Self-Learning Artificial Player

FREDRIK KLEVMARKEN
and KONRAD RZEZNICZAK

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

Self-Learning Artificial Player

F R E D R I K K L E V M A R K E N
a n d K O N R A D R Z E Z N I C Z A K

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Johan Boye
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
klevmarken_fredrik_OCH_rzezniczak_konrad_K10056.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

Reinforcement learning has been an area of great interest the past decades. Self learning machines exist quite commonly in our society today, within industry as well as simple games. This document is an investigation into reinforcement learning. The aim is to implement a self learning game player using Q-learning and an artificial neural network. The learning agent will face a simplistic opponent with a static tactic in a game of advanced noughts and crosses. Using Q-learning and the artificial neural network the learning agent will slowly increase its proficiency in the game and win numerous matches. The document is ended with a discussion on the results of the matches and the effectiveness of the implementation as well as further reading.

Referat

Reinforcement learning är ett område som haft högt intresse de senaste decennierna. Självlärande maskiner är relativt vanliga i dagens samhälle. De finns inom industrin såväl som i simpla spel. Det här dokumentet är en undersökning av reinforcement learning. Syftet med undersökningen är att implementera en självlärande spelagent med Q-learning och ett artificiellt neuralt nätverk. Den självlärande spelagenten kommer att möta en simpel agent med en statisk taktik i en mer avancerad version av spelet luffarschack. Med användning av Q-learning och det artificiella neurala nätverkat kommer spelagenten långsamt öka sin färdighet i spelet och slutligen vinna flertalet matcher. Det här dokumentet avslutas med en diskussion gällande resultaten från matcherna, effektiviteten i implementeringen av den självlärande spelagenten samt en del om fortsatta studier.

Table of Contents

Division of Labor

This investigation is part of a course that requires us to share the division of labor with the reader. During this project both of us have worked on it and due to outside time constraints we have not always been able to work side by side. This is why we partially divided the work between us. In the initial phase of the project both of us gathered data and information regarding both the background of the area of knowledge and how Q-learning and neural networks actually work. During the planning phase we sat down several times together and discussed how to make a good implementation of the self-learning game player and we finally wrote the planning document side by side. During the main phase of the project Konrad became the *chief programmer* because of his extraordinary skills in C#, which the game was implemented with. Fredrik became *chief linguist* because of his experience in writing reports and technical English. Both of us have aided each other during the project by technical assistance as well as help with writing.

1 Introduction

It is of great interest to approximate complex systems and as efficiently as possible mimic intelligence in order to estimate solutions to real life problems. This report is an investigation into how this can be done using reinforcement learning and artificial neural networks. In short, the aim of this investigation is to construct a self learning game agent whom will be able to play noughts and crosses and learn how to win more efficiently over time. In order to investigate the learning capacity of the game agent an experiment will be set up where the agent will face a simplistic agent using a static tactic. The results of this experiment will be conclusive in determining the effectiveness of the learning agent.

This paper is structured as follows. The remainder of section 1 includes a more detailed description of the statement of the problem and the aim of the investigation. The statement of the problem is followed by the background history of reinforcement learning and the state of the art in the area of self learning artificial game agents. Section 2 describes the game noughts and crosses; how it is played and the layout of the game field. Sections 3 and 4 give a detailed account for q-learning and artificial neural networks. In section 5 we will describe how the implementation of q-learning with an artificial neural network was made. Section 6 deals with how the learning agent plays versus the simplistic agent. Section 6 is followed by a results section in which the data collected from the matches between the agents is accounted for. The results are then discussed and evaluated in section 8 which is followed by a concluding chapter.

1.1 Statement of the problem

The aim of this investigation is to implement a self-learning artificial agent for the board game noughts and crosses. The goal of the game will be to first place 5 Xs or 5 Os in a row anywhere on the game field which is composed of 20x20 panes. The artificial game player will be implemented using the reinforcement learning method Q-learning with an artificial neural network. We also aim to investigate how well the agent learns against an opponent with a static strategy. This will be tested by pitting the two against each other several thousand times. Depending on the outcome of the tests we will see how well the learning agent adapts to his opponent.

1.2 Background history

Today's self learning machines and software learn through a concept known as reinforcement learning. The modern theory of reinforcement learning is composed mainly of two different concepts. Firstly the concept of learning through trial and error and secondly the problem of optimal control and how it is solved using dynamic programming and value functions. A third concept developed later, is equally important as its two bigger brothers, is Temporal Difference learning and has also influenced the development of the modern reinforcement machine learning concept [1].

The concept of trial and error traces back to psychology and Edward Thorndike's concept of animal learning, through reinforcement and the Law of Effect, where an animal would associate a particular situation to the outcome of a particular response, given that the response gave a satisfactory or dissatisfactory result for the animal [1]. Trial and error incorporates two important aspects, namely *selection* and *associations*. Selection means that for example an animal or robot evaluates the available paths and selecting which one to take depending on its consequences. While the aspect of association deals with how the chosen path is associated with the particular situation it was selected in [1]. It can hence be referred to as the learning experience.

The problem of optimal control consists of finding a law for a given dynamic system such that one can minimize a criterion of the system over time. In other words an optimal control problem contains a cost function so that it models the state and the control variables, and an optimal control is a set of equations that model the optimal path of the control variables to minimize the cost function or law. For example consider a swimmer in a 50 meter pool. The question is how fast he should swim if he is to maximize total distance swum in a set time. The control law in this case refers to the swimmers chosen technique and amount of strength he uses in each stroke. The system consists of the swimmer and the pool. The optimality criterion is the maximization of distance. The constraints in this particular example can consist of the swimmers stamina, water resistance, streamline shape, time and so on.

Temporal difference learning is a form of prediction technique where an agent learns through sampling its system according some policy that defines how the learning agent will act at a given time [1]. Temporal difference learning is also closely related to dynamic programming techniques, since both techniques approximate their current state based on previous states. Temporal learning has also been incorporated into trial and error learning and goes under the name of actor-critic architecture [1]. Temporal difference learning and optimal control have been merged to form what is today known as Q-learning. Q-learning extended and incorporated parts of all of the three previously mentioned concepts and it is the main tool used in this artificial intelligence project of creating a self-learning game player [1].

1.3 State of the Art

It is very hard to tell exactly which implementation of reinforcement learning is at the highest level of development today. Applications of self-learning algorithms range from robotics, to industrial manufacturing, to combinatorial search problems such as computer game playing and therefore it is difficult to know which area is leading within reinforcement learning. Although when it comes to each area separately, the task becomes much easier mostly because it is very common to hold contests where different systems "compete" with each other in solving the same task. Chess playing is one of the more popular fields that use learning systems and is close to our area of interest. Chess playing has developed rapidly in the past fifty years and by comparing different contest lists it seems obvious that Deep Rybka 3 64-bit is as of 2010 the best chess engine [4]. It was developed by the software engineer and international chess master Vasik Rajlich and his team [5]. These types of engines are a perfect example of usage of learning algorithms. They need to use the newest solutions to stay ahead of the competition. Similar to the chess game is the noughts and crosses game but far simpler compared to the former. To win the game without prior knowledge of the states the agent needs to be implemented with a self-learning algorithm based on a concept such as Q-learning.

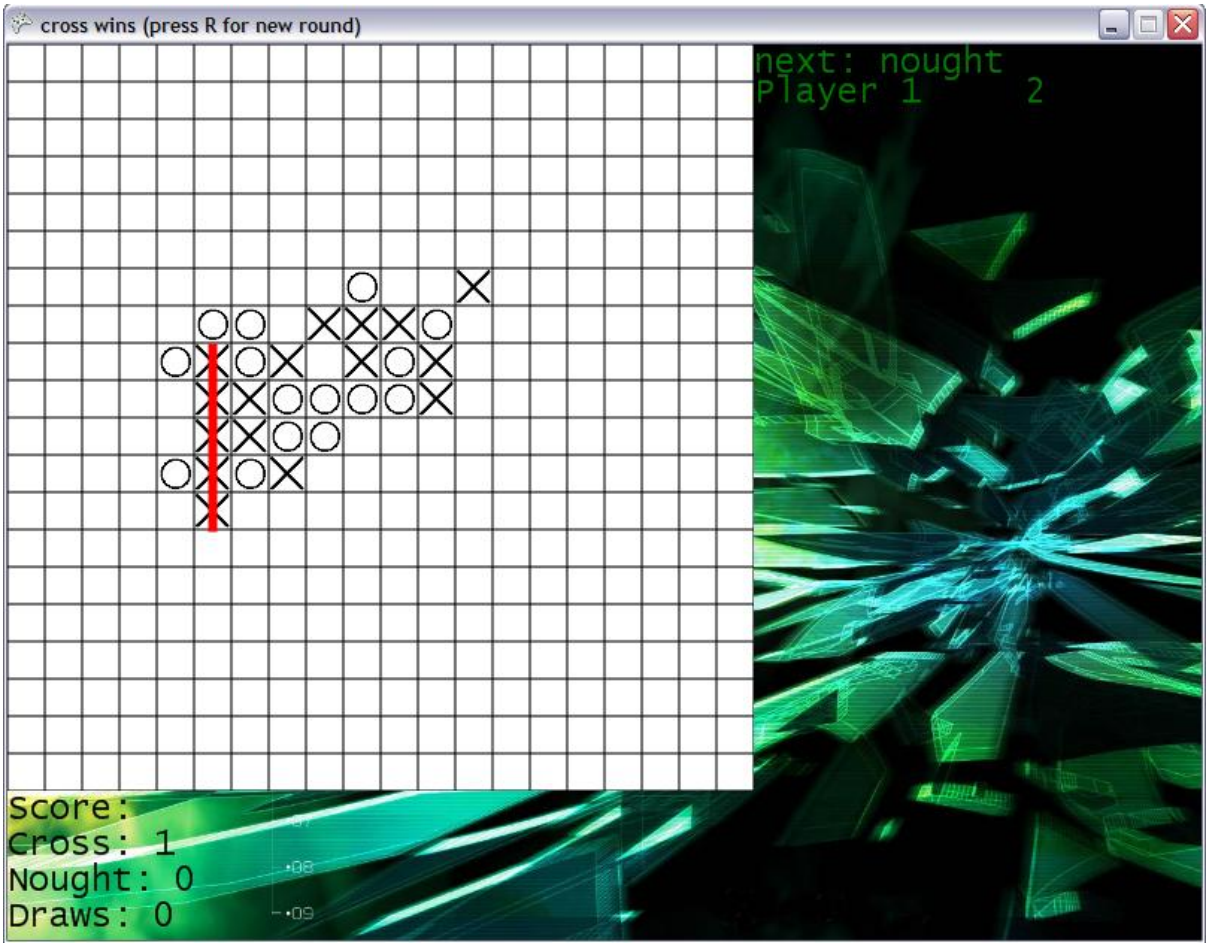
2 The noughts and crosses game

The noughts and crosses game dates back to the classic 3x3 Tic-Tac-Toe. The goal of Tic-Tac-Toe is to attain either 3 Xs or Os in a row on a quadratic game field composed of 3x3 squares. The first one to start is always X while O follows until either some one win or the game ends in a draw.

This project aims to develop a more complex version of the initial noughts and crosses game. The game field will be composed by 20x20 panes that can be occupied either by an X, O or be unoccupied. The player represented by X will always have the first turn during each game. This can be seen as advantageous, especially on smaller game fields. The goal of the game will be to place 5 pieces of your type in a row somewhere on the game field before your opponent does so. The players will be represented by different types of agents whom will play against each other.

In contrast with the simpler version the 20x20 noughts and crosses game will have 3^{400} possible board layouts while the 3x3 noughts and crosses game will only have 3^{9} possible layouts. The 20x20 game-type will also have 400! Different sequences for placing Xs and Os while the 3x3 type will only have 9! Different sequences. This in turn makes it impossible to construct the game using classical lookup-tables to represent the different states. Hence the game will be constructed using an artificial neural network in hand with q-learning algorithms to implement the reinforcement learning aspect of the game.

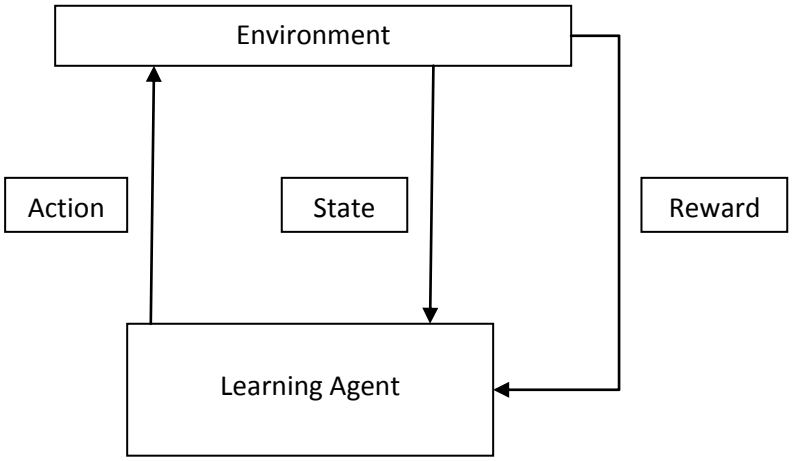
Figure 1: In-game picture of the game noughts and crosses



3 Q-learning

Q-learning as described in the *Background* section is a derivation of reinforcement learning. Where reinforcement learning in our case is a way for a learning-agent, most commonly a computer program, to continuously sample its environment and make decisions based on it. The learning-agent will process the input from the current state and make a decision on which action is the most appropriate to take in order to gain the highest reward. After the action is taken the environment will transform and the agent will receive new input. A reward is normally given after each taken action as part of the feedback from the environment. This results in a cyclic behavior which is represented in *Figure 2*.

Figur 2: Reinforcement Learning



A more technical way of describing Q-learning is to express it as a reinforcement learning algorithm that learns the values of a Q-function *Q(s,a)* to find a optimal policy π to act according to in the future. Where *s* is the current state and *a* is the taken action in that state. The values of the Q-function illustrate how optimal it is to perform a certain action in a specified state. The Q-function is defined as the reward received for an action taken in a state *r(s,a)* plus the discount factor γ of the reward acquired by pursuing an optimal policy. γ is defined as a parameter with a value in the range of [0, 1) where a value of 0 indicates that the agent will be opportunistic in his approach and consider only the current rewards while a γ value that approaches 1 will make it consider long-term rewards in order to maximize the total reward [2].

$$\mathrm{Q}(\mathrm{s},\mathrm{a}) = \underset{\mathrm{a} \in \mathrm{A}}{\mathrm{E}[\mathrm{r}(\mathrm{s},\mathrm{a}) + \gamma \mathrm{V}^{*}(\delta(\mathrm{s},\mathrm{a}))]}. \quad (1)$$

Where *Q(s,a)* is the expected reward E[…] which is estimated using the rewards function *r(s,a)*, the discount factor γ, and the state-value function $V^{*}(x)$ which returns an action value for the state based on the Q-value from the next state. Under the condition that the action *a* exists in all possible action *A*.

With the help of the Q-function an optimal policy is acquired through

$$\pi^*(s) = \max_{a \epsilon A} Q(s, a). \tag{2}$$

This is very interesting since it shows that a learning agent that knows the Q-function does not need to know the reward function nor the state-transition function $\delta(s, a)$ to determine an optimal policy π^*. It is also worth noting that the state-value function and *Q(s,a)* are related as can be seen in equation (3).

$$V^*(s) = \max_{a \epsilon A} Q(s, a) \tag{3}$$

This gives rise to the recursive property of the Q-function which will be used to define the Q-learning algorithm. This is easily perceived by substituting (3) into (1).

$$Q(s, a) = E[r(s, a) + \gamma \max_{a' \epsilon A} Q(\delta(s, a), a')] \tag{4}$$

$$\hat{Q}(s, a) \leftarrow (1 - \alpha)\hat{Q}(s, a) + \alpha(r(s, a) + \gamma \max_{a' \epsilon A} Q(s', a')) \tag{5}$$

Where α is the learning rate parameter and lies in the interval of [0, 1), the lower the parameter is the less will the agent learn. If a higher value is used, only the most recent data will be considered [2] and will override the past data

A learning agent that wants to learn the Q-function approximates it through the Q-learning algorithm described below.

Algorithm 1: The Q-learning algorithm [2].

For all states $s \epsilon S$ and all actions $a \epsilon A$ initialize $\hat{Q}(s, a)$ to an arbitrary value
Repeat (for each trial)
Initialize the current state *s*
Repeat (for each step of trial)
Observe the current state *s*
Select an action *a* using a policy π
Execute action *a*
Receive an immediate reward *r*
Observe the resulting new state *s'*
Update $\hat{Q}(s, a)$ according to Equation (5)
$s \leftarrow s'$
Until *s* is a terminal state

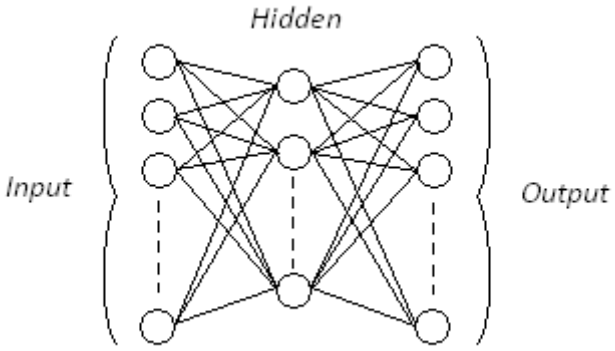
In the above algorithm the learning agent observes the current state *s*, selects an appropriate action a based on the policy π and then executes that action. It then receives a reward and observes the new state *s'*. In the next step an update of the estimate of the Q-function denoted $\hat{Q}(s, a)$ is made accordingly the training rule in equation (5). And this cycle is then repeated until the agent reaches the terminal state.

4 Q-learning with Artificial Neural Networks

If the regular tic-tac-toe game would have been implemented with a learning agent the equations and algorithms in section 3 would have been adequate. The computed Q-values would most easily have been stored in look-up tables where for example each row would consist of the possible actions and the columns would consist of the states. Hence each entry into the table would represent a state-action Q-value. However using this approach on the more complex version of noughts and crosses that has been described above would be problematic due to space complexity. The size of the state-action space would be extremely large and in the form of look-up tables and would hence be infeasible to implement.

To deal with problems like this an approximation function method can be used instead of the look-up tables, for example an artificial neural network. A neural network is in principle constructed by a set of input nodes, one or several layers of hidden nodes, and a set of output nodes.

Figure 3: Multi layered neural network



There are several different types of neural networks, for example the multi layered feed forward network displayed in *Figure 3*. We will discuss the multilayered network in detail since it will be implemented in our learning-agent.

A multilayered neural network consists of several input nodes that take the state of its environment as input. In our case it will take the current state of the game field as its input. That data is then translated through the neural network using a series of transfer functions. The most commonly used transfer functions are sigmoid due to their property to force values into a range of -1 to 1 as well as their reflective property when derived. One such transfer function is *tanh*. *Tanh* is used since it is easy to implement the back propagation algorithms where the derivative of this function is used. The number of hidden nodes that the data translates through can vary depending on the problem and one often has to test what works best. If too many hidden nodes are used they will interfere with each other and the network will not be able to converge on any relevant Q-value approximations. If too few are used they will not aid each other in the approximation. Each output node corresponds to an approximated Q-value resulting from one possible action. In the output layer there are exactly the same number of nodes as there are actions.

The Q-learning algorithm in *Algorithm 1* has been slightly adapted to function with a neural network.

Algorithm 2: Q-learning algorithm implemented with a neural network [2].

Initialize all neural network weights to small random numbers
Repeat (for each trial)
  Initialize the current state *s*
  Repeat (for each step of trial)
    Observe the current state *s*
    For all actions *a'* in *s* use the NN to compute $\hat{Q}(s, a')$
    Select an action *a* using a policy π
    $Q^{output} \leftarrow \hat{Q}(s, a)$
    Execute action *a*
    Receive an immediate reward *r*
    Observe the resulting new state *s'*
    For all actions *a'* in *s'* use the NN to compute $\hat{Q}(s', a')$
    According to Equation (5) compute $Q^{target} \leftarrow \hat{Q}(s, a)$
    Adjust the NN by back propagating the error ($Q^{target} - Q^{output}$)
    $s \leftarrow s'$
  Until *s* is a terminal state

Using the algorithm above the neural network learns a mapping from specific states to Q-values. This is done by using *Equation (5)* to calculate a target Q-value and then with the help of the *back-propagation equations (13-16)* minimize the difference between the estimated Q-values from the neural network and the target Q-value.

In this way the neural network can solve problems with large state-action spaces. It is also worth noting that when the network is trained on a large state-action space it often only trains on a small portion of it. But the neural network can generalize over the states and actions based on its previous experience with already handled state-action pairs. The network is then able to give estimated Q-values for random state-action pairs.

5 Implementation of Q-learning with ANN

5.1 Principle

From the programming point of view, the implementation of a multi-layer network, mentioned in chapter 4, is fairly simple. The weights are represented as matrices (one matrix for each layer) and all related operations can be made by either matrix addition or multiplication, which is excellent for parallelization and optimization [11]. With that in mind, Q-learning has to be implemented in the same way as the network. For Q-learning to work, as specified in chapter 3, we need the state of the environment and the list of actions to be performed on the environment. Both can be represented as multidimensional vectors, where each dimension in the vector corresponds to a specific position on the game board. Moreover, this makes the whole system based on liner algebra and can be described with a few liner expressions.

5.2 Example

In our case the 20x20 game plane becomes a 400-dimensional vector $\vec{S}$ and the opposite applies for the chosen action *a*, which is transformed back to a two-dimensional point on the plane. Furthermore, the size of the first-layer weight matrix *V* will be 266x400 and the second-layer matrix *W* will be 266x400 where number of hidden nodes is 266. Notice that rows correspond to the number of nodes and columns corresponds to the number of inputs from the previous layer. With these predefined elements, the network calculations in chapter 5.2.1 can be described as core of our implementation. The number of hidden nodes was chosen to be $\frac{2}{3}$ of the amount of input nodes. The number $\frac{2}{3}$ was chosen because a similar investigation to ours used this number in their application [2]. Whether or not this was a good number of hidden nodes will be discussed further in chapter 8 and 9.

5.2.1 Q-value calculation

The following sequence is calculated each time we want to obtain an action from the network:

$$\vec{H}^* = V\vec{S} \tag{6}$$

$$\vec{H} = \varphi(\vec{H}^*) \tag{7}$$

$$\vec{Y}^* = W\vec{H} \tag{8}$$

$$\vec{Y} = \varphi(\vec{Y}^*) \tag{9}$$

$$a = \pi(\vec{Y}) \tag{10}$$

V and W are different weights used to adjust the input/output vectors $\vec{S}$ and $\vec{H}$ and $\varphi()$ is the transfer function used on the weighted vectors. Equations (6) and (7) are used to transform and transfer the data to the second layer, in which equations (8) and (9) are used to make another transformation before it forwards the final output which is then handled by the policy π used in Q-learning, mentioned in Chapter 3. The following transfer function is used in equations (7) and (9).

$$\varphi(\vec{x}) = \tanh(x_i) \ \forall i \tag{11}$$

5.2.2 Policy

In this early implementation, we have chosen a simple policy, which may be enough to provide conclusive results:

$$\pi(\vec{x}) = \max(\vec{x}) \tag{12}$$

This means that only the highest Q-value will be chosen and no exploratory moves will be allowed. That is a fast solution but with the risk that it will never converge due to the possibility of never finding the right sequence of moves needed to win against an opponent [13]. Although it should be enough to give results indicating whether or not the learning-agent has increase its proficiency in the game.

5.2.3 Back propagation:

This part is used to update the weights of the neural network and as shown in Algorithm 2 it is crucial for approximation of the Q-function [12]:

$$\delta^Y = e\varphi'\left(\vec{Y}^*\right) \tag{13}$$

$$\delta^H = W^T\delta^Y \otimes \varphi'\left(\vec{H}^*\right) \tag{14}$$

$$\Delta W = \eta\delta^Y\vec{H} \tag{15}$$

$$\Delta V = \eta\delta^H\vec{S} \tag{16}$$

Where e is an error obtained from the Q-function (Algorithm 2), ΔW,ΔV are weight changes and η is the networks teaching speed coefficient. The following derivative of the transfer functions is used

$$\varphi'(\vec{x}) = 1 - \tanh^2(x_i) \qquad \forall \text{i.} \tag{17}$$

With these components, the system is set up for learning. It is importing to keep in mind that the learning process is slow mostly due to the algorithm complexity, which leverages on the optimization, especially with larger input spaces.

6 Learning agent versus simplistic agent

In order to measure whether or not the learning agent increases its proficiency in the game a simple but effective test has been constructed. In which the learning agent will face a simplistic agent with a static tactic.

6.1 Simplistic Agent

The simplistic agent was implemented without any form of learning capability and had only one game tactic which was simply algorithmic. The simplistic agent started its play on the far left and in the middle of the game field. From there it went right until it either hit the far right edge or was blocked and no other spaces where available to the right. When its current row was full and it had not lost it went down a row. When it reached the bottom of the game field it would start over from the top or the next row from the top with free spaces in. It pursued this tactic until it won, lost or played a draw to the learning agent.

6.2 Q-learning Agent

The Q-learning agent was implemented as specified above, with 266 hidden nodes and randomized weights throughout the network. A number of Q-learning parameters where kept constant; learning rate $\alpha = 0.1$, the discount factor $\gamma = 1$ and the networks teaching rate constant η was chosen to be 0.1. Its moves were only governed by its learning policy to always select the actions who would lead to the highest reward in the end. The reward consisted of 1 for winning, 0 for playing equal, and -1 for losing a game [7]. This was easily implemented with a data structure.

6.3 The Game Test

The Q-learning player faced the simplistic agent 350.000 times and was continuously monitored during the games. This was not only done to determine whether the agent learned anything but also to find out if the learning agent had fallen into the trap of diverging rather than converging on relevant q-values; as well to notice fluctuations due to faulty code or bugs.

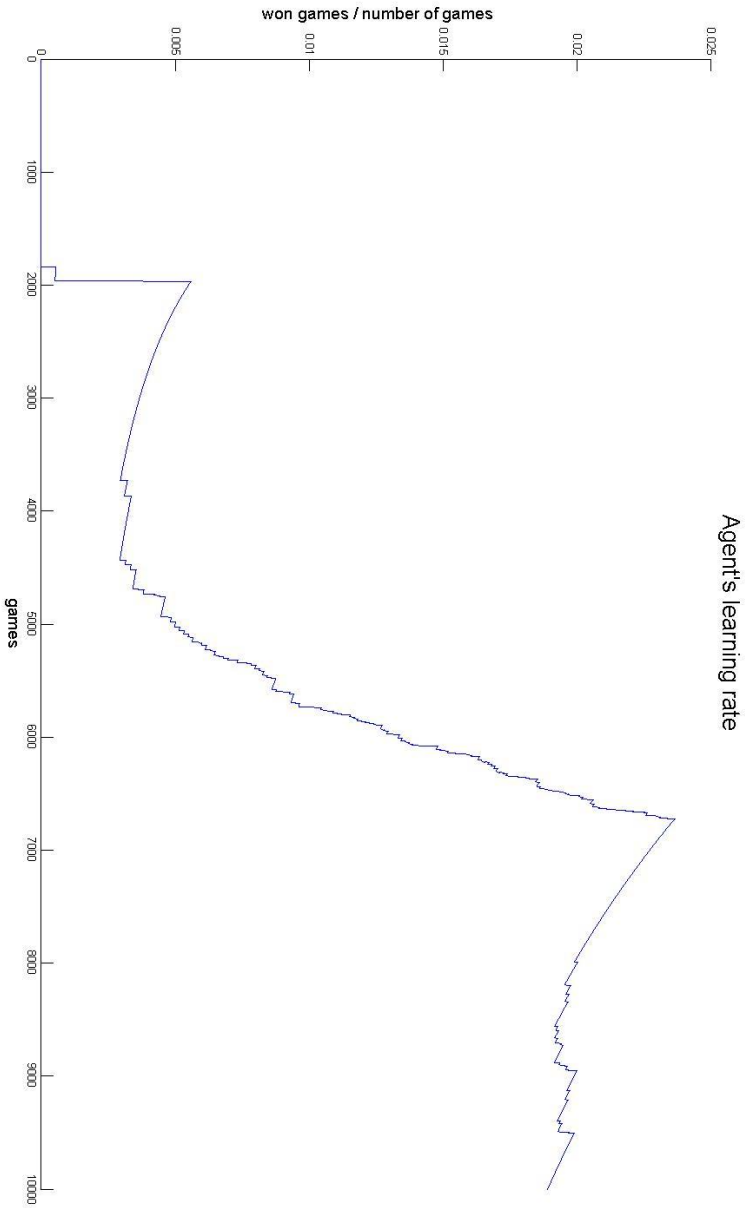
During the games data was continuously gathered on the learning agent. When he won, lost and how many times. The data is displayed in appropriate tables in chapter 7.

7 Results

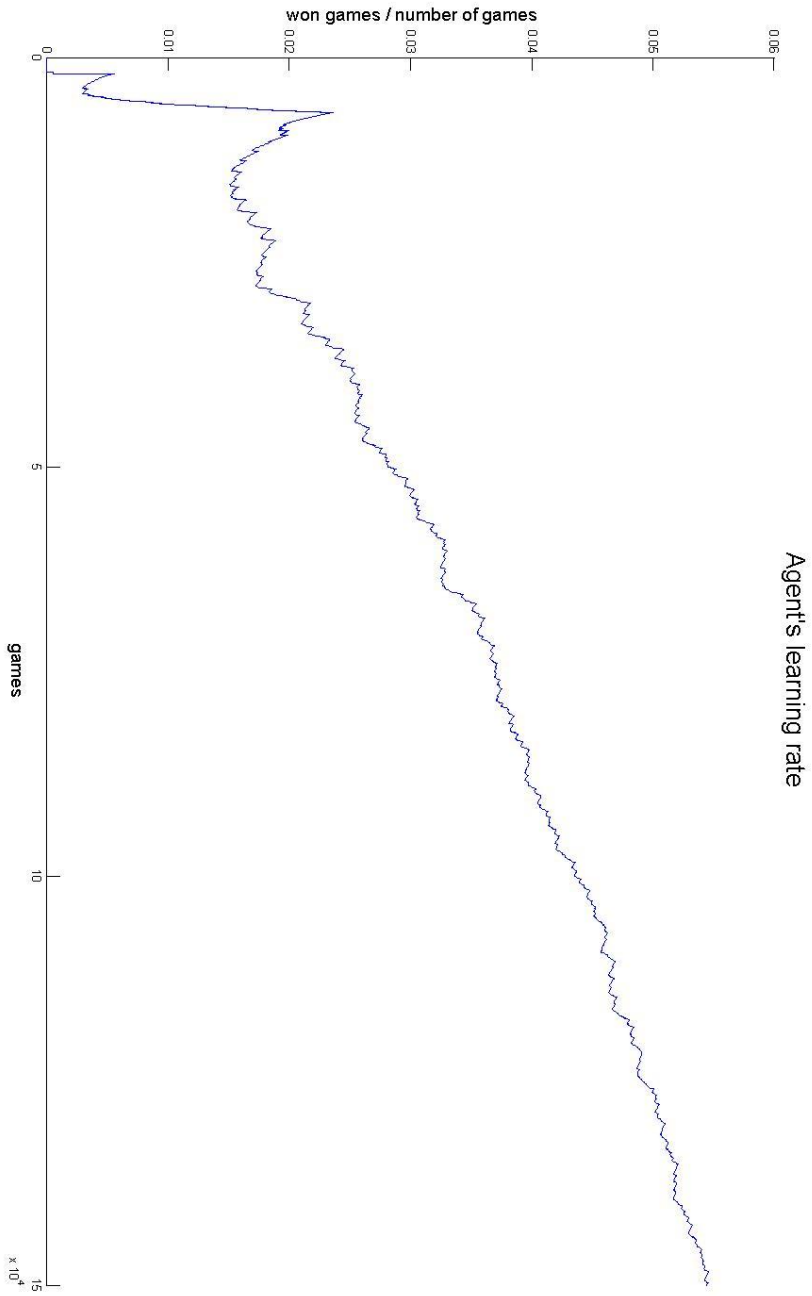
In the section below the data gathered from the matches between the two game agents is displayed. Three graphs are presented, each representing the learning progress after a set number of matches.

7.1 Simplistic Agent versus Q-learning Agent Results

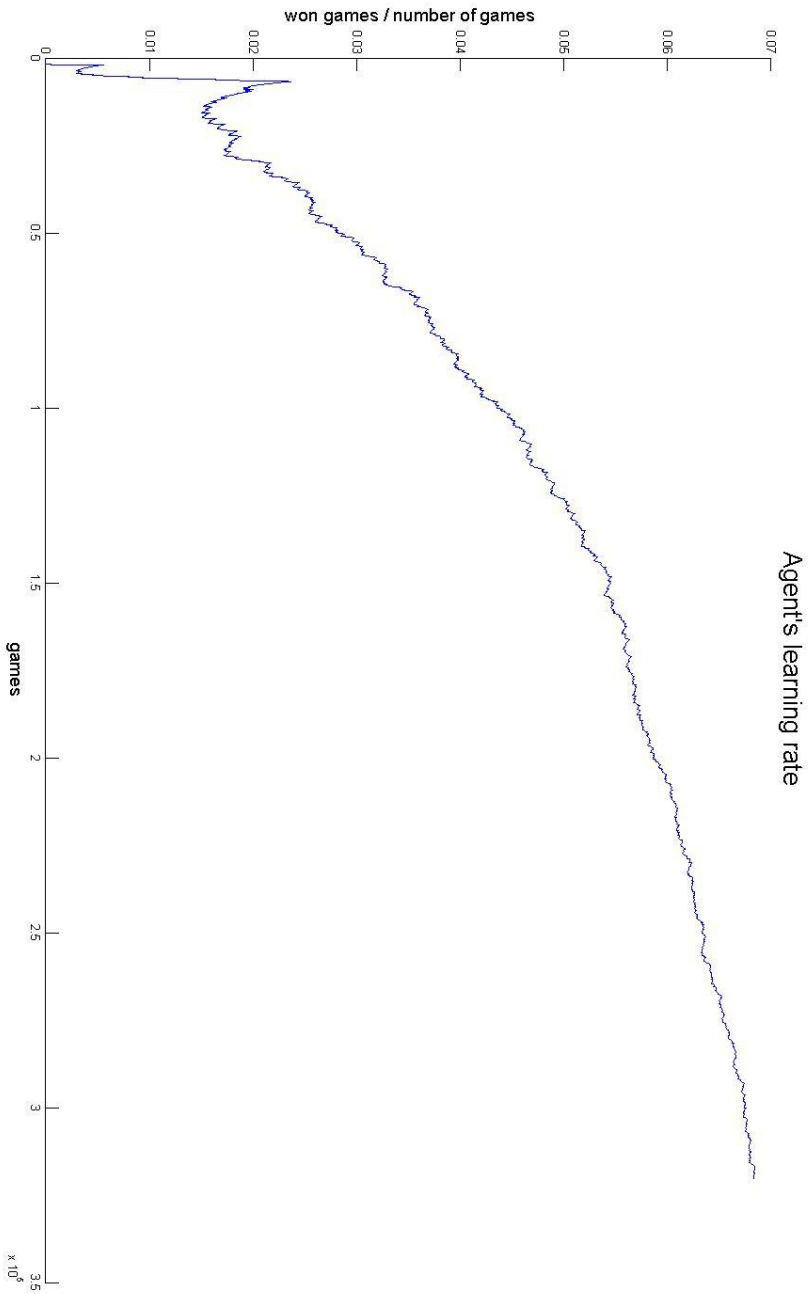
Graph 1: Learning rate after 10 000 games.



Graph 2: Learning rate after 150.000 games.



Graph 3: Learing rate after 350.000 games.



Final learning ratio: after 350.000 games is 0.0683 (6.83%).

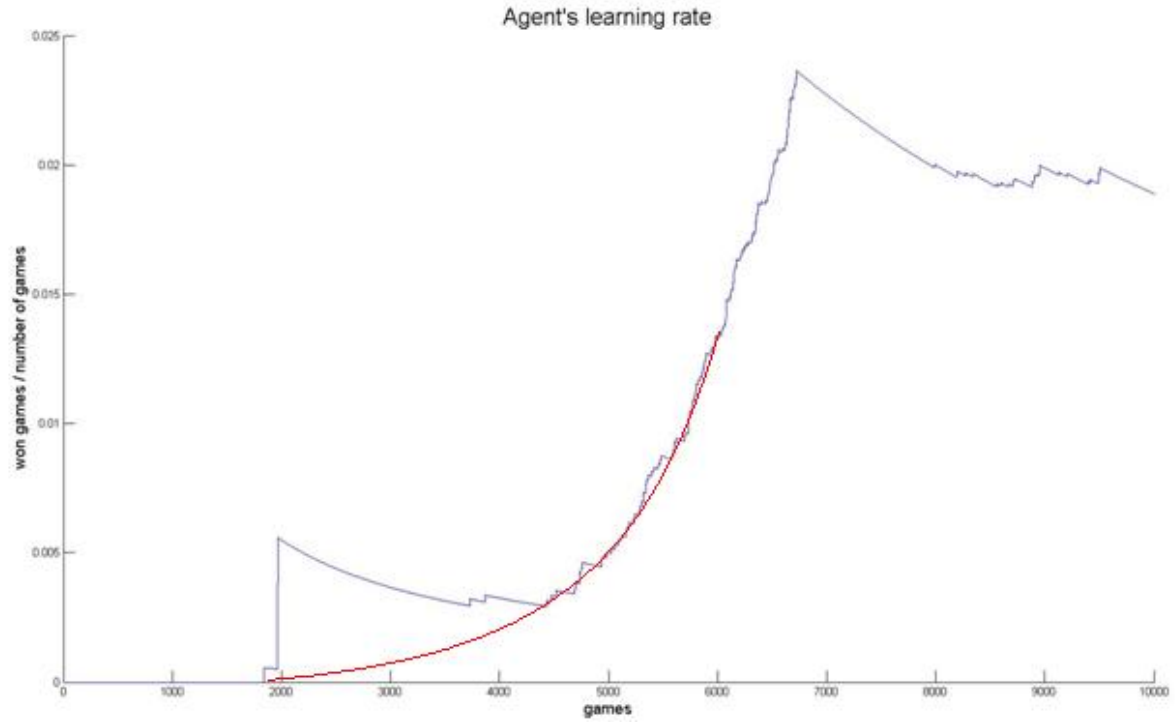
8 Discussion of Results

From *graph 1* we can see that the learning agent had a small period during the initial ~2000 games where it did not win any games. This period can be seen as an exploratory period where the agent had not yet gained any useful experience and was hence moving its pieces more or less randomly across the game field.

Around the 2000th game the agent finally made its first win. However it seems that one single win was not enough for the agent to alter its behavior in any major way, seeing as the period following the win has a slight negative gradient. This means that the learning agent kept on losing games more times than it won them. Nevertheless, it still won games which further increased its experience. This rise in experience seems to takes effect right before the 2000th game where the learning agent wins a series of games within a short number of rounds, resulting in the dramatic spike up towards 0,5% won games.

The spike seems to be coincidental since its inclination is incredibly steep and it is followed by a declination in wins. A scenario that would explain this spike is that the learning agent managed to get very lucky a few times in a row resulting in a series of wins. Judging from the smoothness of the decline the learning agent acquired enough expertise to win more times than it did before the spike, meaning losses would be less frequent.

Graph 4: Remodeled version of *Graph 1*.

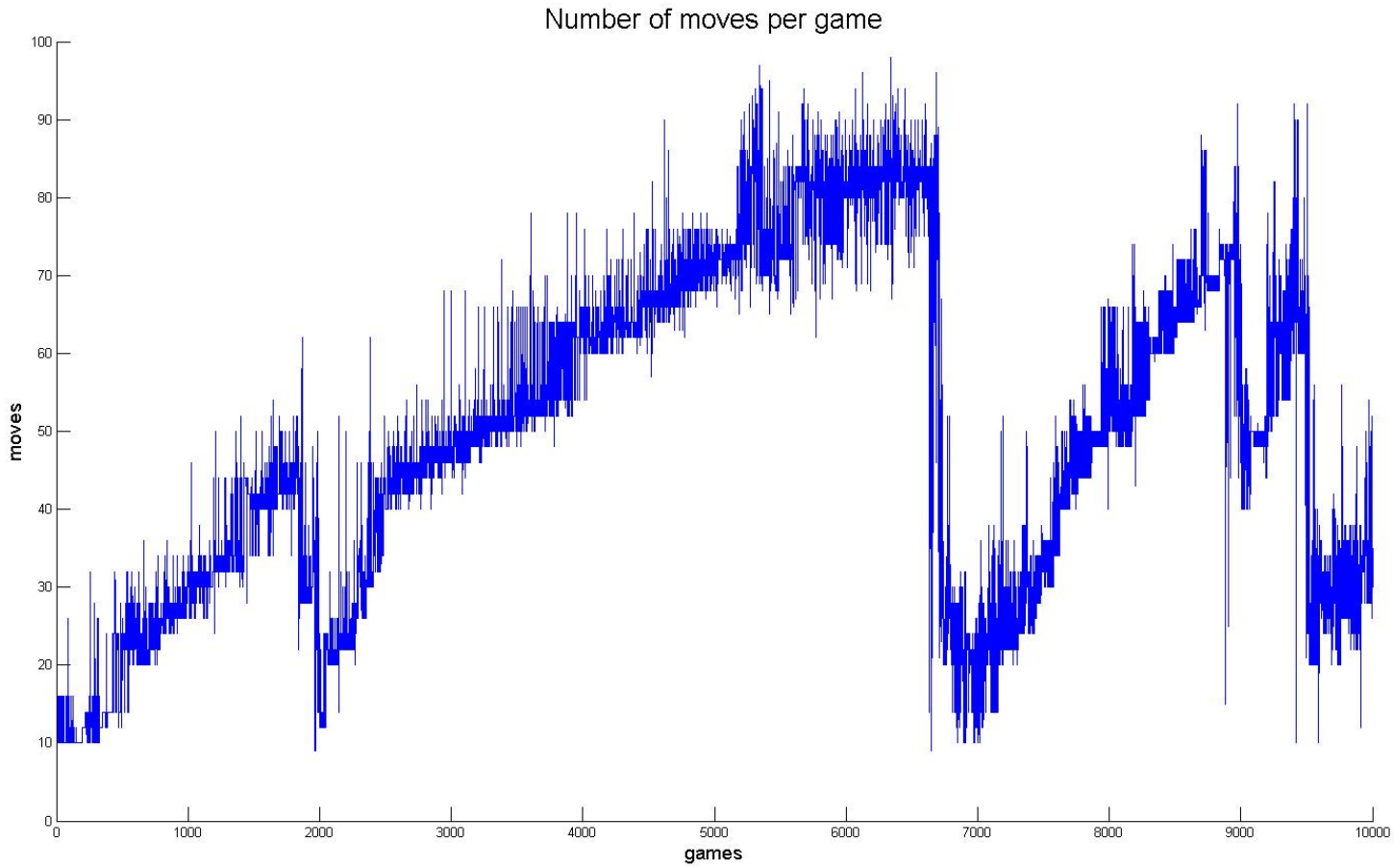


In *graph 4* above we can see a possible scenario (represented by the red line) if the agent would not have gotten lucky.

It is also very obvious in *graph 1* that the learning agent slowly recovers and begins to increase its number of wins between 4000 and 5000 games.

Around 6700 games a second peak is reached at 2,5%. The inclination towards this peak is much too smooth and long to be coincidental. However this peak is also followed by a decline in wins. The evolution of this second peak is most likely due to a feature in the simplistic agent. As stated in chapter 6.1 the simplistic agent starts in the middle and tries to fill everything towards the right and down. When it reaches the bottom of the game field it start over from the top. We believe that the learning agent managed to figure out quite fast how to win against the simplistic agent during the "first phase" of its static tactic. And that the decline is a result of the jump in the simplistic agent's tactic from the bottom the game filed to the top.

Graph 5: Number of moves per game during the first 10.000 games.



We can also see in *graph 5* that the number of moves rapidly declines for the learning agent at the same time as he starts loosing in *graph 2* at about 7000 played games. This indicates that the simplistic agent wins, and wins fast. Most likely, as stated above, due to the learning agent's inability to follow the jump in the simplistic agent's tactic.

In *graph 2* following the great peak there are numerous large oscillations. They are most likely a result from the network trying to find a good balance between the weights and counter the second part of the simplistic agent's tactic. If we follow the development of the curve into *graph 3* we see that the learning agent slowly increases his ability to win. It is also worth noting that the

curve slightly levels off more and more relative to the number of matches that have been played. There can be a number of reasons why the learning agent exhibits this behavior. Firstly the learning curve might not increase as fast as anticipated initially due to the number of hidden nodes. It might be more appropriate with more or less nodes. Secondly the curve might level off after the peak at ~6000 games because of a danger regarding neural networks and how to train them [13]. A network that is trained many times on one singular pattern might become specialized only on that particular pattern and will hence have a hard time adjusting to new patterns. In our case the learning agent might have practiced too much on the first part of the simplistic agent's tactic resulting in the slow learning curve after the second part came into play.

8.1 Possible improvements and problems

What would be of great interest to test is the number of hidden nodes. This is of interest because a more appropriate number of nodes could dramatically increase the effectiveness of the network and hence the learning capability. Another point of great interest would be to test the effectiveness of the network by increasing the number of hidden layers. However there are other, less tedious, improvements that can be done. For example, in this investigation the learning agent was trained versus a single tactic with a duality that came in effect much later. It might have been more effective to train the agent using a pure static tactic or to train the agent using random sequences. We also used a very simple policy, to always take the alternative that had the highest possibility to yield the highest reward. A more advanced policy might have resulted in a more effective learning curve. Another very valid improvement would be to let the agent play more games. Our agent only played 350.000 matches while some of the learning agents [2] trained up to 1.5 million matches.

9 Conclusion and Evaluation

In conclusion we can say that our learning agent increased his winning proficiency from 0% to about 6.7% during 350.000 matches. This is an increase in winnings that is too large to be considered a random increase and hence we can conclude that our learning agent actually learned how to play our version of noughts and crosses versus a simplistic agent with a static tactic. However we would have liked to make the agent practice at least 1 million extra matches but due to our time constraint we did not have sufficient time to train our agent that many times. As is stated in chapter 8, there are many things that can be improved as well as there are options for further investigation to determine a more effective way of developing a learning agent using Q-learning and neural networks.

10 References

[1] - R. Sutton and A. Barto. Reinforcement learning. The MIT Press, 1998

[2] – Reinforcement Learning and its Application to Othello by Nees Jan van Eck, Michiel van Wezel from Economic Institute, Faculty of Economics, Erasmus University Rotterdam, Economic Institute Report EI 2005-47

[3] - Reinforcement Learning: A Survey by Leslie Pack Kaelbling, Michael L. Littman, Andrew W. Moore Journal of Artificial Intelligence Research 4 (1996) 237-285 *Used for background reading.*

[4] - Chess game statistics: http://ssdf.bosjo.net/list.htm Last visit 2010-04-14

[5] - Chess game statistics: http://www.computerchess.org.uk/ccrl/4040/rating_list_all.html Last visit 2010-04-14

[6] - Imran Ghory: http://www.cs.bris.ac.uk/Publications/Papers/2000100.pdf Last visit 2010-04-14 *Used for background reading.*

[7] - Neural Network Algorithms: http://www.adit.co.uk/html/programming_a_neural_network.html Last visit 2010-04-14

[8] - Elman Networks: http://wiki.tcl.tk/15206 Last visit 2010-04-14 *Used for background reading.*

[9] - Learning to Play Draughts using Temporal Difference Learning with Neural Networks and Databases by Jan Peter Patist and Marco Wiering; Cognitive Artificial Intelligence at Utrecht University. *Used for background reading.*

[10] - Q-learning with Look-up Table and Recurrent Neutral Networks as a Controller for the Inverted Pendulum Problem; http://red.csie.ntu.edu.tw/NN/Demo/q_learning/Q-learning_readme.doc http://red.csie.ntu.edu.tw/BT/ClassInfo/Q_demo_WEB/Q-learning%20Demo.htm
Last visit 2010-04-14 *Used for background reading.*

[11] – ref: Örjan Ekeberg http://www.nada.kth.se/~orjan/ Last visit 2010-05-02

[12] – Neural networks with Forward/Backward Pass and Weight Updating http://www.csc.kth.se/utbildning/kth/kurser/DD2432/ann10/forelasningsanteckningar/03-multilayer-2x2.pdf Last visit 2010-04-25

[13] - http://www.learnartificialneuralnetworks.com/backpropagation.html Last visit: 2010-05-02