International Graduate Program Medical Neurosciences

Lab Report

A Reinforcement Learning Framework utilizing Temporal-Difference Learning for the Game of Backgammon.

By: Stephan Mauermann

Matriculation Number: 220686

Date of submission: 30.09.2015
Direct supervisor: Raphael Holca-Lamarre (Ph.D. student)

Responsible Research Group Leader:
Prof. Dr. Klaus Obermayer, TU Berlin,
Institute of Software Engineering and
Theoretical Computer Science,
Neural Information Processing Group,
Marchstraße 23, 10587 Berlin,
oby@ni.tu-berlin.de

Suggested Second Reviewer:
Prof. Dr. Klaus-Robert Müller, TU Berlin,
Institute of Software Engineering and
Theoretical Computer Science,
Machine Learning Group,
Marchstraße 23, 10587 Berlin,
klaus-robert.mueller@tu-berlin.de

Introduction

1 Aim of the Project

The overall goal of this lab rotation was to provide a Reinforcement Learning (RL) system that could be used as an evaluation tool for benchmarking against other newly emerging RL systems. We have chosen to implement a RL model that is inspired by the impressive RL application of Gerald Tesauro's TD-Gammon, which will be described later in chapter 5.

One of the RL paradigms to be tested against the framework presented here will be a model developed by Prof. Jörg Lücke in cooperation with the Obermayer lab (Keck et al. 2012). This RL methodology promises enhanced learning performance over conventional methods. Improved learning capability was realized by adding 'brain-inspired' features like feed-forward inhibition or synaptic plasticity mechanisms (e.g., synaptic scaling) to the learning algorithm. Both of these added features were observed to be employed by hippocampus and certain cortical areas for information processing (Pouille et al. 2009).

2 Reinforcement Learning

From a broad perspective, RL occupies a subfield within Machine Learning (ML). While ML itself is a multidisciplinary field that originates from the intersection of computer science and statistics, and essentially addresses following issue: '*How can one create computer systems that automatically improve with experience (i.e. learn from experience), and what are the fundamentals of all learning processes?'*. So naturally, as learning is involved, Neuroscience plays a vital role for tackling the questions posed by ML as well. However, since our understanding of learning in animals is far from complete yet, there is still a long way to go to achieve a level of machine intelligence that could compete against its biological counterpart. Nevertheless, ML has made significant progress and is now widely used, for example in speech recognition, computer vision (image analysis) or drug discovery.

The idea of RL is inspired by our own sense of learning, that is learning by interacting with the environment and consequently extracting rules of cause and effect. This becomes very intuitive when thinking of playing children. They will learn quickly that riding their bike with no hands can be very hurtful without a teacher telling them so explicitly. Thus RL is considered to be tightly related to how humans and other animals learn. Finally RL as a variety of ML, then aims at providing computational tools for learning as it occurs in many biological systems (Sutton & Barto 2015).

RL agents follow a goal-directed form of learning and are driven by receiving a maximum amount of reward signals over time by trying out actions and choosing those actions, that will return the most reward. Such an agent is not told what actions are the best – instead it explores possible actions and receives rewards for these actions. Importantly, this implies that no prior knowledge about the task to be learned is required and thus, RL can be applied to a large range of control problems. Mechanistically, the RL model is an agent which continuously interacts with an environment. Agent and environment interact in a sequence of time steps and at each step $t = 0, 1, 2, 3, ...$, the agent observes a state of the environment S_t , chooses an action A_t , transitions to the next state S_{t+1} and receives a scalar numerical reward R_{t+1} from the environment (see Figure 1). As stated above, the agent now tries to maximize the total discounted return according to $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. Discounting (via discount rate γ) hereby refers to the concept of decreasing later rewards far in the future as compared to immediate rewards at the current time step (Sutton & Barto 2015).

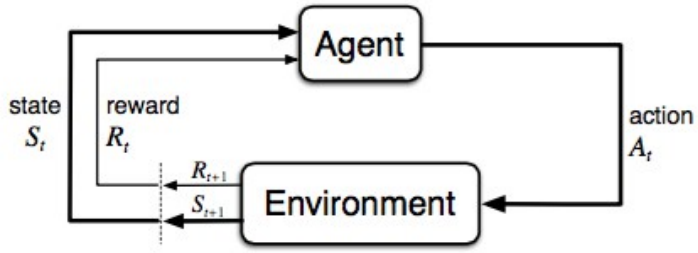

Figure 1: *Agent-environment interaction in the RL model. Adopted from Sutton & Barto (2015).*

In order to assign rewards to distinct states, RL methods need to approximate a function that maps reward values to distinct states – a so-called value function $V_t(S_t)$. These functions predict how much an agent profits from being in a given state. The profit in turn refers to an estimate of how much future reward (return) can be expected by the agent. Approximating value functions is a critical component of RL and can be done using different methods like artificial neural networks (ANNs), decision trees or multivariate regression models. ANNs are probably the most frequently used technique for function approximation today, and the RL framework described here will also facilitate neural networks. Thus, in the next chapter neural networks will be described in more detail.

3 Artificial Neural Networks

ANNs (short neural nets) are an abstract model dating back to the 1950's – arising from the desire to emulate structure and information processing abilities of biological neural networks by computational means. Put simply, the idea is to obtain a structure, much like a black-box, which receives some input and via internal computation generates an output. Importantly, the output can be a complex non-linear function of the input vector. Elementary unit of every ANN is the artificial neuron, a simple mathematical model of a biological neuron. Just like their biological brothers, artificial neurons receive inputs from many other units and process these signals internally. Finally, depending on the course of internal information processing, the neuron forward-propagates an output signal to subsequent neurons (Krenker et al. 2011). The net input of an artificial neuron is mathematically the weighted sum of all input values, according to $net = \sum_{i=0}^{n} w_i \, x_i$. Where each input x_i is linked to a particular weight w_i . The internal information processing squashes the net input through a transfer function like hyperbolic tangent or sigmoid function $\left(\sigma(x) = \frac{1}{1+e^{-x}}\right)$ to delimit the output Y_t of an artificial neuron to a desired interval. Large ensembles of interconnected layers each containing one or more artificial neurons (up to thousands) make up a neural net (see Figure 2). The information flows from left to right through the neural net – entering the ANN via the input layer, information is passed through the hidden layer (there can be multiple hidden layers) and finally reaches the output layer. By precisely tuning networks weights, as well as depending on layer topology and layer sizes, ANNs are able to approximate complex non-linear functions. Laying out ANN-topology is only one part of a sequence of necessary steps before ANNs can be used for solving a given problem. Like biological neural networks need to learn adequate responses to a given stimulus, ANNs must behave accordingly. Consequently, the remaining crucial step is to let the ANN learn a proper response – e.g., via RL. With neural nets, learning is interpreted as tuning the weights by back-propagating an output error when teaching the net on training data.

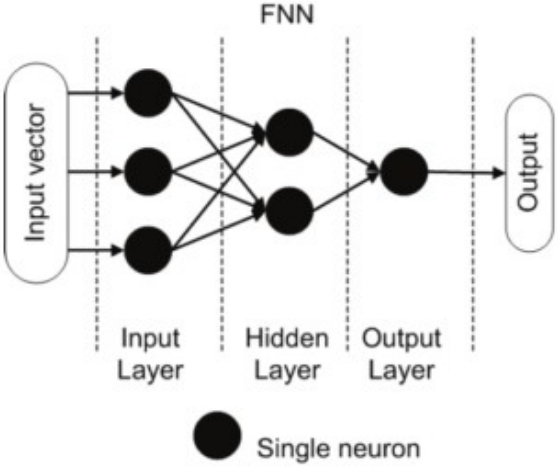

Figure 2: *A simple Feed-Forward neural network (FNN). Besides FNNs, there is a plethora of different network types, e.g, recurrent networks (RNN.) Adopted from Krenker et al. (2011).*

3 Temporal-Difference Learning

As described in the previous chapter, ANNs require a measure of how 'wrong' or 'right' their estimate is in order to train the network's weights. This is rather trivial in the case of supervised learning; here an input is presented to the network, think of a picture showing a cat, and one can immediately judge the neural net's output accuracy – good, if the network predicts that there is a cat; but if the ANN 'sees' a car, the weights have to be adjusted in order to minimize this error. Learning in ANNs however, becomes more challenging when an agent finds himself playing a game like chess or backgammon. It is not immediately apparent how much reward a distinct action (i.e., a move during the game) might yield since the total return an agent receives is directly linked to the final state of winning or losing the game – and this only known after the game (an episode) has ended. In other words, the agent has to deal with delayed rewards. This makes it hard to assign credit to individual actions taken – and the resulting states – which are only a tiny portion of a long sequence of dynamically changing states within the state-space. Now the state-space defines all possible states that can be observed in a particular environment, in the case of boardgames it would comprise the entirety of all legal board configurations.

To tackle this problem, Temporal Difference (TD) learning methods are a suitable way of dealing with the above described temporal credit assignment problem. TD learning procedures were initially proposed by Richard S. Sutton in 1988. The principal of TD learning is to learn from differences between temporally successive value predictions. Or with the words of Sutton, 'to learn a guess from a guess' (Tanner & Sutton 2005). More specifically, the current estimate for the current input is adjusted to better match the following prediction at the next time step. From a mathematical point of view, TD learning is a hybrid of so-called Monte Carlo (MC) methods and Dynamic Programming (DP). MC methods enable an agent to learn from experience without the need for an explicit model of the environment but they are only able to learn after the terminal state of an episode (or game) was reached and the final return is known. This reflects for example in the constant- α MC method $V(S_t) = V(S_t) + \alpha [G_t - V(S_t)]$, with learning rate α and the total return G_t (as described in chapter 2). Blending in DP's bootstrapping, that is updating value estimates based on subsequent estimates without having to wait until the terminal state of an episode has reached, one obtains the most simple form of TD learning methods termed *TD(0)* $V(S_t) = V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ – which looks ahead one time step to adjust the current estimation. By introducing so-called eligibility traces, a means of keeping track of recently visited states, one obtains the *TD(λ)* algorithm. Eligibility traces are a sophisticated

method to highlight only the actually visited states during an episode for contributing to learning – which enables an agent to credit or blame individual actions (states) from a long sequence of those events. For practical reasons and a better understanding, I will discuss here the so-called 'backward view' of *TD(λ)* because it is mechanistically simpler and actually implementable since the theoretical 'forward view' relies on undetermined future events. As stated above, *TD(λ)* utilizes eligibility traces that associate with all states from the state-space. These traces can be computed for non-visited states as follows $E_t(s) = \gamma \lambda E_{t-1}(s)$ and for visited states according to $E_t(s) = \gamma \lambda E_{t-1}(s) + 1$ with λ as the *trace-decay factor* and $E_{t=0}(s) = 0$. Accordingly, eligibility traces of non-visited states decay with $\gamma \lambda$, while visited states' traces decay likewise but at the same time increment by one. By adjusting the *trace-decay parameter* λ , it is now possible to steer how far the algorithm 'looks' back in time. If we set $\lambda = 0$, we obtain the simple *TD(0)* rule, which looks back only one time step. And in the case of $\lambda = 1$, we would end up with the MC method and all past visited states are credited. But how are eligibility traces incorporated in order to update the value estimates? By computing the TD error with $\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)$ and finally, by updating the value function according to following method termed the *true-online TD(λ)* algorithm (Van Seijen & Sutton 2014): $V_{t+1}(s) = V_t(s) + \alpha \left[\delta_t + V_t(S_t) - V_{t-1}(S_t)\right] E_t(s)$ for all states in the state-space. To wrap this concept up I would like to cite from the great RL textbook written by Sutton & Barto 2015: '*The backward view of TD(λ) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to the state's eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states. Where the TD error and traces come together, we get the update given by [the true-online TD(λ) method].*' Of note, throughout this chapter I described the update of state-/action-functions, however these rules can be also adopted to the usage of non-linear function approximators like ANNs, in which case one would had to update the individual weights by back-propagating TD errors through the network. A brief discussion of this issue can be found in the Implementation Details.

3 TD-Gammon

A very impressive approach taking advantage of the previously described RL methods is Gerald Tesauro's take on the backgammon game – TD-Gammon (Tesauro 1992; Tesauro 1995). Without any prior expert knowledge on the game except the basic ruleset and winning conditions, TD-Gammon trained itself to play backgammon so well, that it performs now on par with the best human players and even influenced traditional human backgammon strategies. The TD-Gammon program used a combination of the *TD(λ)* method and non-linear function approximation via a multilayer ANN, which was trained by TD error back-propagation. The neural network then learned to be an evaluation function for game situations by playing against itself and learning from the reward signal at the end of each game. Board games like Go, chess or backgammon are an exceptional testing facility for studying and developing various kinds of machine learning and artificial intelligence applications. The challenge of mastering such board games is their inherent complexity and the need to be able to exploit the game mechanism to derive promising winning strategies. On the other hand, these game environments are governed usually by simple rules and can thus be easily simulated for training purposes.

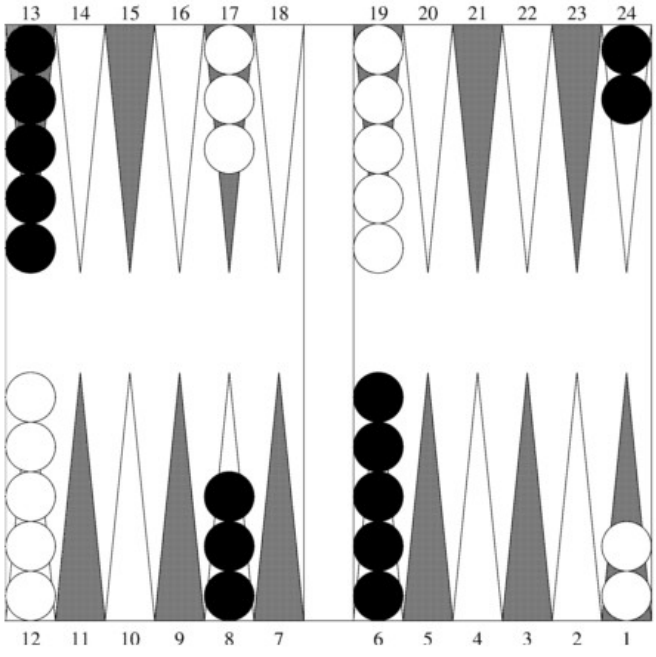

Figure 3: *The standard opening board configuration in backgammon. Black checkers move counter-clockwise around the board, while white checkers move clockwise. Adopted from (Tesauro 2002).*

Backgammon is an ancient game, possibly as old as 5000 years, which is played by two players on a board that effectively resembles a one-dimensional track. The standard opening board configuration can be seen in Figure 3. Both players take turns rolling two dice and moving their checkers in opposite directions across the board. The numbers on the dice indicate how many points a player is allowed to move his checkers, when a player rolls doubles he is allowed to use the rolled number four times. Whichever player gets to remove all of his checkers from the board first wins the game. In order to remove checkers from the board (to 'bear off') the player has to have all his checkers in his homeboard at the opposite end of the board (6...1 for black and 19...24 for white) before he can start to bear off. In other words, backgammon can be understood as a race between the two players to bear off their checkers as quickly as possible. However, the game receives

more complexity by additional game mechanisms. It is possible to land on a point that is occupied by only a single opponent checker (to 'hit'), the opponent checker now has to be moved to the 'bar' in the middle of the board. From there it must re-enter the board from the start before other checkers are allowed to move. Similarly it is possible to block a player by keeping two or more checkers on a point, the opponent is not allowed to move to this point. This can be used to prevent the opponent from re-entering checkers from the bar or making it harder to reach the homeboard. More complexity is introduced by a 'doubling cube' by which a player can offer to double the stakes of the game (backgammon often involves significant amounts of money).

These features lead to a number of sophisticated and complex strategies at the level of experts. All in all, backgammon is estimated to have over 10^{20} different possible states. And the number of possible moves from one point is typically around twenty, for one of the twenty-one possible dice combinations. That is far more complex than other games like chess and cannot be modeled by the otherwise so successful heuristic search methods for games. On the other hand, backgammon seems like a good match for TD learning. There is a complete description of the game state available at all times, the game is highly stochastic depending in the dice and the game is a long sequence of actions and resulting game states, which comes to an end when the terminal state is reached and one player has won and the other has lost. This in turn can be used as a correct prediction for the final reward. TD-Gammon used a standard feedforward network, with an input layer, one hidden layer with 40 units and an output layer. Two output units give an estimate of the value of a state for each player while the two remaining units are reserved for predicting special winning conditions (either a normal win or 'gammon'). The value of a state here can be interpreted as the probability of a player winning the game starting from this state. Now the network was trained by presenting a sequence of board positions starting with the opening board and ending with the final board, with having one player having removed all checkers. The board positions were encoded in a simple binary fashion and fed to the network. This board encoding only contained the raw board information (checkers on each point, the bar, as well as removed checkers) and was presented to the network as a vector x_t with 198 elements at each time step. For each input x_t the network now estimated an output vector Y_t . And at each time step, a version of the *TD(λ)* algorithm for neural networks is applied to change the network's weights in order to minimize the TD error as follows $w_{t+1} - w_t = \alpha \left(Y_{t+1} - Y_t\right) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$, where α is the learning rate, w is the vector of network weights, λ the trace-decay

parameter and $\nabla_w Y_k$ the gradient of network output with respect to its weights. At the end of each game the final reward z is given and then substitutes Y_{t+1} in the equation for the weight update. The weight update will be elucidated more clearly in the next chapter. Now during training, the neural network evaluates and selects moves for both players. At each time step during the game, the neural network scores every possible state generated by a legal move and selects the move with the maximum expected value for the current player.

Implementation Details

As laid out in the previous chapter board games are an excellent opportunity to test and study RL algorithms. Especially, in the case of backgammon with its inherent complexity, the resulting huge state-space and also a large stochastic component due to the randomness of dice rolls. It has been impressively demonstrated by Gerald Tesauro that TD learning methods can be applied to master a complex game like backgammon and to reach the level of the best human players by simply training TD-Gammon to extract important concepts of the game by self-play. Furthermore, backgammon is a major game throughout in the world and thus makes for a great benchmarking tool for newly emerging RL methods.

I chose to implement TD-Gammon using the Python language (version 2.7.10), the source code can be found on my GitHub repository (https://github.com/neurotronix/py_bg). Of note, my version of TD-Gammon only relies on one external library, Numpy, which can be installed easily into an existing Python directory via the package managers 'pip' or 'easy_install'. Python is widely spread in academic and industrial artificial intelligence research because it is a highly dynamic high-level language offering a large toolbox of libraries for very special needs – for example Numpy, which is a performant engine for solving complex mathematical problems and efficiently handling large arrays. Python supports many different programming paradigms like object-oriented programming (OOP) or functional programming. Furthermore, code written in Python exhibits very good readability and thus helps other users than the developer to understand the code or modify the code to fit their needs. With its dense syntax it is generally possible to lay out concepts with less code needed as compared to e.g., Java. This fact also makes it easier to actually learn the language. There is however a downside to the use of Python, the developers of Python clearly emphasize on code readability rather than to focus on actual computing performance. This means that while Python code will always be more conceivable than code written in languages like Java or C++, it will fail to beat these languages in terms of speed and code execution performance.

I implemented a straight-forward approach of a sigmoid feedforward neural network with an input layer (198 units), one hidden layer (40 units) and a simplified output layer with only four units. The input layer receives a raw board encoding as mentioned in the previous chapter and worked as follows: For each point on the board (there 24 points) and for each player, four units represent the amount of checkers. If there are no checkers at a point, all four units become zero. If there's one checker, the first unit takes the value one. This applies to up to three checkers on a point, in which case the first three units get the value one. If there were more than three checkers on a point the fourth unit assumes a value according to $(n-3)/2$ with n denoting the number of checkers. Now this gives a total of 192 input units. Additional units indicate the number of checkers on the bar ($n/2$) per player, and two more units for checkers that were removed from the board ($n/15$). And finally, two units indicated which players turn it is in a binary manner ('0, 1' for white's and '1, 0' for black's turn). This gives a total of 198 units. Every input value is roughly scaled to be between zero and one. These values were now fed to the network (vector x_t) and the network computed an output in the form of a two-dimensional vector $(Y_1(x_t, w_t), Y_2(x_t, w_t))$, where Y_1 indicates the probability of the white player winning the game starting with the current board input and Y_2 indicates the same odds just for the black player. The TD error in this application was simply the difference between two successive network outputs, $\delta_t = Y_k(x_t, w_t) - Y_k(x_{t-1}, w_t)$ for each of the k outputs. This error was then back-propagated to adjust the network weights in order to anneal the two successive outputs by applying following rule $w_t = w_{t-1} + \alpha [R_t + \gamma \delta_t] e_t$ with $\gamma = 1$ and $R_{t+1} = 0$ except for winning ($R_{t+1} = 1$). The eligibility traces needed for the temporal credit assignment of visited states were computed as follows: $e_t = \gamma \lambda e_{t-1} + \nabla Y_k(x_t, w_t)$ with $e_{t=0} = 0$, $\lambda = 0.7$ and $\nabla Y_k(x_t, w_t)$ according to the standard error back-propagation gradient descent routine for neural networks. All eligibility traces are reset to zero after a game has ended. As described in the previous chapter, at each player's turn the network is presented all legal board positions, evaluates these and returns the one board with the highest probability of winning the game.

The source code of the neural network is in the file *neural_net.py* and contains the classes *InputLayer*, *HiddenLayer*, *OutputLayer* and the class *NeuralNetwork* that assembles and connects the network layers. For constructing a new neural net, call the 'constructor' as specified: *NeuralNetwork*(input_size, hidden_size, output_size, restore_from_file). The network offers to save the current network state to a specified file

in the same folder as the script by using *save_network*(filename). By calling *restore_network*(filename) the net can be retrieved from the provided file. Calling *get_network_output*(input), the network performs a feedforward of the input vector and returns the corresponding output. The function *back_prop*(current_output, next_output) computes the network output gradient, the TD error, eligibility traces and finally back-propagates through the network and updates the weights accordingly.

The file *board.py* contains the classes *Dice* and *Board. Dice* simulates the behavior of rolling two six-sided dice by calling *roll*(), and the function *is_doubles()* checks whether doubles were rolled. The *Board* class represents the game board of backgammon and keeps track of checker numbers and checker colors, as well as how many checkers are on the bar or have been already removed. Furthermore it implements the movement of checkers, can determine whether a game is over (*is_gameover*()) and provides the possibility to print a simple text-style representation of the current board status the command line.

Backgammon's ruleset is implemented in the file *move.py*, it contains the classes *BarMove*, *NormalMove* and *BearOffMove*. These three classes hold the rules for the specific move and raise a custom backgammon-exception if the move to be made violates the rules. Furthermore, there is the class *BoardFactory*, which takes into account all possible moves, generates the resulting Board instances, checks for possible duplicates and returns a list of all possible boards that can be achieved given the provided dice roll, player and the board before the move by calling *generate_all_boards*(player, dice, board).

The file *player.py* contains the classes *Player* and *RandomPlayer*, latter inherits some functionality from *Player*. The *Player* class should be used when one wants to use the value estimation via the neural network. It is possible to switch between training mode and pure prediction mode (no back-propagation of errors and no weight updates). The *Player* class possesses an interface *choose_move()* to the neural network and feeds it with all possible boards generated from *BoardFactory*. The network will return the board with the best odds of winning. And if training mode is switched on, it will back-propagate TD errors and update weights as well as when the game is over it will award the final reward depending on the outcome of the game. The prediction efficiency can be easily tested by letting play the

network-backed *Player* against the *RandomPlayer*, which randomly selects moves without any scoring of future winning probability.

The class *Backgammon*, which can be found in the file *backgammon.py* finally wraps up the whole backgammon functionality. A backgammon instance is initialized and constructs two players, the starting board, the neural network and a pair of dice. By calling *run()* the instance will run a game from start until one player has won the game. The function *reset()* resets a backgammon instance to the initial status (without affecting the neural network of course). And last but not least, there are some custom made backgammon play-related exceptions that can be found in the file *bgexceptions.py*.

Results and Discussion

Since I started to learn programming only recently, implementing a simplified version of Gerald Tesauro's highly complex TD-Gammon was an incredibly challenging task for me. Given the targeted time frame of around six weeks for the completion of the project, it was not possible for me to extensively train the network. In a paper from 2002, Tesauro mentions a number of about 300.000 games of self-play it takes to saturate the networks learning performance. This is an actual drawback because training of my TD-Gammon version for 5.000 games took on average a solid 10 to 12 hours, which made it also very time-consuming to test the code for functionality after modifying the neural net. However, I was able to obtain some preliminary results for a network that trained for 5.000 games. This network performed against a random player and was able to beat the player in about 70% of games, but this needs to be repeated more frequently and verified statistically. Nevertheless, it is a clear-cut signal that there is a learning process happening and the network shows an improved 'understanding' of board positions as compared to the random player. Since the training is hugely time-consuming, I would suggest after proper validation of the learning algorithm, to port this version of TD-Gammon using a more efficient language like C++, which should aid in significantly reducing the sheer amount of time needed to train the network. For further validation of the program, it will be necessary to let it play against several established backgammon artificial intelligence engines like GNU Backgammon (http://www.gnubg.org).

References

Keck, C., Savin, C. & Lücke, J., 2012. *Feedforward inhibition and synaptic scaling - Two sides of the same coin? PLoS Computational Biology*, 8(3).

Krenker, A., Bester, J. & Kos, A., 2011. *Introduction to artificial neural networks. Artificial Neural Networks - Methodological Advances and Biomedical Applications*, 19(12), pp.1046–1054.

Pouille, F. et al., 2009. *Input normalization by global feedforward inhibition expands cortical dynamic range. Nature neuroscience*, 12(12), pp.1577–1585.

Van Seijen, H. & Sutton, R., 2014. *True Online TD (lambda)*. *Proceedings of the 31st International Conference on Machine Learning (ICML)*, 32, pp.692–700.

Sutton, R.S., 1988. *Learning to Predict by the Methods of Temporal Differences*. *Machine Learning*, vol, pp.3pp9–44.

Sutton, R.S. & Barto, A.G., 2015. *Reinforcement Learning: An Introduction*, MIT Press.

Tanner, B. & Sutton, R.S., 2005. *TD(λ) networks: Temporal-difference networks with eligibility traces*. *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pp.889–896

Tesauro, G., 1992. *Practical Issues in Temporal Difference Learning*. *Machine Learning*, 8, pp.257–277. Available at: http://incompleteideas.net/sutton/tesauro-92.pdf.

Tesauro, G., 2002. *Programming backgammon using self-teaching neural nets*. *Artificial Intelligence*, 134(1-2), pp.181–199.

Tesauro, G., 1995. *Temporal Difference Learning and TD-Gammon*. *Communications of the ACM*, 38(3), pp.58–68.