NNs. We also present, for the first time, our Portes bot that won the gold medal in the Computer Olympiad in Tilburg, in November 2011. Furthermore, for the Plakoto and Fevga variants, we present new results that improve the performance upon our previous bots by adding new features, and we explain the logic behind our approach.

This paper is organized as follows. Section 2 reviews our training scheme for backgammon variants. Section 3 presents the expert features used in Portes and the additional expert features for Plakoto and Fevga, whereas Section 4 shows the experimental results. Finally Section 5 makes concluding remarks and discusses future work.

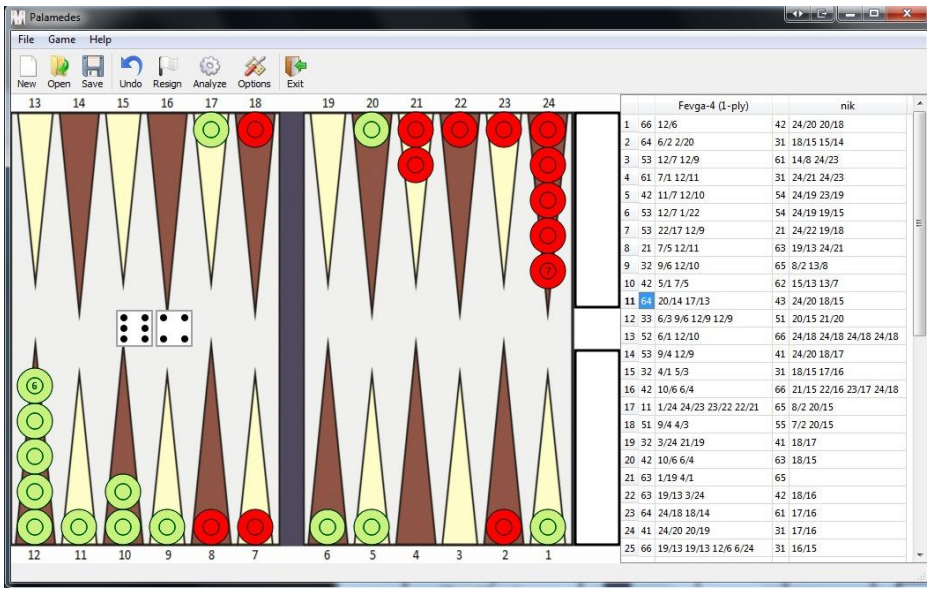


**Fig.1.** ***Palamedes*****:** A program for playing backgammon and variants

## 2 Training procedure

This section presents the details of the procedure used to train the neural networks for the three backgammon variants examined in this paper.

### 2.1 Neural Network Architecture

The core function of the neural network is to score game positions. At any time when the program needs to decide which move to play from a set of legal moves, it scores all available states resulting from the current position/roll and selects the one with the highest estimated expected value. We use multilayer perceptrons (MLP) trained using the backpropagation algorithm [17].

The input layer of the NN is comprised of features capturing the position of the checkers on the board, also referred as "raw features", plus features that capture important concepts of the game, also called "expert features". The set of the features selected for each game is presented in Section 3.

We use one hidden layer in our backgammon NNs. The number of hidden neurons is 160 for backgammon, 100 for Fevga and 100 for Plakoto. These numbers were chosen based on preliminary experiments. A higher number of hidden neurons increases performance cost for evaluating each state. This results in increased thinking time for each move, especially when utilizing lookahead in greater depths (Section 4). Thus, the number of hidden neurons chosen is a compromise between performance and computational cost.

Three output neurons are used in the output layer, codenamed W, WD and LD. These correspond to the minimum probabilities needed by the bot in order to make an estimation of the game-theoretic value of a state: W is the probability of winning the game regardless of the number of points (single or double); WD is the probability of winning only a double game and LD is the probability of losing a double game. Both the hidden and the output layers use sigmoid activation functions for each neuron.

Using the above architecture, the procedure of obtaining an estimation of the game-theoretic value of each state is straightforward: set the inputs of the NN according to the board positioning, execute the forward-propagate procedure of the NN to update the outputs, and finally linearly combine the outputs according to the following formula: V= 2 * W - 1 + WD – LD.

## 2.2 Training the NN using TDL

Training a neural network requires training examples in a supervised learning setting. We use TD(λ) algorithm [10] and the NN's backpropagation algorithm to update the TD error. The exact training procedure is summarized in Algorithm 1. This training scheme, named *reverse offline learning with target recalculation*, was selected among several similar self-play methods [8].

In the adopted training procedure, the updates are applied (Lines 5-15), after a self-play game (Line 4) is ended, starting from the last position of the game and ending at the first (Line 5). At each time step, we recalculate the target for each update (Lines 9-11) in order to get as much accuracy for the estimation of the example label as possible. The function *encoding* (Lines 9, 13), encodes the raw and expert features in their predefined positions at the input layer. Note that the value of the next state is inverted (Line 11). This is necessary because the NN plays the game for both sides always as the first player. When all the moves up to the first are updated, the algorithm starts a new self-game producing the moves according to the updated NN. The procedure is repeated until the selected stopping criterion is satisfied. Possible stopping criteria are: (1) a predefined number of self-play games is reached or (2) no more performance improvement according to a predefined benchmark is found after a prespecified number of self-play games.

**Algorithm1**. Training a backgammon NN using TD(0)

```
    // nn:  the neural network that we want to train
    //  nn.inputs: a vector representing the input layer
    //  nn.outputs: a vector representing the output layer (W, WD, LD)
    //  nn.target: a vector representing the target of the update
    // states: a vector holding the all the positions of a game
1.  nn.initialize(input layer size, hidden layer size, output layer size = 3, learning rate α)
2.  randomize(nn)      // randomize all weights to [-0.5, 0.5]
3.  while (stopping condition) do
4.     states = selfplaygame(nn)
5.     for (t=T to 1 step -1) do
6.       if(states(t) is terminal)
7.          nn.targets = reward(states(t))
8.       else
9.          nn.inputs = encoding(states(t+1))
10.         nn.forwardpropagate()          // calculate outputs
11.         nn.targets = invert(nn.outputs)
12.      endif
13.      nn.inputs = encoding(states(t))
14.      nn.forwardpropagate()   // calculate outputs
15.      nn.backpropagate()      // apply backpropagation algorithm
16.    endfor
17. end while
```

Algorithm 1uses TD(λ) with λ=0, that is the current state is updated only according to the estimation of the next state (Lines 9-11). Thus the target of the update is $V_{target}(s_t) = V(s_{t+1})$. If we want the target of the update to be based on future move estimates of the gameas well ($0<\lambda\leq1$), we can use the forward view of TD(λ) [11] and the target of the update becomes

$$V_{target}(s_t) = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}V(s_{t+n}) + \lambda^{T-t-1}V(s_T)$$

In case of λ>0, lines 8-10 of Algorithm 1 must be changed accordingly. Similarly to $V(s_{t+1})$, all values $V(s_{t+n})$ for *n* being any odd number must be inverted.

The updates of the network weights are done incrementally and not in a batch setting. This procedure is similar to stochastic or "online" training [17]. The main difference is that there are no fixed labels in the training examples; the labels are given by TD(λ). We prefer incremental training because it has been shown to perform at least equally to the standard batch training using fewer computational resources [17].

## 2.3 Choosing a learning rate α and a λ parameter

One of the advantages of incremental training is that one can use a larger learning rate than in a batch setting. We also made some experiments with different values of λ with mixed results. In the Plakoto variant, values of λ>0.6 resulted in divergence, whereas lower values sometimes became unstable. So it was decided to keep λ=0 for this variant. For Portes and Fevga variants it was possible to increase the λ value without problems and this always resulted in faster learning, but unlike other reported results [16], final performance did not exceed experiments with λ=0.

**Table 1.** Selected values of α and λ parameters.

| Games Trained | Portes | Plakoto | Fevga |
|---|---|---|---|
| 0-10000 | λ=0.7 α=1 | λ=0 α=0.3 | λ=0.7 α=1 |
| 10000-100000 | λ=0.7 α=0.3 | λ=0 α=0.3 | λ=0.7 α=0.3 |
| 100000-250000 | λ=0.7 α=0.1 | λ=0 α=0.1 | λ=0.7 α=0.1 |
| 250000-500000 | λ=0 α=0.3 | λ=0 α=0.1 | λ=0 α=0.3 |
| 500000-1500000 | λ=0 α=0.1 | λ=0 α=0.1 | λ=0 α=0.1 |
| 1500000-5000000 | λ=0 α=0.1 | λ=0 α=0.01 | λ=0 α=0.01 |
| 5000000- | λ=0 α=0.01 | - | - |

Previous experiments were conducted with constant λ and α=0.1. Following the above preliminary experiments we use a decreasing value for λ and α for the experiments in this paper (with the exception of Plakoto where λ is kept constant to zero). Starting with high values of λ=0.7 and α=1 we gradually decrease these values when performance starts to flatten. The exact values of these parameters are shown in Table 1. Using this setup the performance of Plakoto and Fevga variants maxes out at 5 million games and Portes at around 15 million games.

## 4. Expert Features

The features included in the input layer of each NN are divided to "raw" and "expert" features. Raw features present to the network the placement of each checker in the board while expert features are important game concepts that would otherwise be very difficult for the NN to infer from the raw encoding alone. The raw features of Plakoto and Fevga are presented in [7], while the raw features of our Portes NN are exactly the same as used in [14]. The remaining of this section presents the selected expert features for the Portes game as well as the new expert features that we used in Plakoto and Fevga. The remaining expert features of Plakoto and Fevga are described in [7].

### 3.1 Expert features for Portes/Backgammon

All the expert features of our Portes/Backgammon bot are shown in Table 2. The features capture important game playing concepts according to the current literature from expert backgammon players. For example EnterFromBar_1 and EnterFromBar_2 capture the concept of home board strength. This feature however is useless when the position has no contact (race feature). The NN takes care of combining the features in the correct way taking the current position into account. Additionally, the hidden neurons can create features not existent in the expert list if necessary. For example, we found that the prime formation (six consecutive made points) was handled correctly by the program so we did not include it in the list of expert features even if it is an important concept. The features PipDiff_1, PipDiff_2, PipBearoff_1, PipBearoff_2 were normalized to the [0, 1] interval by a dividing with 60.

**Table 2.** Expert features for the Portes/backgammon variant.

| Feature name | Description |
|---|---|
| HitProb_1 | Probability of one player checker being hit on the next roll |
| HitProb_2 | Probability of two player checkers being hit on the next roll |
| Race | Boolean feature showing the position is a no contact position |
| PipDiff_1 | Pipcount difference when the player is behind (when ahead = 0) |
| PipDiff_2 | Pipcount difference when the player is ahead (when behind = 0) |
| PipBearoff_1 | Pipcount to bearoff for player on roll |
| PipBearoff_2 | Pipcount to bearoff for opponent |
| EnterFromBar_1 | Probability of player entering from bar |
| EnterFromBar_2 | Probability of opponent entering from bar |
| OppContain_1 | Probability of opponent's last checker escaping from player's home board |
| OppContain_2 | Probability of opponent's second to last checker escaping from player's home board |
| UsContain_1 | Probability of player's last checker escaping from opponent's home board |
| UsContain_2 | Probability of player's second to last checker escaping from opponent's home board |

**Table 3.** Expert features for the Plakoto variant.

| Feature name | Description |
|---|---|
| Race | Boolean feature showing the position is a no contact position |
| PipDiff_1 | Pipcount difference when the player is behind (when ahead = 0) |
| PipDiff_2 | Pipcount difference when the player is ahead (when behind = 0) |
| PipBearoff_1 | Pipcount to bearoff for player on roll |
| PipBearoff_2 | Pipcount to bearoff for opponent |
| ChFrontOfPin_1 | Number of player checkers in front of last pin when the player has the opponent pinned in the player's homeboard |
| ChFrontOfPin_2 | Number of opponent checkers in front of last pin when the opponent has the player pinned in the opponent's homeboard |
| Esc_Prob1 | Escape probability of player's last made point |
| Esc_Prob2 | Escape probability of opponent's last made point |

### 3.2 New expert features for Plakoto

After manual examination and with the help of comments from users that downloaded *Palamedes*, we identified two key problems of our Plakoto bots. The first one presented itself in positions when the bot has pinned the opponent inside the bot's home board. In such positions it is advisable for the bot to "stack" its checkers in the pinned point whenever possible so as to prolong the duration of the pin even in the bearoff situation. Such a strategy most often leads to a double game. However our bots were positioning their checkers as if it was a normal bearoff, greatly reducing their chances for a double game. This problem was addressed by adding the ChFrontOfPin_1 and the ChFrontOfPin_2 features. These two features were scaled to [0, 1] interval by dividing each by 14. We also added the Esc_Prob1 and EscProb2

features hoping that the bot can advance its made points more fluidly, not leaving behind made points that cannot escape easily. Finally we added five features from Portes that are relevant to Plakoto as well. The complete set of features is shown in Table 3.

### 3.3 New expert features for Fevga

The most important concept in the Fevga variant is the existence of a prime formation. In previous work we addressed this by adding one binary feature for every type of prime when it was encountered in the game. While this resulted in the desired effect of the NN learning the concept of making primes when necessary, it did not always understand when it was important to prevent the opponent from making primes of its own. The bot could not understand by this feature alone when the opponent was close to making a prime so as to take immediate measures to disrupt his plan. The inclusion of 2-ply look-ahead improved the situation as now the bot had access to the next moves of the opponent but it would be desirable to have this knowledge without reverting to the computational expensive procedure of looking ahead at greater depths.

To address this problem we changed the binary features of making primes in the following way: When a prime is made the feature is set to one as before. When there is no prime present, instead of setting the feature to zero, we replaced it by a heuristic that computes the probability of making the prime. This was done both for the primes of the bot as well as for the primes of the opponent. Computing accurately this heuristic is very complex and takes much time especially for middle game positions. In order to keep the computational requirements low, we compute the heuristic only for the most common scenario: when there is only one checker left to make the prime. Positions where the prime needs two or more checkers to be achieved are less frequent and usually have smaller probability of success. Thus, the resulting heuristic is a compromise between accuracy and executing time.

These updated features resemble the way we added the pinning probabilities in the Plakoto variant [7]. It has the advantage of putting knowledge in the NN while at the same time keeping low the size of the inputs. We also added the features PipDiff_1, PipDiff_2, PipBearoff_1, PipBearoff_2 of Portes and Plakoto, because they are relevant to Fevga as well.

We also experimented by combining the above new features with the intermediate reward procedure during the training of Fevga3 and Fevga5 bots [8]. Such a procedure results in a strategy that tries to build primes and maintain them at all cost. While the resulting performance was higher than previous bots, it was lower than Fevga6, i.e. without the intermediate reward. One possible explanation is that without the intermediate reward the bot can identify situations where a prime is not the best course of action. It seems that finding exceptions to the rule of building primes even with an incomplete heuristic is more fruitful than a “dogmatic” behavior regarding primes.

## 4 Experimental Results

Being consistent with our previous naming scheme, we name the new bots Plakoto-5 and Fevga-6. We compare them by taking the best set of trained weights and make them playing a tournament against a benchmark opponent without look-ahead (1-ply). For Plakoto and Fevga this benchmark is our best previous bot, namely Plakoto-4, and Fevga-4 respectively. For the Portes/Backgammon we chose the pubeval benchmark because we can indirectly compare the performance with others backgammon bots that published results against it. We also report on the performance when applying a simple look-ahead procedure using the expectimax algorithm [5] at 2-ply depth. The bot is awarded a +1 point for a single win, +2 points for a double win, -1 for a single loss, -2 for a double loss. The result of the tested games sum up to the form of estimated *points per game* (ppg) and is calculated as the mean of the points won and lost. The number of games played are 100000 for 1-ply and 10000 for 2-ply. In order to speed up the testing time of 2-ply, the expansion of depth-2 was performed only for the best 15 candidate moves (forward pruning). Table 4 presents the results.

**Table 4.**Performance of the new bots against benchmark opponents

| Bot | Opponent | ppg |
|---|---|---|
| Portes-1(1-ply) | Pubeval (1-ply) | 0.603 |
| Plakoto-5(1-ply) | Plakoto-4(1-ply) | 0.356 |
| Plakoto-5(2-ply) | Plakoto-4(1-ply) | 0.422 |
| Fevga-6(1-ply) | Fevga-4(1-ply) | 0.215 |
| Fevga-6(2-ply) | Fevga-4(1-ply) | 0.323 |

The performance of the Portes/Backgammon bot is comparable to most top playing bots. TD-Gammon [13] reported a 0.596 performance against pubeval [14] while another backgammon program, GNUBG[1], frequent participant to backgammon Computer Olympiads, recently reported in its mailing list similar performance (0.6046 ppg) while using a more complex training scheme and three different NNs for three different stages of the game [3].

Since the training procedure and the NN architecture is the same for the old and new bots for the Fevga and Plakoto variants, it is safe to assume that the gain was due to the addition/alteration of the expert features. We believe that the common features of Portes that were added to Plakoto and Fevga played a minor role to the improved performance. More important for Fevga, was the alteration of the prime features, and for Plakoto, the addition ChFrontOfPin_1 andChFrontOfPin_2.

## 5 Conclusion and future work

We have presented the complete algorithm of our training scheme for backgammon variants that are included in Palamedes. The computer backgammon winner of the 2011 backgammon Computer Olympiad was also presented in full for the first time.

[1]http://www.gnubg.org

Finally, we have managed to increase the performance of the Plakoto and Fevga variants by adding new expert features based on manual examination and user feedback.

We will continue the search for new features that could improve the playing strength of *Palamedes*. The heuristic for calculating the probability of making a prime formation on the next roll can be improved by including cases with two or more missing checkers, and by making it faster to compute.

Our experiments with different values for the learning rate $\alpha$ and the $\lambda$ parameter show that the best choice for either of them is domain specific. Using our setup, it is possible to start the training with high values and gradually decrease them. As we did not exhaust all possible combinations, it may possible that an even more aggressive approach could yield faster learning. An algorithm that automatically decreases these parameters during training would be interesting to investigate as it would free the human designer of the otherwise cumbersome trial and error approach.

A difficult part of the work so far is the manual examination of the playing style of the trained bots by human experts. This is necessary because NNs cannot easily describe the concepts learned by examining the weights alone. We plan to improve our understanding of the playing style of our NNs by visualizing the weights and by extracting rules [1].

We also plan to increase the number of backgammon variants that can be handled by *Palamedes*. Interesting candidates towards this direction are the acey-deucey, gioul and gul-bara variants. Finally we plan to improve the look-ahead procedure by searching in greater depths and by utilizing cutoff algorithms as in [4].

## References


1. Andrews R., Diederich J., Tickle A.: Survey and critique of techniques for extractingrules from trained artificial neural networks, Knowledge-Based Systems 8(6), 373--389 (1995)
2. BackGammon Variants, http://www.bkgm.com/variants
3. GnuBg Mailing list post, http://lists.gnu.org/archive/html/bug-gnubg/2012-01/msg00034.html
4. Hauk, T., Buro, M., Schaeffer , J.: *-minimax performance in backgammon. In: van den Herik, H., Bjornsson, Y., Netanyahu, N. (Eds.) Computers and Games 2006. LNCS, vol 3846, pp. 51--66 (2006)
5. Michie, D.: Game-playing and game-learning automata, In: L. Fox (Eds.) Advances in Programming and Non-Numerical Computation, pp. 183--200, (1966)
6. *Palamedes*, http://csse.uom.gr/~nikpapa/software.html
7. Papahristou, N., Refanidis, I.: Training Neural Networks to Play Backgammon Variants Using Reinforcement Learning, In: Cecilia Di Chio et al. (Eds.) EvoApplications 2011. LNCS, vol 6624, pp. 113--122, (2011)
8. Papahristou, N., Refanidis, I.: Improving Temporal Difference Learning Performance in Backgammon Variants, In: Advances in Computer Games (ACG-13). LNCS, vol 7168, (2012)
9. Pubeval source code backgammon benchmark player, http://www.bkgm.com/rgb/rgb.cgi?view+610
10. Sutton, R.S.: Learning to predict by the methods of temporal differences. Machine

Learning 3(1), 9--44 (1988)
11. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Indroduction. MIT Press (1998)
12. Tesauro, G.: Practical issues in temporal differnce learning. Machine Learning 4, 257--277, (1992)
13. Tesauro, G.: Programming backgammon using self-teching neural nets. Artificial Intelligence 134, 181--199, (2002)
14. Tesauro, G.: Td-gammon, http://www.scholarpedia.org/article/Td-gammon
15. Tesauro, G.: Temporal Difference Learning and TD-Gammon. Communications of the ACM 38(3), 58--68 (1995)
16. Wiering, M.A.: Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning. Journal of Intelligent Learning Systems and Applications 2, 57-68, (2010)
17. Wilson D.R., Martinez T.R. The general inefficiency of batch training for gradient descent learning. Neural Networks 16(10), 1429--1451 (2003)