TD Learning of
Game Evaluation Functions with
Hierarchical Neural Architectures

Marco A. Wiering[1]

Department of Computer Systems
Faculty of Mathematics and Computer Science
University of Amsterdam

April 7, 1995

[1]e-mail: wiering@fwi.uva.nl

Abstract

This Master's thesis describes the efficiency of temporal difference (TD) learning and the advantages of using modular neural network architectures for learning game evaluation functions. These modular architectures use a hierarchy of gating networks to divide the input space in subspaces for which expert networks are trained. This divide-and-conquer principle might be advantageous when learning game evaluation functions which contain discontinuities, and can also lead to more understandable solutions in which strategies can be identified and explored. We compare the following three modular architectures : the hierarchical mixtures of experts, the Meta-Pi network and the use of fixed symbolic rules.

In order to generate learning samples, we combine reinforcement learning with the temporal difference method. When training neural networks with these examples, it is possible to learn to play any desired game. An extension of normal back-propagation has been used, in which the sensitivities of neurons are adapted by a learning rule. We discuss how these neuron sensitivities can be used to learn discontinuous and smooth game evaluation functions.

Experiments with the games of tic-tac-toe and the endgame of backgammon have been performed to compare the hierarchical architectures with a single network and to validate the efficiency of TD learning. The results with tic-tac-toe show that modular architectures learn faster, because independent expert networks can be invoked for evaluating a particular position without the need of invoking one large neural network every time. Furthermore, the use of high neuron sensitivities has been proven to be useful when learning discontinuous functions. The results with the endgame of backgammon show that TD learning is a viable alternative for supervised learning when only a small learning set is available. For both games, the performance of the architectures is improved when more games are being played. High performance levels can be obtained when a large amount of games are played and the input of the networks contains sufficient features.

ii

Acknowledgements

Inspired by an article of G. Tesauro who had showed how practical neural networks and temporal difference learning can be when learning to play the game of backgammon, I started working on my master's thesis with the goal to study how powerful neural networks are when learning to play games.

This study was performed at the University of Amsterdam in the Intelligent Autonomous System Group headed by Prof. F.C.A. Groen. The supervisor of the research dr. ir. Ben Kröse taught me how real research has to be done, and directed me through the huge space of research questions which I had. I want to thank him in particular for all his helpful remarks and for not losing patience. In the beginning of the research, chaos had made her entrance. I want to express my gratitude to Gerard Schram who taught me that a good researcher never knows anything and that this is the way to acquire knowledge. At the time of his depart from the UvA, the research goals became more clear to me.

Furthermore, I am grateful to Patrick van der Smagt who not only knows a lot about neural networks, but also about using tools to be able to produce and show obtained results. Special acknowledgements are devoted to the other PhD students at the university: Anuj Dev and Joris van Dam for making the right remarks at the right time. I am also indebted to Sander Bosman who performed some tasks for me in a few seconds, while it would have taken much more efforts for me. Finally, I am really grateful to Jan Wortelboer. I have disturbed him many times to ask if it would be possible to get more computing power. I was happy that I never got no for an answer.

At home the situation was made inspiring by my two fiends René and René who helped me a lot when I was not able to think about something new. I hope that we will always continue having parties and talk about meta stories. These two freaks (because they are the freaks and not me) taught me how to live in the real world which I tended to forget once in a while.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Intelligent computer systems (**expert systems**) can be constructed for many different applications such as speech recognition, vision, robot control and game playing programs. The prevailing paradigm to construct expert systems was the use of knowledge engineering to translate human knowledge in a logical model which the computer can use. Although understandable and usable expert systems have been constructed, the knowledge acquisition bottleneck [1] taught the computer scientists that it would cost many man-years to create useful expert systems for complex tasks.

Nowadays machine learning techniques are used to trade off human time for computer time. Learning means relating inputs with outputs by the use of many examples. The use of **neural networks** provides a way to approximate any Borel measurable function [Cybenko89]. However, when learning the required knowledge for a real world application with a single neural network, the learning process takes a long time. Another problem with training single networks is that they can easily get caught in a local minimum. To improve such non-modular expert systems is very difficult, because the knowledge building blocks which are responsible for occasional mistakes can not be identified. A method to overcome these problems is to use a hierarchy of small single networks, in which each single network learns a simpler sub-function.

Expert systems for game-playing require an evaluation function which returns the expected payoff from a position given future optimal play of both sides. Some research studies learning game evaluation functions with neural networks [Tesauro92, Boyan92, Schraudol94]. Games provide domains where the evaluation function can differ drastically for similar positions (**tic-tac-toe**). When learning discontinuous functions, single neural networks tend to generalize over the discontinuities which results in small regions where the local error is very high. For other games (**backgammon**, checkers, chess, go), the

---

[1] The knowledge acquisition bottleneck is the problem of acquiring all necessary human expert knowledge. This has to be done by registering an expert's way of solving different problems, which is a time and money consuming process.

number of possible positions is very large and generalization can only be effective for the same type of positions. Therefore strategies have to be used to make it possible to learn different evaluation functions for positions which fall in different classes. In this way it becomes possible to accurately learn the game evaluation function, and to exploit the acquired knowledge optimally.

In this thesis we are interested in the maximal obtainable performance level of a computer agent which learns a model of simple game evaluation functions when **temporal difference (TD) learning** is combined with neural networks. Temporal difference learning provides a way to learn on examples which are acquired by playing games with an architecture. By using TD learning, we do not have to construct learning samples ourselves; continuously training the architecture by self-play will improve the approximation of the precise game evaluation function.

To solve the problems of having to represent discontinuities and storing large amount of knowledge in one single neural network, we propose to use multiple local neural networks over subregions of the input space. My master's thesis describes a study on three divide-and-conquer paradigms : the use of symbolic rules to divide the input space in fixed subspaces, and the **hierarchical mixtures of experts** [Jacobs91, Nowlan91, Jordan92, Jordan93] and the **Meta-Pi network** [Hampshire89] which use a hierarchy of gating networks to learn to divide the input space in subspaces for which expert networks are trained. These methodologies could be combined with temporal difference learning to learn context specific game-strategies without a priori knowledge. The validation of the methods will consist of two phases :

- The hierarchical neural network architectures will be compared with a simple monolithic network on their abilities to learn a simple function with one discontinuity.

- The games of tic-tac-toe and the endgame of backgammon will be used to compare the methods and to study the efficiency of temporal difference learning.

Game playing and some research to learn the game evaluation function of backgammon will be described in Chapter 2. In Chapter 3 we will give an overview of the principles of neural networks and modular network architectures. This is followed by a comparison on learning a simple discontinuous function with the different architectures. Then in Chapter 4, temporal difference learning of game evaluation functions will be described, and the different methods will be evaluated on the games of tic-tac-toe and the endgame of backgammon. In Chapter 5 we will conclude the research and describe the work which has to be done in the future.

Chapter 2

Game Playing

2.1 Introduction

Games define domains which are easy to represent and evaluate, while expert-level play may require sophisticated abilities of planning, pattern recognition and memory [Boyan92]. Computer game algorithms mostly use a position evaluator function which returns the expected payoff for a given position. To decide on a move in a given situation means comparing all possible positions resulting from the *current* admissible moves or comparing all positions which result from sequences of multiple moves. If only current legal moves are compared, the search tree consists of only one level with a branching factor b which equals the number of possible moves. If the terminals in such a small tree return good approximations of the payoff of the positions, then this is the fastest possible method to choose the best move in a position. If a look ahead strategy is used which builds a search tree of M levels, then the number of positions which have to be compared is b^M. This compares very unfavourably with the first method, but we do not have to make such high demands on the accuracy of the evaluation function. If the position evaluator function for a game is known accurately, then the game is said to be solved, and the game can be played by the fast first method.

Some complex games like draughts, go and chess are played by conventional programs which use symbolic rules to give a rough approximation of the evaluation function. Such programs use rigorous searching strategies where millions of positions must be evaluated before a reasonable solution is found. This is due to many discontinuities (or exceptions) in the evaluation function which are caused by many different combinations of contributions of pieces on the board. For such games we would have to represent all those discontinuities in our model of the evaluation function, which is very difficult and therefore rough approximations with symbolic rules are usually used. Of course this means that searching strategies have to be used which make heavy demands on the speed of the computer. When methods are found to construct more accurate models of the evaluation

functions, the speed of the computer would be less of a problem for playing games at expert level.

One method to construct a precise evaluation function is the use of dynamic programming. Dynamic programming computes and stores evaluations for every possible position in lookup tables, which is very expensive. We will use neural networks to learn an accurate model of the evaluation function, so that we can use the fast first method to play games. We will concentrate on two games: the game of tic-tac-toe and the game of backgammon, although experiments with the game of backgammon only consider the endgame. The game of tic-tac-toe has an evaluation function which contains many discontinuities, and the number of different positions is very low. To approximate its evaluation function means storing those discontinuities in our model, which might be very difficult. Backgammon (and especially the endgame of backgammon) on the contrary has a reasonable smooth evaluation function, because the players throw dice to determine their possible next moves. Because of this probabilistic nature, the branching factor is about 400. For this application it is therefore very important to have an accurate evaluation function. The game has a much larger state space than tic-tac-toe, and it also consists of many conflicting classes of positions. That is why human players use many strategies to play backgammon at expert level. E.g. when a player is well behind in the race, she must try to get more pieces hit and start a back-game, whereas in normal play she should prevent being hit.

2.2   Reinforcement Learning

There are two ways to learn to approximate the correct evaluation function for all possible positions of a game. It could be done by supervised learning on evaluations of positions given by a human expert. The problem is that the required amount of learning examples given by a human expert will become much too large and the evaluations are not very precise.

A better way is to use reinforcement learning in which examples are generated by the system itself. Reinforcement learning means playing games so that the current model of the evaluation function can be tested and refined. Reinforcement learning is attractive, because we only need to design the game-rules and a reinforcement learning module, which takes much less human effort than constructing a whole expert system to play the game. Samuel [Samuel59, Samuel67] was the first to construct a reinforcement learning system. He used a complex algorithm to select parameter adjustments based on the difference between the evaluations of successive positions occurring in a played game to learn the game of checkers.

Reinforcement learning is to construct a policy that maximizes future rewards and minimizes future punishments [Lin93]. We have to construct a learning procedure which effectively handles the temporal credit assignment problem. The temporal credit assign-

ment problem is to assign credit or blame to the experienced situations and actions, which makes it possible to create learning examples.

The most simple paradigm to create learning examples for neural networks is to convert the positions of a played game to input vectors and to use the final result of the game as the desired output for all constructed examples. This supervised method results in a loss of precision, because every move would be held equally responsible for the obtained result, which is almost never true. Especially when we want to learn to play games with stochastic elements, the learning process might become very slow.

Temporal difference (TD) learning [Sutton88, Tesauro92, Dayan92, Dayan94] provides an efficient method to receive learning examples with a higher accuracy, because the evaluation of a given position is adjusted by using the differences between its evaluation and the evaluations of successive positions. In this way the prediction of the result of the game in a particular position is related to the predictions of the following positions. Sutton defined a whole class of TD algorithms, TD(λ), which look at predictions of positions which are further ahead in the game, weighted exponentially less according to their distance by the parameter λ. Recently Dayan has proved that the TD algorithms converge with probability 1 when a linear representation of the input is used (e.g. lookup tables) [Dayan94]. A description of TD learning and an example of its use is given in Appendix A.

There are two advantages to learn the game of backgammon by the TD methods over the game of tic-tac-toe. The first is that a game of backgammon always results in maximal reinforcement (win or lose), the second is the use of dice to experience new positions [1]. To learn tic-tac-toe by the TD method could more easily end up in a local solution in which many positions have never been seen and we might need some exploration-strategy to find the optimal policy. However the game of backgammon is much more complex than the game of tic-tac-toe and together they can be used to evaluate the different learning paradigms which use multiple expert networks.

2.3   Backgammon

Backgammon is played by two persons upon a board designed with 24 points. Each player has 15 men and throws dice to move his men along the points until they reach their home tables from which they are moved from the board (figure 2.1). On the way, single standing men can get hit and they have to reenter in the opponents home table. The player first bearing all his men off is declared the winner. Backgammon depends somewhat on luck, but the player who makes the best moves will win over the long haul. Over the last fifteen

---

[1] The game of backgammon has stochastic elements, because dice are thrown to determine the possible moves in a position. This means that possible transitions to other positions depend on a probability distribution determined by the possible throws of dice.

years human experts have increased their skill by using mathematical rules and tables of
odds. This makes computing more important for playing the game well, which is of course
an advantage for the computer.
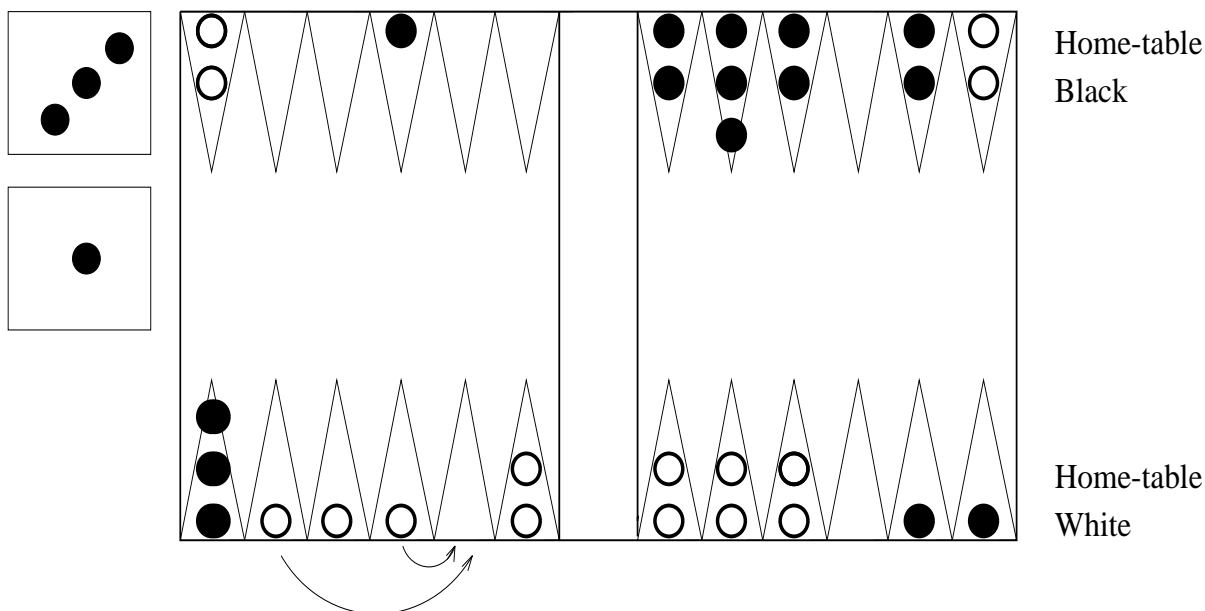
Figure 2.1: Backgammon is a two player zero-sum game in which the first player to bear
off all pieces has won the game. During the game both players try to block their opponent
by making points on fields so that the opponent can not play on or hit pieces from these
fields. Every time two dice are thrown to determine the possible moves. When equal dice
are thrown, the player can play four times the number of eyes on a dice. In the example
white plays a blocking move so that black's two pieces have problems going to black's
home-table.

BKG [Berliner77] was the first backgammon program. It was constructed during a
human knowledge engineering period of four years after which all important features of the
game were used in the reasoning process of the expert system. Many years later Tesauro
[Tesauro89] presented his Neurogammon program which was trained on a massive data
set of expert preferences of best moves in backgammon positions, but he realized that it
would be a better idea to enable a network to see millions of positions and learn from
the outcome of its own play. He adapted the TD method to implement TD-Gammon
[Tesauro92] and trained it first on the simplest case : disengaged bear-off (endgame)
and next he trained the network to learn the whole game. Tesauro created a simple
input encoding scheme which only contained a raw board description, but his monolithic
TD-trained network eventually surpassed the performance level of BKG and the network
trained on expert preferences. This was because the TD-network did not imitate a human

expert, so that it did not get itself into situations that it did not know how to handle. His conclusions were :

- Empirically the TD algorithm always converges to at least a local minimum.

- The quality of solution found by the network is usually fairly good and generally improves with increasing numbers of hidden units.

- Partitioning the game into a number of specialist networks may make it easier to learn the appropriate evaluation functions.

- An improved representation scheme might give substantially better results.

Tesauro's work was extended by [Boyan92] who used a priori knowledge to decompose the input space into subspaces for which independent expert networks were trained. Boyan used a Meta-Pi network [Hampshire89] to combine the evaluations of the trained experts so that a smoother evaluation function would be obtained. Results showed that the use of multiple experts outperformed a single network, although his task-decomposition was very simple and did not use human strategies. A more efficient decomposition would cost much more human engineering time, so we would like to have a methodology which can integrate learning the decomposition and the context-specific evaluation functions.

2.4 Multi-strategy Learning

Neural networks trained by back-propagation provide powerful inductive learning techniques, but when learning complex game evaluation functions with a single network, the global parameters in the network tend to smooth some important details. When we would use methods to divide the input space into less complex subspaces, independent local networks could learn the simpler evaluation functions much more accurately. This **multi-strategy learning** might be very useful to learn games where evaluations of similar positions may differ drastically (tic-tac-toe) or where strategies must be used to divide a complex evaluation function into simpler evaluation functions for a smaller input space (backgammon).

To make use of multi-strategy learning, we need to have some meta module which chooses specialists to evaluate a given position. The more subspaces we would create, the smaller the specialists would have to be and performing will become much faster (provided that each time only a few specialists are selected and the meta module performs fast). This is in particular important when we use searching strategies to improve one-ply evaluations. On the other hand, when there are too many specialists, they will not see many examples and generalization will not take place so that the learning process might become slower or end up in worse local minima.

Another choice we have to make when we want to use multiple experts is between competing or collaborating experts. Selecting one expert (competing experts) to evaluate an example is the fastest method, but then the evaluation function will not be smooth. If we allow data to lie simultaneously in multiple regions (collaborating experts), then the overall function will be much smoother, but because at all times multiple experts have to be invoked this will take more time.

We study three different methods to divide the input space and compare the results with the performance of a single network. The first method uses a a priori knowledge of the game-domain to divide the input space by fixed symbolic rules. This method is a fast way to select an expert network, but the linear regions cannot adapt themselves which results in a poor division when a priori knowledge is inaccurate. The two other paradigms make use of meta-modules which can adapt themselves by using given outputs of the chosen specialists. They use feed-forward neural networks and back-propagation to learn the division, which makes it possible to learn game-strategies without a priori knowledge.

Chapter 3

Function Approximation with Multiple Networks

3.1 Introduction

Neural networks can approximate any Borel measurable function until a specified level of precision [Cybenko89]. However, the problem is to find good parameters for the networks, which has to be done by learning on examples. Published successes in connectionist learning have been empirical results for very small networks, typically much less than 100 nodes. To fully exploit the expressive power of networks on complex real world tasks, they need to get larger and the amount of time to load the training data grows prohibitively [Judd90]. Nowadays, some researchers are focussing on modular architectures which consist of some small specialistic or expert networks which co-operate to learn the desired function [Jordan92, Nowlan91, Hampshire89, Fox91, Hashem93]. For many tasks, this results in more understandable systems which are easier to train, because each expert network learns a specific sub-function and experts can be analyzed and refined independently. Especially for discontinuous functions, the use of such a modular architecture can be advantageous, because otherwise the imbedded generalization of a neural network might smooth important details of the desired function.

Another way to make learning discontinuous functions easier is studied in which activation functions are made steeper, so that hidden units have less problems when approximating a discontinuity. In the experiments, activation functions are made steeper by introducing neuron sensitivity. When the neuron sensitivity is made very large, the slope of the activation function is very steep.

In this Chapter we will formally describe multi-layer feed-forward networks in section 3.2, and in section 3.3 three modular architectures which consist of multiple feed-forward networks will be described. In section 3.4 comparisons between using a single network and modular architectures are made by performing simulations on learning a simple dis-

continuous function.

3.2   Multi-layer Feed-forward Networks

Neural networks consist of many very simple processing units which communicate by sending numeric signals to each other over a large number of weighted connections [Krose92]. The knowledge is stored in the connections and processing units, which are adapted by a learning rule to minimize the error of the output units on the training data. A feed-forward network has a layered structure. Each layer consists of units which receive their input from units from a layer directly below and send their output to units in a layer directly above the unit (figure 3.1). We consider feed-forward networks with full connections between successive layers which minimizes human engineering time, however some researchers [Nadi91, Tresp93] are constructing constrained networks to bias and speed up the learning process. These specialized network topologies will almost always outperform the simple fully connected networks and are more easily to understand. This approach can be seen as an intermediate solution between knowledge engineering and machine learning, but can only be used for learning understandable tasks. One hidden layer with enough

Output Layer

Hidden Layer
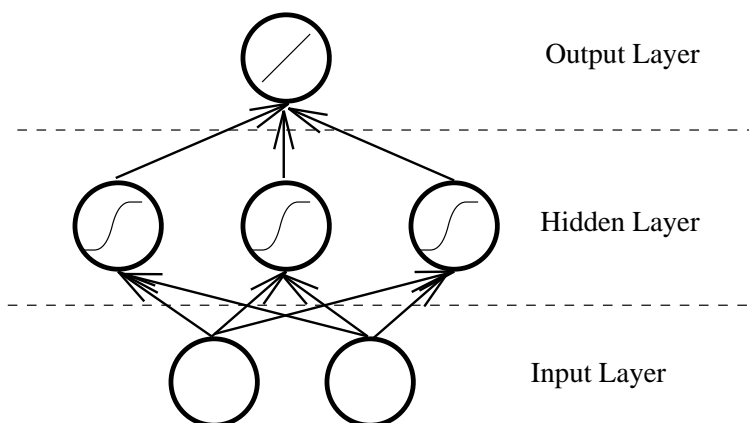
Input Layer

Figure 3.1: a fully connected two-layer feed-forward network, the input layer does not contain processing units and is not counted as a 'processing' layer.

units and non-linear activation functions is sufficient to approximate any Borel measurable function [Cybenko89] so all constructed networks in this paper will have at most one hidden layer.

3.2.1   Forward Propagation

The first step in using a neural network is to encode the problem in an input to output vector mapping and to choose the number of hidden units. When we have constructed

the proper network, we can assign the input vector $\vec{a}$ of an example to the input layer and propagate these inputs to a higher level by computing activations of the units in the hidden layer. The input i_i of a hidden unit is computed by taking the weighted sum over the input

$$i_i = \sum_j w_{ij} a_j + b_i$$

With :

w_{ij} : the strength of the connection between the jth input unit and the ith hidden unit.
b_i : the bias of the ith hidden unit. It can be considered as the strength of a connection from a unit with constant activation 1.

The activation a_i of a unit in the hidden layer is computed by using a non-linear activation function F_i

$$a_i = F_i(i_i)$$

In this paper the sigmoidal function will be used for the hidden units. The sigmoidal activation function is called a basis function and in this research it is given by

$$F_i(x) = \frac{1.01}{1 + e^{-\beta_i x}}$$

β_i is the neuron sensitivity of the ith hidden unit to the input and can be chosen very large to make the activation function of the hidden units very steep (see figure 3.2). In this manner hidden units only change their incoming weights on a specific part of the input, for other inputs weight changes will be zero. When we make the activation functions very steep, the basis functions are made local.

[Sperduti92] has found a learning rule which learns neuron sensitivities (see Appendix B), so that the learning process becomes faster and units have different learning rates. This learning rule will be evaluated in section 3.4 and in Chapter 4 by its ability to adjust the neuron sensitivities for learning game evaluation functions.

When all activations of the units in the hidden layer are known, we can compute the activations s_i of the output units

$$s_i = F_i(i_i) = F_i(\sum_j w_{ij} a_j + b_i) \tag{3.1}$$

F_i can be a sigmoid, but in this research a linear activation function will be used. $\vec{s}$ is the output of the network on a particular input, and w_{ij} is the weight from the jth hidden unit to the ith output unit. When the desired output for an example is known, we can adapt all weights so that the next time the error on this example will be smaller. For this we can use the back-propagation [Rumelhart86] algorithm.
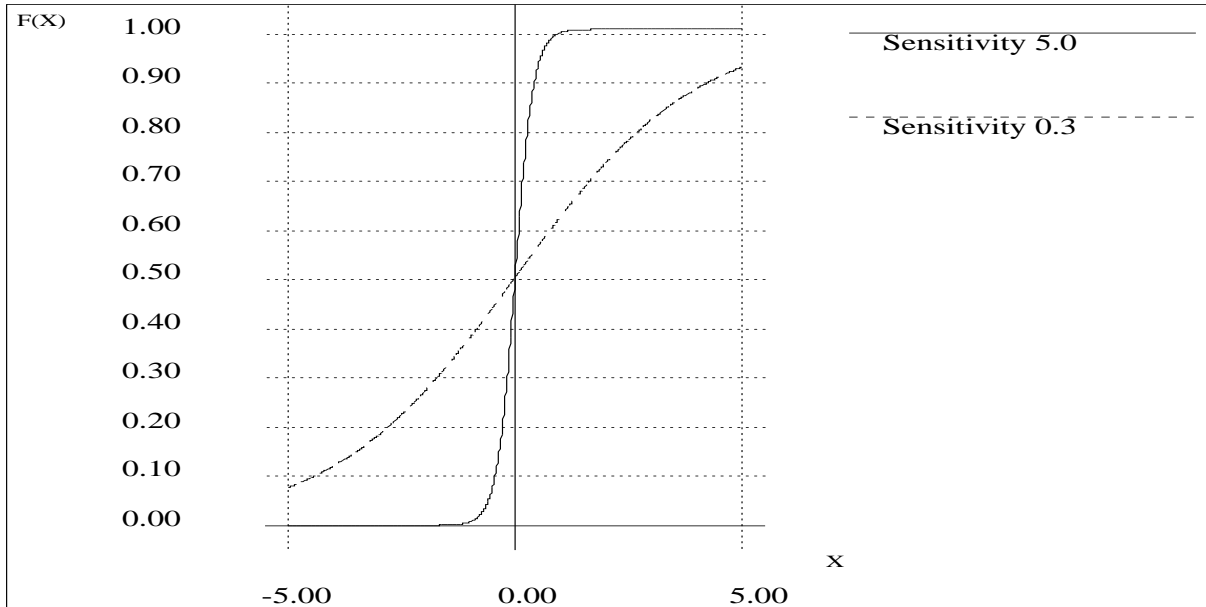
Figure 3.2: The sigmoidal activation function with high and low neuron sensitivities.

3.2.2 Back-propagating the Error

The network designer has to choose an error function, usually the error measure E is defined as the quadratic error for an example with target output vector $\vec{d}$ at the output units

$$E = \frac{1}{2} \sum_i (d_i - s_i)^2$$

By using gradient descent we can change the weights including the bias. First we can write

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial i_i} \frac{\partial i_i}{\partial w_{ij}}$$

And the weights update rule with learning rate γ controls a gradient descent on the error surface. It is given by

$$\Delta w_{ij} = -\gamma \frac{\partial E}{\partial w_{ij}} = \gamma \delta_i a_j \tag{3.2}$$

where in the case of an output unit δ_i can be computed by

$$\delta_i = -\frac{\partial E}{\partial i_i} = (d_i - s_i) F_i'(i_i) \tag{3.3}$$

and in the case of a hidden unit δ_i can be computed by the back-propagation chain rule

$$\delta_i = F_i'(i_i) \sum_h \delta_h w_{hi} \tag{3.4}$$

Sometimes a momentum term is used to speed up the learning process [1]. The momentum term uses the last weights alteration to direct the new update step. The weights update rule with momentum μ at time t is

$$\Delta w_{ij}^t = \gamma \delta_i a_j + \mu \Delta w_{ij}^{t-1} \tag{3.5}$$

3.2.3 The State Space of a Neural Network

The state space is defined by all possible states of a predefined fixed architecture. Learning can be considered as a search in this state space with the aim to find the state w which minimizes the error function $E(w)$ given by

$$E(w) = \int_X |g_w(x) - f(x)|^2 dx$$

with :
X : the input space or all possible inputs.
g_w : $\Re^i$ x W $\mapsto$ $\Re^o$. The function approximation for a given architecture when it is in state w.
f : $\Re^i \mapsto \Re^o$. The desired function.

Usually the error $E(w)$ is computed over a small subset of the possible inputs, which is called the test set. The best approximation w^* depends on the chosen architecture which defines all possible states of the network. When the state space becomes much larger, it takes more time to find the best state. This has consequences for learning : we have to have an architecture which contains a 'good' solution, but we do not want to have too many superfluous states. Some researchers use pruning [Esposito93], and others make it possible for the architecture to grow [Schaffer92, Gruau92, Simon92]. These methods can be used to find an architecture which contains at least one 'good' state, but a minimum of superfluous states.

When we want to learn a complex task, we can expect that the input space for this task is very large and we must use a representation with a large expressive power. The expressive power or VC dimension of a neural network depends mostly on the number of weights [Anthony91], and when we have enough examples and hidden units in the network we are able to learn most functions until a specified level of precision [Vysniausk93]. This means that the architecture contains one or more 'good' states.

When we want to analyze a neural network, we must understand its state space. In neural network literature, it is usual to talk about weight spaces instead of state spaces, because the weights are the parameters which are adjusted. The dimension of the weight

---

[1]The use of a momentum term is usually very effective when off-line or batch learning is used. For on-line learning, the use of a momentum term is much less necessary.

space depends on the number of weights and because weights are continuous parameters, the weight space is usually very large. Finding the best weight-setting has to be done by searching in all directions in this space.
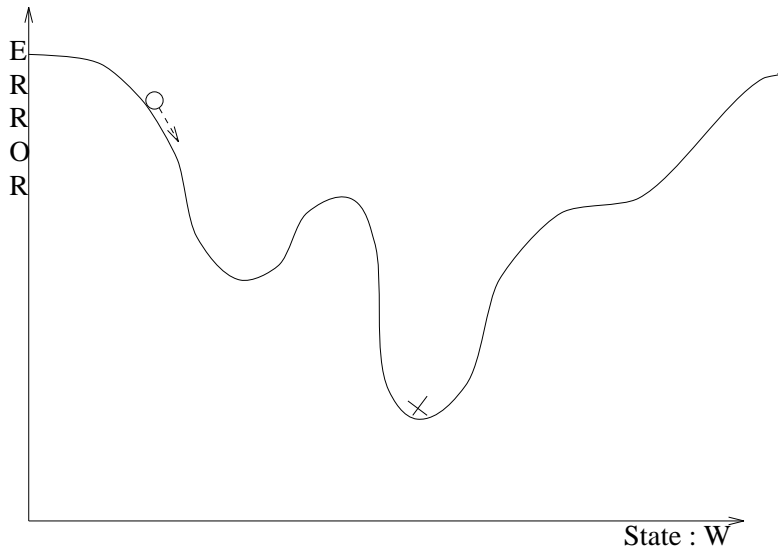
Figure 3.3: Back-propagation learning is a gradient descent method : in a given iteration the search direction is given by the negative gradient of the error, the step along this direction is given by the learning rate. The problem is to find the global minimum (indicated by the cross) in a space which might contain many attractors.

Back-propagation follows computed gradients to minimize the error-function $E(w)$ (figure 3.3). When we use off-line back-propagation, we try to minimize the sum of the error over all learning examples and we can easily be trapped in a local minimum, because we mostly follow the same direction. When we are trapped, this means that there will never be a transition to a state outside of the current local attractor. With stochastic or on-line back-propagation, the weights of a network are immediately changed after each learning example. This results in faster learning and less problems with local minima, but it can still happen that the network gets trapped in a local minimum from which it is not probable that the network will get out.

When we are often trapped in local minima we should use other minimization procedures e.g. simulated annealing [Aarts89], but these are often significantly slower than conventional back-propagation.

3.2.4   Representing Discontinuous Functions

When learning functions which might contain many discontinuities, high accuracy can only be obtained when we use an architecture which can represent these discontinuities.

Single networks with global nondecreasing activation functions (e.g. sigmoids) use superpositions of functions which all have a particular value and derivative at each discontinuity, and when some values are slowly varying around such critical points, we can expect the function approximation of the network to be too smooth when fast jumps are required. Of course this means that we have to use many hidden units from which many have to be located around the discontinuities to approximate them with steep slopes. The rest of the hidden units have to be used to represent the smooth details and to wipe out the effects of the hidden units which are representing the discontinuities.

A technique to circumvent the problem of representing discontinuities is to use multiple local models which perform in regions which are bounded by the discontinuities (figure 3.4). A priori knowledge of where the discontinuities are is often not at hand, so we would like to use an algorithm which learns to place neural networks in each subdivision of the input space which does not contain any discontinuities.
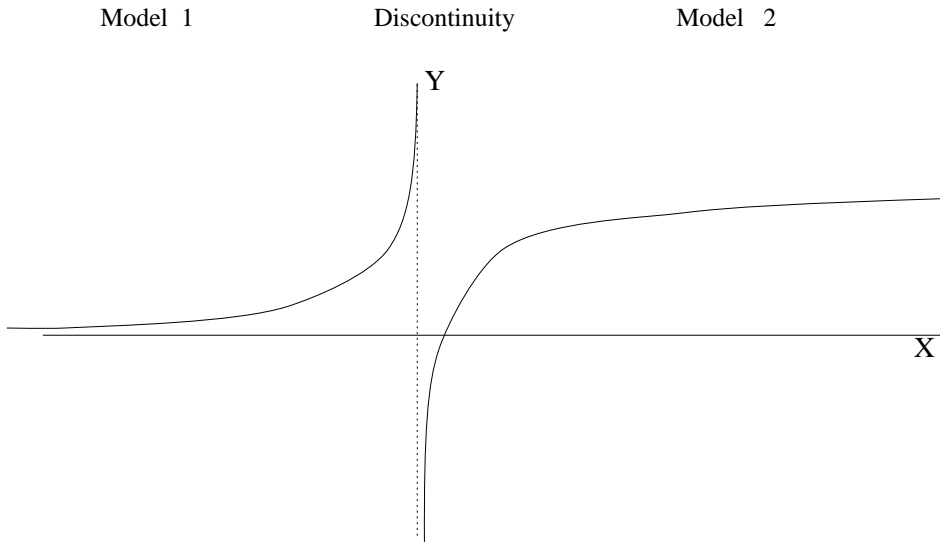
Figure 3.4: Representing a discontinuous function by using two local models. In this way the problem of representing the discontinuity is circumvented.

3.3 Hierarchical Network Architectures

We have shown a possible problem when learning discontinuous functions with a single network : the imbedded generalization will smooth important details of the desired function. Using multiple networks might overcome this problem, but we must find ways to integrate all networks in a learning architecture and decompose the function into smooth sub-functions.

Although there are many ways to integrate different learning modules in an architecture, we will focus here on modular architectures which consist of a number of expert feed-forward networks, where each expert learns a specific part of the desired function and a number of gating feed-forward networks which learn to divide the input space into a number of subregions corresponding to the number of experts. The gating networks use an output for each expert or cluster of experts. This output is used to determine the responsibility of the expert or cluster. By monitoring the competition between the experts or clusters, learning rules are defined for the gating networks so that these responsibilities over the inputs are adapted.

These architectures can use arbitrary hierarchies (figure 3.5) in which each propagate node uses the output vectors of a gating network to propagate the output vectors of the experts to a higher level. The highest-level propagate node will give the final output of the architecture.

The architectures which will be described in this paper can learn to map an input vector of any dimension onto an output vector of dimension one, but a generalization to larger output vectors is straightforward. All gating and expert networks in the architecture receive the same input, although this is not a necessity. We will proceed by summarizing some of the work done in this field, starting with the hierarchical mixtures of experts (HME) methodology [Nowlan91, Jacobs91, Jordan92, Jordan93] in section 3.3.1. This description is followed by the second methodology by Hampshire and Waibel [Hampshire89] in section 3.3.2, which is based upon the Meta-Pi network. The use of a selection threshold to make the propagation of the architectures faster is described in section 3.3.3. The differences between the two architectures will be depicted in section 3.3.4. A third architecture which uses knowledge bases containing fixed symbolic rules as gating networks will be described in section 3.3.4.

3.3.1   Hierarchical Mixtures of Experts

[Jacobs91, Jordan92, Nowlan91] developed a modular gating architecture which can learn to divide the input space in subspaces. Their architecture consists of a number of expert feed-forward networks which receive the same input patterns and compete with each other to produce the desired output vectors. The outputs of the gating networks are used by the propagate nodes to propagate the outputs of the experts to the top-level of the tree.

Gating Networks

In the following a hierarchy which consists of two hierarchical levels (Figure 3.5) will be considered, but one can transform the given learning rules to arbitrary hierarchies.

The system works as follows. The gating networks are linear neural networks and are used to blend the outputs of the experts. First, the outputs s_i and s_{ij} of the gating
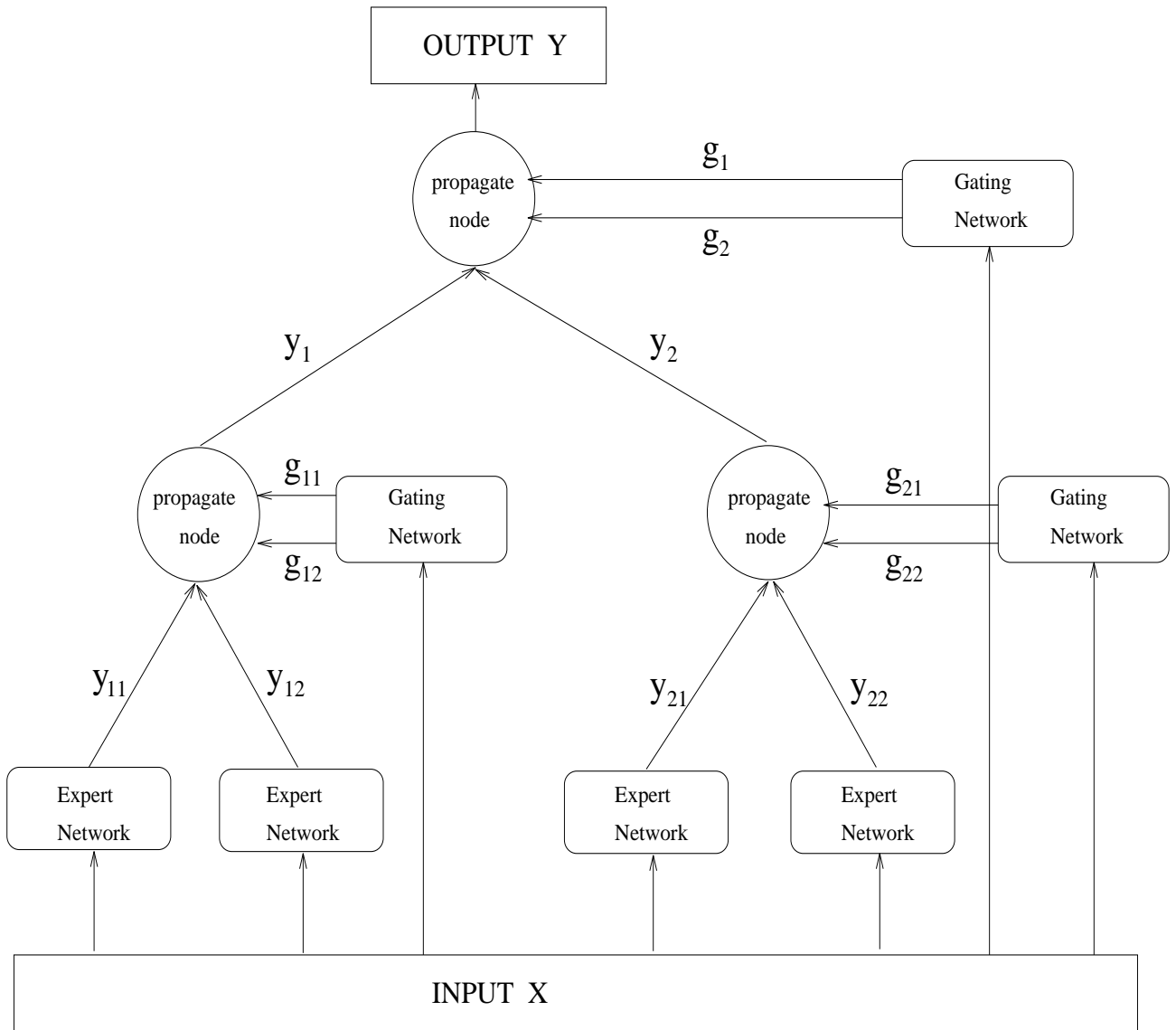
Figure 3.5: Two hierarchical levels of adaptive experts. All networks receive the same input.

networks are normalized to give gate values g_i and g_{ij}.

$$g_i = \frac{e^{s_i}}{\sum_k e^{s_k}}$$ (3.6)

and

$$g_{ij} = \frac{e^{s_{ij}}}{\sum_k e^{s_{ik}}}$$ (3.7)

When the gates are almost equal, a partitioning has not been made for this input vector. When we start training the gating networks, we can make all weights equal which expresses the fact that we do not know how to partition the input space in advance. When the gates begin to diversify, so are the experts and the part of the input space which is shared about equally by multiple experts is becoming smaller.

Propagate Nodes

The propagate nodes use the gates and the outputs of the experts and clusters to propagate the output vectors to higher levels. Different strategies for the propagate nodes make three different ways of using the hierarchical mixtures of experts methodology.

- Mixing the outputs

  The first way is to combine the outputs of all experts, for most functions this makes sense, because then we use more than one random source. This can be considered as the soft approach and will result in a smooth overall function. The output of the ith propagate node or cluster is given by

  $$y_i = \sum_j g_{ij} y_{ij}$$

  where y_{ij} is the output of the jth expert in the ith cluster. The output of the architecture is given by

  $$y = \sum_i g_i y_i$$

  In Chapter 4 this strategy will be used to learn evaluation functions for positions in a game with the HME architecture.

- Winner takes all

  The second way is to start at the top-level gating network and to use the output vector of the cluster with the maximal g_i

  $$y = y_k \quad with \ k = argmax_i\{g_i\}$$

When the highest level propagate node has chosen a cluster, the output of this cluster is the same as the output of the expert with the largest g_{kj} value

$$y_k = y_{kl} \quad with \ l = argmax_j \{g_{kj}\}$$

This winner takes all approach makes sense for learning discontinuous functions, because then we will never use output vectors of experts which are trained at the wrong side of a discontinuity.

- Stochastic choice

  The third way is to switch to a stochastic model in which the gate values give the a priori probabilities of selecting the ith cluster and $(i,j)^{th}$ expert to produce the output vector
  $$P(y = y_i) = g_i$$
  and
  $$P(y_i = y_{ij}) = g_{ij}$$

  This is the original way of interpreting (but not using) the methodology [Nowlan91]. The propagate node acts like a multiple input, single output stochastic switch. This approach makes sense for learning games, because we must avoid making the same repetition of moves and this provides us a way to explore novel states. However, if we have provided an exploration rule, we might better use the mixing the outputs strategy, because otherwise one expert could easily dominate over another. This means that we could end up with 'dead' experts, and many parameters in the architecture will never be used [2]. In this research we will not use the stochastic choice method.

HME as a Probabilistic Model

For understanding the learning algorithm, we must first give the hierarchy a probabilistic interpretation. For this the propagate nodes act like the single stochastic switches as described above. The gate values g_i and g_{ij} determine the a priori probabilities $P(y_i | \vec{x})$ and $P(y_{ij} | \vec{x})$ that the first and second level propagate nodes decide to select the output y_i of the ith cluster and the output y_{ij} of the $(i,j)^{th}$ expert. They are a priori probabilities, because they are obtained without using the target output d. The probability that the desired output d is generated when the input vector $\vec{x}$ is given is

$$P(d | \vec{x}; \Theta) = \sum_i P(y_i | \vec{x}) \sum_j P(y_{ij} | \vec{x}) P_{ij}(d | \vec{x}; y_{ij}(\Theta_{ij}, x))$$

---

[2]When learning to play games, the same argument is also applicable when we would use the winner takes all approach.

Where Θ refers to all parameters in the architecture and Θ_{ij} refers to the parameters in the $(i,j)^{th}$ expert network. The formula corresponds to a form of Bayes' rule and in our case it can be rewritten as

$$P(d|\vec{x};\Theta) = \sum_i g_i \sum_j g_{ij} P_{ij}(d|\vec{x};\Theta_{ij})$$

The second sum represents the probability that the ith cluster will produce the desired output when given the input vector $\vec{x}$. $P_{ij}(d|\vec{x};\Theta_{ij})$ represent the probability that the $(i,j)^{th}$ expert produces the desired output when presented with input $\vec{x}$. P_{ij} is the probability density function for the $(i,j)^{th}$ expert. When we are using a Gaussian density function for modeling a normal distribution, this results in

$$P(d|\vec{x};\Theta) = \sum_i g_i \sum_j g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}$$

The posterior probability h_i is a better estimate for the a priori probability g_i, because the errors of the experts are used. It is computed by using Bayes' rule

$$h_i = \frac{g_i \sum_j g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}}{\sum_i g_i \sum_j g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}}$$

We can also define the posterior probability that an expert has to be selected in each cluster by

$$h_{ij} = \frac{g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}}{\sum_j g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}}$$

Gradient Ascent on a Log Likelihood Function

We will treat $P(d|x;\Theta)$ as a likelihood function in the unknown parameters Θ. A learning algorithm is now developed by using gradient ascent to maximize the log likelihood function given by

$$\mathrm{L}(\Theta,x) = \ln \sum_i g_i \sum_j g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}$$

The expert networks have to learn the examples weighted by their joint posterior probability $h_i h_{ij}$. We use the posterior probabilities to weight the errors of the experts so that the most responsible expert will learn to specialize on examples it already approximates better than the other experts. The partial derivative of the log likelihood with respect to the output of the $(i,j)^{th}$ expert network is

$$\frac{\partial L(\Theta,x)}{\partial y_{ij}} = h_i h_{ij}(d - y_{ij}) \tag{3.8}$$

This gradient can be filled in for δ_i in equation 3.2, and back-propagation will fit experts in regions where $h_i h_{ij}$ is high. Finally, the partial derivatives of the log likelihood function with respect to the output units of the gating networks are

$$\frac{\partial L(\Theta, x)}{\partial s_i} = h_i - g_i \qquad (3.9)$$

and

$$\frac{\partial L(\Theta, x)}{\partial s_{ij}} = h_i(h_{ij} - g_{ij}) \qquad (3.10)$$

The two gradients 3.9 and 3.10 can be filled in for δ_i in equation 3.2 and the weights update rule will shift experts to regions where they outperform other experts.

Discussion

The competition between the experts is controlled by the gating feed-forward networks which use a mixture of Gaussian distributions to model the performance of the experts on the learning examples. Learning is achieved by gradient ascent in the log likelihood of generating the learning examples. This means that when given an input-output pattern, the gating networks learn which expert is to be chosen for maximizing the probability that the desired output will be given. If a smooth approach is used (the mixing the outputs strategy of the propagate nodes), then examples fall in multiple regions (figure 3.6), and for every example all experts have to learn to reduce their error in proportion to their joint posterior probability.

The HME architecture is especially suited when there is an obvious division of the input space in subspaces, and for such domains faster learning is usually achieved. When a good division has been learned by the gating networks (e.g. dividing the function at each discontinuity), the expert networks will have less problems with learning the sub-functions than a single network would have when it tries to learn the whole function including all discontinuities.

3.3.2   The Meta-Pi Network Architecture

The mixture of experts methodology uses soft adaption and competition between the experts, so that all experts can be used to learn a part of the desired function. The Meta-Pi network architecture as presented by [Hampshire89] has the same structure as given in figure 3.5, but instead of competing networks they use co-operating networks. In contrast to the previous section where experts are made more responsible in regions where they have a smaller error than the other experts, the Meta-Pi gating network learns to make experts more responsible in regions where they can be used to minimize the error of the architecture as a whole. For a more formal description we will again refer to the two-level hierarchy shown in figure 3.5.
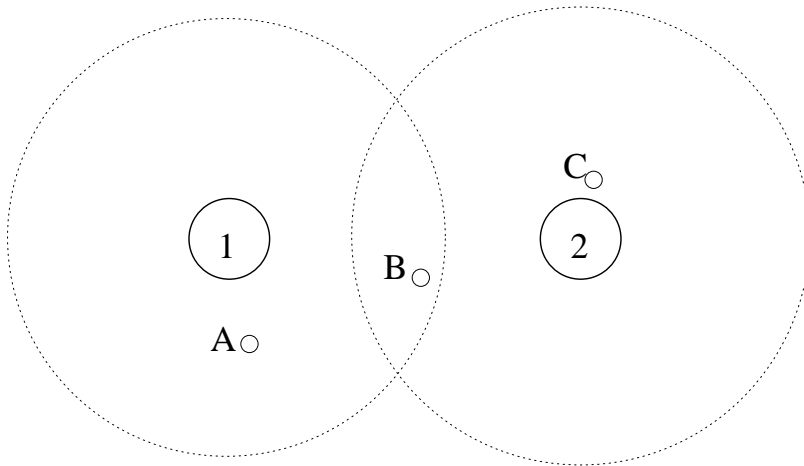
Figure 3.6: Mixing gaussians to decompose the input space into regions where experts 1 and 2 are to be located. The expert which is closest to an example has the largest responsibility for generating the output of the architecture. When presented with example A, expert 1 has the largest a priori probability to generate the desired output and will be used more than expert 2. The two experts will compete on the rights to generate the output for example B. The gating network will shift the expert which has the largest posterior probability to generate the desired output for example B towards the input vector of B. This posterior probability depends on the error both experts make when presented with the learning example B.

Gating Networks

The Meta-Pi gating network uses non-linear activation functions for the output units which are strict positive e.g. RBFs [Hakala94], or sigmoids can be used. The gates of the top-level gating network are given by

$$g_i = \frac{s_i}{\sum_j s_j} \quad , s_i > 0 \tag{3.11}$$

where the activation of each output unit s_i is computed by equation 3.1. Gates with other identifiers can be computed by this equation for the other gating networks.

Propagate Nodes

The propagate nodes have to use the experts in a co-operative way. So mixing the outputs is the only strategy, for this weighted sum is minimized by the learning procedure. The output of the architecture is

$$y = \sum_i y_i g_i$$

Where y_i are the outputs of the propagate nodes in the second level of the hierarchy. They are computed by

$$y_i = \sum_j y_{ij} g_{ij}$$

y_{ij} is the output of the $(i, j)^{th}$ expert network.

Gradient Descent on a Squared Error Cost-function

Until now we can see many similarities with the mixtures of experts method, but the Meta-Pi network uses a squared error cost-function which is to be minimized by gradient descent instead of a likelihood function which is to maximized by gradient ascent. The error of the whole architecture on a given example is given by

$$\begin{aligned} E &= \frac{1}{2}(d - y)^2 \\ &= \frac{1}{2}(d - \sum_i g_i \sum_j y_{ij} g_{ij})^2 \end{aligned}$$

This error depends on the parameters of the gating networks and the expert networks. The partial derivative of the error with respect to the output of the $(i, j)^{th}$ expert network is

$$\begin{aligned} \frac{\partial E}{\partial y_{ij}} &= \frac{\partial E}{\partial y} \frac{\partial y}{\partial y_i} \frac{\partial y_i}{\partial y_{ij}} \\ &= -(d - y) g_i g_{ij} \end{aligned} \tag{3.12}$$

We can use the gradient in equation 3.12 and formula 3.3 to compute δ_i. We see that there are differences compared to the mixtures of experts method. The experts are adjusted to reduce the error $(d - y)^2$ between the desired output and the output of the whole architecture. This could increase the error $(d - y_{ij})^2$ of an individual expert on the given example! The second difference is that an expert is adjusted in proportion to its responsibility $g_i g_{ij}$ in the architecture for the given input, instead of to the more meaningful posterior probability that an expert has produced the desired output. Because this error-term does not reflect how good one expert approximates the target value, it is possible that bad fitting experts are adjusted the most. However this disadvantage is overcome after a while, when the gating networks have learned to assign the correct responsibilities to the experts when given an input vector.

The partial error derivative with respect to the ith output unit of the top-level gating network is given by

$$\begin{aligned} \frac{\partial E}{\partial s_i} &= \frac{\partial E}{\partial y} \frac{\partial y}{\partial s_i} \\ &= \frac{-(d - y)(y_i - y)}{\sum_j s_j} \end{aligned} \tag{3.13}$$

and the partial derivative with respect to the jth output unit of the ith second level gating network is

$$
\begin{aligned}
\frac{\partial E}{\partial s_{ij}} &= \frac{\partial E}{\partial y}\frac{\partial y}{\partial y_i}\frac{\partial y_i}{\partial s_{ij}} \\
&= \frac{-(d-y)g_i(y_{ij}-y_i)}{\sum_k s_{ik}}
\end{aligned}
\tag{3.14}
$$

These gradients reflect how well an expert or cluster approximates the target value with respect to the approximation given by the architecture. When the partial derivative is negative, the expert or cluster can be used to minimize the error of the architecture as a whole and should be made more responsible so that the architecture will better approximate the target value. We will use equations 3.13 and 3.14 to compute δ_i by equation 3.3 and the weight update rule 3.2 will move the experts to the regions where they can minimize the error of the architecture.

Discussion

Hampshire and Waibel proposed to use the Meta-Pi network to learn to combine multiple experts which have already been trained in subspaces of the input space. When the division of the input space is known and experts can be trained independently, the use of the Meta-Pi feed-forward network improves generalization abilities. This was shown on a speech recognition task where experts were trained on data from different speakers [Hampshire89] and on learning to play backgammon with multiple experts [Boyan92]. However, the Meta-Pi network can also be used when the experts are initialized randomly which makes the method more powerful, because we do not have to know an a priori input space division.

Propagate nodes in a Meta-Pi architecture use the gates of the gating networks to propagate a weighted sum of the output vectors of the experts or cluster of experts to above. Learning rules are defined so that experts or clusters which can minimize the error of the weighted sum on an example are made more responsible in the future. One problem with this approach is that all experts have to learn to minimize a part of the error of the whole architecture instead of their own error and this results in interference between experts. When one expert changes its weights, the error function $E(w)$ of the whole architecture changes its evaluation of the current state of the architecture. This results in a changed evaluation of the state of each individual expert. This is why convergence until a stable weight setting can take a long time.

One method to overcome this problem is to use propagate nodes which use some selection device so that an expert is only used when it is important enough. When we increase this selection threshold, we end up with using one expert for each example and interference effects are gone, because the error function of the whole architecture and the

chosen expert become the same. This selection threshold can be seen as a method to make a more localized representation.

3.3.3 A Selection Threshold for Faster Propagation

To speed up the propagation of an architecture, we introduce a selection device which is used by the propagate nodes to select experts. This selection device uses a selection threshold to determine if the responsibility or gate of an expert is high enough, so that only important experts are used in the weighted sum. If an expert's responsibility is not larger than the selection threshold, the expert is not invoked. This can save a lot of time, because if the selection threshold of a propagate node is very high, e.g. .3 for an architecture which uses 2 experts, then many times only 1 expert has to be invoked. Furthermore, when only one expert is invoked, the gating network and the other expert network do not have to be trained.

Another advantage of using a selection threshold, is that we have found a method to attack the problem of interference between experts for the Meta-Pi network architecture. When the selection threshold is high enough we will only select one expert and it can learn to minimize its own error.

For both architectures, the selection threshold can be used to make a more localized representation. When the selection threshold is very high, the performance level of an expert is not degraded by learning too many examples which are not really its destination.

When we want to use a selection threshold, we define the following intermediate step before calculating the final gates

$$\text{if} \quad (g_i > ST_i) \quad \text{then} \quad s_i = s_i$$
$$\text{else} \quad s_i = 0$$

In this equation g_i are the intermediate gates, which are computed by normalizing them over all experts by equations 3.6, 3.7, and 3.11. ST_i is the threshold for the ith cluster or expert and can be changed on-line. After this computation we have eliminated the experts with low responsibilities. Now we must compute the final gates by using the new vector $\vec{s}$ in equations 3.6, 3.7, and 3.11. When learning the new gates we have to make the partial error derivative with respect to the ith output unit of the gating vector zero, if the ith cluster or expert has not been selected. We do this by using 0 for $F'(i_i)$ in equation 3.3.

3.3.4 The Differences between the Two Architectures

As we have seen, the two architecture use two different error-functions for the architectures. The differences between the two architectures arise from these chosen error-functions. When one would like to develop other architectures for some kind of task,

	Meta-Pi architecture	HME architecture
1	co-operating experts	competing experts
2	experts learn to reduce the error of the architecture	experts learn to reduce their own error
3	experts may increase their own error	experts may increase the error of the architecture
4	experts are adjusted in proportion to their responsibility $g_i g_{ij}$	experts are adjusted in proportion to their joint posterior probability $h_i h_{ij}$
5	experts are shifted to a region of the input space if its output is on the 'right' side of the error of the architecture	experts are shifted to a region of the input space if its output out-competes the other experts' outputs

Table 3.1: Differences between the Meta-Pi Architecture and the HME Architecture

e.g. one wants to use multiple experts to solve successive subtasks, one should try to construct an architecture with a chosen error-function which, when minimized, will result in the desired division of the input space. The two error-functions are duplicated here for readers' convenience.

- The HME likelihood function has to be maximized

$$P(d|x) = \sum_i g_i \sum_j g_{ij} e^{-\frac{1}{2}(d-y_{ij})^2}$$

From this we may define the error-function as

$$E = 1 - P(d|x)$$

- The error-function of the Meta-Pi network architecture has to be minimized

$$E = \frac{1}{2}(d-y)^2$$
$$= \frac{1}{2}(d - \sum_i g_i \sum_j y_{ij} g_{ij})^2$$

We can see that the desired output is inside the summations for the HME architecture and outside the summations for the Meta-Pi architecture. From this, the differences which are outlined in table 3.1 arise.

In the following section we will see what for consequences these differences have when learning a simple discontinuous function. We expect that the HME architecture with a winner takes all (WTA) forward propagation will be the best, because it will never use an expert for the function approximation if this expert is located at the wrong side of the discontinuity. All other approaches may always suffer from the problem that an expert is used which makes a large error on an example.

3.3.5 Symbolic Rules Architecture

The third architecture which has been studied is the use of a knowledge base containing symbolic rules to decide which expert has to be chosen for evaluating an example. The knowledge base has to be considered as the gating network, but the symbolic rules are fixed. This means that when no a priori knowledge is available, we can not use this architecture. When the domain is large, a short knowledge engineering period might make the decomposition more useful. Again this is an intermediate way between using knowledge engineering and machine learning. Of course it is almost always possible to make a simple decomposition of the input space, and because the use of symbolic rules is very fast, an increased propagating speed can be obtained. Here we consider using symbolic rules which produce gate values in which only one expert gets a gate value 1, this means that for every example only one expert network is selected.

Gating Networks

The symbolic rules determine under which conditions which expert has to be chosen. Then only one selected expert network is invoked and it produces the output of the architecture. It is possible to invoke multiple expert networks by using symbolic rules, but this would slow down the propagating speed of the architecture. Although the linear regions where the expert networks are situated are not adjusted, an advantage of using this architecture is that the expert networks can be much larger than when the outputs of multiple expert networks are combined. Another advantage is that when the decomposition is very good, it does not have to be learned first.

$$
\begin{aligned}
\text{if } conditions_i \quad &\text{then} \quad g_i = 1.0 \\
&\text{else} \quad g_i = 0.0
\end{aligned}
$$

Propagate Nodes

The propagate nodes propagate the output of the selected expert network to the top of the tree (only one gate value has the value 1.0).

$$ y = \sum_i g_i y_i $$

Gradient Descent on a Squared Error Function

The error-function of the architecture is the same as the error-function of the Meta-Pi network. The difference is that only one one expert has a responsibility to produce an output for an example. This makes the learning rules the same, but many expert networks

do not have to be adjusted, because their gradients are zero (see 3.12). The symbolic rules are not adjusted, although it is possible to use machine learning techniques to adjust the regions in which expert networks perform.

3.4   Experiment : A Discontinuous Function

Experiments have been performed to analyze the differences between using the three hierarchical architectures and a single network (monolithic architecture). We have discussed the problem of learning discontinuous functions, so one simple discontinuous function will be used to validate the methods.

3.4.1   Experimental Design

The different methods have been evaluated on a simple discontinuous function (see figure 3.7). The function is defined as

$$\begin{aligned} f(x) = & \sin(2\pi x + \pi) + 1 \qquad 0 \le x \le .5 \\ = & \sin(2\pi x - \pi) - 1 \qquad .5 < x \le 1 \end{aligned}$$
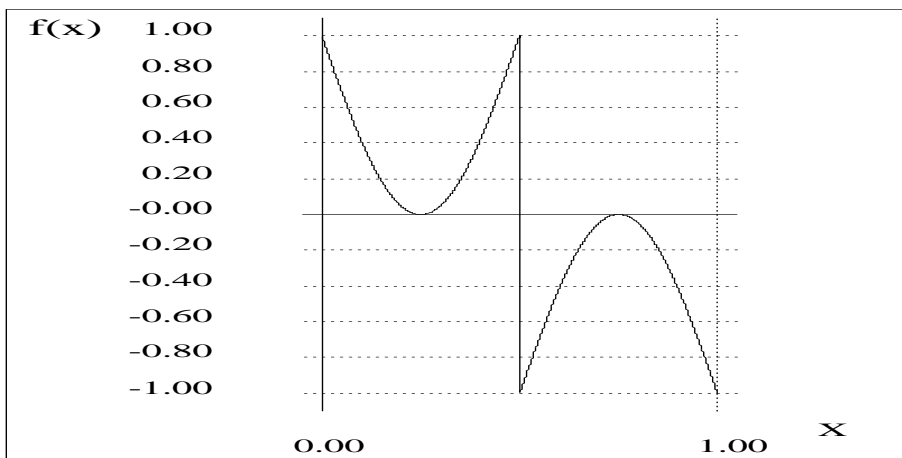
Figure 3.7: the target discontinuous function.

All modular architectures used one linear gating network and two expert networks containing 3 hidden units, which is enough to learn the split and the two sub-functions. Simulations were performed with hidden units which used sigmoids with two different neuron sensitivities :

- 1.0 (global), which is almost always used

- 5.0 (local), which gives the hidden units a more local orientation and updates their incoming weights faster.

Used Methodologies

Monolithic Architectures:

- Single Networks with 3 hidden units.

- Single Networks with 6 hidden units.

Modular Architectures with 2 Expert Networks and 1 Gating Network

- Symbolic Rules which choose one expert for values below .5, and the other expert for values above .5. Thus, they encode a perfect decomposition.

- Hierarchical mixtures of experts : Mixing the outputs (MTO).

- Hierarchical mixtures of experts : Winner takes all (WTA).

- Meta-Pi : Without selection threshold.

- Meta-Pi : With increasing selection threshold.

The Expert Network Parameters

- Learning rate : all expert used a learning rate which was slowly increased in the beginning of the learning process. This was done so that the decomposition was found before the expert networks had stabilized. Good starting learning rates were found to be between 0.007 for the networks with high neuron sensitivities and 0.04 for the networks which use low neuron sensitivities, hereafter they were multiplied by 1.0001 after each cycle for the first 20,000 cycles. Thus, the learning rates were increased from 0.007 to 0.052 and from 0.04 to 0.30, and kept constant afterwards.

- Momentum : 0.3

- Initialization of the weights : between -0.2 and 0.2

The Gating Network Parameters

- Learning rate : The learning rate of the gating networks was decreased so that once a decomposition was learned, the regions of the experts did not change too much. The learning rate was initialized at .7 and was multiplied by .999999 after each cycle.

- Momentum : 0.0

- Initialization of the weights : When we start we do not know a decomposition so we can make the responsibilities of the experts equal over the whole example space. We did this by setting the connections between units at 0.5 and the bias of the output units at 0.0.

Presentation of Learning Examples

- Examples were drawn randomly and learned on-line, each presentation of an example is a cycle or iteration.

- The root of the mean squared error (RMS) was computed after each 25,000 cycles on a uniform distribution of thousand points, and the smallest RMS over all tests per simulation was kept as the final result. This was done because the error was not decreasing all the time.

- 1,000,000 cycles was the maximum for one simulation, when the RMS was lower than .01 the simulation was stopped earlier.

3.4.2   Experimental Results

In table 3.2 the simulations are summarized with expert networks which use low neuron sensitivities in the hidden units. We can see that single networks are not able to approximate the function. This is not very surprising, because a global model cannot store a discontinuous function. What is not very surprising either is that by using rules to encode a perfect split, the two experts have no problem to learn their sides of the discontinuity: all simulations reached a RMS < 0.01. More surprising is that the mixtures of experts hierarchy with the winner takes all forward propagation always converged to good solutions, which is as good as using the perfect rules. The HME architecture with the mixing the outputs strategy does not do as well what was expected, although in this case the same good divisions have been learned. The higher error can be explained by the fact that around the discontinuity the outputs of both experts are mixed, because the gating network cannot learn to switch from one to the other expert at once (this would require representing the discontinuity, what we want to avoid). Finally the Meta-Pi hierarchy could not approximate the function better than the single networks. The single

Architecture	hidden u.	Mean RMS	SD RMS	Lowest RMS
Single Net	3	.337	.014	.329
Single Net	6	.331	.000	.331
Rules	3*	.0093	.0013	.0072
HME WTA	3*	.0077	.0023	.0036
HME MTO	3*	.103	.001	.102
Meta-Pi	3*	.330	.001	.328

Table 3.2: Simulations with a hidden unit sensitivity of 1.0 on the discontinuous function. Simulations were repeated 20 times and each time the best RMS was recorded. Only the rules and the winner takes all strategy of the HME architecture converged to solutions with RMS < .01. * means per expert.

networks and Meta-Pi hierarchy were not able to approximate the discontinuous function when neuron sensitivities of 1.0 were used. There were no hidden units which could learn the discontinuity (figure 3.8), because the region with the discontinuity is too small and contains conflicting details.

Table 3.3 shows that using a neuron sensitivity of 5.0 in the hidden units makes a big difference for the single networks and also for the Meta-Pi hierarchy. The hidden units with a high neuron sensitivity were able to learn the discontinuity, although it is clear that using 3 hidden units is not enough, the smallest single network was able to approximate the function better than the larger single network with the low neuron sensitivity! We can also see that the rules and HME with the winner takes all propagate strategy do as well as before with the low neuron sensitivity. The HME with the mixing the outputs strategy is the only architecture which obtains worse results with the high neuron sensitivities. The high standard deviation may be explained by the fact that in some cases the architecture learns to make one expert more responsible than the other for all inputs. Some different approximations are given in Figure 3.9.

Table 3.4 shows the number of cycles needed to converge to an approximation with RMS < .01. It shows no significant difference between using the high or low neuron sensitivity, although the fastest simulations were using the high sensitivity. The rules converge faster, but this difference is not surprising because the division does not have to be learned first.

Experiments with the Selection Threshold

Simulations with the use of a selection threshold in the propagate nodes confirmed our hypotheses that learning would become much quicker. With a selection threshold of 0.01 the time saved is about 30% for the Meta-Pi and HME architectures with low neuron sensitivities. This speed up can be explained by the fact that most examples do not
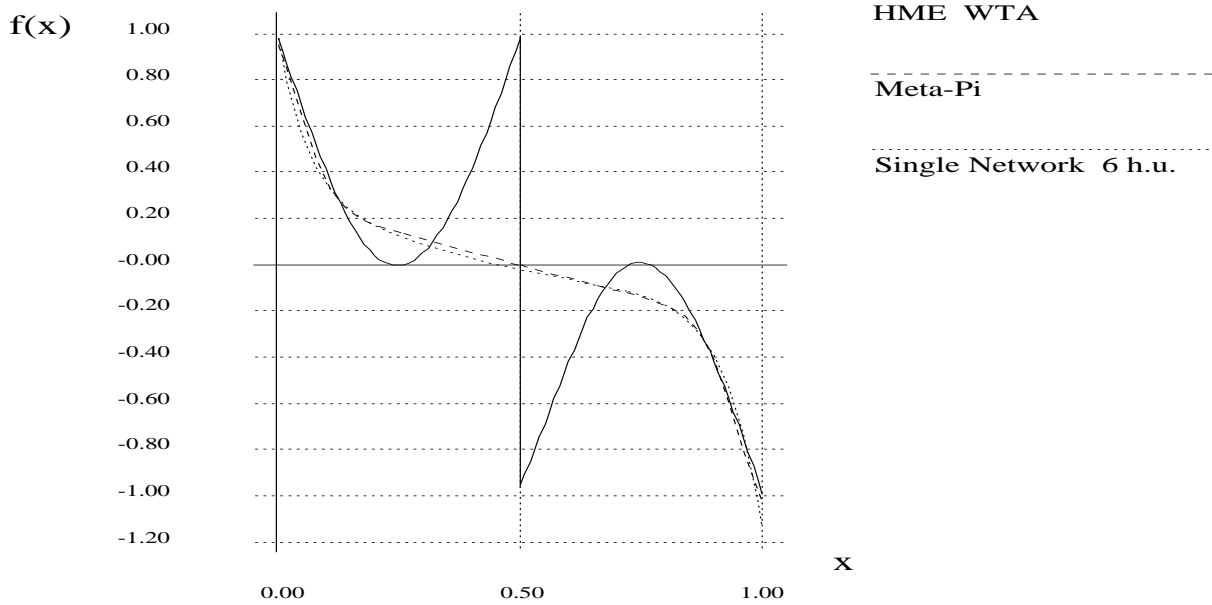
Figure 3.8: the function approximations of the different methods with neuron sensitivities 1.0
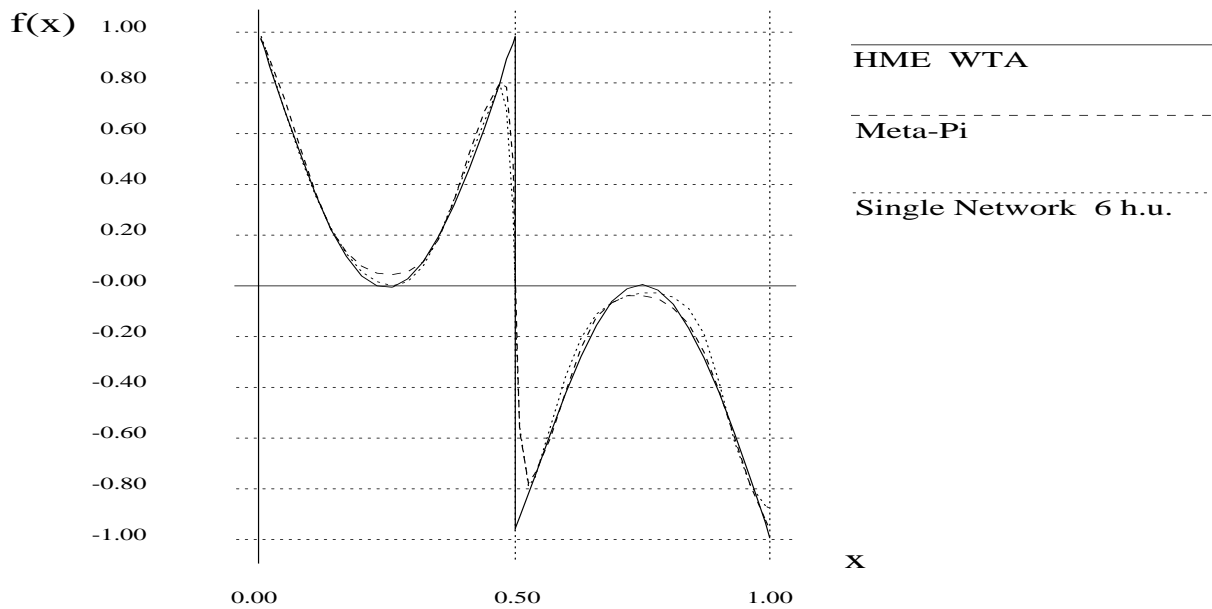
Figure 3.9: The function approximations of the Meta-Pi hierarchy and the single network with 6 hidden units. With a neuron sensitivity of 5.0 in the hidden units, they were better able to learn the target function.

Architecture	hidden u.	Mean RMS	SD RMS	Lowest RMS
Single Net	3	.263	.015	.214
Single Net	6	.093	.013	.083
Rules	3*	.0097	.0015	.0079
HME WTA	3*	.0077	.0011	.0060
HME MTO	3*	.118	.049	.109
Meta-Pi	3*	.097	.015	.081

Table 3.3: Simulations with a hidden unit sensitivity of 5.0 on the discontinuous function. Simulations were repeated 20 times. The single networks and Meta-pi architecture now have less problems in approximating the discontinuous function than they had with a neuron sensitivity of 1.0. * means per expert.

Architecture	β h.u.	Mean cycles	SD cycles	minimum
Rules	1.0	$254 * 10^3$	$46 * 10^3$	$175 * 10^3$
Rules	5.0	$139 * 10^3$	$127 * 10^3$	$60 * 10^3$
HME	1.0	$428 * 10^3$	$203 * 10^3$	$150 * 10^3$
HME	5.0	$439 * 10^3$	$241 * 10^3$	$125 * 10^3$

Table 3.4: Number of cycles needed to converge to states with RMS < .01. The simulations were repeated 20 times.

have to be learned by the gating network and one expert network, but only by one expert. Performance levels remain the same with this low threshold. The mixtures of experts methodology does not allow a fast rising selection threshold, for often one expert dominates over the other and this would result in a dead expert which is never used anymore. For the Meta-Pi network, a fast rising selection threshold changes the results of the simulations as we can see in table 3.5. Although it is very difficult to find an algorithm which increases the selection threshold very carefully, some simulations show good results. Good divisions (between x =.49 and .51) have been found for expert networks with 3,4 and 5 hidden units in respectively 70%, 75% and 85% of the simulations. The Meta-Pi methodology can profit from the selection threshold to increase the competition between the experts and to minimize the co-operation.

Experiments with Extended Back-propagation

Finally some experiments have been performed to evaluate the learning rule **extended back-propagation** (BP+) [Sperduti92], see Appendix B for a description. We have seen that the settings of the neuron sensitivities are important for learning the discontinuous function. BP+ is able to change neuron sensitivities so that learning rates are adapted

Architecture	hidden u.	Mean RMS	SD RMS	Lowest RMS
Meta-Pi	3*	.110	.082	.0065
Meta-Pi	4*	.078	.073	.0065
Meta-Pi	5*	.056	.050	.0064

Table 3.5: Simulations with an increasing selection threshold. The threshold is equal for both experts and is raised from 0.0 until 0.5, so that finally only one expert is chosen for an example. At first the threshold is increased fast, later on it is increased slowly. The simulations were repeated 20 times.

on-line. E.g. when one wants to decrease the learning rate during the learning process, this is not longer necessary, because BP+ can decrease the neuron sensitivity of the output unit itself. Table 3.6 summarizes an experiment with the same single network with six hidden units and high initial neuron sensitivities as before. It is trained by normal back-propagation and with BP+. The learning rates for the neuron sensitivities are :

- Hidden unit : 0.05

- Output unit : 0.001

The number of cycles until convergence (RMS < 0.15) is compared by performing 10 simulations in which each 10,000 cycles the network is tested to determine the RMS. The results (see table 3.6) indicate that using BP+ results in faster learning, so in the next Chapter BP+ will be used for learning game evaluation functions.

Learning Rule	Mean cycles	SD cycles	minimum
BP	$239 * 10^3$	$54 * 10^3$	$170 * 10^3$
BP+	$157 * 10^3$	$30 * 10^3$	$120 * 10^3$

Table 3.6: The number of cycles needed to converge to a state with a RMS < .15 for a single network. Simulations were repeated 10 times. The results show that using BP+ results in faster learning.

3.5   Discussion

In this Chapter we have described the difficulty of learning discontinuous functions with single neural networks. The global network parameters will smooth important details of the function around the discontinuity, which results in a large local error. Hierarchical architectures can circumvent this problem by fitting local expert networks on both sides of the discontinuity. Three methodologies for constructing hierarchical neural network

architectures are described and the learning rules are given which follow from the chosen error-functions.

The first methodology we have described is the hierarchical mixtures of experts hierarchy and it is shown to be a powerful methodology for learning discontinuous functions. In all simulations the gating network learned to decompose the function into two sub-functions which could easily be approximated by two independent expert networks (see table 3.2). The methodology only works well when a hard division is used, but when the outputs of the experts are blended, the architecture does not work better than a single network with high neuron sensitivity in the hidden units (see table 3.3). The second methodology which uses the Meta-Pi network was shown to work no better than a single network with the same amount of parameters. The third modular architecture uses symbolic rules to decompose the input space. When learning the discontinuous function and a perfect decomposition is known, this architecture obtains the same performance level as the winner takes all strategy for the HME architecture. Of course the learning speed is increased, because a decomposition does not have to be learned first.

Some simulations with neuron sensitivities in the activation function of the hidden units have been studied. If these hidden neuron sensitivities are set on higher values, then the slopes of the activation functions are steeper. The results (see tables 3.2 and 3.3) showed better local tuning of the parameters for single networks and the Meta-Pi hierarchy, if the neuron sensitivities were set on higher values. BP+ is an extension of normal back-propagation which can learn the neuron sensitivities. Results have indicated that the use of BP+ makes the learning process faster (see table 3.6). The next Chapter will show how these findings can be used to learn to play games by temporal difference learning.

Chapter 4

TD Learning with Multiple Networks

4.1 Learning to Play a Game

In this Chapter we will use the different architectures to learn to play the games of tic-tac-toe and the endgame of backgammon (disengaged bear-off). Playing games are tasks which require the computer agent to differentiate between positions and to decide, depending upon the position, which move to make. Playing a game is a Markov decision process (see Appendix A), because all states, actions, rewards, and transition probabilities between states are known. The Markov property states that all transitions and rewards depend only upon the current state and the current action [Whitehead92]. If we have an evaluation function for positions in the game, we can generate all possible moves, use the evaluation function to compare them and select the move which results in a position with the highest evaluation. When the evaluation function is very accurate, the Markov property states that it is useless to use look ahead strategies to improve the evaluation. The generate/evaluate/compare/select procedure is the control policy of the agent.

Initially the evaluation function is unknown and we will have to learn it with e.g. neural networks. A neural network models an evaluation function V which is defined as :

$$V(\vec{x_t}) = E(r|x_t)$$

with :
$\vec{x_t}$: the state vector of the position after t moves.
$E(r|x_t)$: the expected result r of the game given that we start in x_t and follow the current control policy.

So a neural network acts as a predictor of the result of the game when given a state vector.

A state vector represents a board position and must include all important features of the position. For the state vector it does not matter if it is white's or black's turn. This is very useful, because the input-pattern implicitly encodes the fact that the opponent has to move next.

As long as we are improving the evaluation function, the control policy will improve the agent's performance. Learning game evaluation functions with a neural network requires two procedures. One for acquiring learning examples and one for training the multi-layer networks. We have seen that the error back-propagation algorithm can be used for the latter. In this work acquiring learning examples will be done by performing simulations with the agent.

Several methods exist to create board positions and to calculate evaluations with which we can improve our evaluation function. One way to create examples $\{x_t, E(r|x_t)\}$ for improving the evaluation function is by the use of dynamic programming [Whitehead92] in which we compute examples for all possible positions. In dynamic programming, we start with an arbitrary control policy which is iteratively improved by changing it so that for all possible positions from the endgame until the start of the game, the move which maximizes the merits is chosen. The evaluation of each position is the same as the evaluation of the position which results when the chosen move is played. This approach is difficult to use however, because for games the number of possible positions is very large and the computations involved inhibit an efficient learning process.

A more efficient way is to use heuristic dynamic programming or reinforcement learning (RL) in which the network plays games against itself or a fixed opponent. When playing games, external feedback or reinforcement r is received when a game is finished and the game rules conclude that the agent has won ($r = 1$), has lost ($r = -1$), or has played equal ($r = 0$). From a played game and reinforcement we can generate examples by the temporal difference (TD) methods (Appendix A).

Two RL-formalisms have been developed : AHC-learning [Sutton88] and Q-learning [Watkins92]. Both are able to learn a control policy which maximizes an agent's performance level, provided that a linear representation of the input (lookup tables) is used, all actions are repeatedly sampled in all states and a proper scheduling of the learning rates is made [Watkins92, Dayan94]. The formalisms are formally described in Appendix A. Here we will concentrate on using both formalisms to learn game evaluation functions. AHC-learning learns an evaluation function for states. This V-function can be used to compare all possible positions which result from playing a legal move in the current position. The disadvantage of this is that all resulting positions have to be computed first. Q-learning learns an evaluation function for state-action pairs. This Q-function can be used to compare all legal moves in the current position, without the need of computing the resulting positions, although the considered moves have to be encoded in the input vector of an architecture, which enlarges the input space.

In the next section we will use the game of tic-tac-toe, and we will evaluate AHC-

learning and Q-learning with the different architectures. The different architectures will be compared by their abilities to learn a control policy which beats a fixed opponent as much as possible. The succeeding section uses the endgame of backgammon and AHC-learning to test the obtained accuracy of the evaluation function modelled by the different architectures. This will be done by the use of the program BOINQ, which uses dynamic programming to accurately calculate the evaluation of endgame positions in the game of backgammon.

4.1.1   AHC-learning of Game Evaluation Functions

Adaptive heuristic critic (AHC) learning directly learns the V-function. A move is selected in a position x_t by the following procedure (control policy) :

- Generate all legal moves

- Compute the resulting positions for all legal moves

- Evaluate all positions with the V-function

- Select the move which results in the positions with the highest evaluation. Thus, $m_i = argmax_j\{V(x_j)|m_j \in moves(x_t); x_j = T(x_t, m_j)\}$

When we assume that the current control policy is followed throughout the future, the evaluation $V(\vec{x_t})$ of a position x for white after he has made his move must be equal to the complement of the evaluation $V(\vec{y_{t+1}})$ of position y for black after his move. Therefore, the optimal V-function must satisfy :

$$V(\vec{x_t}) = -V(\vec{x_{t+1}}) \tag{4.1}$$

The difference between the two evaluations in equation 4.1 is the TD(0) error and can be used to create an example. Suppose we have played a game and stored all positions and their evaluations which occurred in that game. Then we can use TD(λ) to calculate the temporal differences so that the played game is translated into examples (see figure 4.1).

We will not always follow the same control policy, because this will probably lead to a local minimum in which the agent repeatedly makes the same mistakes. Therefore some kind of exploration is needed. A natural trade-off between exploitation for maximizing the agent's performance level and exploration is to choose moves randomly to a probability distribution determined by the evaluation of the resulting positions [Thrun92]. The probability $P_t(m_i)$ of selecting a move m_i when looking at position x_t is computed by the Boltzmann distribution

$$P_t(m_i) = \frac{e^{V(\vec{y_i})/T}}{\sum_{m_k \in moves(x_t)} e^{V(\vec{y_k})/T}} \tag{4.2}$$

TD(λ) procedure for AHC-learning:

Goal :   map a played game onto a set of examples $(\vec{x_t}, V'(\vec{x_t}))$.   One example
is made for each position which has occurred in the game. $V'(\vec{x_t})$ is the
desired evaluation for the state vector $\vec{x_t}$ and is calculated by the TD(λ)
method.   Notice that the V-value of the starting-position is also
calculated, but never used by the control policy.

Input   :
$\vec{x_t}$:   t = [0..M]
$V(\vec{x_t})$:   t = [1..M]
r
λ

Output :
A set of examples $Example(i, \vec{x_t}, V'(\vec{x_t}))$ i = [1..M+1], t = [0..M].

Algorithm :
1) $V'(\vec{x_M})$:= r;
2) store[$Example(1, \vec{x_M}, V'(\vec{x_M}))$];
3) i := 2;
4) t := M-1;
5) While (t $\geq$ 0) do
6) $V'(\vec{x_t})$:= − $(\lambda V'(\vec{x_{t+1}}) + (1 - \lambda)V(\vec{x_{t+1}}))$;
7)   store[$Example(i, \vec{x_t}, V'(\vec{x_t}))$];
8)   t := t-1; i := i+1;

Figure 4.1: The TD(λ) algorithm for AHC-learning to translate a played game into examples for the neural networks. All state vectors $\vec{x_i}$ are positions in which we consider that white has played the last move.

Where a move m_i (which is an element of the set of possible moves : $moves(\vec{x_t})$) results in position y_i ($T(x_t, m_i) = y_i$), and the temperature T adjusts the randomness of move selection.

4.1.2   Q-learning of Game Evaluation Functions

Q-learning learns a Q-function in which $Q(\vec{x_t}, m_t)$ represents the evaluation of making move m_t in position $\vec{x_t}$. A move is selected in the current position by the following control policy :

- Generate all legal moves

- Evaluate all moves by using the Q-function

- Select the move which results in the highest evaluation.
  Thus, $m_i = argmax_j\{Q(x_t, m_j)|m_j \in moves(x_t)\}$

For games $Q(\vec{x_t}, m_t)$ is equal to $V(\vec{x_{t+1}})$ where $\vec{x_{t+1}}$ is the resulting state vector, because games are deterministic and the same move in a particular position always results in the same position. The control policy is to choose the move m_t, for which $Q(\vec{x_t}, m_t)$ is maximal over all moves. Therefore, the optimal Q-function must satisfy :

$$Q(\vec{x_t}, m_t) = -Max\{Q(\vec{x_{t+1}}, m_{t+1})|m_{t+1} \in moves(x_{t+1})\} \tag{4.3}$$

The difference between both sides of equation 4.3 is again the TD(0)-error and will be used to improve the Q-function. Figure 4.2 shows how a played game can be used to create a set of examples by the TD(λ)-learning Algorithm for Q-learning.

The Q-function not only depends on the position, but also on the move. With Q-learning we do not have to generate and compare all possible resulting positions, but only all possible moves. The Q-value of the different moves in the position x_t will be used to select a move by the Boltzmann distribution

$$P_t(m_i) = \frac{e^{Q_t(\vec{x_t}, m_i)/T}}{\sum_{m_k \in moves(x_t)} e^{Q_t(\vec{x_t}, m_k)/T}} \tag{4.4}$$

The difference with AHC-learning is that the architecture must know which move is evaluated. A possible way is to encode the move in the input vector for the network, but this might decrease the important discriminating abilities for the different moves. Another way to let the architecture know which move is being looked at, is to use a representation in which for each possible move a different expert network is selected, which only has to evaluate the current position [Lin93]. We will use this approach to learn tic-tac-toe by Q-learning. The problem with this is that for a game like backgammon, the set of possible moves is very large ($\gg$ 1,000), so that the amount of required parameters becomes too large. When this is the case, we must encode the move in the input vector.

TD(λ) procedure for Q-learning:

Goal : map a played game onto a set of examples $(\vec{x_t}, m_t, Q'(\vec{x_t}, m_t))$. One example is made for each position which has occurred in the game. The move is stored so that it can be used for selecting the corresponding expert network or for expanding the state vector. $Q'(\vec{x_t}, m_t)$ is the desired evaluation for playing the move m_t when looking at the state vector $\vec{x_t}$, and is calculated by the TD(λ) method. Notice that the Q-value of playing no move (m_0 = 0) in the starting-position is also calculated, but never used by the control policy. The move m_M in the position x_M resulted in the final position with reinforcement r.

Input :
$\vec{x_t}$: t = [0..M]
m_t : t = [0..M]
$Max\{Q(\vec{x_t}, m_i) | m_i \in moves_t)\}$: t = [1..M]
r
λ

Output :
A set of examples $Example(i, \vec{x_t}, m_t, Q'(\vec{x_t}, m_t))$ i = [1..M+1], t = [0..M].

Algorithm :
1) $Q'(\vec{x_M}, m_M)$:= r;
2) store$[Example(1, \vec{x_M}, m_M, Q'(\vec{x_M}, m_M))]$;
3) i := 2;
4) t := M−1;
5) While (t $\geq$ 0) do
6) $Q'(\vec{x_t}, m_t)$:= − $(\lambda Q'(\vec{x_{t+1}}, m_{t+1}) + (1 - \lambda) *$
                    $max\{Q(\vec{x_{t+1}}, m_i) | m_i \in moves_{t+1}\})$;
7)   store$[Example(i, \vec{x_t}, m_t, Q'(\vec{x_t}, m_t))]$;
8)   t := t-1; i := i+1;

Figure 4.2: The TD(λ) algorithm for Q-learning to translate a played game into examples for the neural networks. All state vectors $\vec{x_i}$ are positions in which white has played the move m_i.

4.2 Tic-Tac-Toe

4.2.1 Problem Definition

The networks have to learn to play tic-tac-toe, therefore I have designed a knowledge base
(TTT) which acts as an opponent. The knowledge base contains the following three rules

```
TTT program

1) IF a given move wins the game THEN play this move.
2) ELSE IF a given move for the opponent would win the game,
        THEN block this move by playing on this field.
3) ELSE play a random move.
```

This teacher can be beaten by constructing a winning fork, which is a position in which
the player has two possible moves which win the game on his next turn. The maximal
obtainable performance level is about 0.614 (see Appendix C). This performance level
(match-equity) stands for (wins - losses)/games when one plays a match of a large number
of games with both white and black against the opponent TTT.

Playing games against an opponent with an architecture, instead of playing against
itself, requires some changes to the TD-procedures in figures 4.1 and 4.2. When we use
the control policy of the network, and use the evaluation of a position which results from
an action by this control policy, we might never be able to learn that in some positions
the opponent makes a lot of mistakes. That is why we use the moves of the opponent and
calculate the V-value and Q-value with the network on this move and position. So instead
of using the maximal Q-value, we use the Q-value of the move which is selected by TTT.
This is not only faster, but without this we might learn that a position is non-paying,
because the opponent can always play the best move, whereas he can make many mistakes
in such a position.

The input of the networks will consist of 9 units : each unit encodes one field. The
activation of an input unit will be :

- +1 for a circle (white)

- -1 for a cross (black)

- 0 for an empty field

The output V of the networks will lie between $+1 \Longleftrightarrow$ P(white wins) = 100%
and $-1 \Longleftrightarrow$ P(white loses) = 100%.

4.2.2   Experimental Design

For learning tic-tac-toe against TTT, a priori knowledge of how many expert networks to use is not available. Several different architectures have been tried out and the results of the best of them are used in the following. After pilot experiments, the temperature T, λ, and the parameters of the expert and gating networks were chosen to be the same for all architectures. Extended back-propagation was only used for learning the expert networks, which not only produced a small gain in performance compared to normal back-propagation, but also made searching for learning parameters easier and decreases the learning rates automatically (which is important for TD learning).

List of Experiments

- Monolithic architectures with 30, 50 and 80 hidden units.

- HME architectures with 2 experts containing 40 hidden units each. One architecture uses a selection threshold of 0.3 and the other does not use a selection threshold ($ST = 0.0$).

- Meta-Pi architectures with 2 experts containing 40 hidden units each. One architecture uses a selection threshold of 0.3 and the other does not use a selection threshold ($ST = 0.0$).

- Symbolic rules architectures with 9 experts containing 20 and 30 hidden units.

- Lookup tables which store the evaluations of all different positions in different entries.

- For all neural network architectures, experiments with high (3.0) and low (1.0) initial neuron sensitivities are performed.

The Expert Network Parameters

- Learning rate : .3

- Momentum : .5

- Initialization of the weights : between -.2 and .2

- Initial hidden unit neuron sensitivity : 3.0

- Initial output unit neuron sensitivity : 0.2

- Hidden unit neuron sensitivity learning rate : 0.1

- Output unit neuron sensitivity learning rate : 0.001

The Gating Network Parameters

- Learning rate : .1

- Initialization of the weights : between -.2 and .2

- Output neuron sensitivity : 1.0

Design of the Simulations

- After each played game, examples are constructed by the procedure in figure 4.1 for AHC-learning or the procedure in 4.2 for Q-learning. These examples are passed to the learning module which uses (extended) back-propagation to alter the weights.

- In one simulation an architecture played 40,000 games, alternating with white and black. This produces about 340,000 learning examples.

- After each 2,000 training games, 2,000 test games were played in which always the best move was selected by the control policy. The match equity ((the number of wins - the number of losses) / the number of played games) over these 2,000 test games was used in the results.

- The temperature T which is used in equation 4.2 and equation 4.4 for determining the amount of exploration, was annealed from .2 to .05

- λ was annealed from .8 to .2

4.2.3   Experimental Results

We will first discuss experiments with the architectures which use high initial neuron sensitivities, because this results in better performances; many simulations reached match-equities which proximate the maximal obtainable match-equity of 0.614.

An experiment with an architecture consists of 10 simulations. In the figures, the learning curves are presented which are averaged over the 10 simulations. The tables present the results which were averaged over the 10 simulations, in which for every simulation the 2,000 test games which obtained the highest average was kept as the result of that simulation. This was done because the performance is not monotonously improving when more games are played, although we could always copy an architecture when it has reached its maximal performance level so that the best state of the architecture in a simulation is kept. The most important feature in the experiments is the average of the maximal match-equities over all 10 simulations with an architecture; these results are presented in the tables.

The average over all maximal match-equities per simulation often reaches an equity which is little below 0.600. The standard deviation over the simulations is decreased from about .15 after 2,000 games, to about .1 after 10,000 games to about .02 after 40,000 games. Because the standard deviations in the first 10,000 games are so high and the equity rises from -.7 to about .4, this part of the simulation is not important and will not be shown in the figures.

Tic-Tac-Toe Experiment 1 : Monolithic vs. Mixture of Experts

In the first experiment we compare the performances of single neural networks with the HME architectures. We have tried out several single neural networks and found out that larger networks are performing better than smaller ones, although there is a maximum size after which the performance degrades. After we have found the single network architectures we wanted to use, we searched for hierarchies of expert networks with about the same amount of parameters as the largest single network. The best HME architectures used only two expert networks, because larger hierarchies did not perform better. Finally we experimented with using the selection threshold. This resulted in a comparison between the following architectures:

- A single network with 30 hidden units.

- A single network with 50 hidden units.

- A single network with 80 hidden units.

- A HME architecture with two expert networks of 40 hidden units each, without the use of a selection threshold.

- A HME architecture with two expert networks of 40 hidden units each, but with a selection threshold of 0.3.

Architecture-type	h.u.	E(equity)	SD(equity)	time
Single Network	30	0.526	0.030	51 min
Single Network	50	0.577	0.014	85 min
Single Network	80	0.600	0.010	136 min
HME S.T. = 0.0	2 * 40	0.592	0.011	138 min
HME S.T. = 0.3	2 * 40	0.592	0.009	78 min

Table 4.1: The average maximal match equities obtained by the monolithic and the HME architectures. The HME architectures perform as good as the largest single network, but with the use of a selection threshold, they become faster.
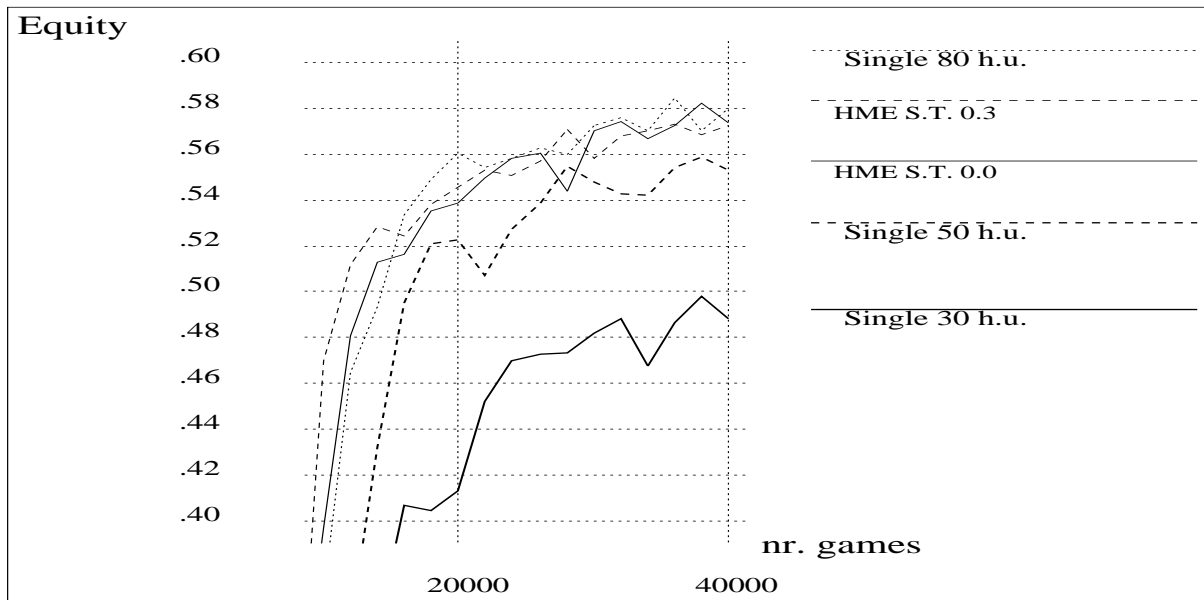
Figure 4.3: The results obtained by the single networks and the HME architectures. Each experiment with an architecture consists of 10 simulations. The figure shows the average match-equity after each 2,000 training games against TTT.

Figure 4.3 and table 4.1 show that the HME architectures with two experts (40 hidden units) and the largest single network obtained the best results when they are compared with the smaller architectures. The largest single network with 80 hidden units performed a little bit better, but the HME architecture with a selection threshold of 0.3 saves a lot of time by not invoking one of its experts in 98% of the times! The smaller single networks were not performing much slower, but reached a significant smaller expected match-equity (see table 4.2). The results show that architectures with a lot of parameters perform better than smaller architectures. The selection threshold is an efficient method to use more parameters, without decreasing the propagating speed of the architecture.

*	HME S.T. = 0.0	HME S.T. = 0.3
Single 30	≪	≪
Single 50	<	<
Single 80	≈	≈

Table 4.2: A comparison between the results obtained by the monolithic and HME architectures produced by t-tests. < means significantly worse ($\alpha = 5\%$) and ≪ means significantly worse ($\alpha = .1\%$).

When the WTA strategy was used in the HME architecture, the results were much worse. This was because often only one expert was chosen to evaluate all positions. When this strategy was used, the combination with TD learning resulted in examples which were constructed by using the evaluation of one of the experts. Therefore one expert is better able to approximate these examples, and will be chosen for all inputs.

Tic-Tac-Toe Experiment 2 : Monolithic vs. Meta-Pi

In this experiment the results are compared between the single networks from experiment 1 and architectures which use the Meta-Pi network. Just as in experiment 1, we have experimented with some different hierarchical architectures and kept two Meta-Pi archi-tectures which use two experts with 40 hidden units each, one without and the other with a selection threshold of .3. The learning curves are presented in figure 4.4.
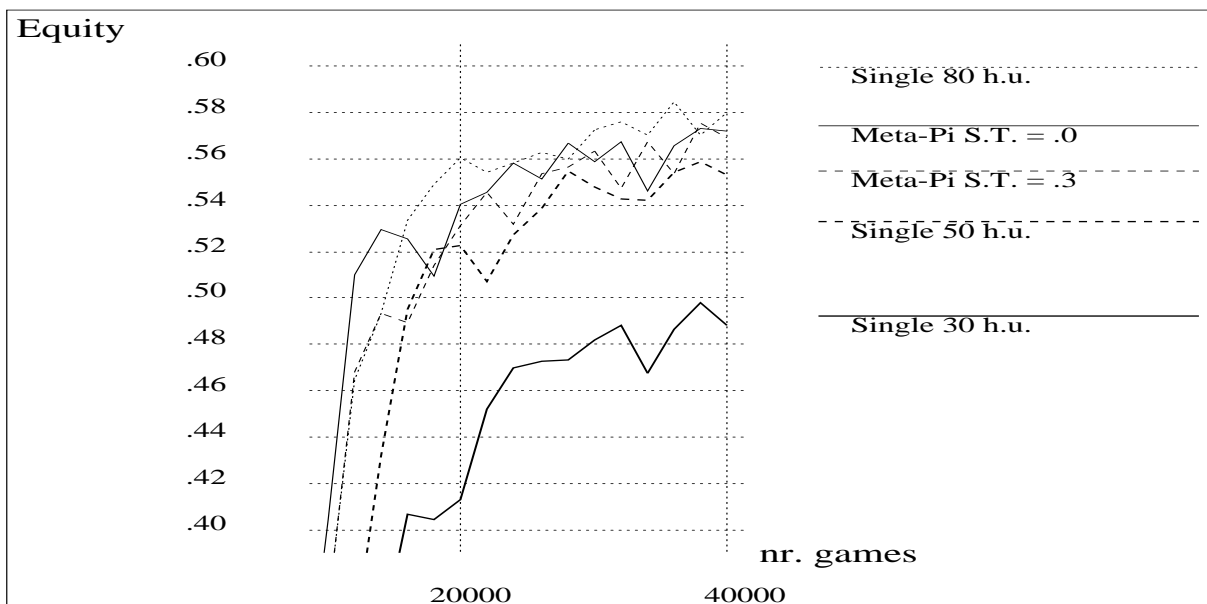
Figure 4.4: The results obtained by the single networks and hierarchies with a Meta-Pi network. The Meta-Pi hierarchies do not reach match-equities which are as good as the largest single network, but perform better than the single network with 50 hidden units.

As we can see in tables 4.3 and 4.4, the Meta-Pi architectures perform worse than the largest single network, and the fastest Meta-Pi architecture does not perform significantly better than the single network with 50 hidden units. The Meta-Pi network with a selection threshold could be used to trade-off learning speed against performance level, but performs slightly worse than the HME architecture.

Architecture-type	h.u.	E(equity)	SD(equity)	time
Single Network	30	0.526	0.030	51 min
Single Network	50	0.577	0.014	85 min
Single Network	80	0.600	0.010	136 min
Meta-Pi S.T. = 0.0	2 * 40	0.588	0.011	138 min
Meta-Pi S.T. = 0.3	2 * 40	0.587	0.016	78 min

Table 4.3: The average maximal match-equities obtained by the monolithic and Meta-Pi architectures

**	Meta-Pi S.T. = 0.0	Meta-Pi S.T. = 0.3
Single 30	≪	≪
Single 50	<	≈
Single 80	>	>

Table 4.4: A comparison between the monolithic and Meta-Pi architectures

Tic-Tac-Toe Experiment 3 : Monolithic vs. Symbolic Rules

This is another experiment in which we used the single networks. Now we compared them with an architecture which uses symbolic rules to select an expert network. The symbolic rules select for each move a different expert network to evaluate that move. This reduces the input space for each expert network, because the chosen field always contains a piece so that one particular bit in the input vector is always on. After some pilot experiments, we decided to keep the architectures which use 10 expert networks with 20 and 30 hidden units. Notice that one expert network is used for evaluating playing no move in the starting position, and therefore it is never used by the control policy.

Architecture-type	h.u.	E(equity)	SD(equity)	time
Single Network	30	0.526	0.030	51 min
Single Network	50	0.577	0.014	85 min
Single Network	80	0.600	0.010	136 min
Symbolic Rules	10 * 20	0.590	0.013	35 min
Symbolic Rules	10 * 30	0.598	0.004	51 min

Table 4.5: The average maximal match-equities obtained by the monolithic and symbolic rules architectures.

Figure 4.5 and table 4.5 show that having one expert network for evaluating the merits of playing a particular move gives a very good performance. These architectures have a lot of parameters, but each time only one expert needs to be invoked so they perform very

Figure 4.5: The results obtained by the single networks and hierarchies which use symbolic rules to select an individual expert network for evaluating each different move. These hierarchies are very fast and the hierarchy which uses expert networks with 30 hidden units reaches match-equities which are as good as the largest single network and performs much better than the single network with 50 hidden units.

***	Symbolic 20	Symbolic 30
Single 30	≪	≪
Single 50	<	≪
Single 80	>	≈

Table 4.6: A comparison between single networks and symbolic rules architectures

Equity
.60
.58
.56
.54
.52
.50
.48
.46
.44
.42
.40

Rules 30 h.u.
Single 80 h.u.
Rules 20 h.u.
Single 50 h.u.

Single 30 h.u.

20000   40000   nr. games

Figure 4.5: The results obtained by the single networks and hierarchies which use symbolic rules to select an individual expert network for evaluating each different move. These hierarchies are very fast and the hierarchy which uses expert networks with 30 hidden units reaches match-equities which are as good as the largest single network and performs much better than the single network with 50 hidden units.

***	Symbolic 20	Symbolic 30
Single 30	≪	≪
Single 50	<	≪
Single 80	>	≈

Table 4.6: A comparison between single networks and symbolic rules architectures

fast. Because the performance level of the largest single network is not better than the hierarchy which uses expert networks with 30 hidden units for different moves, but is much slower, the use of different action expert networks can be promising if the set of possible actions is small enough. Maybe this is why in some studies the results of Q-learning with one expert for one action outperformed the use of AHC-learning with one single network [Lin93]. The results of using the one expert for one move and the best HME architecture are not significantly different, although the HME architecture is slower.

Tic-Tac-Toe Experiment 4 : AHC-learning vs. Q-learning

This experiment compares AHC-learning to Q-learning. The use of an architecture with symbolic rules to select different networks for different moves fits well in Q-learning. This provides a method to let the architecture know which move it is evaluating, instead of encoding the move in the input vector. Again expert networks with 20 and 30 hidden units are used for both TD-paradigms.

Architecture-type	h.u.	E(equity)	SD(equity)	time
AHC-learning	10 * 20	0.590	0.013	51 min
AHC-learning	10 * 30	0.598	0.004	85 min
Q-learning	10 * 20	0.590	0.008	51 min
Q-learning	10 * 30	0.595	0.011	85 min

Table 4.7: The average maximal match-equities obtained by the architectures which use AHC-learning and Q-learning

****	Q-learning 20	Q-learning 30
AHC-learning 20	≈	≈
AHC-learning 30	>	≈

Table 4.8: A comparison between AHC-learning and Q-learning

Table 4.7 shows that using the division of selecting one expert for one move when used for AHC-learning or Q-learning results in almost the same performance levels. Both architectures are very fast, but can use a lot of parameters. The only difference with this AHC-learning paradigm and Q-learning in this experiment is that this AHC-learning paradigm shows a position after a move and Q-learning shows the position before the move is played to the chosen network. This makes selecting a move faster if computing the resulting position is not trivial, but Q-learning can not be used if we do not discriminate between different moves when we present a position and a possible move to the architecture. AHC-learning might use classes of moves, and because the state vector

encodes the position after a move has been played, discriminating abilities still exist. So, for a game such as backgammon we can better use AHC-learning and make use of classes of moves or classes of positions to choose between different expert networks.

Tic-Tac-Toe Experiment 5 : Lookup Tables

In this experiment we study lookup tables. Lookup tables use a different entry for each position to return the evaluation of that position. Lookup tables can store non-linear functions, but because they have no generalization ability, they have to use many parameters. Therefore the use of lookup tables is no viable paradigm when the state space is very large. However, Dayan has proved that lookup tables trained with temporal difference learning will find a global minimum [Dayan94]. Lookup tables are similar to the previous architectures with high neuron sensitivities for the hidden units, in the sense that most parameters are only adjusted in a specific part of the input space.

The results show that the use of lookup tables requires about 4,600 parameters, which is much more than the 880 parameters needed by the largest single network. The learning traject of the lookup tables is shown to be much faster than the use of neural network architectures (figure 4.6). Furthermore when we look at table 4.9, we can see that the lookup tables obtain the best results of all the architectures which are considered here. This means again that using many local basis functions results in faster learning and a better performance level when learning a discontinuous game evaluation function. The time to train a lookup table depends on its representation. The representation which was used here, is using a list so that everytime an evaluation of a position was needed, this position had to be compared with many entries in the list. This implementation is very inefficient, but the time needed for a simulation was about the same as for the neural network architectures. Naturally, a smart representation (e.g. a tree, or a hashing table) would require much less learning time than even the smallest network architecture.

Tic-Tac-Toe Experiment 6 : Architectures with low neuron sensitivity

In the last experiment, the use of low initial neuron sensitivities in the activation functions for the hidden units in the neural networks is studied. Instead of the previous starting values 3.0 for the neuron sensitivities, neuron sensitivities will be initialized on 1.0. This makes the networks more globally oriented, and (as we could see in Chapter 3) this makes it more difficult to learn discontinuous functions. The results of the experiment are given in table 4.10 and show that the architectures perform worse with low neuron sensitivities than with high hidden unit sensitivities. This confirms the results obtained in Chapter 3. When discontinuous functions have to be learned it is a good idea to start with activation functions with steep slopes. The results also show that using modular architectures work better than the monolithic architectures when low neuron sensitivities are used. It appears that the modular architectures are able to decrease the number of discontinuities which
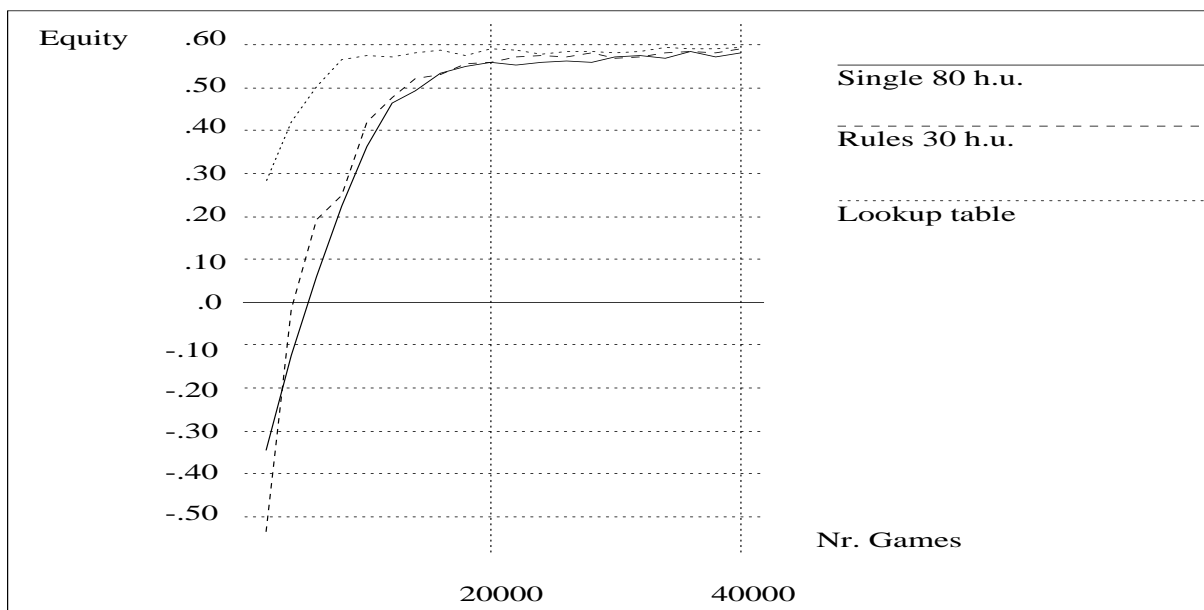
Figure 4.6: The results obtained by the lookup tables compared to the large single network and the architecture which uses symbolic rules to select an expert network. The use of lookup tables results in faster learning and a better final performance.

Architecture-type	h.u.	E(equity)	SD(equity)	time
Single Network	30	0.526	0.030	51 min
Single Network	50	0.577	0.014	85 min
Single Network	80	0.600	0.010	136 min
HME S.T. = 0.0	2 * 40	0.592	0.011	138 min
HME S.T. = 0.3	2 * 40	0.592	0.009	78 min
Meta-Pi S.T. = 0.0	2 * 40	0.588	0.011	138 min
Meta-Pi S.T. = 0.3	2 * 40	0.587	0.016	78 min
Symbolic Rules	10 * 20	0.590	0.013	35 min
Symbolic Rules	10 * 30	0.598	0.004	51 min
Lookup table	4,560*	0.606	0.007	102 min

Table 4.9: The results obtained by all architectures with high initial neuron sensitivities or local basis functions. The lookup tables obtain the best results, but they use the largest amount of parameters. In general we can say that the amount of parameters in an architecture is a good predictor for its performance.

Architecture-type	h.u.	E(equity)	SD(equity)	time
Single Network	30	0.384	0.060	53 min
Single Network	50	0.421	0.019	87 min
Single Network	80	0.434	0.027	138 min
HME S.T. = 0.3	2 * 40	0.498	0.031	82 min
Meta-Pi S.T. = 0.3	2 * 40	0.485	0.031	80 min
Symbolic Rules	10 * 30	0.507	0.042	53 min

Table 4.10: The results obtained by the architectures with an initial neuron sensitivity of 1.0. The architectures obtain worse results than the architectures which use high initial neuron sensitivities. We can see that the single neural networks obtain worse results than the modular architectures.

have to be learned by the expert networks. With low neuron sensitivities, the need to decompose the input space appears to be much larger than with high neuron sensitivities.

4.2.4 Discussion

We have seen that TD learning is an efficient way to learn a control policy for an agent. The obtained results (see table 4.6) show that by using large neural network architectures, we can get close to the maximal performance level of playing games of tic-tac-toe against an imperfect fixed opponent. The results also show that using many local basis functions to store the required knowledge results in an improved performance. Using multiple expert networks is an efficient way to use many parameters, because we can select independent neural networks for evaluating different moves and positions, which is much faster than always invoking one large monolithic network. The use of lookup tables provides an efficient way to store an evaluation function. They are not only very accurate, but when a smart representation is used, they are also very fast. Because every possible position needs a different entry in a lookup table, the amount of parameters grows prohibitively as the number of dimensions of the state space of a game increases. That is why the use of lookup tables is no viable alternative when the state space of a game is very large.

We have compared the results between the monolithic architectures and three architectures which use multiple expert networks. The HME architecture could make use of the selection threshold in an efficient way so that many times only one expert was invoked. This made it a lot faster than the largest single network, although the performances were about equal. The Meta-Pi architecture performed worse than the HME architecture, which confirmed the results obtained in Chapter 3. Using symbolic rules to select an expert to evaluate playing a specific move resulted in very good performance. These architectures can use a lot of expert networks, because each time only one expert needs to be invoked. The results obtained by these architectures were about equal to the

results obtained by the HME and largest monolithic architectures, but they performed a lot faster.

Another comparison was made between AHC-learning and Q-learning. The experiments showed that when both reinforcement learning methods are applied on the same architectures, the results are about the same. An advantage of using AHC-learning is that the architecture does not have to know which move is played. So when we want to learn a game which allows many possible moves (e.g. go, backgammon, draughts, chess) AHC-learning is the paradigm we have to use. Using Q-learning would require encoding the move in the state vector which might decrease generalization abilities, because the moves which resulted in a specific position might differ.

When we compare the obtained results with [Boyan92], who obtained a maximal match-equity of 0.474 after playing more training games, they are impressive. Except for a different input encoding and the use of TD(λ) instead of TD(0), the use of high initial neuron sensitivities in the hidden units could be the reason for the difference between the results. The results of Chapter 3, which showed that using high neuron sensitivities in the activation functions of the hidden units to learn a discontinuous function was advantageous, were confirmed in the considered task of learning a discontinuous game evaluation function.

The next section will show if using modular architectures or high hidden neuron sensitivities also improves learning a smooth evaluation function.

4.3   The Endgame of Backgammon

4.3.1   Problem Definition

In this section we study adaptive experts and TD learning for the endgame (bear-off) of backgammon. In contrast to the previous section where a discontinuous evaluation function had to be learned, the game evaluation function considered here is very smooth. However, the number of different positions is $1.5 * 10^9$, which is much larger than the $4.6 * 10^3$ positions which are used by the lookup tables to store the evaluation function of tic-tac-toe.

Our goal is to learn the V-function given by

$$V(\vec{x_t}) = E(r|\vec{x_t})$$

There exist two possible methods to learn this evaluation function :

- Supervised learning on learning samples $\{(x_1, V(x_1)), ..., (x_M, V(x_M))\}$ which are created by dynamic programming.

- Playing games with the architectures and creating examples with the TD methods.

We will compare supervised learning to TD learning with the different architectures, to evaluate the efficiency of TD learning. For supervised learning, learning samples consisting of the evaluation V for a given state $\vec{x}$ are generated from a program BOINQ which is able to compute these evaluations for the endgame directly. BOINQ uses dynamic programming to create a lookup table with the evaluations for all possible positions from 1 to 14 stones for both sides.

The networks used 68 inputs, 56 inputs encode the possible fields 0-6 where the stones for both players are allowed to stand. The maximal number of stones that can be on a particular field is 14, and so the number of stones on a field is binary encoded by 4 inputs. For the other 12 inputs some features are used which are hard to learn for the network itself. The following features were used (which were scaled between -3.0 and 3.0)

- Features for both players :

    - the pip-count.
    - the number of pieces out.
    - the standard deviation of the placement of the stones on all fields.
    - the mean of the placement of the stones on the fields.

- Overall features

    - A bit which indicates which player has taken off the largest amount of stones.
    - The difference in pip-counts.
    - A bit which indicates whether this difference has passed 15%.
    - A bit which indicates if the player has taken all pieces off.

4.3.2 Experimental Design

We compared temporal difference learning to supervised learning. The experiments exist of two parts : the first part of the simulations compare the performances of the different architectures on TD learning and supervised learning for a short learning time. The second part of the experiments compare the use of TD learning to supervised learning for a longer training time.

Experiment 1 : Effect of Architectures

In this experiment, small training times are studied to compare the different architectures.

For supervised learning the experimental setup is as follows. The architectures are trained on-line, repeatedly presented with a learning set of 500 samples for 120 epochs. The experiment is repeated 10 times, each time with a different set of 500 learning samples.

We have used 10 different learning sets, so that the results do not depend on one arbitrary drawn learning set. The total amount of iterations in a simulation is 60,000.

A simulation with temporal difference learning consisted of 10,000 games of self-play starting with randomly drawn positions in the endgame with a maximum of 14 against 14 stones. The mean number of pieces on the board in a starting position for a player is 7.5. Playing one game produces about 6 positions, which makes the amount of learning examples equal to the supervised paradigm. The difference in the presented learning samples is that for TD learning the distribution of the shown positions are on average closer to an end-position than the uniform distribution of the supervised learning set. This could be a disadvantage for TD learning, but is almost inevitable.

For both learning strategies, an independent test set of 2,000 randomly drawn perfect examples generated with dynamic programming was used. The best results of one simulation was kept as the final result of that simulation. For this, the supervised trained architectures were tested after each epoch, and the architectures which were trained by TD learning were tested after each 100 games. The experimental results were averaged over all simulations.

After a coarse search through the parameter space, we decided to keep the following architectures :

- Monolithic architectures with 0, 5 and 10 hidden units.

- The HME architecture with two expert networks with 5 hidden units without selection threshold.

- The Meta-Pi architecture with two expert networks with 5 hidden units without selection threshold.

- A symbolic rules architecture to choose one out of six expert networks with 10 hidden units. The symbolic rules use the difference in pip-count between both players (pip-count player 1 - pip-count player 2). The pip-count is the total number of fields all pieces have to advance from their current field before they can be taken off. We constructed the following division :
  ```
  1) IF the difference < -34 THEN category := 1
  2) ELSE IF the difference < -14 THEN category := 2
  3) ELSE IF the difference < -4 THEN category := 3
  4) ELSE IF the difference < 6 THEN category := 4
  5) ELSE IF the difference < 26 THEN category := 5
  6) ELSE category := 6
  ```

Although the propagate speeds of these architectures are the same (except for the smaller single networks), the architecture which uses symbolic rules has the largest number of parameters. For TD learning, initial neuron sensitivities of 1.0 and 3.0 were used in

the hidden units, so that we were able to compare the use of the different initial neuron sensitivities when learning a smooth evaluation function. Extended back-propagation was again used to train the expert networks.

Experiment 2 : TD vs. Supervised Learning

In this experiment, we use the single network architecture with 10 hidden units. We compare its performances on learning on a learning set, learning by self-play and learning by a combination of a learning set and self-play. We study the following experiments

- 1,600 epochs on 500 learning samples.

- 160 epochs on 5,000 learning samples.

- 130,000 games of self-play.

- 65,000 games of self-play combined with 800 epochs on 500 learning samples.

Each experiment is repeated five times. After each 40,000 iterations, the architectures are tested on a test set of 5,000 examples created by BOINQ. When the combination of self-play and TD learning is used, TD learning is continuously interchanged with supervised learning during the learning process. The architecture is tested after playing 3,250 games and learning on the learning set of 500 examples for 40 epochs.

For both experiments the same parameters were used :

The Expert Network Parameters

- Learning rate : .3

- Momentum : .0

- Initialization of the weights : between -.2 and .2

- Initial hidden unit neuron sensitivity : 1.0

- Initial output unit neuron sensitivity : 0.25

- Hidden unit neuron sensitivity learning rate : 0.05

- Output unit neuron sensitivity learning rate : 0.001

The Gating Network Parameters

- Learning rate : .1

- Initialization of the weights : between -.1 and .1

- Output neuron sensitivity : 1.0

Design of the Simulations

- TD learning

  - After each played game, examples are constructed and passed to the learning module which uses (extended) back-propagation to alter the weights.

  - The temperature T which is used in equation 4.2 for determining the amount of exploration, was annealed from .005 to .0025.

  - λ was annealed from .3 to .0.

- Supervised learning

  - Examples were learned on-line.

4.3.3   Experimental Results

Experiment 1 : Effect of Architectures

We present separately supervised and TD learning, for the latter we also study the effect of neuron sensitivity. We will first discuss the results of networks with low neuron sensitivities in the activation functions of the hidden units, because they obtained better results than the architectures with steeper activation functions.

Experiment 1.1 : Low Neuron Sensitivity

Table 4.11 shows that when the architectures are trained by supervised learning on 500 examples, the single networks with 5 and 10 hidden units give the best results. The linear network obtains the worst results. The supervised trained single networks with hidden units obtain the best generalization performance on this amount of examples. When a modular architecture is used to decompose the input space, the generalization performance decreases. This is especially shown by the results of the architecture which uses symbolic rules to decompose the input space. In figure 4.7 we can see the learning curves of the architectures when supervised learning is used. We can see that the architecture which uses the symbolic rules is not able to decrease its error after a while during learning. The reason for this might be that the architecture is being overtrained, because the different

Architecture	h.u.	RMS	SD
Single	0	0.189	.008
Single	5	0.107	.007
Single	10	0.103	.010
HME	2*5	0.115	.006
Meta-Pi	2*5	0.111	.018
Symbolic	6*10	0.145	.005

Table 4.11: The performances of the different architectures when supervised learning is used. A supervised learning set consists of 500 examples. This learning set is presented 120 times to the architectures. For each simulation a different learning set is randomly selected from a lookup table created by dynamic programming. Simulations were repeated 10 times. The best RMS after a test was kept as the final result of a simulation.

expert networks contain too many parameters and do not receive enough learning samples. The architecture finds a good approximation for the learning set but generalizes poorly. So when we have to choose an architecture for supervised learning of a smooth evaluation function with a small learning set, it is best to use a small single network.

Architecture	h.u.	RMS	SD
Single	0	0.219	.005
Single	5	0.113	.005
Single	10	0.115	.005
HME	2*5	0.112	.004
Meta-Pi	2*5	0.113	.005
Symbolic	6*10	0.124	.004

Table 4.12: The performances of the different architectures when TD learning is used. One simulation consists of 10,000 games of self-play. Simulations were repeated 10 times. The best RMS after a test was kept as the final result of a simulation.

The results when TD learning is used are shown in table 4.12. For TD learning, the architecture which uses symbolic rules to select an expert network obtains almost the worst performance. The worst performance is obtained by the linear neural network, but this is not very surprising. It is more surprising that the architecture with the largest amount of parameters works worst for TD learning of a smooth evaluation function. Again this architecture obtains worse generalization performance, because the architecture contains too many hidden units to be trained with so few accurate learning samples.

The Meta-Pi, HME and single network architectures work about the same. Because the evaluation function is very smooth and the number of parameters in the architectures
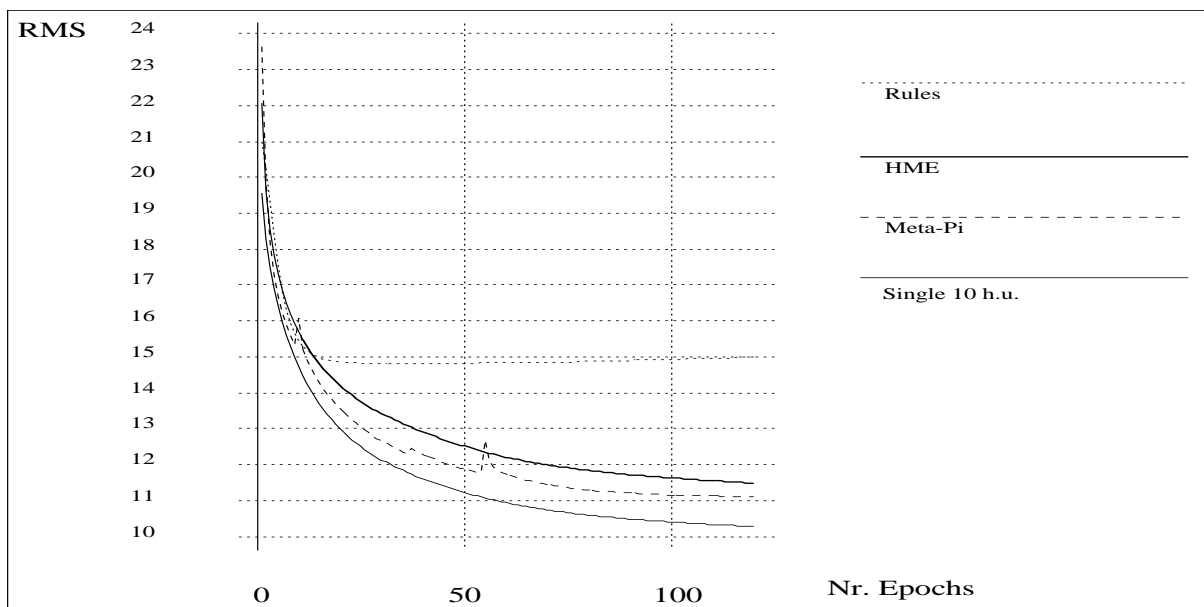
Figure 4.7: The learning curves for the architectures which use supervised learning. The curves are averaged over 10 simulations.

are about equal, it is not surprising that one of the architectures does not generalize better. The small advantage of the performance of the single network with 5 hidden units compared to the single network with 10 hidden units is more surprising. This means that the generalization performance of an architecture is maximal with a low number of parameters. When longer training times are used and more accurate learning samples are shown, this difference will probably change its sign. The learning curves for TD learning are presented in figure 4.8. We can see that for all architectures the RMS error gradually decreases.

When we compare supervised to TD learning with the architectures, we can see that it is to be expected that TD learning will give better and better generalization performance, while supervised learning will end up in overtraining the network on the learning set. After 60,000 iterations, the results are about equal for most architectures. Obtaining a learning set for supervised learning might be very difficult, because we would have to do this by dynamic programming or asking a human expert which is very expensive. So heuristic dynamic programming is a viable alternative for supervised learning when it is difficult to construct a large learning set.
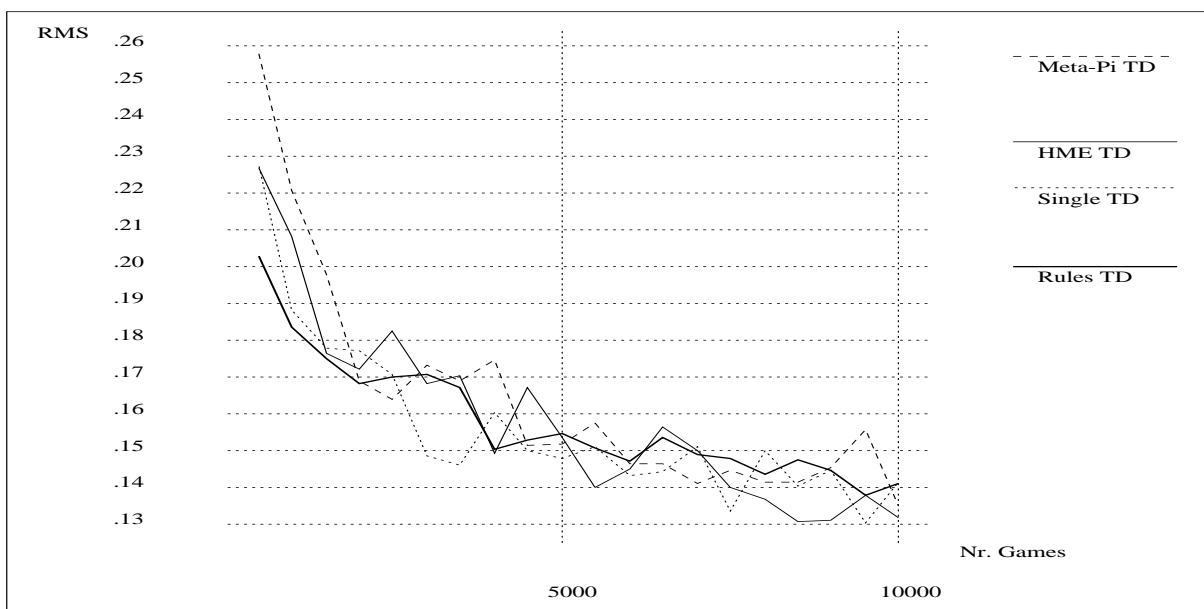
Figure 4.8: The average learning curves for the four architectures trained by TD learning. The simulations were repeated 10 times, one simulation consists of 10,000 games of self-play starting with randomly drawn positions in the endgame with a maximum of 14 against 14 pieces.

Experiment 1.2 : High Neuron Sensitivity

Table 4.13 shows all results of the experiments when TD learning of architectures with initial neuron sensitivities of 3.0 were used. The results show that when using high neuron

Architecture	h.u.	RMS	SD
Single	10	0.143	.004
HME	2*5	0.141	.011
Meta-Pi	2*5	0.144	.007
Symbolic	6*10	0.125	.004

Table 4.13: The performances of the different architectures with high neuron sensitivity when TD learning is applied. Simulations were repeated 10 times. The best RMS after a test was kept as the final result of a simulation.

sensitivities, the symbolic rules architecture obtains the best results. When using high neuron sensitivities, the architectures must consist of many parameters to be able to generalize well. The symbolic rules architecture is the only architecture which obtains the same performance levels when low or high initial neuron sensitivities are used. The other

architectures obtain much better results when using a low number of hidden units with low neuron sensitivity to learn the smooth game evaluation function for the endgame of backgammon.

Experiment 2 : Supervised vs. TD Learning

For this experiment we used longer training times to really compare supervised to TD learning. For all simulations a single network is used with 10 hidden units. We compare its performance on supervised learning on a learning set size of 500 and 5,000 examples, and on TD learning with or without combining supervised learning on 500 examples. Table 4.14 shows all results of the experiments when initial neuron sensitivities of 1.0 are used. The RMS is computed after every 40,000 presented learning samples. The total number of iterations for all methods is 800,000, so the RMS is computed 20 times for each simulation.

Method	nr games or epochs	RMS	SD
500 examples	1,600	0.101	0.006
5,000 examples	160	0.044	0.002
TD learning	130,000	0.080	0.002
TD + 500 examples	65,000 + 800	0.068	0.003

Table 4.14: The results of training a single network with 10 hidden units on supervised learning and TD learning. The method specifies if supervised learning is used, TD learning is used or a mixture of TD and supervised learning is used. The simulations were repeated 5 times.

The results show that using a large example set works best. Overtraining does not occur and the obtained RMS show that the single network is able to accurately approximate the desired game evaluation function. Further training and using larger learning sets will improve the obtained approximation.

We can see in figure 4.9 that with a learning set size of 500 examples overtraining occurs. This problem does not occur with TD learning or the mixture of TD learning and supervised learning. Both methods obtain better results than supervised learning on the small learning set of 500 examples. The mixture of TD learning and supervised learning works better than either one of these methods alone. So it seems that this combination is advantageous. When using TD learning, one could try to construct a small set of learning examples, so that the combination of TD learning and supervised learning can be tried out. In our experiment, the supervised learning set is very accurate, but some researchers have tried to use action replay so that examples acquired by TD learning can be reused [Lin93]. When these examples are known to be accurate, it might be a good idea to store these examples so that a supervised learning set is created.
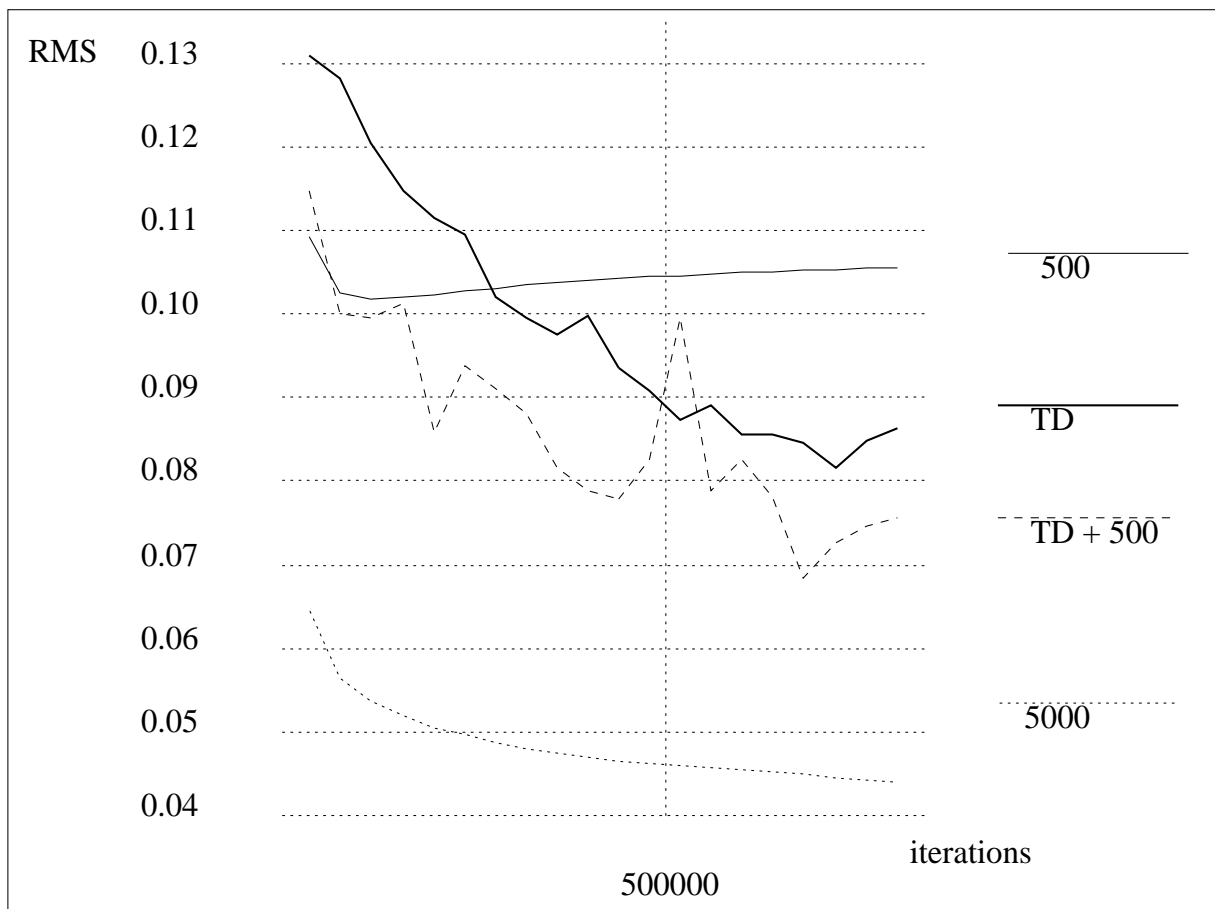
Figure 4.9: The learning curves for a single network with 10 hidden units when longer training times are used. The methods which are studied are : supervised learning on 500 and 5,000 examples, TD learning and a mixture of TD learning and supervised learning on 500 examples. After each 40,000 learning examples the RMS is computed. The learning curves were averaged over 5 simulations.

Finally we studied if a lower error of the approximation of the evaluation function means that the control policy of the agent is better. For this we used the trained architectures from the previous experiment. A tournament between the following architectures with the given RMS error was held

- 5,000 examples architecture : RMS = 0.046

- 500 examples architecture : RMS = 0.100

- TD learning architecture : RMS = 0.080

- Mixture TD + 500 architecture : RMS = 0.070

- Rules which bear-off as many pieces as possible. If different moves take off the same amount of pieces, one is chosen randomly.

Each competition is repeated 5 times in which every time 5,000 games are played. Table

Method player A	Method player B	equity player A	SD
500 examples	5,000 examples	0.014	0.011
500 examples	TD learning	0.002	0.008
500 examples	Mixture TD+500	0.013	0.010
500 examples	Rules	0.012	0.019
5,000 examples	TD learning	0.007	0.010
5,000 examples	Mixture TD+500	-0.006	0.010
5,000 examples	Rules	-0.013	0.017
TD learning	Mixture TD+500	0.005	0.007
TD learning	Rules	0.006	0.011
Mixture TD+500	Rules	0.016	0.004

Table 4.15: The tournament between the single network which is trained by different methods and a simple knowledge base which takes off as many pieces as possible. Each competition consists of 5,000 test games and is repeated 5 times.

4.15 shows that when the differences in the accuracy of the evaluation function are significant, this does not mean that the control policy is also better. It is a big surprise that the architecture which is trained on 500 supervised learning samples wins against the architecture which is trained on 5,000 supervised learning samples. Although the latter architecture approximates the desired evaluation function much more accurate, this does not mean that the control policy is also better. It is possible that the first architecture has better learned to discriminate between possible positions, and that minor differences in the evaluation of different moves are less of a problem for this architecture, because it

has reached its maximal approximation. The other architectures were all able to improve their approximation, and maybe this is why small differences in the evaluation of different positions are more of a problem for these architectures. Look ahead strategies might be a method to exploit the differences in the accuracy of the evaluation function, but we did not research this possibility.

Although higher accuracy of an approximation of an evaluation function does not mean that the control policy of the agent performs better, it is very useful to have an accurate approximation. When we would like to learn the whole game of backgammon, we will use the approximation for the endgame situation to adapt evaluations of positions. When the approximation of the endgame is very accurate, there is no need to play the game further, because reinforcement can be returned when we reach the endgame.

4.3.4 Discussion

The different architectures have been used to learn the game evaluation function for the endgame of backgammon. When supervised learning on a small learning set is used to learn the evaluation function, there is no need to use a modular architecture to learn this part of the game (see table 4.11). The game evaluation function of the endgame of backgammon is very smooth so when an architecture decomposes the input space, the generalization performance only decreases.

When TD learning is used, the HME, Meta-Pi and monolithic architectures obtain the same performance levels, and the symbolic rules architecture obtains a lower performance level (see table 4.12). This latter may be caused, because the architecture uses more experts, so that some experts are trained on a small number of learning samples. The effect of neuron sensitivity was studied and the results show that increasing the initial neuron sensitivity results in lower performance levels, except for the symbolic rules architecture.

We have also compared supervised learning to TD learning with a single network. When a large amount of perfect examples is available, supervised learning on these learning samples obtains the best results. However, when only a small amount of perfect examples is available, this results in overtraining of the network. That is why TD learning works better. When TD learning is applied, the generalization error gradually decreases if more games are being played. This is not surprising, because Dayan has proved that TD learning always converges to at least a local minimum [Dayan94]. We have also studied using a mixture of supervised and TD learning. This method works better than either of these methods alone. So when only a small learning set is available, this may computationally be more effective than constructing a large learning set with dynamic programming. Another method which might be advantageous is to store examples acquired by TD learning, so that these examples can be reused [Lin93].

TD learning of the game evaluation function for the endgame of backgammon is reasonable efficient. When a large amount of training games are played, the architecture

can reach a high level of precision. One surprising result is that when an architecture better approximates the game evaluation function than another architecture, this does not mean that the architecture also plays better (see table 4.15). However, it is very important to have a very precise approximation of the game evaluation function for the endgame, because this approximation may be used to return reinforcement for learning stages before the endgame. By using this kind of hierarchical learning [Lin93], there is no need to play the game any further, so that the learning process will become faster.

Chapter 5

Conclusion

5.1 Discussion

We have studied learning game evaluation functions with modular neural network architectures. Game evaluation functions usually contain many discontinuities, which makes them difficult to learn. To make the learning process faster and the acquired approximation more accurate, we studied using multiple expert networks which are located in subspaces of the total input space. We have described two learning algorithms to divide the input space in subspaces. The first is the hierarchical mixtures of experts (HME) methodology in which a likelihood function of generating the desired outputs is maximized, so that expert networks are located in regions where they outperform the other experts. The second methodology uses a Meta-Pi gating network to learn to locate expert networks where they can help to minimize the error of the architecture as a whole. These architectures are compared with monolithic architectures and architectures which use a priori knowledge to divide the input space in non-overlapping regions.

All architectures have been studied on learning a simple discontinuous function. The results show that using the HME architecture with a winner takes all selection strategy of the experts, and the architecture which uses symbolic rules to divide the input space at the discontinuity, outperform the other architectures.

In our work we have used an extension of back-propagation, which is able to adapt neuron sensitivities. An activation function of a neuron multiplies the neuron sensitivity with the input, after which the output is computed as normal. We have studied initializing the neuron sensitivities on high values, so that the activation functions become steeper. The results of using low and high neuron sensitivities indicate a big advantage of using hidden units with high neuron sensitivities when learning a discontinuous function (see section 3.4.2). The results also show that using the learning rule extended back-propagation which learns neuron sensitivities results in faster learning compared to normal back-propagation.

The methodologies are used to learn to play the games of tic-tac-toe and the endgame of backgammon. We have described how temporal difference (TD) learning can be used to generate learning samples from played games. We have seen that TD learning is an efficient way to learn a control policy for an agent. The obtained results on learning the discontinuous game evaluation function of tic-tac-toe show that by using architectures which contain many adjustable parameters and high neuron sensitivities, we can get close to the maximal performance level of playing tic-tac-toe against a fixed opponent. When more parameters are used in an architecture, the performance improves. Using multiple expert networks is an efficient way to use many parameters, because we can select independent neural networks for evaluating different moves and positions, which is much faster than always invoking one large single network.

Experiments with the endgame of backgammon show that TD learning is a viable alternative for supervised learning when only a small amount of training examples is available. Overtraining of the architectures does not occur with TD learning, and it is to be expected that the generalization error will gradually decrease. A combination of supervised learning and TD learning obtains better results than using one of these learning paradigms alone. For supervised learning on a small example set, it was not advantageous to use modular neural network architectures. This can be explained by the fact that the evaluation function is very smooth in this part of the game. For learning a smooth evaluation function, the use of modular architectures and high neuron sensitivity is not advantageous. However, when learning a discontinuous game evaluation function, the opposite is true.

5.2   Prospects and Future Work

The recent interest in neural networks and reinforcement learning might make an enormous contribution to machine learning. When computers are getting larger and faster, new challenging domains could be conquered. We want to make a contribution to this research by showing how accurate game evaluation function can be approximated, and how modular architectures can be applied to speed up the learning process.

For this we want to use the game of backgammon. The state space of backgammon will be divided in some non-overlapping subspaces by symbolic rules, which results in the fastest architecture. These symbolic rules will be acquired by a short knowledge engineering period. Hereafter, many expert neural networks and TD learning will be used to learn to evaluate positions which fall in the different classes.

When this learning process is in an advanced state, we will study how the architecture can be combined with the HME and Meta-Pi architectures, CMACS or fuzzy logic. This could improve the generalization performance which results in a smooth evaluation function. To save time, we will look at ways to improve the generalization performance without having to invoke too many experts at the same time.

Finally, we want to study the use of modular architectures and high neuron sensitivities by researching TD learning of game evaluation functions of games like draughts, checkers and chess, because these games contain a large number of discontinuities. We hope that this research will be performed so that in future more difficult games can be solved, and the expressive power of large neural network architectures can be really evaluated.

Bibliography

[Aarts89]   E.H.L. Aarts & Jan Korst. *Simulated Annealing and Boltzmann Machines.* Wiley, Chichester, 1989.

[Anthony91]   M. Anthony. *Uniform Convergence and Learnability*. PhD thesis, University of London, 1991.

[Berliner77]   H. Berliner. Experiences in evaluation with BKG - a program that plays backgammon. *Proceedings of IJCAI*, (428-433), 1977.

[Boyan92]   J. Boyan. *Modular Neural Networks for Learning Context-Dependent Game Strategies.* Thesis report B.S. , University of Chicago, 1992.

[Cybenko89]   G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Math. Control Signals Systems*, 2, (303-314), 1989.

[Dayan92]   P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8, (341-362), 1992.

[Dayan94]   P. Dayan & T.J. Sejnowski. TD(λ) Converges with Probability 1. *Machine Learning*, 14, (295-301), 1994.

[Esposito93]   F. Esposito, D. Malerba & G. Semeraro. Decision Tree Pruning as a Search in the State Space. In P. B. Brazdil (ed.), *Proceedings of the 1993 European Conference on Machine Learning*, (166-184), Vienna, 1993.

[Fox91]   D. Fox, V. Heinze, K. Möller, S. Thrun & G. Veenker. Learning by error-driven decomposition. In T.Kohonen, K. Mkisara, O. Simular & J. Kangas (eds.), *Proceedings of the 1991 International Conference on Artificial Neural Networks*, (207 - 212), Amsterdam, North-Holland, 1991.

[Gruau92]   F. Gruau. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In L.D. Whitley & J.D. Schaffer (eds.), *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, (55-72), Baltimore, MD: IEEE, June 1992.

70

[Hakala94]   J. Hakala, C. Koslowski & R. Eckmiller. 'Partition of Unity' RBF Networks are Universal Function Approximators. *Neural Networks*, 5, (459-462), 1994.

[Hampshire89] J.B. Hampshire & A. Waibel. *The Meta-Pi network: Building distributed knowledge representations for robust pattern recognition.* Tech. Report CMU-CS-89-166, Carnegie Mellon University, August 1989.

[Hashem93]   S. Hashem. *Optimal Linear Combinations of Neural Networks.* PhD thesis, Tech. Report SMS 94-4. Purdue University, December 1993.

[Jacobs91]   R.A. Jacobs, M.I. Jordan, S.J. Nowlan & G.E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1), 1991.

[Jordan92]   M.I. Jordan & R.A. Jacobs. Hierarchies of adaptive experts. In J. Moody, S. Hanson & R. Lippmann (eds.), *Advances in Neural Information Processing Systems*, 4, (985-993), San Mateo, CA: Morgan Kaufmann, 1992.

[Jordan93]   M.I. Jordan & R.A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. Submitted to *Neural Computation*, Tech. Rep. 9301, April 1993.

[Judd90]   J.S. Judd. Neural Network Design and the Complexity of Learning. The MIT press, Cambridge, 1990.

[Krose92]   B.J.A. Kröse & P. van den Smagt. An Introduction to Neural Networks. July 1992.

[Lin93]   L.J. Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, Tech. report CMU-CS-93-103, Carnegie Mellon University, Pittsburgh, January 1993.

[Nadi91]   F. Nadi. Topological Design of Modular Neural Networks. In T. Kohonen, K. Mäkisara, O. Simular & J. Kangas (eds.), *Proceedings of the 1991 International Conference on Artificial Neural Networks*, (213 - 218), Amsterdam, North-Holland, 1991.

[Nowlan91]   S.J. Nowlan. *Soft Competitive Adaption: Neural Network Learning Algorithms based on Fitting Statistical Mixtures.* PhD thesis, Tech. report CMU-CS-91-126, Carnegie Mellon University, Pittsburgh, April 1991.

[Rumelhart86] D.E. Rumelhart, G.E. Hinton & R.J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart & J.L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1, Chapter 8, The MIT press, 1986.

[Samuel59]   A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, (210-229), 1959.

[Samuel67]   A. Samuel. Some studies in machine learning using the game of checkers: II - recent progress . *IBM Journal of Research and Development*, 11, (601-617), 1967.

[Schaffer92]   J.D. Schaffer, D. Whitley & L.J. Eshelman. Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art. In L.D. Whitley & J.D. Schaffer (eds.), *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, (1-37), Baltimore, MD: IEEE, June 1992.

[Schraudol94]   N.N. Schraudolph, P. Dayan & T.J. Sejnowski. Temporal difference learning of Position evaluation in the Game of Go. In J.D. Cowan, G. Tesauro & J. Alspector (eds.), *Advances in Neural Information Processing*, 6, Morgan Kaufmann, San Fransisco, 1994.

[Simon92]   N. Simon, H. Corporaal & E. Kerckhoffs. Variations on the Cascade-Correlation learning architecture for fast convergence in robot control. *Neuro Nimes*, 5, (455-464), 1992.

[Sperduti92]   A. Sperduti. *Speed Up Learning and Network Optimization With Extended Back Propagation*. Tech. report TR-10/92, University of Pisa, May 1992.

[Sutton88]   R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, (9-44), 1988.

[Tesauro89]   G. Tesauro. Neurogammon : A neural network backgammon learning program. In D.N.L. Levy & D.F. Beal, (eds.), *Heuristic Programming in Artificial Intelligence : The first Computer Olympiad*, Chichester, England, Ellis Horwood lim, 1989.

[Tesauro92]   G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4), (257-277), Kluwer Academic Publishers, May 1992.

[Thrun92]   S.B. Thrun. *Efficient Exploration in Reinforcement Learning*. Tech. Report CMU-CS-92-102, Carnegie Mellon University, Pittsburgh, 1992.

[Tresp93]   V. Tresp, J. Hollatz & S. Ahmad. Network structuring and training using rule-based knowledge. *Advances in Neural Information Processing Systems*, 5, (871-878), 1993.

[Vysniausk93] V. Vyšniauskas, F.C.A. Groen & B.J.A. Kröse. The optimal number of learning samples and hidden units in function approximation with a feedforward network. Tech. Report CS-93-15, University of Amsterdam, November 1993.

[Watkins92] C.J.C.H. Watkins & P. Dayan. Q-Learning, *Machine Learning*, 8, (279-292), 1992.

[Whitehead92] S.D. Whitehead. *Reinforcement Learning for the Adaptive Control of Perception and Action*. PhD thesis, University of Rochester, February 1992.

Appendix A

Temporal Difference Learning

A.1   TD(λ)-methods

Temporal difference (TD) methods [Sutton88] are a class of learning procedures specialized for prediction problems. In prediction problems, an observation or state vector is used to predict the final outcome with e.g. a neural network. So we can predict the outcome of a game when we see a board position, and use this prediction as the evaluation of the position. With such an evaluation function, we can perform tasks by comparing the evaluations of the states which result from taking all possible actions in the current state. So we want to use a neural network to model an evaluation function V given by

$$V(x_t) = E(r|x_t)$$

with :
x_t : the state or observation at time t.
$E(r|x_t)$: the expected outcome or result r of the game when we are in state x_t.

We would like to learn this model with a neural network which is based on minimizing the error over a learning set. We can generate a learning set by playing a game with our network. Weights are adjusted according to an error measure E_t. We can define two possible ways to define the error measure. We might use a supervised learning procedure which defines the error as the difference between $V(x_t)$ (the predicted outcome) and r (the actual outcome)

$$E_t = \frac{1}{2}(r - V(x_t))^2$$

The total error E of a played game is

$$E = \sum_{t=1}^{M} E_t$$

74

We minimize the total error by gradient descent with learning rate α

$$\Delta W = \sum_{t=1}^{M} -\alpha \frac{\partial E_t}{\partial W_t} = \sum_{t=1}^{M} -\alpha \frac{\partial E_t}{\partial V(x_t)} \frac{\partial V(x_t)}{\partial W_t}$$

We may write this as

$$\Delta W = \sum_{t=1}^{M} \alpha(r - V(x_t)) \nabla_w V(x_t) \tag{A.1}$$

in which the gradient $\nabla_w V(x_t)$ is computed by the back-propagation procedure. This supervised paradigm compares the evaluations of the state vectors with the actual out-come of the experience, but does not use the differences between temporally successive predictions. Thus, learning is not possible when the result of the game is unknown.

Unlike the previous method which compares $V(x_t)$ with r, there are TD methods which are driven by the error or difference between temporally successive predictions $V(x_{t+1}) - V(x_t)$. This makes learning possible when the result is (still) unknown. We can construct a TD method which makes the same updates as equation A.1 by rewriting the error $r - V(x_t)$ as

$$r - V(x_t) = \sum_{k=t}^{M} (V(x_{k+1}) - V(x_k)) \tag{A.2}$$

Where $V(x_{M+1}) = r$. Now by using A.2, we can rewrite A.1 as

$$\begin{aligned}\Delta W &= \sum_{t=1}^{M} \alpha \sum_{k=t}^{M} (V(x_{k+1}) - V(x_k)) \nabla_w V(x_t) \\ &= \sum_{t=1}^{M} \alpha(V(x_{t+1}) - V(x_t)) \sum_{k=1}^{t} \nabla_w V(x_k)\end{aligned}$$

We can convert this rule to an incremental update rule

$$\Delta W_t = \alpha(V(x_{t+1}) - V(x_t)) \sum_{k=1}^{t} \nabla_w V(x_k) \tag{A.3}$$

Equation A.3 is the supervised or TD(1) rule. The advantage of TD(1) compared to A.1 is that the computations to perform back-propagation are spread out over time. However it does not really use the information in successive predictions either. The advantage of the TD-methods is that they can use the information contained in intermediate predictions and do not rely too much on the actual outcome. When the difference $V(x_{t+1}) - V(x_t)$ is very large, this difference is in equation A.3 used to equally adjust the evaluation of states $x_1,\ldots,x_t$. TD(λ) methods make greater alterations to more recent states by weighting the recency exponentially with λ. The general TD(λ) algorithm has the following form

$$\Delta W_t = \alpha(V(x_{t+1}) - V(x_t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V(x_k)$$

with :
ΔW_t : the adaptions of the weights of the network at time t.
$V(x_t)$: the network's evaluation of state x_t.
α : the learning rate.
$0 \leq \lambda \leq 1$: the discount factor which is used to weight TD(0) errors exponentially by recency.

The case $\lambda = 1$ corresponds to the supervised pairing of each input pattern with the final reward signal r $(= V(x_{M+1}))$. The case $\lambda = 0$ corresponds to an explicit pairing of each input pattern x_t with the next prediction $V(x_{t+1})$. In this case the difference $V(x_{t+1}) - V(x_t)$ is used to direct the network through the weight space. We will call this difference the TD(0) error.

   Until now we have seen that temporal difference learning can be used after a sequence of M actions which results in a final reinforcement. It can also be used for other classes of problems. A class which generalizes the previous class is when actions a_t from nonterminal states x_t are allowed to return reinforcement $r(x_t, a_t)$. When we allow non-absorbing goal states, we might want the evaluation $V(x_t)$ of a state to approximate the expected discounted cumulative reward V_t when we start in state x_t

$$V_t = E(\sum_{k=0}^{\infty} \gamma^k r(x_{t+k}, a_{t+k}) | x_t)$$

Where γ is the discount factor and determines how much the agent has to aim at immediate and future rewards. For this class of problems we can write down the following recursive equation [Sutton88]

$$V(x_t) = r(x_t, a_t) + \gamma V(x_{t+1})$$

The difference between both sides is the TD(0) error : $r(x_t, a_t) + \gamma V(x_{t+1}) - V(x_t)$. We use this TD(0) error to construct the following incremental TD learning rule

$$\Delta W_t = \alpha(r(x_t, a_t) + \gamma V(x_{t+1}) - V(x_t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w V(x_k)$$

Example : What is the effect of λ ?

Suppose we have taken two actions which results in a win, the starting position is x_1 and white's evaluations of the positions are :
$V(x_1) = 0.6$ $V(x_2) = 0.4$ $V(x_3) = 0.8$.
For notational purposes we will define the final reinforcement r as $V(x_4) = 1.0$.

$\lambda = 1$ gives :
$\Delta W_1 = \alpha$ * (0.4 - 0.6) * $\nabla_w V(x_1)$
$\Delta W_2 = \alpha$ * (0.8 - 0.4) * $(\nabla_w V(x_1) + \nabla_w V(x_2))$
$\Delta W_3 = \alpha$ * (1.0 - 0.8) * $(\nabla_w V(x_1) + \nabla_w V(x_2) + \nabla_w V(x_3))$
This sums up to :
$\Delta W = \alpha$ * $((1.0 - 0.6)$ * $\nabla_w V(x_1) + (1.0 - 0.4)$ * $\nabla_w V(x_2) + (1.0 - 0.8)$ * $\nabla_w V(x_3))$
which is a supervised pairing of each prediction with the actual outcome of the experience.

$\lambda = 0$ gives :
$\Delta W_1 = \alpha$ * (0.4 - 0.6) * $\nabla_w V(x_1)$
$\Delta W_2 = \alpha$ * (0.8 - 0.4) * $\nabla_w V(x_2)$
$\Delta W_3 = \alpha$ * (1.0 - 0.8) * $\nabla_w V(x_3)$
Which associates each observation vector with only the next time-step's prediction.

$\lambda = 0.5$ gives :
$\Delta W_1 = \alpha$ * (0.4 - 0.6) * $\nabla_w V(x_1)$
$\Delta W_2 = \alpha$ * (0.8 - 0.4) * $(0.5$ * $\nabla_w V(x_1) + \nabla_w V(x_2))$
$\Delta W_3 = \alpha$ * (1.0 - 0.8) * $(0.25$ * $\nabla_w V(x_1) + 0.5$ * $\nabla_w V(x_2) + \nabla_w V(x_3))$
This sums up to :
$\Delta W = \alpha$ * $(0.05$ * $\nabla_w V(x_1) + 0.5$ * $\nabla_w V(x_2) + 0.2$ * $\nabla_w V(x_3))$
which lies in between the two extremes.

A.2   Markov Decision Processes

We want to learn a control policy which maximizes an agent's performance level. The control policy has to be able to differentiate between possible actions in a particular state and to choose the action with the highest merit. Like [Whitehead92], we will use Markov decision processes to define a mathematical framework with which we can describe the task. The Markov property states that a description of the current state is sufficient to choose between the possible actions so that the agent's performance level is maximized. The Markov property is violated when an action depends on actions which have been performed earlier (e.g. when a box might contain a banana, because the agent has put a banana in the box, but is not able to see this in the current state). For such tasks, re-current networks have been proposed [Lin93], but in this work we concentrate on playing games and the Markov property holds. We will also concentrate on deterministic worlds, so an action in a particular state always results in the same next state (the transition probability function of states and actions is a true function). The discussion can be extended to non-deterministic world by taking the probabilities of multiple action outcomes into consideration.

We will first define the Markov decision process :

x_t : the state in which the agent is at time t.

a_t : the action the agent makes at time t.

$A(x_t)$: the set of possible actions in state x_t.

$T(x_t, a_t)$: the state x_{t+1} which results from performing action a_t in state x_t.

$r(x_t, a_t)$: the reinforcement which is emitted when the agent makes action a_t in state x_t.

$\Pi(x_t)$: the action the agent will select when it is in state x_t and follows the control policy Π.

In TD learning we learn an evaluation function of states (AHC-learning) or an evaluation function of state-action pairs (Q-learning). The control policy can use this evaluation function to choose an action in a particular state. To learn the evaluation function, simulations with the agent are performed, so that the agent can evaluate its own actions and the states it was in. When the agent has repeatedly tried all state-action pairs, she can learn which actions are expected to have the highest merit in each state. For trying out all state-action pairs, the control policy is usually not strictly followed while learning. Instead some kind of exploration strategy is used, in which the action which is expected to have the highest merit has the largest probability of being chosen. This means that $a_t = \Pi(x_t)$ is not always valid.

A.3   AHC-learning

AHC-learning is used to learn an evaluation function (V-function) of states. We want the evaluation $V(x_t)$ of a state to approximate the discounted cumulative reward V_t when we start in state x_t

$$V_t = E(\sum_{k=0}^{M-t} \gamma^k r(x_{t+k}, \Pi(x_{t+k}))|x_t) \tag{A.4}$$

Where γ is the discount factor and determines how much the agent has to aim at immediate and future rewards. For learning a game evaluation function, we set γ to -1 which expresses the fact that the evaluation of a position for player 1 is the negative of the evaluation of a succeeding position for player 2. For such classes of problems, the reinforcement $r(x_M, a_M)$ or r which is returned when the last move is played, is the only reinforcement we can use. When an optimal V-function has been learned and the control policy Π is followed throughout the future, the V-function must satisfy

$$V(x_t) = r(x_t, \Pi(x_t)) + \gamma V(x_{t+1}) \tag{A.5}$$

When the V-function is not optimal, the difference between both sides in equation A.5 can be used to change the V-function. This difference or TD(0)-error is defined by

$$E_t^0 = [r(x_t, a_t) + \gamma V(x_{t+1})] - V(x_t)$$

Where we have defined $V(x_{M+1}) = 0$. We will use [Lin93] to define the TD(λ)-error as

$$E_t^\lambda = \sum_{k=0}^{M-t} (\gamma\lambda)^k E_{k+t}^0 \tag{A.6}$$

Now we define

$$V'(x_M) = r(x_M, a_M)$$
$$V'(x_t) = V(x_t) + E_t^\lambda$$

This can be rewritten as

$$V'(x_M) = r(x_M, a_M)$$
$$V'(x_t) = r(x_t, a_t) + \gamma[(1-\lambda)V(x_{t+1}) + \lambda V'(x_{t+1})]$$

$V'(x_t)$ is the estimated evaluation for state x_t. We can create the following set of examples: $\{(x_1, V'(x_1)), \ldots, (x_M, V'(x_M))\}$ and present these to the back-propagating module to minimize the TD(λ)-errors.

A.4   Q-learning

Q-learning learns an evaluation of state-action pairs. We want the Q-value $Q(x_t, \Pi(x_t))$ to approximate the discounted cumulative reward in equation A.4. When an optimal Q-function has been learned, and the control policy Π chooses the action which results in the maximal Q-value, the Q-function must satisfy

$$Q(x_t, a_t) = r(x_t, a_t) + \gamma Max\{Q(x_{t+1}, a_{t+1}) | a_{t+1} \in A(x_{t+1})\}$$

We can rewrite this as follows

$$Q(x_t, a_t) = r(x_t, a_t) + \gamma Q(x_{t+1}, \Pi(x_{t+1})) \tag{A.7}$$

When the Q-function is not optimal, the difference between both sides in equation A.7 can again be used to change the Q-function. The TD(0)-error is now defined by

$$E_t^0 = [r(x_t, a_t) + \gamma Q(x_{t+1}, \Pi(x_{t+1}))] - Q(x_t, a_t)$$

The TD(λ)-error can be defined as in equation A.6. We define

$$Q'(x_M, a_M) = r(x_M, a_M)$$
$$Q'(x_t, a_t) = Q(x_t, a_t) + E_t^\lambda$$

$$\Longleftrightarrow$$

$$Q'(x_M, a_M) = r(x_M, a_M)$$
$$Q'(x_t, a_t) = r(x_t, a_t) + \gamma[(1 - \lambda)Q(x_{t+1}, \Pi(x_{t+1})) + \lambda Q'(x_{t+1}, a_{t+1})]$$

$Q'(x_t, a_t)$ is the estimated evaluation for performing action a_t in state x_t. We can present the set of examples $\{(x_1, Q'(x_1, a_1)), \ldots, (x_M, Q'(x_M, a_M))\}$ to the back-propagating module to minimize the TD(λ)-errors.

Appendix B

Extended Back-propagation

Extended back-propagation [Sperduti92] is a learning rule which introduces a new parameter to speed up the learning process of normal back-propagation and to make the network find learning rates for individual neurons itself. Neuron sensitivity is used in the activation function $F_i(x)$, e.g. a sigmoid, in the following way

$$a_i = F_i(i_i) = \frac{1}{1 + e^{-i_i \beta_i}}$$

So the activation a_i of a unit depends on the weighted sum over its input i_i multiplied with its neuron sensitivity β_i. When the sensitivity of a neuron is very low, its activation will always vary around the same value so that the neuron is less important for the overall network and it will adapt itself slower. Sperduti found the following learning rule for adapting the neuron sensitivities by using gradient descent

$$\Delta \beta_i = \alpha i_i \delta_i \qquad \text{(B.1)}$$

With α as the learning rate for the neuron sensitivity and δ_i the gradient which is computed by back-propagation (see equations 3.3 and 3.4).

Proof :

The error function which is to be minimized is the mean squared error over the outputs

$$E = \sum_j \frac{1}{2} (d_j - a_j)^2$$

By using gradient descent we can change the neuron sensitivities. We can define the neuron sensitivity update rule as

$$\Delta \beta_i \;\; = \;\; -\frac{\alpha}{2} \sum_j \frac{\partial (d_j - a_j)^2}{\partial \beta_i}$$

$$= \alpha \sum_j \frac{\partial a_j}{\partial \beta_i}(d_j - a_j) \tag{B.2}$$

1) For an *output* unit the learning rule follows from the steps:

From equation B.2 and because a_i only depends on β_i

$$\Delta\beta_i = \alpha \frac{\partial a_i}{\partial \beta_i}(d_i - a_i) \tag{B.3}$$

A simple derivation gives

$$\frac{\partial a_i}{\partial \beta_i} = i_i F'_i(i_i) \tag{B.4}$$

Substituting equation B.4 in B.3 gives

$$\Delta\beta_i = \alpha i_i F'_i(i_i)(d_i - a_i)$$

When we use the fact that back-propagation computes $\delta_i = F'_i(i_i)(d_i - a_i)$ (see equation 3.3) we get the learning rule B.1.

2) For a *hidden* unit the learning rule follows from the steps:

First we use the back-propagation chain rule and change indices to define equation B.2 as

$$\Delta\beta_k = \alpha \sum_i F'_i(i_i)(d_i - a_i) \sum_j w_{ij} \frac{\partial a_j}{\partial \beta_k}$$

Because a_k only depends on β_k

$$\Delta\beta_k = \alpha \sum_i F'_i(i_i)(d_i - a_i)w_{ik} \frac{\partial a_j}{\partial \beta_k} \tag{B.5}$$

Substituting B.4 in B.5 gives

$$\Delta\beta_k = \alpha \sum_i F'_i(i_i)(d_i - a_i)w_{ik}i_k F'_k(i_k)$$

Again we make use of the fact that back-propagation has computed $\delta_k = F'_k(i_k) \sum_i \delta_i w_{ik}$ and this results again in the learning rule B.1.

The steepness parameters are adjusted after the weight changes. When the network has been loading examples for a long time and is not able to learn them, the sensitivity of the output neurons will gradually decrease. Sperduti showed that a method to prune hidden units away can be easily integrated in the architecture. Neurons with a very low neuron sensitivity can be pruned away, because this hardly changes the overall performance.

Appendix C

Perfect Performance against TTT

The maximal obtainable performance level against the knowledge base which plays tic-tac-toe (see section 4.2) is about 0.614. In the following we will call this opponent TTT. The maximal performance level against TTT can be computed in two steps by forward dynamic programming:

C.1 The Agent Begins the Game

- When the agent begins, she must play in a corner to maximize the probability that TTT will make a mistake.
- TTT plays a random move : If this move is not in the middle (see figure C.1), then the agent can win. The chance that TTT will not play in the middle is 7/8. This means that the expected payoff $E(r|x_t) = 7/8$.

- if TTT plays in the middle then the agent must play in the corner of the same diagonal

X	.	.
.		.
.	.	.

Figure C.1: X wins when TTT plays at . $\implies E(r|x_T) = 7/8$.

to maximize her winning probability (see figure C.2). X wins when TTT plays at . $\implies$ Probability = 1/3

$\Longrightarrow$ The expected payoff when the agent begins the game:
7/8 * 1 + 1/8 * 1/3 * 1 = 11/12

C.2   TTT Begins the Game

When TTT (O) begins the game, there are three possible ways :

- Possibility 1 (see figure C.3), TTT plays in a corner and the agent must play in the opposite corner.

- When TTT plays at . he will lose the game.

- When TTT plays at a field with 1/5 in it, this means that the agent can play a move after which she has an expected game-result of 1/5.

- When TTT plays on an empty field the result of the game is maximal a draw.

$\Longrightarrow$ equity = 2/7 + 2/7 * 1/5 = 12/35

- Possibility 2 (see figure C.4), TTT plays in the middle of a line, but not in the centre. The agent plays in the corner aligned to white's piece.
$\Longrightarrow$ equity = 2/7 + 2/7 * 1/5 = 12/35

- Possibility 3 (see figure C.5), TTT plays in the centre.
$\Longrightarrow$ equity = 2/7 * 1/5 = 2/35

$\Longrightarrow$ expected payoff for the agent when TTT begins : 4/9 * 12/35 + 4/9 * 12/35 + 1/9 * 2/35 = 98/315

C.3   Total Equity

The total expected payoff for the agent = 1/2 * 11/12 + 1/2 * 98/315 = 1547/2520, 1547/2520 $\approx$ 0.614

X		.
	O	
.		X

Figure C.2: X wins when TTT plays at . $E(r|x_t) = 1/3$.

O	.	
.		1/5
	1/5	X

Figure C.3: Possibility 1 : TTT plays in a corner, the agent plays in the opposite corner. $E(r|x_t) = 12/35$.

X	O	.
	1/5	.
		1/5

Figure C.4: Possibility 2 : TTT plays in the middle of a line, but not in the centre. The agent must play in an aligned corner for maximizing the expected payoff. $E(r|x_t) = 12/35$.

X	1/5	
1/5	O	

Figure C.5: Possibility 3 : TTT plays in the centre, the agent plays in a corner to maximize the expected payoff. $E(r|x_t) = 2/35$.