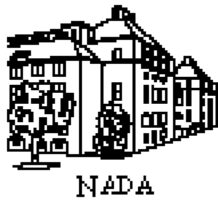
Using Temporal Difference Methods in Combination with Artificial Neural Networks to Solve Strategic Control Problems

ANDERS PERSSON

KTH Numerical Analysis
and Computer Science

Master's Degree Project
Stockholm, Sweden 2004

TRITA-NA-E04086



Numerisk analys och datalogi
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Using Temporal Difference Methods in Combination with Artificial Neural Networks to Solve Strategic Control Problems

ANDERS PERSSON

TRITA-NA-E04086

Master's Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology year 2004
Supervisor at Nada was Anders Lansner
Examiner was Anders Lansner

Abstract

In this thesis the possibilities of using temporal difference learning techniques in combination with artificial neural networks to solve strategic control problems, as seen in arcade-like games, are investigated. Earlier studies have often concentrated on simple control problems and board games. The application used here was the classic arcade game Pong, a simple 2D-tennis played by two players. Two versions of the game were implemented and studied: one discrete and one with continuous state variables but discrete actions. Firstly the discrete version was studied using the SARSA(λ)-algorithm together with different artificial neural networks and thereto a tabular version with λ set to 0, which performed very well, converging to a near-perfect solution. Another version, with λ set to 0.95 using a sigmoidal artificial neural network to store the action value function, converged to a perfect solution. These two versions were then compared and studied in detail. Secondly the continuous version was examined, using the SARSA(λ)-algorithm together with a sigmoidal artificial neural network. The results of this were promising but not satisfactory. No stable conversion to an acceptable solution was ever observed. However, altogether the results clearly show the potential of temporal difference learning methods together with function approximation in this domain.

Användningen av temporala differensmetoder tillsammans med artificiella neurala nätverk för att lösa strategiska styrproblem

Sammanfattning

I detta examensarbete utreds möjligheterna kring att använda temporala differensmetoder tillsammans med artificiella neurala nätverk för att lösa strategiska kontrollproblem, som finns i t.ex. arkadspel. Tidigare studier har ofta behandlat enkla kontrollproblem och brädspel. Applikationen som användes här var det klassiska arkadspelet Pong, ett enkelt 2D-tennis som spelas av två spelare. Två versioner av spelet implementerades och undersöktes: en diskret och en med kontinuerliga tillståndsvariabler och diskreta handlingar. Först undersöktes den diskreta versionen under användning av SARSA(λ)-algoritmen tillsammans med olika typer av artificiella neurala nätverk, samt en tabular version med λ satt till noll, som presterade mycket väl och konvergerade till en nära nog perfekt lösning. En annan version med λ satt till 0.95, som använde ett sigmodialt artificiellt neuralt nätverk för att spara handlings-värdematrisen konvergerade till en perfekt lösning. Dessa två versioner jämfördes sedan och studerades i detalj. Sedan undersöktes den kontinuerliga versionen, varvid SARSA(λ)-algoritmen användes tillsammans med ett sigmodialt artificiellt neuralt nätverk. Resultaten därifrån var lovande men inte tillfredställande. Ingen stabil konvergens till en acceptabel lösning observerades. Sammantaget visar dock resultaten på den stora potential som temporala differensmetoder har inom detta område.

Acknowledgments

This master's thesis was written at Dipartimento di Eletronica e Informazione at Politecnico di Milano and at NADA, KTH. The project was done in close collaboration with MSc student Mårten Byström. I would like to thank Professor Andrea Bonarini and PhD Matteo Matteucci, who acted as supervisors in Milano, and Professor Anders Lansner, who acted as supervisor at KTH, for their support and help. I would also like to thank PhD Mattias Wahde for his help and guidance. Thereto I would like to thank MSc student Mårten Byström for the fruitful collaboration, all the help and inspiration. Without these people this thesis could not have been done.

Contents

1 Introduction **1**
- 1.1 Previous work 1
- 1.2 Motivation 2

2 Reinforcement learning **3**
- 2.1 Overview 3
- 2.2 Finite Markov decision processes 4
- 2.3 Discrete reinforcement learning 5
- 2.4 Dynamic programming 6
- 2.5 Exploration vs exploitation 7
  - 2.5.1 ϵ-greedy action selection 7
- 2.6 A short introduction to Monte Carlo methods 8
- 2.7 The TD(0) and SARSA(0) algorithms 8
- 2.8 The TD(λ) and SARSA(λ) algorithms 9
- 2.9 Continuous reinforcement learning 10
  - 2.9.1 $SARSA(\lambda)$ with continuous state space, discrete time and discrete action space 11

3 Function approximators **12**
- 3.1 Feedforward artificial neural networks 12
  - 3.1.1 Linear discriminant functions 13
  - 3.1.2 The perceptron 15
  - 3.1.3 Gradient descent 16
  - 3.1.4 Backpropagation 17
  - 3.1.5 Radial basis function networks 19

4 Combining RL with ANN **21**
- 4.1 Gradient descent methods 21
  - 4.1.1 TD(λ) 21
  - 4.1.2 SARSA(λ) 22

5 Pong, experiments and results **23**
- 5.1 Background 23
- 5.2 Implementation 23

5.2.1 Update schematics 24
5.2.2 Reward functions 24
5.3 Extracting the results 25
5.4 The discrete version 25
5.4.1 Discrete Pong - tabular $SARSA(0)$ 27
5.4.2 Discrete Pong - $SARSA(\lambda)$ using an ANN . . . 29
5.4.3 RBF network 30
5.4.4 Sigmoidal ANN with sigmoidal output 30
5.4.5 Sigmoidal ANN with linear output 32
5.4.6 Comparisons between tabular $SARSA(0)$ and $SARSA(\lambda)$ with a sigmoidal ANN 33
5.5 The continuous version 34
5.5.1 Input to the learning algorithm 35
5.5.2 Experiment setup 36
5.5.3 Observed behavior 36

6 Conclusions and future work **38**
6.1 Usage of RL algorithms on arcade-like games 38

Bibliography **40**

Chapter 1

Introduction

In this thesis the possibility to combine existing reinforcement learning and function approximation techniques and to apply them to general and strategic gaming applications is investigated. The main application used here is Pong, a 2-dimensional tennis for two players. The intention is to test the performance of some of the most popular reinforcement learning and function approximation algorithms in order to clarify their limitations and possibilities.

1.1 Previous work

The idea of building intelligent machines has fascinated the human for ages. Already the Egyptians were considering this fact more than 4000 years ago. Over the years many theories have been developed, but it is not until in the last 50 years, after the introduction of the computer, that the research in the artificial intelligence and machine learning community has evolved into its own scientific branch.

In 1950 Claude Elwood Shannon [24] wrote a proposal for a chess program, and in 1959 Arthur Samuel [22] constructed a computer program which learnt how to play checkers by playing against itself. The research has in the recent years more focused on reverse engineering of biological models in the attempts of trying to create machines that can solve problems and reason like humans. Neural networks, a very simplified model of the brain, have been used successfully in several applications. Sutton's and Barto's [25] work on reinforcement learning algorithms also has gained a large interest in the machine learning community. In 1992 Tesauro [27] successfully implemented these two theories and constructed a Backgammon player which could compete with the top human players in the world. This demonstration showed the potential in combining these types of algorithms, but despite that many researchers have tried to apply them to similar problems during

the last decade, few have gained a great success [5] [23] [12]. This thesis handles an application with a non-trivial task, where the learning agent has to adapt a strategic behavior playing a two-dimensional tennis called Pong. There has not been any research for this particular type of problem earlier.

1.2 Motivation

One definition of intelligence is that an intelligent individual should be capable of surviving in an environment that it never has seen before, and therein be able to adapt an appropriate behavior. This clearly demands a way to evaluate and classify new situations, and thus advanced function approximators need to be developed in the search for intelligent machines. This is especially important when the individual is living in a continuous environment.

To be able to learn and adapt new behavior, the individual also needs to consider the inputs from the environment and draw conclusions out of this information. Reinforcement learning provides the theory of training an agent to maximize the rewards available in its surroundings, and hence to maximize its probability to survive in the environment. Certain tasks are also too complex to describe in a static computer program, which is the common procedure today. A way of dynamically learning and developing a program is therefore needed for some tasks.

If these two research areas can be combined in an efficient and stable framework, a more intelligent behavior can be expected to be developed than today's machines possess. The theory for machine learning is still in its infancy, and one big difficulty among others is the lack of sufficient computational power. Computer technology though, is also progressing very fast today, and this will probably help the machine learning area to make significant progress the coming years.

The field of computer games is a large application area for artificial intelligence algorithms. Much research has gone into developing intelligent agents for board games such as backgammon and othello but arcade-like game applications have often been neglected. This thesis aims to introduce self learning into this field and investigate the weaknesses and strengths of today's algorithms.

Chapter 2

Reinforcement learning

The technique studied in the reinforcement learning community may be seen as a formalization of the old behavioristic ideas in psychology. All the algorithms discussed search for an agent, which learns to perform a task. It learns this task by using a reward signal, which informs it on how well it is performing for the moment. The agent is not told how to act. It is rather the environment that the agent lives in that informs it on whether it is progressing or not. The reward signal is often noisy, as in the case of game playing. Here it is in most cases delayed until the point of the game, when it is known if the game was played well or badly - when somebody wins. One of the major difficulties, called the credit assignment problem, is then to know which actions that were most responsible for the win or the loss.

This section presents several discrete and continuous *reinforcement learning* (*RL*) methods. The field of RL has been summarized by, amongst others, Kaelbling [11] and Sutton and Barto [25].

2.1 Overview

Reinforcement learning algorithms optimize the behavior of an agent trying to solve a given task. The agent interacts with the environment it finds itself in by taking actions. The environment is presented to the agent through a signal, which contains the parts of the environment the agent perceives. Here Sutton and Barto's definition of the environment is used, and anything that the agent cannot change arbitrarily is considered to be outside of the agent, and thus part of the environment [25]. Each unique signal value is called a *state* s. A given *reward function* $r(s)$ maps states to rewards, which are scalar values. At each time step the agent has to make a decision on which currently available *action* a to take depending on the state of the environment. In doing this it struggles to maximize the reward it receives in the long

run. This is the only goal of the agent.

As a measure of a state's desirability, in terms of how much reward that can be expected after visiting that state, reinforcement learning algorithms model a *value function* V. This function maps a state to a scalar value, which is the measure of the state's desirability. The value function is specific to the *policy* π, which the agent is following, where the policy instructs the agent what to do in each state. The policy π maps each state to an action. When the agent is in a certain state of the environment it uses the policy to choose its next action. The state value function V maps each state to a value, which expresses the expected reward gained from that state, including the immediate reward given to the agent upon entering the current state. In a similar way the *state action value function* Q maps a state and an action taken in that state to a value defined in the same way as for V. The term value function will be used as a collective name for V and Q when both are applicable.

The reward function gives feedback on how well the agent is performing for the moment. If the reward for being in a state is positive the agent is encouraged to visit that state again, and if the reward is negative the agent is discouraged to visit that state again. This information is represented in the value function, which, as stated above, contains the longtime expected reward of visiting a certain state.

In some algorithms a model of the environment can be used, but in other algorithms it is omitted.

2.2 Finite Markov decision processes

Many reinforcement learning problems can be fully described by a finite *Markov Decision Process* (*MDP*). A finite MDP consists of:

- a finite set of states, S. This set contains all reachable states.
- a finite set of actions, A. This set contains all doable actions.
- a reward function $r(s)$ This function maps a state to the reward the agent receives when visiting a particular state.
- a state transition function, $P(s, \acute{s}, a)$ defined on all states in S for all actions in A. This function gives the probability of a transition from one state s to $\acute{s}$ when taking action a.

If the transition function only depends on the current state and not on any previously traversed states or any previous actions, the process is said to be *markovian.* A state s is said to be absorbing when

$P(s, \acute{s}, a) = 0$ for all $s \in S$ and any $a \in A$ except for when $\acute{s} = s$. Thus as soon as the agent reaches an absorbing state, it will not exit that state. An absorbing state can normally be thought of as a goal state for the problem, it is therefore also often referred to as a *terminal state*. When dealing with game playing problems, there normally exists at least one terminal goal state. If S contains at least one reachable absorbing state, then, using the law of large numbers, it is also known that the agent, after some time, will end up in this state. The reward for remaining in an absorbing state after the entry is zero.

2.3 Discrete reinforcement learning

This section summarizes the concepts of reinforcement learning in the discrete case. Discrete here implies that the state space of the environment is discrete and finite, and that the time is divided into discrete steps.

The policy, the value function and their determination

The policy π maps a state to an action, as mentioned in section 2.1. The optimal policy $V^*(s)$ has a value function with the following property

$$V^*(s) \geq V^{\acute{\pi}}(s), \forall s, \tag{2.1}$$

for all possible policies, $\acute{\pi}$. A more formal definition of what up until now has been called the expected reward received in the long run is

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{0}^{T} \gamma^k r_{t+k+1}. \tag{2.2}$$

Hence, R_t is defined to be the total reward received by the agent traversing the state space from state t, and following policy π until reaching an absorbing state at time T. Remember that the reward for staying in an absorbing state is 0.

The constant γ is the discount factor, a number in [0,1]. The effect the discount factor has on the learning may be seen as a tool for weighting rewards closer in time higher than rewards further away. It also makes the sum in Eq. 2.2 bounded and therefore simplifies calculations and proofs based on it. The R_t term is often referred to as the *return* of a state t. Sutton and Barto [25] defines $V(s)$ and $Q(s, a)$ as

$$V(s)^{\pi} = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{2.3}$$

$$\text{and} \quad Q(s,a)^{\pi} = E_{\pi}\{R_t | s_t = s, a_t = a\} =$$
$$E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\}, \tag{2.4}$$

where E_{π} is the expected value for an agent that follows policy π.

2.4 Dynamic programming

In 1957 Bellman [2] published a key paper in which he introduced the term *dynamic programming (DP)*, which refers to a set of algorithms that may be used to calculate an optimal policy for an MDP. Dynamic programming algorithms need a perfect model of the environment in order to be able to function, and they all rely on the Bellman equation

$$\begin{aligned} V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} = \\ = E_{\pi}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} = \\ = \sum_a \pi(s,a) \sum_{\acute{s}} P^a_{s\acute{s}}[R^a_{s\acute{s}} + \gamma E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} = \\ = \sum_a \pi(s,a) \sum_{\acute{s}} P^a_{s\acute{s}}[R^a_{s\acute{s}} + \gamma V^{\pi}(\acute{s})]. \end{aligned} \tag{2.5}$$

$P^a_{s\acute{s}}$ is the probability of moving from state s to state $\acute{s}$ taking action a. The Bellman equation states that it is possible to calculate the value of a state s, i.e. $V(s)$, by recursively stepping through all states that are *connected* to s. The term connected is here defined as: two states s, $\acute{s}$ are connected if and only if $P^a_{s\acute{s}} > 0$, for any $a \in A$. The procedure is repeated for each $\acute{s}$ connected to s until an absorbing state is reached. The probability $P^a_{s\acute{s}}$ weighs the impact that every $R^a_{s\acute{s}} + \gamma V^{\pi}(\acute{s})$ term has on the value of state s. It is important to notice that the transition probability function P is needed for the evaluation of the equation. If P is known the equation expands into a system of equations that can be solved in a straightforward manner. Bellman pointed out one problem he called *the curse of dimensionality*, i.e. the number of calculations needed to solve the system of equations grows exponentially with the number of input dimensions. This is a well known problem in machine learning and is more a problem concerning the fact that the state space grows exponentially with the number of input dimensions than with the Bellman equation approach of solving MDPs.

The *policy improvement theorem* states that when the policy is changed to follow a new path, which is better than the old path according to the value function, an improved policy is produced. The policy update can be achieved by a simple greedy update of the policy with respect to the value function. In general this means that as long as the policy is changed in a greedy fashion with respect to the value function the policy is guaranteed to improve, unless the optimal policy already has been found. This property is exploited by dynamic programming methods as well as by the vast majority of reinforcement learning algorithms.

2.5 Exploration vs exploitation

During training, when using an estimation of the true value function, the agent may be able to solve the problem at hand, when it in every state only chooses the optimal action according to the estimated value function. However, there is no guarantee that this is the best possible policy. A better solution might be found, if the agent is allowed to explore new policies. This is the problem of balancing the exploration versus the exploitation. The procedure of choosing which action to take is often referred to as *action selection*.

The ϵ-greedy method described below is a very straightforward technique. There exist many other techniques of doing action selection, some more advanced than others. Further examples can be found in the work of Thrun [29].

2.5.1 ϵ-greedy action selection

One of the simplest approaches in the action selection is to introduce a parameter ϵ into the procedure, which determines the probability of performing a random action. Hence, at every action selection step the agent will choose to take a random action with probability ϵ and a greedy action with probability $1 - \epsilon$.

It is often advantageous to take many random actions in the beginning of a search trying to explore as much as possible of the state space, and then, as learning progresses, lower the amount of random actions. Lowering the value of ϵ is equal to taking more greedy actions and exploiting what has been learned. A problem with the ϵ-greedy method is that there exists no straightforward way of choosing the ϵ value. It is in most cases very hard to choose when to increase or decrease the amount of random actions taken by the agent.

2.6 A short introduction to Monte Carlo methods

The dynamic programming methods discussed previously use an exact model of the environment, the reward and the transition functions associated with it. In most real world applications this information is not known. *Monte Carlo methods*, often referred to as *MC-methods*, is a broad class of methods applied e.g. in solving integrals, which also may be used to solve RL problems. McKay's work provides a survey of MC-methods [14]. MC-methods differ from dynamic programming methods in one essential way - in order to function they only need samples from an agent interacting with the environment. The samples contain information about which states were visited, which actions were made, when rewards were received and the magnitude of those rewards.

A simple Monte Carlo update rule could for example be as in equation

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)] \tag{2.6}$$

2.7 The TD(0) and SARSA(0) algorithms

Temporal difference (TD) methods were used in, amongst others, Samuel's checkers player [22] and in Holland's Bucket Brigade [9], an adaptive general purpose machine learning algorithm partly similar to $SARSA(0)$. It was not until Sutton published a key paper on Temporal Difference learning in 1988 [26] though, that the methods were properly understood. Sutton proved some of the convergence properties of the $TD(\lambda)$ algorithms discussed below.

One of the key features in temporal difference learning is that the updates of the value function are partly based on other value function estimates. This is called *bootstrapping*. There exists a close relationship between TD, DP and MC methods, which all are reinforcement learning methods capable of solving MDPs. TD methods use the bootstrapping of DP, and they have the essential property of MC that they do not need an exact model of the environment. Comparing the MC update rule in equation 2.6 with the TD(0) update rule in equation 2.7 it becomes obvious that the main difference is the target. The TD(0) update rule uses $r_{t+1} + \gamma V(s_{t+1})$ as a target, where a simple MC would use R_t. This means that the TD(0) algorithm updates the estimation of $V(s_t)$ making use of the estimation of $V(s_{t+1})$. It is often a great advantage not having to wait for the final return before any update

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):

> Initialize s
> Choose a from s using policy derived from Q (e.g. greedy)
> Repeat (for each step of episode) :
>
> > Take action a, observe r, $\acute{s}$
> > Choose $\acute{a}$ from $\acute{s}$ using policy derived from Q (e.g. greedy)
> > $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(\acute{s},\acute{a}) - Q(s,a)]$
> > $s \leftarrow \acute{s}, a \leftarrow \acute{a};$
>
> until s is terminal

Table 2.1. Pseudo code for the SARSA(0) algorithm.

takes place, as in MC methods, especially when the *episodes* are long. An episode in this work is defined to be one application run, beginning in a starting state and ending in a terminal state. The 0 in TD(0) points out the fact that the algorithm only updates one previous state, whereas the TD(λ) updates an arbitrary number of previously visited states.

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.7}$$

In 1994 Rummery and Niranjan [21] proposed an algorithm which they called "modified Q-learning". It was later renamed to SARSA by Sutton in 1996. The SARSA and TD algorithms are very similar. SARSA is a temporal difference method designed to solve control problems. Hence, it uses the Q(s,a) function instead of V(s). The SARSA(0) algorithm may be written in pseudo code as in table 2.1.

2.8 The TD(λ) and SARSA(λ) algorithms

$TD(0)$ updates only the neighboring state with respect to the visited one, i.e. $V(s_t)$ is updated using $V(s_{t+1})$ and r_{t+1}. The novelty of the $TD(\lambda)$ and $SARSA(\lambda)$ algorithms is that they provide means of propagating the knowledge that is handed to them at time $t+1$ backwards to as many states as wanted. As a tool for making this possible a data structure called eligibility traces and a weighting factor called λ are introduced. Each state passed through is associated with an eligibility trace. The eligibility trace for a state stores a weight measuring the impact that any later event should have on the state value. At

Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$
Repeat (for each episode):

Initialize s
Repeat (for each step of episode):

$a \leftarrow$ action given by π for s
Take action a, observe reward, r, and next state, $\acute{s}$
$\delta \leftarrow +\gamma V(\acute{s}) - V(s)$
$e(s) \leftarrow e(s) + 1$
For all s :

$V(s) \leftarrow V(s) + \alpha\delta e(s)$
$e(s) \leftarrow \gamma\lambda e(s)$

$s \leftarrow \acute{s}$

until s is terminal

Table 2.2. TD(λ).

each time step the eligibility traces are updated according to Eq. 2.10. The scalar λ, which lies in $[0, 1]$, sets the horizon of this impact. The generalized target used for TD(λ), the λ-return R_t^λ, is

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}, \tag{2.8}$$

where $R_t^{(n)}$ is the corrected n-step truncated return.

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \tag{2.9}$$

The $TD(\lambda)$ algorithm is given in table 2.2. Another way of looking at the $TD(\lambda)$ algorithm is as a bridge between MC methods and temporal difference methods, where the two extremes being lambda equals zero, which gives $TD(0)$ and λ equals one, which gives an MC method.

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), if & s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1, if & s = s_t \end{cases} \tag{2.10}$$

2.9 Continuous reinforcement learning

The standard reinforcement learning algorithms discussed were all designed to solve discrete problems with discrete time, state and action

spaces. Reinforcement learning in continuous environments is a subject to intense research and many questions remain unanswered. Partly continuous reinforcement learning has been studied by amongst others Baird et al. [1] and Duff et al. [7]. Doya [6] was the first to propose a completely continuous reinforcement learning algorithm.

In the case of game playing problems one often have a continuous state space but a discrete action space and discrete time steps. This is a problem group, which this thesis partly investigates.

2.9.1 $SARSA(\lambda)$ with continuous state space, discrete time and discrete action space

Since TD(λ) uses only the state value, this algorithm demands that the next possible states must be calculated and evaluated in order to compare which one is the preferred. This can be very difficult for continuous search spaces, and thus the control algorithm SARSA(λ) is preferable. When dealing with continuous state variables some sort of function approximator is necessary for the modeling of the action value function. A normal tabular data structure may not be used simply because there is no way to index a continuous variable into a matrix. The presentation of the state variables normally also need to be handled in a different way using e.g. a gaussian interval coding as discussed in chapter 4. Apart from this however, the $SARSA(\lambda)$ algorithm may be used without any changes.

Chapter 3

Function approximators

Function approximation is a very important element in artificial intelligence. There is no way to store all possible situations an intelligent agent can appear in, unless the environment is represented by a finite set of states. Even with a finite number of states it might not be realizable to store all states with today's computer power if the number of states is large. Hence a way to generalize and classify input data never seen before from previously learned experiences is needed. This technique of mapping an input to an output is the task of the function approximator.

3.1 Feedforward artificial neural networks

One of the most interesting and investigated function approximator is inspired from nature's by far most skillful generalizer, the brain. This section begins with a brief presentation of the history of *artificial neural networks* (ANN), and it then proceeds with a more detailed study of several interesting topics.

The first proposal of how the brain might work in terms of threshold logic and cooperating nodes was presented by Bain [16] already in 1873. However he did not have the theoretical tools to develop a justified and formal model. It took until 1943 before McCulloch and Pitts [13] formally stated how the smallest known computing parts of the brain, the neurons, might work. Together, Pitts - a logician and McCulloch - a neurophysiologist, approached the idea of how the logic of the neuron computation might work. In 1957 Rosenblatt [19] presented the first concrete neural network model, the perceptron depicted in figure 3.2. 1960 Widrow and Hoff presented a slightly modified version of this called ADALINE (ADAptive LINEar element) [31]. Their work together led to the formalization of the Delta learning rule in 1962, also called the Widrow-Hoff learning rule. This rule provides a

way to train the neurons based on a pattern recognition error, and it is the basis for the backpropagation networks discussed below.

However, in 1969 Minsky and Papert [15] presented a paper in which they described the severe limitations of the perceptron. They showed that it could not separate two nonlinearly separable classes. These negative results diminished the interest in the neural network research for two decades.

Not until the 1980's would the interest for neural network research bloom again. Inspired by Hopfields work in 1982 [10] on building neuronal networks out of the neuron model, an intense work was pursued to extend the Delta rule to multiple layers. In 1986 Rumelhart et al. [20] all independently of each other developed the same ideas which were to be called the Backpropagation Algorithm. Multiple layered perceptrons do not suffer from the limitations that the single perceptron does. The describing name of training these comes from the way in which the algorithm propagates the pattern recognition error backwards in the network. A deeper description on ANNs can e.g. be found in the books of Bishop [3] and Gurney [8].

3.1.1 Linear discriminant functions

A discriminant function $y(\mathbf{x})$ classifies an input vector $\mathbf{x}$ into a certain class C_k depending on the output $y(\mathbf{x})$ that the input vector produces. For a simple case with only two classes one can define the discriminant function $y(\mathbf{x})$ so that it assigns vector $\mathbf{x}$ to class C_1 if $y(\mathbf{x}) \geqslant 0$ and to class C_2 if $y(\mathbf{x}) < 0$. A linear discriminant is a discriminant function linear in the components of $\mathbf{x}$ such that

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \omega_0 \tag{3.1}$$

Here $\mathbf{w}$ is referred to as the *weight vector* and ω_0 as the *bias*. If the input vector space is two-dimensional the decision boundary will be a straight line. The weight vector $\mathbf{w}$ will thereby determine the orientation of the decision plane, and the bias determines the perpendicular distance to the origin of the input space. One can consider the linear discriminant as a function approximator, where $\mathbf{w}$ must be adjusted to get the desired output $y(\mathbf{x})$, when a specific $\mathbf{x}$ is presented. Linear discriminant functions as in Eq. 3.1 can also be represented as a network diagram as shown in figure 3.1.

An interesting fact in this matter is that the simple mathematical model of the biological neuron stated by McCulloch and Pitts mentioned above, is a form of the linear discriminant. This model also adds an *activation function* $g(\cdot)$ to Eq. 3.1 producing the following equation

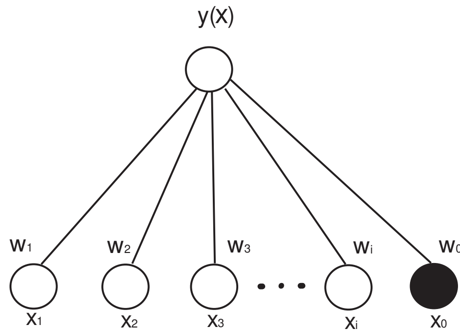


Figure 3.1. The Linear Discriminant illustrated as a network.

$$y(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + \omega_0) \tag{3.2}$$

The activation function $g(\cdot)$ is normally a monotonic and nonlinear function. The decision boundary generated from Eq. 3.2 is linear because of the monotonic nature of $g(\cdot)$. McCulloch and Pitts used the Heaviside step function as activation function

$$g(a) = \begin{cases} 0, when & a < 0 \\ 1, when & a \geq 0 \end{cases} \tag{3.3}$$

In this neuron model the input vector $\mathbf{x}$ represents the input activity transmitted from neighbouring neurons, the weight vector $\mathbf{w}$ the strength of the corresponding connections, called synapses, and the bias $\mathbf{w_0}$ is the firing threshold for the neuron. This thresholding unit has some very interesting properties inspired by the biological neuron, and these are the type of units that Rosenblatt and Widrow and Hoff conducted work upon.

Another interesting activation function worth mentioning here is the sigmoid function given by

$$g(a) = \frac{1}{1 + e^{-a}}, \tag{3.4}$$

which is the most used activation function for backpropagation networks partly due to its simple derivative.

Linear separability and generalized linear discriminants

One of the key limitations of linear discriminants is that the decision boundary is linear, or for higher dimensions, hyperplanar. Hence only linearly separable problems can be solved.

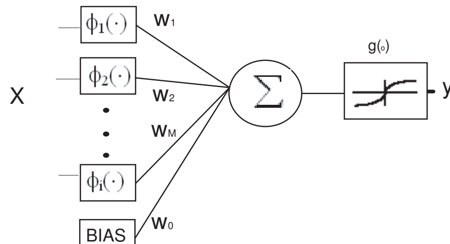


Figure 3.2. The perceptron sums up all its weighted inputs and adds an activation function g()

What may be done is to map a non-linear problem to a linear separable problem by transforming the input vector $\mathbf{x}$ with a set of M predefined non-linear functions $\phi_i(\mathbf{x})$, called *basis functions*. The output can then be represented as a linear combination of these functions

$$y(\mathbf{x}) = \sum_{i=1}^{M} w_i \phi_i(\mathbf{x}) + \omega_{k0} \tag{3.5}$$

With the right choice of basis functions the non-linear problem is now mapped into a new representation space. Now the identical but linear problem may be solved. These generalized versions of the linear discriminant is called *generalized linear discriminants*.

3.1.2 The perceptron

Rosenblatt studied single layer nodes in networks with threshold activation functions. He called these type of nodes perceptrons. In figure 3.2 the functionality is described. The perceptron uses preprocessing input units, which also the generalized linear discriminant in Eq. 3.5 does. These static elements must be well designed for the particular classification problem. It also has a threshold activation function as described in Eq. 3.2 which may be of the same type as in Eq. 3.3. The equation for the perceptron becomes

$$y(\mathbf{x}) = g\left(\sum_{i=1}^{M} w_i \phi_i(\mathbf{x}) + \omega_0\right), \tag{3.6}$$

where M equals the number of inputs. The perceptron's output is then classified into one of the classes described by the threshold function. The bias ω_0 can be included in the weights as w_0 without any loss of generality, if the corresponding input x_0 is a bias term always set to a

constant, i.e $x_0 = 1$, and thus

$$y(\mathbf{x}) = g\left(\sum_{i=0}^{M} w_i \phi_i(\mathbf{x})\right), \tag{3.7}$$

which is an easier representation to manipulate.

Perceptron training

To train the perceptron into a better performance an *error* E with respect to the perceptron's output needs to be defined, to clarify if and how the weights $\mathbf{w}$ need to be adjusted. The perceptron's performance can be measured from how often the desired output of a specific $\mathbf{x}$ is received, and an E can then be calculated. A procedure which minimizes the error by changing the weights is also needed. The perceptron moves the weight vector towards the input vector in regard to the size of the error according to

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \eta E \mathbf{x_t}, \tag{3.8}$$

where $\mathbf{w_{t+1}}$ is the new updated vector and η is the *learning rate*. This parameter is needed because there must only be a small step taken toward the correct output vector in order to not disrupt previously learned patterns. The perceptron training rule uses $(y - v)$ as the error expression, where v is the target vector describing the correct output for the input vector $\mathbf{x}$, and Eq. 3.8 becomes

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \eta(y_t - v_t)\mathbf{x_t}. \tag{3.9}$$

3.1.3 Gradient descent

The error E presented in Eq. 3.8 depends upon the parameters of the output, and thus it can be expressed by $E(\mathbf{w})$, where $\mathbf{w}$ is the complete parameter vector. If the function $E(\mathbf{w})$ is differentiable, then the minimum error can be found by changing $\mathbf{w}$ towards the direction in function space where $E(\mathbf{w})$ decreases most, i.e. in the direction of the negative *gradient* $-\nabla_{\mathbf{w}} E$. By repeated iterations of this procedure an update rule of the new parameter vector becomes

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta \frac{\partial E(\mathbf{w_t})}{\partial \mathbf{w_t}}. \tag{3.10}$$

It can also be useful to state the update rule as the difference between the new and the old weight vector

$$\Delta \mathbf{w} = \mathbf{w_{t+1}} - \mathbf{w_t} = -\eta \frac{\partial E(\mathbf{w_t})}{\partial \mathbf{w_t}}. \tag{3.11}$$

Delta rule

The simplest form of error function for $E(\mathbf{w})$ that may be used for one output pattern y_k and correct target pattern v_k, is the sum-of-square error function

$$E(\mathbf{w}) = \frac{1}{2}(v_k - y_k)^2. \tag{3.12}$$

A derivation of this function with respect to the weights w_i, if y_k is on the same form as the generalized linear network function in Eq. 3.5, yields

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = -(v_k - y_k)\phi(\mathbf{x}). \tag{3.13}$$

Writing the weight update on the same form as in Eq. 3.11 leads to following equation

$$\Delta\mathbf{w} = -\eta(v_k - y_k)\phi(\mathbf{x}), \tag{3.14}$$

which is more commonly known as the *delta rule* first referred to by Widrow and Hoff in their training of the ADALINE nodes, and it is therefore also known as the Widrow-Hoff rule.

3.1.4 Backpropagation

The backpropagation rule was first popularized by Rumelhart et al. but similar ideas had also been developed earlier by a number of researchers including Werbos [30] and Parker [17]. The backpropagation rule is actually an extension of the delta rule in Eq. 3.14 and can be used for multilayered networks. To easier understand the nature of the backpropagation rule, first the delta rule in Eq. 3.14 will be more closely examined and the update more formally defined . Thereafter the rule for multiple layers will be explained.

Applying the chain rule for partial derivatives to the derivative of the error $E(\mathbf{w})$ in Eq. 3.13 yields

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = \frac{\partial E}{\partial y(\mathbf{x})}\frac{\partial y(\mathbf{x})}{\partial w_i}, \tag{3.15}$$

where one can see that these two terms correspond to

$$\tfrac{\partial E}{\partial y(\mathbf{x})} = -(v_k - y_k) \quad and \quad \tfrac{\partial y(\mathbf{x})}{\partial w_i} = \phi(\mathbf{x}). \tag{3.16}$$

Hence the delta rule can be written

$$\Delta\mathbf{w} = -\eta\frac{\partial E}{\partial y(\mathbf{x})}\frac{\partial y(\mathbf{x})}{\partial w_i}. \tag{3.17}$$

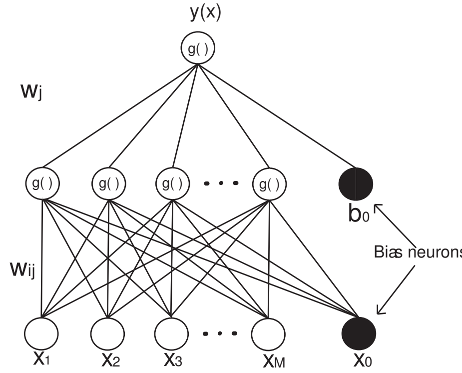


Figure 3.3. The multilayered perceptron

The chain rule is now applied to the perceptron update rule from Eq. 3.7, which also adds an activation function, and one more term comes into the derivative

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = \frac{\partial E}{\partial y(\mathbf{x})} \frac{\partial y(\mathbf{x})}{\partial neti} \frac{\partial neti}{\partial w_i}, \tag{3.18}$$

where *neti* is the net input to the perceptron. Hence the full derivative of the perceptron is

$$\frac{\partial E}{\partial y(\mathbf{x})} \frac{\partial y(\mathbf{x})}{\partial neti} \frac{\partial neti}{\partial w_i} = -(v_k - y_k) g'(neti) \phi(x_i) \tag{3.19}$$

Multiple layers

In the *multilayered perceptron* (MLP) the basis functions are no longer used, since the MLP can solve non linear problems [3]. An example of a multilayered perceptron with one output is presented in figure 3.3. The equation for this type of network becomes

$$y(\mathbf{x}) = g\left(\sum_{j=0}^{N} w_j \, g\left(\sum_{i=0}^{M} w_{ij} x_i\right) + w_0 b_0\right). \tag{3.20}$$

Equation 3.19 described how to find the derivatives for the weights w_j leading to the output layer. The derivatives for the weights w_{ij} leading to the hidden layer can be found with the backpropagation procedure. In order to derive Eq. 3.20 the chain rule is used again. The last term in Eq. 3.18 is exchanged to

$$\frac{\partial neti}{\partial w_i} = \frac{\partial neti^O}{\partial w_{ij}},$$

since E now shall be derived with respect to w_{ij}. The superscript indicates to which layer the corresponding node belongs(O=output, H=hidden). This term is now extended with the chain rule in the same manner as in Eq. 3.18

$$\frac{\partial neti^O}{\partial w_{ij}} = \frac{\partial neti^O}{\partial y(\mathbf{x})^H}\frac{\partial y(\mathbf{x})^H}{\partial neti^H}\frac{\partial neti^H}{\partial w_{ij}}. \tag{3.21}$$

Inserting Eq. 3.21 into Eq. 3.18 yields the full derivative of E with aspect to w_{ij}

$$\frac{\partial E(\mathbf{w})}{\partial w_{ij}} = \frac{\partial E}{\partial y(\mathbf{x})^O}\frac{\partial y(\mathbf{x})^O}{\partial neti^O}\frac{\partial neti^O}{\partial y(\mathbf{x})^H}\frac{\partial y(\mathbf{x})^H}{\partial neti^H}\frac{\partial neti^H}{\partial w_{ij}} =$$

$$-(v_k - y_k)g'(neti)^O w_j\, g'(neti)^H x_i \tag{3.22}$$

Input space representation

A critical issue for the performance of the neural network is the input space representation. Since a neural network consists of a fixed number of input nodes, the input space must be transformed into signals leading to these inputs. What here is called *Interval coding* is a representation that seems to work well for most discrete problems. In interval coding one uses as many input neurons as the total length of all the dimensions of the input space. In a 2-D problem with NxM states one would thus use N+M input neurons. The input nodes in each dimension respectively corresponding to the current state is set to one, while all the other are set to zero. Hence a representation of the state X=4 and Y=5 in a 2-D problem with 10x10 states, would set input node 4 and input node 15 to one and all the rest to zero.

For continuous state spaces this kind of representation does not work. One alternative representation is to use a gaussian distribution over the input nodes instead of setting the nodes to either zero or one. Here this is called a *Gaussian interval coding*. Combining different gaussians in this fashion can be seen as a special case of using mixture models [3].

3.1.5 Radial basis function networks

In 1988 Broomhead and Lowe [4] proposed a new kind of artificial neural network, the Radial Basis Function Network (*RBF Network)*. It would later gain both industrial and academic importance. One of the novelties was that it used the gaussian function as the basis function.

$$\phi(x) = e^{-\frac{\|x-\mu\|}{2\sigma^2}} \tag{3.23}$$

Assign points randomly into K sets

Do until no change in grouping

Compute mean vector of each set.

Assign points to nearest mean vector.

Table 3.1. K-means, batch training.

Here x is the input vector, μ the center of the basis functions and σ the width of the later. If different widths for the different input dimensions are used, σ will be a covariance matrix, Σ. The RBF network is built up from gaussian hidden nodes and linear output nodes.

Training procedure

RBF networks are normally trained using the Backpropagation algorithm, where the parameters μ and σ are regarded as constants. They may however be included in the backpropagation procedure as a part of the gradient descent search. If they are treated as constants another way of determining them is needed. To accomplish this a supervised training, using a training set, is often first performed as a separate part of the training procedure. One such supervised clustering algorithm is K-means [3], as in table 3.1, which determines K centers in the input space with an iterative procedure.

Chapter 4

Combining RL with ANN

In the two previous chapters the theory behind the reinforcement learning algorithms and function approximation was explained. It has been proven very advantageous to combine these two areas.

4.1 Gradient descent methods

The gradient descent theory was explained in chapter 3. It was assumed that the correct target output could be derived for the training. This is not the case for reinforcement learning problems using a function approximator, since an estimate of the true target then is used.

4.1.1 TD(λ)

The estimate used for TD(0) is $r_{t+1} + \gamma V(s_{t+1})$ as seen in Eq. 2.7. If the true target vector used in the error calculation in Eq. 3.13 is exchanged for the R_t^λ in Eq. 2.8, the weight update for the gradient descent form of TD(λ) becomes

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta(R_t^\lambda - V(s_t))\frac{\partial V(s_t)}{\partial \mathbf{w_t}}. \tag{4.1}$$

Convergence to a local optima can be guaranteed for this update if η decreases over time [25], and if the estimated target is an unbiased estimate of the true target. Unfortunately this is not the case for R_t^λ if $\lambda < 1$, and thus this method is not guaranteed to converge to a local optima. It is still useful to view Eq. 4.1 as a gradient descent method with a boot-strapping approximation of the true target, and these methods do have some very successful empirical results [27].

Equation 4.1 can be expressed in terms of eligibility traces presented in section 2.8 yielding

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta\delta_t \mathbf{e}_t, \tag{4.2}$$

where δ_t is the *TD-error*

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t), \tag{4.3}$$

and $\mathbf{e}_t$ is a vector of eligibility traces, one for each weight in $\mathbf{w}_t$,

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \frac{\partial V(s_t)}{\partial \mathbf{w_t}}, \tag{4.4}$$

with $e_0 = 0$.

4.1.2 SARSA(λ)

The extension of Eq. 4.1 and Eq. 4.2 to control problems is straightforward. Exchanging the value function estimate $V(s)$ for the state-action function estimate $Q(s, a)$ in Eq. 4.1 yields

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \eta(R_t^\lambda - Q(s_t, a_t))\frac{\partial Q(s_t, a_t)}{\partial \mathbf{w_t}}. \tag{4.5}$$

The TD-error δ in 4.3 becomes

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t), \tag{4.6}$$

and the eligibility traces $\mathbf{e}_t$ in Eq. 4.4 becomes

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \frac{\partial Q(s_t, a_t)}{\partial \mathbf{w_t}}, \tag{4.7}$$

The gradient descent SARSA(λ) converges in the same way that TD(λ) does.

Chapter 5

Pong, experiments and results

5.1 Background

Pong was one of the first computer games ever invented. It was produced by Nolan Bushnell, founder of Atari Inc, and when released in 1971 it was an immediate hit. The goal of the game is to manoeuvre a racquet, upwards or downwards, trying not let the ball end up outside the screen (see figure 5.1). The ball will bounce and change direction when hitting either one of the racquets or the upper or lower walls. The game is played by two players, and the one who first lets the ball pass by his racquet and out of the screen loses the game.

5.2 Implementation

In the versions of Pong implemented for this thesis, the players had a varying possibility, depending on the implementation, to play the game in a strategic manner. If, for example, player one receives the ball in the middle of his game side and perceives that player two is located in the upper right corner, it would in most cases be of much greater value to steer the ball towards the lower right corner than towards anywhere else. In the continuous version of the game the player can to quite some extent manoeuvre the ball's direction using the speed of the racquet as a steering tool. This will be explained further in the coming sections, where the two different game implementations will be discussed in detail. Throughout the experiments one separate agent controlled the left racquet and one the right.

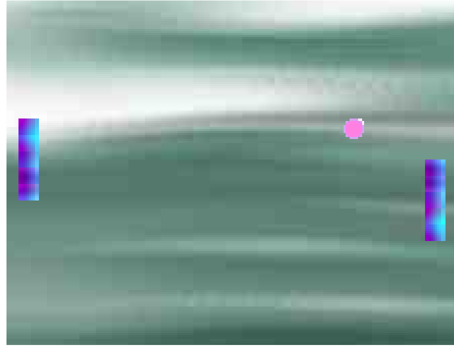


Figure 5.1. The Pong game.

5.2.1 Update schematics

In both versions implemented the update was made in the following order

- do until goal state reached
  - the current state is presented to the agents
  - agents return action and the speed of the racquets are updated
  - ball position and direction are updated
  - racquet positions are updated

5.2.2 Reward functions

All discrete versions and one continuous used the following reward function

$$R(s) = \begin{cases} \text{win}, & 1.0; \\ \text{lose}, & \text{-1.0}; \\ \text{else}, & 0.0. \end{cases}, \tag{5.1}$$

except the sigmoidal ANN with sigmoidal output neurons used with the discrete version. This version used the following reward function

$$R(s) = \begin{cases} \text{win}, & 1.0; \\ \text{lose}, & 0.0; \\ \text{else}, & 0.5 \end{cases}. \tag{5.2}$$

A few continuous experiments used the reward function

$$R(s) = \begin{cases} \text{win}, & 1.0; \\ \text{lose}, & \text{-1.0}; \\ \text{racquethit}, & 0.1 \\ \text{else}, & 0.0 \end{cases}. \tag{5.3}$$

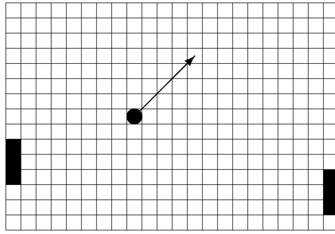


Figure 5.2. Difficult game situation.

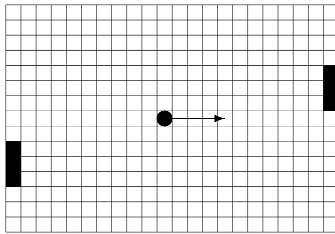


Figure 5.3. Simple game situation.

5.3 Extracting the results

Conducting experiments on an unstandardized application is not an easy task, particularly when the application has a large state space in many dimensions which is hard to visualize. Many of the results had to rely on human testing and evaluation. By observing the behavior of the agents produced by different learning algorithms, a distinction between different types of game situations came clear.

A difficult game situation was defined as one where the agent must take the correct action many times after another e.g. if the agent would have to move from one corner to the other to be able to catch the ball. This is exemplified in figure 5.2.

An easy situation on the other hand would be where the agent had already steered itself to a good position, and did not have to move much to be able to catch the ball. This is exemplified in figure 5.3.

Note that the equal distribution of random actions upwards and downwards usually leaves the position of the racquets unchanged over time.

5.4 The discrete version

This version of the Pong game was done as a step towards the larger and continuous version later implemented. The task turned out to be significantly more difficult than imagined, especially in the case of versions using a sigmoidal ANN.

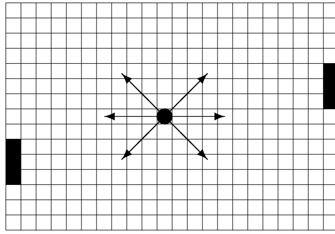


Figure 5.4. Discrete Pong layout.

The game itself was based on a 22 by 15 squared playing field with one racquet on each side of the field. The direction of the ball was discretized into 6 different directions: north west, west, south west, north east, east and south east, as shown in figure 5.4. Due to the fact that the ball would have a speed in the x direction of either 1 or -1 it is possible to play this version in a way which guarantees not losing, i.e. if the agent at every action selection would adjust its y position towards the y position of the ball. The racquets had length 4 and width 1, whereas the ball had a diameter of 1.

When the ball hit the south or north wall, the direction was mirrored, e.g. the ball hitting the north wall with direction north east, would after the collision go south east. When hitting a racquet, the calculation was a bit more complex: the direction of the racquet at the time of collision was also taken into account. If the ball was moving horizontally when hitting the racquet, and the racquet was also moving, in an arbitrary direction, the ball would be steered towards the direction of motion of the racquet. The table used to calculate the resultant direction after a collision with racquet one (left side) is given in table 5.1. The table used for racquet two was the same as table 5.1 but mirrored. When a racquet was hit by the ball this event would be registered and used in later evaluation. The term *racquet hit* will be used throughout this section as a term for this event. A completely random agent averaged approximately 0.75 racquet hits per game.

Input to the learning algorithms

The following parts of the environment were presented to the different learning algorithms: racquet one y position, racquet two y position, ball y position, ball x position, ball direction, action.

Input coding

When using the sigmoidal network versions, the input to the network was coded with an interval coding. Thus, the number of input neurons to the network were: 15 (racquet one y position) + 15 (racquet two

Ball dir. before collision	Racquet speed	Ball dir. after collision
north west [-1,-1]	up [1]	north east [1,-1]
north west [-1,-1]	none [0]	north east [1,-1]
north west [-1,-1]	down [-1]	east [1,0]
west [-1,0]	up [1]	north east [1,1]
west [-1,0]	none [0]	east [1,0]
west [-1,0]	down [-1]	south east [1,-1]
south west [-1,-1]	up [1]	east [1,0]
south west [-1,-1]	none [0]	south east [1,-1]
south west [-1,-1]	down [-1]	south east [1,-1]

Table 5.1. The resultant direction after a collision with racquet one

y position) + 15 (ball y position) + 25 (ball x position) + 6 (ball direction) + 3 (action) neurons. A version with only five input neurons, with each neuron taking the actual value of the state dimension was also tried, but unsuccessful. The action was coded with 3 nodes, one for each possible action. At each action selection the state together with the three different actions were compared.

Experiment setups

The following experiments were all performed using the discrete Pong version. The main purpose of the experiments was to examine the behavior of different types of learning algorithms together with different types of function approximators. First simple tabular algorithms were examined and then the more complex versions using ANNs as function approximators. As a final experiment the behavior of the tabular $SARSA(0)$ agents were compared to sigmoidal agents with linear output nodes trained by $SARSA(\lambda)$.

It was found that some degree of exploration was always needed to avoid the agents to getting stuck in a playing pattern. Hence, the ϵ-value was larger than zero during most of the experiments, except for during validation when observing the behavior the ϵ was always set to 0.

5.4.1 Discrete Pong - tabular $SARSA(0)$

This version used the $SARSA(\lambda)$ algorithm with λ set to zero. Hence, the value updates only smooth out the action value function as in 2.7, i.e. between two states at a time. The value function was stored in simple matrices. This version, was shown to be very robust in terms of its sensibility to the values of ϵ and γ. Storing the function values in a

matrix gives very fast access times and thereby faster execution speeds in comparison to versions using ANNs as function approximators.

However, the absence of any generalization mechanism was shown to give slower learning in terms of needed training games, i.e. a larger number of visits to each state was needed.

Observed learning behavior

One fertile way of judging the quality of learning was found to be that of observing the agents during learning; slowing down the speed of the games at fixed intervals. For the tabular SARSA(0) with ϵ set to 0.4 and σ to 0.99, the following observations were made.

After a few hundred games the first signs of improvement would be noticeable. When approximately 10 000 games had been played the tabular agents had usually learned to control the simplest games positions, e.g. the situation in figure 5.3. After 20 000 games the agents learned to handle some of the corner balls as in figure 5.2. It was then hard to notice any major difference until approximately 100 000 games had been played. At this time the agents had normally learned how to handle most of the situations that occur in the game. After this the agents usually got stuck in a playing pattern in which they would very seldom lose. Observing the agents at this time they seemed perfect, but when confronted with a human player several flaws could always be found. After 50 million games a very good player had been produced. After extensive human testing only one type of flaw could be found. Provoking the flaw was a three-step process involving first steering the ball so that the computer agent was located in the lower part of the playing field. Then the ball was steered upwards to go diagonally. The exact position of the computer player at the time of the last steering of the ball was a key factor, i.e. if the position was not exactly right, the tabular agent was able to hit the ball. This points out the tabular version's inability to generalize between states.

Progression of the learning process

The mean number of racquet hits per game was used as a simple measure of how the learning progressed. During the first 20 000 training games this was also found to be an adequate measure. After this, more sophisticated methods had to be used. Particulary the matrix version's agents tended to get stuck in movement patterns, and hence the mean racquet hits per game went up. Even though the mean racquet hit per game value does not qualify as a measure of the quality of the learned behavior after roughly 20 000 training games, it was found to give an estimate of how much the agents were changing their behavior. Figure

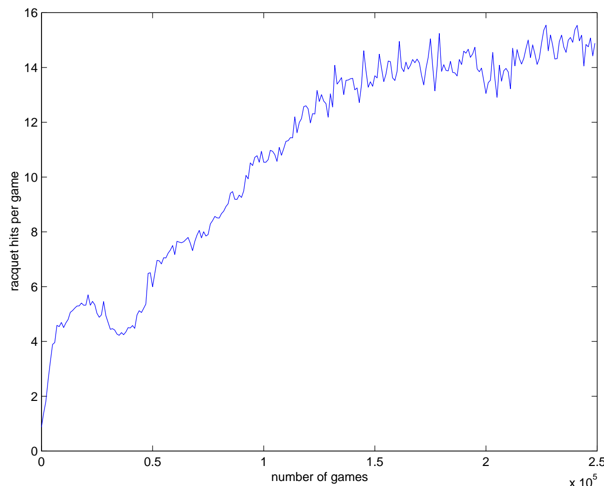


Figure 5.5. Mean racquet hits per game for tabular SARSA(0) with ϵ set to 0.4. The horizontal axis represents the number of games in thousands

5.5 shows the number of racquet hits per game for the first 500 000 games using an ϵ value of 0.4. It should also be pointed out that two matrix agents that have trained with these settings for more than 500 000 games average about 100 000 racquet hits per game when ϵ is set to 0.

The mean racquet hits per game value was calculated during 1000 games and was then reset.

5.4.2 Discrete Pong - $SARSA(\lambda)$ using an ANN

The last discrete version implemented was the $SARSA(\lambda)$ algorithm together with different types of ANNs. The ANNs used were:

- a sigmoidal network with one sigmoidal output neuron and 100 hidden sigmoidal neurons.
- an RBF network using sample data from random agents and a k-means algorithm which placed the center, μ, of the hidden layer nodes.
- a sigmoidal network with a linear output neuron and 100 sigmoidal hidden neurons.

Several different λ and γ settings were tested. The best performance was obtained with λ set to 0.9 and γ to 0.99 for the sigmoidal network with a linear output. The other two versions showed no great sensibility to these settings and the same values were finally used in these versions too.

5.4.3 RBF network

This version used an RBF network containing 500 hidden radial basis function nodes. The μ parameter, i.e. the center of the hidden nodes, was set by a k-means algorithm. The sample set used for this contained 10 000 game states, extracted from two random agents playing against each other. The σ value was equal in all dimensions and set as a function of distance to the nearest data point in the training set

$$\sigma \leftarrow d/5. \tag{5.4}$$

Here d is the distance to nearest data point in the training data set. The main difficulty concerning this version was the slowness of the network. A network with 500 hidden nodes was approximately as fast as the sigmoidal networks later studied but the performance was much worse. Most probably a version with more hidden nodes would have performed better but lack of computer power made such experiments impossible.

Observed learning behavior

Even though the RBF agents outperformed a random agent, the actual learning was hard to observe and not very successful. After a few hundred games the agents had often learned to master a difficult situation as in 5.2, and this behavior would often stay for the rest of the training process. The most of the learning took place during the first few thousands of games, and then no improved performance was observed. The number of racquet hits per game never went up as for the tabular $SARSA(0)$ or $SARSA(\lambda)$ using a sigmoidal ANN with linear output. This may be observed in fig 5.6. Most probably the bad performance of these agents was due to the limited resolution power of such a small network.

5.4.4 Sigmoidal ANN with sigmoidal output

Due to the nature of the sigmoidal transfer function, which goes from 0 to 1, the reward function for this version was redefined 5.2. If a tangential output neuron had been used 5.1 could have been used. A sigmoidal output was chosen because of its commonness in other research experiments. Having a reward of 0.5 at every non winning or losing state the values of γ and λ also have to be lowered to keep the target from not always being close to one. Therefore γ was set to 0.099 and λ to 0.09. The number of hidden nodes was set to 101.

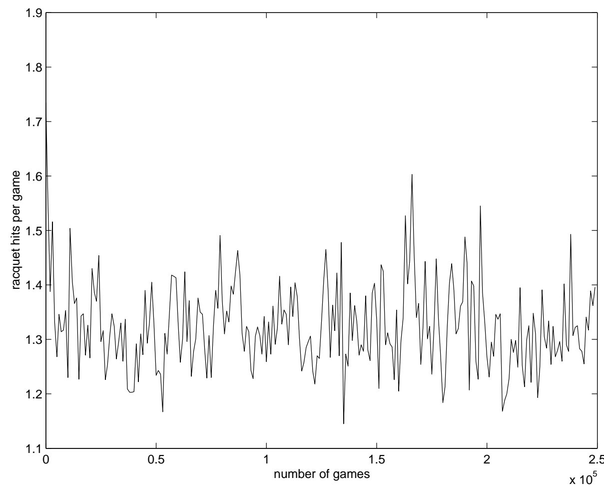


Figure 5.6. Mean racquet hits per game for $SARSA(\lambda)$ with a 500 hidden nodes RBF network and ϵ set to 0.4.

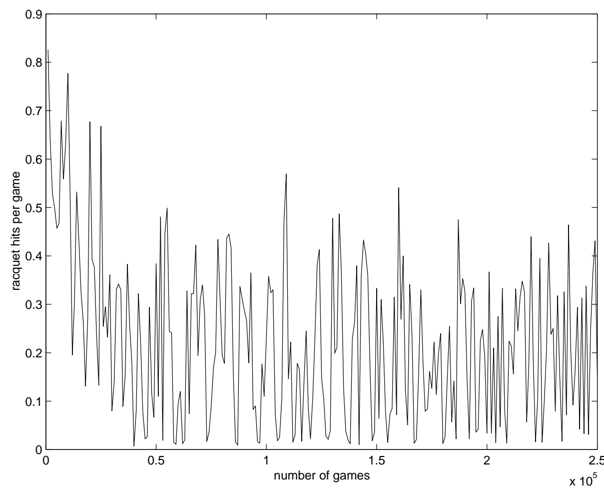


Figure 5.7. Mean racquet hits per game for $SARSA(\lambda)$ with a sigmoidal network with sigmoidal output and ϵ set to 0.4.

Observed learning behavior

These agents performed worst of all agents, and they got stuck in local optima after only a few hundred training games. The agents often learned to stand still in the middle or the top of the playing field and generally performed much worse than a random agent. Mean racquet hits for these agents can be found in fig. 5.7

5.4.5 Sigmoidal ANN with linear output

These sigmoidal ANN agents were found to be much more sensitive to the setting of the constants ϵ, γ and λ, than the tabular agents. When ϵ was set to a low value (e.g. 0.2) at least one of the playing agents got stuck in a local optima which often took many thousands of training games to get out of. With ϵ set to 0.1, no learning whatsoever was observed. Through empirical search an ϵ value of 0.4 was found to be a good compromise between exploring and exploiting. In all experiments performed this setting resulted in a stable convergence. Several experiments with a decreasing ϵ were performed but without any success. No decreasing function which performed well in all experiments was ever found. A decreasing function which worked well on one occasion did normally not work a second time. The best agents, in terms of fast convergence, were found when the ϵ value was manually set during training. Variations in the number of hidden neurons were also tested, and different experiments showed that a number around 100 gave the best results.

Observed learning behavior

As for the tabular SARSA(0), the behavior of the agents was observed at fixed intervals. With ϵ set to 0.4, γ to 0.99 and λ to 0.9, the following observations were made.

After about 1000 games the first noticeable signs of improvement could usually be observed. The learning was not as stable as for the matrix versions. A behavior learned early on could suddenly disappear and then later reappear after several thousands of training games. After 10 000 games the agents often seemed to follow the movements of the ball. However, the observed behavior was still not robust. Alternatingly one of the agents often won two thirds of the games for several thousands of training games. After 20 000 games at least one of the agents would play very well in some game situations. Unlike the matrix version it was not always the most common situations that were mastered at first. It was often observed that an agent missed a simple situation as in figure 5.3, and then performed well in a situation as in figure 5.2. When roughly 60-70 000 games had been played, the agents reached an equilibrium, where they won approximately the same number of training games. This may also be observed by looking at figure 5.8. The mean number of racquet hits per game would commonly rise from about 4 to 7. After
180 000 games the agents were through extensive human testing found to be unbeatable, i.e. no game situation was found in which the agent would miss the ball.

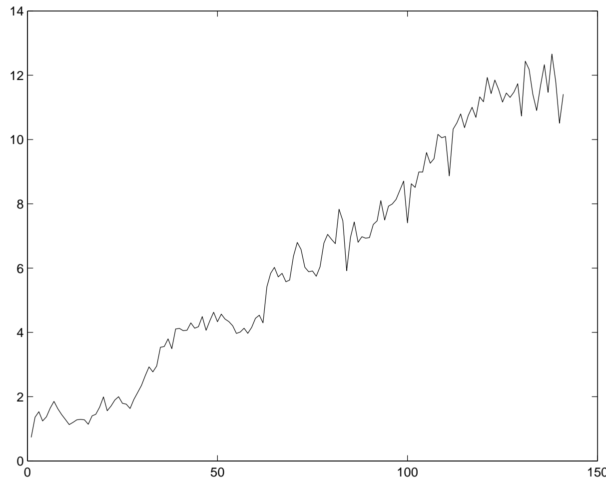


Figure 5.8. Mean racquet hits per game for $SARSA(\lambda)$ with a sigmoidal network with linear output neurons and ϵ set to 0.4.

The mean number of racquet hits per game was measured as for the tabular SARSA(0).

5.4.6 Comparisons between tabular $SARSA(0)$ and $SARSA(\lambda)$ with a sigmoidal ANN

When comparing the statistics of the two implementations it was found that the learning behavior was quite different in some aspects. The matrix version was not as sensitive to the settings of the constants as the ANN version, and it converged much faster to a stable solution where both players won an equal amount of games. This was found to lead to other problems. The tabular agents were much more inclined to get locked in a playing pattern. They adapted to the other agent's behavior and got stuck in a pattern in which no one lost. This pattern is by no means the perfect solution to the problem; when confronted with a new player with a different way of playing, it often lost. The ANN agents however, were much more unstable during learning. It took longer time for them to show any signs of improvement, and during learning a behavior learnt would often disappear for several thousands of games.

- The storage size of the ANNs used were exactly 75 times smaller than the matrices used.
- The ANNs showed low robustness. They were sensitive to the size of game variables and only functioned well for large λ and γ settings. When using a small ϵ (e.g. 0.2) the agents did not

converge in a reasonable amount of time. When using a decreasing ϵ they often did not converge at all. Manually changing the ϵ value during training resulted in the fastest convergence.

- A tabular player game took approximately one twentieth of the time an ANN game took.

Tournament system

As a tool for comparisons between the tabular agents and the ANN agents, a tournament system was implemented. The experiment was conducted in the following manner.

1. 3 pairs of tabular agents were instantiated, trained in pairs for n games and then stored.
2. 3 pairs of ANN agents were instantiated, trained in pairs for n games and then stored.
3. each left side tabular agent played a total of of 100 games against all right side ANN agents.
4. each left side ANN agent played a total of 100 games against all tabular right side agents.

The number of training games, n, used, was 10 000, 20 000, 30 000, 40 000 and 50 000. Figure 5.9 shows the percentage of tournament games won by the ANN agents.

5.5 The continuous version

The last version of the game that was implemented and examined had continuous state variables and a discrete action space. This approach was chosen because it was appropriate for the game, and applicable in many other arcade-like game applications. Using a discrete action space also simplifies the learning algorithms drastically and less computer power is needed. The size of the playing field, racquets and ball were the same as in the discrete version but now implemented as `double` variables instead of `integers`. The player controlled the game with a thrust toggle, choosing whether to accelerate up or down or to keep the current speed. If a player chose to accelerate upwards a constant was added to the current racquet speed, and if it chose to decelerate the constant was subtracted. The final version used a speed thrust constant of 0.1. The ball was initialized in the mid point of the

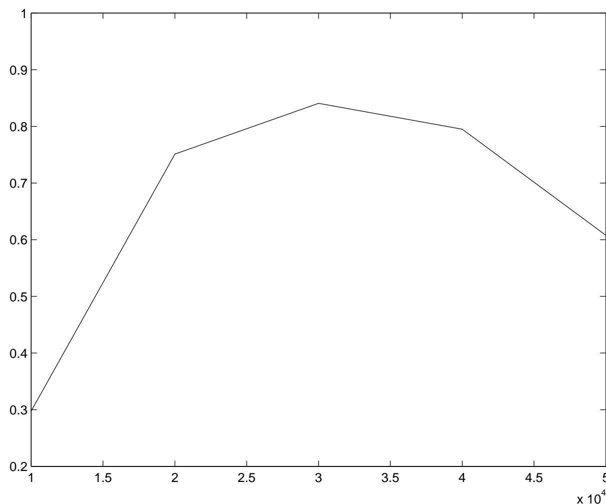


Figure 5.9. Percentage of games won by the $SARSA(\lambda)$ agent using a sigmoidal ANN with a linear output node when playing against a tabular $SARSA(0)$

playing field with straight right or straight left direction depending on which agent had won the previous game (the last winner would receive the first ball of the next game). The direction of the ball was represented with two vectors describing the speed in x and in y. The ball's x direction was initialized to one or minus one. The speed of the ball in the y direction was limited to be in [-1, 1]. The speed of the racquets was controlled in the same way.

When the ball collided with a racquet the new speed of the ball was calculated according to (Newton's second law)

$$ball.xspeed_{t+1} = -ball.xspeed_t \tag{5.5}$$

$$ball.yspeed_{t+1} = -ball.yspeed_t + racquet.yspeed_t \tag{5.6}$$

A completely random agent averaged approximately 0.7 racquet hits per game.

5.5.1 Input to the learning algorithm

The following parts of the environment were presented to the learning algorithm: Racquet one y position, Racquet two y position, Racquet one y speed, Racquet two y speed, Ball y position, Ball x position, Ball x speed, Ball y speed, Action. Each input dimension was coded separately using a gaussian interval coding. The center, i.e. the μ variable, was set in equidistant intervals for the specific dimension.

The width, i.e. the σ value, was set to be equal to the distance to the nearest neighbor node. As for the input nodes of the sigmoidal network, for the positional inputs as many input nodes as the max value of the dimension were used. For the inputs relating to speed 10 units were used per dimension. The action was coded with 3 nodes, one for each possible action. At each action selection the state together with the three different actions were compared.

5.5.2 Experiment setup

The following experiments were performed using the $SARSA(\lambda)$ algorithm together with a sigmoidal ANN with a linear output. The ϵ value was set 0.4 but in some experiments it was updated manually during the learning process. Several different numbers of hidden nodes were tried, ranging from 31 to 101. No drastic performance difference was ever observed and the final version used in the experiments below used a number of 71. Like the discrete version γ was set to 0.99 and λ to 0.95.

5.5.3 Observed behavior

The new and more complex game mechanics of this version implies that more training is needed to obtain a well performing agent. When using the reward function in 5.1, used for the discrete version, the agents showed signs of improvement after only a few hundred games. However, after a few thousands training games, at least one of the agents always got stuck in a local optima. No progress after 10 000 games was ever observed. After 100 000 games the performance would often deteriorate. This may be observed in figure 5.10. Normally only one agent's performance would deteriorate, but this would after a few hundred training games lead to a worse performance from the other agent too. After this no progress was observed if not the ϵ value was raised to a high value, e.g. 0.9. One might expect that extensive training with millions of training games would solve some of these problems, but unfortunately computer power limitations made such experiments impossible.

As a step towards speeding up the learning process a new reward function 5.3 was used. This included intermediate rewards when an agent would hit the ball with its racquet. When using this, learning progressed faster than with the other reward functions used. The agents would normally learn to master several difficult situations, as in fig. 5.2, after only a few thousand training games. However, since this reward function redefines the goal of the game, the agents learned

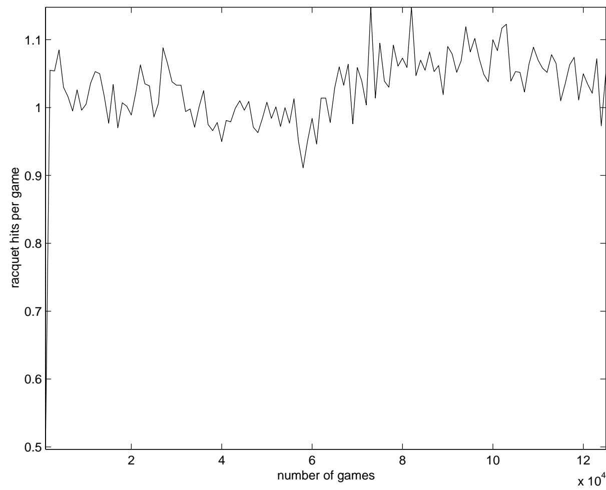


Figure 5.10. Racquet hits per game for a $SARSA(\lambda)$ agent with 71 hidden nodes, ϵ set to 0.4 using reward function 5.3.

the new definition. The new reward function rewards the agents more when they keep playing the ball back and forth many times than when they win and hence, after 10 000 training games the agents had normally gotten stuck in a pattern in which they never missed the ball.

Chapter 6

Conclusions and future work

The presented neural network theory seems to be the major limitation for a simple and powerful learning framework, but the results from the experiments performed on the discrete Pong version clearly shows the potential of temporal difference methods with function approximation in this domain.

6.1 Usage of RL algorithms on arcade-like games

The tabular $SARSA(0)$ produced a near-perfect agent after 50 million training games, and $SARSA(\lambda)$ with a sigmoidal ANN with linear output produced a perfect agent after only 180 000 training games. The computational power needed for the sigmoidal ANN agents was much greater than for the tabular ones. The discrete Pong application was approximately 10 times faster when using tabular agents than when using sigmoidal ANNs. The sigmoidal ANN was 75 times smaller than the matrix used by the tabular agent but the learning was much more sensitive to the settings of parameters. A structured procedure for setting these, e.g. genetic algorithms, might simplify and speed up the learning procedure. The version using an RBF network did not perform well in the experiments. Even for rather small numbers of hidden nodes it was much too computationally demanding. If one would use more hidden RBF nodes performance would probably be better.

As for the continuous version, only $SARSA(\lambda)$ with a sigmoidal ANN with a linear output was used. No well functioning agents were ever found but many of the results were promising. Also here the usage of genetic algorithms for setting the parameters could be of use. A multi player system, training many agents against each other could

also be of use. In this fashion many different strategies would be competing leading to a faster convergence. The use of a reward function, which gave intermediate rewards for hitting the ball, improved the performance of the agents during the initial stages of the learning process. However, in the end it would lead to a worse performance due to the intermediate rewards. These made it more profitable for the agents to keep the ball in play for a long time than winning the game. Experimenting with different reward functions, e.g. using intermediate rewards during the first phases of learning, and then removing or lowering them might lead to performance improvements.

A correct exploration rate turned out to be crucial for obtaining a good performance. Developing a robust framework for setting this parameter during learning is a possible future research area well worth investigating.

Bibliography

[1] Baird III L. C. and A. Klopf H. 1993, "Reinforcement learning with high-dimensional, continuous actions." emphTechnical Report WL-TR-93-1147, Wright-Patterson Air Force Base Ohio: Wright Laboratory, VA 22304-6145. 76, 83

[2] Bellman, R. E. 1957, "Dynamic Programming" *Princeton University Press, Princeton, NJ.*

[3] Bishop, C.M. 1995, "Neural networks for pattern recognition" *Oxford university press*

[4] Broomhead, D.S Lowe 1988, D. "Multivariable functional interpolation and adaptive networks" *Complex Syst., vol. 2, pp. 321–355, 1988*

[5] Coulom, M.R. 2002, "Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur" *Institut national polytechnique de Grenoble*

[6] Doya, K. 1996, "Temporal difference learning in continuous time and space" *Advances in neural information processing systems 8: 1073-1079*

[7] Duff, M. O. Bradtke S. J. 1995, "Reinforcement methods for continuous time markov decision problems" *Advances in neural information processing systems 7: 393-400*

[8] Gurney K. 1997, "An introduction to neural networks" *Routledge, London*

[9] Holland, J. H. 1986, "Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems" *Machine learning: An artificial intelligence approach, Volume II Los Altos, CA.*

[10] Hopfield, J.J. 1982, "Neural networks as physical systems with emergent collective computational abilities" *Proc. National Academy of Sciences of USA, vol.72, pp.2554-2558*

[11] Kaelbling L. P. Littman M. L. 1996, "Reinforcement Learning: A Survey" *Journal of Artifcial Intelligence Research 4:237-285*

[12] Markkula, G. 2004, "Playing risk aversive go on a large board using local neural network position evaluation functions" *Department of physical resource theory and complex systems group, Chalmers university of technology Göteborg*

[13] McCulloch, W.S. Pitts, W. 1943, "A logical calculus of the ideas immanent in nervous activity" *Bulletin of Mathematical Biophysics, 5:115 133*

[14] McKay D.J.C. "Introduction to Monte Carlo methods.", *http://www.inference.phy.cam.ac.uk/mackay/erice.pdf*

[15] Minsky, M.L Papert, S.A , "Perceptrons: Introduction to Computational Geometry" *MIT Press, Massachusetts, 1969.*

[16] Olmsted D.D. "History", *http://www.neurocomputing.org/History/body_history.html*

[17] Parker, D.B 1982, "Learning Logic" *Invention Report S81-64, File 1, Office of Technology Licensing, Stanford University*

[18] Pavlov, I.P. 1953, "Sämtliche Werke" *Zeller. Berlin*

[19] Rosenblatt, F. 1957, "The Perceptron: A perceiving and recognizing automaton" *Math. Stat., 22:400–407*

[20] Rumelhart, D.E. Hinton, G.E. Williams, R.J. "Learning internal representations by error propagation" *D.E. Rumelhart, J.L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, MIT Press, Cambridge, MA, pp. 318–362*

[21] Rummery, G. A. and Niranjan, M. 1994 "On-line q-learning using connectionist systems" *Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department*

[22] Samuel, A. L. 1959, "Some studies in machine learning using the game of checkers." *IBM Journal on Research and Development, 210-229*

[23] Schraudolf, N. N. Dayan, P. Sejnowski, T. 2001, "Learning to evaluate go positions via temporal difference methods." *Vol. 62 of Studies in fuzziness and soft computing. Springer Verlag*

[24] Shannon, C.E. 1948, "A mathematical theory of communication" *Bell Sys. Tech. Journal, vol. 27*

[25] Sutton, R. S. Barto, A. G. 1998, "Reinforcement learning: An introduction" *MA: MIT Press. Cambridge*

[26] Sutton, R. S. 1988, "Learning to predict by the methods of temporal differences." *Machine Learning 3: 1988, 9-44*

[27] Tesauro, G. 1992, "Practical issues in temporal difference learning." *Machine Learning 8, pp. 257-277*

[28] Thorndyke, I.P. 1911 "Animal Intelligence" *MacMillan Comp. New York*

[29] Thrun, S. B. 1992, "The role of exploration in learning control"

[30] Werbos, P. 1974, "Beyond regression: new tools for predictions and analysis in the behavioral science" *PhD Thesis, Harvard University*

[31] Widrow, B. Hoff, M.E. 1960, "Adaptive switching circuits." *IRE WESCON Convention Record, pt. 4, pp. 96-104*