Neural Approximation of Monte Carlo Policy Evaluation Deployed in Connect Four

Stefan Faußer and Friedhelm Schwenker

Institute of Neural Information Processing, University of Ulm, 89069 Ulm, Germany
{stefan.fausser,friedhelm.Schwenker}@uni-ulm.de

Abstract. To win a board-game or more generally to gain something specific in a given Markov-environment, it is most important to have a policy in choosing and taking actions that leads to one of several qualitative good states. In this paper we describe a novel method to learn a game-winning strategy. The method predicts statistical probabilities to win in given game states using a state-value function that is approximated by a Multi-layer perceptron. Those predictions will improve according to rewards given in terminal states. We have deployed that method in the game Connect Four and have compared its game-performance with Velena [5].

1 Introduction

In the last 30 years, artificial intelligence methods like *Minimax* [8] have been used to build intelligent agents that have the task to choose a qualitative good move out of a set of all possible moves so they might win the game against another agent or even a human player. The basic approach in these methods are roughly the same: A game tree with a given depth will be calculated while exploring possible states and the move yielding to a state with minimum loss will be choosen using a heuristic evaluation function. While it is theoretical better to choose a large game tree depth, it is pratical impossible for games having an extensive set of possible states because of computing reasons. Furthermore such agents have not the ability to improve their computed strategy, although some agents might change it.

In contrast to the artificial intelligence methods above, the modern field of *Reinforcement learning* as defined in [1] were available since the late 1980s. Using those methods, the intelligent agent is learning by trial-and-error to estimate state values instead of exploring a game tree and taking the move with minimal maximum possible loss. In the specific case of Monte Carlo methods, this is achieved by first assigning each state an arbitrary initialized floating-point value that are then updated while playing in dependence of *Return values*, typically at intervall $[-1, +1]$, earned in terminal states. More generally, the intelligent agent is learning by experience to estimate state values and improves those estimations with each new generated episode. State values of all visited states in one episode will be updated by averaging their collected Returns. The simplest policy to take

L. Prevost, S. Marinai, and F. Schwenker (Eds.): ANNPR 2008, LNAI 5064, pp. 90–100, 2008.
© Springer-Verlag Berlin Heidelberg 2008

the best move is to choose the state out of all current possible states with the highest state-value.

While Monte Carlo methods require much computing time to estimate state values, because they theoretical need an infinite set of generated episodes for high quality estimations, it only takes linear time to use the derived policy to choose the best move. Overall Monte Carlo methods exhaust much less computing time than Minimax. Unfortunately in the classic way of assigning each state a state value, which is saving those values in tables, Monte Carlo methods cannot be used for environments with an extensive set of states because of the large requirement of computing space. This statement is also true for other reinforcement learning methods like *dynamic programming* and *temporal-tifference learning*.

Tesauro however has desribed a method of combining a Multi-Layer Perceptron with the Temporal-Difference learning to save a lot of computing space which he has applied in TD-Gammon [2] in 1992-1995. The basic idea was to approximate the state-value function $V(s)$ using a neural network trained by backpropagating TD errors. Although TD-Gammon performed well against human opponents as stated in [2], details like the used learning rate η and values of the initialized weights w in the Backpropagation algorithm are not given.

Encouraged by such scientific progress in the field of game playing, we have found a method to combine Monte Carlo Policy Evaluation with a Multi-Layer Perceptron that will be described in Section 3. We have applied this method to the well known game *Connect Four*, which has been choosen because of its simple rules and extensive set of possible states.

In Section 2, the basic rules of Connect Four as well as the complexity of the game will be presented. Afterwards the main ideas leading to our training algorithm, the learning process and the assembly of the resulting Multi-Layer perceptron will be discussed. Moreover, Section 4 provides the experimental results following by a conclusion.

2 Connect Four Rules and Game Complexity

Connect Four is a two-player game whereas one player has 21 red and the other player has 21 yellow discs. By turns, the players drop one disc into one of seven columns, which is sliding down and landing on the bottom or on another disc. The object of the game is to align four own discs horizontal, vertical or diagonal prior to the opponent. As no fortune is involved in winning the game, it is 100% a strategy game. Speaking of strategies, one of the better one is to arrange the own discs so that there are multiple opportunities to set the forth disc and win whilst preventing to loose.

Examining the game field, it is 42 fields, consisting of 7 columns and 6 rows, large and has 3 possible states per field. These states are red disc, yellow disc and empty. There are 3^{42} different possibilities to place up to 42 tokens on 42 fields. Plainly comparing this to english draughts, which is another popular board game, it has about $3^{42} * \frac{1}{5^{32} - 5^{(32-24)}} \approx 0.0047 \approx \frac{1}{200}$ of its complexity.

3 Derivation of the Training Algorithm

Suppose we want to estimate the state-value function $V^\pi(s)$ for a given policy π using a standard Monte Carlo policy evaluation method as listed in [1]. Iteratively two main steps are repeated, whereas in step one a game episode is created using π and in step two the Return values following each visited s are added to a set $Return(s)$. The updated state value-function is then $V^\pi(s) = average(Return(s))$. Now assume only one Return $\in \{0, 1\}$, depending on the outcome of the game, i.e. the terminal state, is given per episode for all visited states s. This allows us to remove the neccessity of the set $Return(s)$:

$$V^\pi(s) = (n(s)V^\pi(s) + Return)\frac{1}{n(s)+1}, \text{ for each } s \text{ visited in episode} \quad (1)$$

Therefore $V^\pi(s)$ is the mean of $n(s) + 1$ Return values that were received in observed terminal states, starting from s under policy π. Now consider policy π is a function that chooses one successor state s' out of all possible states $S_{successor}$ with the highest state-value:

$$\pi(s) = \underset{s'}{\operatorname{argmax}}(V^\pi(s') | s' \in S_{successor}) \quad (2)$$

This equation implies that an improvement of the estimation of $V^\pi(s)$ results in a more accurate choice of the successor state s'. Thus, learning to estimate $V^\pi(s)$ results in learning policy π. In general, generating a game episode under policy π is an interaction between our learning agent that does his move decision under policy π and a more or less intelligent opponent agent or human player. Note that in following notations, the state-value function is shortened to $V(s)$ because only one policy is targeted.

To evaluate and improve the policy, it is required to save the state-values on a computer storage. Having the sizes of the Connect Four game field, as introduced in Section 2, we calculate how much computing space the function itself consumes, if saved in tables: $3^{42} * 4$ bytes per state $\approx 40,76 * 10^{10}$ TByte. As this is much more space than a state-of-the-art computer, at present time, can deliver, it cannot be done in a straight-forward manner. Following Cybenko's Theorem [9] which denotes, that a Multi-Layer Perceptron (MLP) with at least one hidden layer and a sigmoid transfer function is capable of approximating any continuous function to an arbitrary degree of accuracy, we use a MLP to neural approximate the state-value function $V(s)$.

3.1 Neural Approximation of the State-Value Function $V(s)$

Assume we have generated an episode $\{s_1, s_2, ..., s_{TC}\}$ and have received one Return value $\in \{0, 1\}$ in terminal state s_{TC}. Let us now train a Multi-Layer Perceptron (MLP) as shown in Figure 1, so that the assigned output values $\{V(s_1), V(s_2), ..., V(s_{TC})\}$, which shall represent the statistical probabilities to win in given game states $\{s_1, s_2, ..., s_{TC}\}$, will be approximately updated like it

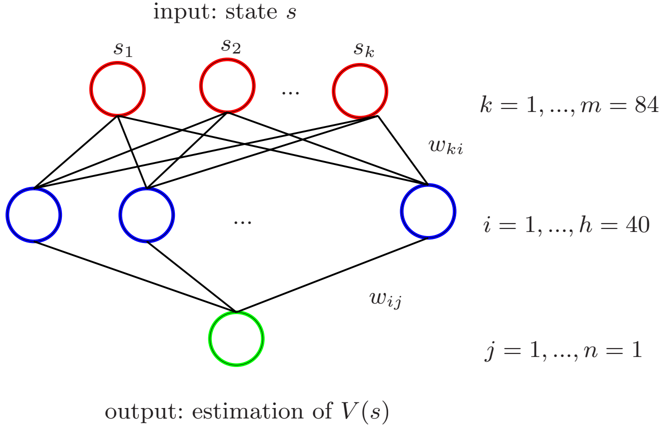

Fig. 1. Multi-Layer Perceptron schemata

is done in Monte Carlo Policy evaluation described above. In general, we would declare the training signals exactly like in equation (1), but we don't have space for the number of received Returns $n(s)$, similar like we don't have space for the state-values $V(s)$ as discussed in Section 3. Instead, our proposal is to define a training signal that itself approximates equation (1) due to the nature of the MLP to only slightly reducing the error function E if learning rate η is rather small. Our training signals T_s for each state s are defined as follows:

$$T_s = Return\gamma^{TC-t(s)} \tag{3}$$

In this equation, T_s equals the discounted Return value, whereas the discounting factor is build up by TC which represents the amount of all visited states in this episode and $t(s)$ which returns the index number of state s between 1 and TC. Apparently there is no discounting in the last state s_{TC} and increasing discounting of the Return value towards the first state s_1 in the episode. The discounting strength can be manipulated by discounting parameter $0 < \gamma \leq 1$, whereas higher values cause smaller discounting steps. The Return value is given in terminal state s_{TC}:

$$Return = \begin{cases} 1, & \text{if agent has won} \\ 0, & \text{if agent has lost} \end{cases} \tag{4}$$

As $V(s)$ for a specific s should converge T_s to a certain degree, both are used to express the error function E_s:

$$E_s = ||T_s - V(s)||^2 \tag{5}$$

Using error E_s, which is the mean squared error (MSE) between T_s and $V(s)$, we can define the error of the whole episode:

$$E = \sum_s E_s \tag{6}$$

Having target function E defined and properly introduced, it can be minimized by updating the weights in the MLP. This can be achieved by calculating a gradient vector in the error surface, build-up by given weights, with a starting position equal to the current weight values and following the gradient vector in opposite direction. Approach to derivate the weights w_{i1} in the output layer:

$$\Delta w_{i1} = \eta - \nabla E_s = -\eta \frac{\partial E_s(w_{i1})}{\partial w_{i1}} = -\eta \frac{\partial E_s(u^{(2)})}{\partial u^{(2)}} \frac{\partial u^{(2)}(w_{i1})}{\partial w_{i1}} \tag{7}$$

Approach to derivate the weights w_{ki} in the input layer:

$$\Delta w_{ki} = \eta - \nabla E_s = -\eta \frac{\partial E_s(w_{ki})}{\partial w_{ki}} = -\eta \frac{\partial E_s(u_i^{(1)})}{\partial u_i^{(1)}} \frac{\partial u_i^{(1)}(w_{ki})}{\partial w_{ki}} \tag{8}$$

Analyzing the target function (6), it has the following effects on our learning agent:

- Due to the applied gradient descent method defined above, the weights are not updated immediately to match $E = 0$ but rather will be updated to minimize E slightly, taking a maximum step η of the gradient descent vector. For small values of η, this has the impact, that $V(s)$ for all s visited in episode are increasing slightly if Return is 1 or are decreasing slightly if Return is 0. Another view is that a new state value $V(s)$ is calculated based on weighted older state values of the same state s. State values occuring more often have a stronger weight than state values occuring seldom. Summed up, the behavior is similar to averaging Returns like it is done in the standard Monte Carlo Policy Evaluation method
- Discounting factor $\gamma^{TC-t(s)}$ causes states, that are more closely to the terminal state, to get higher state values. This forces the intelligent agent to maximize the received Return values in the long run
- As the error of the whole episode is minimized, the performance is not affected by state ordering in the episode (Offline / Batch learning)

3.2 Implementation and Assembly Details of the MLP

Before actually feeding a game state s into the input layer of the MLP, it has to be encoded. As stated in Section 2, Connect Four has 42 fields with 3 states per field which means that we are in need of ≥ 2 neurons per field if one neuron equates one bit. To reach a balanced distribution, we have choosen bit sequence 01 for red disc, 10 for yellow disc and 00 for empty field. Counting from left to right, the first two input neurons define the upper left and the last two input neurons define the lower right portion of the game field as seen in Figure 2. The coding of the output value is simple, as it represents one state-value $V(s)$ at intervall $[0, 1]$, for which only one single neuron is needed. Considering the prior defined coding scheme, the MLP needs $m = 84$ input neurons in the input layer and $n = 1$ output neuron in the output layer. The accurate count of hidden neurons h in the hidden layer does not result immediate out of the count of neurons $m+n$

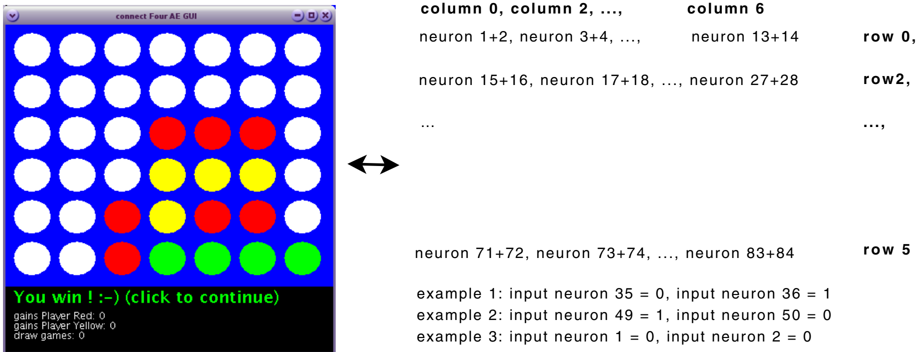

Fig. 2. Relation of the input neurons and the game field demonstrated by our Connect Four software

but depends on it and on the complexity of the problem. Typical for Multi-Layer Perceptrons, each neuron in the input layer is weighted connected to each neuron in the hidden layer and each neuron in the hidden layer is weighted connected to each neuron in the output layer. Positive weights are supporting and negative weights inhibitoring to the dendritic potential of the target neuron. All weights are initialized at intervall $[-a, +a]$, whereas the exact value of a as well as hidden neurons h is available in Section 4. For the transfer function in the hidden layer and the output layer, we have choosen a nonlinear and sigmoid logistic function which seems to be natural because we want to assign probabilities:

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{9}$$

$$f'(x) = f(x)(1 - f(x)) \tag{10}$$

As usual, the transfer function in the input layer is the identity-function.

3.3 Overview of the Training Algorithm

Exploitation and Exploration. Following the interaction cycle between the intelligent agent and an opponent as shown in Figure 3, the intelligent agent is spawning a pseudo-random value at intervall $[0, 1]$ after he received state s_t of the opponent and has builded possible successor states $s_{ta}, s_{tb}, ..., s_{tg}$ which result through an own move starting in s_t. If this value is $> \epsilon$ he is exploring, else he is exploiting the following game state s_{tx}. Exploiting denotes, that he calculates the state-values $V(s_{ta}), V(s_{tb}), ..., V(s_{tg})$ by feeding the game states one by one into the neural network to get the state values. Then he chooses the move resulting in state $s_{tx} \in \{s_{ta}, s_{tb}, ..., s_{tg}\}$ with the highest assigned state value $V(s_{tx})$. Exploration is carried out at rate $1 - \epsilon$ which means, that the intelligent agent randomly sets s_{tx} to one of successor states $s_{ta}, s_{tb}, ..., s_{tg}$ that he will visit but not include in the training set. This is not only important to speed up learning,

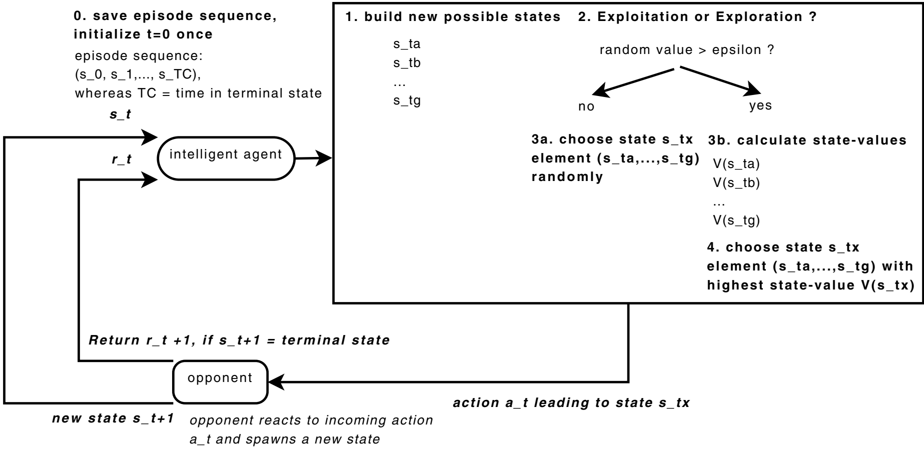

Fig. 3. Modell showing the interaction between the intelligent agent and an opponent (environment). Note that this cycle ends if the agent enters a terminal state or receives a terminal state of the opponent. In that case he would get a Return value and train the MLP with visited episode sequence and Return value to better estimate state-values in general.

but rather crucial to make good learning possible. The explanation is, that the once arbitrarily initialized weights in the MLP may have values so that certain state values $V(s)$ are fairly small. That in turn causes the intelligent agent to never visit those states s using Exploitation because of their low state values $V(s)$ and his policy to choose the highest state value. Exploration however enables to visit one of those states and to learn based on this state if the following state will be exploited.

4 Experiments and Results

4.1 Setting

To train the intelligent agent, we use an opponent that shares the same knowledge base, which is the weights of the MLP, to plan his moves. Additional, the opponent does random moves at rate $1-\epsilon$ to avoid generating same episodes and to reach a balanced spawn of Return values. Having trained the intelligent agent with a certain amount of episodes, we are using another different opponent that does 100% random moves to plainly measure the quality of the learned policy. The quantity of successes of the intelligent agent is given in a winning quota after $500,000$ test-episodes:

$$\text{winning-quota} = \frac{\text{number of games won intelligent agent}}{\text{number of games won opponent}} \tag{11}$$

Note that the upper defined winning-quota is disregarding the number of draw games, which were in all test cases pretty low. As the behaviour of the intelligent

Table 1. Algorithm 1: Offline (Batch) training of a MLP to estimate state values using Monte Carlo Policy Evaluation

- **Given**: Episode sequence $\{s_1, s_2, ..., s_{TC}\}$ consisting of TC game states s, Return value *Return* and MLP weights and parameters
- **for** $index = 1, ..., TC$
  - Choose pattern: $s = s_{index}$
  - Calculate $V(s)$:
    - **for** $i = 1, ..., h$
      - $u_i^{(1)} = \sum_{k=1}^{m} s_k w_{ki}^{(1)} - \theta_i^{(1)}$
      - $y_i^{(1)} = f(u_i^{(1)})$
    - **end for**
    - $u^{(2)} = \sum_{i=1}^{h} y_i^{(1)} w_{i1}^{(2)} - \theta^{(2)}$
    - $V(s) = f(u^{(2)})$
  - Calculate error on output neuron for pattern s:
    - $T_s = Return * \gamma^{TC-t(s)}$
    - $\delta_{index}^{(2)} = 2(T_s - V(s)) * f'(u^{(2)})$
  - Calculate error on hidden neurons for pattern s:
    - **for** $i = 1, ..., h$: $\delta_{index,i}^{(1)} = \delta_{index}^{(2)} * w_{i1}^{(2)} * f'(u_i^{(1)})$
- Update weights:
  - **for** $k = 1, ..., m$
    - **for** $i = 1, ..., h$
      - $w_{i1}^{(2)} = w_{i1}^{(2)} + \eta \sum_{index=1}^{TC} \delta_{index}^{(2)} * y_{s,i}^{(1)}$
      - $w_{ki}^{(1)} = w_{ki}^{(1)} + \eta \sum_{index=1}^{TC} \delta_{index,i}^{(1)} * s_{index,k}$
      - $\theta^{(2)} = \theta^{(2)} - \eta \sum_{index=1}^{TC} \delta_{index}^{(2)}$
      - $\theta_i^{(1)} = \theta_i^{(1)} - \eta \sum_{index=1}^{TC} \delta_{index,i}^{(1)}$
    - **end for**
  - **end for**

agent is sensitive to six parameters, i.e. the number of hidden neurons h, gradient step η, also referred as learning rate, the number of generated training-episodes, weights intervall a, discounting parameter γ and exploitation parameter ϵ, we have trained more than one agent to compare their performances and have assigned them a number. Prior each training, the weights of the MLPs have been initialized once at weight intervall a. The results are listed in Table 2, whereas parameters $a = 0.77$, $\gamma = 0.97$ and $\epsilon = 0.7$ are identical for each MLP.

Further on we have developed a software interface, which allows the intelligent agent to play against Velena [5]. Velena is a shannon C-type program that is based on the theory of Victor Allis master thesis [6]. It combines eight rules and a PN-search engine and claims to play Connect Four perfectly in difficulty 'C'. Further difficulty levels 'A' and 'B' are available, which limit Velena's ability to look ahead. In various test runs we have observed that Velena is superior in all difficulty levels to an opponent that does 100% random moves. Table 3 lists the results, whereas we have to note that Velena is not acting deterministic,

Table 2. Winning-quota measure of multiple intelligent agents, performing versus opponent that does 100% random moves

MLP No.	**winning-quota**	**h**	η	**episodes**
1	146.00	40	0.1	$2,000,000$
	246.63		0.05	$+4,000,000$
	317.66		0.025	$+8,000,000$
2	722.56	60	0.1 to 0.025	$14,000,000$
3	$1245,86$	100	0.1 to 0.025	$14,000,000$
4	1514.14	120	0.5 to 0.03125	$6,000,000$
5	**5049.42**	250	0.5 to 0.0625	$14,000,000$

Table 3. Performance measure of multiple intelligent agents versus Velena. *1st* notes that our intelligent agent had the first move, *ep diff* is the observed count of different episode sequences out of 500 possible.

MLP No.	**diffic.**	**ep diff 1st**	**ep diff 2nd**	**won 1st**	**won 2nd**	**draw 1st**	**draw 2nd**
1	A	92	161	18.2%	11.4%	6.8%	3%
	B	2	176	0%	16.8%	0%	1%
2	A	2	193	0%	25.4%	0%	10.8%
	B	2	194	0%	18%	0%	10.4%
3	A	44	265	1.6%	21.4%	3%	17.2%
	B	40	284	2%	25%	2.2%	18.4%
4	A	182	267	31%	40%	0.2%	14.6%
	B	176	258	36%	35%	0%	13.8%
5	A	57	323	89.6%	46.6%	0.004%	38.6%
	B	31	323	81%	47%	0%	36%
	C	11	35	0%	**0.002%**	0%	**0.004%**

i.e. is using a pseudo-random number generator in his policy. Different observed episode sequences are counted in column *ep diff*.

4.2 Discussing the Results

Analyzing the results in Table 2 it is apparent, that the success of gaining a good game-winning strategy is in dependence of the right parameter values. As expected, the winning rate improves with an increasing learning rate η as well as with an increasing number of training episodes. However, both values have to be limited, because if η or the number of training episodes get too large, then the success measured by the winning-rate is flattening. Further concentrating on the number of hidden neurons h, we have observed, that a MLP with a higher amount of h and more training-episodes performs a cut above which shows a well scaling behavior.

Recalling that our training signal T_s is just directing to the true value but is not equal to it, it is intuitive clear that the learning rate has to be decreased with an increasing amount of training episodes. Promising results have been reached

by starting with a higher gradient step $\eta = 0.1$ that is step-wise decreasing to about $\eta = 0.025$. We have set the weights intervalls $a = 0.77$ as sugggested in [3] where they have produced the best neural approximators with the highest generalization rate:

> An fixed weight intervall of 0.2, which corresponds to a weight range of $[-0.77, 0.77]$, gave the best mean performance for all the applications tested in this study.

Overall, the best result was obtained by using a step-wise decreasing learning rate $\eta = 0.5$ to $\eta = 0.0625$, hidden neurons $h = 250$, $a = 0.77$ and $14,000,000$ generated training episodes, which rewarded us with a winning-rate of about $5,000$.

Observing the results of our intelligent agents performing against Velena in Table 3, it is striking, that the intelligent agents with more hidden neurons h are better than those with less ones. Analyzing agents 1-3, they won average 20% of all games or ended about 10% draw, if they had the second move in difficulty 'A' and 'B'. Having the first move, it is unlikely for them to win a game. Observing this behavior, it is clear that they have not found their perfect opening position. Furthermore, intelligent agent 5 performed best: He lost only about 10% of all games in 'A' and 'B' while he even won once in difficulty 'C' and reached two draws.

5 Conclusion

We have described a method to approximate the Monte Carlo Policy Evaluation using a Multi-Layer Perceptron in this paper. In the experiments, we have statistically determined parameters for that algorithm to apply well in gaining a game-winning strategy in Connect Four. After the training had been carried out, we have used that strategy to perform against Velena. Although Velena claims to be unbeatable in difficulty 'C' and slightly weaker in difficulty 'A' and 'B', our intelligent agent 5 has experimental proven it's strength in all difficulties, where he has lost only about 10% of all games in 'A' and 'B' while he even *won once* in difficulty 'C' and reached two draws. Overall, we've achieved a good game-winning policy in Connect Four that can compete against experienced human players.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
2. Tesauro, G.: Temporal Difference Learning and TD-Gammon. Communications of the ACM 38(3) (1995)
3. Thimm, G., Fiesler, E.: High order and multilayer perceptron initialization. IEEE Transactions on Neural Networks 8(2), 249–259 (1997)

4. Thimm, G., Fiesler, E.: Optimal Setting of Weights, Learning Rate and Gain. IDIAP Research Report, Dalle Molle Institute for Perceptive Artificial Intelligence, Switzerland (April 2007)
5. Bertoletti, G.: Velena: A Shannon C-type program which plays connect four perfectly (1997), `http://www.ce.unipr.it/~gbe/velena.html`
6. Allis, V.: A Knowledge-based Approach of Connect-Four, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (1998)
7. Lenze, B.: Einführung in die Mathematik neuronaler Netze. Logos Verlag, Berlin (2003)
8. Russel, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall, Englewood Cliffs (2002)
9. Cybenko, G.V.: Approximation by Superpositions of a Sigmoidal function. Mathematics of Control, Signals and Systems 2, 303–314 (electronic version) (1989)