Adaptation and Learning

A Comparison of AI and Control Views

Larsson, Jan Eric

1993

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280–5316
ISRN LUTFD2/TRFT--7505--SE

Adaptation and Learning

A Comparison of AI and Control Views

Jan Eric Larsson

An Essay for the Course in Adaptive Control
Department of Automatic Control, Lund Institute of Technology
April 1993

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Internal report

Date of issue
April 1993

Document Number
ISRN LUTFD2/TFRT--7505--SE

Author(s)
Jan Eric Larsson

Supervisor
Karl Johan Åström, Björn Wittenmark

Sponsoring organisation

Title and subtitle
Adaptation and Learning—A Comparison of AI and Control Views

Abstract

This report contains an essay written for the course in Adaptive Control given by Björn Wittenmark at the Department of Automatic Control in Lund during the spring of 1993.

The point of the essay is to describe some classical learning projects in artificial intelligence and to interpret them in automatic control terms. The projects treated are Samuel's Checkers program, Hsu's estimation of evaluation function parameters in Chess, Tesauro's and Sejnowski's neural network for Backgammon, and Michie's and Chambers' BOXES model for balancing a pole on a cart. In addition, a short overview on neural networks in general is given.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
29

Recipient's notes

Security classification

1. *Introduction*

Once upon a time the research area of *Cybernetics* was born. In the beginning it comprised several subjects, among other both artificial intelligence and control theory. But after some years, these two fields became more and more estranged, and nowadays they form two completely separate schools of research.

There are, however, points of strong common interest. One of these is definitely *learning,* maybe the most profound area of AI and at the same time one of the least successful. Some versions of learning are quite similar to *adaptive control,* a well established area of control theory.

This report will describe some common points of AI learning and adaptive control in the area of parameter adjustment, and it will interpret the algorithms developed in AI in the terms of control theory. It was written for the course "Adaptive Control," given by Björn Wittenmark at the Department of Automatic Control, Lund Institute of Technology, Lund, in the spring of 1993.

First, AI learning and adaptive control will be described. After that, some different learning projects will be interpreted from an adaptive control viewpoint.

2. *What is Learning?*

One of the most intriguing research areas of artificial intelligence is *learning.* It is a common opinion that computers cannot be called intelligent until they are able to learn to do new things and to adapt to new situations. This is certainly wrong, as many intelligent behaviors do not include any learning aspect. In spite of this, learning *is* an important part of AI, and maybe also one of the areas that have had the least success so far.

It is difficult to define learning, (more difficult than to define adaptation, as we shall see below). A general definition is that learning is change of behavior in a given situation brought about by the repeated experiences of such situations. But such a definition is much too general to be useful, and it would include several computer programs that are not normally referred to as learning programs.

As is often the case in AI, learning is best defined by examples of what is done in the research area. At the same time, learning is a vast field which is difficult to cover. Thus, the following overview only serves to give some examples of the different subjects:

- *Rote learning.* This is the simplest kind of machine learning, and it simply means to store large quantities of data and using these data to permit a learning behavior. Once a result has been computed, it is stored in a large table and used again instead of new calculations. A well known example of this is the first of Samuel's Checkers programs, see Samuel (1963). This program used a large database of

previously encountered positions to enhance its evaluations. If a previously stored position was found during the tree search, its value was used, instead of the evaluation function or deeper search. When a significant part of the leaf-nodes could be found in the table, it gave the same effect as that of a much deeper search.

- *Parameter adjustment.* A large variety of programs rely on the procedure of weighting several features together into a single summary value. In such cases it is sometimes possible to begin with an estimate of the best value and then slowly adjust the weighting parameters according to some loss function, so that they reach or come closer to the best value. An example of this is the second of Samuel's Checkers programs. This program played games against itself and adjusted the weighting parameters of the evaluation function according to the outcome of the played games. The results were, for that time, quite impressing, and Samuel's second program is sometimes viewed as the most successful example of AI learning. The program managed to win a game against a human Checkers master. Another good example of parameter adjustment is the training of neural networks by backpropagation or other algorithms. A neural network approach has also been used in a program to play Backgammon, see Tesauro and Sejnowski (1989).

- *The General Problem Solver.* The GPS program, see Newell and Simon (1963), has been used for learning. GPS is really a sort of inference engine, using a data structure where states are represented, together with operators to change these states. One state is the initial state and some states are goal states, and GPS applies the operators in turn, (based on a difference table), to reach from the initial to a goal state. One way of using learning in GPS is to let the program try and solve several similar problems while gathering statistical information in order to construct the difference tables by itself. Another way is to view the learning task as general problem solving, and describe the initial and goal states, and the operators of learning itself in GPS, and then let the program perform the learning task. In this way, learning is equated with problem solving. This solution demands full knowledge of the learning task, though, and provides a good example of the principle that in order to learn something, one must already know a lot.

- *Dynamic Programming.* The well established technique of dynamic programming can be used in learning tasks. In Bellman (1961) such a method is used to solve the *two-armed bandit problem,* where a gambler tires to optimize his winnings when playing two slot machines. The winning probability of one of the machines is not known. Thus, dynamic programming is used to find the optimum between gathering information that will later give a better strategy, and "playing it safe" by using the current knowledge and simply making the optimal choice at every instant. This is a typical example of the fact that

testing to gather more information, or in other words, to excite the process, may be suboptimal in a short perspective but valuable in the long run.

- *Concept learning.* In many AI programs, it is important to classify objects, in order to apply reasoning rules about them. It may, however, be difficult to devise the classification mechanism. For this, learning has been used. A good example is Winston's learning *blocks world* program. This program uses a set of primitive graphical concepts, such as lines and points, and tries to invent new structural concepts, characterized by the presence of certain subsets of basic concepts. Thus, such concepts as archs, bridges, and houses may be learned, see Winston (1975). Classification can also be done by linear weighting by a scoring polynomial, and in this case, concept learning by parameter adjustment may be possible.

  The learning of concepts from a set of examples has also been treated by Haussler (1988), and Haussler *et al* (1991). Here, Valiant's learnability model and learning framework, see Valiant (1984), is used and Haussler arrives at necessary and sufficient conditions for learning to be possible. This is determined by the finiteness of the *Vapnik-Chervónenkis dimension*, a simple combinatorial parameter of the class of concepts to be learned, see Vapnik (1982), and Vapnik and Chervónenkis (1971). This theory can also be used to measure the efficiency of some different learning methods. For example, Haussler has investigated the PAC model for neural net learning, see Opper and Haussler (1991) and Haussler (1992).

  Another concept learning project is described in Ioannidis *et al* (1992), where a system uses learning for database design.

- *Learning from Examples.* Some projects use learning from examples. Pitas *et al* (1992) use examples to learn rule for a rule-based system. Gasarch and Smith (1992) describes a system that tries to learn a mathematical function from examples and is allowed to pose extra questions in order to improve the learning efficiency.

- *Learning as Discovery.* Doug Lenat's program AM, see Lenat (1977), uses a frame-based knowledge representation together with a heuristic search among some 250 rules to discover mathematical proofs. For example, it could discover the concept of prime numbers, which is quite impressing. However, AM relied heavily on its heuristics, and what it basically did, was to perform generalization from the implicit knowledge hidden in its rules.

- *Learning by Analogy.* Analogy is a powerful tool for inferencing, but it also demands a proper interpretation. Much interest has gone into this field, see for example Winston (1980). This system uses a frame representation to draw analogies between objects.

- *Learning Belief Networks and Expert Systems.* Some projects use learning to build knowledge-based systems. Hsu and Yu (1992) de-

scribes a fault diagnosis system that learns by examples, in much the same way as the BOXES algorithms to be described later. Torasso (1991) describes learning in a diagnostic expert system. Mahmoud *et al* (1992) describes a learning rule-based system for control. Neal (1992) describes a system that learns a belief network, i.e., an influence diagram by neural network techniques. the system can then be used more or less like an expert system.

- *Learning Mathematics.* Learning algorithms have been used to learning mathematical concepts. Helmbold *et al* (1992) describes a system that learns integer lattices, while the system of Arikawa *et al* (1992) learns simple formal systems.
- *Learning in Control Theory.* Learning is also used in several control applications, and the following can only server as a collection of some interesting examples. A survey is found in Moore *et al* (1992). Messner *et al* (1991) presents a new adaptive learning rule for parallel calculation. The target processes are robots. Mahadevan and Connell (1992) describes a robot learning to push boxes with learning of the BOXES (!) type. Heinzinger *et al* (1992) gives some stability results for learning control algorithms. Kalaba and Udwadia (1991) use learning for structural identification in mechanical systems. Ho *et al* (1992) describes a learning neural network for short-term load forecasting in power systems. Li and Tzou (1992) describes a learning fuzzy controller.

There are several other areas in learning, for example different versions of learning or generalization from examples, case-based reasoning, and also more connectionistic ideas, such as primitive agents that causes a global behavior to adapt or learn.

From this overview, it is clear that AI learning comprises several quite different methods. The aim of this essay is to give an adaptive control view of learning techniques, thus it will be limited to treating learning through *parameter adjustment.* The other techniques will not be further treated here.

3. *What is Adaptive Control?*

There has been difficulties in defining what is to be meant with adaptive control. The everyday meaning of the word "adapt" is to change behavior in order to better conform to new circumstances. The similarity to learning should be noted. The idea of adaptive control is to take a standard controller, such as a PID or RST controller, and change its behavior to suit changing operating conditions, i.e., to adjust its parameters, and maybe, (in case of the RST controller), its structure, to preserve a good tuning when the controlled process is changing. Thus, adaptive control may be defined as dealing with controllers *viewed as* consisting of two levels, one standard control level, and one level of parameter (and maybe structure) adaptation.

But as is the case with AI and learning, adaptive control is best defined by examples. There are two main approaches to adaptive control: model-reference adaptive systems, (MRAS), and self-tuning regulators, (STR). Both these approaches build on the general idea of using the difference between observed and wanted behavior, and to adjust parameters according to this. First, the MRAS approach will be described, as it is the simpler and more direct of the two. The following part is based on Åström and Wittenmark (1989).

Model-Reference Adaptive Systems

The model-reference adaptive system is based on a specification of a reference model, which describes how the controlled system should ideally behave. A block diagram description is given in Figure 1.

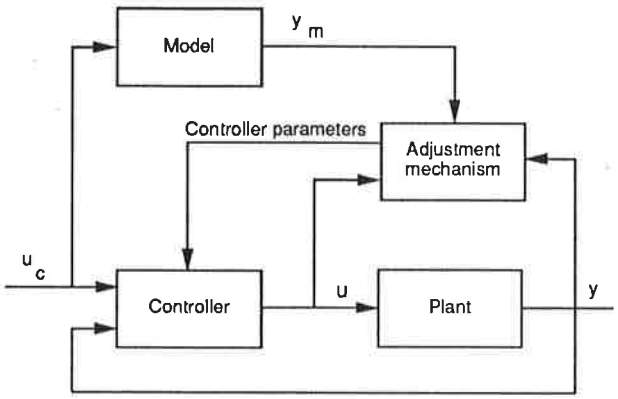

Figure 1. A block diagram of a model-reference adaptive system. From Åström and Wittenmark (1989).

The system consists of two control loops, one inner in which the controller controls the process, and one outer, where the adjustment mechanism adjusts the parameters of the controller so as to make the controller + process system behave similarly to the reference model.

The parameters of the controller are adjusted depending on the difference between the process and model outputs, i.e., the model error

$$e = y - y_m.$$

The sensitivity derivative of the model error for the parameter is also weighed in, and the value multiplied by a gain γ. The following adjustment mechanism, the MIT rule, was used in the first MRAS:

$$\frac{d\theta}{dt} = -\gamma e \frac{\partial e}{\partial \theta},$$

where e is the model error and γ the adjustment gain. In other words, the parameters of the controller are adjusted proportionally to the model error and the sensitivity derivative of the parameter in question.

In summary, a model-reference adaptive system has the following general properties:

- The processes handled are physical processes, which are continuous and usually can be described by linear or reasonably nice nonlinear equations. The equations are for example usually continuous, and the nonlinearities are often limited so that the parameters of the linear approximations do not vary within more than some orders of magnitude.
- The model error is derived as the difference between the process behavior and a reference model.
- The parameters of the controller are adjusted according to a gradient method, so that the model error is driven to zero.
- Under some reasonable assumptions, such as that the controlled process is linear, that the signals are persistently exciting, and that modeling errors and noise are small, it is possible to prove stability and convergence, or at least to hope that convergence will occur with reasonable speed in most cases.

Self-Tuning Regulators

The self-tuning regulator, (STR), is based on estimation of a parameterized model of the process. This model is then used to calculate controller parameters to achieve a specified behavior. A block diagram describing the general idea is found in Figure 2.

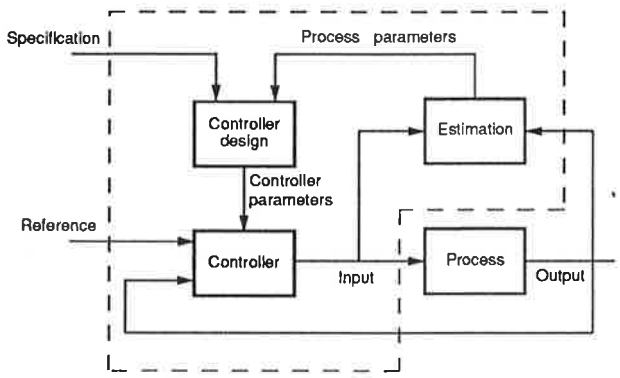

Figure 2. A block diagram of a self-tuning regulator. From Åström and Wittenmark (1989).

The *estimation* block is usually performed with a *recursive least squares* algorithm, which basically implies that the parameters of the model are updated according to the following equations:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)(y(t) - \varphi^T(t)\hat{\theta}(t-1))$$
$$K(t) = P(t-1)\varphi(t)(\lambda I + \varphi^T(t)P(t-1)\varphi(t))^{-1}$$
$$P(t) = (I - K(t)\varphi^T(t))P(t-1)/\lambda$$

This equation has a strong intuitive appeal. The estimate $\hat{\theta}$ is obtained by adding a correction term to the previous estimate $\hat{\theta}(t-1)$. This correction is a function of the difference between the process output and the output predicted by the model; it is a *prediction error* method.

Here, the similarity with the MRAS approach should be noted. Both methods adjust parameters to minimize the difference between an actual value and one supplied by a model, but one uses a gradient method, while the other uses a least squares algorithm. In an MRAS, controller parameters are adjusted, as is the case in a *direct* STR, while in an *indirect* STR, the adjusted parameters describe a model, which is used to compute controller parameters. The latter is done in the *controller design* block of Figure 2.

It is now possible to summarize the properties of a self-tuning regulator:

- The processes handled are, (once again), physical processes, which are continuous and usually can be described by linear or reasonably nice nonlinear equations. The equations are for example usually continuous, and the nonlinearities are often limited so that the parameters of the linear approximations do not vary within more than some orders of magnitude.
- The model error is derived as the difference between the process behavior and an estimated model.
- The parameters of the model are adjusted so that the prediction error is driven to zero.
- Under some reasonable assumptions, such as that the controlled process is linear, that the signals are persistently exciting, and that modeling errors and noise are small, it is possible to prove stability and convergence, or at least to hope that convergence will occur in most cases.

With these simple characterizations of adaptive controllers in mind, the time has come to investigate the different methods of AI learning, and compare them to the adaptive control ideas.

4. *Checkers*

The first, and also most classical example of learning by *parameter adjustment* is the second Checkers program of Samuel, see Samuel (1963). In order to interpret Samuel's ideas in control terms, let us first describe the learning algorithm.

A Description of the Program

Samuel's programs, as most other board game programs relied on a tree search. This means that from each position where a move must be made, the program investigates the possible moves and resulting new positions several moves ahead. At the end points, called leaf nodes, an evaluation

function is applied, giving a quantitative value describing how good the position is for the program.

The values of the leaf nodes are then backed up in a minimax fashion, so that whenever the program is to move, it chooses the maximal value among the successor nodes, while if the opponent is to move, the minimal value is chosen. From the root node, which corresponds to the position from which a move actually must be made, that move is chosen which leads to the position with the highest backed-up score. This minimax search has the advantage that it guarantees the program to find the best possible move within the search horizon and given that the opponent plays perfectly. If it is possible to search to the end of the game, the program will play a theoretically perfect game. The algorithm can be enhanced in several ways, for example with the $\alpha\beta$-algorithm, which means that out of a total of N nodes in the tree, only $2\times\sqrt{N}$ need be investigated. For non-trivial games, such as Chess, Checkers, and Othello, it is seldom possible to search to the end. In this case a heuristic *evaluation function*, $\hat{f}$, is used instead of the true value, f, of the leaf node positions. The better this evaluation function can approximate the correct value, the better the program will play.

The evaluation function of Samuel's second Checkers program consisted of a linear polynomial, where parameters, k_i, were used to produce a weighted score of sixteen terms, f_i, each being a quantitative measure of some feature of the current position, p, on the board:

$$\hat{f}(p) = \sum_{i=1}^{16} k_i \times f_i(p).$$

The efficiency of the evaluation function, f, and thereby the strength of the program's play depended critically on getting a good weighting of the different terms, i.e., of finding an optimal choice of values for the parameters k_i. Typical examples of terms, f_i, were the mobility, (the number of available moves), the advancement of pieces, (the number of rows the pieces had moved forward), the centrality, (the distance of the pieces from the center of the board), etc.

The evaluation function terms were 38 all in all, out of which 16 were used at a time. If one term had been given the lowest weighting for 32 moves, it was taken out of the scoring polynomial and replaced by one of the terms from the waiting pool. This strategy was used to allow any number of terms while simplifying the adaptation problem by not using more than 16 of them at a time.

The material advantage, (computed as 200 points for every man and 300 points for every king, the opponents pieces counted negatively), was deemed to be the most important term, and to give some basic stability and direction to the learning algorithm, this term was computed separately and always kept unchanged. Thus, the material advantage was in fact made the loss function for the learning. In Checkers, the gain of a piece usually leads to the win of the game; thus the assumption seem quite reasonable.

The program was run in two variants, α and β. The α version adapted parameters in its evaluation function, while β worked as an invariant opponent. Whenever α had won sufficiently many games against β, the latter took over the new parameter settings of α. If α lost three consecutive games, it was deemed to be on the wrong track and "a fairly drastic and arbitrary change was made in its scoring polynomial, (by reducing the coefficient of the leading term to zero)." This idea is somewhat similar to the *simulated annealing* methods used in neural networks. Samuel explains the problematic effect by referring to local maxima, but it is not clear that this really is the explanation.

The basic idea of the adaptation was that after each tree search the value of the evaluation function in the root node, $\hat{f}_r$, was compared to the backed up value of the search, $\hat{f}$. The difference,

$$\delta = \hat{f}_r - \hat{f},$$

was used as a kind of error, and the parameters in the evaluation function was slightly adjusted so that the evaluation in the root node, $\hat{f}_r$, matched the backed up value, $\hat{f}$, better.

The rationale for this was that under reasonable assumptions, (which are true for the game of Checkers), a value backed up by an $\alpha\beta$-search is more reliable than a direct application of the evaluation function. The program actually was trying to reach an evaluation function that approximated the value found by a deep tree search.

The inputs to the adaptation procedure were the sign of δ and the signs of the evaluation function parameters, k_i. A correlation coefficient for each parameter was updated during play, and the parameters computed using these coefficients. Specifically, if the ratio of two correlation coefficients, c_i, was bigger than n but smaller than $n + 1$, i.e., if

$$n \leq c_i < n + 1,$$

then the corresponding weighting parameter was set to 2^n, in order to give a large span between the weighting parameters.

Some stabilization measures were also taken. If δ was below a certain limit, no updating was done, while if the material balance was affected, the change in correlation coefficients was doubled, and if δ indicated a win or loss, (the largest possible values), it was quadrupled.

Results

Samuel describes two test runs with the program. The first one consisted of 28 games played, and resulted in violent changes in both which terms that were used and discarded, and in the weighting parameters. At least 20 different terms were at the leading position during different parts of the experiment. Even at the end of the run Samuel conceded that "the learning procedure was still not completely stable." The parameter setting resulted in a program that was first "tricky but beatable" and later "better than average."

Prior to the second test, several stabilization measures had been taken. The program was often fooled by bad play on behalf of the opponent, so the adaptation was made slower when α was leading. The terms were replaced every 32nd move, instead of originally after every 8th. It was also decided to demand that α should have a majority of wins over β before the replacement of scoring polynomials took place.

The results of the second test was a more stable but slower learning. Here, a seemingly semi-stable state was reached after some 30 games played. Still, after each parameter transfer, several oscillations occurred before α landed on a parameter setting that enabled it to win over β.

Samuel observed that the rote learning program efficiently learned to play opening and endgames, but never became good in the middlegame, while the parameter adaptation program quickly learned to play a good middlegame. However, it never learned to play in a conventional manner, and its openings were weak. Also, after 28 games, it still had not learned how to win with two kings against one in a double corner, a reasonably trivial task.

Interpretation

The updating mechanism used in Samuel's second Checkers program is a bit similar to an MRAS system; maybe most similar to the *sign-sign algorithm,* a simple version often used in telecommunications:

$$\frac{d\theta}{dt} = -\gamma \operatorname{sign}\left(\frac{\partial e}{\partial \theta}\right) \operatorname{sign}(e).$$

There are some important differences, however, between such an MRAS and Samuel's algorithm:

- Samuel's "process" is a minimax $\alpha\beta$ tree search. It consists of several maximum and minimum computations, together with logical selections, and as opposed to most physical processes, it is highly nonlinear and discontinuous. Thus, no stability proof is even within sight, and it is indeed not clear why an adaptation procedure should be able to converge, nor that Samuel's tests really do so.
- The "model error," δ, is discretized into four levels, while the correlation coefficients are computed by a stable and "calm" correlation algorithm, (Samuel does not give any precise details).
- Most difficult is probably the problem of *credit assignment,* i.e., the adaptation may be fooled by faults by the opponent. This corresponds somewhat to the problem of *persistent excitation.* The problem is how to find out exactly when the important choices were made, and to separate these moments for others, where faults by the opponent and consequences of earlier play are the reasons for a changing δ. Samuel's solution tries to avoid the worst effects, but all in all, this is still an unsolved problem. An obvious improvement would be to use the adaptation on lost games only.

Evaluation

Samuel's second Checker program is probably the most successful AI learning example, and for its time, (1963), it was certainly impressing. From today's horizon, however, the glory has faded somewhat:

- It is not clear that the adaptation procedure is stable or converges to a good value. The degree of calmness reached would not be deemed satisfactory in any adaptive control system.
- The level of play attained was, in spite of all rumors, not very high, neither compared to the human skill of those days, nor to the level of today's computer programs. Samuel's program won one game over one state master in the US, but that was a master of blind players, and there was still a long way to go in order to reach world class level. Today, the Checkers program Chinook, see Schaeffer *et al* (1992), plays on par with Marion Tinsley, the Checkers world champion, and is probably the world's second or third best player. This program relies on an efficient tree search, multiprocessing, and large databases of endgame positions. It uses no adaptation or learning whatsoever, but it can search up to 20 plies, (half moves), and often evaluates the leaf positions with 100% accuracy, as they can be found in the endgame databases. The 20 plies depth should be compared to the 3 or 4 ply searches of Samuel's program. With some knowledge of state of the art game programming techniques it is trivial to write a program that searches some 10 to 15 plies deep and outperforms Samuel's program by far.

Some Final Comments

Samuel's results were impressive and maybe the best example of successful learning in its time, but they are now of little importance to game-playing programs. This shows that a large portion of humility is needed in the field of learning. It is very difficult to reach working results at all, and there are some few successful results of this. To hope for better performance than with other techniques is so far unrealistic.

5. *Chess*

The *Chiptest, Deep Thought,* and *Deep Blue* programs are all part of a long term project of VLSI construction for chess processors, see Hsu (1987) and Anantharaman *et al* (1991). These programs are really a combination of software and hardware, to perform very fast minimax $\alpha\beta$ tree searches. The current version of Deep Blue is the world's strongest chess machine, but it still has some way to go before it can compete with the best human chess players.

Deep Blue and some other game-playing programs use databases of human master games to adapt their evaluation function parameters. The idea is simple:

- From information about which side that won a particular game, and maybe also who played it, the positions of the game are scored as good, bad, or even.
- The evaluation function polynomial is then tested in the different positions and the parameters adjusted to better agree with the previously computed scores. This adjustment is usually performed with a least squares or maximum likelihood criterion.

Interpretation

These methods are clearly very similar to the *least squares* and *recursive least squares* algorithms. The main difference is that the "process" in question, (the function telling the game-theoretic value of a position), is discrete, highly nonlinear and discontinuous. Making a single, seemingly insignificant move may change the value of a position from won to lost. Therefore it is unclear how well the approximations can work.

Evaluation

Several game-playing programs use methods like the one described to automatically adjust the parameters of their evaluation functions. The fact that Deep Blue uses the method has given it an unfairly good reputation, while the truth is that Deep Blue owes it first place among chess computers to its enormous speed, not its evaluation. The best evaluation functions are probably to be found in commercial programs running on slower hardware, and these evaluation functions have been produced by careful hand crafting.

As with Samuel's approaches, the method is troubled by the problem of *credit assignment,* and it is also very difficult to arrive at a reliable evaluation of the input positions. The nonlinearities of the problem may also cause troubles. It seems that the main advantage of the method is that it provides a simple way to give a hopefully reasonable tuning of a large set of parameters.

6. *Backgammon*

A well known game-playing achievement was when the program BKG 9.8 won a match of eight games of Backgammon over the then reigning world champion Luigi Villa with 7–1, see Berliner (1980). It should be noted, though, that the match did not concern the world champion title , and that post mortem analysis showed that the program did not play better than the human opponent, but was lucky with the dice rolls. However, it was the first time a computer program had won a match over a human world champion in any board or card game, and the program certainly played on par with the human opponent.

Berliner's program used no learning, but it is still interesting because Backgammon is best played without extensive tree searching. Instead, a good evaluation function is crucial. BKG relies on a well-tuned

hand crafted evaluation function where the weighting parameters, k_i, are smoothly varying with the type of position, making it nonlinear. The difference between Backgammon and other board games, such as Chess, Checkers, and Othello, is probably due to two factors. First, the dice rolls give a large degree of randomness to the game, so that any single move will not change the winning possibilities radically. Secondly, the relative simplicity of the game makes the game-theoretically correct evaluation smoother than in the other games.

Tesauro's and Sejnowski's Learning Program

The evaluation of Berliner's program was hand crafted and essentially concerned with estimating the value of different board patterns. It was therefore natural to try and use a neural network for evaluating the different patterns and selecting Backgammon moves, see Tesauro and Sejnowski (1989). This program used a three layer network to choose between generated moves. The inputs were coded into 459 nodes, of which 8 consisted of precomputed features such as number of men in different regions of the board, the number of *blots,* (pieces that may be hit, which are weak points), etc. The network had from 12 to 48 hidden nodes, used a training set of 3202 positions evaluated by a human player, (Tesauro), and needed about 100 to 200 hours of training on a SUN 3/160, using backpropagation. The resulting program played an intermediate level of Backgammon and had a 60% performance against the commercial program Gammontool, by SUN Microsystems.

Evaluation

The fact that the program managed to learn to play a respectable game of Backgammon is impressive. The domain is one of the largest that have been successfully tackled by a neural network approach. There are several weak points, however:

- Some hand crafted evaluation features were used. This helped to speed up the learning, but the program became reliant on a priori information.
- Lots of hand crafting was needed in selecting the examples, so that the program learned to handle different types of problematic positions.
- The human evaluation of the positions in the learning set was not completely reliable.
- The level of play was not so high. Berliner's BKG program plays far better using standard game-playing techniques and a hand crafted evaluation function.

The conclusion of this example is that it is indeed possible to learn a good evaluation function for Backgammon with a neural network. This is an impressive achievement in itself. On the other hand, conventional techniques still outperform the ones based on learning.

7. *Othello*

Another game that has been the target of research is Othello. Frey (1986) and Mitchell (1984) describe a project in which a large database of late middlegame Othello positions were used to tune the parameters of an evaluation function, much in the similar way to that used by Hsu *et al* for Deep Blue. The true score of the positions were computed by deep, (and thus very time-consuming), searches to the end of the game, and then the parameters of an evaluation function was adjusted to match the correct scores as well as possible.

As an important part of an Othello evaluation function seems to be to evaluate different edge and corner patterns, it is an obvious idea to try and train a neural network to perform this evaluation. Such project is described in the thesis Walker (1993). Genetic algorithms hav ealos been used in Othello evaluation, see Gupton (1989). The best Othello programs today use simple, hand crafted evaluation functions and rely on deep tree searches to play on par with the strongest human players. A project at the Department of Computer Engineering, Lund Institute of Technology, aims at building very fast hardware for Othello.

8. *Boxes*

Michie and Chambers have devised an general learning algorithm called BOXES, see Michie and Chambers (1968). It began as an algorithm for playing trivial games as 3×3 *naughts and crosses*. The idea is quite simple. A problem is broken down into sub-problems, and a score is kept over every possible action in every sub-problem. The algorithm plays a large number of games and chooses the actions of each sub-problem that has given the best outcome so far. In the case of a board game, there is a sub-problem, (a box), for every possible position of the game, and the possible actions are the possible moves from the position in question. As the number of possible positions for any non-trivial game is very large, (10^{30} for Checkers and Othello, and 10^{120} for Chess), the BOXES algorithm is not possible to use for them. However, Michie and Chambers used it to learn a somewhat different "game," that of balancing a pole on a cart.

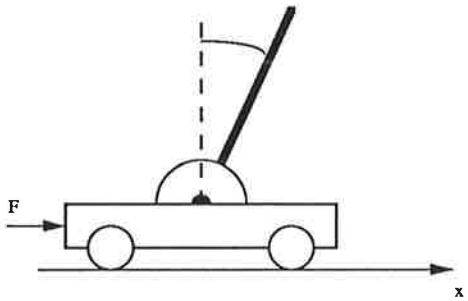

Figure 3. Michie's cart and pole system.

Figure 3 shows the *cart and pole* system used by Michie and Chambers in their experiment. A wheeled cart can move back and forth along a plane. On top of the cart there is a pole, and the task is to balance the pole by moving the cart. The input signals used by the BOXES algorithm were:

x The position of the cart on the track.

$\dot{x}$ The velocity of the cart.

θ The angle of the pole.

$\dot{\theta}$ The angular velocity of the pole.

The BOXES model was obtained by a rough quantization of the state space. Thus, x and θ were discretized into five values, and $\dot{x}$ and $\dot{\theta}$ into three. The x value intervals were three of equal size and two unbounded. The θ intervals were one quite small in the middle, two wider, and two unbounded. The $\dot{x}$ and $\dot{\theta}$ intervals consisted of one narrow interval and two unbounded. This means that the state space was divided into $5 \times 5 \times 3 \times 3 = 225$ *boxes.*

The output signal was the force F controlling the acceleration of the cart. It was discretized into only two values, + and −, or *left* and *right.* Each box contained a randomly chosen action from start. In the original experiment, the system was simulated on a computer with a sampling time of 20 Hz. It should be noted that "nice" values were chosen on the masses and length, so that the system was fairly easy to control. Even so, Michie and Chambers also put in a weak spring that tried to pull the pole towards the upright position, in order to make the system more easy to control. Later, physical systems have in fact been built, but the small number of boxes demands that the dimensions of the system be "nice" enough, otherwise the number of boxes will be too large for the learning algorithm to work in practise, see Bernhardsson and Larsson (1989), where a physical inverted pendulum was investigated. This process demands some $100 \times 40 \times 6$ boxes in order to be successfully controlled, which makes the learning impractically slow.

Learning Algorithm

The original learning algorithm was quite simple, and it is this algorithm that will be treated here. For every local box, some values are computed:

L_l The *left life.* A weighted sum of the number of sampling points during which the system managed to keep the pole from falling, (the life time), while using the action *left* in the box.

L_u The *left usage.* A weighted sum of the number of *left* actions actually used.

R_l The *right life.* A weighted sum of the life times while using the action *right* in the box.

R_u The *right usage.* A weighted sum of the number of *right* actions used.

In addition to this, two global values are updated:

G_l The *global life.* A weighted sum of the life times of the runs.

G_u The *global usage*. A weighted sum of the number of decisions taken.

All the sum are weighted so that there is a forgetting factor of 0.99. Thus, the updating rules for the global values are:

$$G_l = 0.99G_l + T_F$$
$$G_u = 0.99G_u + 1,$$

where T_F is the time in sampling points from the start of the run to when the pole fell. Thus, the quotient G_l/G_u is proportional to the average life time of the test runs.

The local values are updated according to the following formulas, (where it is assumed that the box in question is using the *left* action during the current run):

$$L_l = 0.99L_l + \sum_{i=1}^{N}(T_F - T_i)$$
$$L_u = 0.99L_u + N$$
$$R_l = 0.99R_l$$
$$R_u = 0.99R_u$$

With these values, the following calculations are performed:

$$v_L = \frac{L_l + 20\frac{G_l}{G_u}}{L_u + 20}$$
$$v_R = \frac{R_l + 20\frac{G_l}{G_u}}{R_u + 20}.$$

The control action is selected as *left* or *right* according to whether v_L is greater than v_R or not.

Results

The results of the learning tests varied within large magnitudes. After some 300 test runs the system was often able to balance the pole for 20 to 40 seconds. In one case it was never able to balance it more than in average 15 seconds, while in another case, after 600 tests it could balance it for 300 seconds in average. Michie and Chambers point out one successful run of more than 72 000 sampling points, corresponding an hour of simulated time.

As can be seen from these results, the learning process is not very stable, and it converges very slowly and unreliably. This may seem strange, as it is certainly possible to control the system with a standard state space feedback controller, and the learning algorithm should reach a discretized version of this sooner or later. The conclusion must be that the learning procedure is indeed extremely slow. Later experiments by other groups

have tried to better the speed of the learning by using more a priori information about the process.

Barto, Sutton, and Andersson improved on the experiment by designing two adaptive, neuron-like elements. They still used a predefined division of the state space and their program was able to balance the pole after some 100 trials. One reason for the better results may also have been that a higher sampling frequency of 50 Hz was used, see Barto, Sutton, and Andersson (1983).

Andersson used two predefined two-layer neural networks, one learning an evaluation function and the other learning the control actions. In this way, no predefined division of the state space was necessary. On the other hand, this program took some 10 000 trials to learn to balance the pole for an average of 140 seconds, see Andersson (1986).

In the CART experiment, the state trajectory was traced through each trial and a continuous interpolation-function replaced the state division. The program only considered states that was actually reached, and an elaborate analysis was used to pinpoint the erroneous control actions. The program was able to balance the pole indefinitely after only 16 trials, but a large amount of a priori knowledge was used in the algorithm, see Connell and Utgoff, (1987).

Interpretation

Michie's and Chambers' original algorithm is quite simple. It forms an average of the life times achieved by either *left* or *right* for each box. This average is weighted by a forgetting factor of 0.99 so that more recent observations will be more important. Then, very simply, the action with the higher life time is chosen, but the life time average is summed with the global life time average and normalized, so that when global life times increase, the adjustment of actions will be slower. For each box, the algorithm is similar to a proportional controller working on an averaged input with exponential forgetting, and the output is discretized into two levels only and the gain decreased in proportion to how successful the controller is.

Evaluation

Michie's and Chambers' BOXES algorithm is quite general, and *can* learn to control the cart and pole process. It still uses important a priori knowledge about the division of the state space, though, and given this, it is fair to say that it shows a low order of stability and an extremely slow convergence. An MRAS or STR would converge after a couple of pole movements while BOXES needs hundreds of complete test runs. Other approaches are either more general than BOXES, but then even slower in learning, or more efficient but then dependent on more specific knowledge. In the latter cases, it seems that the point of the experiment has been lost, and that the solutions are far too specific to the process. As such solutions, the proposed algorithms does not compare favorably with standard state space controllers.

Some Final Comments

Once again it is seen that the goal of learning will not be to outperform standard algorithms. Indeed, it is difficult enough to achieve successful learning at all. Michie himself has emphasized this, Michie (1990). It is also easy, it seems, to loose track of what is relevant research. The original BOXES experiment is general enough to be applied to many problems, while the solutions that are more efficient in learning use a lot of a priori information, which makes their general value doubtful.

But the most important observation is probably that a conventional model such as a state space description,

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

contains a large amount of useful a priori knowledge, and that learning without knowing even the structure of the process is very difficult.

9. *Neural Networks*

In recent years, neural networks have been successfully applied to pattern recognition and optimization tasks, see for example Hopfield (1982), Hopfield and Tank (1985), Burr (1988), Gorman and Sejnowski (1988), Sejnowski and Rosenberg (1987), and Widrow, Winter, and Baxter (1988). Two main types of neural networks have been used, the multilayer network and the Hopfield net.

The area of neural networks have grown immensely in the last years. Thus, the following is only an interpretation of the basic techniques, and no overview of the field. A good presentation of the research area, with a twist towards control, is found in Miller *et al* (1990).

Multilayer Networks

The most common neural network consists of a set, (layer), of input nodes, some layers of hidden nodes, and a layer of output nodes. Each node in the hidden and output layers receive as input a weighted sum of the nodes in the preceding layer. Some function, (often a sigmoid or truncation), is applied to this sum, and the result becomes the output of the node, to be used by the next layer, see Figure 4.

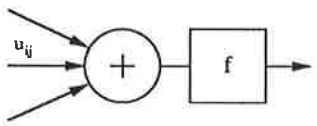

Figure 4. A single node of a multilayer neural network.

Some special types of nets have limitations on the number of layers and the functions used. Adaline is basically a one layer net, see Widrow and

Hoff, (1960). The multilayer network was originally called a Perceptron, see Rosenblatt (1962). In a CMAC network, (a special case of a Perceptron), the function applied is the identity, thus a CMAC is a linear summing device, see Albus (1975 a, b).

This general structure means that a multilayer neural network can be described as a static nonlinear map f, where

$$f(x) = F(WF(VF(Ux))).$$

Here F is the (nonlinear) function and U, V, and W the weighting matrices corresponding to the connections in the network, see Figure 5.

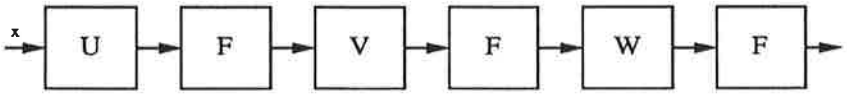

Figure 5. A multilayer neural network in block diagram form.

According to Weierstraß' theorem, every map $C(R^n, R^m)$ can be approximated to any degree by a polynomial, and it has been shown that a three layer neural network with an arbitrarily large number of nodes in the hidden layer can approximate any continuous function over a compact subset of R^n. This implies that neural networks with one hidden layer are capable of performing any characterization.

Interpretation

There is no special control interpretation of a multilayer neural network. It is simply a nonlinear map, and the use of neural networks as opposed to, say, polynomial approximations depends on whether the representation is versatile enough to provide a good basis for analysis and algorithm design. A neural network is simply one general way of "packing" a static nonlinear function.

Hopfield Networks

A Hopfield network usually consists of a layer of nodes and a feedback via time delay, see Figure 6.

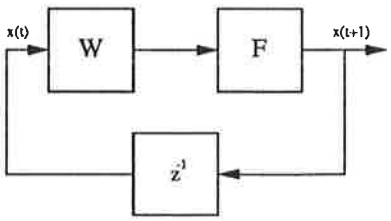

Figure 6. A Hopfield network in block diagram form.

Here, the inputs are the weights of the nodes and the output the stable states.

Interpretation

A Hopfield network is based on feedback, and in the case of a one layer network, the control interpretation is simple and well-known. In this case the Hopfield net is identical to a linear system with no inputs,

$$\dot{x} = Ax.$$

The difference from the standard use of such systems is that here the parameters of the matrix A are the "inputs" and the stable states of the system are the "outputs."

Weight Adjustment

Neural networks became popular when the *backpropagation* method for adjusting the weights was introduced, see Narendra and Parthasarathy (1988). In this method, the partial derivatives of an error criterion with respect to the weights in a multilayer neural network are determined and the weights are adjusted along the negative gradient to minimize the error function.

In order to perform backpropagation, the same network of nodes may be used, but the signals flow in the other direction, which explains the name backpropagation.

Interpretation

The backpropagation algorithm is more or less a pure MRAS method. The change of the weights, ω_i, is proportional to the partial derivative of the output error, e:

$$\frac{d\omega}{dt} = \gamma \frac{\partial e}{\partial \omega_i}.$$

This is a variant of an MRAS adjustment rule.

Thus, it can be concluded that backpropagation is a reliable method, but that it will suffer from all the same problems as the MRAS approach, the most important being that of ensuring persistent excitation.

Adaptive Control with Neural Networks

It is possible to use neural networks both for system identification and adaptive control, see Narendra (1990). Here one neural network is trained to behave as the process, (estimation), and one network is trained to use the estimated model to make the system behave as a reference model, see Figure 7.

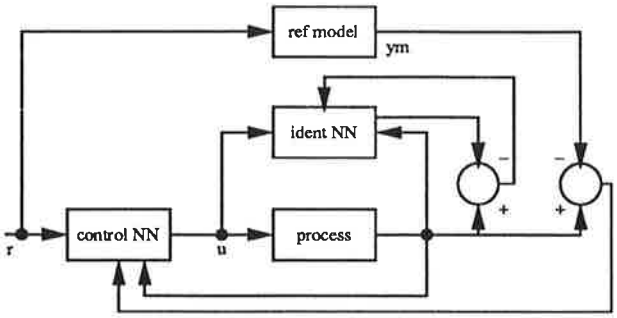

Figure 7. A self-tuning regulator based on neural networks. From Narendra (1990).

Interpretation

It is straight-forward to see that the proposed controller architecture is an indirect STR, where the estimation and design blocks have been implemented with neural networks. It is interesting to note that so far, no methods for making *direct* controllers based on neural networks exist, see Narendra (1990).

In conventional adaptive control, several assumptions are made in order to guarantee that the methods will work:

- The sign of the high frequency gain is known.
- The order of the plant is known.
- The relative degree of the plant transfer function is known.
- The zeros of the plant lie inside the unit circle.
- The reference model is linear.

These assumptions make it easier for the method to work with a successful result, and indeed they are necessary for allowing proofs of convergence and stability. It seems rather obvious that if a controller based on neural networks is based on a process that does not obey these assumptions, problems will occur. When the assumptions are obeyed, however, the neural network approach could be feasible.

Evaluation

It is hard to evaluate the success of neural networks as a whole. It is clear that they have been successful in specific tasks as pattern recognition, and that they may be useful in for example intelligent sensor methods.

Concerning control, very few results have so far been reached. When applied to problems which conventional adaptive controllers handle well, it may be supposed or at least hoped that the neural networks will also be successful. For more difficult problems, such as processes that violate some of the assumptions listed above, it should not be hoped that neural

networks will be better able to solve them, because the problems originate from the process and other circumstances, and may not be solvable whatever method is used.

An important difference between conventional control theory and neural networks is that, as neural networks are nonlinear, almost nothing of the known theory of stability and convergence can be used. So right now, using a neural network approach means that very little theoretical analysis can be done and no stability proofs given. The only way left is to use the methods and see what happens, which is a worse situation than is the case for conventional controllers.

10. A Comparison

Parameter adjustment has been used to provide a learning behavior in many domains, as can be seen from the examples above. Some general similarities and differences should be noted:

- The type of "process" may vary drastically. Adaptive controllers usually operate on reasonably nice and smooth processes, and are therefore often successful. The same is true for BOXES and neural networks for control, while the game-playing applications meet a very different process, the game-theoretic value function. This functions is discrete, but also highly nonlinear and irregular. Thus, there is no possibility to ensure stability and convergence. Indeed, it is quite surprising that good results are possible to obtain at all.
- A few learning algorithms can be proved to work under specified conditions, (Vapnik-Chervónenkis), but most of the algorithms do not provide any such proofs.
- The "difficulty" of the process greatly affects the speed of convergence. The adaptive controllers operate on the nicest processes and are thus the fastest, while Samuel's parameter adjustment is slower, even though it is still a kind of gradient method. Here, the complex process makes the convergence more difficult.
- The problem of persistent excitation varies greatly. In adaptive control, this problem has been thoroughly studied, and some countermeasures against so called "estimator windup" and other bad effects caused by lacking excitation are usually included in the algorithms. In alla the other examples mentioned, the problem is not handled at all. Instead, the methods rely on the general randomness of the experiment situations to provide enough stimulation.
- Some learning algorithms use quite much a priori information, e.g., the adaptive controllers. These algorithms are usually fast in convergence. Other algorithms, e.g., BOXES and neural networks use much less a priori information. The only predefined knowledge in these cases is a set of input signals and a failure or error signal. These algorithms are consequently slower in learning.

- Algorithms like BOXES, which rely on gathering a large statistic, (Monte Carlo-like methods), are considerably slower than gradient methods, (MRAS, backpropagation, Samuel's program), which in turn are slower than least square algorithms, (STR, Deep Blue, Frey's Othello evaluation).

11. Conclusions

Learning is a fascinating prospect, and much longed for. If a problem is difficult to solve, or demands knowledge that is not available or hard or costly to gain, how good it would be to build a machine that could learn by itself.

However, in most problem areas there are lots of knowledge, and furthermore, learning is much more difficult than one may at first guess, and as has been seen from the examples, it is often very slow and costly. Thus, learning is very seldom a viable option, and so far it has not led to any system that is *better* than a corresponding conventional non-learning system. So one conclusion is that the ideas about learning as the crown of methods, that will solve all problems, are wrong. Learning has so far not given better results than conventional techniques.

Learning is still an important research target, however, as learning processes are present in many activities. Therefore it is important to study and learn how learning works. But the goals must be modestly set. If learning is at all possible, that is a good feat in itself. In some limited areas, especially concerning neural networks for pattern recognition, learning has been quite successful, and hopefully, it will come to use in more areas and grow to a more and more successful discipline, slowly but firmly, in a step by step fashion.

The final observation is that it is important not to forget the conventional adaptive control methods, i.e., the MRAS and STR approaches. They are certainly the most successful examples of learning, and so far the only ones that have migrated from research to practical application in large numbers. They can also be used to give examples of what problems the other approaches may meet in the future. So far, the best example of learning *is* adaptive control.

12. References

ALBUS, J. S. (1975 a): "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller, (CMAC)," *Transactions of the ASME*, September, 220–227.

ALBUS, J. S. (1975 b): "Data Storage in the Cerebellar Model Articulation Controller, (CMAC)," *Transactions of the ASME*, September, 228–233.

ANANTHARAMAN, T. S., M. S. CAMPBELL, and F.-S. HSU (1992): "Singular Extensions: Adding selectivity to Brute-Force Searching," *Artificial Intelligence*, **53**, 1, 99–109.

ANDERSSON, C. W. (1986): *Learning and Problem Solving With Multilayer Connectionist Systems*, Ph. D. Dissertation, Coins technical report 86–50, Amhearst, Massachusetts.

ARIKAWA, S., T. SHINOHARA, and A. YAMAMOTO (1992): "Learning Elementary Formal Systems," *Theoretical Computer Science*, **95**, 1, 97–113.

ÅSTRÖM, K. J. and B. WITTENMARK (1989): *Adaptive Control*, Addison-Wesley, Reading, Massachusetts.

BARTO, A. B., R. S. SUTTON, and C. W. ANDERSSON (1983): "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, **13**, 5.

BELLMAN, R. (1961): *Adaptive Control Processes: A Guided Tour*, Princeton University Press, Princeton, New Jersey.

BERNHARDSSON, B. and J. E. LARSSON (1989): "Learning State Space Controllers," Technical report, Department of Automatic Control, Lund Institute of Technology, Lund.

BERLINER, H. (1980): "Computer Backgammon," *Scientific American*, June 1980, 54–62.

BURR, D. J. (1988): "Experiments on Neural Net Recognition of Spoken and Written Text," *IEEE Transactions on Acoustic, Speech, and Signal Processing*, **36**, 1162–1168.

CONNELL, M. E. and P. E. UTGOFF (1987): "Learning to Control a Dynamic Physical System," *Proceedings of the AAAI '87*, American Association for Artificial Intelligence, Seattle, Washington, pp. 456–460.

FREY, P. W. (1986): "Algorithmic Strategies for Improving the Performance of Game-Playing Programs," in Farmer, D., A. Lapedes, N. Packard, and B. Wendroff, (Eds.): *Evolution. Games, and Learning*, North-Holland, Amsterdam.

GASARCH, W. I. and C. H. SMITH (1992): "Learning via Queries," *Journal of the ACM*, **39**, 3, 649–674.

GORMAN, R. P. and T. J. SEJNOWSKI (1988): "Learned Classification of Sonar Targets Using a Massively Parallel Network," *IEEE Transactions on Acoustic, Speech, and Signal Processing*, **36**, 1135–1140.

GUPTON, G. M. (1989): "Genetic Learning Algorithm Applied to the Game of Othello," in Levy, D. N. L. and D. F. Beal, (Eds.): *Heuristic Programming In Artificial Intelligence: the First Computer Olympiad*, Ellis Horwood, pp. 241–254.

HAUSSLER, D. (1988): "Qualifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework," *Artificial Intelligence*, **36**, 1–2,

172–221.

HAUSSLER, D. (1992): "Decision Theoretic Generalizations of the PAC Model for Neural Net and Other Learning Applications," *Information and Computation*, **100**, 1, 78–150.

HAUSSLER, D., M. KEARNS, N. LITTLESTONE, and M. K. WARMUTH (1991): "Equivalence of Models for Polynomial Learnability," *Information and Computation*, **95**, 2, 129–161.

HELMBOLD, D., R. SLOAN, and M. K. WARMUTH (1992): "Learning Integer Lattices," *SIAM Journal on Computing*, **21**, 2, 240–266.

HO, K. L., Y. Y. HSU, and C. C. YANG (1992): "Short-Term Loaf Forecasting Using a Multilayer Neural Network with an Adaptive Learning Algorithm," *IEEE Transactions on Power Systems*, **7**, 1, 141–149.

HOPFIELD, J. J. (1982): "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of Sciences, USA*, **79**, 2554–2558.

HOPFIELD, J. J. and D. W. TANK (1985): "Neural Computational of Decisions in Optimization Problems," *Biological Cybernetics*, **52**, 141–152.

HSU, F.-S. (1987): "A Two Million Moves/Sec CMOS Single Chip Chess Move Generator," *1987 ISSCC Digest of Technical Papers*, p. 278.

HSU, Y. Y. and C. C. YU (1992): "A Self-Learned Fault-Diagnosis System Based on Reinforcement Learning," *Industrial and Engineering Chemistry Research*, **31**, 8, 1937–1946.

HEINZINGER C., D. FENWICK, B. PADEN, and F. MIYAZAKI (1992): "Stability of Learning Control with Disturbances and Uncertain Initial Conditions," *IEEE Transactions on Automatic Control*, **37**, 1, 110–114.

IOANNIDIS, Y. E., T. SAULYS, and A. J. WHITSITT (1992): "Conceptual Learning in Database Design," *ACM Transactions on Information Systems*, **10**, 3, 265–293.

KALABA, R. E. and F. E. UDWADIA (1991): "An Adaptive Learning Approach to the Identification of Structural and Mechanical Systems," *Computers and Mathematics with Applications*, **22**, 1, 67–75.

LENAT, D. B. (1977): "The Ubiquity of Discovery," *Artificial Intelligence*, **9**, 3.

LI, C. J. and J. C. TZOU (1992): "A New Learning Fuzzy Controller Based on the P-Integrator Concept," *Fuzzy Sets and Systems*, **48**, 3, 297–303.

MAHADEVAN, S. and J. CONNELL (1992): "Automatic Programming of Behavior-Based Robots Using Reinforcement Learning," *Artificial Intelligence*, **55**, 2–3, 311–365.

MAHMOUD, M. S., S. KOTOB, A. A. ABOUELSEOUD, H. M. ELSAYED (1992): "A Learning Rule-Based Control System," *Information and Decision Technologies*, **18**, 1, 55-66.

MICHIE, D. (1990): private communication.

MESSNER, W., R. HOROWITZ, W. W. KAO, and M. BOALS (1991): "A New Adaptive Learning Rule," *IEEE Transactions on Automatic Control*, **36**, 2, 188–197.

MICHIE, D. and R. A. CHAMBERS (1968): "Boxes: An Experiment in Adaptive Control," in Dale, E. and D. Michie, (Eds.): *Machine Intelligence 2*, Oliver & Boyd, London.

MITCHELL, D. H. (1984): "Using Features to Evaluate Positions in Expert's and Novice's Othello Games," Master's thesis, Northwestern University, Evanston, Illinois.

MOORE, K. L., M. DAHLEH, and S. P. BHATTACHARYYA (1992): "Iterative Learning Control—A Survey and New Results," *Journal of Robotic Systems*, **9**, 5, 563–594.

NARENDRA, K. S. (1990): "Adaptive Control Using Neural Networks," in Miller, W. T., R. S. Sutton, and P. J. Werbos, (Eds.): *Neural Networks for Control*, MIT Press, Cambridge, Massachusetts.

NARENDRA, K. S. and K. PARTHASARATHY (1988): "A Diagrammatic Representation of Back Propagation," Technical report 8815, Center for Systems Science, Department of Electrical Engineering, Yale University, New Haven, Connecticut.

NEAL, R. M. (1992): "Connectionist Learning of Belief Networks," *Artificial Intelligence*, **55**, 1, 71–113.

NEWELL, A. and H. A. SIMON (1963): "GPS, A Program That Simulates Human Thought," in Feigenbaum, E. A. and J. Feldman, (Eds.): *Computers and Thought*, McGraw-Hill, New York.

OPPER, M. and D. HAUSSLER (1991): "Generalization Performance of Bayes Optimal Classification Algorithm for Learning a Perceptron," *Physical Review Letters*, **66**, 20, 2677–2680.

PITAS I., E. MILIOS, and A. N. VENETSANOPOULOS (1992): "A Minimum Entropy Approach to Rule Learning from Examples," *IEEE Transactions on Systems, Man, and Cybernetics*, **22**, 4, 621–635.

ROSENBLATT, F. (1962): *Principles of Neurodynamics*, Spartan, New York.

SAMUEL, A. L. (1963): "Some Studies in Machine Learning Using the Game of Checkers," in Feigenbaum, E. A. and J. Feldman, (Eds.): *Computers and Thought*, McGraw-Hill, New York.

SCHAEFFER, J., J. CULBERSON, N. TRELOAR, B. KNIGHT, P. LU, and D. SZAFRON (1992): "A World Championship Caliber Checkers Program," *Artificial Intelligence*, **53**, 2–3, 273–290.

SEJNOWSKI, T. J. and C. R. ROSENBERG (1987): "Parallel Networks that Learn to Pronounce English Text," *Complex Systems*, **1**, 145–168.

TESAURO, G. and T. J. SEJNOWSKI (1989): "A Parallel Network that Learns to Play Backgammon," *Artificial Intelligence*, **39**, 3, 357–390.

TORASSO, P. (1991): "Supervising the Heuristic Learning in a Diagnostic Expert System," *Fuzzy Sets and Systems*, **44**, 3, 357–372.

VALIANT, L. G. (1984): "A Theory of the Learnable," *Communications of the ACM*, **27**, 11, 1134–1142.

VAPNIK, V. N. (1982): *Estimation of Dependences Based on Empirical Data*, Springer Verlag, New York.

VAPNIK, V. N. and A. Y. CHERVÓNENKIS (1971): "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities," *Theor. Probab. Appl.*, **16**, 2, 264–280.

WALKER, S. (1993): According to a Usenet posting, Steven Walker publsihed a thesis on a neural network to play Othello, somewhere in Australia. No record has been found of this, however..

WIDROW, B., R. G. WINTER, and R. A. BAXTER (1988): "Layered Neural Nets for Pattern Recognition," *IEEE Transactions on Acoustic, Speech, and Signal Processing*, **36**, 1109–1118.

WIDROW, B. and M. E. HOFF, JR. (1960): *IRE WESCON Convention Record*, pt. 4, pp. 96–104.

WINSTON, P. H. (1975): "Learning Structural Descriptions from Examples," in Winston, P. H., (Ed.): *The Psychology of Computer Vision*, McGraw-Hill, New York.

WINSTON, P. H. (1980): "Learning and Reasoning by Analogy," *Communications of the ACM*, **23**, 12, 689–703.