ADAPTIVE PARALLEL DISTRIBUTED PROCESSING:
NEURAL AND GENETIC AGENTS

PART ONE:

NEURO-GENETIC AGENTS AND A STRUCTURAL THEORY OF SELF-REINFORCEMENT LEARNING SYSTEMS

Stevo Bozinovski

CMPSCI Technical Report 95-107

November 1995

Preface

I hesitate to comment on the history of reinforcement learning. Being deeply absorbed in this research myself, my view is hardly detached, and I hope that the most outstanding developments are yet to come. But this report requires me to provide some historical comments. It chronicles research on aspects of machine learning that Dr. Stevo Bozinovski conducted when he visited what was then the Department of Computer and Information Science at the University of Massachusetts in 1980 and 1981. This research built upon his earlier machine learning work under the influence of what we were doing at the time. I was a postdoctoral researcher working with graduate students Rich Sutton and Chuck Anderson on a project whose goal was to investigate an approach to machine learning that later became widely known as reinforcement learning. Stevo arrived as we were working on an approach to the credit assignment problem of reinforcement learning that had roots in animal learning theory and early artificial intelligence. This work reached one level of fruition in Sutton's 1984 Ph.D. thesis "Temporal Credit Assignment in Reinforcement Learning". We used one of Sutton's algorithms as part of the "actor–critic" architecture that we illustrated using the pole–balancing problem, having been inspired by the 1968 "Boxes" system of Michie and Chambers. Sutton further developed this work into the TD(λ) class of algorithms. Motivated by the credit assignment problems we were studying, Stevo developed his own approach, calling it the Crossbar Adaptive Array (CAA) architecture. Although no one knew it at the time, Stevo had developed an architecture falling in the other of the two main classes of reinforcement learning architectures. Unlike the actor–critic architecture, which learned a value function of states, the CAA architecture learned a value function defined on state–action pairs. Although learning values for state–action pairs had already been present in the Boxes system, this general approach did not become widely appreciated until Watkins presented the Q–learning class of algorithms in 1989 within the context of dynamic programming, proving that they could find optimal values under certain conditions. Stevo's CAA architecture should therefore be recognized as an early example of this class of state–action value learners. Moreover, a special case of the general CAA architecture that Stevo calls the "greedy–emotion CAA", turns out to be the same algorithm Watkins independently developed in 1989, calling it one–step Q–learning. Consequently, Stevo's work with the CAA architecture deserves recognition as an early contribution to this class of reinforcement learning architectures. Having acknowledged this, however, I am certainly not willing to credit Stevo with the "invention of Q–learning", since the most significant part of this invention was Watkins' development of Q–learning within the context of optimal control theory and dynamic programming, and his proving that these algorithms converge to optimal solutions, something that is not true for all instances of the CAA architecture. But I invite readers of this report, as does Stevo, to form their own opinions about how the CAA architecture deserves to be regarded by reinforcement learning researchers. This report was produced while Stevo was a visiting Fulbright scholar during 1995/96. I was his hosting faculty associate for this visit. I was not, and never have been, his academic adviser, as he states in his recent book.

Andrew Barto
Amherst, 1996.

ACKNOWLEDGMENTS

This is the first written report of the project "Adaptive parallel distributed processing: Neural and genetic agents" carried out under the Fulbright Grant at the Adaptive Networks Group, Computer Science Department, University of Massachusetts, Amherst, USA.

Ms. Susan Krause, the Director of the American Center in Skopje, was the person who carried out the competitive procedure which enabled me, along with other candidates, to obtain the Fulbright Grant.

Professor Andy Barto enabled me to rejoin the Adaptive Networks Group after 14 years. My first visit to this group was in 1980-1981, also under the Fulbright Grant. At that time, under his leadership, and influenced by his broad insight into learning theory, I was able to produce results which turned out to be highly relevant in light of contemporary research on learning systems, and which I discuss in this report. This time he had patience to read this report in several iterations of its development, and give valuable criticism for improving its quality.

Professor Spinelli was the person who accepted my application (suggested by professor Michael Arbib) fourteen years ago and who enabled me to meet scientific workers like Andy Barto, Rich Sutton Chuck Anderson, and Harry Klopf, and to share my previous experience with them, and learn from them. Professor Spinelli also gave me the financial support to extend my stay with the Adaptive Networks Group beyond the Fulbright Grant Period. That was indeed the period when I developed the Crossbar Adaptive Array architecture which I discuss in this report.

To all of them I express great gratitude to be able to write this research report.

ABSTRACT

The project *Adaptive Parallel Distributed Processing: Neural and Genetic Agents* has three parts. The first part is a review of our previous work with emphasis on neuro-genetic agents and its significance for contemporary and future work. The second part deals with distributed reinforcement learning agents. The third part is concerned with genetic agents involved in models of adaptive parallel distributed production, a notion we developed in relation to the protein biosynthesis system.

This first part of the project connects previous research of the author with the contemporary issues of adaptive and learning systems. The research in Adaptive Arrays, carried out fourteen years ago, is evaluated from the perspective of the contemporary state of the art in reinforcement learning and dynamic programming. The previously proposed systems are considered within a framework called the *structural theory* of reinforcement learning systems.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

ABSTRACT

1. INTRODUCTION

1.1. The framework
1.2. Agents and architectures
1.3. Neural architectures
1.3.1. Crossbar Adaptive Arrays
1.4. Problems, solutions, and tools

2. A STRUCTURAL THEORY OF LEARNING AGENTS

2.1. Basic notions
2.2. Learning interfaces and a taxonomy of the learning paradigms
2.3. Consequence learning systems
2.4. A generic learning agent: Neuro-genetic agent
2.5. Infering learning rules from the agent's structure
2.6. Structural adaptation of the NG agents

3. CLASS A LEARNING SYSTEMS: ADVICE TAKING LEARNING AGENTS

3.1. Class A learning agents
3.2. Greedy policy perceptrons
3.3. Harware designs: Adaptive arrays
3.3.1 A classical conditioning model
3.3.2. Class A adaptive array
3.3.3. Isothreshold adaptive network
3.4. Toward a theory of teaching systems
3.5. Parallel design of the teacher-learner interaction
3.6. Teaching strategies
3.7. Teaching grammars and languages
3.8. The tutorial algorithm
3.9. Class A teaching as integer programming
3.10. Class A teaching as dynamic programming

4. CLASS B LEARNIG SYSTEMS:
ADVICE FREE, EXTERNAL REINFORCEMENT LEARNING AGENTS

4.1. Introduction
4.2. Associative Search Network
4.3. Actor-Critic Architecture

5. **CLASS C LEARNING SYSTEMS: SELF-REINFORCEMENT LEARNING AGENTS**

5.1. Conceptual framework
5.2. Self-reinforcement learning and NG agents
5.3. The Crossbar Adaptive Array class C architecture
5.4. How it works
5.4.1. The one-step computations
5.4.2. The sequence of the one-step computations
5.4.3. Defining primary goals from the genetic environment
5.4.4. The secondary reinforcement mechanism: backward chaining
5.4.5. The CAA learning method
5.4.6. The higher order system: Modulating actions and emotions
5.5. Example of a CAA architecture
5.5.1. Actions: learned and modulated
5.5.2. The state evaluation function
5.5.3. The learning and value backpropagating function
5.6. Solving problems with the CAA architecture
5.6.1. Learning in emotional graphs: Delayed reinforcement learning problem: Maze running example
5.6.2. Learning in loosely defined emotional graphs: The undeterministic environment problem: Pole balancing example
5.6.2.1. Representing the problem as emotional graph
5.6.2.2. Dynamics of the Cart-pole system
5.6.2.3. Parallel programming for pole-balancing learning
5.6.2.4. Some results of the experiments
5.6.2.5. The limited value backpropagation method for solving problems in learning in losely defined graphs
5.7. Another example of a CAA architecture: Greedy emotion CAA
5.8. Using entropy in Markov Decision Models
5.8.1. Compuing entropy with known transition probablities
5.8.2. Compuing entropy with unknown transition probablities
5.9. CAA as a neuro-genetic agent
5.9.1. Species vectors and subjective graphs
5.9.2. Two-chromosome genome CAA: Building an environment model
5.9.3. CAA architecture as optimization architecture
5.9.3.1. Convergence in deterministic environment
5.9.4. Distinstion from the genetic algorithms
5.9.5. Self-reinforcement based on the genetic environment

6. **RELEVENCE OF THE CAA TO THE DEVELOPMENT OF THE CONTEMPORARY REINFORCEMENT LEARNING THEORY**

6.1. After 1982...
6.2. Q-learning is a CAA-learning method
6.2.1. The problem: Credit assignment
6.2.2. The approach
6.2.3. Q-values
6.2.4. Q-learning: A special case of the CAA learning method
6.2.5. A taxonomy of CAA-method based learning algorithms
6.3. Producing optimal solution in stochastic environment
6.4. A summary of the observed relevance
6.4.1. CAA as a 1981 neural architecture
6.4.2. CAA as a state-of-the-art neural architecture

7. **CLOSING DISCUSSION**

7.1. Unified theory of learning agents
7.2. CAA architecture
7.3. Q-learning as a CAA learning
7.4. CAA as neuro-genetic agent
7.5. Adaptive Networks Group

8. **CONCLUDING REMARKS**

9. **APPENDICES**
9.A. First CAA written report, Nov 25, 1981
9.B. First talk about the CAA, Dec 2, 1981
9.C. First announcement of the CAA pole balancing solution, Dec 10
9.D. First published report, April 13, 1982
9.E. The early papers of the ANW group, 1981-1983

10. **REFERENCES**

CHAPTER 1

INTRODUCTION

In this chapter we will present the global concepts discussed in this report. The issues discussed are: the three-environment framework assumed in this work, the notion of agent and its architecture, and some issues on problem solving using particular agent.

1.1. THE FRAMEWORK

The main *motivation* for this part of the project is to explore some issues around the notion of **neuro-genetic agents**. Those are agents which for their function take advantage of a neural architecture but also rely on their connection to the genetic environment.

The general framework can be denoted as *three-environment framework,* as shown in Figure 1.1. According to the Figure 1.1. we assume existence of three environments: 1) the genetic environment, 2) the organismic environment, usualy represented by an *agent*, and 3) the behavioral environment, or some kind of reality, where the agent express its presence and/or behavior.

This framework assumes that all the three environments are performing some kind of an optimization process which reflects itself on the agent. The behavioral environment optimization loop is actualy a learning loop: it represents itself in the software structure of the agent. The genetic environment optimization loop is a hardware and firmware loop. It represents itself in the optimizing structure of the agent or variety of agents, and also in their primary intentions, drives (hunger, thirst, ...) underlying behavior. Also, it optimizes the solutions of some problems produced by the agent, as we discuss further in the text.

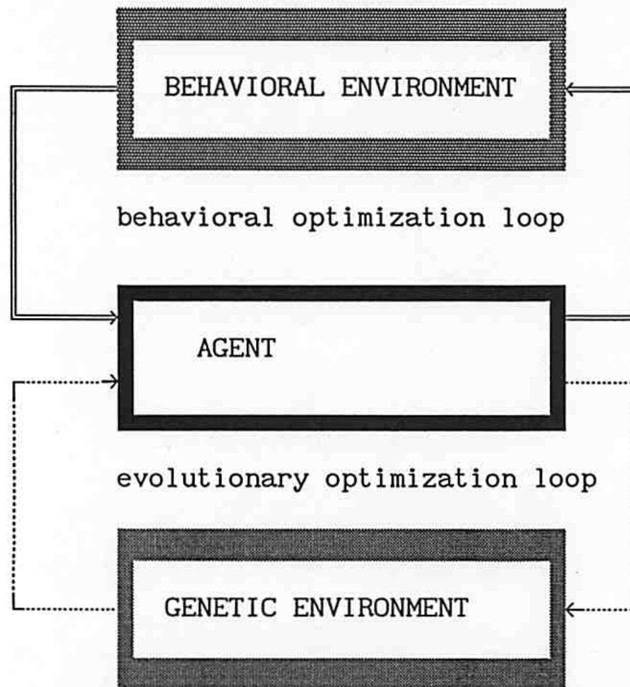


Figure 1.1. The three-environment framework

A number of issues can be pursued in such stated research framework. Let us mention that there is contemporary a remarkable research effort in the area of *genetic algorithms* [Holland 1975]. The problem considered in genetic algorithms research is to produce an agent at the genetic environment level and test its performance at the behavioral level. Usualy no optimization in sense of learning is considered at the behavioral level. If the performace is below some level, a probability exists that that agent (organism) will be removed and some other will be generated. The objective of the optimization process is to produce organisms which will express high level of performance.

This research will use the basic idea of the genetic algorithms but will be mainly focused on the learning process, i.e. the behavioral optimization loop. We will be interested in constructing artificial organisms, which will receive initial information from the genetic environment, which in turn will *bias* their learning behavior. After learning, the learned information is exported to the genetic environment in form of genetic message (genome). The exporting process contains an optimization mechanism. Alhough (as far as we know) there is no evidence of such a process in the nature, it seems a feasible hypothesis for building simple agents with small memory requirements.

1.2. AGENTS AND ARCHITECTURES

We will be interested in *learning agents* which have sophisticated design to the extent that they are able to solve some nontrivial problems in the behavioral environment in which they exist. The notion of *agent* will be assumed intuitively understandable in the sense of an organism or/and a social entity. The notion of an agent acting in an environment is a common paradigm in contemporary understanding of the intelligent systems [Kaelbling 1993, Bozinovski

1993, Russel and Norvig 1995]. An agent can represent an animal, robot, or other system with some assumed level of intelligence.

An agent has its functional *architecture.* The notion of architecture encompasses the conceptual relations among the sybsystems and elements of a system. The architecture can be represented at various level of abstraction, and by various types of basic elements. An architecture can be of neural type, of protein type, or some other even abstract type. Considering the functional architecture of an agent, an assessment can be made about the ability of the agent to survive in an environment or to solve some problems represented by the environment. The notion of architecture has implicit engineering sense: for a given problem we usualy *construct* an agent architecture which will be used to solve the considered problem.

1.3. NEURAL ARCHITECTURES

This work will be interested in agents having neural nerwork architectures. The research of artificial neural networks, starting by work of McCulloch and Pitts (1943) has been greatly intensified after the influential book of Rumelhart, McClelland and the PDP Group (1986). It is very difficult to make systematisation over the great number of neural network architectures proposed so far. An example of a successfull taxonomy is the attempt of Simpson (1990).

For purposes of this work, we will use only a general taxonomy dividing the neural networks in two architectural classes: 1) multilayer perceptrons and 2) adaptive arrays. Figure 1.2. shows these basic neural architectures.

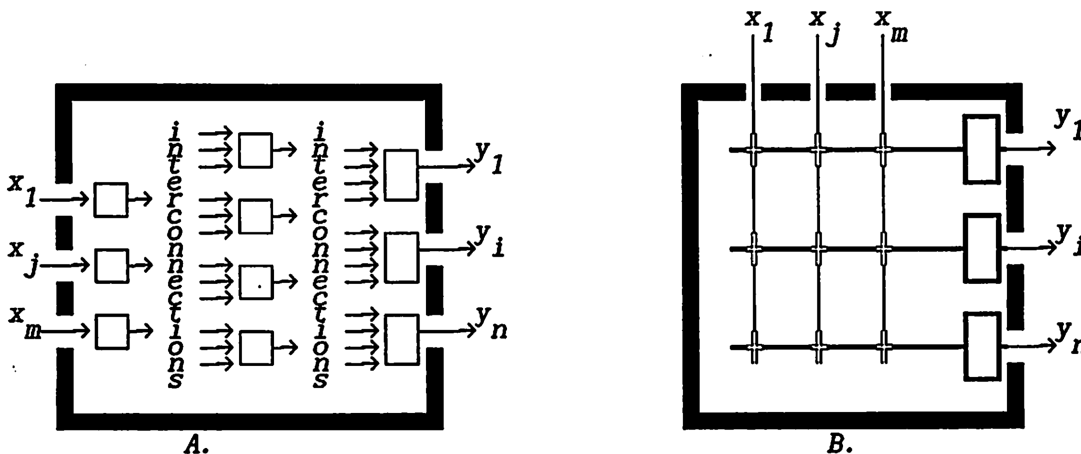


Figure 1.2. Basic neural network architectures.
A. Multilayer perceptron B. Adaptive array

Multilayer perceptrons are cascaded multistage networks, where basic processing units, neurons, are placed in layers. The usual terminology is that we should recognize and input layer and the output layer. There are several layers in between, denoted as hidden layers.

Adaptive arrays are matrix memory structures. The neurons are represented with their *dendritic structure*, to which input signals are connected. The matrix is usualy refered to as synaptic matrix,

or learning matrix [Steinbuch 1960]. The synaptic weights {w} (on the Figure 1.2. represented by the symbol ╫) are indexed in a straight-forward manner, by the indices $i \in [1,..,n]$ representing outputs, and $j \in [1,..,m]$ representing inputs.

This report will be concerned with adaptive arrays, although the multilayer perceptron representation will also be mentioned.

Important issue of an architecture is the interpretation of its input and output set of signals. For our purpose we adopt that the input signals represent a situation (or set of possible situations) and the output an action (or set of admissible actions) performed by the agent.

1.3.1. CROSSBAR ADAPTIVE ARRAYS

A special class of neural architectures, denoted as crossbar adaptive arrays, will be of particular interest in this text. Figure 1.3. shows example of such a crossbar architecture.

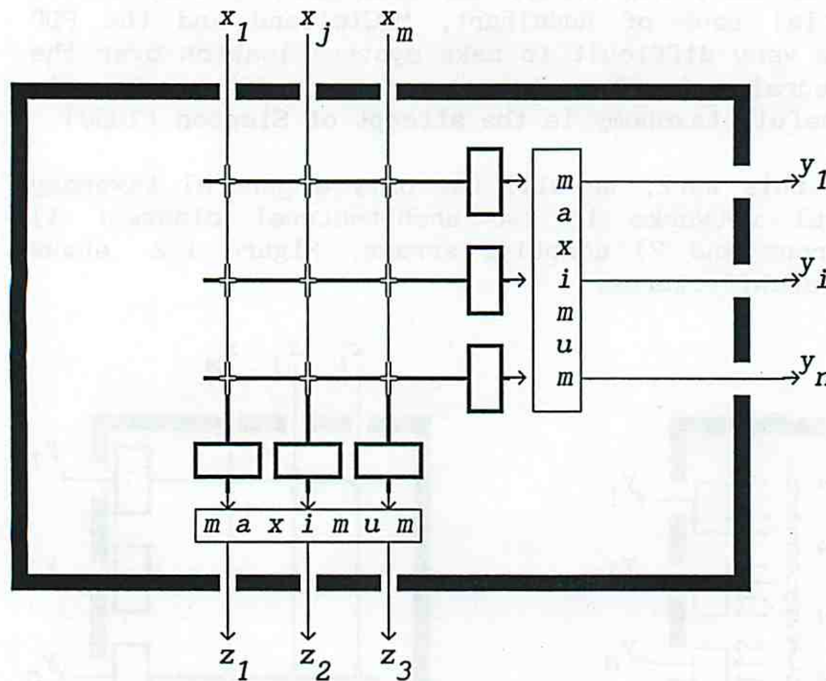


Figure 1.3. A crossbar neural architecture

The architecture shown on Figure 1.3. computes two different types of outputs. Several interpretations can be given for the nature of the different types of actions. We can think of y-actions as *physical* actions and of z-actions as *emotional* (and/or *mental*) actions. For example, the y-actions can be used for moving objects in the behavioral environment, whereas the z-actions can be used to represent an emotion, internal evaluation of some state of the agent and/or the environment.

The architecture shown on Figure 1.3. also computes a maximum function over the set of actions. In general case, for this type of architecture it is not neccesary that the maximum selector function be used. Some other function, depending on the problem challenged,

can also be defined.

The crossbar adaptive array architectures can also have *recurrent* connections. Figure 1.4 shows an architecture which will be of major interest in this report.

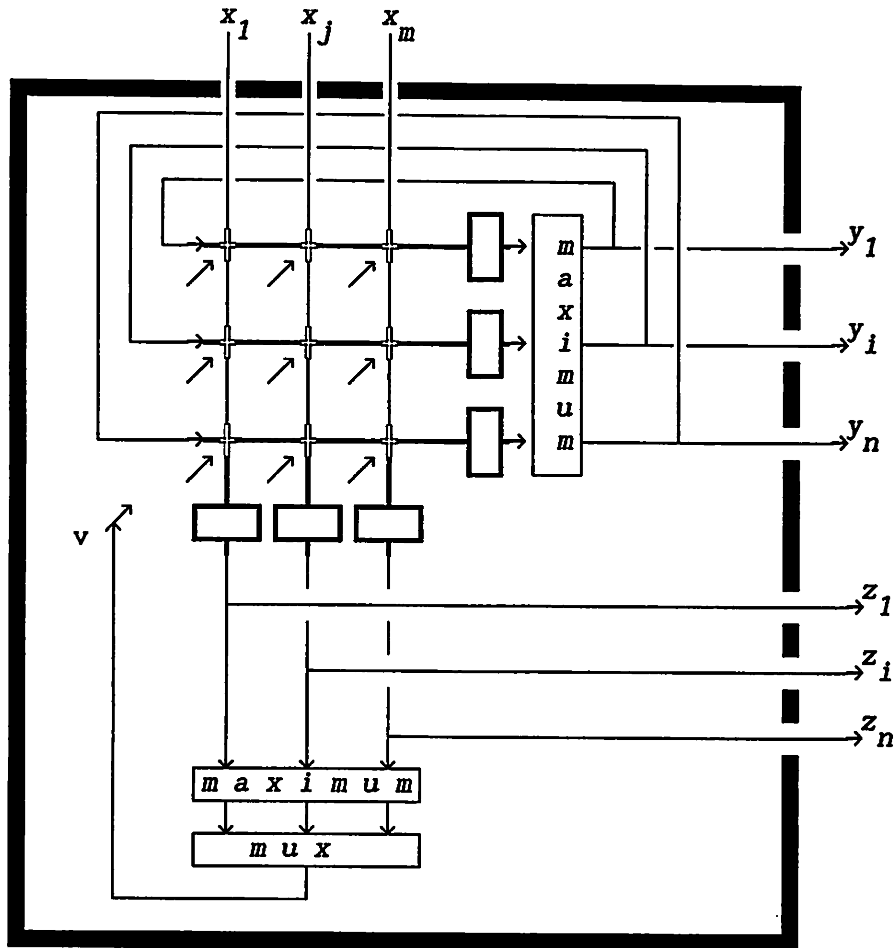


Figure 1.4. A recurrent crossbar adaptive array

The architecture on Figure 1.4. has recurrent connections toward its memory matrix. It has also two types of outputs, interpreted as physical actions and emotional actions. The emotional actions in this architecture are expressed toward the environment, possibly shown to some other agent. In addition, a computation is performed to extract the most dominant emotional component and to feed it back to the system. It is internal emotion of the system, and in this architecture it is *broadcast* back to the memory matrix, such that it will have an impact on the learning process. In some interfaces with the environment, the vector z is not expressed outside the agent.

Let us note that the emotional computation is not neccessarily made by the functions *maximum* and *multiplex*. Some other functions can be used in this architecture. Example is a function *sum,* which will sum the feelings and somehow give a global emotional state of the system. A function *average* can also be used. What kind of emotion computing functions will be chosen will depend on some higher order emotion controlling strategy, which in turn depends on the considered task objectives.

1.4. PROBLEMS, SOLUTIONS, AND TOOLS

A notion of *problem* is central in Artificial Intelligence (AI) and many textbooks in AI (e.g. Russel and Norvig 1995) consider that notion as an introductory one in exploring intelligence. Assuming the notion as intuitively understandable, we will observe that the notion of *problem* is always contingent with the notion of *solution* of that problem.

An important issue around the concept of problem is its *representation*. There are various ways a problem can be described and represented. A standard problem representation in AI is the state space graph, where there is given a set S of states, a subset $S.S$ of starting states, and a subset $F.S$ of final states. Some of the final states are *goal states*, i.e. $G.S \subseteq F.S$ where $G.S$ is a set of goal states. The *states* are arranged such that they are nodes of a *graph*, in which the edges are denoted as *actions* or operators, with interpretation that if being in a state $s \in S$ and taking action $a \in A(s)$ will cause a *transition* from the state s to some other state $s' \in S$. Here $A(s)$ is a set of admisible actions being in the state s. A solution of the problem in this representation concept, is a path between some given starting state and some given final state. Sometimes the problem is stated such that more starting states and/or more final states can apear in the solution(s) of the problem.

An important class of state space graphs are the graphs where the edges and/or nodes are assigned elements of some (partially) ordered set, for example real numbers. If numbers are assigned to actions, the graph is usualy denoted as "weighted". If numbers are assigned to states, the graph is usualy denoted as "valued" or "colored". Such a set can be the set $\mathcal{E}$ of emotionally represented (cartooned) human faces, and the partial ordering relation can be the subjective human (e.g reader's) judgement. Figure 1.5 shows an example of a *subjective emotional graph*.

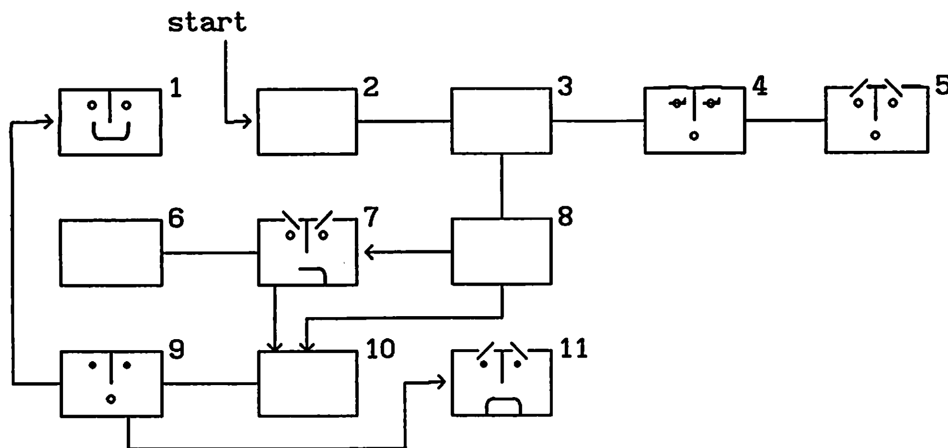


Figure 1.5. Example of an emotional graph

The graph on Figure 1.10 shows that some transitions between states are unidirectional, and some bidirectional. Some states are terminal, as state 1 and 11, and state 2 is a starting state. The goal state is the state 1. A solution is found if *a path* from state

2 to state 1 is found. Emotional graphs are very useful for qualitative analysis and visualization of a problem. They show which states are *desirable*, and which are not desirable. Having those notions, the solution of a problem can be seen as a path from a starting state to a goal state *while avoiding* undesirable states. It emphasizes the *subjectivity of agent's evaluation* of the environment states. It also implies that two agents can view the same envronment state with different emotion. In the sequel this representational concept [Bozinovski 1982b] will be used whenever possible.

We will also distinguish between arbitrary and *optimal* solutions. An arbitrary solution is any solution of the problem. Consider going from Amherst to Boston. If the problem is solved somehow, and an agent is moved from Amherst to Boston, then it is a solution of the problem. If a problem assumes *constraints*, for example going to Boston but not using bicycle, than any solution which solve this problem is *solution under constraints*. Often such types of solutions can be denoted as *satisfactory* solutions. An important class of problems are problems where some *optimality criterion* is stated. For example, going to Boston as fast as possible. Here the time spent for traveling to Boston is measured, and is used as a measure of performance of the problem solving agent. In so stated problems we usualy already have a set of possible solutions; what is interesting is the "best" one, according to some criterion. Usualy, problems are stated both with constrains and an optimization criterion.

The notion of solution is assumed to be accompanied by the notion of *tools* for solution of a problem. Although it is not widely emphasized in AI, the tool is an important issue in *statement* of a problem. Although we can state a problem not specifying tools for its solution, in real life we always assume a set of tools given to us when we are going to solve a problem. Tools can be understood as given constraints in the statement of a problem. Simple example is stating a problem of solving 2+3 in ordinary sense. If no tool is specified, it is an easy problem, we will use our brain processors, where that pattern is almost associated with some tabular value; the arithmetic routines of our brain will probably not even be activated. The problem of 2+3 is *trivial* and *not interesting* if the assumed tool is our brain. But, if we state the same problem using a Neural Architecture, then the problem becomes more complicated, and for some people maybe interesting. So, specifying the tool as a part of a problem can formulate a new problem, with possibly challenging nature.

In that conext, this report is about **crossbar adaptive array architectures as tools** *for studying and solving some problems.* The general problem of learning, and some important paradigms of learning, as learning with a teacher which gives advice, learning with a teacher which only gives reinforcement, and learning with no teacher at all, will be considered.

Further in the text, a separate chapter will be dedicated to those problems and agent architectures used for solutions.

CHAPTER 2

A STRUCTURAL THEORY OF LEARNING AGENTS

It is assumed that an agent (or actor) *acts* in an environment. Over time, an *abstract impartial observer* outside the agent and the environment, could observe *a behavior* of the agent with respect to the considered environment. The environment in which a behavior of an agent is observed is the agent's *behavioral environment*. For example, for humans the behavioral environment can be the social environment if we consider social relationships. For some philosophical discussion we can extend the behavioral environment to be the "reality" in which the agent exists: the reality can be "real", dreamed, or computer generated (so-called virtual). We can assign the behavioral environment to a certain problem space relevant for some considered task. For a robot, the behavioral environment can be its working environment in some flexible production system. For a neural network controller, the behavioral environment can be the plant or process it controls.

An agent can express a *learning behavior* in some environments. We assume *learning* is a process which represents itself on at least two levels:

1) in the knowledge base (memory) of the learning agent: a *portion of knowledge is being gained* which contributes toward the understanding (*building a model*) of the environment

2) in the observers memory (it can be a scratch-pad memory or some longer term memory): *the entropy of the agent's behavior is decreased*. The behavior is shifted from possibly totaly random to possibly totaly deterministic.

2.1. BASIC NOTIONS

The basic conceptual variables of the theory we are going to present are given on Figure 2.1.

On that Figure, S is a set of signals which at time step t is presented from a behavioral environment to a learning agent.

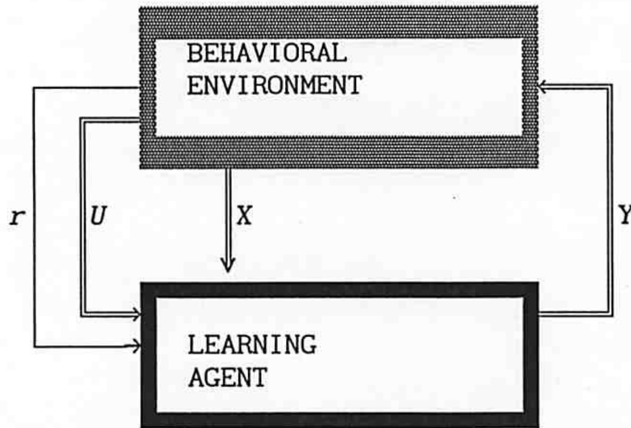


Figure 2.1. The basic inteface concepts

In this theory we assume that the set S consists of three subsets, i.e.

$$S = \{r, U, X\} \tag{2.1}$$

where

r is the *reinforcement*: it is a distinguishable signal in S which is usualy presented to the learner to show how good the learner's performance is according to the evironment; also it is a variable which the agent will tend to *optimize* during the learning process.

U is *action advice*: it is a distinguishable signal, or set of signals, representing the action which should be performed by the learning system in a situation specified by the environment

X is the *situation*: the set of signals $S-\{r,U\}$; usually considered as "neutral" signals, or context signals

We say that S is a *generalized situation*. We assume that it represents the *state of the environment* at the observed time t. Below we distinguish between the term generalized situation, S, and situation, X. We will operate with the situation X, unless otherwise specified. Later we will consider environments where S=X. In these environments, X is a situation which represents the state of the environment as seen by the agent.

Further, we assume that the acting agent generates the values of a set of signals, Y, representing an *action* toward the environment.

2.2. LEARNING INTERFACES AND A TAXONOMY OF THE LEARNING PARADIGMS

The pair

$$LI = \{S, Y\} \tag{2.2}$$

denotes the *learning interface* between the environment and a learning agent. Note that here the learning interface is represented as a set. However, in some cases it is convenient to consider LI as

ordered n-tuple to emphasize the sequence of events appearing in the learning process.

In this theory we will consider generalized situations where not all of the three components are present. Let us discuss briefly some learning paradigms which are defined by ommiting some components of S. We now present 9 learning interfaces. The first two are interfaces where X does not appear. We call them *context-free* interfaces. The interfaces where X appears we denote as context sensitive, or *context dependent*, interfaces.

LI1 = (U,Y). This interface usualy represents a *forced training paradigm*. There is a teacher which gives an advice as to what action should be performed. There is no X-Y association learning. The learner can learn to repeat the actions advised by the teacher. This is a primitive type of *imitation* learning.

LI2 = (Y,r). This is a typical example of *context-free reinforcement learning*. The learning agent *tries* actions in order to optimize the reinforcement signal. This learning paradigm is used in learning automata and function optimization.

LI3 = $([X,U],Y)$. In this paradigm, the situation X and the advice U appear *simultaneously*, as a pair. The teacher forces the learner to adjust the mapping X-Y to be the desired mapping X-U. This is sometimes called supervised learning paradigm. This paradigm was widely used in associative memory programming paradigm.

LI4 = (X,U,Y). In this paradigm it is important to emphasize that between X and U there is a time difference. This paradigm is used in classical conditioning modeling [Sutton and Barto 1981].

LI5 = (X,Y,U). This paradigm in contemporary theory is used in the backpropagation shemes of neural learning [Rumelhart, Hinton and Wiliams 1986]. The learner receives the situation X, produces an action Y, and receives the advised action from the teacher. The learner, after that, computes the error and backpropagates the error through the network. Important property of the paradigm is that the advice is supplied in each step.

LI6 = (X,Y,r). This is the *context-dependent reinforcement learning paradigm*. It is the classical reinforcement learning paradigm. The agent learns to associate the appropriate action to the situation X, in order to optimize the reinforcement r. This paradigm is also a model of the instrumental conditioning paradim in animal learning theory. [Barto, Sutton, Brouwer 81].

LI7 = $(X,Y,r,(U))$. This paradigm we denote as *tutorial learning paradigm*. The main property is that the teacher gives advice *only if neccesary*, and not in each step. The learner is assumed to know what to do. If, in an examination trial, there is something the learner doesn't know, then it first receives a negative reinforcement r, (for example the word "No!"), and after that advice as to what to do. The learner tends to adjust its X-Y mapping toward the requested X-U mapping, and also to minimize the training trials and the amount of negative reinforcement received. This paradigm was investigated in [Bozinovski 1981b].

LI8=(X,Y,(r)). This is a *delayed reinforcement learning paradigm.* The learner receives a reinforcement only occasionaly. This paradigm is of interest in contemporary reinforcement learning theory.

LI9 = (X,Y). This is the case of a *self-learning paradigm.* The environment represents its state only by a set of neutral signals. Neither advice nor reinforcement is given, not even delayed one. The learning agent should develop its internal reinforcing mechanism in order to exhibit a learning behavior. There should be some mechanism inside the agent which will evaluate the state of the environment. This paradigm is investigated by Bozinovski (1982). A chapter of this book is devoted to that paradigm.

Let us just note that the paradigm LI9 should be considered distinct from the so-called "unsupervised learning", where the input data is clustered according to some measure of similarity. There is no point of arguing about names, but we believe the term *adaptive clustering* should be used instead of the term unsupervised learning in clustering tasks.

2.3. CONSEQUENCE LEARNING SYSTEMS

In the further discussion we will be interested in the latter four paradigms: LI6 till L9. Those are *context dependent consequence learning interfaces.* In LI6, LI7 and L8,a special reinforcing signal r comes as part of the consequence. In LI9 only the "neutral" situation X comes as a consequence. Figure 2.2 shows the three interfaces which encompass the mentioned learning paradigms. They define three *classes* of learning agents.

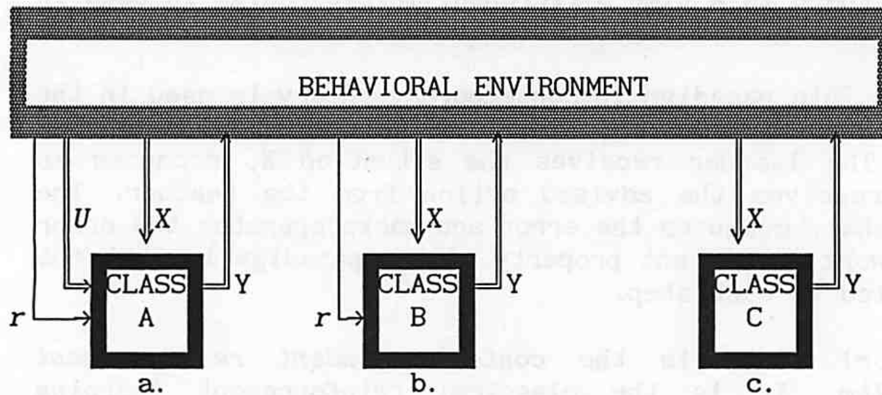


Figure 2.2. Context-dependent consequence learning agents

In Figure 2.2.a. we have an agent which receives evaluation (reinforcement) about the previous performance, and also advice *U* for the action Y to be performed in the situation X. We denote it as a *class A* agent. *Class B agents*, in contrast, receive only the reinforcement which the learner tends to optimize. *Class C agents* can learn in an advice-free and reinforcement-free environemnts.

We will use the observation from Figure 2.2 to define a generic learning agent which will be able to learn in all the mentioned interfaces.

2.4. A GENERIC LEARNING AGENT

Figure 2.3 shows the concept of our generic learning agent, which we call *Neuro-Genetic Agent* (*NG agent*, or *NGA*). The name suggests the dependence of the genetic environment. (A more suitable name would be maybe "Geneto-Neural Agent" but somehow it does not sounds).

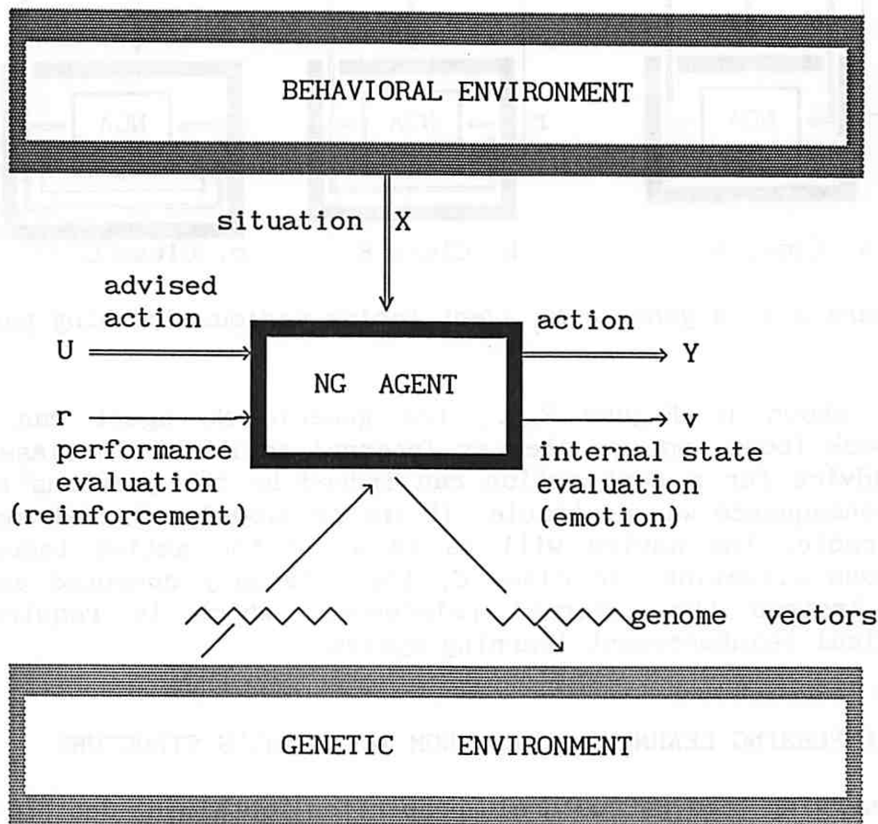


Figure 2.3. Generic learning system

As Figure 2.3 shows, we assume that our generic learning agent is a reinforcement learning system. In addition to the previously mentioned inteface components, the NG agent computes its *emotion*. The emotion is actualy the *value of the state* the agent is in, evaluated by the *agent itself*. An agent needs that component in order to build up an *internal reinforcing mechanism* when facing a non-advising and non-reinforcing environment.

The genetic environment is needed to supply the NG agent with the basic drives, primary instincts and drives, basing on which the agent can learn to behave to fullfill the goals stated by the drives. The prinary drives are supplied by means of genome vectors which are imported from the gentic environment. It is assumed that genome vectors can also be exported toward the environment.

Using the generic NG agent concept shown on Figure 2.3 we will now show the simple idea which allows the generic NG agent to perform as class A, B and C agent. Figure 2.4 gives the solution.

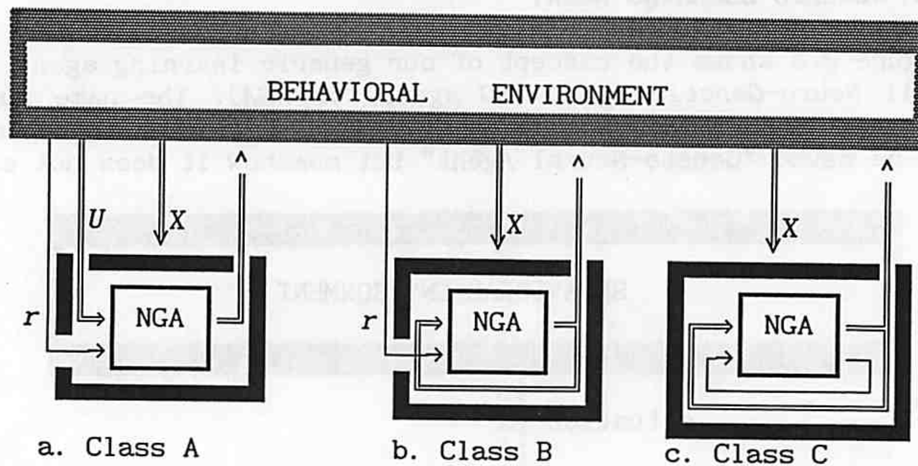


Figure 2.4. A generic NG agent facing various learning paradigms

As shown in Figure 2.4., the generic NG agent can develop feedback loops and *use them as internal advisers.* In class B and C the advice for a next action can indeed *be* the previous action if the consequence was desirable. If the previously elicted emotion was undesrable, the advice will be to avoid the action taken in the previous situation. In class C, the internaly computed emotion is used instead the external reinforcer, which is required in a classical reinforcement learning system.

2.5. INFERRING LEARNING RULES FROM THE AGENT'S STRUCTURE

Observing the structural properties obtained due to defined feedback loops, we can now recognize some neccesary conditions for the learning rules for the agents in class A, B and C.

Let W be a memory variable which represents the associativity strength between the situation X and action Y in time step t. The equation

$$\Delta W = cf(X,Y,U,r,v) - dg(W,X) \qquad (2.3.1)$$

$$W(0) = W_o \qquad (2.3.2)$$

is usualy denoted as *learning rule.* Here $\Delta W = \Delta W(t-1) = W(t)-W(t-1)$, c and d are positive constants, and f and g some functions. Other simbols are defined in Figure 2.3. The first term in the equation (2.3.1) we denote as *reinforcing (or refreshing)* term, and the second one we denote as *forgeting (or extinction)* term [Bozinovski 1981b]. In both functions f and g there can be a time difference for some parameters in the equation; for example, the action Y can appear as a term $Y(t-1)$. Often, the generic equation (2.3.1) is written in the form

$$W(t) = W(t-1) + h(f'(U) - g'(W,X))X \qquad (2.4)$$

which is known as *error-correction form.* In that form $f'(U)$ is the a function of desired (target) behavior, and $g'(W,X)$ represents the current behavior. Interpretation is that the current behavior is changing as to reduce the discrepancy with the desired behavior. The

parameter h is the learning step size parameter. Other interpretation is often given in terms of function approximation: function $f'(U)$ is gradualy approximated by a function $g'(W,X)$ incrementing the learning variable W.

For the discussion that immediately follows we will not be concerned with the forgetting term (see 2.3.1), and we set d=0. Further in the text this term will be addressed.

As a results of the analysis in the previous section, we can see that the class A has no feedbacks. So no time difference will neccessary appear in the learning rules of this class agents. Example of a learning rule is

$$\Delta W(t) = cX(t)U(t)r(t) \qquad (2.5)$$

However, a time difference can appear if some variable is *buffered*, i.e. stored for a limited time steps. For example, if X is buffered from the previous step, the rule can be of form

$$\Delta W(t) = cX(t-1)U(t)r(t). \qquad (2.6)$$

A bufer of different size can be used to remember several steps behind; most common is the one-step-behind buffer.

For class B agents, facing the classical *reinforcement learning* paradigm, the previous action is used as an advice for some next actions. Thus, in the learning rule of a reinforcement learning system, must appear a term $Y(t-k)$, $k \geq 1$. Examples of such learning rules are

$$\Delta W(t) = cX(t)Y(t-1)r(t) \qquad (2.7)$$

and

$$\Delta W(t) = cX(t-1)Y(t-1)r(t), \qquad (2.8)$$

where c is some constant or additionaly computed function. In equation (2.8) a term X also appears as time shift not neccesarily due to a feedback; it can be obtained by some buffer. The learning rule (2.8) is an associative learning rule, and is used in learning tasks about which we will talk in this text. The larning rule (2.7) has not been used as far as we know. It only predicted by this theory as possible. It could make sense in some schemes of learning automata application, where the consequence $X(t)$ is searched for by using the action Y. Also, in classical reinforcement learning the learning rules can be more complex, and higher order then the first order rules shown here. However, the contemporary reinforcement learning theory uses mainly the first order learning rules.

For the class C system (*subjective consequence learning system*) we have two loops in the architecture. Thus, if we want to build such a learning system, in the learning rule a term $X(t-h)Y(t-k)$, $h,k \geq 1$ should appear. An example of this rule is

$$\Delta W(t) = cX(t-1)Y(t-1)v(X(t)), \qquad (2.9)$$

where $v(X(t))$ is the internaly computed *value of the internal state* of the NG agent due to the received situation $X(t)$, which in turn

is a consequence of the state-action pair $X(t-1)Y(t-1)$. We call the learning rule (2.9) *state-action-concequence (SAC) learning rule*. This is the rule used in the self-reinforcement learning systems. A forgetting term can also be present.

In that way, as a result of this theory, we developed learning rules *from the structure*. In such a way, we defined learning rules for the NG agents which will be considered further in the text.

2.6. STRUCTURAL ADAPTATION OF THE NG AGENTS

This theory allows an assumption of *structural adjustment* toward an environment. We can easily visualise that a class A architecture can evolve to a class B architecture, and further to class C architecture by building feedback connections. Figure 2.5. illustrates that structural transformation.

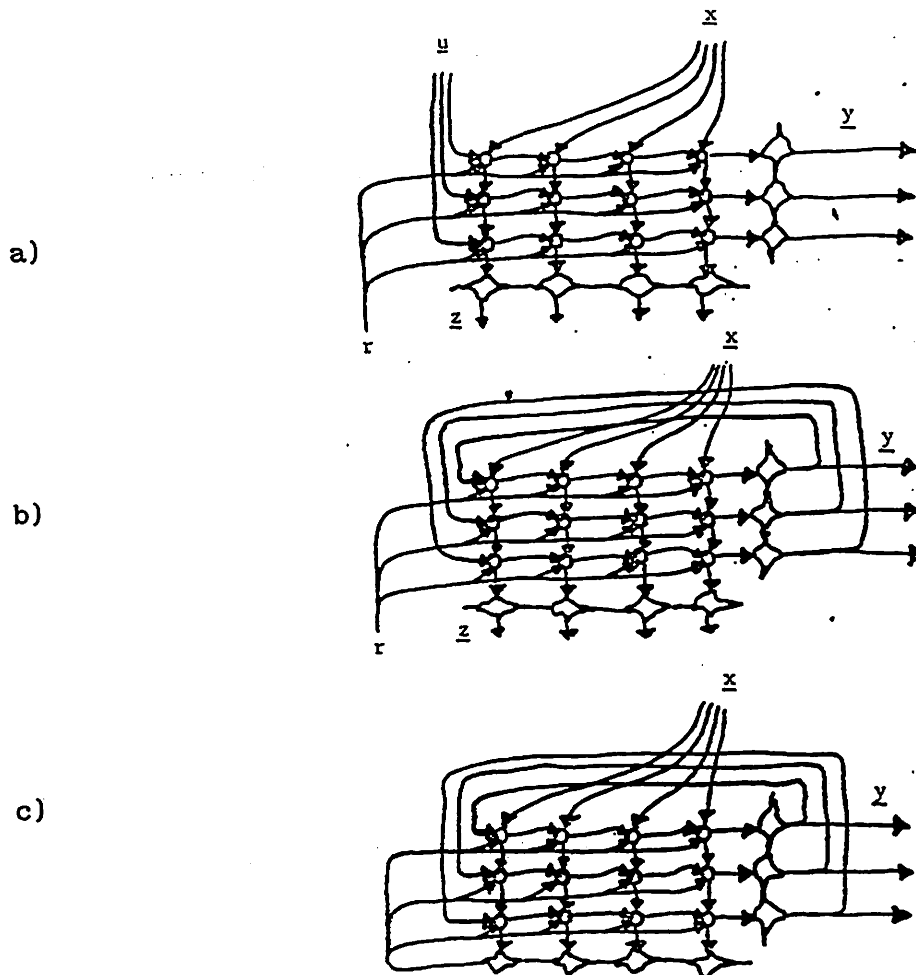


Figure 2.5. Structural transformation of the NG agent, represented as a crossbar neural network. a) class A, b) class B c) class C

As shown in Figure 2.5, a NG neural network of class A can develop recurrent axon connections toward the inputs and can be transformed into a class B neural network. Also, the class B neural

network can transform itself into class C, by apropriate adjustemend of its connections from the state evaluation outputs toward the reinforcement input.

A neuro-genetic mechanism which can justify this assumption is dendritic filament growing, of microtubili growing. A pure neural mechanism could be another set of adjustable connectionist weights, which are used for programming the neural architecture to transform its structural class. Such a transformation can be reversible.

In the next chapters, within this theoretical framework, we will consider some examples of class A, class B, and class C agents.

What we would like to stress is that the systems described later in this text can be represented as a particular NG modification of the generic NG agent defined in this chapter. In a sense, it represents *an architecture of an unified view toward the learning paradigms.*

CHAPTER 3

CLASS A LEARNING AGENTS:

ADVICE TAKING REINFORCEMENT LEARNING SYSTEMS

In this chapter we will describe our work on the class A learning systems. Those are systems which are taking advice (or request, or the desired answer) from some kind of teacher or *reference model.* The goal of the process is to transfer the reference knowledge of the teacher to the knowledge base of the learner.

There are a number of issues going around this paradigm and a large amount of research effort has been allocated to this paradigm in control theory, neural learning, pattern recognition learning among others. It is not the intent of this report to review that effort. Here we will describe our research in this area, since it is one line of the beackground which lead to the design of the CAA architecture described later.

Firstly we will describe some of the learner's architecture we have studied in connection with the class A learning paradigm. After that we will give some directions within the teaching system theory we developed in connection with this paradigm. The teaching system theory described here was mainly developed within the Adaptive Networks Group, although some initial results were stated before.

3.1. CLASS A LEARNING AGENTS

In the previous section we defined a NG agent to be an agent capable of learning in all the defined learning classes. Figure 3.1. shows the a NG agent working as a class A learning system.

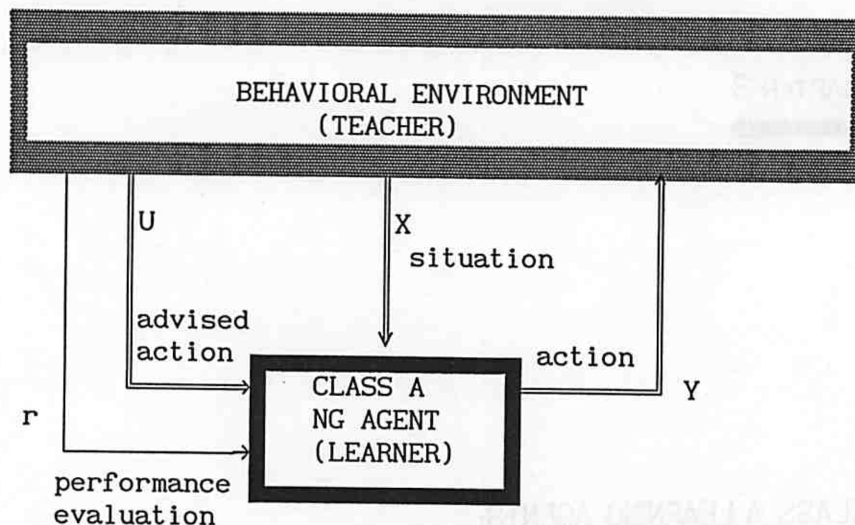


Figure 3.1. Class A learning paradigm

In the class A learning paradigm, the environment supplies complete information about the desired behavior of the learner. The genetic environment is always present, but here it is not crucial, and we ommited it from the picture. For arbitrary initial conditions received from the genetic environment, the class A learning system can be directed toward the desired behavior, provided that the learner accepts the teacher's advice.

We will be interested in paradigms where the temporal sequence of events is X,Y,r,U, which we call tutorial learning paradigms. In this section we will describe an algorithm, denoted as *tutorial algorithm*, which we used in our research with class A systems. Before that, we will describe the type of learners which we dealt with in our previous work.

3.2. GREEDY-POLICY PERCEPTRONS

Here we will describe a class A learning system known as perceptron [Rosenblatt 1958]. Actualy, the perceptron epitomizes a broad class of neural networks, and there are different visualisations about what perceptron is. Our standpoint is that pattern classifying machines which use neuron-like elements should be considered a type of perceptron. Early works usualy consider perceptrons which can classify input patterns into two classes. Example of such a perceptron is the Adaline network [Widrow and Hoff 1960], among others. We call it Adaline-type perceptron. Our work since the begining [Bozinovski 1972a] was concentrated with perceptrons which could classify into *several* classes and which use the principle of maximum-selector. We call those perceptrons *greedy policy perceptrons*, or sometimes *pandemonium-type perceptrons*, due to Selfridge [Selfridge 1959] who we believe introduced maximum selector in the neural network research. The term greedy policy is used from dynamic programming, on which we will talk later.

Figure 3.2. gives a description of the early perceptrons on which we started our neural network research, in 1971.

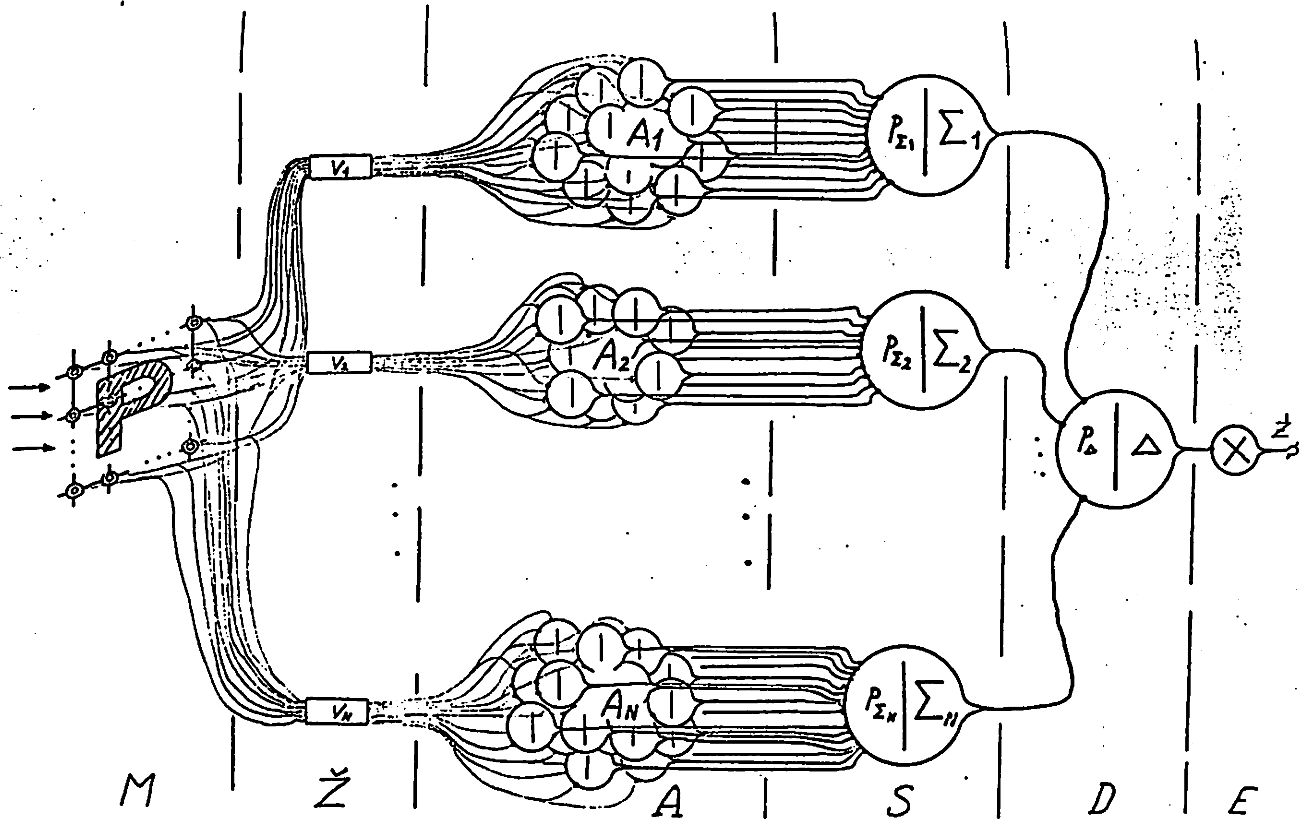


Figure 3.2. Pandemonium-type perceptron
(from Bozinovski 1974)

The perceptron shown in Figure 3.2. consists of several sets of *A-elements* (Rosenblatt 1958). They recognize features (or predicats [Minsky and Papert 1969]) as well as patterns as a whole. An A-element has excitatory and inhibitory inputs. The state of an A-element is given by a real number w. Its output is produced when sum of excitatory inputs exceeds the sum of inhibitory inputs by a given threshold value; in that case the output is equal to w. In other words, the A-element, if properly excited, ***tells its state.***

The output of all the A-elements representing a pattern (or a pattern predicate) is received by a S-element. This element computes the sum of inputs, and if the sum is above given threshold, it proceeds the sum as an output signal.

A special element, which we denoted as Δ-element, detects the maximum of all the received sums. If the sum representing the i-th pattern is maximum, then the output of the system is i. If there is no single maximal element, the output is 0.

Each pattern is represented as a binary matrix M, named as retina. Each element of the retina is named receptor. For each A-element a ternary matrix V is given representing its connection to the retina. The dimension of V is the same as dimension of M. An

entry 1 in the matrix V means that the A-element is connected to the respective receptor in the retina. If the entry is -1 that means that an inhibitory input of the A-element is connected on that receptor of the retina.

The system shown in Figure 3.2. can classify m patterns into n classes , m ≥ n. It can also be used to recognize predicates (features) about patterns (Minsky & Papert 1969). In our feature recognition experiments (Bozinovski 1972a), we used this network to recognize feature "horizontal line" and "vertical line" regardless on where on the retina such a feature exists. In our pattern classification exeriments (Bozinovski 1974) we used this network to recognize letters from aplhabet, as they appear on various computer terminals. In both cases we used training trials in the temporal sequence X,Y,r,U.

3.3. HARDWARE DESIGNS: ADAPTIVE ARRAYS

During 1972-1974 important part of our work was devoted to build neural units and networks in hardware. We developed a hardware simulator of a McCulloch-Pits neuron [McCulloch and Pitts, 1944] with modifiable weights and threshold, and receptor neuron as voltage controlled oscillator, both in discrete (transistor) technique [Bozinovski 1972c]. Also we developed in hardware, in integrated circuits [Bozinovski and Mesaric 1972b], a network simulating classical conditioning according to Pavlov [Pavlov 1927]. Also we developed a complete design of the perceptron as shown on Figure 3.2 in integrated circuits hardware, but as difference from the previous mentioned models which were physicaly built, the perceptron model was left as paper design only. However the design was detailed up to the timing diagrams and all the logical equations [Bozinovski 1974].

The work in hardware was usefull for establishing a notion of *adaptive arrays*. Also it was important to understand deeply the temporal relationships in an adaptive array, which was very helpfull later during our development of the Crossbar Adaptive Array. In the two subsections that follow, we will describe briefly the harware realizations of our classical conditioning model and the perceptron model.

3.3.1. A CLASSICAL CONDITIONING MODEL

Our hardware implementation of classical conditioning is given in Figure 3.3.

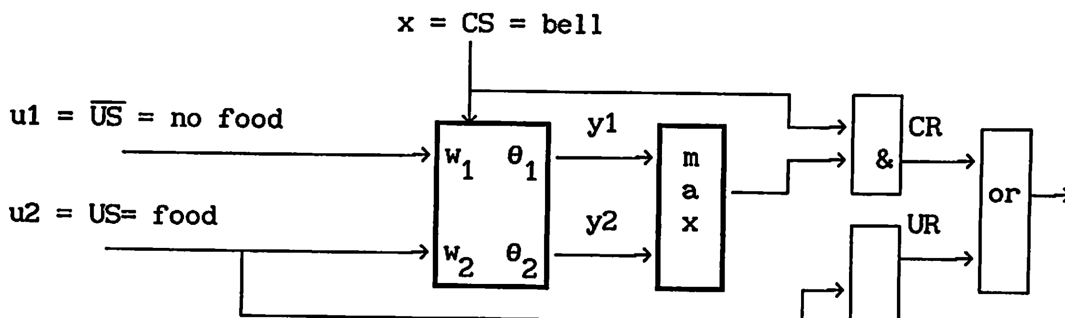


Figure 3.3. A simple model of Pavlovean conditioning

The neurons used have the following learning rule (x and u binary)

$$w(t+1) = w(t) + x(t)u(t) , \qquad (3.1)$$

and the following output rule

$$y(t) = w(t) \text{ if } w(t)x(t) > \theta, \quad 0 \text{ if not} \qquad (3.2)$$

Note that we actualy used A-elements as neural elements.

The learning rule (3.1) was easy to implement in digital hardware using only up-counters. This however limits the number of training trials to the limit of the counters capacity. In each experiment the counters should be reset before the experiment starts.

For the design of the digital counters which represented the weights w1 and w2, we proposed and tested the memory elements realized solely by inverters, taking advantage of WIRED-OR feature of some chip technologies. Figure 3.4. shows our design. It was also stated that if a technology has a WIRED-OR feature, then all the circuits can be built by inverters only.

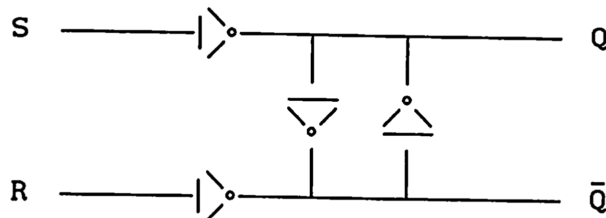


Figure 3.4. Our inverter-only design of a flip-flop

Let us note that it was a pre-Rescorla-Wagner [Rescorla and Wagner 1972] model of classical conditioning, and it that sense naive comparatively to todays knowledge about classical conditioning. Also, other models of conditioned reflexes existed at that time (Uttley 1962). However, it is interesting that it was built in digital hardware. It contributed toward our understanding of adaptive arrays.

3.3.2. CLASS A ADAPTIVE ARRAY

The notion of *adaptive arrays* become natural during our work on hardware design of the greeedy policy perceptrons (Bozinovski 1974, 76, 78). Figure 3.5. shows such a design.

The Figure 3.5. is a figure from [Bozinovski 1978] and it uses notation Z instead of the notation U, which we use in this text as the advising input coming from the teacher. In the further text we will use the notation U.

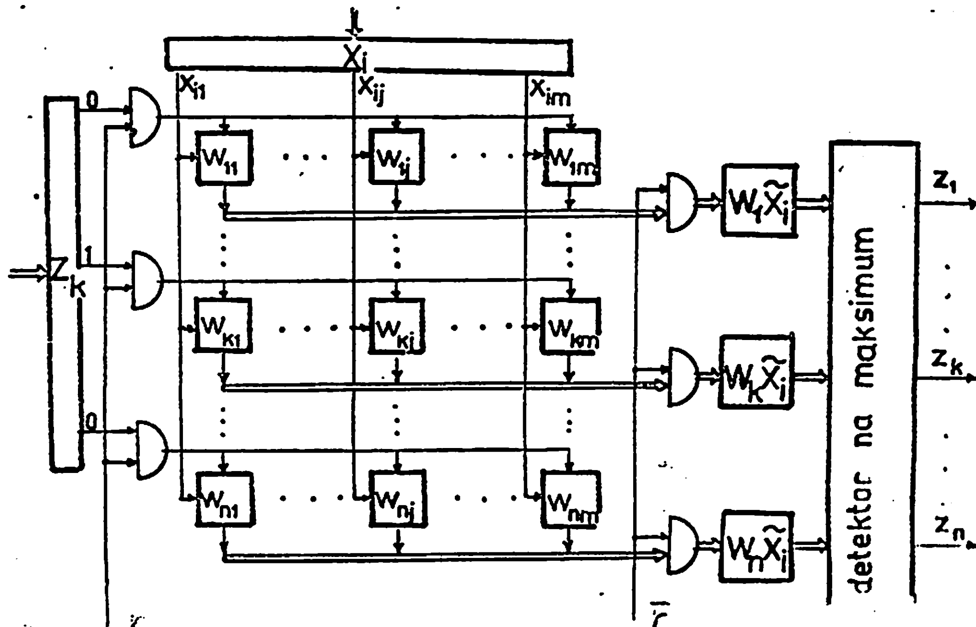


Figure 3.5. Class A adaptive array
(From [Bozinovski 76, 78])

The learning rule used in the development of the adaptive array shown on Fugure 3.5. is a three component learning rule [Bozinovski 1978]

$$\Delta W(t) = X_i(t)U_i(t)r(t) \qquad (3.3)$$

where r is a binary *penalty* applied if in the previous trial the pattern X_i was *not* recognized, and U_i is the advising input, showing in which class the pattern should have been classified.

Also it is important to note that the design on Figure 3.4 uses buffers to remember the previous inputs. Having that, we can have the learning rule

$$\Delta W(t) = X_i(t-1)U_i(t)r(t) \qquad (3.4)$$

as we stated in the previous chapter.

This adaptive array we denote as One-way Adaptive Array (OAA), to be distinguished from the Crossbar Adaptive Array (CAA) which we developed later.

3.3.3. ISOTHRESHOLD ADAPTIVE NETWORK

In the time of our hardware design of neural networks the question arised about the neural design on the maximum selector element. It was not trivial to be realized in digital hardware and required a lot of connetions as the number of classes to be recognized grows. A short study about that with a boolean expression solving the problem is given in [Bozinovski and Mesaric 1972]. However it was obvious that the neural mechanism for that is not of digital nature but of analog one. A possible solution is assuming a

separate neuron with maximum selector ability instead of summing ability [Bozinovski 1974, see Figure 3.2]. We proposed a mechanism which we denoted as *isothreshold principle*. The mechanism was first described in [Appendix B], and later in a report [Bozinovski 1985a]. Another solution is the *winer-take-all* mechanism [see Grossberg 87] implementing lateral inhibition concept [e.g. Spinelli 1970]. We will describe our solution.

According to the isothreshold principle, the neurons in a pattern classification array share the *common threshold.* The common threshold is a function of all the summary postsynaptic potential of the individual units. The function can be defined in several ways, and for maximum selector mechanism it is defined as

$$\theta = \max_i \{s_i\} \tag{3.5}$$

where θ is the common threshold and s_i is the neuron potential of the i-th neiron.

Once the threshold is defined that way, than the maximum selector is given in a natural way, by the output equation of the neuron in Mcculloch-Pits sense

$$y_i = \begin{cases} 1 \text{ if } s_i \geq \theta \\ 0 \text{ otherwise} \end{cases} \tag{3.6}$$

Figure 3.6. shows the the isothreshold adaptive network as and adaptive array.

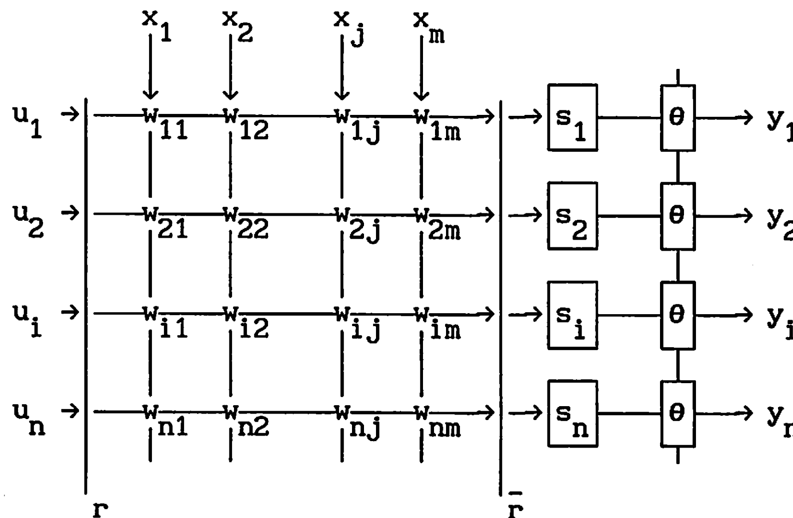


Figure 3.6. Isothreshold Adaptive Network

Figure 3.7. Shows the same network as an assembly of neural units.

Note that the neurons are considered as consisting of two separate parts, the soma with the dendritic tree, and the threshold part, which is connected to other neural units of the *assembly*. The assembly can *grow* by attaching another neuron at the threshold part and at the dendritic tree.

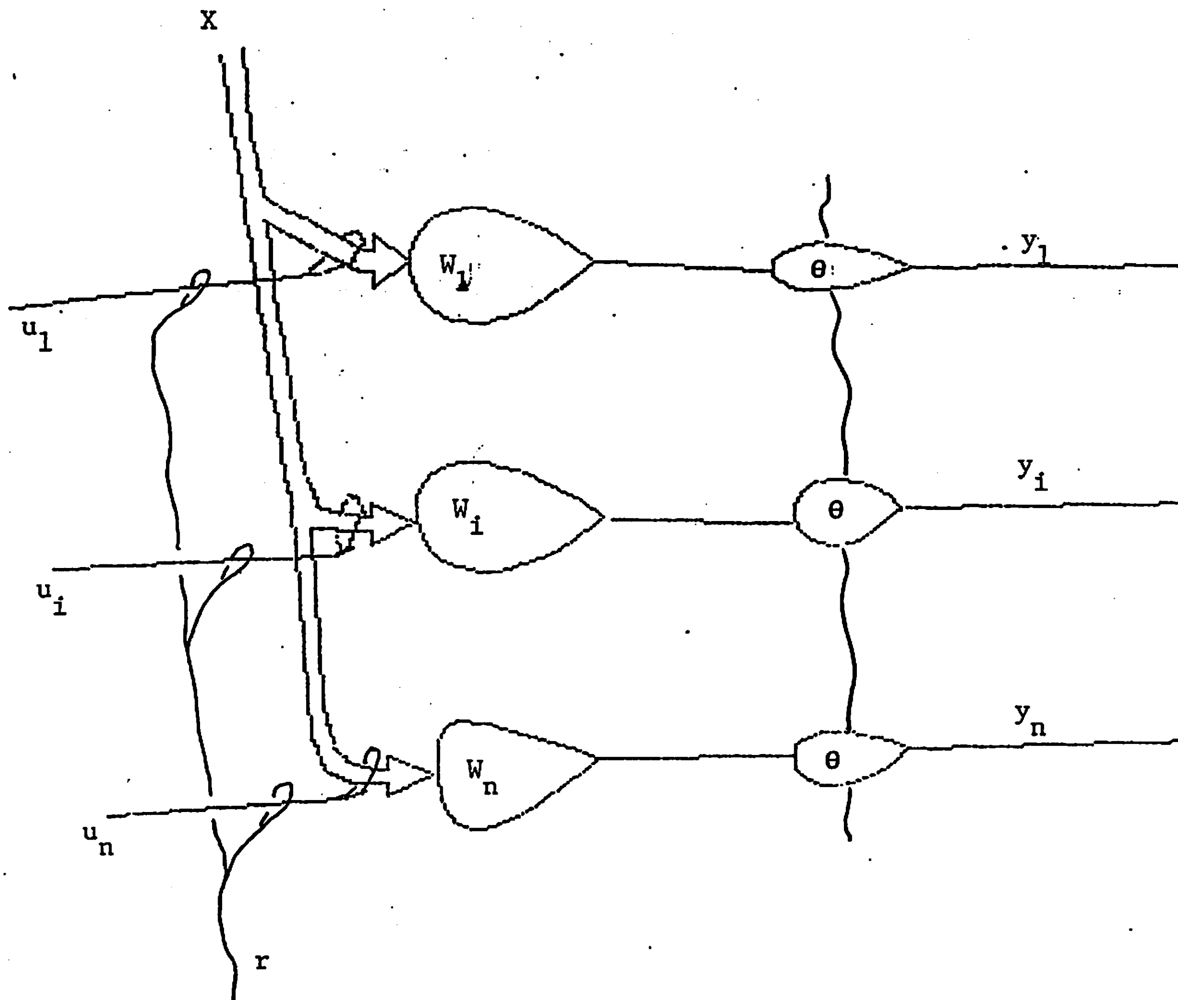


Figure 3.7. The Isothreshold Adaptive Network in neural representation (Drawing software written by Rich Sutton)

We have not seen evidence from the biological research that such a mechanism is confirmed. However it is a simple maximum selector mechanism which we do not see is not feasible to assume for neural computation.

3.4. TOWARD A THEORY OF TEACHING SYSTEMS

So far we discussed the learners in the class A teaching paradigm. In our research in class A learning paradigms the teacher plays important role. Our research in this teaching paradigm was actualy a rearch toward a teaching system theory.

"Toward a teaching systems theory" was the title of the first talk to the Adaptive Networks Group, delivered in September 10, 1980. It was our introductory talk in front of the Group.

In the next several sections we will give a brief overview of some results of the research of the teaching systems in the class A learning paradigm.

The conceptual model of the teacher-learner system that we considered in our research is given on the Figure 3.8. [Bozinovski 1978].

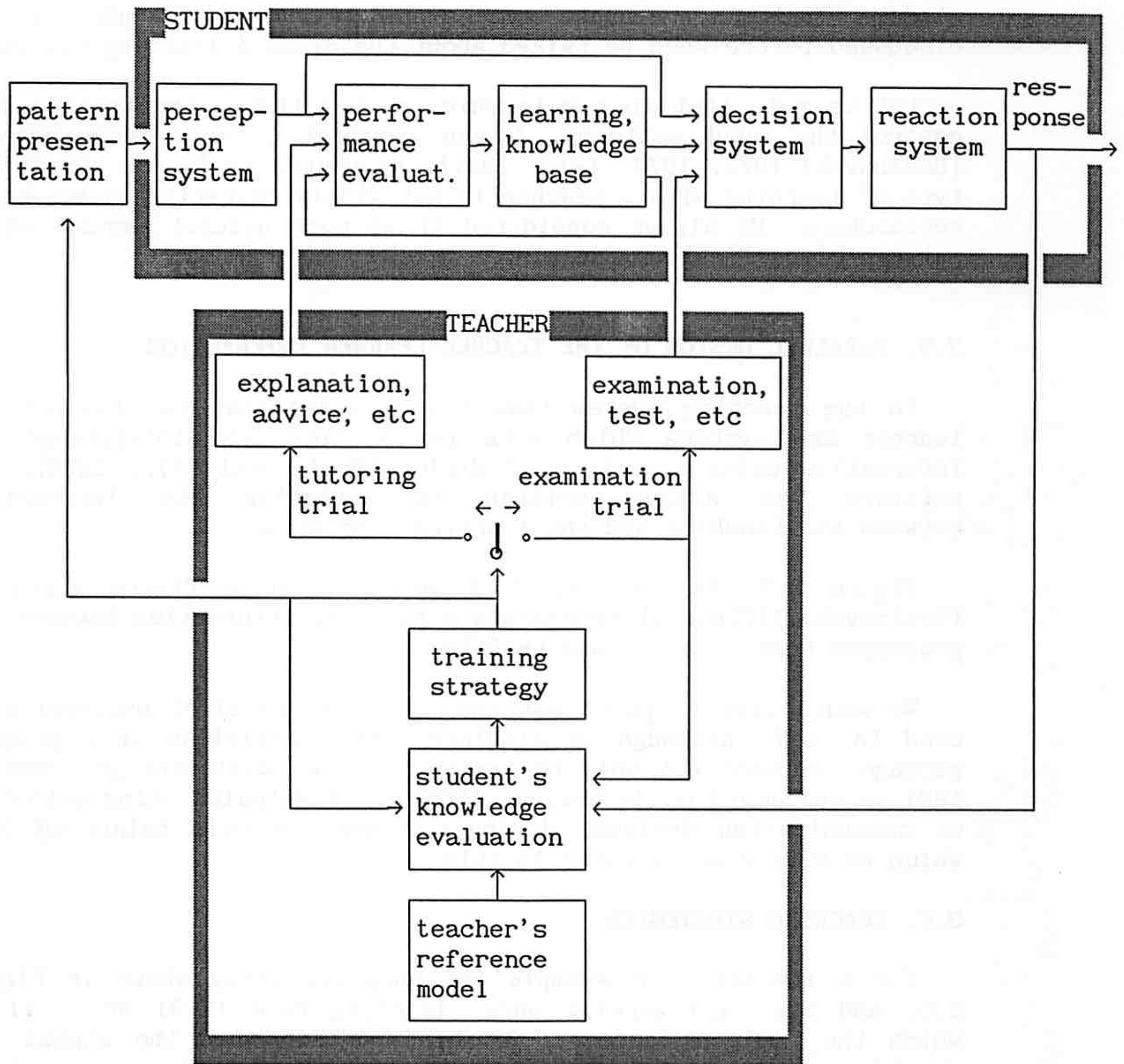


Figure 3.8. The teacher-learner setup in the teaching system theory

Let us note that in the teaching system theory as conceptualized in the Figure 3.8, both the teacher and the learner can be considered interchangeably in the role of agent or in the role of the environment. Actualy, the concept of a *game* applies in the setup shown in Figure 3.8.

Viewing the problem from the teacher side, the teacher *searches through the learners knowledge base* in a systematic way in order to find inconsistencies with its reference model. It introduces the tutoring trial *only when needed* in order to guide the students behavior to converge toward the reference behavior.

Vieweng from the learner's side, the learner will tend to minimize that penalty it receives each time a tutorial trial is applied. Before the tutorial trial is applied in the examination trial it is asked "Which is this pattern?". After the student's answer, if it is wrong, the teacher says "No!, that is not correct. Now I will tell you the correct answer (or correct behavior)". That "No!" is always considered penalty. Adaptive systems tend to avoid

that, and the learner is traing to minimize appearence of that signal. That is reinforcement signal for the learner, as we discussed before when we talked about the class A learning systems.

Let us note that in our teaching system theory the teacher *does* control the input patterns. In our research it was always present [Bozinovski 1972, 1974, 1978, 1981]. As ponted in [Barto 1991] that type of learning with a teacher is not widely investigated by other researchers. We always considered it as most natural assumption in pattern classification training.

3.5. PARALLEL DESIGN OF THE TEACHER-LEARNER INTERACTION

In the teaching system theory we assumed that the teacher and learner are systems which work *in parallel*, and interchange the information using some type of dialog [Bozinovski 1972, 1974]. The software was always written to emphasize the interaction between the teaching and the learning subroutine.

Figure 3.9. is an English translation of a figure given in [Bozinovski 1977b]. It represents a *parallel* interaction between the processes named TEACHER and LEARNER.

We would like to point out the notion of parallel processing we used in 1977. Although we simulated the parallelism in a program package, we were not able to implement it on different programs. In 1981 we succeeded to do that on different terminals, using mailboxes as communication devices, during our work on pole balancing task which we will discuss later in this text.

3.6. TEACHING STRATEGIES

For a learner, for example the adaptive array shown in Figure 3.5. and 3.6. and working under learning rule (3.3) or (3.4) in which the r-signal appears, it is important *when* the signal r=1 should appear. In other words, it is important when to apply a teaching trial.

Also, if the signal r=1 appears and a teaching trial is to be introduced, which pattern will be taught in that teaching trial? That is a crucial problem in the teaching system theory, and in learning with a teacher paradigm. We call it the teaching strategy problem. A *teaching strategy* should exist that will control the appearence of the patterns both in teaching and in the examination trials.

Various strategies can be used, the simplest one being random presentation of the patterns. We considered several strategies, one of which is indeed shown in the Figure 3.9.

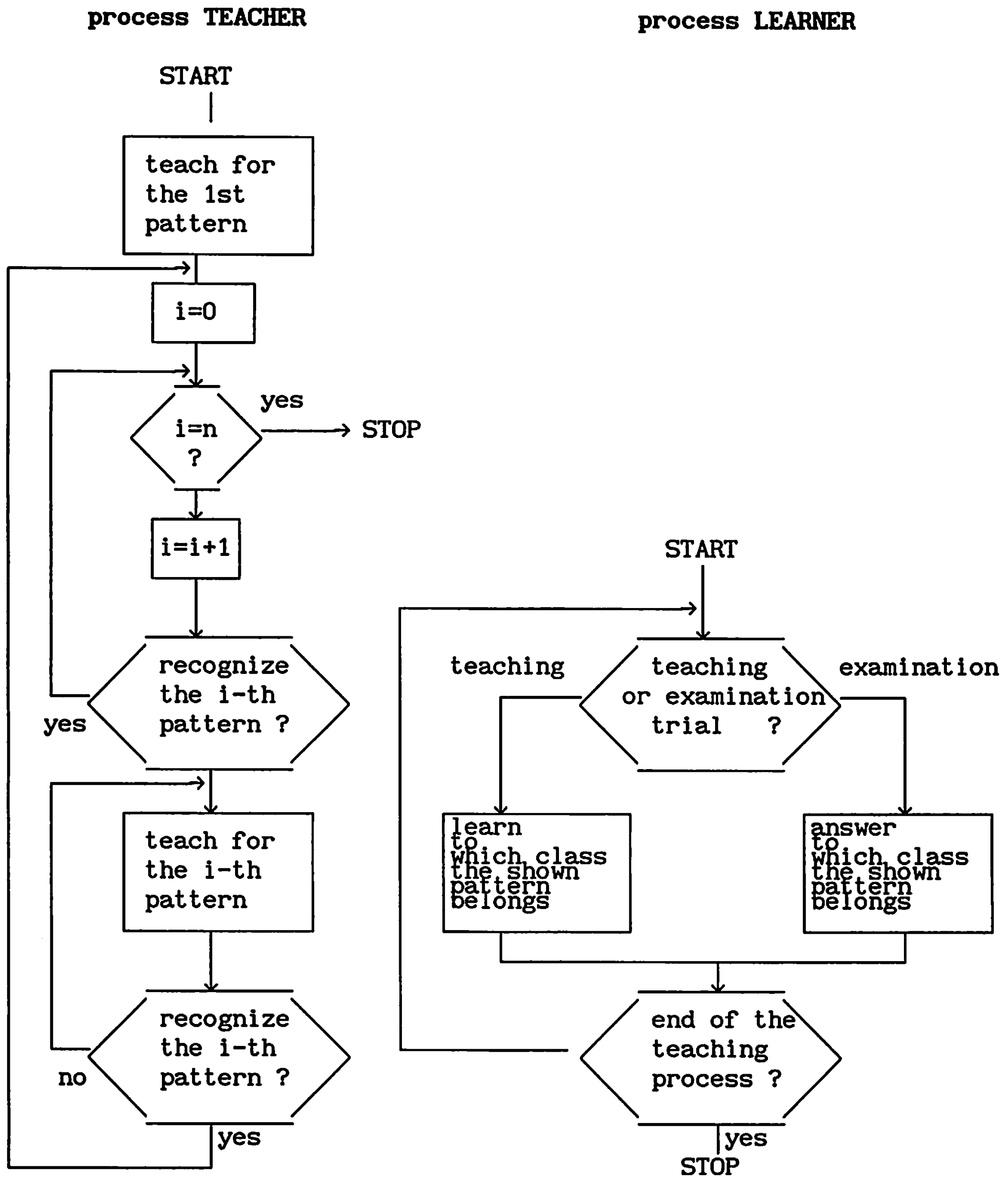


Figure 3.9. Parallel design of the training process implementing the Poem-learning strategy

The teaching strategy in Figure 3.9 we call natural strategy or a poem-learning strategy. It is assumed that a poem consists of several parts (lessons) which are in particular order. The student first learns the first part, and goes to learn to the second part. It will no advance further untill it is clear that it knows the first two parts. Following that recursive procedure, the student will advance in its knowledge. The basic strategy is to learn new knowledge only after it is sure that the previous knowledge is assimilated in the students knowledge base. The whole teaching process has only one global iteration. It has learning trials (learn new lesson) and examination trials (test assimilated knowledge).

We experimented with this strategy in 1978 and presented the results in the mentioned talk to the ANW group. A set of 26 English capital letters as the appear on a computer terminal (Figure 3.10.) were presented to the adaptive array shown on the Figure 3.4. and the system was trained to recognize them.

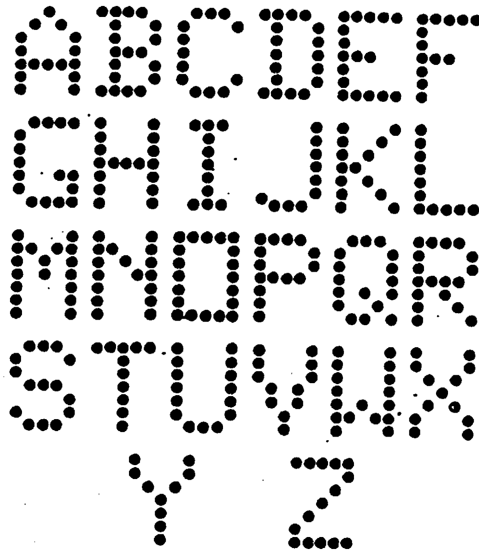


Figure 3.10. An experiment with class A system: The training patterns

The result of the training, the learning curve using the poem-learning strategy is given on Figure 3.11.

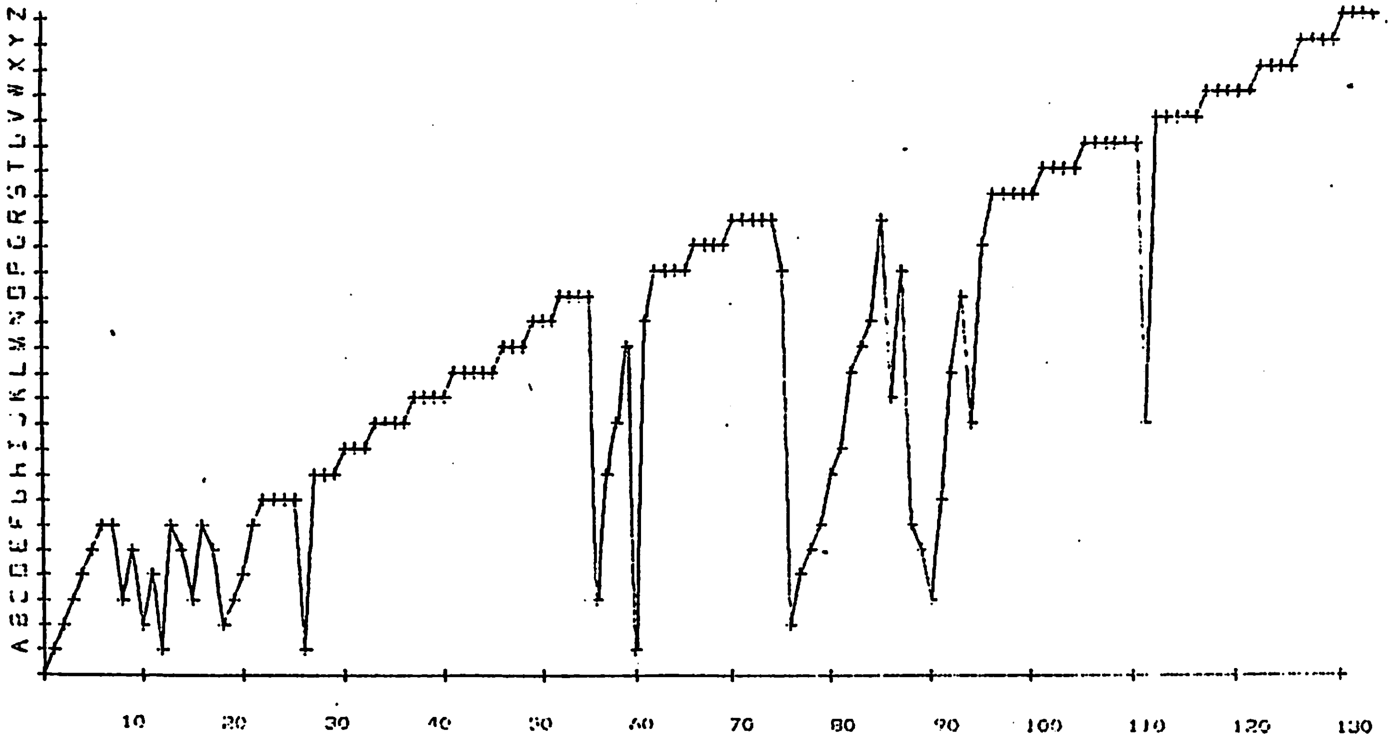


Figure 3.11. Result of the training experiment: The learning curve

Viewed as an anytime algorithm (Zilberstein 1995), this strategy, if tested in examination phase, tells immediately how far the process has gone with training: if we see that the process moves toward learning a new lesson, then we know that all the lessons taught before are have been learned. If lessons are patterns, we know that all the patterns thaught before will be recognized if we ask the algorithm to stop with training period and to go to exploatation (application) phase. This is not the case with usual perceptron training strategies (Minsky & Papert 1969) with random presentation of patterns, in which we know that the algorithm will converge but if we stop training in a moment, we do not know what is learned and what is still to be learned.

Teaching strategies are interesting issue in pattern classification teaching and in teaching theory in general. We have investigated several if them and reported elsewhere (Bozinovski 1981b, 1981c).

3.7. TEACHING GRAMMARS AND LANGUAGES

In our research in teaching grammars we introduced a gramatical represenatation of the teaching process [Bozinovski 1981b, 1985a]. Here we will give a short description of that research.

Let all the lessons (e.g. patterns) to be learned are ordered according to some *didactic order*. Let X_i denotes appearence of the i-th pattern a teaching trial, and x_i denotes appearence of that pattern in an examination trial. Than the training process can be viewed as a string of uppercase and lowercase letters. That string we denote as *curriculum*. The teaching process can be viewed as shown in Figure 3.12.

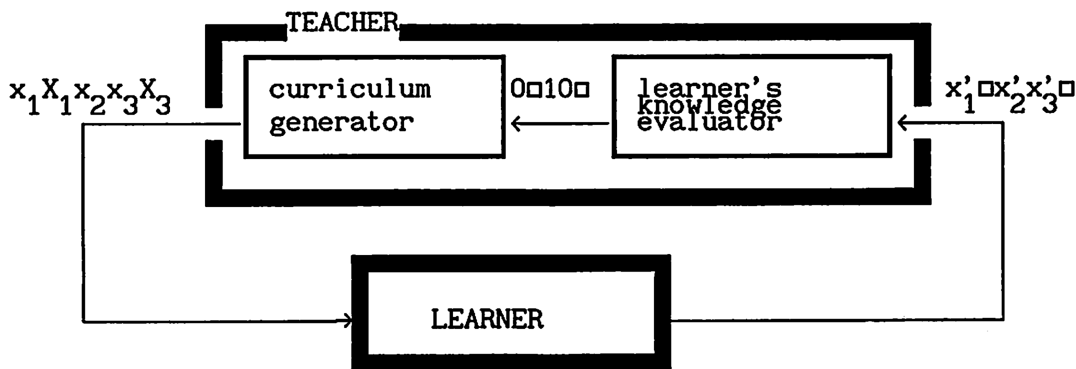


Figure 3.12. The teaching process as string generation process

The Figure 3.12. shows that the teacher is recognized as being composed of two main parts: 1) the *evaluator* of the student's knowledge and 2) *generator* of the curriculum. If the teacher performs an examination for the first pattern, x_1, it will receive an answer x'_1 for that pattern. The evaluator will evaluate with 1 (good) or 0 (bad response). In a case when 0 is given as evaluation (see Figure 3.12) the teacher will apply a *teaching trial* X_1 to

correct the learner's knowledge for the noticed error. As a result of this behavior, the teacher will generate a string consisting of uppercase and lowercase letters. The set of all strings generated by that process forms a *language*. That language can be described by a grammar.

With this representation we can describe the languages generated by different strategies using grammars. For example the poem-learning strategy shown on Figure 3.9. can generate the languages covered by the following grammar

$$S \longrightarrow x_i M$$
$$x_i M \longrightarrow x_i x_{i+1} M \mid x_i X_i x_i M \mid x_i X_i x_i S$$
$$x_n M \longrightarrow x_n X_n x_n M \mid x_n S \mid x_n.$$

The languages generated by those grammars we denote as curriculum (or teaching) languages. It is interesting to observe the generation of a curriculum language as a Markov source. We considered that and computed entropies of such languages and compared them to some other natural languages [Bozinovski and Cundeva 1985]. The computing procedure is discussed later when we talk abot the CAA network.

3.8. THE TUTORIAL ALGORITHM

Here we will decribe the algorithm used in our research with the class A systems.

Consider a class A system with learning rule

$$\Delta w = cx(t-1)u(t)r(t) \qquad (3.7)$$

where

x is a situation (pattern) vector
u is the advised action vector
r is the penalty, applied each time when the learner fail to take appropriate action in the given situation
c is some positive constant

Then naturaly the problem of teaching for pattern classification can be expressed as an optimization problem. From a teacher point of view the problem can be stated as: find a training sequence with minimal length. From a learner point of view the problem can be stated as: adjust the behavior (answer to questions) such that receive minimal amount of penalty (punishment).

The learning rule shown above is essential for definition of the training algorithm which we call tutorial algorithm, shown in Figure 3.13.

```
TUTORIAL ALGORITHM
repeat
  examination:  show next x; receive action y(x)
  evaluation:   if y(x)=u(x) then go to examination
  teaching:     r=1: adjust Δw(u(x))= cx; go to examination
until  y(x)=u(x) true for all x
```

Figure 3.13. The tutorial algorithm

In other words, whenever the learner fails to recognize **x**, the learning rule

$$w = w + cx \text{ , (x belongs to the class represented by w)} \qquad (3.8)$$

is applied.

Two points we would like to stress in connection with this algorithm. The first is the sequence in the algorithm routine. It is a closed loop three-step routine which is repeated until endcondition. Not always the third step is applied; only when needed. It is a *response sensitive algorithm*. However, it doesnt say which pattern is next to show. That is a responsibility of the training strategy.

The second point is the learning rule. It is *not a difference learning rule*, like for example the so-called perceptron learning rule

$$w(t+1) = w(t) + c\,(u(t)-y(t))x(t) \qquad (3.9)$$

where we can have increments and decrements. This is a *selective increment* learning rule. Basing on this rule we developed our integer programming representation of the pattern recognition task [Bozinovski 1981, 1985], which we are going to describe next.

3.7. CLASS A TEACHING AS INTEGER PROGRAMMIMG

In the pattern classification theory, the geometric interpretations are usualy given in the *feature space* and/or in the *weight space* [Duda and Hart 64]. In our work [Bozinovski 1974, 78, 81, 85] we introduced the teaching space (or *curriculum space*) approach as a convenient representation technique.

Instead of working with N real weight vectors {**w**}, it can be shown that the problem can be represented in terms of *one* N-dimensional vector **p** which components are number of appearences of the situations in the teaching trials. Its length is actualy number of execution of the teaching trial step in the above algorithm. The objective is to minimize the penalty (r=1) received each time the teaching trial is applied.

Figure 3.14 shows the geometrical representation of this process as an *integer programming* process. Three patterns, E,T, and F are represented in a 7x5 dot matrix and the learner is taught to recognize them. The figure shows a guided search toward a point in a convex polyhedral cone which represents a solution. Let us note that solutions of the training process are always in such a cone.

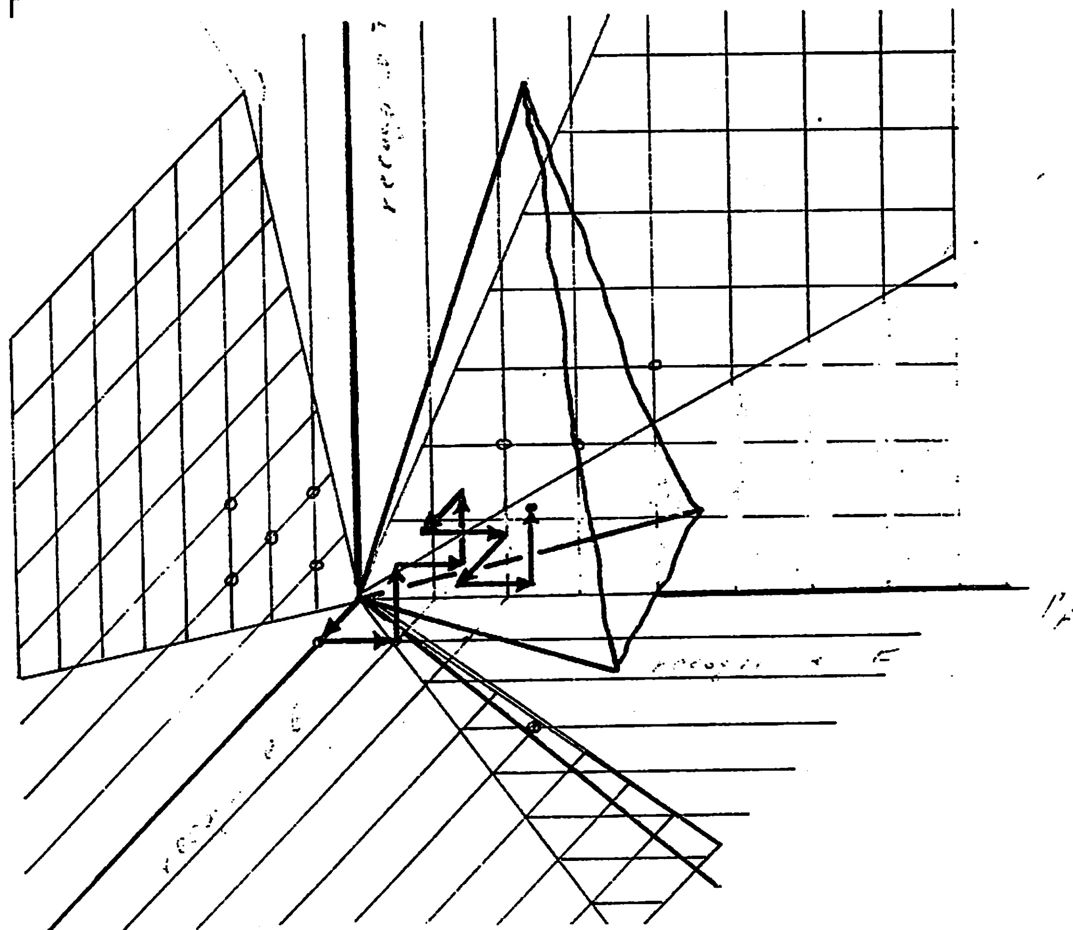


Figure 3.14. Pattern classification as integer programming problem

3.8. CLASS A TEACHING AS DYNAMIC PROGRAMMIMG

Instead of teaching as an integer programming problem, using our curriculum space approach the pattern classification teaching can be viewed as a kind of dynamic programming problem. Figure 3.15 gives the basic idea.

Figure 3.15. shows a discrete space in which each point is a state of the space. The space is divided into two subspaces, a subspace of desirable states and a subspace of undesirable states. In Figure 3.15. the subspace of desirable states is represented by a two-dimensional cone, i.e. as an open end angle. Each (discrete) point in that region is a desirable state. In the space there are shown four agents, A, B, C, and D. They are starting from different states, depending on their initial knowledge about the pattern classification task. The values of the states can be defined in various vays, depending of how we want to define the optimization function. For example we can define the state values inside the desirability region to be 0, and outside the desirability region to be -1; in that case we can define optimization task to be a minimum acquired penalty in the space. In such a task each agent will tend to avoid the undesirable states in order to obtain minimum penalty. Or, we can define the values inside the desirability region to be +1 and outside 0. In that case we can define a quest for maximum.

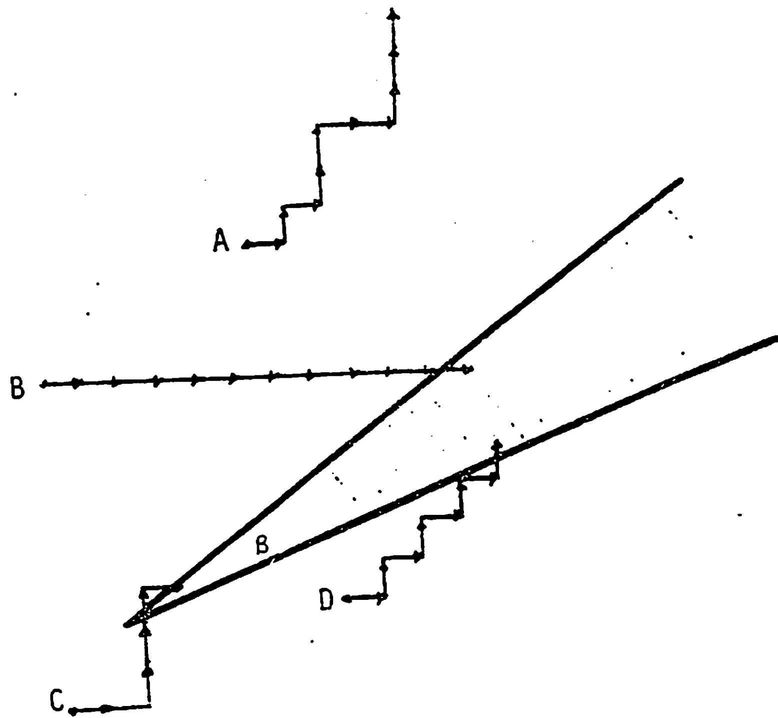


Figure 3.15. Search through a teaching space for solution of recognition of two patterns (From Bozinovski 1981c)

The problem can be represented in a "grid world" which is more usual representation in dynamic programming. Figure 3.16. gives such a representation.

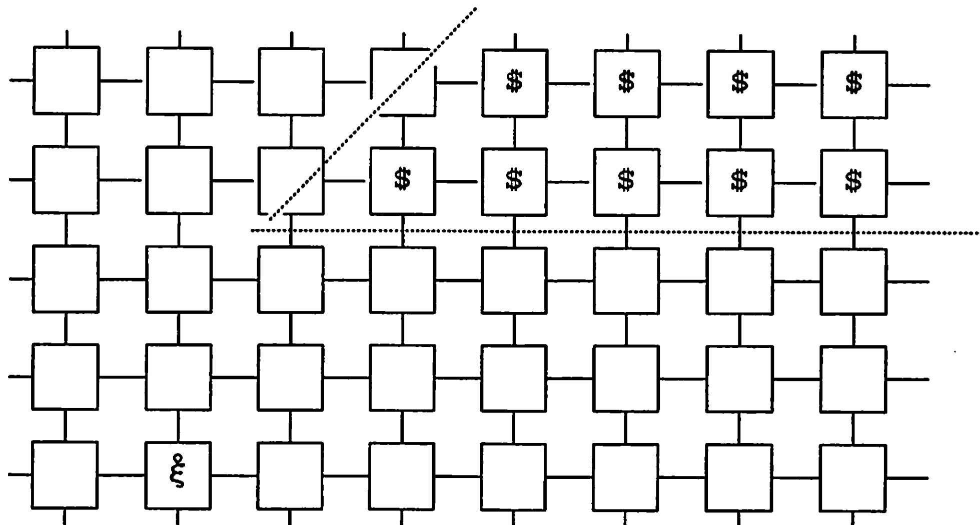


Figure 3.16. Teaching for pattern classification as a dynamic programming problem

As we can see from the Figure 3.16. we can have an agent being in arbitrary initial state in a discrete, *valued state space*, which is basic representation for a discrete dynamic programming task. In this case we have only two values assigned to states, but other

assignment can easily be visualized. However, in the teaching space approach, we usualy have situations as represented in Figure 3.16. The admissible steps under the tutorial algorithm shown in Figure 3.13 are "east" and "north". The problem is to learn *a policy* which will reach a goal state, possibly in minimum steps.

We would like to emphasize that during our work on integer programming representation [Bozinovski 81, 85] we were not aware of research in dynamic programming. However, as we can also see later, some of the crucial issues as state value, and state evaluation as interpretation notions naturaly emerged during our research. Further in the text we will return to this issue again.

As short summary of our work in class A systems we can now say that it was in a sense not in the main line of the conventional research in pattern classification learning. The conventional research was mainly interested in patten clasification *learning* where [Barto 1991] the teaching sequence is not controlled. In contrast, we were dealing with with pattern classification *teaching*, where a teacher has control over the teaching sequence, and deals with optimization issues such as to produce the shortest efficient curricullum. To make the teaching process more observable, we developed a curriculum space approach and the algorithm which transforms the problem of teaching to a problem of integer programming.

This is the frst area where we explored the possibility of our generic NG agents. In the next chapter we will deal with teacher which gives only evaluation and not advice as what to do.

CHAPTER 4

CLASS B LEARNING SYSTEMS:

ADVICE FREE, EXTERNAL REINFORCEMENT LEARNING SYSTEMS

4.1. INTRODUCTION

Important class of learning systems are the learning systems which do not take advice how to act in a given situation, but do take information related to their performance. Example of such a learning system is a student who takes exam consisting of several questions, answers the questions, and receives a grade for his overall performance in the exam, but no corrections to each answer. So, the reaction of the environment is a scalar value (grade) as *overall performance* on a typicaly multicomponent action (answers on several different questions).

The received scalar reaction is used as *reinforcer* for future actions. To put it simple, if the grade was good, keep doing the same way as before; if the grade was not satisfactory, choose a correction of the behavior. However, the environment does not say in which direction the behavior should be changed. That must be done by the learning system itself. In order to change direction, it must have ability to estimate the gradient of its behavior. So it must *have retained the previous action*, y(t-1), or the sequence of the previous actions, p(t-k), k=1,..,K. That is actualy stated in our structural theory: a feedback must exist from the previous action (Figure 4.1).

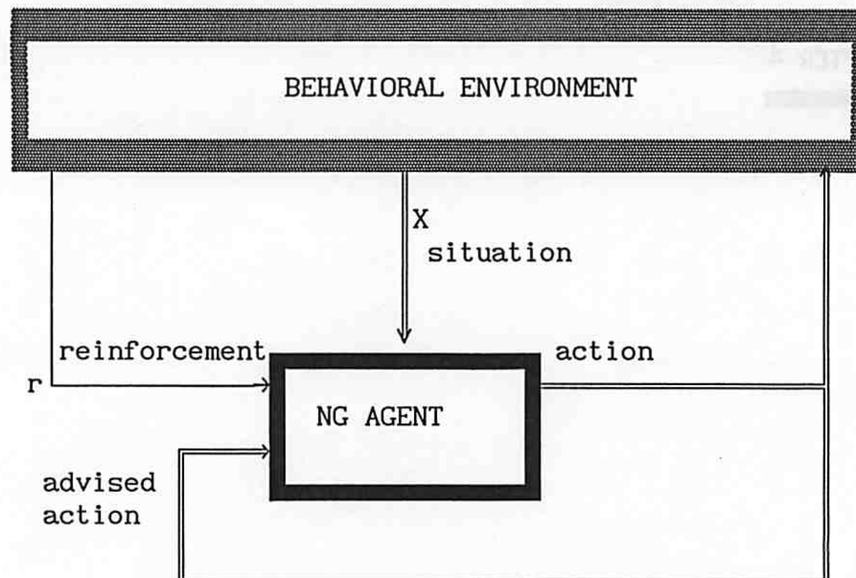


Figure 4.1. Our NG agent as a reinforcement learning system

Figure 4.1 shows a clasical reinforcement learning system. The system *tries* an action in a given situation, and receives a new situation and also a distinguishable signal recognized as reinforcing (payoff, penalty, reward) signal. The behavior of the system is devoted to the optimization of the signal r.

The work on reinforcement learning has long history [Minsky 1954; Mendel, McLaren 1970; Widrow, Gupta, Maitra, 1973, among others] but the most prominent work on the subject in recent times was carried out within the Adaptive Networks Group. We believe that the most significant work is the design of the Associative Search Network (Barto, Sutton, and Brouwer 1981, Barto and Sutton 1981).

In this chapter we will just briefly describe the main concepts of reinforcement learning as class B learning systems. More about that could be found in other reports from the Adaptive Networks Group [Appendix E].

4.2. ASSOCIATIVE SEARCH NETWORK

Early works on advice free reinforcement learning neural network was done by Minsky [Minsky 1954]. However, the Associative Search Network (ASN) (Barto et al. 1981a,b) is the most influential reinforcement learning neural network.

Two basic versions of the ASN were proposed, a 1) neural assembly version [Barto, Sutton, Brouwer 1981a] and a 2) neural array version [Barto and Sutton 1981b]. The neural assembly version is important because it introduces a *predictor* component, which later evolved to a *heuristic critic element* of the Actor-Critic architecture [Barto, Sutton, Anderson 1983]. The neural array version is shown on Figure 4.2.

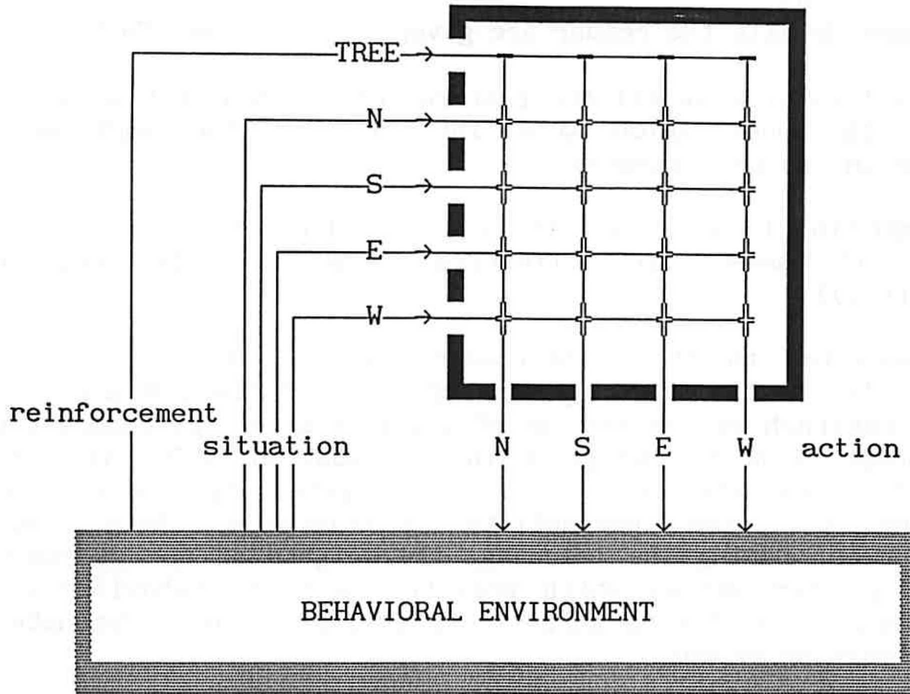


Figure 4.2. The ASN adaptive array architecture

The ASN as shown in Figure 4.2. is designed to solve a navigation task. It senses some landmarks denoted as N(orth), S(outh), E(ast) and W(est). It also receives an reinforcement input denoted as "TREE". It can take four basic actions in the directions of N, S, E, and W, but combinations are also possible, like SW, since two actons can be taken simultaneously. Which action will be taken depends of the strength of associations between the situations and the actions. The modifiable weights, shown with ╫, represent the association strength. The ASN task is to find a goal place, (TREE) learning how to behave in vicinity of a landmark.

The basic learning rule which builds the association strength is given by

$$w_{ij}(t+1) = w_{ij}(t) + c[z(t)-z(t-1)]y(t-1)x_i(t-1) \qquad (4.1)$$

where w_{ij} is the association strength (synaptic weight), z is the reinforcement, x_i, (i=1,2,3,4) are the situations, and y is the action.

The basic output rule is given by the following pair of relations

$$s_j(t) = w_{oj}(t) + \sum_{i=1}^{4} w_{ij}(t)x_i(t) \quad \text{and} \qquad (4.2)$$

$$y_j(t) = \begin{cases} 1 & s_j(t)+NOISE_j(t) > 0 \\ 0 & \text{otherwise} \end{cases} \qquad (4.3)$$

The term w_{oj} has its own learning rule which we do not consider here. More details the reader are given in [Barto and Sutton 1981].

We will not review all the features of the ASN, but we will only mention the ones which were influent for the work we will describe in the next chapter.

As equation (4.1) shows, it is a second order, class B learning system. It uses the difference term for the reinforcer, $(z(t)-z(t-1))$.

In addition to the reinforcement learning rule given in the relation (4.1), the most important feature of the ASN architecture for our reserach is the concept of *neural search*, introduced through the concept of NOISE, as given in the equation (4.3). The idea is that before learning the actions of the system are randomly chosen, according to some probability distribution, here *Gaussian* distribution. During the learning, the weights of the strength are gaining greater values which prevails, and the behaviour of the system tends toward some purposively one, which is a demonstration of the learning process.

Besides the advice free architecture, the main difference between this network and the network described in the previous chapter is that this network has no maximum selector as the output decider. It is a class of so call *linear associative memories*, studied by Kohonen (1974). If such a network is used for pattern classification, the pattern to be distinguished should be linearly independent. That means that if we have example where three vectors are given such that **x1** = [0 0 1], **x2** = [1 1 0], and **x3** = [1 1 1], then they cannot be distinguished since **x1+x2=x3**. As contrast, the networks we have used before, with maximum selector and implementing the tutorial algorithm, can solve this problem. The maximum selector introduces nelinearity in the network.

Also, very important for the appearence of the ASN architecture was indeed the exciting experiment of the spatial navigation we briefly desccribed above. Before that, the neural networks were mostly engaged in patern classification task.

It is our belief that the landmark learning task, and the ASN solving it, was a major event in the neural network research since the appearence of the perceptron.

4.3. THE ACTOR-CRITIC ARCHITECTURE

The Actor-Critic (AC) architecture [Barto, Sutton, Anderson 1983] is a reinforcement learning neural network architecture, which can be represented by two principal neural units, as shown on Figure 4.3.

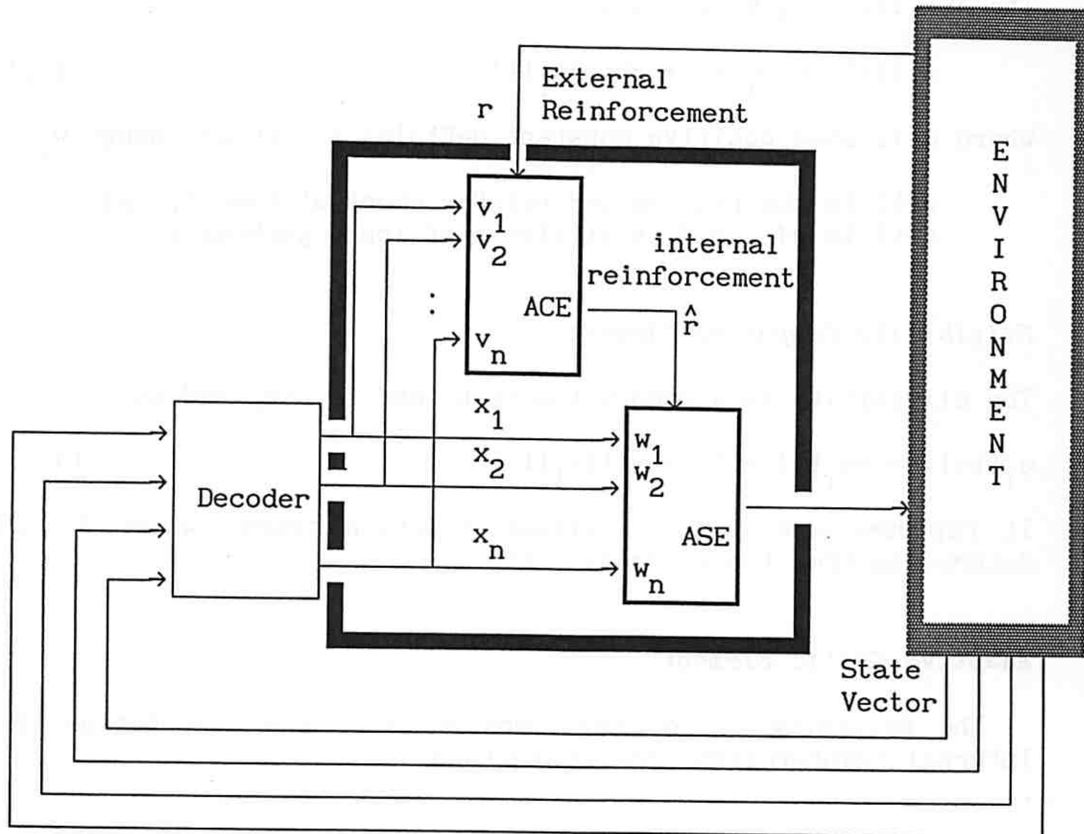


Figure 4.3. The design of the Actor-Critic architecture

The first unit is the Associative Search Element (ASE) which is responsible for actions, and the second unit is the Adaptive Critic Element (ACE) which is responsible for supplying internal reinforcement to ASE. ASE is constructed such that it must receive a reinforcement signal in each time step, and ACE must supply it. ACE is actualy a predictor for reinforcement, and evolves from the predictor element of a previously developed ASN network [Barto at al. 1981].

The AC architecture is a powerfull and rather complex neural architecture, implementing higher order learning rules. It has two principal neural elements, The ASE and ACE elements, but one can recognize two additional elements of computation within the system, for computing memory traces. Here we give a brief description of all the four elements.

Associative Search Element

The output equation of the ASE element is

$$y(t) = f\left[\sum_{i=1}^{n} w_i(t)x_i(t) + noise(t)\right] \qquad (4.4)$$

where

$$f(x) = \begin{cases} +1 & \text{if } x \geq 0 \quad \text{(control action right)} \\ -1 & \text{if } x < 0 \quad \text{(control action left).} \end{cases} \qquad (4.5)$$

The ASE learning equation is

$$w_i(t+1) = w_i(t) + \alpha r(t) e_i(t) \tag{4.6}$$

where α is some positive constant defining a rate of change w_i,

$r(t)$ is the real valued reinforcement at time t, and
$e(t)$ is *eligibility* at time t of input pathway i.

Eligibility Computing Element

The eligibility is a memory function, and is computed as

$$e_i(t+1) = \delta e_i(t) + (1-\delta) y(t) x_i(t) \tag{4.7}$$

It represents a previous situation-action trace, where δ, $0 \le \delta < 1$, determines the trace decay rate.

Adaptive Critic Element

The following set of equations describe the computation of the internal reinforcement computed by ACE

$$\hat{r}(t) = r(t) + \gamma p(t) - p(t-1) \tag{4.8}$$

where

$$p(t) = \sum_{i=1}^{n} v_i(t) x_i(t) \tag{4.9}$$

where

$$v_i(t+1) = v_i(t) + \beta [r(t) + \gamma p(t) - p(t-1)] \bar{x}_i(t) \quad . \tag{4.10}$$

In the equations (6.8)-(6.10) we can see that the internal reinforcement is computed from the external reinforcement $r(t)$ and the difference between the the predicted reinforcement $p(t)$ and the prediction in the previous step. The predicted reinforcement is "discounted" [Witten 1977] by a prediction factor γ, here used $\gamma=0.95$. The predicted reinforcement is computed by the ACE neural element as a weighted sum of the input stimuli, where weights are updated according to the learning rule (6.10). The learning rule contains the positive constant β, the internal reinforcement, and the trace of the input variable. Here $\bar{x}$ represents the memory trace of the input signals, computed by a separate memory equation.

Input Trace Computing Element

The trace of the input signals is computed as

$$x_i(t+1) = \lambda \bar{x}_i(t) + (1-\lambda) x_i(t) \tag{4.11}$$

where λ is a trace decay factor and $0 \le \lambda < 1$.

The Actor-Critic architecture is even more popular than the Associative Search architecture. In parallel with the Crossbar Adaptive Array architecture, which will be discussed in the next chapter, it solved the problem of delayed reinforcement learning. Also, it demonstrated its abilities on the problem of learning for pole balancing. In short, both the reinforcement learning architectures, ASN and AC, had a great impact on neural networks research.

In the next chapter we will consider an other method of solution of the problems considered by the AC architecture, the CAA method, which is a self-reinforcement learning method.

In connection to our work, we should say that the Associative Search Network was of great influence to our understanding and view toward new horizons in neural network research. It was a line of research which greatly contributed to the design of a self-learning system which we wished to know how to design. The concept of *neural search* and taking advantage of a random move which turns out to be a good one, was something which we lacked in our work in class A systems. As contrast, the AC architecture had no impact on our research, since it was developed in parallel to our CAA architecture and they were developed as significantly different architectures.

CHAPTER 5

SELF-REINFORCEMENT LEARNING AGENTS

This chapter deals with agents which exhibits *self-learning*. The notion of self-learning has been around since the appearence of the problem of learning within the framework of cybernetics. There exist various definitions on that notion. The basic assumption is that it is a paradigm of learning without a teacher. Self-organization is a term also used. It is difficult to make a taxonomy of various models od understanding around this notion, and we will not make such an atempt.

In the previous chapters we discussed the paradigms of learning with environments which give advice, and environments which give reinforcement (reward or punishment) only, but no advice. In both paradigms, there is a notion of goal of learning, or a task that has to be learned. In the advice giving paradigm, the goal is to minimize the discrepancy between some default behavior and the advised (required) one. Using control theory terminology, we call it *setpoint learning*. In the second paradigm, the goal is to maximize some reward function, or to minimize some penalty function. This is a typical optimization task from control theory, and we can call it *optimization learning*.

In this chapter we are concerned with the problem how to set a task, and how to define a goal of learning in a self-learning paradigm, with no external reinforcement present. We will describe a method which to the best of our knowledge is the only method of self-reinforcement learning.

But, before we go in discussion about our self-learning agents, let us describe our early work with self-reinforcing neurons, which gave us a motivation for searching for a solution to the self-learning problem.

Let us consider the following problem [Bozinovski 1972a]. Consider a pattern classification network on Figure 5.1. It has a task to organize its neurons to recognize two letters, the letter L and the letter I. It has two sets of neurons, which sum their outputs in ΣI and ΣL, after which a discrimination unit D decides which pattern is recognized.

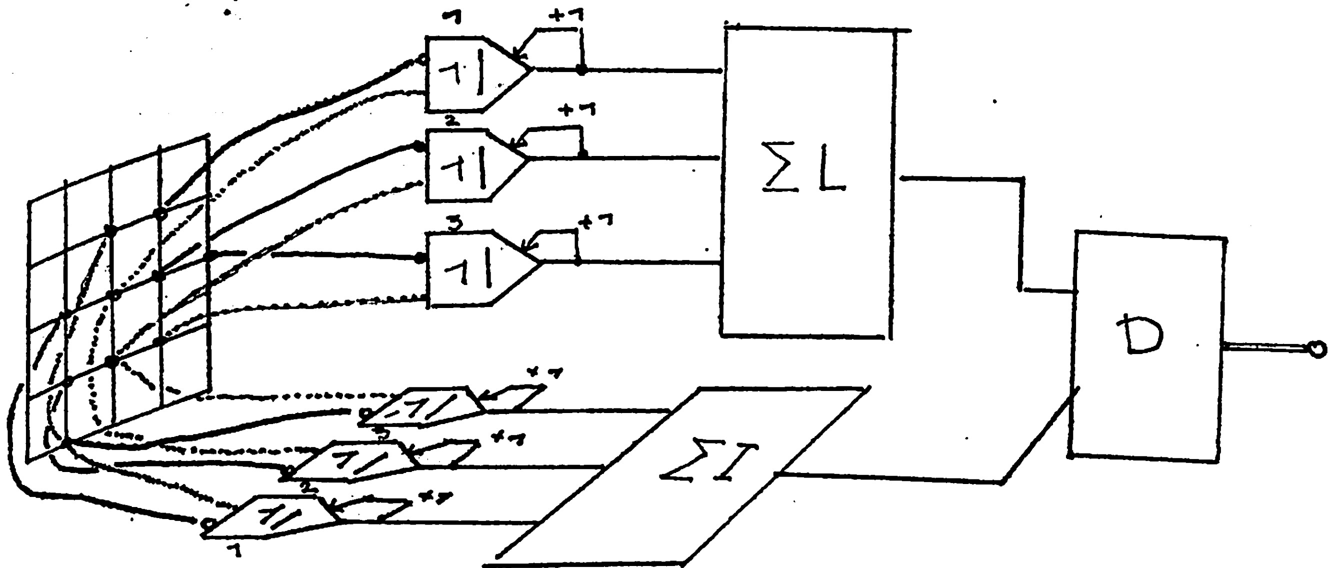


Figure 5.1.Pattern classification network with self-reinforcing neurons (From Bozinovski 1972a)

The neural elements used in the network are actualy the Rosenblatt A-elements, but with the following learning rule: *Each time the neuron produces a (positive) signal, its state is increased by one.* Symbolicaly,

$$w(t+1) = \begin{cases} w(t) + 1 & \text{if } y(t) = w(t) \geq 0 \\ w(t) & \text{if } y(t) = 0 \end{cases} \qquad (5.1)$$

This is self-reinforcing neural element. Since the network does not receive additional signal besides the shown pattern, it is a self-reinforcing network. Two questions can be rased? Can this system be adjusted to solve the stated problem of pattern classification? Can this system learn by itself? The answer on the first question is yes. This is a simple system, and a human operator can adjust its weights appropriately. The answer on the second question is no, since the learning task is not well defined. There is no indication how the inputs are affected by the produced output: there is no indication how the feedback through the environment is defined. It can be defined by means of a human operator, but that is not what is interesting in self-reinforcement learning paradigm.

It took us almost 10 years until a satisfactory concept of self-reinforcement learning was found. Now we will describe the framework and our solution of the self-learning problem.

5.1. CONCEPTUAL FRAMEWORK

The basic idea is that a learning agent, when it apears in a behavioral environment, already has some knowledge in its memory, *from the genetic environment*. This is hereditary, innate knowedge. This knowledge is providing the basic *instincts*, basic *preferencies* (desires) of what is "good" and what is "bad" in the behavioral environment in which the agent is going to perform. Having basic desires, the agent can generate wishes, hopes, expectancies, predictions, and feelings which underline its behavior. Having that, and using the principle of *secondary reinforcement*, it can state goals and *generate subgoals*. Having that, it can *generate plans* how to achieve the goals. In acting toward goals, it can *learn* how to make it more efficient, by building a model of the environment and building policies for its behavior. That is our understanding of how an intelligence evolves in an environment.

5.2. SELF-REINFORCEMENT LEARNING AND THE NG AGENTS

Figure 5.2 shows the design of a self-learning system according to the structural theory presented in the second chapter.

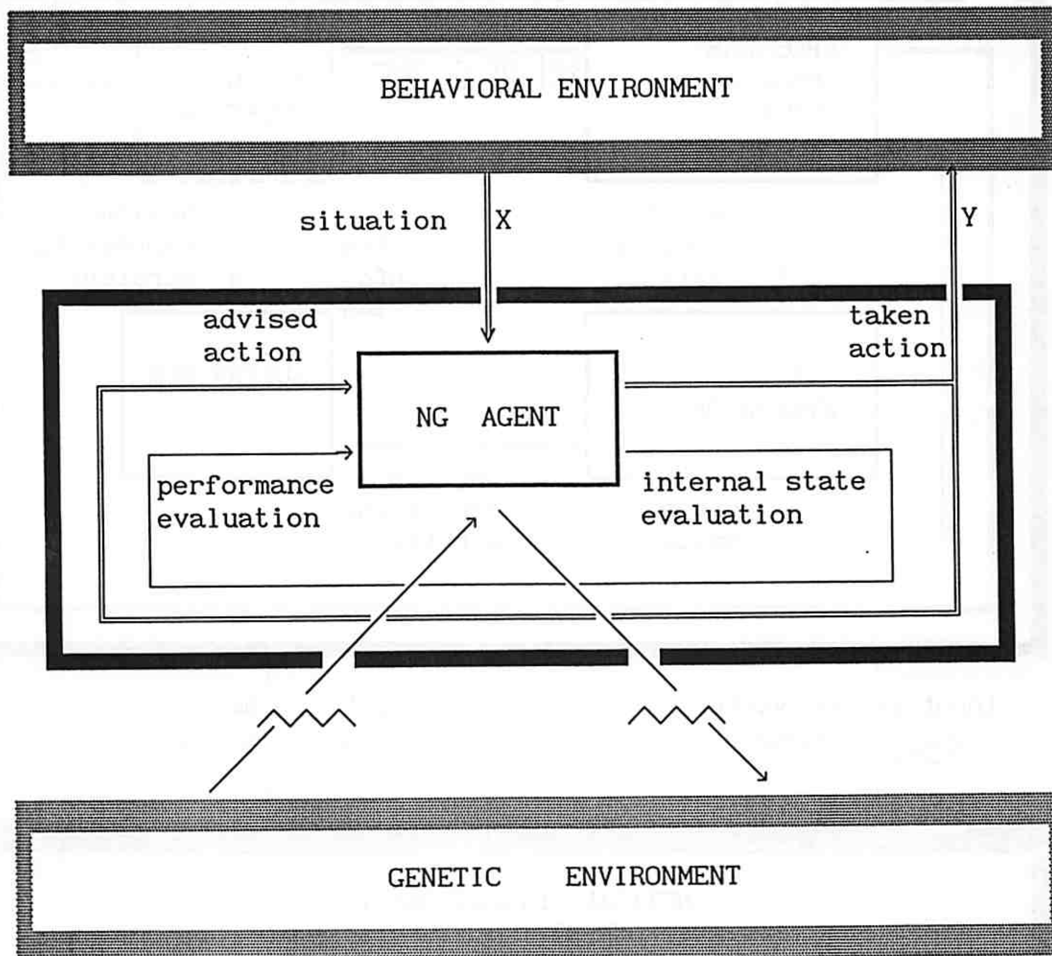


Figure 5.2 A class C, self-reinforcement learning system

In this system, the primary drives will be defined by means of the input genome vectors. The behavior will be developed using secondary reinforcement principle (e.g. Saltzman 1949), i.e.

goal-subgoal principle known in AI (e.g. Gelernter and Rochester, 1958). The double-feedback architecture will be used for that purpose.

In the sequel we will describe a construction of an agent which is a result of this conceptual framework.

5.3. THE CROSSBAR ADAPTIVE ARRAY CLASS C ARCHITECTURE

Now we will describe the construction of the agent, which we simply call Crossbar Adaptive Array (CAA). Figure 5.3 shows the CAA architecture.

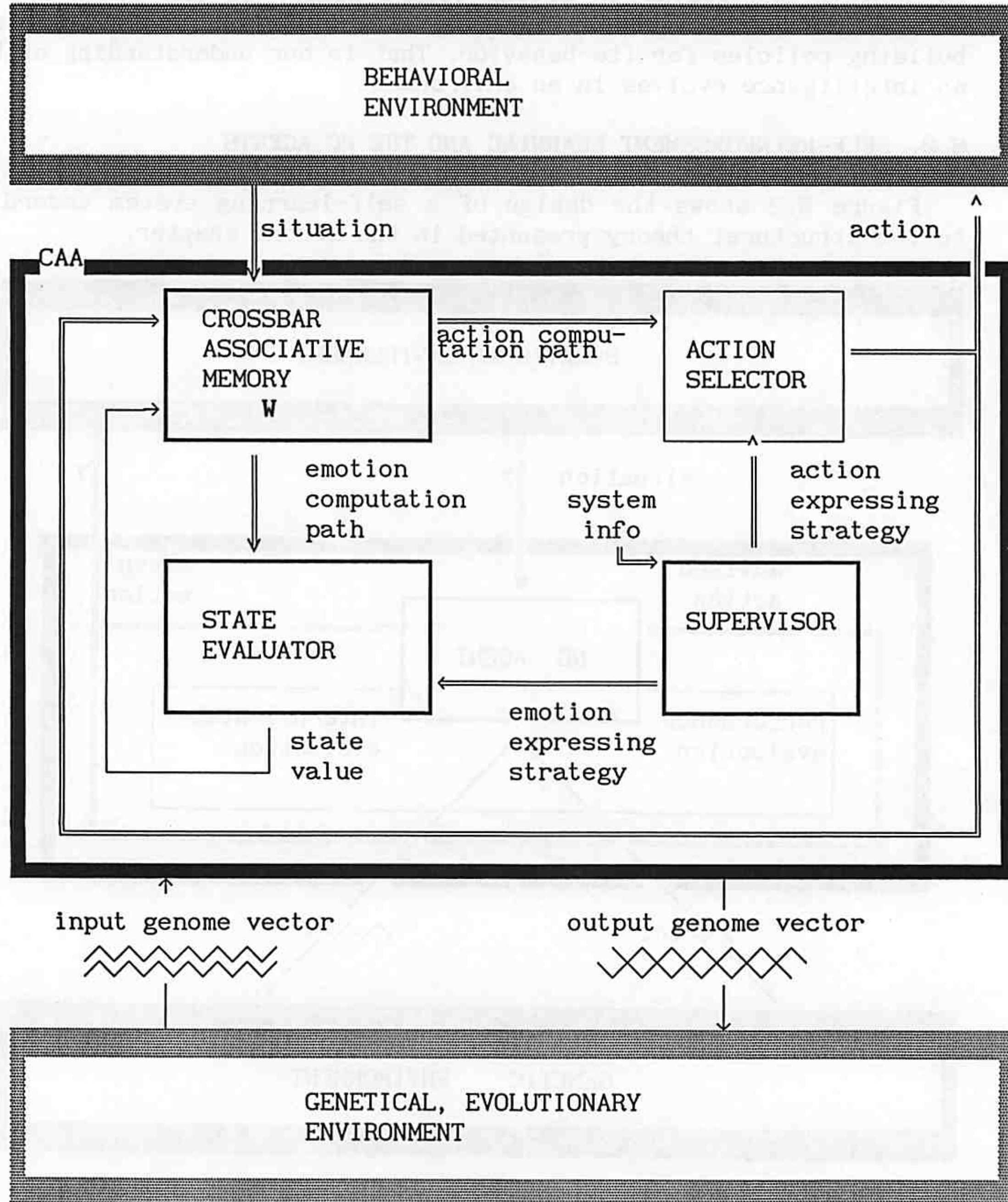


Figure 5.3. The CAA concept

The basic conceptual blocks of the CAA are:

1) BEHAVIORAL ENVIRONMENT: It is the systems in which the CAA system expresses itself. It receives the agent's action and computes the next situation *using its internal state*. The state of the environment is not directly observable by the agent. However, it is indirectly observable by means of the situation X. The situation X is an approximation of the environment state. The situation can represent a vector of situation features, in which case the components of X are in general real numbers, or it can be a bit-singleton vector, i.e. vector having only one component with value 1, others being zeroes. In the sequel we will assume the latter case, and deal with a set of orthonormal vectors as situation inputs.

2) CROSSBAR ASSOCIATIVE MEMORY. It represents the *internal state of the system*. It is a matrix **W**, which components, as we will see, are used both for *state evaluation* (column-wise) and for *action evaluation* (row-wise). With assumption of bit-singleton vectors, each column of the matrix represents an *internal state* of the system *stimulated by the received situation*. Each component of that column vector is the evaluation component for an action that should be chosen *as reaction* to the stimulating situation. The matrix **W** can be viewed as a table of *situation-action-evaluation* (SAE) components. Saying in terms of dynamic programming, the matrix is carefully designed such that it can be used *both for policy iteration and state value iteration*, in a crossbar fashion.

3) STATE EVALUATOR. It evaluates the internal state of the agent. The internal state is a function of the situation the system is in, which in turn is function of the environment state. In terms of dynamic programming, this system computes the *state value* of the environment. It should be noted that the value of the state of the environment is evaluated *with reference to the internal state* of the agent induced by the considered environment state.

4) ACTION SELECTOR. It takes into accound two tendencies for taking action: one is the *action taking policy* learned and stored in the associative memory, and the other is the action proposed by the higher order system. Again, for computing actions it uses the SAE components associated with the received situation X.

5) HIGHER ORDER SYSTEM (SUPERVISOR). It is a component of the hierarchical control system of the CAA architecture. It can observe information from all the mentioned systems, and also from the environment, but can interfere only on two systems, the action selection and state evaluation, stating a *strategy* for them. A strategy for action selection may be, for example: "now apply random walk"; or "now do what you want"; or "now proceed the same way as before" etc. A strategy for state evaluation may be, for example: "it is good only if it is maximum", or "it is good only if it is minimum", or "it is good only if it is an average", or "it is good only if you are cautious, there is a danger associated", or "it is good now if you feel nothing anymore" etc. The latest modulation of feeling is important for learning in stochastic environments, on which we will talk later in the text.

6) GENOME VECTORS. The genome vector, when imported from the genetic environment, represents the *initial state* of the associative memory. It can also be exported, representing some *exporting state* of that memory. It is assumed that the "maturity", i.e. the exporting state, is achieved after some process of learning. It is detected by the supervising unit. Exporting strategy is part of the optimization process carried out in a CAA system. Initial state of the memory can be arbitrary, but we will further assume that all the column vectors of the memory matrix are zero except some of them, which are only with positive SAE values, or only with negative SAE values. The positive vectors represent the desirable initial states of the CAA agent, and the negative vectors are undesirable states. It is further assumed, that during the proces of evolution and implemented genetic algorithms in the genetic environment, the imported genome vectors represent the environment *properly*. If the environment changes dramaticaly, the CAA system with imported genome vector before that change, there is a probability that such "born" CAA system will not survive.

7) GENETIC ENVIRONMENT. This system connects the CAA concept to the evolutionary systems. It is assumed that exported genome vectors can be imported to some other species. Also, before that, some genetic operations can be performed over CAA genome vectors such as mutation, crossover etc. Also, some optimization computation can be performed in some evolutionary fashion. We will not be concerned how optimization processes are performed *in* this environment. We assume that in some way, if CAA exports a genome to this enviironment, some optimization process can be perfomed on that genome, by the same or some other species.

Now having explained the basic CAA anatomy and the interface toward the environments, in the sequel we will describe its physiology, i.e. how it works.

5.4. HOW IT WORKS:

5.4.1. THE ONE-STEP COMPUTATIONS

For a discussion about the computational procedure which is performed in the CAA, we need firstly to explain the one-step routines, which are part of that procedure.

Figure 5.4 shows the system memory, the action selector, and the state evaluator of the CAA. It is important to observe that the actions and the state values are computed *using the same memory matrix*. In terms of dynamic programming, the policy iteration and value iteration processes can be performed using the same matrix.

The architecture is generic, so there are some functions left unspecified. Those are functions e, f, g, and h. The functions e and f are responsible for the state-value computation, whereas the functions g and h are used for action computation.

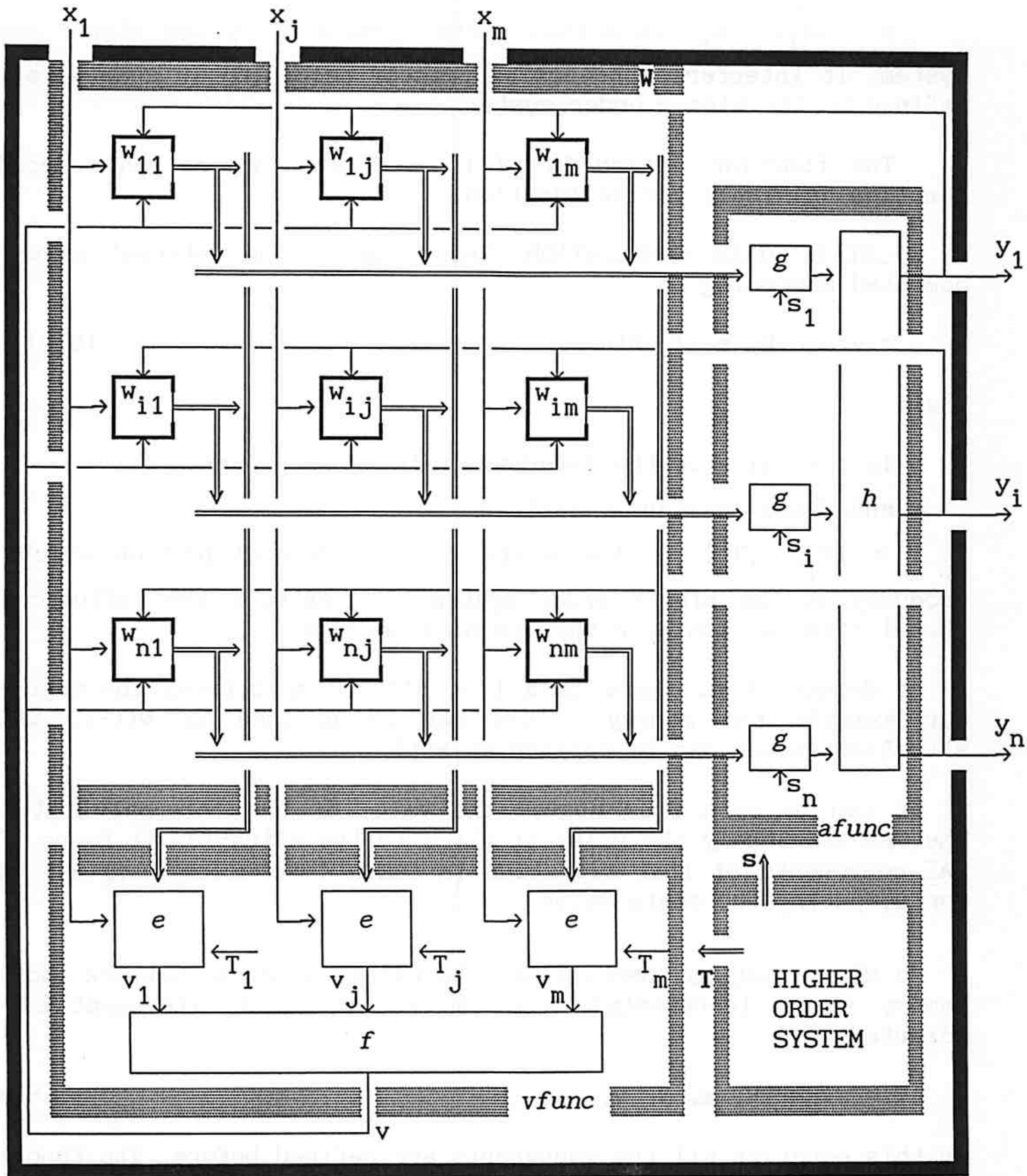


Figure 5.4. The CAA architecture

Now we will describe, in general terms, the action, state value, and the memory values computational processes performed in the CAA.

1) ACTION COMPUTATION. The action is computed according to

$$\mathbf{y} = afunc(\mathbf{W}, \mathbf{x}, \mathbf{s}) \tag{5.1}$$

where

$\mathbf{y} = (y_1,..,y_n)$ is bit-singleton action vector

$\mathbf{W} = \{w_{ij}\}$, i=1,..,n, j=1,..,m is the memory matrix of the SAE-components, or W-components. In general, they are real numbers.

$\mathbf{x} = (x_1,..,x_m)$ is bit-singleton situation vector

$\mathbf{s} = (s_1,..,s_n)$ is a real vector generated by the higher order system. It interferes the action process according to some strategy defined by the higher order system.

The function *action* is will be denoted as *action selection function* or simply *action function.*

2) STATE VALUE COMPUTATION. The value of the internal state is computed according to

$$v = vfunc(\mathbf{W}, \mathbf{x}, \mathbf{T}, \Delta t) \qquad (5.2)$$

where

v is the value of the internal state, a real number

$\mathbf{x}$ and $\mathbf{W}$ are previously defined

$\mathbf{T} = (T_1,..,T_m)$ is the state evaluation modification which is produced by the higher order system. It controls the influence of the elicted emotion over the learning process.

Δt denotes that there is a time difference between the arguments for example the memory values can be defined as $\mathbf{W}(t-1)$ while situation values can be defined as $\mathbf{x}(t)$.

As can be seen from above, the value of the internal state of the CAA is actualy the value it gives to the situation it faces. The SAE components *of that situation* is taken into acount and are used for computing the state value.

3) MEMORY VALUES COMPUTATION. The CAA is a neural network and its memory is of incremental type. In each step an increment $\Delta\mathbf{W}$ is computed as

$$\Delta\mathbf{W} = wfunc(\mathbf{W}, \mathbf{x}, \mathbf{y}, v, \Delta t) \qquad (5.3)$$

In this equation all the components are defined before. The function *wfunc* is the *learning rule* of the system.

5.4.2. THE SEQUENCE OF THE ONE-STEP COMPUTATIONS

The CAA is not a linear architecture, and, as Figure 5.5. shows, the computation loop is somehow "folded".

The left side of the Figure 5.5. shows a standard sequential computing architecture denoted by SA. On the right side it is shown that the CAA performs a crossbar computation which can be seen as *three passes* through the memory matrix: two times vertically and once horizontally *in the same time step.*

The first vertical flow (denoted with 1) passes through the current situation, the second vertical from passes through the previous situation, and the third flow passes through the current action.

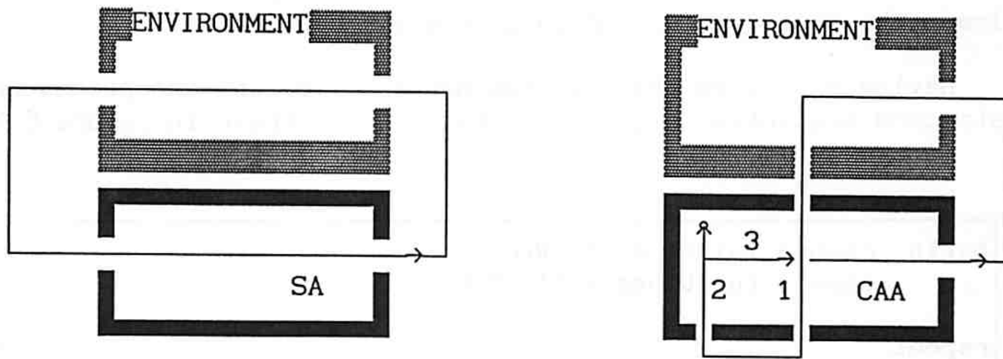


Figure 5.5. The three pass computation flow through the CAA memory

The flow through the CAA matrix can be "linearized", and the linearized sequence of steps is given on Figure 5.6.

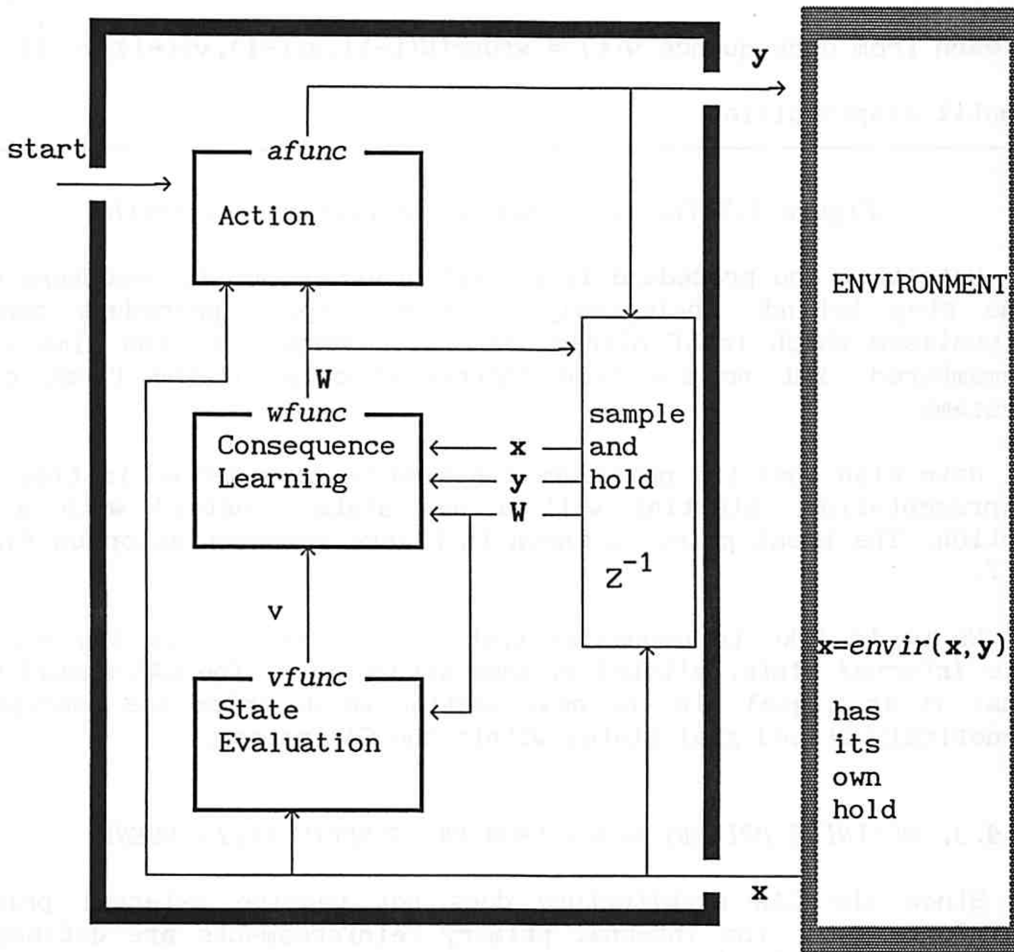


Figure 5.6. The CAA first order learning procedure

Figure 5.6. emphasizes that the behavioral loop *starts with an action.* It is an important feature of the reinforcement learning systems in general, self-reinforcement learning system being a subset of them. The CAA *tries* an action, the environment responds

with a new situation, and then the CAA computational procedure can be considered as an automaton routine: 1) perform evaluation, 2) learn the consequence, and 3) generate a new action.

Having explained that, we can now describe the CAA procedure in a standard sequential algorithmic fashion, as shown in Figure 5.7.

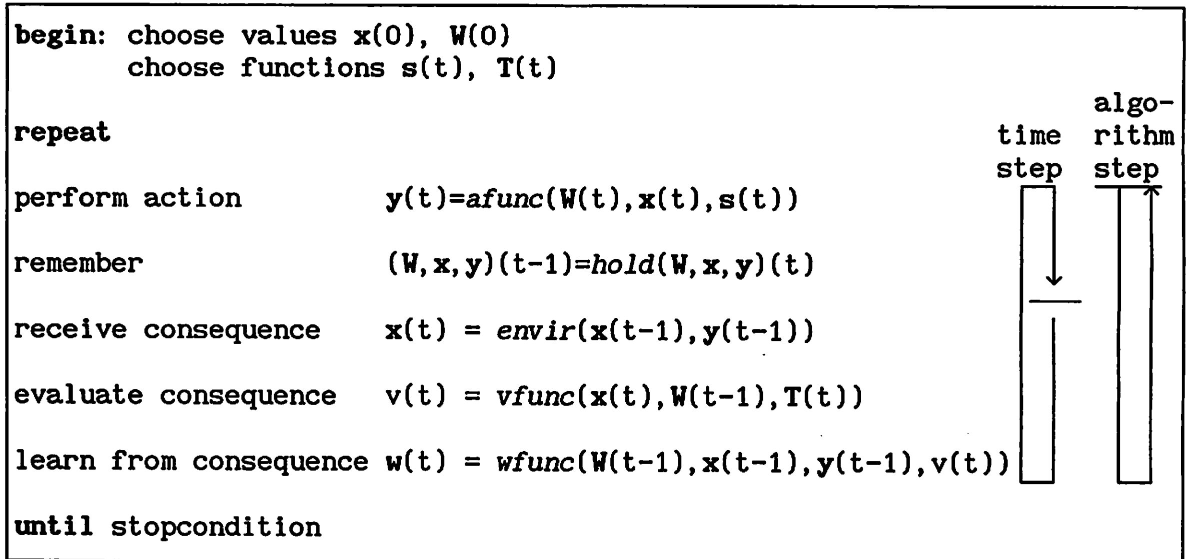


Figure 5.7 The CAA consequence learning algorithm

Note that the procedure is a first order, since it remembers only one step behind. Analogously, a more complex procedure can be visualised which is of higher order with respect to the time steps remembered. But we are here interested only in the first order systems.

Note also that the procedure can also be represented in *time step representation*, starting with a new state, instead with a new action. The break point is shown in Figure representation on Figure 5.7.

We would like to emphasize that the CAA recognizes the goal as its *internal state*, elicted by some situation **x**. The CAA should know that it is a goal. In the next section we describe the concept of geneticaly stated goal states within the CAA memory.

5.4.3. DEFINING PRIMARY GOALS FROM THE GENETIC ENVIRONMENT

Since the CAA architecture does not receive external primary reinforcements, the internal primary reinforcements are defined by means of initial values of the CAE-components in the matrix **W**. The initial values are defined as species vectors of the system. The initial values will affect: 1) column vectors: initial internal (or emotional) preference toward some environment states, and 2) row vectors: being in some state, preference toward some actions eligible in that state. Figure 5.8 gives an example.

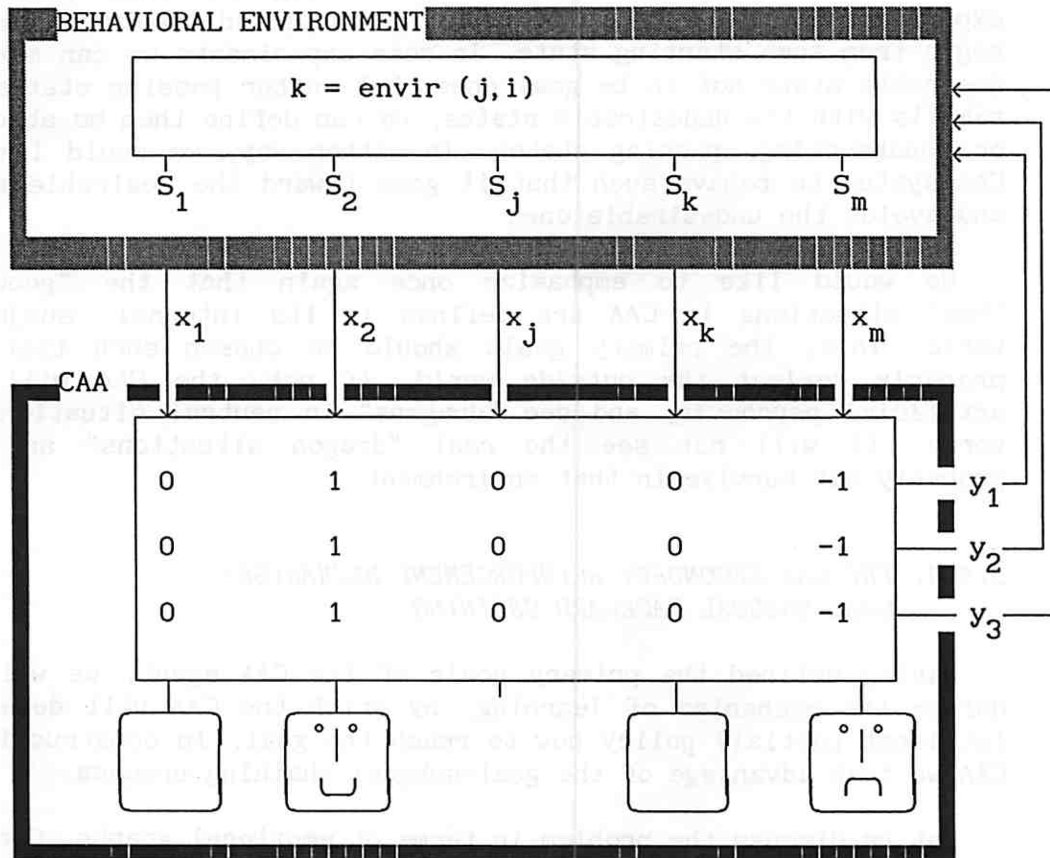


Figure 5.8. The initial definition of the primary reinforcers (positive, negative, and neutral subjective preferences toward the objective environment situations)

Figure 5.8. shows an arbitrary environment, *unknown to the CAA*, and an initial set of values for the CAA memory. Note that the matrix is chosen such that some column vectors have values +1, some have values -1 and some have zero values. Such a matrix can actualy be represented as a vector,

$$[0\ 1\ \ldots\ 0\ \ldots\ 0\ \ldots\ -1]$$

That vector we denote as a genome or a *species vector*. That vector is assumed imported from the genetic environment.

Having imported such a vector, as Figure 5.8 shows, the second situation (which represents the second environment state) is understood as a desirable situation, whereas the m-th situation is undesirable one. The desirable situations are usualy defined as goal situations whereas the undesirable ones should be avoided. From the Figure 5.8 we also see that because all the SAE values in the column vectors are equal, the CAA will not initialy have a preference toward any particular action. However, we can imagine that from the genetic environment could come an arbitrary genome, which is a result of a previous learning taken place in some other CAA systems, and it could cause a more specified initial behavior.

Having specified preference toward the situations, we can make some of them goal situations, if that state is defined as an

terminal, or absorbing state. Once being in that state, an experimental trial with a CAA system is completed; another trial can begin from some starting state. In some experiments we can make the desirable state not to be goal ones, but rather passing states. The same is with the undesirable states, we can define them as absorbing or nonabsorbing, passing states. In either way, we would like the CAA system to behave such that it goes toward the desirable states and avoids the undesirable ones.

We would like to emphasize once again that the "good" or "bad" situations in CAA are defined in its internal, subjective world. This, the primary goals should be chosen such that they *properly reflect the outside world.* If not, the CAA will have *artificial psychosis*, and see "dragons" in neutral situations; or worse, it will not see the real "dragon situations" and will probably not survive in that environment.

5.4.4. THE CAA SECONDARY REINFORCEMENT MECHANISM: GOAL-SUBGOAL BACKWARD CHAINING

Having defined the primary goals of the CAA agent, we will now define the mechanism of learning, by which the CAA will develop a (at least partial) policy how to reach the goal. In construction of CAA we took advantage of the goal-subgoal chaining process.

Let us discuss the problem in terms of emotional graphs. Consider the Figure 5.9. It shows two nodes in a graph, the j-th node and the k-th node. The number of action that can be taken from the j-th node is n. One of those ections, the i-th action, leads to the k-th node. The k-th node has its value, $val(x_k)$.

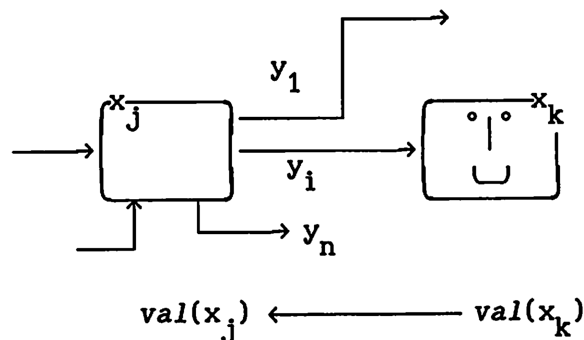


Figure 5.9. The goal-subgoal chaining step

Let the CAA agent takes the i-th action in the j-th situation, and experiences a pleasure, due to the evaluation of the k-th state. Since now it knowns that the k-th state can be reached from the j-th state, the j-th state will be treated as a discovered goal state, actualy a subgoal state toward the k-th state. In such a way, a backward chaining step has occured. The value of the goal state has been *backpropagated* to a preceeding state by some value-backpropagation function.

Here the k-th state is the primary reinforcer defined by the

system genetics, the j-th state is induced, secondary reinforcement, defined by conditioning. The concept can be extended to higher order conditioning. As a result, *by doing*, the system will *learn a plan* how to go from some starting point to a goal point.

Let us consider the whole value backpropagation process in the CAA architecture. Figure 5.10 shows the process loop, consisting of action generation, new situation production, subjective consequence evaluation, and subjective *learned moral* backpropagation and learning. Here we mention the concept of *moral* in a sense of learning a moral from a told tale, or from a performed action.

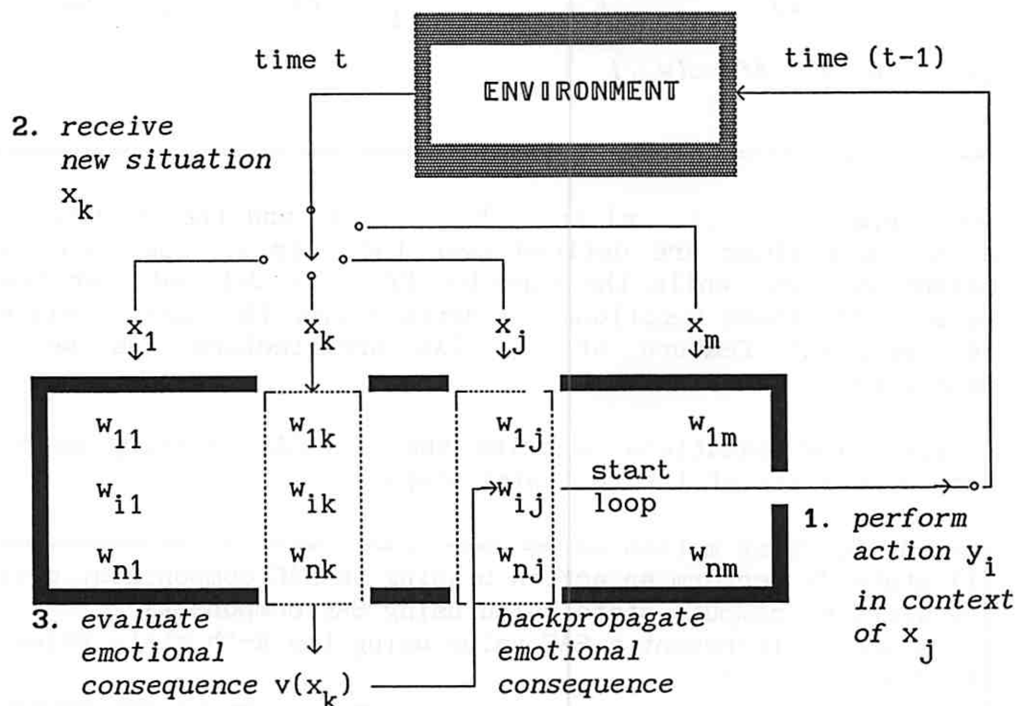


Figure 5.10. The CAA state value backpropagation (goal-subgoal chaining) process

As we can see from the Figure 5.10., the loop starts, let us say, with the i-th action in context of the j-th situation. That produces environment reaction, and as consequence, the k-th situation is received. That consequence is evaluated, and the reinforcing signal is sent to update the strength of the association between the j-th situation and the i-th action. If the internal reinforcement was positive, the system has learned two things: 1) taking i-th action in the j-th situation is good, since 2) the i-th action taken in j-th situation leads to the k-th situation which is evaluated as a desirable situation. In that way the CAA *is discovering its goal*, the pleasure which is met in the k-th situation. The same will happen for the j-th situation, since now it will receive a positive SAE value, by the process of learning. It is to be noted however, that the j-th situation will have only one SAE component positive, while in k-th situation all the SAE components were positive. Now, being in the j-th situation, the CAA agent *will have a preference* toward taking the action which will lead toward the goal.

5.4.5. THE CAA LEARNING METHOD

Examining the Figure 5.10. we can write the following basic equations of the CAA architecture

CAA learning method

state j: $y = \underset{a}{Afunc}\{w_{aj}\}$ result y = i (5.4)

state k: $v_k = \underset{b}{Vfunc}\{w_{bk}\}$ (5.5)

state j: $\Delta w_{ij} = Ufunc(v_k) = Ufunc(\underset{b}{Vfunc}\{w_{bk}\}) = \underset{b}{Wfunc}\{w_{bk}\}$ (5.6)

state k: $y = \underset{b}{Afunc}\{w_{bk}\}$ (5.7)

where $a,b = (1,..,i,..n)$ and $j,k = 1,..,m$, and the functions *Vfunc, Afunc,* and *Wfunc* are defined over the sets of components of the column vectors, while the function *Ufunc* is defined over the state values. All those functions are defined over the same matrix **W**. This is important feature of the CAA architecture, as we already mentioned.

The above equations describe the the *CAA learning method.* The method consists of the following steps

CAA learning method

1) state j: perform an action biasing on SAE components; obtain k
2) state k: compute state value using SAE components
3) state j: increment a SAE value using the k-th state value.
4) j = k; goto 1

So, in a sort of forth-and-back computational procedure, the CAA increments its SAE components and performs learning. Figure 5.11. shows how the CAA method is viewed in some state space.

As Figure 5.11 shows, the method can be visualised such that each node of the graph is assigned a state value and each action is assigned a SAE component, an element of the memory matrix. So, tne CAA agent being in state j, takes the actin i, enters the state k, takes its state value, returns it to the previous state, updates the values in the previous state, and then takes action to from the k-th state. The whole procedure is in rhythm forth-back-forth. This *rhythm* is known in AI, and is used in evaluating game trees (e.g Samuel 1959). The CAA procedure is distinct with usage of its SAE componets in all the three steps of the rhythm.

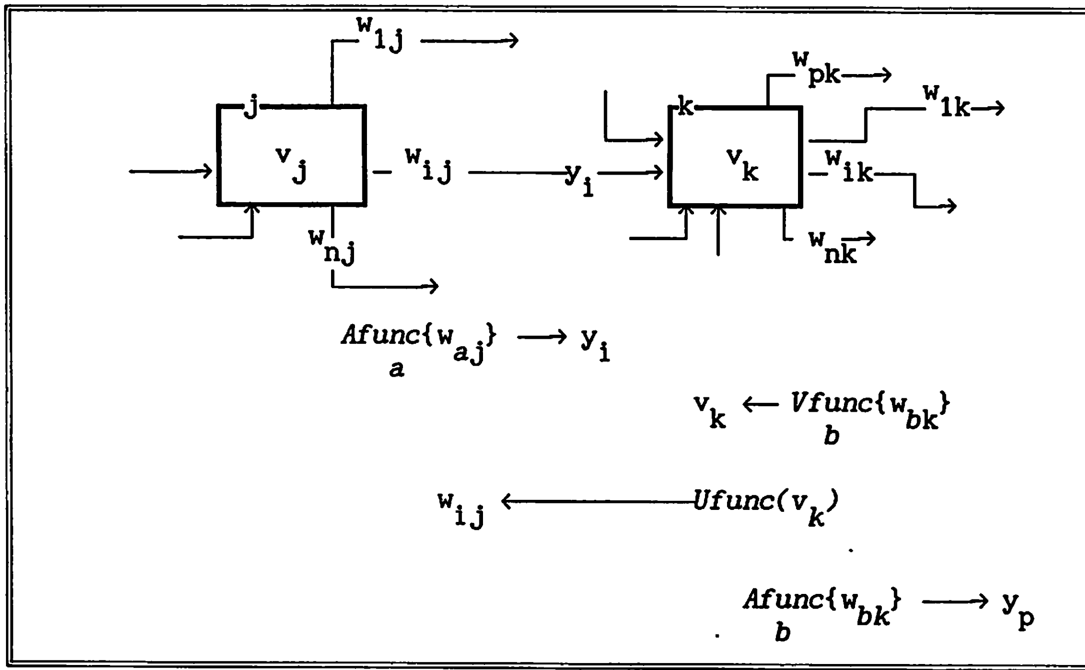


Figure 5.11. The CAA learning method

Let us note that the CAA learning method is given here in generic terms. Specification of the functions *Afunc, Vfunc, Ufunc (and Wfunc)* is dependent of the considered specific task. To ensure random behavior, we state that *Afunc* computes a *bias* for an action, not the action itself.

The *computational method* shown on Figure 5.11 is important for some modern trends in dynamic programming about which we will talk in the next chapter.

5.4.6. THE HIGHER ORDER SYSTEM: MODULATING ACTIONS AND EMOTIONS

The higher order system is the CAA architecture plays a role of long range behavioral modulator, in some sense a "hormonal system" of the CAA agent. Its internal structure is left *undefined*. Here wi will consider how it modulates the action (intention) and state-evaluation (emotion) of the CAA system.

MODULATING ACTIONS. The action selection process as stated above is given by

$$y = \underset{u}{Afunc}\{w_{uj}\} \qquad (5.8)$$

if the higher system modulation is not present. If it is present, we have

$$y = \underset{i}{Afunc'}\{w_{ij}\ ,\ s_i\} \qquad (5.9)$$

where *s* is a real number which we denote as searching strategy parameter, and *Afunc'* is the modified action selection function by

introducing the parameter vector s. It is generated according to some action taking strategy of the higher level system. For example, if the higher level system decides that the acton-taking strategy should be a random walk, then the set {*s*} is generated according to some uniform distribution. In that case, the SAE components are only bias, and the behavior will actualy have a statistical character. The random walk strategy is used when the CAA system performs exploration in an unknown state space. Even if a policy has been learned, and SAE components have some values, the higher order system can introduce a random walk activating sufficiently large random values of its searching strategy parameters.

MODULATING EMOTIONS. The emotion computation is modulated as

$$v = \underset{i}{Vfunc'}\{w_{ij}, T_j\} \tag{5.10}$$

where {T} are parameters of some emotional value selecting strategy. For example, with this parameter the behavior can be modulated to be more cautious, or more agressive. For example, if there is a danger ahead, should warning (in form of fear) be generated or not. If warning is not generated, the CAA will have no concept of risk and pursue as usual. If warning is generated, than possibly there will be some more explorations before a state is designed as desirable.

5.5. EXAMPLE OF A CAA ARCHITECTURE

So far we have discribed the generic CAA architecture and its features. In this section we will give an examplefication of the CAA architecture. Figure 5.12 shows an example of a CAA architecture. In this CAA system the functions *e, f, g* and *h* are chosen to be *sign* Σ, *mux*, Σ, and *max*.

5.5.1. ACTIONS: LEARNED AND MODULATED

The action computation in this CAA is performed in a *neural* way

$$y_i(t) = \begin{cases} 1 \text{ if } g_i(t) = \max\limits_h \{g_h(t)\}, \; h=1,..,n \\ 0 \text{ otherwise} \end{cases} \qquad i=1,..,n \tag{5.11}$$

where

$$g_i(t) = \sum_{j=1}^{m} w_{ij}(t)x_j(t) + s_i(t) \tag{5.12}$$

Since we are assuming that the input vectors are orthonormal, and each component represents a situation from the environment, the above equation reduces to

$$\text{for all } j=1,..,m \quad \text{if} \quad x_j = 1 \text{ then } \quad y_i = \text{sgn} \max_i \{w_{ij} + s_i\}, \tag{5.13}$$

where sgn( ) is a function which gives 1 for nonnegative and 0 for negative arguments, and

$$s_i = montecarlo_i[-.5,+.5] \tag{5.14}$$

where *montecarlo* is a random function which gives values which are *uniformly* distributed in the defined interval. Having that, the CAA will perform random walk until the SAE components receive values which will dominate the behavior. Gradualy, the behavior will shift from nondeterministic to deterministic one.

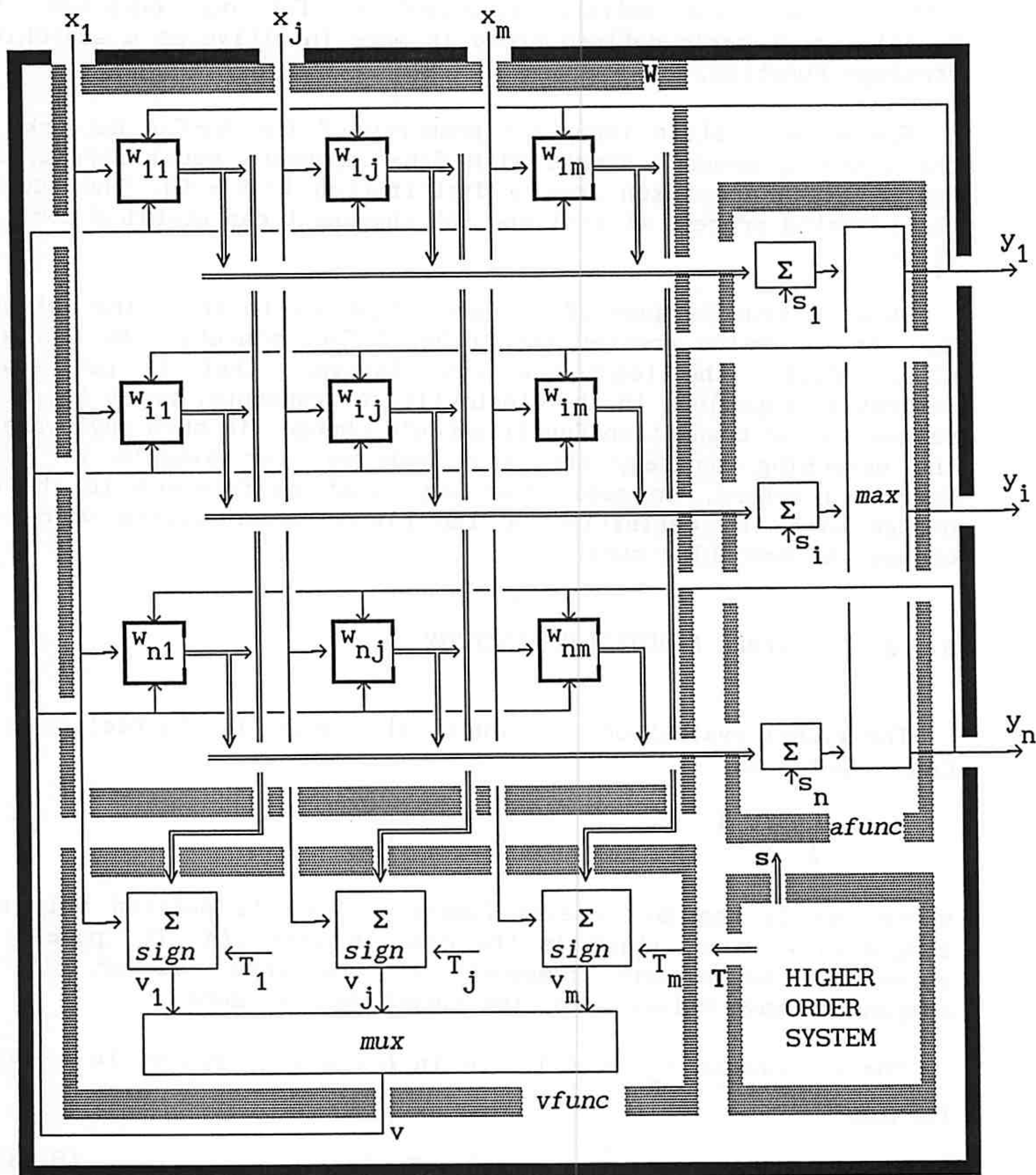


Figure 5.12. Exemplification of the CAA architecture

However, sometimes it is desirable CAA to behave randomly even after learning, for example to explore some environment. Than the searching strategy can generate s-componets with values

$$s_i > \max_h \{w_{hj}\} \tag{5.15}$$

Another example is if {s} are chosen such that

$$s_i = \max_h \{ 0, w_{hj} \} \quad (5.16)$$

In this case the negative SAE components will not affect the action selection.

Let us note that in the modern theory the searching strategy is implemented as a "temperature function" (Keerthi and Ravindan 1995) instead the uniform distribution. For our purposes, the function *montecarlo* defined above is more intuitive as a searching strategy function.

Now we will state important property of the NN-CAA network. If the learning process starts with SAE-components equal zeroes, and the s-values are taken from a distribution [-.5,+.5], then during the learning process at most one SAE-component can obtain a positive value.

This is true because of the greedy policy in selecting actions. Once having value greater then other SAE-components, the CAA will always follow the learned action. However, that is not always desirable, especialy in the stochastic environments, where the state values and/or transition function could change. In such environments the seraching strategy should be changed, for example in a way described before, or some other way. That is foreseen in the CAA design with the definition of the higher order system which can change the searching strategy.

5.5.2. THE STATE EVALUATION FUNCTION

The global evaluation function of the system is its feeling v, in CAA computed as

$$v = \mathit{mux}_k \{v_k\} \quad (5.17)$$

where *mux* is the *multiplexer* function which is defined only over singleton vectors, which is the case in this CAA. It "pases" the value of the nonzero component of the input vector. If all components have values zero, the output is also zero.

The components v_k, $k = 1,..,m$ in are also computed in a neural fashion:

$$v_k(t) = \mathit{sign}\,(x_k(t)(\sum_{i=1}^{n} w_{ik}(t-1) + T_k)) \quad (5.18)$$

where *sign*(.) is a function which gives 1 for positive, 0 for zero, and -1 for negative arguments. Here the temporal difference is important. It emphasizes that the situation represented by x_k will be evaluated by the vector w_k which has been computed in the previous step.

The above equation can be rewritten as

$$v_k(t) = \begin{cases} sign\ (\sum_{i=1}^{n} w_{ik}(t-1) + T_k) & \text{if } x_k(t) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.19}$$

The value of T_k is computed as

$$T_k(t) = \omega\ \{w_{ik}(t-1)\} \tag{5.20}$$

where ω is some *warning* function, which monitors the appearences of negative SAE-components in v_k . It is actualy a *strategy* how to deal with negative SAE-componets in the evaluation of the state.

To explain that, let us rewrite the state value computation equation as

$$v_k(t) = sign\ (E_k^+ - E_k^- + T_k) \tag{5.21}$$

where E_k^+ is the sum of all positive and E_k^- the sum of all negative SAE-components for the in the evaluation vector for the k-th state. The term E_k^- has important interpretation: it is a number of closed alternatives (number of actions that was learned not to be taken) in the k-th state.

The warning function is defined as

$$T_k = \begin{cases} 0 & \text{if} \quad E_k^- = n \\ C_k & \text{if} \quad n > E_k^- > E_k^+ \\ 0 & \text{if} \quad E_k^- \le E_k^+ \end{cases} \tag{5.22}$$

Let us consider all the alternatives:

If $E_k^- = n$, that means all the alternatives for taking actions in the k-th state are closed. So, there is no sense of going to that state and the state value is

$$v_k = \text{sign}\ (0 - E_k^- + 0) = -1 \tag{5.23}$$

Expressing it in terms of the set of SAE-components we have

$$v_k = sign \max_i \{w_{ik}\} \tag{5.24}$$

If $E_k^- < E_k^+$ that means that in the k-th state there are actions leading toward a goal state, so the value of the k-th state is

$$v_k = sign\ (E_k^+ - E_k^- + 0) = +1 \qquad (5.25)$$

If $E_k^- = E_k^+$ then it is the case when the state value is zero

$$v_k = sign\ (E_k^+ - E_k^- + 0) = 0 \qquad (5.26)$$

If $n > E_k^- > E_k^+$ then we have cases where there should be decided

when to signal to the previous (e.g. j-th state) that this (k-th state) has alternative paths leading toward a dangerous state. This state evaluation sheme includes warning (caution) threshols C_k. If $C_k=2$,

then the j-th state will not receive the value p=-1 before at least two alternatives are closed in the k-th state. If we do not require warning signals, then we define $C_k = E_k^-$. In that case we have a *warning-free*

state evaluation strategy which is expressesd as

$$v_k = sign\ (E_k^+ - E_k^- + E_k^-) = sign\ E_k^+ \qquad (5.27)$$

If we define the state evaluation strategy to be a warning-free one, we have

$$T_k = \begin{cases} 0 & \text{if} \quad E_k^- = n \\ 0 & \text{if} \quad E_k^- \leq E_k^+ \\ E_k^- & \text{if} \quad E_k^- > E_k^+ \end{cases} \qquad (5.28)$$

which gives a definition of state evaluation function as

$$v_k = sign\ \max_i \{w_{ik}\} \qquad (5.29)$$

This state evaluation function gives no backpropagation signal to the j-th state if there are negative alternatives in the next, k-th state. As long as the state have at least one nonnegative alternative the k-th state is considered neutral. It will be considered negative, only in the case when all the alternatives in the k-th state lead toward a negative outcome.

In short, the "threshold value" T plays a role of the modulator of the cautions with which the CAA will evaluate the situations which are on the way. If the warning is disabled, the CAA will evaluate acording to a greedy policy. If the warning is included, the CAA will have to visit a situation several times before it decide it is a "good" situation and will backpropagate a signal that from that moment on that situation should be considered to represent a subgoal state.

5.5.3. THE LEARNING AND VALUE BACKPROPAGATION FUNCTION

The learning function (learning rule) is defined as

$$w_{ij}(t) = w_{ij}(t-1) + x_j(t-1)y_i(t-1)v(t) \quad (5.30)$$

where $v(t) = val(x_k)$ (5.31)

and $x_k = envir(x_j, y_i)$ (5.32)

This function is actualy the *value backpropagation function:* the value of the k-th state is affecting the i-th evaluation component of the j-th state.

Let us note that the learning rule could affect all the states *except the absorbing* ones. If a state is initialy valued, but not absorbing, its value can be modified by this value backpropagation process. That is a desirable feature in some problems, where we would like to have a behavior which directs toward a pleasant goal state but has to to pass *through* an unpleasant state along the way. Also, is some problems we would like the CAA agent to go toward an absorbing goal state, but also to tend to pass through some other, non absorbing pleasant states.

5.6. SOLVING PROBLEMS WITH THE CAA ARCHITECTURE

This section will illustrate the ability of the self-reinforcement learning systems to solve some problems. In particular, the examplefied CAA architecture will be considered. Two general problems will be considered, delayed reinforcement learning and learning in loosely defined graphs problem.

5.6.1. LEARNING IN EMOTIONAL GRAPHS: DELAYED REINFORCEMENT LEARNING PROBLEM: MAZE RUNNING EXAMPLE

Here we will consider the problem of assignment of credit, stated by Minsky (1961). The notion of *delayed reinforcement* is also used to describe the paradigm. We already talked about this paradigm in the second chapter when we listed the learning paradigms. The task of a learning agent is to exhibit a learning behavior if the teacher (environment) just observes the learners behavior and gives neither advice nor evaluation for a long period of time. However, occasionaly an evaluation is given, and the learner should figure out which action in the past was responsible at most for the obtained reinforcement at the present time. Contemporary it is a widely investigated paradigm in external reinforcement learning systems, and it is interesting to see how a self-reinforcement learning system will solve such a problem. In this section we will use the described CAA network and experimentaly prove a solution of such a problem.

A typical metaphore for a delayed reinforcement learning paradigm is the *maze learning problem,* widely investigated in animal learning

theory. However, here we will talk in terms of *arcade games* instead of animal mazes. A typical map of such a game is given in Figure 5.13.

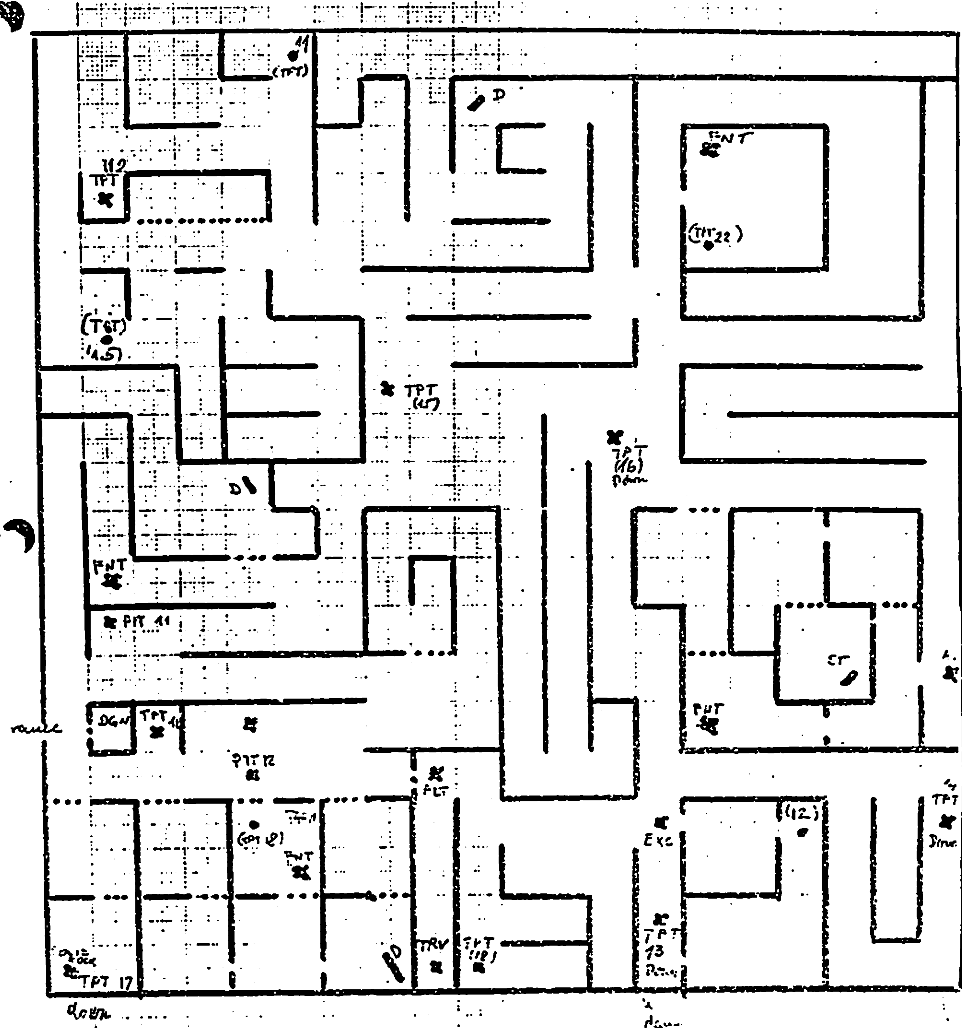


Figure 5.13. A typical environment for CAA (Courtesy of Daryl Lawton, 1981)

Figure 5.13 shows the map of a dungeon "Telengard" of the computer game "Dungeons and Dragons" (Lawton 1981). This game, which we played a lot, motivated the task which was stated as a challenge of the CAA in 1981.

As Figure 5.13 shows, a dungeon is a "grid world". An agent is entering that world looking for the "ORB" (whatever it is). Each place (node) of the world has adjacent at most four places to which can move using actions E(ast), W(est), N(orth), S(outh). Some of the movements are restricted by walls. There are also some *teleportation* places: from that place the agent will be transported to a nonadjacent node, possibly on some other level, such that no knowledge of the vicinity is known. There is no model of the world: a player must explore the world in order to (at least partialy) learn a policy for behaving in this world.

Let us note here that the map similar to the map drawn on the Figure 5.13 has become popular maybe ten years later among the researchers in reinforcement learning, when reinforcement learning

was connected to dynamic programming. As it turns out in 1981 we considered fairly complex task even with respect to the problems considered in real-time dynamic programming 10 years after. Let us continue with description of our challenging task.

Along the way the agent encounters *bad events* and *good events*. A bad event could be entering a dragon layer, or stepping on a bottomless pit. A good event could be finding an armor which will increase the agents level of strenght. Besides making decisions which way to go, the agent is constantly under decision pressure. For example, if you find a book should you read it: you never know whether you strenght will increase or decrease after reading a book.

A simplified model of the environment shown in Figure 5.13 is given in Figure 5.14. Each room of the dungeon is a state of the environment.

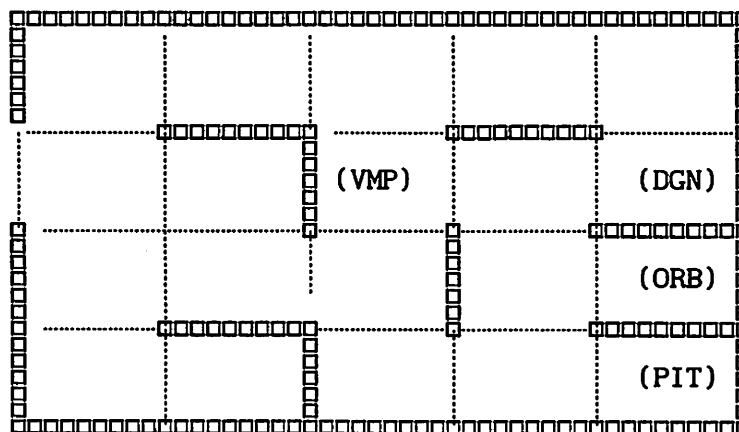


Figure 5.14. A colored grid world environment

Environments of the type shown in Figure 5.14 we call *Dungeons-and Dragons* (DND) environments. They are characterized by a *nonderivable value function*. If a value of a state is "good" the adjacent state can have a value "bad". A good state can even be surrounded by bad states, such that an agent should pass a bad state in order to enter a good state. In this example, there is a state denoted VMP where a vampire resides, a state DGN where a dragon resides, a state PIT which is actualy a bottomless pit, and a state ORB which is a good, goal state. It is assumed that all the "event-colored" states are terminal states. For this environment, which gives evalation only at the event-colored states, the CAA agent can learn (at least a partial) policy for its behavior. The task for CAA is to find a path in this graph toward the goal state avoiding the unpleasant state.

It is convienent to represent the statement of the problem in form of subjective emotional graph, as in Figure 5.15. This representation is more general since it can easily shown teleportation movements, which in gridworld representation are not natural. Also, from the emotional graphs we can immediately recognize good and bad states.

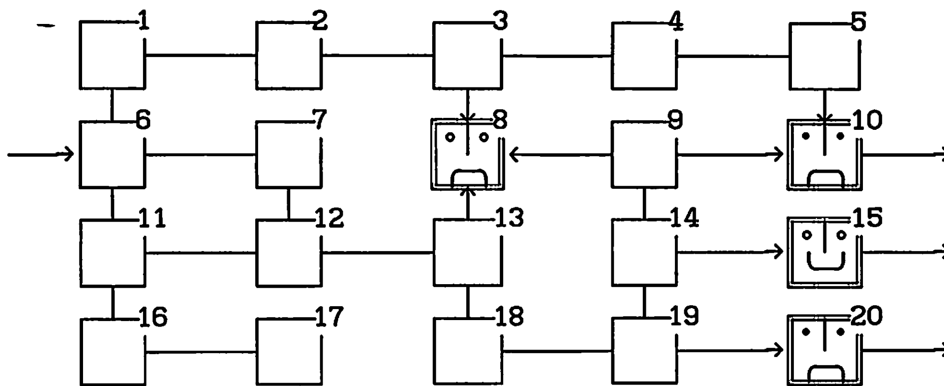


Figure 5.15. The emotional graph for a DND environment

Now, when we described the problem, let us discuss briefly the importance of this problem from the standpoint of 1981, when it was stated.

This is an assigment of credit problem. Stated by Minsky (1961), that problem has been considered in AI, and solution existed, examples being Samuel's checker player program [Samuel 1959]. However, there was no solution of that problem by a neural network. That was recognized within the Adaptive Networks Group, and was stated as a challenge. We took that challenge to solve the problem with the CAA architecture. Indeed the CAA development was influenced by the graphs as the one shown on Figure 5.15.

Now we will descibe the solution of *a delayed reinforcement learning problem* stated as: Find a path in the graph shown in Figure 5.15, where reinforcement is experienced only at the emotionaly colored states.

Since in this problem we have only three levels of desirability (desirable, undesirable, and neutral) the CAA genome vector will have ternary values (1, -1, and 0 respectively). So, we will start the CAA with the initial conditions which will assign initial values to the internal states according to the task stated. Figure 5.16. gives that assignment.

As Figure 5.16 shows, the assignment of state values is internal, subjective. The environment and its "real" values are not known to CAA, neither are known the transitions between states. All it is assumed is that the states are distinguishable to the agent and there is enough column vectors to be assigned to the environment states.

Having number of environment states, we define the number of columns of the memory matrix. The number of rows are defined by the maximal number of actions taken from a state in the environment (the fan-out of a state). In this environment it is 3, and that is reflected on the chosen CAA architecture.

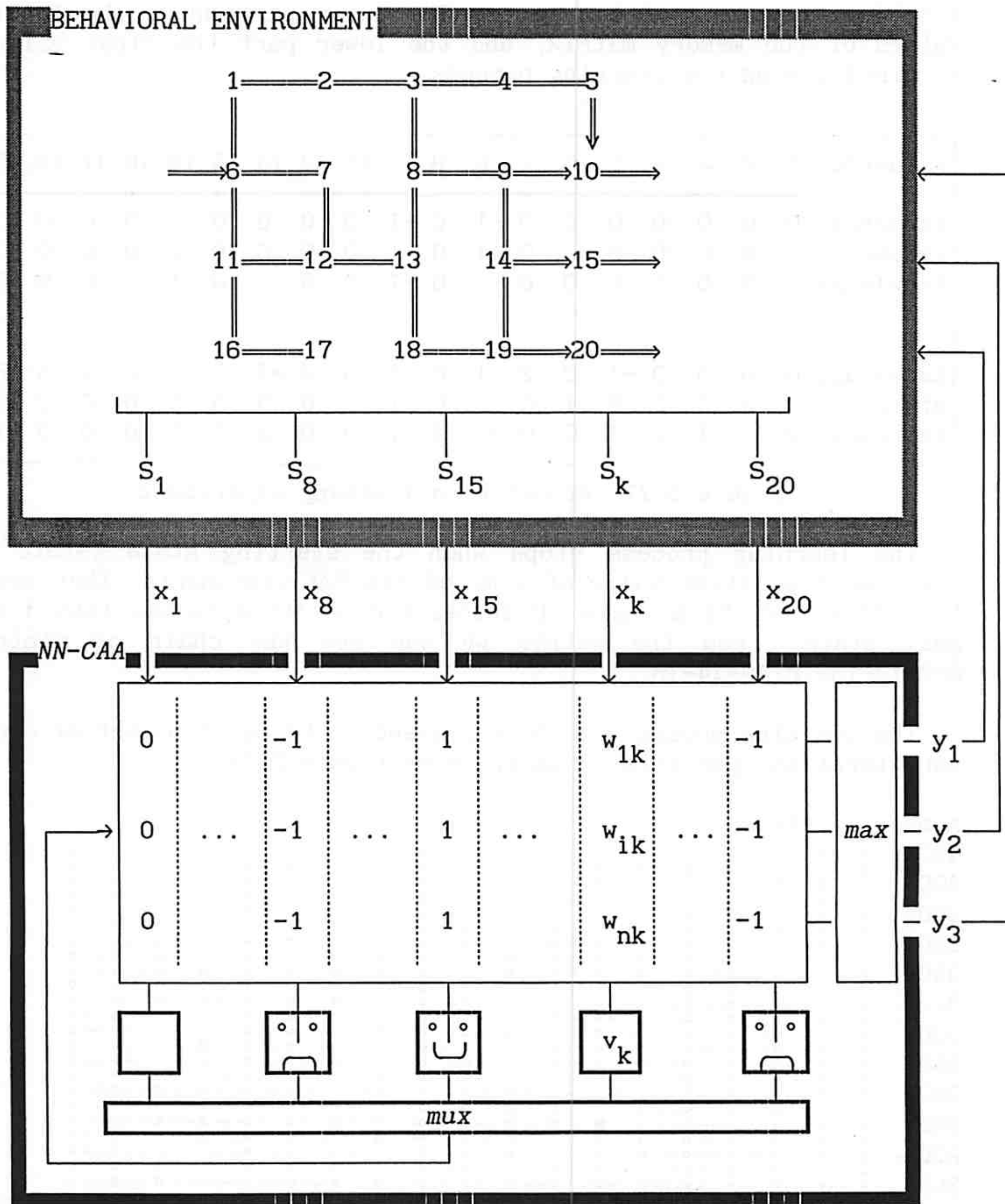


Figure 5.16. Internal representation of the preferencies toward the environment states in the CAA agent

A typical experiment consists of several *trials*. In each trial the CAA agent is wandering randomly through the graph until it reaches a goal place or some other absorbing place. Each trial starts from the same entry point. After number of trials the agent is learning how to go from the starting place to the goal place directly, without wandering around. The agent is learning a plan how to find the goal place starting from the entering place, avoiding unpleasant states.

Figure 5.17 shows result of a simulation experiment. It is shown a table consisting of two parts. The upper part shows the initial values of the memory matrix, and the lower part the final values, acquired during the learning process.

situation	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
imported	0	0	0	0	0	0	0	-1	0	-1	0	0	0	0	1	0	0	0	0	-1
genome	0	0	0	0	0	0	0	-1	0	-1	0	0	0	0	1	0	0	0	0	-1
knowledge	0	0	0	0	0	0	0	-1	0	-1	0	0	0	0	1	0	0	0	0	-1
knowledge	0	0	0	0	-1	0	2	-1	0	-1	0	3	-1	9	1	0	0	6	8	-1
after	0	0	0	0	0	1	0	-1	0	-1	0	0	0	0	1	0	0	0	0	-1
learning	0	0	-1	0	-1	0	0	-1	0	-1	0	0	5	0	1	0	0	0	-1	-1

Figure 5.17. Result of a learning experiment

The learning process stops when the starting state (state 6) computes a positive value of some of its SAE-components. That means that *there exists* a chain of increasing positive values toward the goal state. From the matrix we can see the chain of states: 6-7-12-13-18-19-14-15.

The learning process can be expressed in terms of number of steps per iteration (per trial), as shown on Figure 5.18.

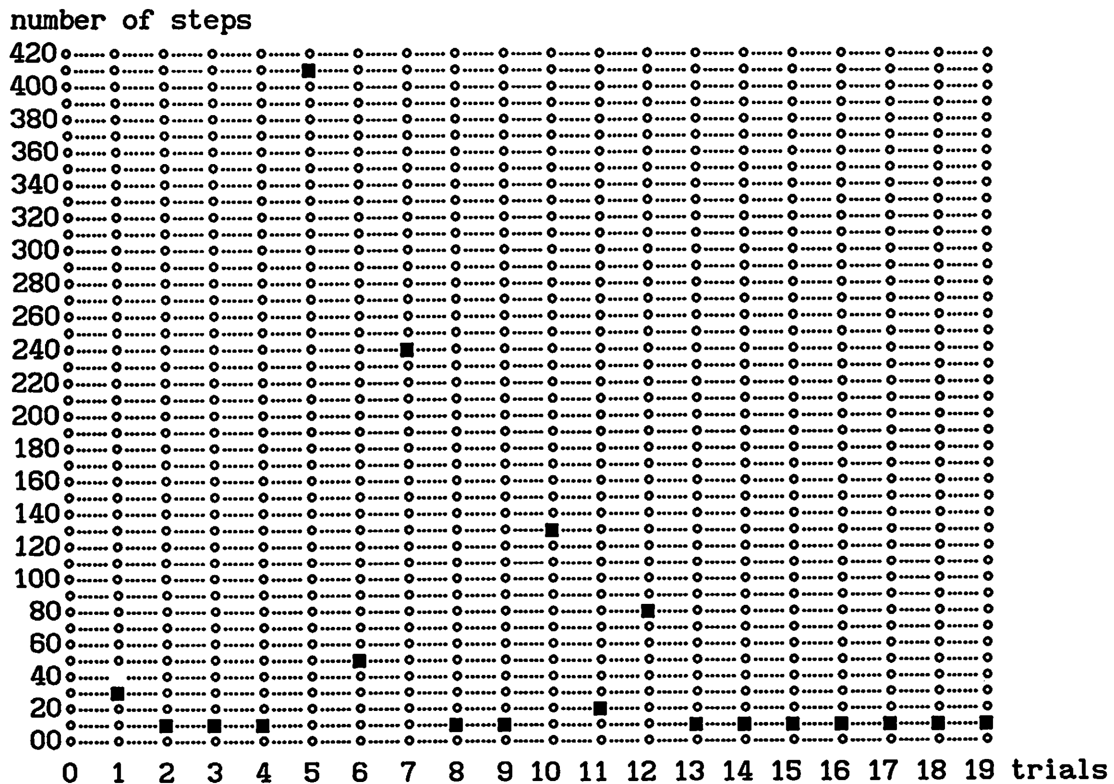


Figure 5.18. A learning curve in terms of number steps per trial

As result of the learning process, the CAA agent has learned a *policy* how to behave in this environment, and that policy is written in its memory matrix. The representation of that policy back to the initial problem space is shown on Figure 5.19.

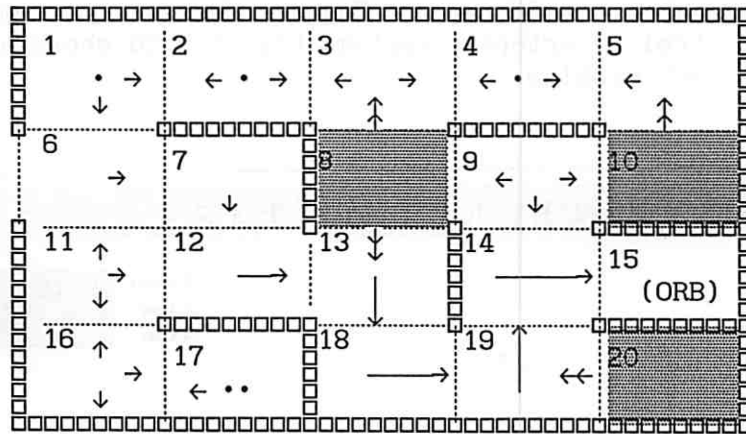


Figure 5.19. The policy learned in the described experiment

From the Figure 5.19 the reader can see that not all the states are assigned a policy other then random one. For example states 9, 11, 16, and 17 are assigned merely a random policy. The states with assigned policy have arrows which gradualy elongates (greater SAE values), representing the level of certainty (actualy expectancy) to which the action is taken. Some states (3, 5, 13, and 19) have experienced undesirabile consequence of some actions ity of some actions, and thay have memorized actions that should not be taken again (represented with double arrow repelling from a state). The dots in some states represent the action "stay in place".

The experiment shown above is one of the first CAA experiment in solving the maze running problem and by them the delayed reinforcement learning problem. As we said before, the development of the CAA was done in parallel with the AC architecture. The fact that the CAA architecture produced the solution faster, is not of particular importance. Those architecture concerning the maze running problem should be considered in parallel.

The path found in this experiment is the only one possible and in that sense optimal. In general a single CAA in one solution cyclys will not find the shortest path. The optimization takes place in the supervising unit. It counts the length of "the shortest path produced so far", and will not allow solutions with longer path to be exported. In such a way, CAA has offsprings with gradually improved ability to live in the considered environment.

In short, CAA architecture takes advantage of the genetic environment to solve the problem of optimality. But, more crucial is that this neural network was able in 1981 to solve the credit assignment problem in a self-reinforcement fashion.

5.6.2. LEARNING IN LOOSELY DEFINED EMOTIONAL GRAPHS: THE UNDETERMINISTIC ENVIRONMENT PROBLEM: POLE BALANCING EXAMPLE

The next problem considered by CAA in 1981 was the problem of learning to control a cart-pole system. Figure 5.20 shows a computer animation of that problem.

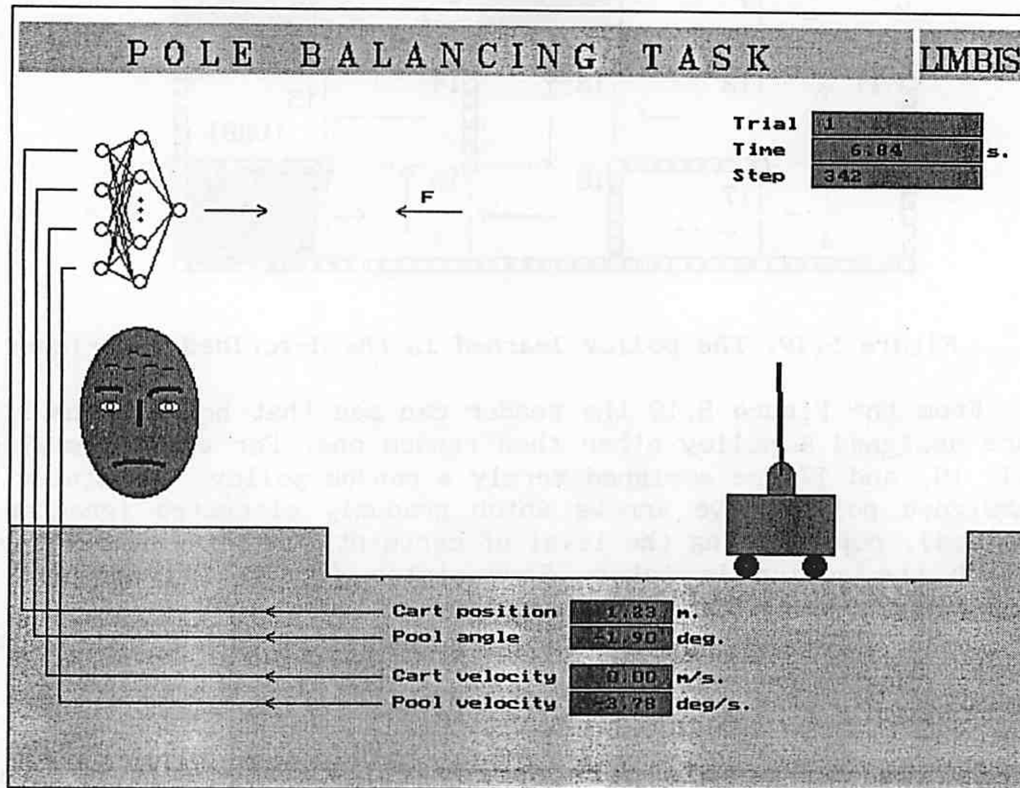


Figure 5.20. The cart-pole system
(From Bozinovski and Beochanin, 1994)

Figure 5.20 shows a symbol of a computer network which controls the cart-pole system by a discrete force. A human face animation is used to represent the desirability of the current state. The variables of the process (pole angle θ and agular velocity ω, as well as cart position xand linear velosity v) are mapped into the eyes, mouth, forehead, and the eyebrows of the human face. The face smiles when the cart is in the center of the screen and the pole is in upright position, and it changes its emotional expression gradually as the state of the process deviates from the desired one. We use human face emotional expresion to represent a multidimansional optimization problem.

The problem is to develop a self-learning system which will balance the pole. The pole-balancing problem is widely well-known. The are two versions of the problem: In the first version, the controler's task is to balance the pole *regardless* the position of the cart; in the second version the pole should be balanced keeping the position of the cart within certain limits, as is shown in

Figure 5.20. The first version will be denoted as *context free* pole-balancing problem, and the second version will be denoted as context-restricted pole baancing problem. Note that in both versions, both the cart and the pole are controlled. The difference is what is optimized. In praxis, the restricted optimization of θ will require movement of the pole in very narrow discourse (e.g. $\pm 6^{\circ}$). If only θ is optimized, the discourse can be much larger (e.g 45°). We choosed the context-free problem.

The reason why the context-free version was chosen is because it is simply representable as emotional graph. In addition, there is no loss of generality: the idea is to show an example of a self-learning neural control, and for that task the context-free version is sufficient. If a problem is represented as emotional graph, a CAA architecture can be implemented *as a tool* for solution. We wanted to explore the problem of applicability of CAA to the pole balancing control.

5.6.2.1. Representing the problem as emotional graph: Partitioning the problem space: Value function approximation

In order to represent the problem as (discrete) emotional graph the (continuous) control space of the problem should be partitioned into control *regions*. The problem was previously considered by michie and Chambers (1968) who divided the control space in 162 regions. Since we considered simpler, context-free θ-optimization, in our experiments [Bozinovski 1981g, Bozinovski and Anderson 1983] we divided the control space in 10 regions, using the heuristics shown in in tabular form on Figure 5.21.

state	1	2	3	4	5	6	7	8	9	10
defined by										
θ	-	-	-	[±ε]	[±ε]	[±ε]	+	+	+	$\lvert\theta\rvert > \theta_{crit}$
ω	-	0	+	-	0	+	-	0	+	
action	-F	$\overline{+F}$	0	0	0	0	0	$\overline{-F}$	+F	giveup

Figure 5.21. Heuristics for definition of the states for the context-free pole balancing task

The basic or the heuristics used is definition of two angles denoted as *critical angle*, θ_{crit}, and *goal defining angle*, ε. The angle ε is introduced because the intuitive goal state (θ=ω=0) is difficult to achieve, and thus the goal should be defined in a *fuzzy* rather then in a crisp fashion. If the angle is within the region [-ε,+ε] we say that the system is in its goal region. If the controlled angle is ouside the region defined by the critical angle, the trial is denoted as a failure trial. The measure of performance of the system can be the percentige of time spent in the goal region, or the total number of failure steps, in a predefined period

of time. Both the angles are defined heuristicaly, depending on the force used to be used for moving the cart. We experimented with critical angles between 45° and 90°, and goal defining angles betweem 0° and 5°.

From Figure 5.20 we can see that in some cases the control force is not defined in a crisp way; in states 2 and 8 te heuristics say which action should not be applied, leaving a set of actions (e.g $\overline{-F}=\{0,+F\}$) to be choosed from. The heuristics actualy defines that an "oposite" action should be undoubthely applied only in two cases, when $\mathrm{sign}\theta \neq \mathrm{sign}\ \omega$.

The associated, heuristicaly drawn emotional graph which explains the control heuristics is given on Figure 5.22.

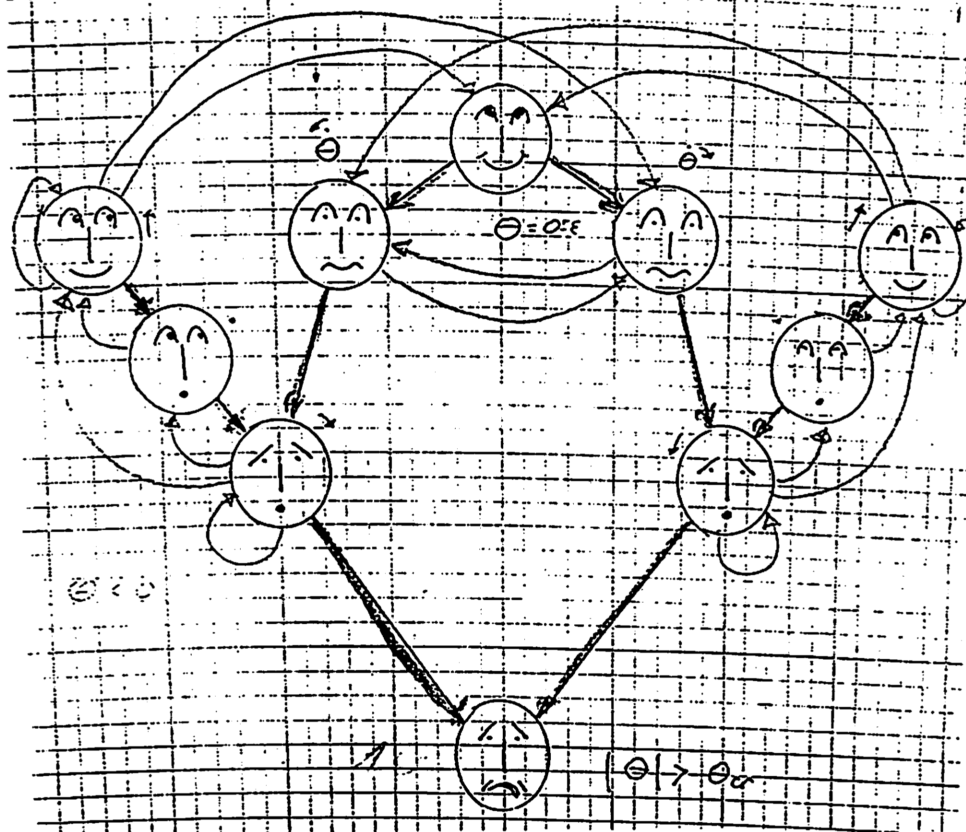


Figure 5.22. Emotional graph for context-free pole balancing problem

Figure 5.22 represents the states using different emotions. However, it is only for purposes of analysis. For the purpose of implementation, because we use CAA network, we used only three values, -1, 0, and +1. In some experiments we used only the -1 value, assigned to the failure state. Thus, we have an *avoidance learning task*, where the CAA network learns to avoid the failure state.

It is important to observe on Figure 5.22 that it represents a *loosely defined graph* [Bozinovski and Anderson 1983]. Under this notion we mean a *non-deterministic graph*, transitions in which are not defined in a crisp way. For example, being in a state ($\theta>0$, $\omega>0$) and taking action $F>0$, it is not known how the environment will respond. If there is a probability distribution assigned to the possible responses, then the problem is in stochastic nature, and we talk of learning in stochastic

environments. However, in this case *we have stochastic environment with unknown probabilities.*

The non-deterministic nature of the transition is emphasized on Figure 5.23. It is shown how an action is taken and how that action can make transition to several states, but in only one at a time. It uses divergent arrows to represent the non-deterministic transitions. For example, if the system is in state 9, and the action +F is taken, the transition cannot be stated with certainty.

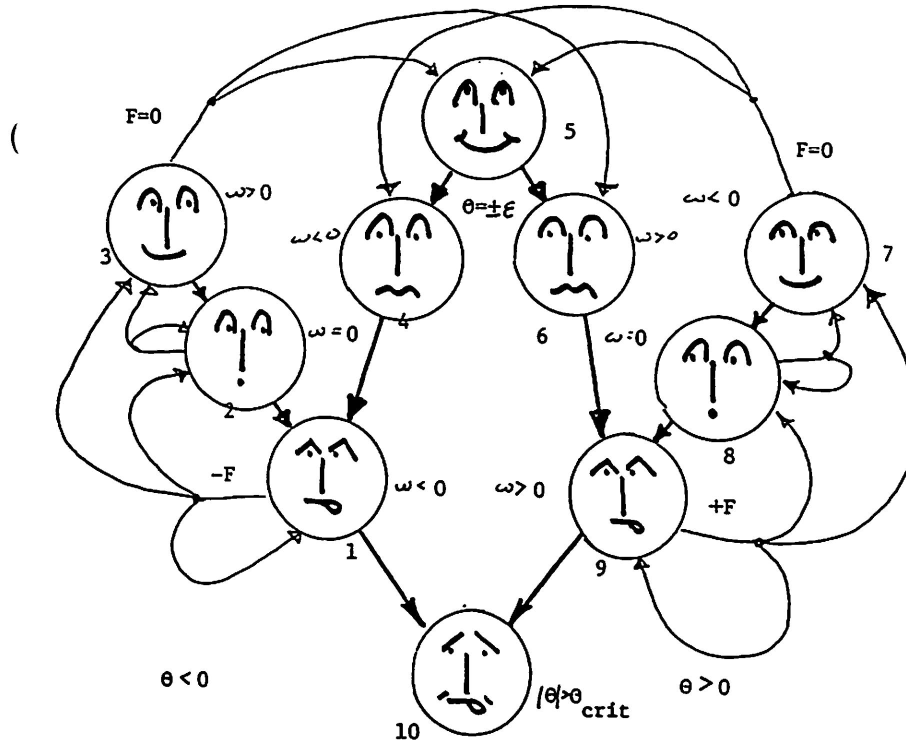


Figure 5.23. Using divergent arrows to represent a non-deterministic transitions in a losely defined emotional graph

Analysis of the problem shows that the problem can be represented even with fewer states. Figure 5.24 shows that the analysis of the considered inverted pendulum system can be carried out in terms of functions *sign(θ)* and *sign(ω)* instead of θ and ω. The goal states of the system can be defined as states satisfying $sign(\theta)sign(\omega) \leq 0$. The states in which a controler action is required are states satisfying $sign(\theta)sign(\omega) > 0$. It is also to notice that although 6 states are shown, the problem can be actualy represented with 4 states. Those are: state 1: $sign(\theta)sign(\omega)>0$, $\theta<0$; state 2: $sign(\theta)sign(\omega)<0$; state 3: $sign(\theta)sign(\omega)>0$, $\theta>0$, and state 4: $|\theta|>\theta_{crit}$. For such a representation the heuristic control policy is straight-forward and is also shown in Figure 5.24.

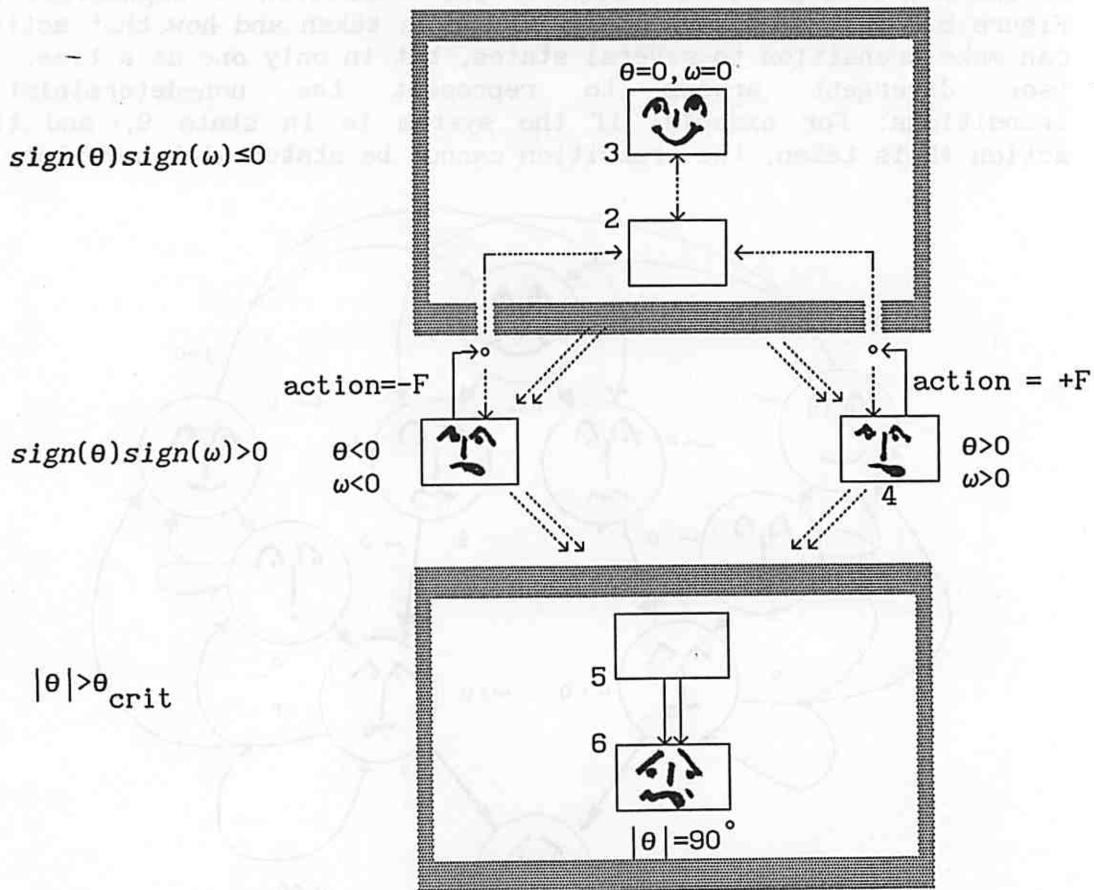


Figure 5.24. Representing the problem with losely defined graph with four states

Figure 5.24 shows that the goal of the system is relaxed: it is not to reach the state 3: ($\theta=0, \omega=0$) since it is very difficult; it is enough to state the goal as state 2: $sign(\theta) \neq sign(\omega)$. The union of those two region gives the state 2∪3: $sign(\theta)sign(\omega) \leq 0$ which is enough to develop a tendency of keeping the pole in an upright position.

5.6.2.2. Dynamics of the cart-pole system

For purpose of simulation, the cart-pole system can be described by following differential equations (adopted from [Cannon 1967]):

$$\dot{\omega} = \frac{f + \mu_c \mathrm{sign}v + m\ell\omega^2 \sin\theta - M(mg\ell\sin\theta - \mu\omega)/(m\ell\cos\theta)}{\ell(m\cos\theta - 4M/(3\cos\theta))} \quad (5.33)$$

$$\dot{v} = \frac{f + \mu_c \mathrm{sign}v + m\ell\omega^2 \sin\theta - (3/4)(mg\sin\theta - \mu\omega/\ell)\cos\theta}{M - (3/4)m\cos^2\theta} \quad (5.34)$$

where M is *the* total mass of the system (cart+pole), m is mass of *the* pole, g is acceleration due to gravity, ℓ is distance between pivot and pole's center of mass, $\dot{\omega}$ is angular acceleration of the pole, $\dot{v}$ is linear acceleration of the cart, and F is the applied force generated by the controller. The friction coefficients between the cart and the ground μ_c, and pole and pivot, μ, in some simulations can be ommited. However, in our simulations they were taken into account.

A phase plane, $(\theta-\omega)$, for the was drawn to observe the system dynamics. Figure 5.25 shows such a plane.

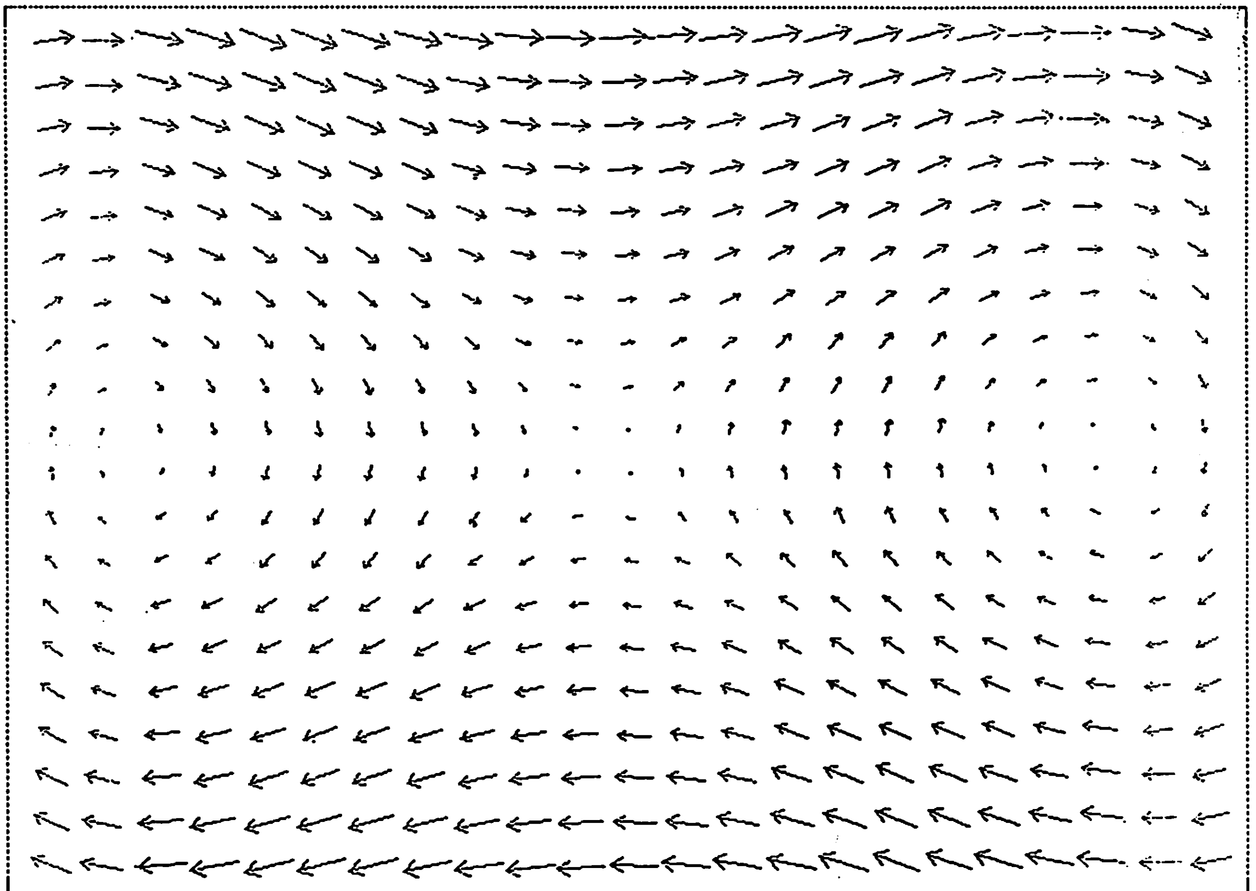


Figure 5.25. A phase plane, $(\theta-\omega)$, for pendulum dynamics (Figure by Anderson, 1981)

Horizontal axis on Figure 5.25 shows θ and vertical shows ω. As result we obtain the well-known spiral dynamics of the system, described in textbooks of dynamic systems, (e.g. Cannon 1967).

Let us note that the complete dynamics of the cart-pole system during our experiments with the CAA as pole balancing controller, was responsibility of Chuck Anderson (Bozinovski 1981g, Bozinovski and Anderson 1983).

Having performed emotional-graph analysis and dynamics of the cart-pole balancing problem we can now describe the experimental setup for self-learning control in non-deterministic envronments using the CAA controller.

5.6.2.3. *Parallel programming for pole-balancing learning*

The basic setup for our experimental investigation is given on Figure 5.26.

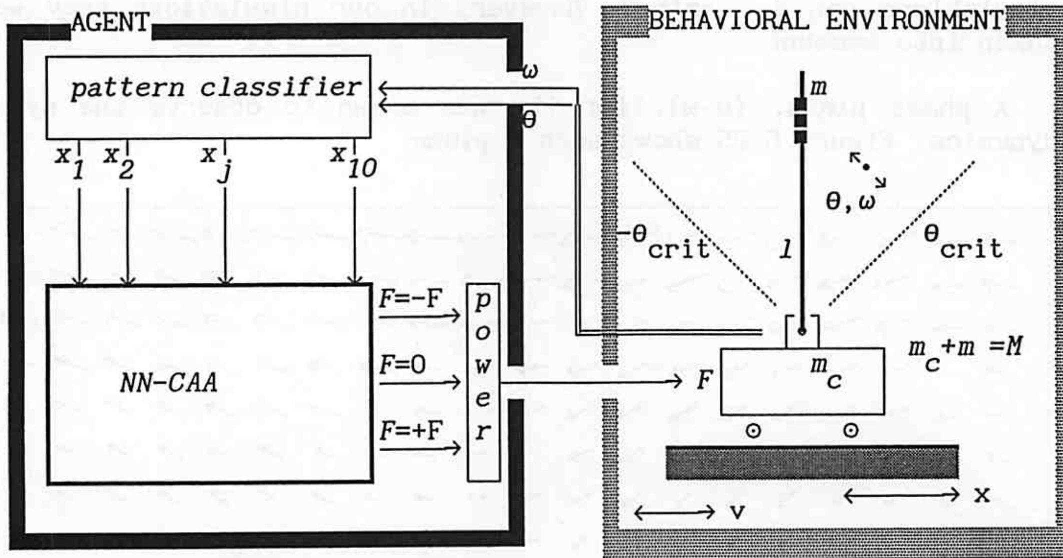


Figure 5.26. The basic setup for CAA pole balancing

Figure 5.26. shows that we chosed to control the cart-pole system by three actions: -F, +F, and 0, which corresponds to movements left, right and stay. Let us note that, to the best of our knowledge, the learning controlers considered before (Widrow and Smith, 1964, Michie and Chambers, 1968, and Russel and Rees, 1975) used only two actions, left and right. The acton "stay" allows the system not to move if it is not neccessary: for example, if $\theta>0$ but $\omega<0$, than probably the best decision is not to act, since things are improving anyway. The introduction of the third action greatly simplifies the representation of the problem, as shown previously, in Figure 5.24, and we believe gives better experimental setup for the pole balancing problem.

The interaction between the CAA controler and the pendulum dynamics was made using the parallel programming technique. Figure 5.27 describes the parallel programming concept implemented in our experiments.

```
CAA CONTROLLER
create mailbox Action, Situation
initialisation
        m = 10
        W(m)=(-1)
        chose backpropdepth
restart:
        force = 0
main loop
loop:   write Action,force










        read Situation,pole
        state = recognize(pole)
        evaluate state
        if endtrial then restart
        update W
        compute force
        goto loop
```

```
CART-POLE DYNAMICS
share mailbox Action, Situation
initialisation



restart:
        θ=θo; ω=ωo; x=xo; v=vo

read Action,force
display animation(θ,ω,x,v,force)
compute
    ω̇           ; v̇
    ω = ω + ω̇ Δt; v = v + v̇ Δt
    θ = θ + ω Δt; x = x + v Δt
pole = (θ,ω)
write Situation,pole
```

Figure 5.27. A parallel programming concept used in our 1981 experimental work

As Figure 5.27 shows, we used mailboxes as communication devices. We succeeded to realize the parallel programming interaction, on which we talked in the third chapter, where we conceptualized parallel teacher-learner interaction. The technique of creating and using mailboxes was shown to us by Bob Heller (Appendix C).

Figure 5.27 does not need a special comment except for one parameter, the *backpropdepth*. This parameter is used to determine how far from the terminal state the state value will be backpropagated. This is a crucial parameter which introduces the solution method for undeterministic, stochastic, environments we used for solving this task. We will describe the parameter shortly, after we describe results of our experiments.

5.6.2.4. Some results of our experiments

Using the described experimental setup we carried out several successfull experiments of self-learning for pole balancing. Figure 5.28 shows result of an experiment.

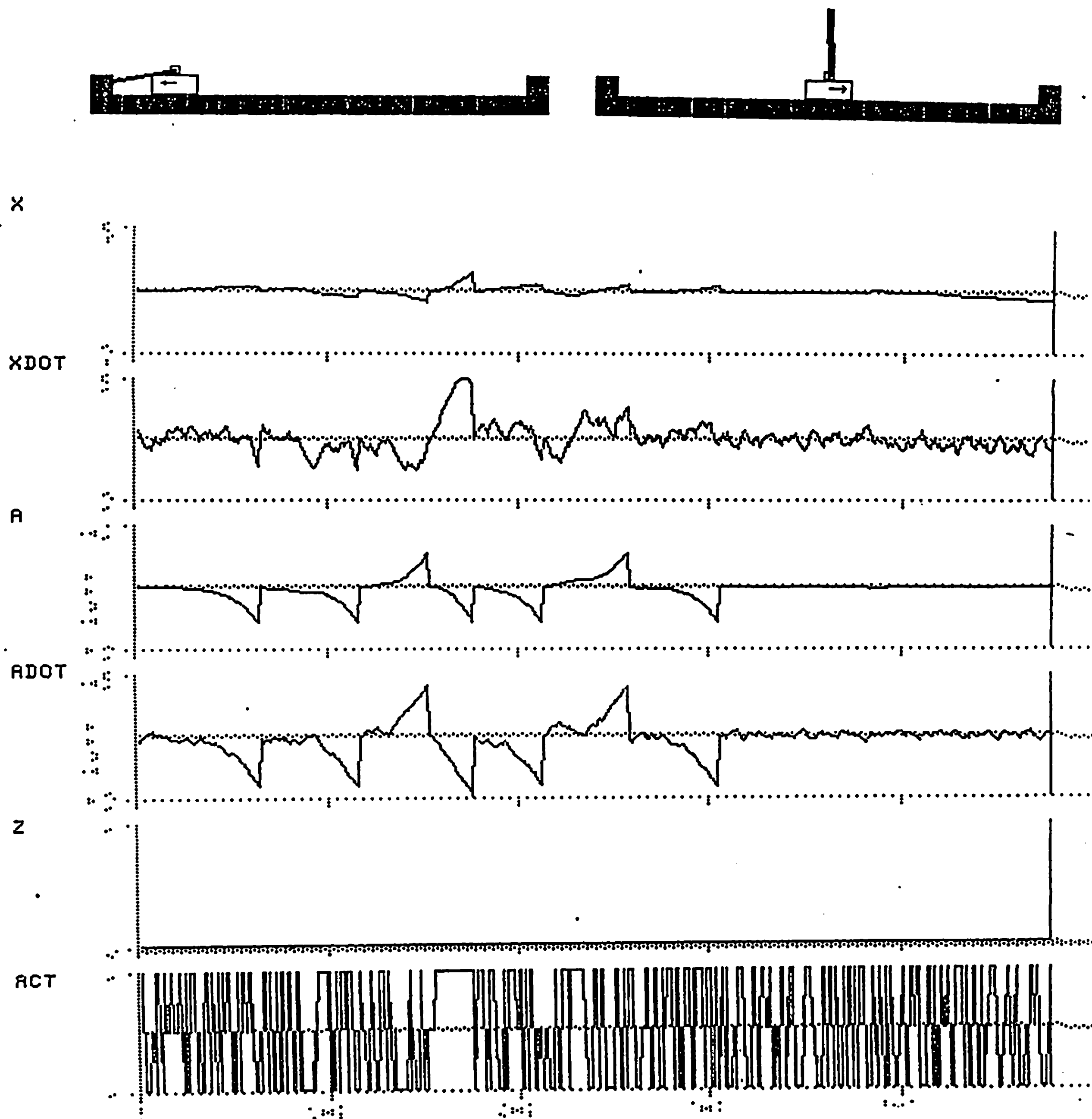


Figure 5.28. A learning control experiment with initial conditions $(\theta,\omega,x,v)=(0,0,0,0)$ (Animation program written by Chuck Anderson, time diagrams program written by Alan Morse)

The time diagrams on Figure 5.28 shows one of our early experiments. Two screens are used, one for pendulum animation (shown on the upper section) and other for time diagrams display (shown on the lower section). All the four componets (X,XDOT,A,ADOT=x,v,θ,ω) of the cart-pole system are displayed. The actions are displayed at

the bottom of the screen. The Grinell Systems for graphic display was used. The learning control process is observed on the A and ADOT trace. The controller has experienced two times left side fall ($\theta = -90^\circ$) and 5 times failure at the right side ($\theta = +90^\circ$) before it learns how to balance the pole. After that, both θ and ω are kept around their zero values. Initial conditions in this experiment were all zeroes.

The experiment shown on Figure 5.28, and in Appendix C were the first experiments proving that the CAA is able to learn a control strategy in non-deterministic environments. The actual strategy, learned by CAA is a *bang-bang* strategy which can be described as

if sign(θ)sign(ω)>0 then
if θ>0 then go right
if θ<0 then go left.
otherwise stay.

According to the experiment shown on Figure 5.28, and in the Appendix C, the CAA learns this strategy rather quickly. However, the strategy has not been told to the CAA. *CAA discovered it by itself.* The penalties received as results of failures, caused beckpropagation of values which are preventing actions which caused failure. Once that has been learned, the CAA system is balancing the pendulum.

In the previous mentioned experiments we used zero initial conditions on all parameters. It was also interesting to explore possibility of CAA to balance the pole with other initial conditions. Particularly, in initial condition for θ is outside the zero region, for example at 10°. Figure 5.29 shows a result of such an experiment.

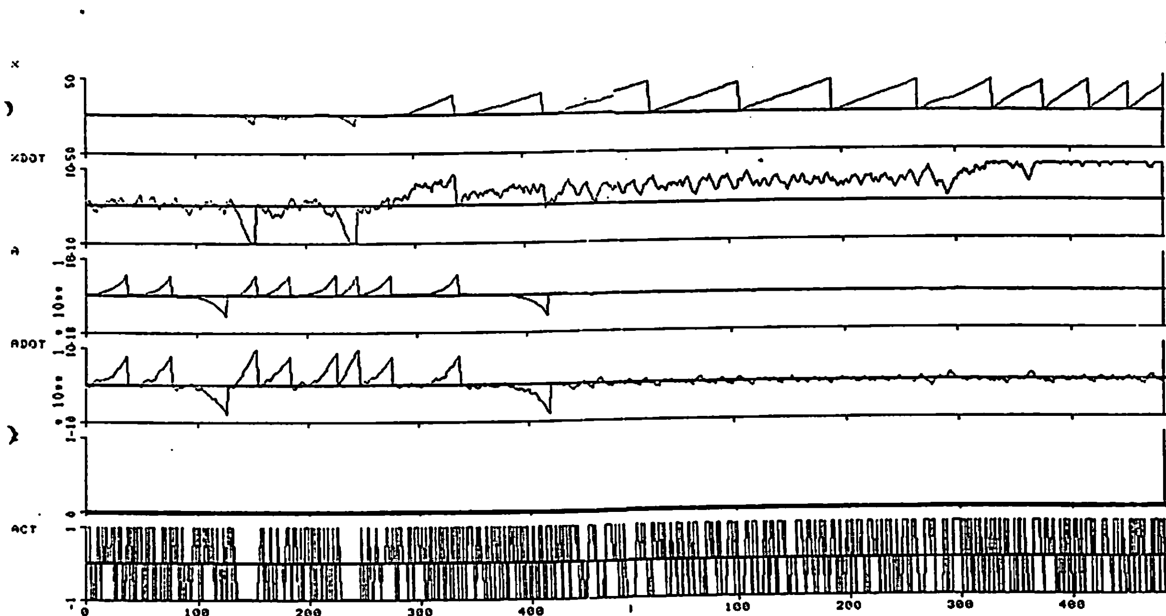


Figure 5.29. A learning-for-pole-balancing experiment with initial conditions (θ,ω,x,v) = (10,0,0,0)

Figure 5.29 shows two screens from the mentioned Grinell system, concatenated together. Each screen shows 500 steps of learning history of the CAA controller. Two things should be pointed: it takes 8 times right failure and 2 times left failure to learn the control strategy. After about 450 trial and error learning, the pole is balanced around its $\theta=0$, and $\omega=0$ values. However, the cart is traveling to the right, since there is no restriction from the context of x.

This experiment rises a question of context restrictions. If we restrict movement of the pole around zero for $\pm 12^{\circ}$, (as was done before by Michie and Chambers (1968), then we can restrict also x-discourse to be, for example, ±0.54m (Michie and Chambers 1968). But in that case it is very difficult or impossible to start a successful experiment at $\theta=+10^{\circ}$, as we experimented with. As Russel and Rees (1975) reported, they were able to recover from nonzero initial conditions in the initial discourse if θ was less than $\pm 7^{\circ}$. So, as we showed, it is possible to learn to control for greater initial angles, if we chose the unrestricted case of cart movement.

That all shows, that our study of the inverted pendulum learning control, although considering context-unrestricted case, is highly relevant for issues around the self-reinforcement learning systems. Some authors, for example Dean and Wellman (1991) also use the context-unrestricted case. The point is to show that a CAA system can learn to control in a stochastic environments with unknown probabilities, such as pole balancing problem is.

5.6.2.5. The limited value-backpropagation method for solving the problem of learning in loosely defined emotional graphs

The problem of learning in loosely defined emotional graph which we considered in our experiments of balancing inverted pendulum can be stated as learning in stochastic environments problem, which is a issue in conteporary reinforcement learning theory. Here we will describe our approach, which produced successfull solutions, as shown in the preceeding subsection.

Since the loosely defined graph is a graph with non-deterministic transitions, then some learned action which is good now can be wrong later, and in turn, there is no point of learning the whole path from the starting state to the goal state. Instead, the system should learn *partial segments* of the path. That is the main idea of our approach.

In other words, the value backpropoagation process should not go from some terminal state to a starting state, as we did in *well defined emotional graphs* described in the previous section. Instead, the state value should be backpropagated only few steps back. In our experiments with the pole balancing we have chosen only one step back. Figure 5.30 explains the approach:

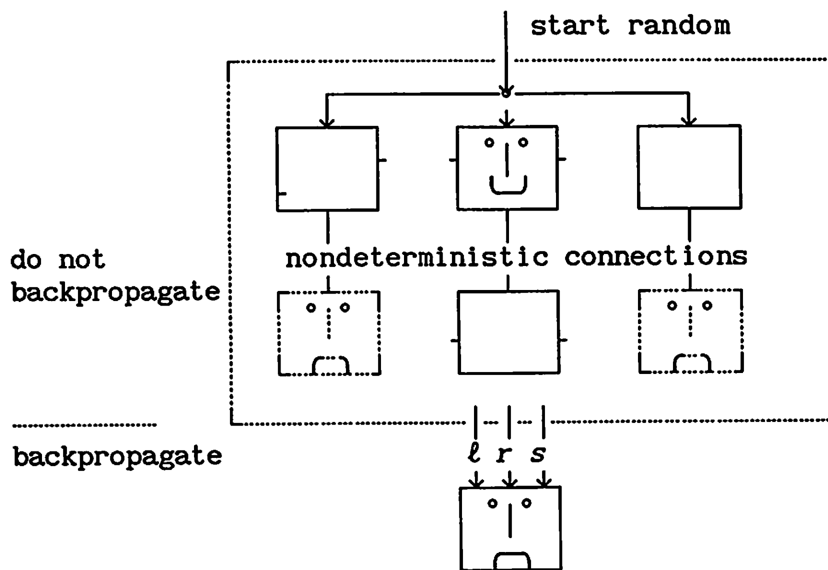


Figure 5.30. *Selective value backpropagation process as a solution method in stochastic environments, used in learning for pole balancing experiments*

Figure 5.30 shows a view toward the pole balancing problem. There is a situations which is recognizable as failure situation. Three possible actions are to be blaimed for possible failure, from whichever previous situation they are taken. So, a mechanism should be found according to which

whenewer a failure occurs,
the CAA is to search for better action.

As long as the CAA is in the game, i.e. is somewhere in the state space, no learning is needed. The state value will be backpropagated one situation behind, so the preceeding state should know not to go in certain direction. But, the state value backpropagation will stop there. Although some states inside the losely defined graph are colored "bad", the knowledge about that is not backpropagated, and the previous states can allow any actions. So, the CAA system will pass through the desirable and undesirable states but will avoid the final, terminal state.

The Figure 5.31 shows the learning in the CAA SAE components in an experimental run, in fact the one shown in Figure 5.29 above, where we have nonzero initial conditions. As we can see the CAA starts with genetic knowledge saying that all the SAE components for the state 10 are -1, meaning that state is undesirable. Consider the states 1 and 9. The state 1 is defined by $(\theta<0, \omega<0)$, and the state 9 is defined by $(\theta>0, \omega>0)$. We restrict the state value backpropagation process to one state back only, and we have the evolution of CAA knowledge as shown in Figure 5.31.

run #	number of steps in the run	states	1	2	3	4	5	6	7	8	9	10
0		genetic	0	0	0	0	0	0	0	0	0	-1
		knowledge	0	0	0	0	0	0	0	0	0	-1
			0	0	0	0	0	0	0	0	0	-1
		new	0	0	0	0	0	0	0	0	0	-1
1	35	knowledge	0	0	0	0	0	0	0	0	-1	-1
			0	0	0	0	0	0	0	0	0	-1
		new	0	0	0	0	0	0	0	0	0	-1
2	40	knowledge	0	0	0	0	0	0	0	0	-1	-1
			0	0	0	0	0	0	0	0	-1	-1
		new	0	0	0	0	0	0	0	0	0	-1
3	51	knowledge	-1	0	0	0	0	0	0	0	-1	-1
			0	0	0	0	0	0	0	0	-1	-1
		new	0	0	0	0	0	0	0	0	-1	-1
4	27	knowledge	-1	0	0	0	0	0	0	0	-1	-1
			0	0	0	0	0	0	0	0	-1	-1
		new	0	0	0	0	0	0	0	0	-1	-1
5	30	knowledge	-1	0	0	0	0	0	0	0	-2	-1
			0	0	0	0	0	0	0	0	-1	-1
		new	0	0	0	0	0	0	0	0	-1	-1
6	41	knowledge	-1	0	0	0	0	0	0	0	-2	-1
			0	0	0	0	0	0	0	0	-2	-1
		new	0	0	0	0	0	0	0	0	-2	-1
7	20	knowledge	-1	0	0	0	0	0	0	0	-2	-1
			0	0	0	0	0	0	0	0	-2	-1
		new	0	0	0	0	0	0	0	0	-2	-1
8	30	knowledge	-1	0	0	0	0	0	0	0	-3	-1
			0	0	0	0	0	0	0	0	-2	-1
		new	0	0	0	0	0	0	0	0	-3	-1
9	63	knowledge	-1	0	0	0	0	0	0	0	-3	-1
			0	0	0	0	0	0	0	0	-2	-1
		new	0	0	0	0	0	0	0	0	-3	-1
10	81	knowledge	-1	0	0	0	0	0	0	0	-3	-1
			-1	0	0	0	0	0	0	0	-2	-1
		solution	0	0	0	0	0	0	0	0	-3	-1
11	> 562	knowledge	-1	0	0	0	0	0	0	0	-3	-1
			-1	0	0	0	0	0	0	0	-2	-1

Figure 5.31. Experiment in pole balancing: the learning process

As Figure 5.31 shows, the CAA is searching for policy : "do LEFT in state 1 and do RIGHT in state 9". In mathematical terms, it is searching for solution in the form

$$-W_{LEFT,1} = \max_a \{W_{a,1}\}$$
$$W_{RIGHT,9} = \max_a \{W_{a,9}\} \qquad (5.35)$$

where $W_{a,j}$ is the SAE component (weight) of the CAA matrix.

And it finds that solution after 10 trials.

From the Figure 5.31 it is clear how the CAA method for learning in stochastic environment works. Only the SAE components of the states leading to the terminal states are updated according to the CAA learning method. In other words, only the vicinity around the terminal states are learned, all other space is just wandered randomly. In the pole balancing problem experiments which we carried out in 1981, it was shown that is enough to learn that the terminal states should be avoided.

Let us note that the pole balancing solution using the CAA controller withn the ANW group was done before the AC solution. In fact, when the experiments described above were performed, the development of the code for the AC controler has not been started yet. Together with Chuck Anderson, under supervision of Andy barto, we produced the above result. We must admit that we did not know the theory how it was done. Now it is easy to explain. Then it was only important that it works, *somehow*.

In short, the limited value-backpropagation method which we used, proved successfully for the pole balancing problem. We believe that is can used also in other problems of learning in stochastic environments.

5.7. ANOTHER EXAMPLE OF A CAA ARCHITECTURE

In section 5.5. we described in detail an example of our generic CAA architecture. We used the described architecture in our experimental work described in the previous, 5.6, section. Now we will describe another example of the CAA architecture which we considered but which we did not experimented with in 1981. Figure 5.32 depicts the architecture.

The CAA architecture on Figure 5.32 [Bozinovski 1981b] takes advantage of computing supremum on each input and output. If the numbers which CAA deals with are real numbers, or from a totaly ordered set, then supremum reduces to maximum. So we have CAA architecture with controlled maximum computation on each input (situation and teacher input) and each output (action and state-value output). The controling signals c1 through c4 decide whether the maximum selector will multiplex the *value* of the maximal components of only its existence in form of binary value 1 and 0.

The input maximum computers are used to orthonormize input signals. The action output computes the existence of maximum of sums, and the emotion output computes value of the maximum of SAE components in a single column, since input vectors are orthonormized. So the main difference between the CAA considered before and the one shown on Figure 5.32 is the state-value computation.

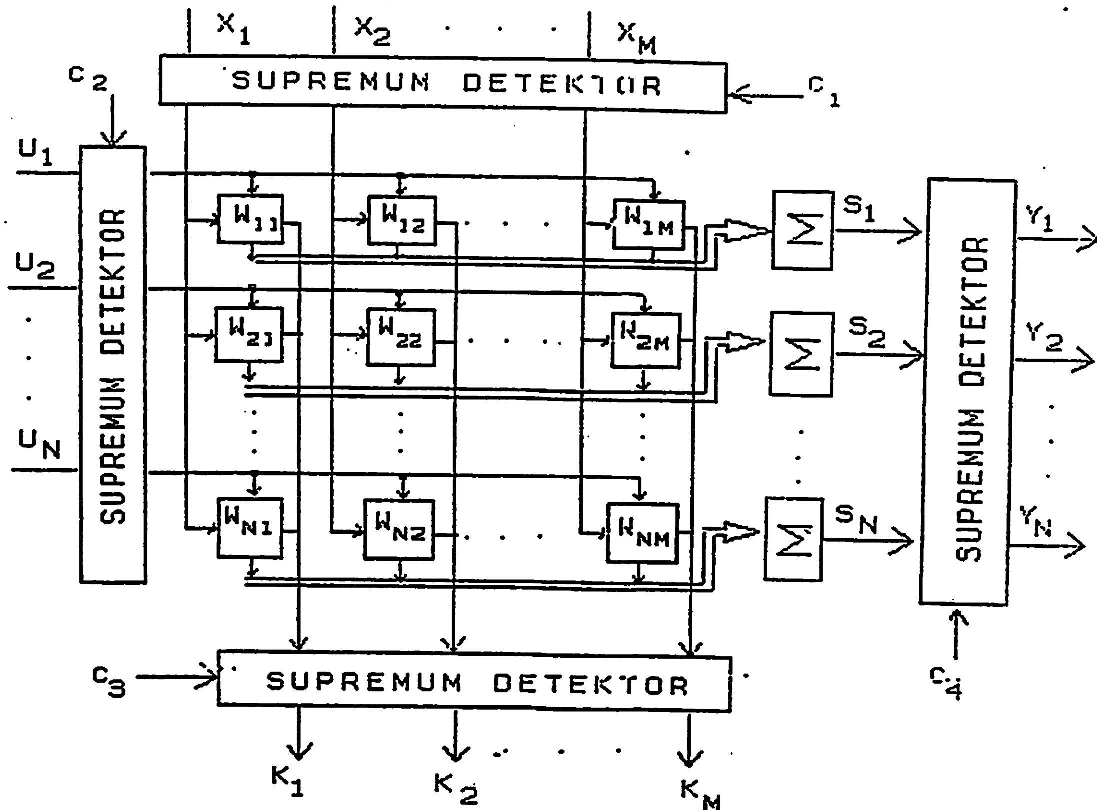


Figure 5.32. The greedy-emotion CAA architecture
(Figure from Bozinovski 1981;
Drawing software written by Rich Sutton)

The state-value in this architecture is computed "in greedy fashion",

$$v_k = \max_a\{W_{ak}\} \quad , \qquad a = 1,..,n \qquad (5.36)$$

We will not talk here about this 1981 version (we can name it *greedy-emotion CAA)* of the CAA. But, it will be of interest in the next chapter where we compare our CAA with some later proposed, dynamic programming motivated learning methods.

5.8. USING ENTROPY IN MARKOV DECISION PROCESSES

In this section we will discuss the usage of entropy as a measure of learning, which we have done in our experiments with the CAA architecture.

In optimization processes it is convenient to use an optimization function which will have a physical meaning. Such a function makes the optimization process more intuitively clear. In neural network resarch, the energy function has become popular with the work on Hopfield networks (Hopfield 1982) and Boltzman machines (Hinton, et al. 1984, 1986). However, we do not believe that the learning process should be explained in *terms of energy*. In our research we used the entropy function, which is not widely used in neural network resarch, but is well known in other areas, as communication theory [Shannon, 1948, 1951].

We have used the notion of entropy in connection with learning in state-valued graphs. The concept of entropy is connected with transition probabilities between the states. If we consider the state-valued graphs emphasizing transition probabilities, we actualy consider the *Markov Decision Problem* (MDP). Let us note that usual notions in the area are state space, control space, transition probabilities, transition cost, optimal cost-to-go, optimal policy, Bellman equation, value iteration, policy iteration, and discount, among others. It is interesting observation that so far the entropy seems not to be considered as a concept in MDP [Kemeny and Snell, 1960, White 1993]. In this section we describe how we used entropy in MDP.

Two cases should be considered in computing entropy in dynamic programming problems: 1) if transition probabilities are known and 2) if transition probabilities are not known. If transition probabilities are known, then we have a classical MDP where the entropy is relatively easy to define. If they are not known, then the entropy should be computed from the behaviour of the agent. We will describe the both cases and the methods we used for computing the entropy of the dynamic programming problems.

5.8.1. COMPUTING ENTROPY WITH KNOWN TRANSITION PROBABILITIES

Figure 5.33 describes a graph with known transition probabilities. It is a graph from one of our first experiments with emotional graphs. In this particular graph, only one state is desirable, and three states are undesirable. All the valued (emotionaly colored) states are absorbing ones.

The method, which we call the *reducing alternatives method* is based on the intuition that learning of where to go is actualy reducing the choice of alternatives offered at choice points, i.e. the graph states. So, to each state we assign an entropy measure, and the total entropy of the system is the sum of all state entropies. The entropy of an absorbing state is 0, since there is no alternatives in that state.

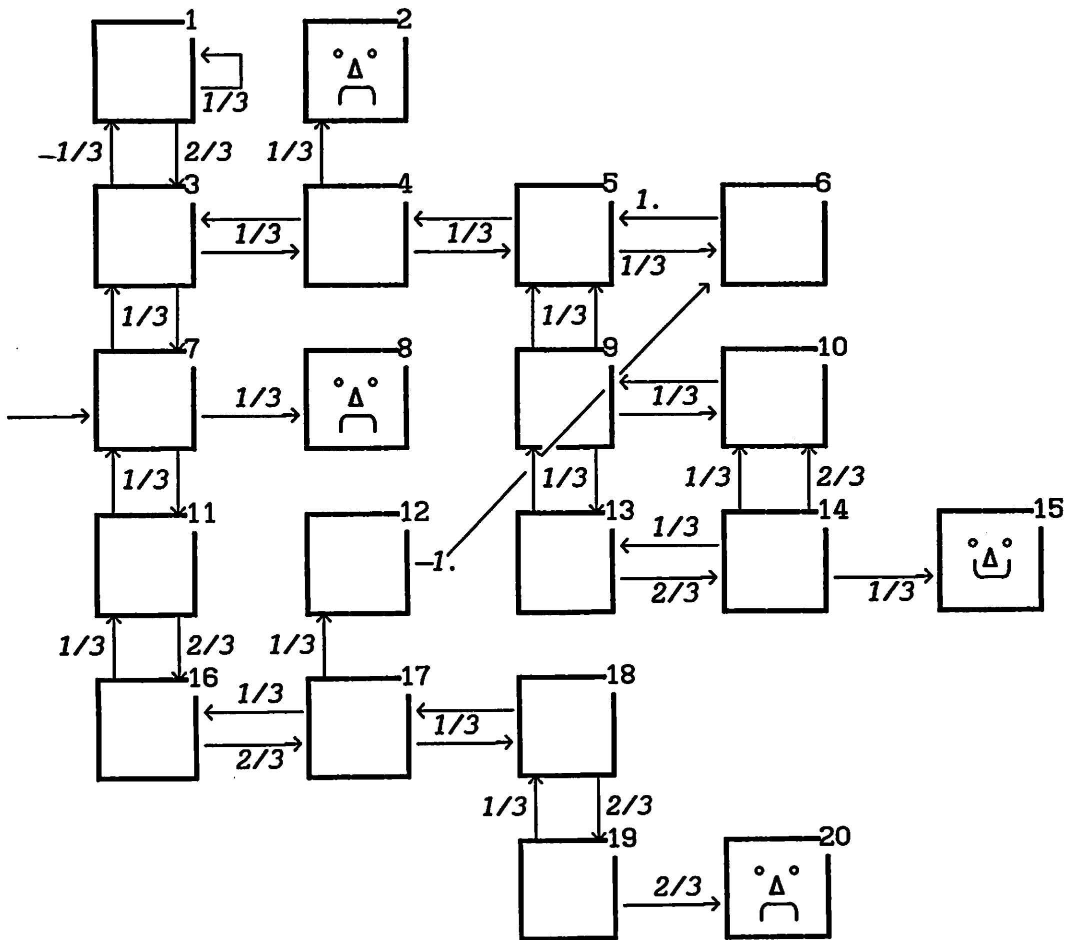


Figure 5.33. A Markov Decision Problem Graph

The entropy measure assigned to each state is computed as [Shannon 1948]:

$$H_j = -\sum_{i=1}^{N_j} p_{ij} \mathrm{ld}(p_{ij}) \tag{5.36}$$

where

H_j is the entropy of the j-th state

N_j is the number of possible actions from the j-th state

p_{ij} is the probability of taking the i-th action in the j-th state

$\mathrm{ld}(.) = \log_2(.)$

The total system entropy is

$$H = \sum_{j=1}^{m} H_j \quad . \tag{5.37}$$

In the example above the total initial entropy is

$$H(t=0) = -(H_1(t=0) + H_2(t=0)+ \ldots + H_{20}(t=0)) =$$

$$= -\{[(1/3)ld(1/3)+(2/3)ld(2/3)]+ [0] + \ldots + [0]\} = \qquad (5.38)$$

$$= 17.45 \text{ bits.}$$

Now, let a CAA agent enter the graph at the starting state 7, which has initial entropy

$$H_7(t=0) = -3[(1/3)ld(1/3)] = 1.59 \text{ bits} \qquad (5.39)$$

and let it choose the action which will lead it to the state 8 which it realizes is undesirable. Due to its learning ability, the CAA agent will reduce the number of alternatives from state 7: it will have two instead of three. Let the two alternatives be equiprobable. That means that in the next iteration the entropy of state 7 will be

$$H_7(t=1) = -2[(1/2)ld(1/2)] = 1 \text{ bit} \qquad (5.40)$$

and because of that the total system entropy will be

$$H(t=1) = 17.45 - .58 = 16.87 \text{ bits.} \qquad (5.41)$$

Due to a learned fact about taking action, the entropy has lowered. Each iteration the entropy will be lower and lower, having its minimum when the CAA agent has learned a path to some goal state. The entropy is usualy not zero, since usualy neither all the states are visited nor all the actions per state are tried. If the entropy is zero, it means that a total action taking policy, over all the states, is established.

Note that the transition probabilities *are changing* during the teaching process, with respect to the CAA agent involved in exploration learning. In each run the CAA agent is facing a *new MDP with different transition probabilities*. The intropy endeed measures the *stochasticity of the agent's policy* in an MDP.

5.8.2. *COMPUTING ENTROPY WITH UNKNOWN TRANSITION PROBABILITIES*

A more intersting case for the CAA and other learning agents in MDP environments is when the transition probabilities are not known. It includes situations when we have *nondeterministic environments*, like the pole balancing problem; that also includes stochastic environments, when the environment responds with a new situation in a probabilistic manner.

If the transition probabilities are unkknown, we use a method of computing the entropy from the *behavior strings,* as computing the entropy from strings of some language [Shannon 1951]. The strings are computed during the experiments with CAA agent, solving the real-time state-valued graph traversing. Here we will describe an our experiment where behavioral strings are obtained and the entropy is computed.

Figure 5.34 shows a graph used in the experiment. The valued nodes are represented by uppercase letters and the neutral with lowercase. The same idea is used as in our work with teaching languages, as we talked about in the third chapter.

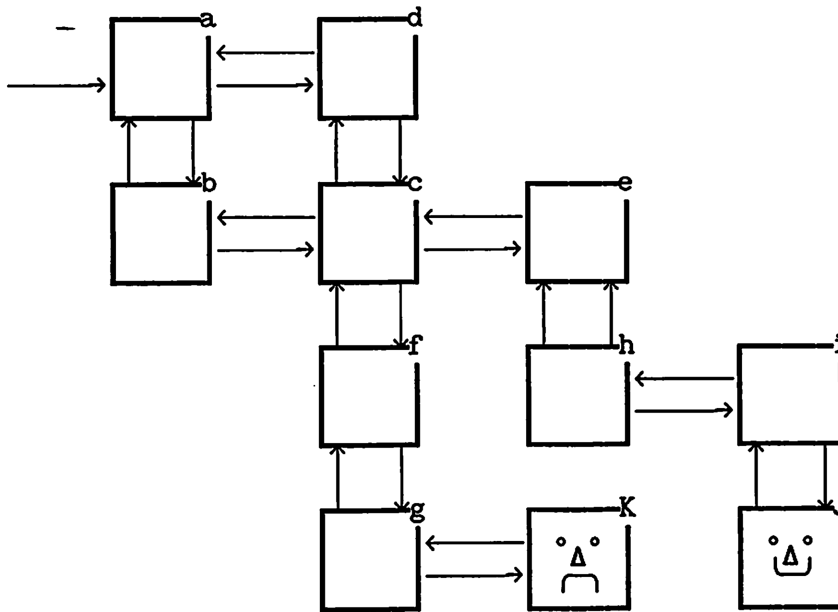


Figure 5.34. An experiment with CAA in a graph traversing problem with unknown transition probabilities

The table 5.1. shows a series of learning trials in a CAA learning experiment.

Table 5.1. Behavioral strings obtained in a CAA learning experiment in the graph shown in Figure 5.34

trial	behavioral sequence
1.	adadcececfcecehecfgfcecfgK-
2.	abababadadcfgfcfcfcehececececfcehececehiJ+
3.	abcececehecehecfgfcehecececehIJ+
4.	abadcfgfgfcfceHIJ+
5.	adadabadcfgfgfcEHIJ+
6.	abCEHIJ+
7.	adadaBCEHIJ+
8.	ABCEHIJ+
9.	ABCEHIJ+
10.	ABCEHIJ+

Having these strings various measures can be used for observing the learning process. A convienent measure is the first order Markov chain entropy

$$HM = \sum_{jk}^{mm} p(x_j)p(x_k/x_j)ld(p(x_k/x_j) \qquad (5.42)$$

where $p(x_j)$ is the probability of appearence of the j-th state and $p(x_k/x_j)$ is the the conditional probability of appearence of the k-th state after the i-th state in the behavioral string.

Figure 5.35 shows the points of the entropy curve obtained for the experiment shown on the Table 5.1.

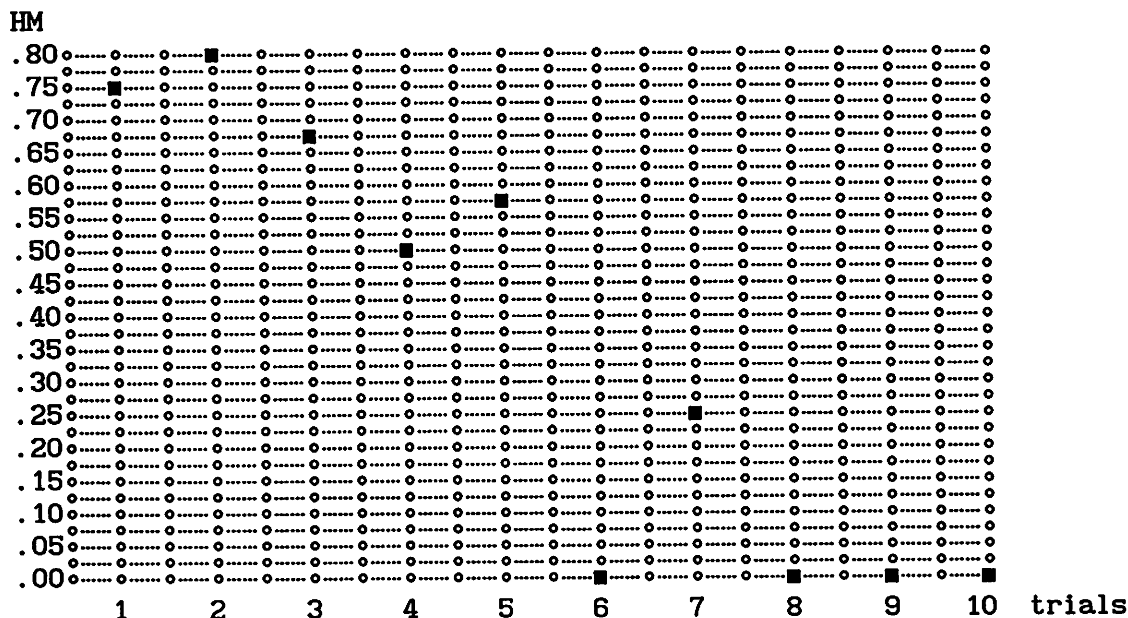


Figure 5.35. An entropy curve as a learning curve

Let us note that the table 5.1. gives motivation for other definition of learning curves. For example the table 5.1. can be viewed as a (horizontaly shown) histogram curve. Such a curve converges toward a number given by the length of the learned path. Also, the ratio of the lowercase letters and uppercase letters can be a learning curve. However, a learning curve is terms of entropy is convenient for its broad sense interpretation, as describing a process which organizes itself. The smaller the entropy, the more organization is in the system. If the system is undergoing the process of behavioral self-organization then the entropy is a measure of progress of that process.

Let us also mention that the entropy (as well as energy) is an optimization measure, rather then a solution measure. It does not say by itself whether a solution of a problem is found: it just says that the process *is converging to some behavior*. We should know from some other assumptions that if the process converges, then it converges to a desired (expected) behavior. So, we have to have assumptions or knowledge about the convergence issues of learning processes when we use entropy as a measure of learning. The issue of convergence will be discussed later.

5.9. CAA AS A NEURO-GENETIC AGENT: SOME ISSUES ON THE GENETIC ENVIRONMENT

The genetic environment is an important concept in this project and_a salient feature of the CAA architecture. Here we will discuss four issues concerning the genetic environment: 1) having different species due to the different species vectors 2) the export of the behavioral environment model to the genetic environment, 3) optimization processes in the CAA using the genetic environment concept, and 4) distinction from the genetic algorithms.

5.9.1. SPECIES VECTORS AND SUBJECTIVE GRAPHS

Since the CAA system does not receive external primary reinforcements, the internal primary reinforcements are defined by means of initial values of the SAE-components in the matrix W. The initial values are defined as species vectors [Bozinovski 82b] of the system. The initial values will affect: 1) column vectors: initial internal (or emotional) preference toward some environment states, and 2) row vectors: being in some state state, preference toward some actions eligible in that state. Figure 5.36 gives an example.

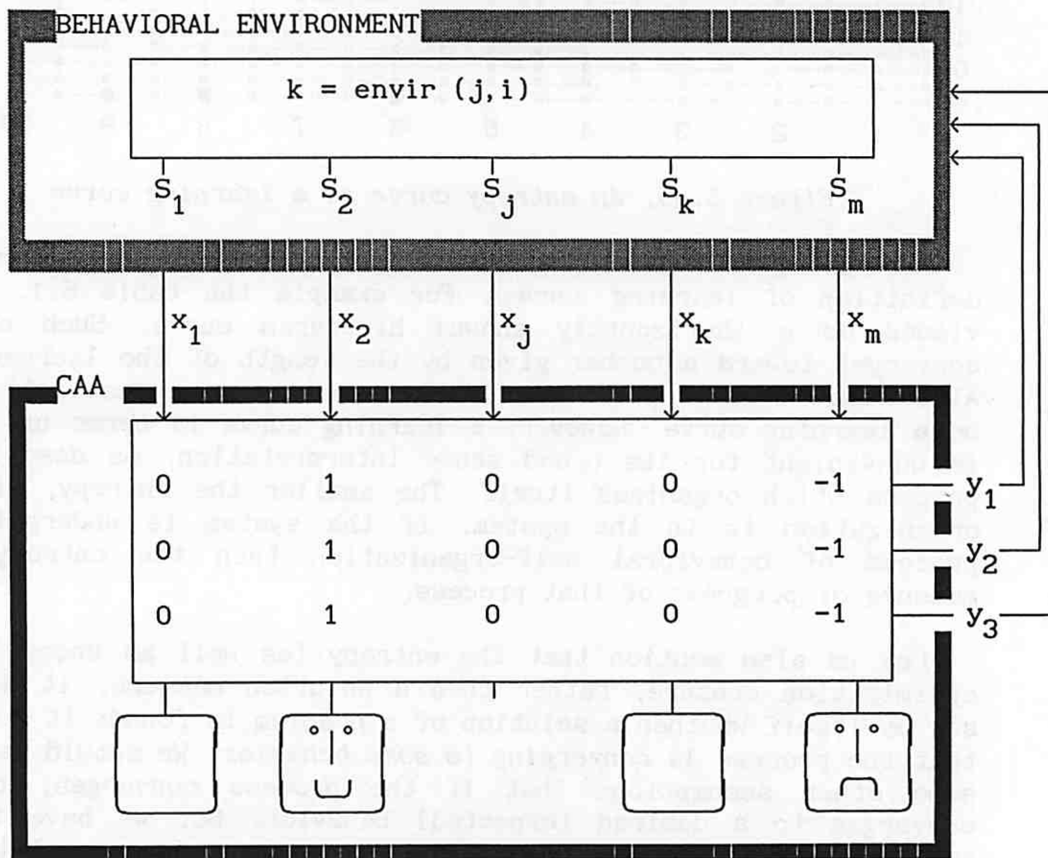


Figure 5.36. The initial definition of the primary reinforcing (goal) states

Figure 5.36 shows a case where initial values are chosen such that the second situation (which represents the second environment state) is denoted as desirable situation, whereas the m-th situation is undesirable. The desirable situation is a goal situation whereas

the undesirable one should be avoided. From the Figure 5.36 we also see that only the columns are affected, so there is no preference toward some action. However, we can imagine that from the genetic environment could come an arbitrary genome, which is a result of a previous learning taken place in some other CAA systems.

In our experiments we used species vectors only of the type shown in Figure 5.36: all the column vectors having equal SAE components within themselves. Thus a goal states in the environment are defined

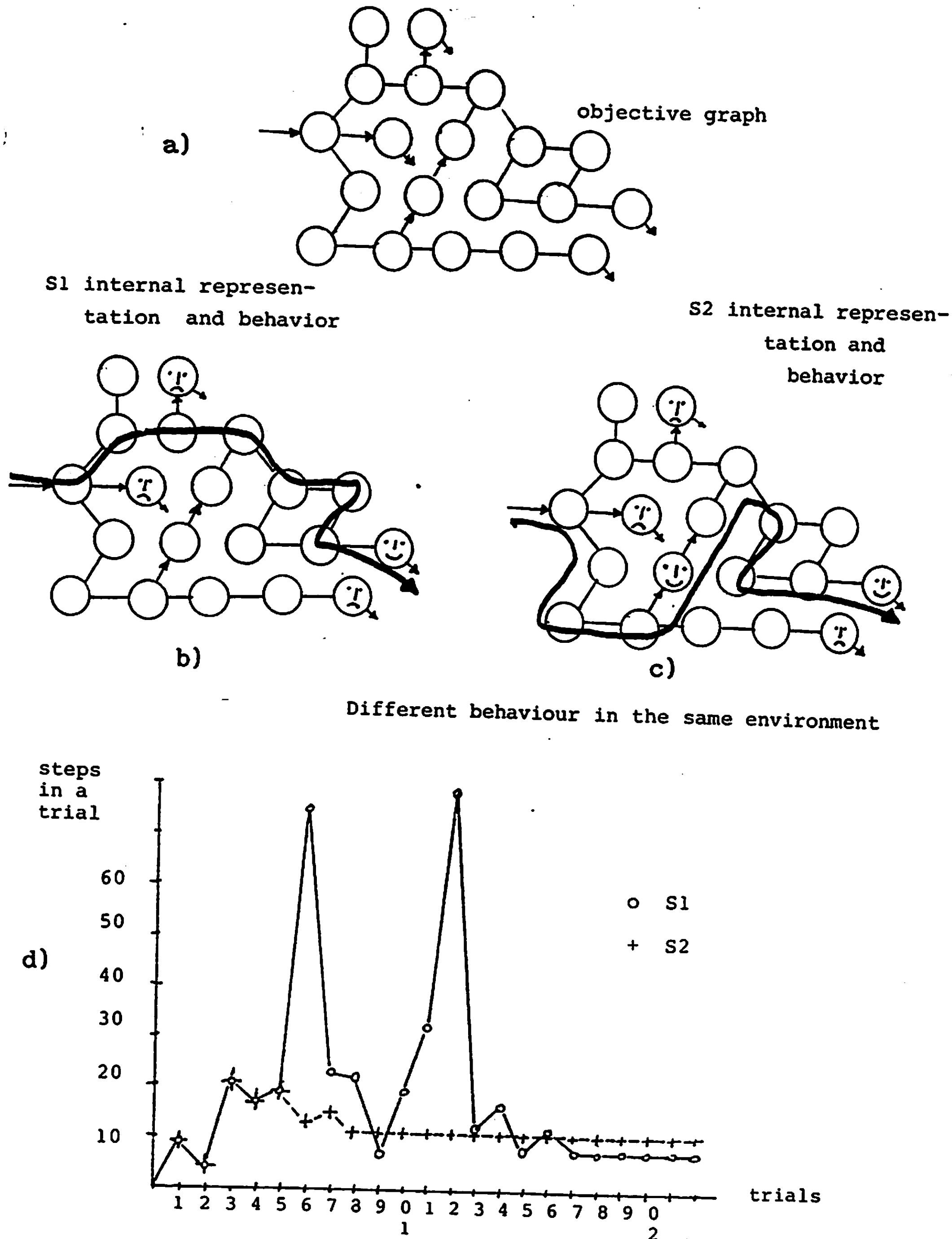


Figure 5.37. Having different species in the same environment.
a) objective environment b) the same environment represented in the species S1, and the solution found, c) the same environment represented in the species S2 and the solution found d) the learning curves

with all the components being 1, and the states which should be avoided with all the SAE components with values -1. No preference toward the initial actions are given; the actions are initialy randomly chosen.

To ilustrate a usage of the concept of species vector, let us describe an experiment. Figure 5.37 gives the illustration.

Figure 5.37 shows an objective graph (a) and two subjective representations of the same graph due to the species vectors imported in the two CAA systems, S1 and S2. It is a typical dynamic programming problem stated for the CAA "species".

Searching for a solution, the species will exhibit different behavior in the same environment. The species S1 is learning to go toward a goal state avoiding the unpleasant states. The species S2 in addition of avoiding the unpleasant states will optimize toward acquiring maximum pleasure: It will learn to pass through a nonabsorbing desirable state along the way toward a desirable absorbing state.

The point is that both the CAA architectures are the same; the only difference is the genetic knowledge *imported* from the genetic environment which makes a difference in the behavioral environment.

5.9.2. A TWO-CHROMOSOME GENOME CAA: BUILDING A MODEL OF THE BEHAVIORAL ENVIRONMENT

Within the framework of genetics, it is natural to suppose that the exporting genome can contain knowledge other then the state-action pairs, as is provided by the species vector W(0). We can assume that a CAA system can be extended to develop some other knowledge to export in the genetic environment. For example, an extended CAA system can build a model of the environment, i.e. the mapping agent-action--environment-reaction.

Figure 5.38 gives a procedure of an extended CAA system, which we denote CAT (Crossbar Adaptive Tensor) system. The procedure uses the wait/post primitives of parallel programming.

```
procedure CAT
repeat
 begin
 import chromosome vector s(0), if given.
 import chromosome vector w(0), (must be given)
 chose x(0)
 repeat
   begin
   compute action: y(t) = action(w(t),x(t))
   post y(t)
   remember previous step: (w,x,y)(t-1) = (w,x,y)(t)
   next time step: t=t+1
   receive consequence: wait x(t)
   let   i=indmax y(t-1), j=indmax x(t-1), k = indmax x(t)
   build environment model s_ij(t)= k

  evaluate  emotion:  v(t)  =  statevalue(x(t),w(t-1))
   learn consequence w_ij(t)=w_ij(t-1)+x_j(t-1)y_i(t-1)v(t)

 until endcondition
 export genome w(t), s(t)
 end
forever

Procedure environment
repeat
  receive agent action: wait y(t)
  compute next situation: x(t) = envir(x(t-1),y(t))
  post x(t)
forever
```

Figure 5.37. The CAT computational procedure

The CAT maintains two matrices: 1) for learning its behavioral policy and 2) for learning model of the environment. Both the matrices are exported as chromosomes. The CAT system does not take advantage of the environment model. It is just a transformer of knowledge to some other system which will receive that model from the exported chromosome.

5.9.3. CAA ARCHITECTURE AS AN OPTIMIZATION ARCHITECTURE

Connection to the genetic environment enables the CAA to perform optimization over the set of learned policies in the behavioral environment. The basic procedure is the following

```
CAA optimization procedure
   define a measure of goodness, J(.).
   choose initial value of goodnes, G.
do forever:
   produce new behavioral policy, π, using CAA learning method.
   measure the goodness, J(π).
   if J(π) no_worse_then G then store G ← J(π) and export π.
```

The CAA optimization procedure described above is generic; it performs search over the set of possible solutions learned by the CAA learning method. It produces solutions to a problem and measures its goodness in one variable, *G*. The solutions are generated forever, but are exported to the genetic environmet only if the generated solution is better or equal to the previous best. So at any time, the genetic environment has the best offspring the CAA was able to produce. Since the process is forever, and the CAA learning method has no restriction in finding any solution, it will eventualy export the best possible solution, and will continue to do so thereafter.

For example, if the CAA architecture is used to find the shortest path in a graph, then the measure of goodness is the length of a path, and the operation *no_worse_then* becomes the operation *no_greater_then* (or "≤").

As a variant, insted of using selection criterion "no_worse_then", CAA can use criterion "better_then". In that case, the best solution will be prodiced in one copy only, and the CAA will stop producing solutions. On the basis of that a criterion can be established so the CAA is removed from the behavioral environment. As we can see the whole CAA concept has its issues in the *artificial life* research, and the CAA can be viewed as an artificial life agent.

The measure of goodness in CAA is defined in the supervising unit. Is a part of the exporting strategy of the CAA architecture. However, we can admit that not always the measure of goodness is defined. Or even if it is defined, it is not applied for some reason. Does it make sense to talk about optimization even in that case? It seems it does, and we can imagine two processes of optimization inherent to CAA.

1) Offsprings selected by the CAA. If the measure of goodness (optimization criterion) is known, (and is applied), then the optimization process gradualy converges toward optimal solution among the possible solutions of the problem; the offsprings of the CAA are always the best solution found up to the considered moment. The CAA knows when the solution found is worse then the previous one.

2) Offsprings selected by the environment. If the optimization criterion is not known, (or the CAA is not applying it) then the optimal solution is found by means of genetic algorithms: CAA generates solutions in genetic environment that are used by some species which test the solutions in the same behavioral environment; the optimal solution genome vector will produce an optimal behavior of some offspring in the behavioral environment, and has better chance to survive then the other offsprings of the solutions generating CAA.

In short, we would like to stress that the CAA is an *inherently* optimization machine, which is a feature due to its connection to the genetic environment.

5.9.3.1. Convergence Issues

Closely related to the issue of optimality is the issue of *convergence* of the solution process toward an optimal solution. Here we will discuss that issue in connection with Dengenos and Dragons environments, for which CAA was originaly designed.

Our discussion on this issue will be in the form of unformal, *arguing proof*, that the CAA can produce an optimal solution in such, deterministic environment. In the next section we will adress the issue of finding optimal solution in stochastic environment. Let us first define all the relevant element in the discussion.

Problem Definition

The Dungeons-and-Dragons environment
The environment is given by a deterministic state-valued graph $\mathbb{VG}$ with n nodes (states). From each node a number transition (alternatives) can lead toward other nodes, the number of transitions (fan-out) being between 0 and m. The graph is arbitrarily connected and arbitrarily directed (some routes can be directed, some undirected). In $\mathbb{VG}$ is defined a set G of absorbing, goal nodes, with assigned positive values. Also it is defined a set U of absorbing, undesirable states, with assigned negative values. One state is chosen as a starting state, S. Also, there is no regularity for placement of the desirable and undesirable states. It is assumed that a path exists which connects the starting node with some goal node, and a path with that property is denoted by $\mathbb{P}$.

The CAA agent
It is understood that the CAA agent is chosen such that it has a *capacity to understand* the above described environment. That means, the CAA has at least n columns, and at least m rows. Also, the *input genome vector* is chosen to be a *surviving one* for the described environment, i.e. which gives appropriate distribution of the primary drives with respect to the given environment. That means, for each goal state, there is a CAA column with all ones, and for each undesirable state there is a column vector with all components with values -1. However, the CAA has no knowledge about the graph connections.

The optimality task
The CAA agent task is to learn to find a path $\mathbb{P}$, and to produce the shortest path $\mathbb{P}min$.

Methods Definition

The SAE state desirability backpropagation method
The state evaluation method is the SAE method. A desirability of a goal state is backpropagated and stored as desirability of the action which was taken in the subgoal state in order to reach the goal state. If the (sub)goal state is the *k*-th state, and the previous state was the *j*-th state, and the action taken was the *a*-th action, then the component w_{aj} will receive the *k*-th state desirability. If desirability of the *k*-th state is 1, then $w_{aj} = 1$.

The backward-chaining solution finding method

The method which CAA is using, is the *backward chaining* method. It has two phases: 1) search for a goal state, and 2) when a goal state is found define a previous state as a subgoal state. The search is performed using some searching strategy, in this discussion *random walk*. When, executing a random walk, a goal state is found, then a subgoal state is defined, and it becomes a new goal state. A proces of moving from a starting state to a goal state is a single *run* (iteration, trial) through the graph, since the found goal state is absorbing one. The next *run* starts again from the starting state, and will end in a goal state. In each run a new subgoal state is defined. The process finishes when the starting state becomes a subgoal state. That completes a solution finding iteration.

The CAA solution optimization method

The opmizatin method is based on assumption that the criterion of goodness is known, and it is minimum path length. The solutions found are tested against the smaller length found so far, and are selectively generated.

Now when we have described the neccessary notions, we can state

The CAA convergence theorem.

Given a CAA agent, which implements 1) the SAE state desirability backpropagation method, 2) backward-chaining solution finding method, and 3) CAA solution optimization method. Given a Dungeons-and Dragons environment by the 𝕍𝔾. The CAA will produce the path ℙmin with probability one.

Proof of the theorem:

The proof will be given as a *set of ehxaustive arguments*:

1) Has CAA chance of visiting each state during the initial search?. The CAA has a random walk searching mechanism implemented as a random number generator with unform distribution. That will provide an *equiprobable* choice of actions in a given situation, i.e. from each state every state is reachable equiprobably during the initial search.

2) Will CAA find a goal state? Since there is a path to a goal state by assumption, there by argument 1), there is probability *p* that a goal will be found. As number of steps in a run approaches infinity, the probability of finding a goal state approaches unity.

3) How the convergence process is observed? By observation of the narrowing the search space. In each iteration at least one state is eliminated from the search space.

4) How an action selection process, from initialy random choice is transformed to a deterministic one? By SAE state desirability backpopagation method. If a state is a (sub)goal state, the action

which has connected it to some previous state is valued by the value backpropagation process. The value that action receives is above the value that random generator produces. Depending of the value received, +1 or -1, the action will be determined always to be chosen, or respectively, never to be chosen again. If the value was -1, the action process will still be random, but on smaller set of possible actions. If the backpropagated value was +1, the action is later on chosen deterministicaly.

5) Is it possible that several subgoal states are formed from the *same* goal state? Yes, it is. CAA can approach a goal state from different states in different iterations. Each approach will create a separate subgoal state.

6) By argument 5), after several iterations there will be a several separate chains of states which will lead from (same or different) goal states toward different directions in the graph. Is it possible that a loop can be formed so that a backward chain toward the starting state is never established?

This is a crucial issue which needs more elaboration.

As the task is described, the only way the CAA not find a path is to enter to a cyclic path, to an infinite loop. We will show that no loops can be learned by a CAA system. The illustration of the problem is given on the Figure 5.38.

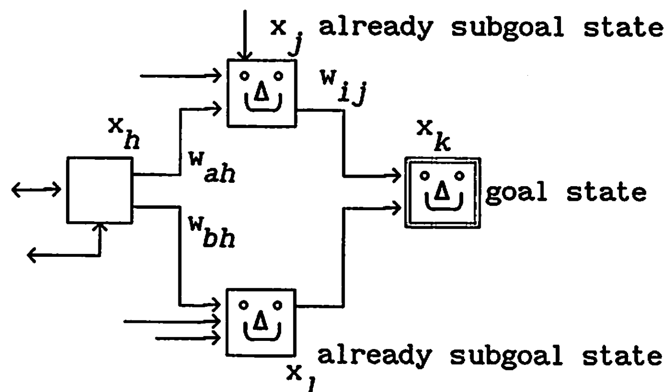


Figure 5.38. Avoiding cycles during the CAA value backpropagation process

As Figure 5.38 shows, two alternatives can be chosen from the state h, and if both are activated, a loop will be formed toward the goal state, in this case the k-th state. But that never happens, since if one of the action is chosen, the others are eliminated. That is because of the greedy policy in choosing actions, which is in the definition of the CAA system. The learned alternative will disregard the other possible backward paths toward the state to which the learned alternative is assigned. In the Figure 5.38, if alternative *a* is chosen, then *b* is disregarded, and the possible loop through the *l*-th state is disabled.

7) When a learning epoch is completed? When the starting state becomes a (sub)goal state. That means that a path with values (subgoal) states is established toward a goal state. Then the memory of the system containing behavioral policy stored in the SAE

components is prepared for exporting to the genetic environment.

8) How we know that the optimal solution will eventualy be produced? Whenever a learning epoch is completed, the obtained path is checked in its length and compared with the shortest length obtained up to that epoch. While the new learning epoch always starts, the solution is generated only if it is now worse that the previous one. Since solutions are continously generated, and since are generated randomly, then as time approaches infinity the procees will generate all the possible solutions. Among all the possible solutions, the best solution is contained with probability one. Since CAA will recognize and store the shortest path length, it will produce the optimal solution with probability one. Once exporting an optimal solution, all the solutions exported thereafter will be optimal solutions only.

That exhausts all the relevant issues and proves the convergence theorem. We believe that this nonstandard *arguing proof* will be of interest to the reader interested in issues around the optimal policy finding in valued graph. The process described proves how CAA can perform, *in real time, by doing,* search for an optimal solution using only one memory structure, the matrix of SAE components.

5.9.4. DISTINCTION FROM THE GENETIC ALGORITHMS

The communication between a CAA system and both the environments it is connected to, can be depicted as shown in Figure 5.38.

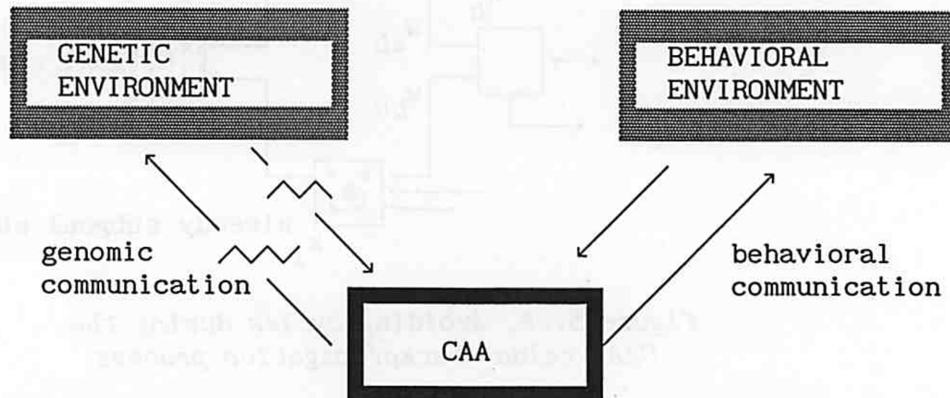


Figure 5.38. The CAA agent processing and comunicating with a genetic and a behavioral environement

As Figure 5.38 shows the CAA can be viewed as an information converter between the genetic and behavioral environment.

Let us note that Figure 5.38 can also be viewed as a classical setup in research in *genetic algorithms*. The difference which we would like to stress is that CAA uses the idea of genetic algorithms in *reverse direction.* Instead of performing stochastic genetic operations as mutation and crossover, and export a new species in the behavioral environment, the CAA applies the learning operators and exports the obtained offspring chromosome back in the genetic environment. So, the CAA architecture effectively demonstrates the

idea (e.g. Hinton and Nowlan, 1987) of speeding up an evolution process due to learning.

5.9.5. SELF-REINFORCEMENT UTILIZING THE GENETIC ENVIRONMENT

At the end of this chapter we would like to emphasize that the genetic environment enabled the CAA to be viewed as a self-reinforcement learning system.

So far, the learning system research has been maily concerned with the behavioral environment. The notion of self-reinforcement was considered as marginal, assuming that one can always view as the reinforcing mechanism to be either part of the agent or of the environment. That is not a case with the CAA. The CAA architecture has reinforcing mechanism included in its memory structure. The reinforcing mechanism of the CAA is based on the SAE components, and they cannot be considered as a part of the behavioral environment. As we see, they are unseparable part of the CAA agent as a self-reinforcement learning system.

This report is indeed an attempt to provide a theoretical framework for the self-reinforcement learning agents which we believe will be of interest in the reinforcement learning theory and in exploring the notion of self-learning in general.

CHAPTER 6

RELEVANCE OF THE CAA TO THE CONTEMPORARY ISSUES OF THE REINFORCEMENT LEARNING THEORY

Here we will discuss some issues about the relevance of the work on our Crossbar Adaptive Array architecture to the contemporary reinforcement learning theory. Special attention will be given to the relation with the Q-learning, a contemporary widely used method in delayed reinforcement lerning. We will argue that it is actually the CAA method, just rediscovered 8 years later.

This chapter is result of our *research* on the contemporary issues in reinforcement learning and relevance of the CAA system in that respect. Although we tried to be impartial, we are aware of the fact that it is still a personal evaluation of one's own system design, and it is neccessarily subjective. We believe that an impartial reader will benefit from this analysis, and will be able to judge its objectivity.

6.1. AFTER 1982...

Our work on CAA took place mostly in 1981. Part of the work on entropy and pole balancing continued after that. The reports on CAA work was made through the internal reports given in the Appendix of this text. Two short reports were published, one on the CAA method solving the maze learning task and the other on learning in loosely defined environments and solving the pole balancing task. Both the tasks were stated by the ANW group and all the reports were given to the members of the ANW Group.

Since then, several influent research report appeared. Firstly, the paper of Barto, Sutton and Anderson (1983) on Actor Critic Architecture and the solution of the pole balancing task using the Michie and Chambers (1968) control space function approximaton. Other very important report is the Sutton (1988) report on temporal-difference learning method. Finaly, and most important for

our research, appeared the research report of Watkins (1989) as his PhD Thesis. The two former mentioned reports describe methods which are remarkably different that the CAA method. Even the AC architecture which was in process of development in 1981 parallel to CAA architecture, is remarkably different.

Watkins (1989) proposed the wellknown Q-learning method which confirmed the CAA method proposed eight years earlier. His method works also with one memory structure and uses the SAE components as principal computing concepts. In short, it is interesting to explore the apparent similarity between the two methods. In the next section we will give our observation about the relation between the two methods. In short our conclusion is:

6.2. Q-LEARNING IS A CAA LEARNING METHOD

In his Ph.D Thesis Watkins (1989) proposed a method denoted as *Q-learning method*. The importance of that method for the reinforcement learning and dynamic programming is comparable to the importance of the Backpropagation algorithm [Rumelhart, McClelland, and the PDP group, 1986] for the neural multilayered supervised learning. Both the events have managed to turn the attention to a great number of researchers for the mentioned areas.

Even not published, the method was immediately recognized as important solution method for the delayed reinforcement learning problem (Barto, Sutton, Watkins, 1990; Sutton 1990). The work is influenced by the Dynamic Programming and the Temporal Difference framework. Alhough there was a work suggesting relation between dynamic programming and reinforcement learning earlier, (e.g Werbos 1977) Watkins did it in more explicit and evident way.

This section will examine the Q-learning and its relation to the CAA architecture. We will also present a taxonomy of various Q-learning algorithms in contemporary *dynamic programming reinforcement learning theory*, as a result of our research.

6.2.1. The problem: Credit-assigment

Let us first describe the problem which *motivated* the proposition of the Q-learning method as a solution.

Given an environment which responds with a reinforcing reaction only occasionaly, and not after every action an agent takes in the environment. Construct a learning agent which will learn a policy in such an environment.

This problem is known as (temporal) credit-assigmnet (Minsky 1961). An action the agent takes in the step t, will be evaluated (reinforced) by the environment several steps (*and actions*) later. In the meantime, the agent is faced to make decisions for other actions in several *stages*, until a reinforcement is (eventually or finaly) *evident*.

This is the same motivational problem which produced the CAA architecture.

6.2.2. The approach

As [Watkins 89] emphasises (p.44), "Dynamic programming is a method of solving the credit-assignment problem in multi-stage decision processes. ...The basic principle of dynamic programming is to solve the credit-assignment problem by constructing an evaluation function."

This is the same approach as CAA aproach: we introduced state-evaluation function with no knowledge of dynamic programming. We also assigned *interpetation* of the evaluation function as emotional entity, desirability being in a state.

Dynamic Programming (DP) [Bellman 57] is actualy a theory which is concerned with graphs (state space in AI terminology), nodes of which are assigned a value (usualy a real number), known as value (*V-value*) of the state represented by the node. V-values The concern is 1) to find state-action pairs (so-called *policy*) over the (part or the whole) graph 2) which will create a path from a starting state (or each state) to some goal state, which will optimize some variable known as *return.* The return is a function of state values visited along the way and rewards obtained at each step along the way. The graphs considered are usualy representation of the Markov Decision Problem (MDP), where each transition between states is given a probability value; they are usualy loosely defined, i.e. they can have manu different arcs between the same nodes. As difference to other MDP methods, DP method works *backwards,* in searching for solution.

When we took the challenge of solving the credit-assignment problem in 1981, we took a metaphore for representation of the problem: The computer game "Dungenos and Dragons", and the grid world of from its dungeons, example being the one shown on Figure 5.33. We constructed an emotional graph giving values to the states. We also assigned probabilities to transitions, as shown in Figure 5.33. And we constructed an agent (CAA) for that, classical discrete dynamic programming graph. Also, we solved the problem using the goal-subgoal approach, i.e. working backwards. In other considered problem, the pole balancing, we considered loosely defined graphs and working backwards we solved that problem too. So we worked od DP and MDP, although without knowing that.

In MDP, it is usual that in addition to transition probabilities, actions are assigned an *immediate reward* values. We did not take into account the immediate reward. We did it to emphasize that the CAA system does not need it: it is an *external-reinforcement-free* system, and can learn without the external reward, and rely only on the rewards received from the state values. However, in our philosophy, we do accept immediate cost, which is computed internaly. And in our model we assumed zero internal cost per action.

6.2.3. Q-values

The problem is also how to represent the state-value function in the agent. (Watkins 89) introduced the notion of Q-values for representation of the state-values.

The Q-value is a value of a function Q(x,a) where x is the

current state and a is an action assignet to that state. It is a memory value. For each state x there are |A(x)| stored Q-values, A(x) being a set of all admissible actions from state x.

In short, Q-values are the SAE components of the CAA matrix. To *compare the notation* we just write

$$W_{ax} = Q(x,a)$$

In words, the Q-values are, *exactly*, the CAA SAE components.

The notation Q(x,a) is widely used. If we want to represent the CAA associative matrix in Q-notation it would be as in Figure 6.1.

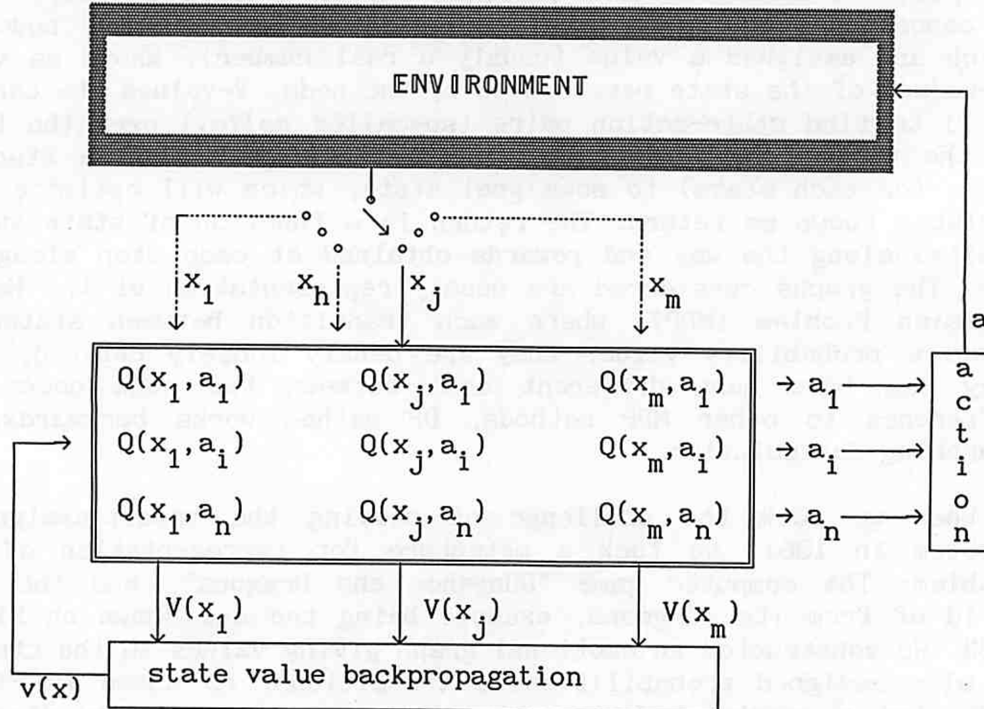


Figure 6.1. The Crossbar Adaptive Array in Q-notation

This representation is often called *Q-table* by the researchers dealing with Q-learning.

Further in the text we will use the Q-notation and our W-notation interchangeably, as synonyms. The W-notation is more appropriate for the nature of the CAA architecture as a *neural network architecture*, since W associates to classical notation for incrementable weights. Other authors [e.g. Dean and Wellman 1991] also use the W-notation, but in W(x,a) form. Also, as analog to Q-values we will sometimes use the term W-values instead of SAE components.

6.2.4. Q-learning: A special case of the CAA learning method

Introducing Q-values Watkins (1989) proposed a method how they can be updated and backpropagated. In short, he used a CAA learning method.

Let us recall that the CAA learning method is given by the following procedure.

CAA learning method

1) state j: perform an action biasing on SAE components; obtain k
2) state k: compute state value using SAE components
3) state j: increment active SAE value using the k-th state value
4) j = k; goto 1

This *forth-back-forth procedure performed over the SAE components* is the CAA learning *and* value-backpropagating method (Bozinovski 1981, 1982). If we ommit the forth step of the procedure and concentrate only on the functions computed, we can write the following symboloc form the CAA method:

CAA learning method

state j: $y = \underset{a \in A(j)}{Afunc}\{w_{aj}\}$ result y = i

state k: $v_k = \underset{b \in A(k)}{Vfunc}\{w_{bk}\}$

state j: $\Delta w_{ij} = Ufunc(v_k) = \underset{b \in A(k)}{Wfunc}\{w_{bk}\}$

where $a,b = (1,..,i,..n)$ and $j,k = 1,..,m$, and the functions *Vfunc, Afunc,* and *Wfunc* are defined over the sets of components of the column vectors, while the function *Ufunc* is defined over the state values. All those functions are defined over the same matrix W. To ensure random behavior, we state that *Afunc* computes a *bias* for an action, not the action itself.

In what follows we will review several *versions* of the CAA learning method. First we will compare Watkins' (1989) version which we denote as Q-CAA version, and our (1981) version which we denote as NN-CAA version, emphasizing that it is a neural network version. After that we wil, give a taxonomy of some other versions of the CAA learning method.

Watkins (1989) used the following specifications for the above mentioned functions

Q-CAA version (Watkins 1989)

state j: $y = \underset{a \in A(j)}{argmax}\{w_{aj}\}$ result y = i

state k: $v_k = \underset{b \in A(k)}{max}\{w_{bk}\}$

state j: $w_{ij} = (1-\alpha)w_{ij} + \alpha(r_{ij} + \gamma v_k)$

where α is a tunable parameter, denoted as convergence rate. In our theory it is a *forgetting* parameter (see section 2.5), which is usualy used in the learning rules where memory decay is used. It is important for learning in stochastic environments. The parameter γ is _so called discount factor, and is a parameter introduced in dynamic programming. The r_{ij} is imediate reward, the external reinforcement, standard feature of the external reinforcement learning systems.

The NN-CAA version is defined as

NN-CAA version (Bozinovski 1981)

state j: $y = \underset{a \in A(j)}{argmax}\{w_{aj}\} + \sigma\{w_{aj}\}$: result y = i

state k: $v_k = \underset{b \in A(k)}{neur}\{w_{bk}\}$

state j: $w_{ij} = w_{ij} + v_k$

where

$$\underset{b \in A(k)}{neur}\{w_{bk}\} = sign(\underset{b \in A(k)}{sum}\{w_{bk}\} + \underset{b \in A(k)}{T}\{w_{bk}\}),$$

is the *neural computation function*, with threshold *T* as a function of the current set of weights. Function $T\{.\}$ modulates the state-value backpropagation process. It is important for solving problems in stochastic environments and also in DND environments, where it serves as a warning function. The function $\sigma\{.\}$ modulates the action generation process. It enables the NN-CAA system to make exploration steps, regardless the value of the W-values. In more general cases, function *sign(.)* can be some other activation function, for example the sigmoidal one.

We can make a comparison between the two CAA algorithms observing each step of computaion, i.a. how they compute the action, how they compute state value, and how they update the memory components.

The *action computation* process is the same for both algorithms. The Q-CAA algorithm does not use the function $\sigma\{.\}$ explicitely, but it uses it implicitely. In NN-CAA it is result of the subsumption architecture. In both cases, that function enables the agent to behave randomly until it learns how to behave purposively. Explicit introduction of the function $\sigma\{.\}$ in the NN-CAA algorithm enables a more general exploration strategy then just a random walk.

The *state-value computation* in the Q-CAA uses greedy strategy and NN-CAA uses neural strategy to compute the state value from the W-values. The greedy computation in the Q-CAA algorithm has no parameters, as difference to the NN-CAA. The neural computation

strategy requires the threshold as parameter function, and is task dependent. The neural computation can generate a greedy policy, as we showed in the previous chapter, and which we actualy used in our experiments. It also takes care of situation in Dungeons-and-Dragons environments, or in some "walking near the edge" environments, where a *warning* should exist if a "dangerous" state is ahead. This function considers the problem of decision under risk, not considered in Q-CAA version of the CAA method.

The *memory update* function of the NN-CAA vesion is a reduced case of the more general Q-CAA version. The NN-CAA version does not use the forgetting factor α. The factor α is the *tunable parameter* of the Q-CAA algorithm, as is the threshold function $T\{.\}$ for the NN-CAA algorithm. The parameter γ is an additional parameter of the Q-CAA algorithm, which is used as constant. What is an important conceptual difference is the external reward r. The CAA architecture does not receive external rewards. However, it can compute internal costs. So, for the CAA architecture the reward function r is the internal cost for taking an action. Since in NN-CAA version we do not assume internal cost per action, we have r=0.

After this brief comparison we can see that the Q-CAA version has a tunable parameter in the memory update function and NN-CAA has such a parameter in the state evaluation function. For both parameters there is no defined procedure how they change: they are both task dependent.

6.2.5. A taxonomy of CAA-method based learning algorithms

In the conteporary reinforcement learning research under the notion of Q-learning method there is general feeling [e.g Lin 1993] that:

> The idea of Q-learning is to construct an evaluation function,
>
> Q(state, action) $\longrightarrow$ utility
>
> which will be used for evaluation of some utility, for each pair (state,action) given that the agent is in that state and executes that action.

In fact, the Q-learning method as proposed by Watkins, was also assumed as a *family* of Q-learning methods [Watkins 89, p.97]. There has been noted by other authors [e.g. Barto, Bradtke, Singh 1995] that there are a number of Q-learning algorithms. So far there has not been an attempt for presenting a taxonomy.

Here we will present such a taxonomy. We will emphasize that it is a taxonomy of CAA learning methods, in which Q-CAA is a special case. As axes of the taxonomy, we will consider the three generic function of the CAA learning method, *Afunc{.}*, *Ufunc(.)*, and *Wfunc{.}* for action computation, state-value computation and memory update computation, respectively.

A taxonomy tree for computing the action is given on Figure 6.2.

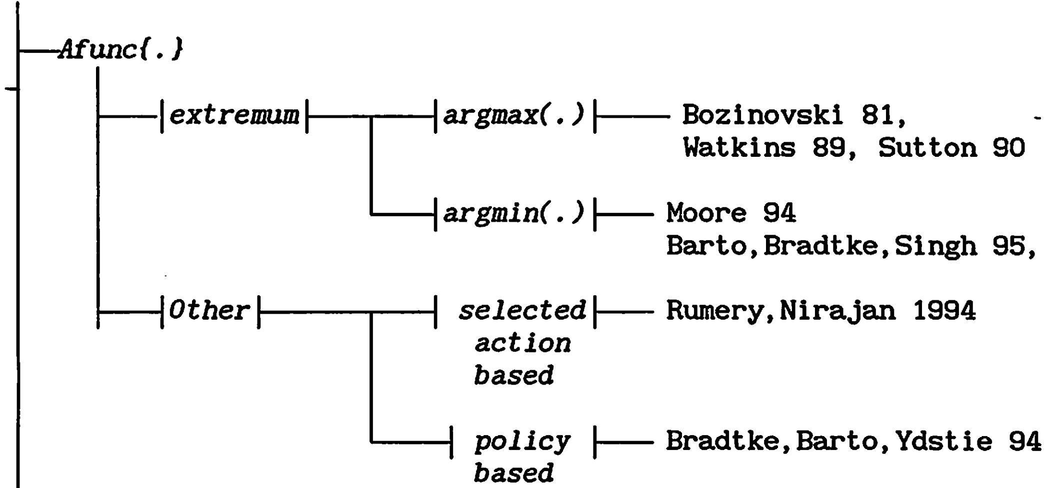


Figure 6.2. A taxonomy tree for various choice of Afunc{.} function

Besides saying that the most frequently used function is the maximum function, at this point we will not go into details of the algorithms corresponding this issue.

For the *Vfunc(.)* function, we see the following taxonomy tree

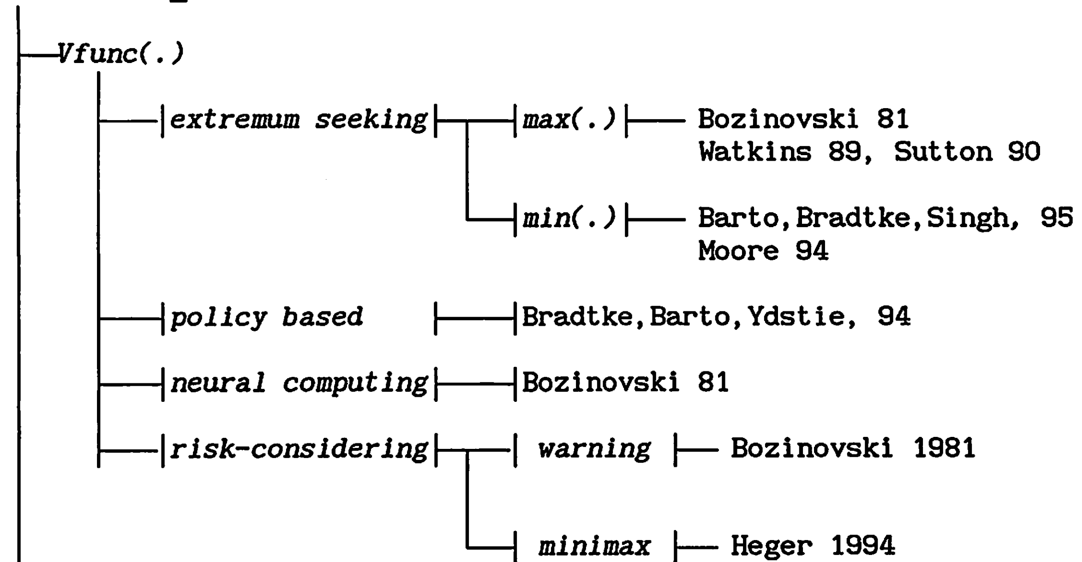


Figure 6.3. A taxonomy tree for various choice of Vfunc(.) function

Most frequently used function is the maximum function,

$$v_k = \max_{b \in A(k)} \{w_{bk}\}$$

The minimum function is used usualy in contingency with the *argmin{.}* function for action selection.

There are algorithms which do not use the extremal value. Example is the policy-based algorithm of Bradtke, Barto and Ydstie (1994). They use

$$v_k = w_{policy(k),k}$$

where *policy(.)* is some designated policy, which may or may not be the policy that is actually followed during learning.

Some algorithms consider the risk of taking the next action. Besides the NN-CAA which uses warning concept, the issue of *decision under risk* is considered in the $\hat{Q}$-learning algorithm proposed by Heger (1994). Heger considers the problem of the risk that the total cost of the actions will exceed some value. Instead of using extremal functions he uses a minimax algorithm for computing Q-values.

For the *Wfunc(.)* function, we see the following taxonomy

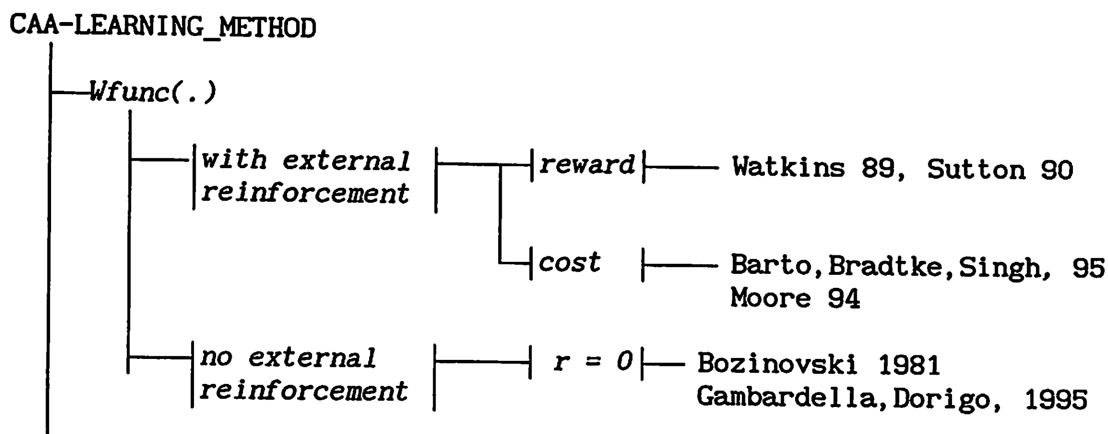


Figure 6.4. A taxonomy tree for various choice of qfunc(.) function

The model for a *Wfunc(.)* function is the original Watkins' (1989) function

$$w_{aj} \leftarrow (1-\alpha)w_{aj} + \alpha(r_{aj} + \gamma v_k)$$

which we already discussed. Here we should add that considered as a reward, the value of r should be maximized, and that is why the functions *Afunc(.)* and *Vfunc(.)* are maximum functions. This types of *CAA*-learning algorithms we denote and maximum-and-reward algorithms.

In other types of algorithms, the immediate reinforcement is considered as a *cost*. A usual formulation is [e.g. Moore 1994]

$$w_{aj} \leftarrow c_{aj} + v_k$$

where $v_k = \min_{b\in A(k)} w_{bk}$, and c_{aj} is the cost for taking action a in j.

In those algorithms the *Afunc(.)* is the *argmin(.)* function. Those *CAA*-learning algorithms are minimum-and-cost algorithms. We would

like to remark that the distinction between the two types of algorithms, as we see it, is important. Both the type of algorithms are viewing the reinforcement and the state-value to be of same desirability.

The NN-CAA algorithms are reinforcement-free algorithms. The *wfunc(.)* is computed as

$$w_{aj} \leftarrow w_{aj} + v_k$$

This is the simplest form of among the algorithms considered so far. The simplicity is gained by using the genetic defined knowledge, as we discussed in previous chapters. Recently, other authors have implemented reinforcement-free algorithms. Gambardella and Dorigo (1995) in their Ant-Q algorithms have implemented the rule

$$w_{aj} \leftarrow (1-\alpha)w_{aj} + \alpha\gamma V_k .$$

However, they have implemented it only partially. Although the above equation is explicitely written in their algorithm, at some point of their algorithm they also use the external reinforcement *r*.

The above taxonomy is not exhaustive. It is useful just to point out that there are differencies how researchers use different functions for examplefication of *Afunc*, *Vfunc* and *Wfunc*, proposing versions of what is meant by Q-learning. What is not changing in all versions, is the essence of the method. An that is 1) it consists of three functions *Afunc*, *Vfunc* and *Wfunc*, 2) the order of their computation is (*Afunc*, *Vfunc*, *Wfunc*) or some permutation, and 3) they operate over the table of W-values (or more popular, Q-values).
That method was proposed in 1981, as the CAA learning method.

6.3. PRODUCING OPTIMAL SOLUTION IN STOCHASTIC ENVIRONMENT

Since proposal of the CAA method in 1981, and its explanation in 1989, many issues have been explored and become established concerning learning using the CAA methods. However, there are issues which are of active research. One of them is the convergence toward optimal solution of the CAA learning methods in the stochastic environments.

In 1981 when we dealt with the pole balancing problem, which can be considered as an MDP with *unknown probabilities*, we were not aware of the general importance of the problem of finding an optimal policy in a stohastic environment. We will discuss this problem from the contemporary state of the art viewpoint.

Our discussion will begin with considering two representation of the CAA learning method: path acting, and point acting representation

```
CAA learning method, path acting
1) state j: perform an action biasing on SAE components; obtain k
2) state k: compute state value using SAE components
3) state j: increment active SAE value using the k-th state value
4) j = k; goto 1
```

```
CAA learning method, point acting
1) state j: perform an action biasing on SAE components; obtain k
2) state k: compute state value using SAE components
3) state j: increment active SAE value using the k-th state value
4) choose j; goto 1
```

As we can see the difference between the two representations is only in the step 4. In point acting representation, the CAA learning method is applied at possible arbitrary state in the state space. In path acting, the method is always appllied on the state along the trajectory.

To the best of our knowledge, the approach taken so far for giving a convergence proof of finding an optimal policy in stochastic environment is using the point acting version (e.g. Watkins 1989).

The result of such an attempt is the wellknown conditioned convergence of the Q-learning method stated that "the Q-learning will converge with probability one toward optimal policy providing that each states are visited infinitely many times".

To achieve that convergence, it is required that the procedure exists of how the tunable parameter α should be tuned along the way. Such a procedure seems to be a problem, and is not part of the Q-CAA version proposed by Watkins. Because of that, despite several convergence proof of the researchers in the area, there is a feeling that "Q-learning cannot find stochastic stationary policies, because it is designed for finding stationary deterministic policies in MDPs" (Kimura, Yamamura, Kobayashi 1995). As main reason, the mentioned work states, is that "policy and the exploration strategy are threated separately in Q-learning".

In this report we will not go into that issue further. What we will present next is our solution to the problem using the path acting approach in stochastic environments, instead of point acting approach.

In the next frame we give our at-subgoal-go-back CAA algorithm for finding a shortest path in a stochastic environment.

```
CAA at-subgoal-go-back algorithm
repeat
   forget the previously learned path.
   define starting state
   repeat
      from the starting state
      find a goal state moving randomly;
      produce a subgoal state using CAA learning method;
      mark the produced subgoal state as a goal state;
   until starting state becomes a goal state.
   export the solution if beter then the previous one.
forever
```

The main difference between this and the original (1981) algorithm is that in the original one new iteration started always when a *goal state* is reached. Here new iteration starts if a *subgoal* state is reached. To the best of our knowledge, this modification of the algorithm was not proposed before. That is probably because the difference seems unimportant, and in fact it is conceptualy unimportant for deterministic environments. (It is important computationaly, since it will spare some computational effort). But, for stochastic environment it is conceptualy important and in fact we claim *is a solution of the problem* of learning on-route in stochastic environments. However, the efficiency is not claimed, and it should be tested with other methods for on-route learning in stochastic environments.

As we noted in our experiments with pole balancing, a control should be somehow defined over the state-value backpropagation process. The backpropagation should be made (for example only one step behind as we did in pole balancing), only at the goal states. Extending that principle, here CAA backpropagates *only the newly defined goal state*, actualy only at the newly found subgoal state. After that, the CAA agent can start new iteration. In some variants, it can even continue to explore the environment, but no more learning is alowed for that iteration. That prevents obscuring the previously learned segment toward the goal.

As a result, the CAA will find a possible probabilistic path toward the goal. The rest of the algorithm is the inherent CAA optimization procedure. Since the measure of goodness is defined, the CAA exporting procedure will optimize over all the possible paths generated in the process, and will find the best one with probability one.

Again, we should emphasize that this procedure works only if the measure of goodness is well defined. Consider the following example, on Figure 6.5.

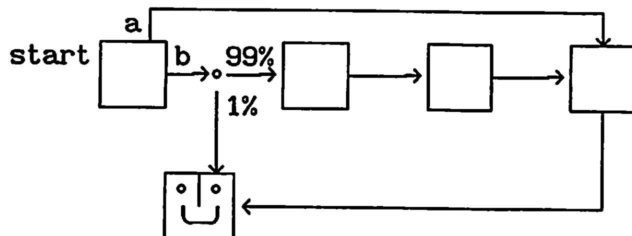


Figure 6.5. Example for policy chosing in a stochastic environment

As Figure 6.5. shows, the shortest possible path to the goal state can be achieved only by chosing the action *b* at the starting state. The optimization procedure defined above, as a solution will produce the action *b*, since it seeks a shortest path that can possibly be produced in a set of possible paths. But a question is is it good in this environment? This illustrates that the measure of goodness should be defined properly if we want to obtain an optimal solution in some desired sense.

As closing observation to this discussion let us note that this CAA procedure does not use any parameter tuning. Also, it is exactly defined when learning should stop; The issue of exploration/exploitation is not of interest in this optimization procedure. That demonstrates the power of the CAA architecture and the three environment concept.

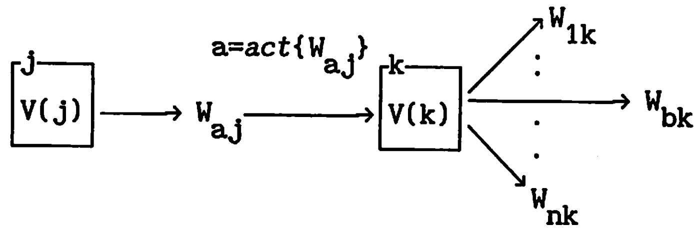
We shold also note that the proposed optimization procedure is general, and not specific in connection to the genetic environment. For example, a system with two memory structures, can replace the concept of genetic environment. Consider CAT system with two memory tables of SAE values, one for learning an arbitrary solution, and the other for keeping the so far best found solution. This system will also produce the optimal policy in a stochastic.

6.4. A SUMMARY OF THE OBSERVED RELEVANCE

At the and of this chapter we sumarize some issues around the CAA architecture which are interesting from today's perspective. The relevance is viewed 1) as an early, 1981 neural architecture, and 2) as a state-of-the-art architecture.

6.4.1. CAA as an 1981 neural architecture

1. In 1981 the CAA architecture was the first architecture which introduced a learning method, contemporary *known as* a Q-learning method. Figure 6.6 shows the method used in CAA architecture



$$W_{aj} \longleftarrow W_{aj} + f(V_k), \quad \text{where } V_k = g\{W_{bk}\}, \ b=1,..,n$$

Figure 6.6. The CAA learning method

To the best of our knowledge, there was no architecture before, using this forth-and-back procedure *over the state-action components* (today known as Q-values). The procedure was used in DP but not over state-action components. Also, state-action components were used before in DP, but not in a learning process.

This method was rediscovered, and explained in DP framework by Watkins (1989). A specific Watkins' (1989) algorithm is different then the specific NN-CAA learning algorithm, and we do not clame that NN-CAA is the same algorithm as Q-learning algorithm, nor it is the same to other later proposed CAA-method-based algorithms. The specificity is expressed in functions *f*(.), *g*{.}, and *act*{.}. What we do clame is that the procedure on Figure 6.6. was first time discovered in 1981 and used in CAA architecture.

2. Using that learning method, in 1981, CAA architecture solved the maze running problem, a problem in the class of *credit assignment problem* stated by Minsky (1961). To the best of our knowledge, it is a first solution of such a problem *using a neural network*. Minsky (1954) worked on that but is is not clear whether a solution was found, at list is not clear from Minski (1954) and is not given in Minsky (1961); in Minsky (1961) only the problem is stated as a challenging one. Before our 1981 neural network solution, there were several solutions in AI, examples being Samuel (1959) and Mickie and Chambers (1968). Parallel to CAA, the AC architecture also gave a neural solution of that problem.

3. Using the learning method described in 1, in 1981 the CAA architecture solved the pole-balancing problem. The solution was applied on the context-unrestricted task, imposing control over pole angle and not over cart position. The problem of state-value function approximation was solved using 10 regions. In addition, the problem of action function approximation was solved using 3 actions, which is remarkable improvement over similar solutions using 2 actions. The classical approach (Michie and Chambers 1968) used 162 state-value approaximation regions and 2 actions. The number 162 is due to the complexity of the task considered, taking into accont also the cart position. But 3 actions (left, right, and *do-nothing*) approach is improvement over the classical (left,right) approach, and greatly simplifies the task. It is interesting that all the known solutions (Widrow and Smith 1964, Mischie and Chambers 1968, and Russel and Rees 1975) before the the CAA 1981 solution, used the two-action methods.

The pole balancing problem is in a class of nederministic environments problem, where taking an action gives only a probability which situation well be received next. In this problem

even the probabilities are not known. The method of *segments of the path* backward-learning was introduced for solution of this problems, as difference from the method of whole path backward learning, used in maze learning problems, which is in the class of deterministic environment problems.

4. Genetic environment was introduced as an important part of the CAA architecture. Primary reinforcers are defined as drives comming as innate knowledge, on which an intelligent behavior is developed using secondary reinforcement principle. That connected reinforcement learning (RL) research to the genetic environment based reserach areas, as Genetic Algorithms and Artificial Life.

5. The relevance of the CAA can be seen also in observing some facts about conceptual expressions and representation methods used. Some of the points are

- The concept of *searching strategy* is introduced in CAA for what is later known as exploration strategy. The concept is based on the idea proposed by Barto, Sutton and Brouwer (1981) where as neural search mechanism the Gaussian noise was used. The 1981 CAA used the random walk strategy. Contemporary, the concept of searching strategy is widely used, but not the concept of noise.
- The *algorithmic expression* of the learning procedure is used in CAA report (1982b), not only the equations of learning. In contemporary research reports in reinforcemnt learning this representational technique is widely used, which was not the case in RL before 1982.
- The *human face state-value* representation was introduced in RL. Today this is widely used representational concept in RL and DP.
- *Parallel programming* was for the first time in RL and DPused during the CAA experiments. Pole balancing task was carried out that way.
- The *entropy concept* was used as representational parameter for learning.

The importance of these facts is not clear, and may be neglected. However, the importance will become more clear in some time later, in positive or negative sense.

6.4.2. CAA as a state-of-the-art neural architecture

6. CAA is a *neuro-genetic architecture*. It takes advantage of the genetic environment in a way complementary to the Genetic Algorithms approach.

14. The CAA is inherently *otimization architecture*. Taking advantage of genetic environment concept, it performs continuing optimization over the solution learned in the behavioral environment.

15. CAA is a stochastic environment optimizer system. Taking advantage of its *at-subgoal-go-back algorithm*, it is able to produce optimal solutions (policies) in the stochastic environments, working on-route.

16. CAA is a *self-reinforcement neural network*. It does not need the external reinforcement to learn. From DP viewpoint, it only uses

the state-value, not the immediate reward, in its learning rule. CAA has internal mechanism how to extract a state-value from a state, and using the concept of desirability, to learn from consequence. In solving the problem of self-reinforcement, the genetical environment concept is used.

As conclusion we can say that, viewing from this perspective, the 1981 CAA neural architecture had implemented some *far-reaching issues*, some of them confirmed by the Watkins' (1989) work. Some of them are explored in this report. However, the CAA architecture is unknown in the community, and this is an attempt toward understanding the relevance of a work done in 1981 within the Adaptive Networks Group.

CHAPTER 7

CLOSING DISCUSSION

At the end of this report we will give a short discussion about general issues relevant to this report.

7.1. Unified Theory of Learning Agents

In this report we discussed various learning tasks that can be solved by generic reinforcement learning agents which we named as NG agents. The report contains three parts describing the most important variants of the *situation sensitive reinforcement learning agents.*

In the first part we discussed our work on learning agents that besides the reinforcement (as performance evaluation from the teacher), receive an advice (from the teacher) what to do.

In the second part we briefly mentioned classical reinforcement learning systems, the systems that only receive the performance evaluation in a form of scalar signal from the environment. They do not receive advice what to do; rather should discover themselves what to to. But they do receive an *external* reinforcement in addition to the "neutral" situation signals, as contrast to the agents we consider in the third part.

The third part is the crucial in this report. Here we described a learning agent, based on our Crossbar Adaptive Array architecture, which is capable of learning even without external reinforcement. Instead, it has internally defined drives which serve as primary reinforcers. It receives those drives in a hereditary fashion. It receives a genome information with those drives, and exports a genome with some changes in it due to learning. The assumption about

exporting a learned information in a genetic fashion is convenient in this theory. The principle of self-reinforcement is indeed based on the connection between genetic and behavioral environment.

All the three parts are results of the Adaptive Networks Group produced in the period in 1980-1981 during our first work with the group. However, the structural theory of reinforcement learning system, has been further developed in this text. It is a framework for consideration of all the three areas described in the first parts of this report, and also others described in the first chapter.

All the three parts are components of our structural theory of reinforcement learning systems. What is a theory? We consider it to be some kind of explanation of some existing facts. In a sense, it a "curve fitting" between some fact points, such that some other fact points that will appear in the future could be predicted.

This theory covers the existence of the self-learning systems and it provides a framework for their analysis. There can exist theories which ignore the existence of self-reinforcement learning systems. They will tend to make different "curve fitting".

Besides covering self-reinforcement-only learning systems, this theory also covers wide range of different kinds of learning paradigms and learning tasks. In a sense it is an extension of some other similar theories, for example [Barto 1991]. Besides describing various types of learning paradigms, this theory goes a step further. It proposes a learning agent that can be used in different paradigms, in fact in all the paradigms described in the first chapter. In a sense it gives a unified theory of learning agants.

7.2. CAA Architecture

This report, besides presenting a new theory of reinforcement learning systems, has a historical component. What is a history? It is also a theory, also a "curve fitting", dealing with *dated* data. If we have data points, and some regression analysis for which we have favorable interpretation, and if there is a data point which does not fit in that explanation, we have to either ignore the point, or change the regression analysis to fit all the data points. The CAA *is* a data point in the history of the credit assigmnment problem and delayed reinforcement learning. It introduced a learning method which later was rediscovered, interpreted in terms of dynamic programming and become a widely used delayed reinforcement learning method.

7.3. Q-learning as a CAA learning

The work on CAA was done in 1981 and forgotten. Papers published [Bozinovski 1982, Bozinovski and Anderson 1983], showing state-valued (emotional) graphs and presenting the CAA agent as a solution for delayed reinforcement learning problems in such an environments were forgotten. It took the work of Cristopher Watkins (1989) to focus the attention to the "grid world problems" and state-valued graphs among the machine learning community. There was no such big interest before 1990.

This report is partly provoked by the work of Watkins (1989). Obviously, our reports on CAA [Bozinovski 1981, 1982] were not shown to him. Also we did not know about his work until recently. Wihout his work, the CAA would have remained as some neural architecture solving the credit assignment problem in state-valued graphs, which is maybe not worthy mentioning, because if it solved the problem it is not easy to see how. Especialy if for solution of that problèm somebody needs two memory structures [e.g. Barto, Sutton, Anderson 1983, Sutton 1984] and CAA has only one. Watkins showed that for a solution of that problem one memory structure which will be used both for value iteration and policy iteration *is sufficient*, and confirmed the CAA approach. Even more, he gave a deep, dynamic-programming-framework backed-up explanation of that memory structure. Eight years before, not being aware of the dynamic programming work, we constructed the CAA architecture by imagination over emotional graphs and knowledge of adaptive array neural networks.

However, we argue that what we constructed as a learning method, is indeed rediscovered in Watkins' work. Having in mind emotional graphs and weights assigned to their arcs, as well as memory matrix where column-wise the state-values are computed, we developed a forth-and-back learning method over the SAE components. The SAE components can have *different interpretations*. In 1981, we interpreted them in terms of neural networks (state-action associativity weights) and in terms of state evaluation (emotions evaluation components). Eight years later, Watkins (1989) interpreted them in terms of dynamic programming. We do not know whether they were used as learning and state-evaluating variables before 1981, but if they were, they could have probably some other intrepretation. In essence they are the memory values used for 1) action computation, 2) state-value computation and 3) learning. To the best of our knowledge that was for the first time introduced in the CAA algorithm. We argue that this is indeed what is contemporary meant as being a Q-learning method.

There are various versions of the Q-learning method. What is common to all the methods is actualy the CAA learning method. The basic CAA learning method functions, the action function, the state evaluation function, and the memory update function can be found defined differently. What is constant is the CAA method: 1) compute action in state j, 2) compute state value in next state k, 3) update memory in state j; and 4) all that using only one memory structure, the SAE values.

7.4. CAA as a neuro-genetic agent

As a result of the project on neural and genetic agents, this reports explores the usefullness of the CAA architecture being connected to the genetic environment. The utility observed so far is two-fold: 1) the genetic environment enabled the CAA to be designed as a self-reinforcement learnig system 2) the genetic environment enable the CAA to be an inherent optimizing architecture. The concept also opens some issues toward the artificial life research, as is the issue of controlled offspring production.

7.5. Adaptive Networks Group

The CAA architecture discussed in this report is a ANW architecture. The major influence was received from Andrew Barto and Rich Sutton. Without their statement of the problems worthy to work on, the CAA would have never been constructed. With Chuck Anderson we solved the pole balancing task, and had pleasure to experinece parallel programming. Nico Spinelli was always in progress with our work and offered valuable discussions and suggestions. With Michael Arbib we had conversations and advice for our work in various areas, especialy for the concept of entropy and surprise. And in presence of Harry Klopf we took a challenge to solve the assignment of credit problem, which we did and sent a report on that to him first.

Also, this report could have never be written in this form, if it wasn't for Andy Barto to challenge relevant issues not addressed and/or not threated well in the previous versions. The classes taught by him, and the meetings of the ANW Group, all of that has reflexion in this report. As 15 years before, the ANW Group provided an inspirative atmosphere for carrying out a research work, part of which is this report.

CONCLUDING REMARKS

In this first report of the project "Adaptive Parallel Distributed Processing: Neural and Genetic Agents" we examined and evaluated our previous work with respect to the modern trends in science, and also explored some issues on neuro-genetic agents.

This is rather lenghtly report, written from March to November 1995. It has actualy two parts. It introduces a theory of learning systems as a state-of-the-art work, and also deals with some important work done in the past within the Adaptive Networks Group which are relevant for the history of the reinforcement learning systems research.

It is our belief that this report will shed new light on reinforcement learning agents research, and also on relation between neural and genetic research. This report has shown that the period of 1980-1981 was succesful for the Adaptive Networks Group in solving the delayed reinforcement learning problem and introducing a method whose importance was not recognized immediately, but whose importance was confirmed by the eight years later proposed Q-learning method.

The solution of self-reinforcement learning paradigm is yet to be evaluated, as well as the unified theory of learning agents proposed in this report. We believe that those issues will also become relevent in the reinforcement learning theory.

APPENDICES

In the following appendices we give some reports of the relevant events which happened in the development of the CAA architecture.

In particular, we give

Appendix A: The first written report about the CAA, sent to Harry Klopf, November 25, 1981

Appendix B: The announcement of the COINS Seminar on December 2, 1981 where the first time the CAA was talked about

Appendix C: The ANW Memo about the first successfull experiments in pole balancing task, given at the ANW meeting on December 10, 1981

Appendix D: The first published report about CAA, in the book of abstracts of the Sixth European meeting on Cybernetics and Systems reserach, Vienna, April 13-16, 1982

*Appendix E:*The list of the early reports written by the ANW group, from 1981 till 1983

APPENDIX A

Amherst, 11/25/81

A. Harry Klopf
Avionics Laboratory
Air Force Wright-Aeronautical Laboratories
(Attn: AAAT)
Wright-Patterson Air Force Base
Ohio 45433

Dear Harry,

Enclosed please find a short report I prepaired for you on my work on the problem you statrd at the latest meeting of the ANW group with you. I think I have solved that, and I will be happy to show you a real time simulation if you are comming in Amherst in December.

I will be here till December 28. If you are not comming till that time I will appreciate if you give me some feedback on this work.

Sincerely,

Stevo Bozinovski
Computer Science Dept.
University of Massachusetts
Amherst, MA, 01003

CROSSBAR ADAPTIVE ARRAY

A SELF-LEARNING SYSTEM USING SECONDARY REINFORCEMENT

by

Stevo Bozinovski

Introduction

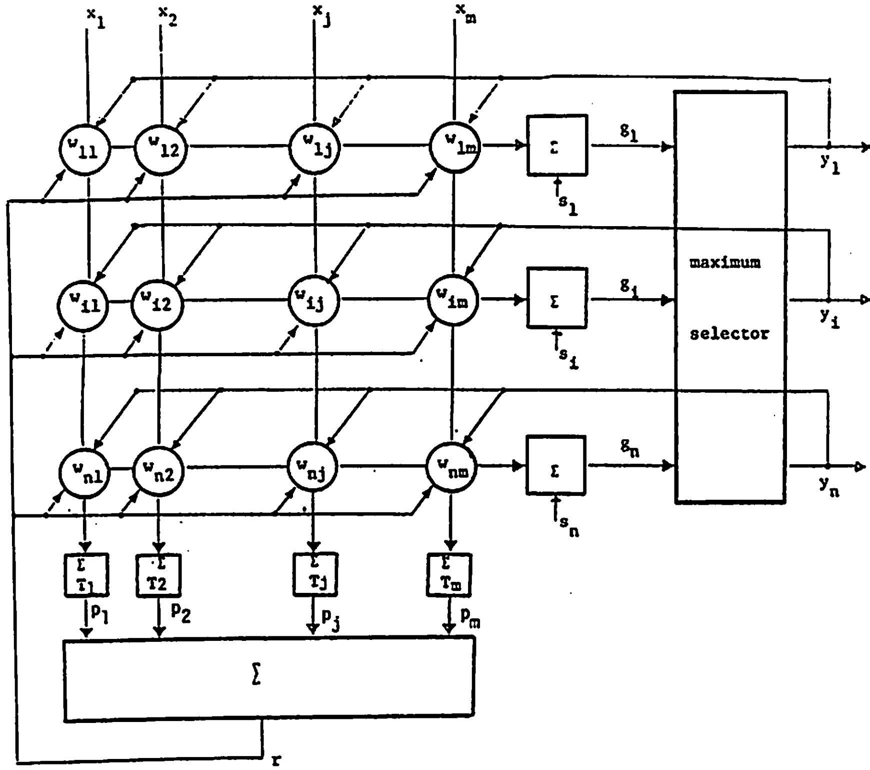
Here is presented a learning system, called Crossbar Adaptive Array, (CAA) which using some "genetically build sense of pleasure", shows the shift of the behaviour from totaly random to totaly organized and purposfull one. The information it receives from its environment is only the present situation; the environment does not supply the system with any evaluation of how "good" was its behavior. The princip on which the self-organization of CAA is based is the secondary reinforcement.

The Crossbar Adaptive Array

Figure 1 shows the system called Crossbar Adaptive Array.

Figure 1

The input situation is received through the m possible senzor pathways x_j, j=1,...,m, which can be active one at a time. That means that the pattern recognition process is not concern of this system: it is assumed solved in some



in the experiments

m=20, n=3

Fig. 1.

preproccesing system. So, this system distinguishes between m different input situations. There are n possible output actions y_i, i=1,...,n, each represented by one output pathway. The interconections between given situation and selected action are described by the nxm weight matrix. The initial values of the matrix are specified, by some "genetical" program. The system has internal "pleasure" senzors p_j, j=1,...,m, which measure the pleasure felt when any of the input situation appears. If the sum produced by a column of the memory matrix is positive, the system is assumed to register pleasure, and if it is negative – pain. The evaluation unit produces the ternary signal +1 indicating the presence of the pleasure, 0 in the stimulus is neutral and -1 if the pain is felt. Producing the signal r, the weight w_{ij} in the intersection of the situation x_j and the performed action y_i which has lead to the situation which has produced the pleasure or the pain, is updated. Formaly, the learning rule of the system is defined by

$$w_{ij}(t) = w_{ij}(t-1)+c.\,y_i(t-1).\,x_j(t-1).\,r(t) \qquad (1)$$

where c is some constant. The reinforcement function r(t) is defined as

$$r(t) = \sum_{j=1}^{m} \mathrm{sign}\Big(\sum_{i=1}^{n} w_{ij}(t-1).\,x_j(t)+T_j\Big) \qquad (2)$$

where T_j is some predefined fixed threshold and sign(p) is a function which gives +1 if the p>0, gives 0 if p=0, and gives -1 if p<0. Note that f(t) takes values from the set {-1, 0, 1}.

The actions are taken according to the decision making rule

$$\text{if } g_i > g_k \text{ for all } k \neq i \text{ then } y_i = 1 \text{ else } y_i = 0 \qquad (3)$$

where

$$g_i(t) = \sum_{j=1}^{m} w_{ij}(t) \cdot x_j(t) + s_i(t) \qquad (4)$$

where s_i is a parameter of some searching strategy, possibly governed by some higher level system. However, in our case there is no sych a system and we set s_i to be a random number from the uniform distibution U[-0.5,0.5]. In such a way, the system performs random walk at the begining of the learning process. The genetically built knowledge is always chosen not to affect that random walk: for given i all w_{ij}, $j=1,\ldots,m$, have the same value. Changing that due to learning process, CAA shifts its behavior from random walk to more deterministic one.

The classical maze learning problem

As the simple task which can be solved by the CAA we will consider the problem of maze learning, one of the challenging problem in cybernetics and psychology of learning. Let us assume a maze in which system enters and is running randomly through it with no special goal. If something is found by chance, and the pleasure is felt, then the system remembers that place and will recognize the vicinity of that place in some next run through that maze. Now in his memory there is a notion of the "goal" , the

knowledge gathered due to the experience is there, not just a geneticaly built-in knowledge. In the next run, the place from which it reaches the goal is subgoal. In a repeating sequence of runs the system can build a internal model of the maze, and can exhibit totaly deterministic behaviour in going directly to the goal.

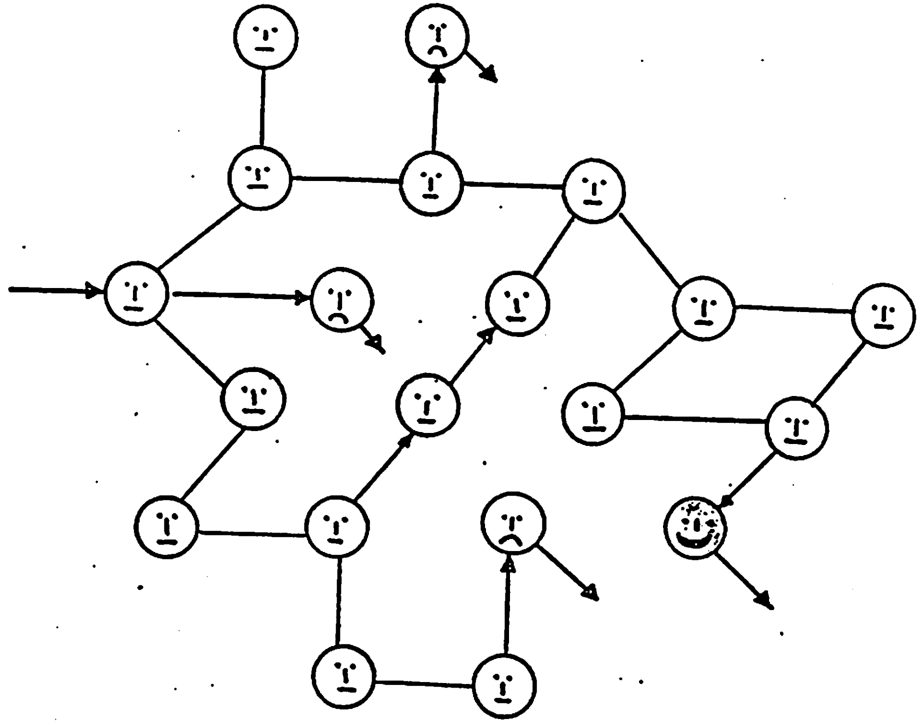
Simulation experiments

One of the mazes we have considered in the experiments is shown on Figure 2. The maze is expressed by a "life graph" which we introduce here as a graph which has assigned to each vertex a "value of diserability". We express that using the human faces as the graphical symbol. Our maze has one input and four outputs one of which is associated with the pleasure but all others with the pain. Some connections are assumed to be one-way for reason of variety.

Figure 2

The result of the experiment of running the CAA through this maze is shown on the following table:

iteration	number of steps before an exit is found	pleasure + pain −
1	9	−
2	4	−
3	21	+
4	17	+
5	19	+
6	75	+
7	23	+
8	22	−
9	7	−
10	19	+
11	32	+



12	78	+
13	12	+
14	17	+
15	8	+
16	12	+
17	8	+
18	8	+

After 17 iterations the system is not changing the behaviour. It runs through the maze reaching its goal in 8 steps, which is the shortest path possible. The further learning, if it occurs, is not observable; CAA is exhibiting totaly deterministic behavior.

Conclusion

A learning system is presented which can exhibit purposive and goal-seeking behaviour receiving no information from the environment other the encountered situation. The concept of geneticaly built sense of pleasure and pain is used instead of the concept of payoff function received from the environment. The problems of secondary reinforcement and assignment of credit are succesfuly solved by this learning system.

COMPUTER AND INFORMATION SCIENCE

UNIVERSITY OF MASSACHUSETTS, AMHERST

CCINS SEMINAR

Stevo Bozinovski

Computer Science Department
University of Massachusetts

Wednesday, December 2, 1981

3:45 p.m.

Graduate Research Center, Room A307

Coffee and cookies at 3:15 p.m.
Graduate ResearchCenter, Room A305

ABSTRACT

ADAPTIVE ARRAYS

Adaptive arrays as distributed memory systems capable of learning and self-organization are considered.

The problem of optimal pattern recognition training is considered in connection with the linear adaptive arrays. Chanllenging task: recognition of the computer terminal symbols. A new representation technique is introduced which naturally includes the notions of transfer of training and interpattern similarity. Some issues concerning goal seeking nature of the process are discussed.

A crossbar adaptive array is proposed as a system capable of self-learning. Challenging task: "Dungeons and Dragons". Some issues concerning problem solving and planning are discussed. Supporting experimental evidence is provided.

Possible mechanisms involved in the self-organization of the neural adaptive arrays are hypothesized. Dendritic bundles as a possible structures are considered. On that basis a hypothetical model called isothreshold adaptive network will be presented.

The announcement of the first talk about the CAA

APPENDIX C

Memo on the
Inverted pendulum control program

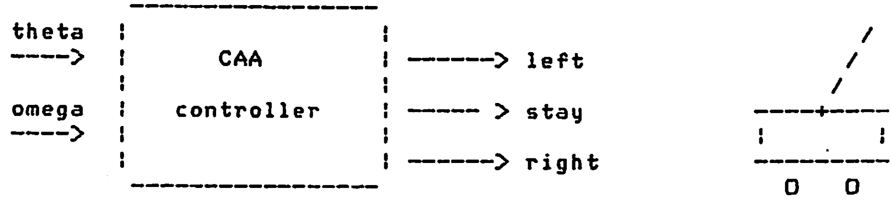
Author: Stevo Bozinovski
Date: December 10, 1981

Problem: Inverted pendulum adaptive control
Although considered several times before, the statement of this problem which has leaded to the development of the software system described bellow was made by Chuck Anderson.

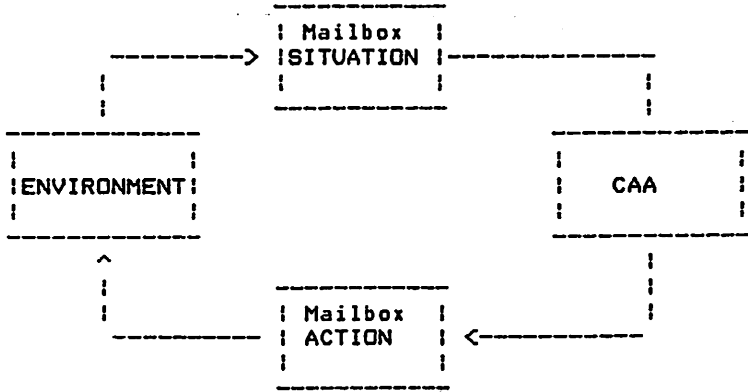
Statement: Construct an adaptive network which will be able to learn how to control the balancing of an inverted pendulum

Possible solution: Crossbar Adaptive Array (CAA) as controller.
Basic principles of CAA are genetical knowledge assumed and secondary reinforcement mechanism.

Program CONTROL
This is the main program of the programming system
It simulates CAA controler of an inverted pendulum
The input is the angle and angular velocity
The output is ternary action (-1,0,1)
(can be interpreted as 'move left', 'do nothing' and 'move right')



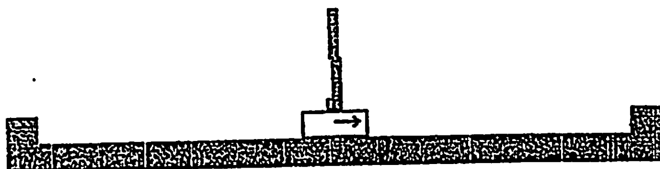
Organization of the softvare system
Two software machines are interacting via mailboxes, running simulatneously on two terminals.
The ENVIRONMENT simulates the dynamics of the inverted pendulum;
The CAA simulates the behavior of the Crossbar Adaptive Array.

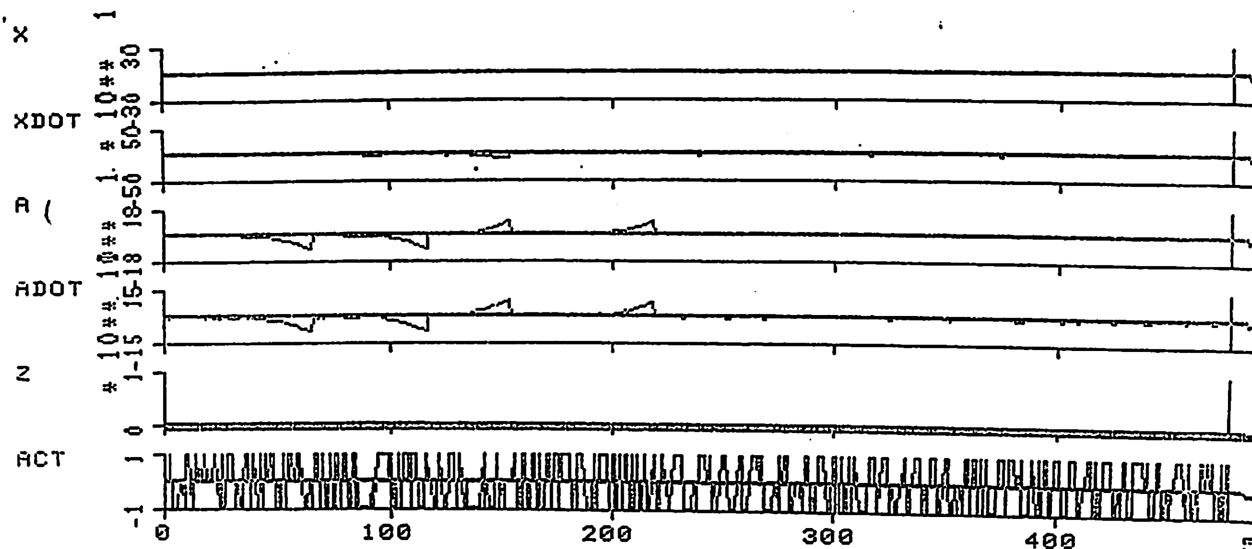


The programming system representing ENVIRONMENT was created and runned by Chuck Anderson. The mailbox communication between the two programming systems was established by Bob Heller.

An experiment of balancing inverted pendulum using CAA controller

(Computer graphics display courtesy of Chuck Anderson)





After four unsuccessful runs, in which the pendulum has fallen down two times on the left and two times on the right side, the controller had learned to perform the appropriate actions in the particular situations, and is balancing the pendulum.

X and XDOT are the positions and the velocity of the carrier. They are not significantly changing during this experiment.

A and ADOT are the angle and the angular velocity of the pendulum. It can be noted their increasing due to inapropriate control at the begining of the learning process. After learning, they are kept in the vicinity of their zero values.

Z is not relevant variable for this experiments.

ACT is the action taken: left(-1), right(1), or stay stil(0).

ÖSTERREICHISCHE STUDIENGESELLSCHAFT FÜR KYBERNETIK
(Austrian Society for Cybernetic Studies)
in Cooperation with
UNIVERSITY OF VIENNA
DEPARTMENT OF MEDICAL CYBERNETICS

Sixth European Meeting on Cybernetics and Systems Research – 1982

Vienna, April 13–16, 1982
in the
Main Building
of the
University of Vienna

ABSTRACTS

Secretariat:
ÖSTERREICHISCHE STUDIENGESELLSCHAFT FÜR KYBERNETIK
A-1010 WIEN 1, AUSTRIA
Schottengasse 3 Telephone (0222) 66 32 81-0, [illegible]

APPENDIX D

A SELF-LEARNING SYSTEM USING SECONDARY REINFORCEMENT

Stevo Božinovski
Cybernetics Department, Electrical Engineering Faculty
University of Skopje, Skopje, Yugoslavia

It is presented a learning system called Feedback coupled Crossbar Adaptive Array, which using no teacher of any kind is capable of exhibiting a purposive and goal-seeking behavior in rather complex environment. The main assumption is some genetical memory which in addition of storing the internal model of the environment is a part of the internal reinforcement mechanism: The interpretation of a "pleasure" and "pain" within the system develops a notion of goal during the learning process. The mechanism of reaching that goal is based on secondary reinforcement: some desireable state becomes a reinforcer for the preceeding state.

Figure 1 shows the Crossbar Adaptive Array (CAA). The input situation is received through m sensor pathways, which are assumed to be active one at a time. The associative matrix $W = \|w_{ij}\|$ $i=1,..,n$, $j=1,..,m$ contains the weight values of the association between i-th action and j-th situation. At the begining of the learning process it is assumed that each column vector w_{*j} of the matrix W has either all zeroes, or all ones, or minus ones. Encountering situation, say x_j, then if $w_{*j}=\underline{1}$ the system feels "pleasure", if $w_{*j}=-\underline{1}$ the system feels "pain", and if $w_{*j}=0$ the system feel neutral stimulus.

The CAA produces only one action at a time providing the computation

$$g_i(t) = \sum_{j=1}^{n} w_{ij}(t)x_j(t) + s_i(t) \qquad \text{for all } \underline{i}$$

and then

$$\text{if } g_i > g_k \text{ for all } k \neq i \text{ then } y_i=1 \text{ else } y_i=0$$

where s_i is a random number from a uniform distribution U(-o.5,o.5).

Now let us consider an example. Let $w_{*m}=\underline{1}$ and all other column vectors of the matrix $\underline{W}$ are zeroes. Let all the threshold values T_j are zeroes too. Performing random walk, the CAA will eventually reach the situation x_m. Assume that those situation is reached from the situation x_j performing action y_i. The following is computed in CAA:

$$- \quad r(t) = \operatorname{sign}\sum_{i=1}^{n} w_{im}(t-1)x_m(t) = 1$$

and

$$w_{ij}(t) = w_{ij}(t-1)+r(t)y_i(t-1)x_j(t-1) = w_{ij}(t-1)+1$$

Thus, only the weight w_{ij} is incremented and association between the situation x_j and action y_i is established. In addition, the column vector w_{xj} is no longer zero. Next time x_j occurs, the system will feel pleasure being in the situation x_j, and the previous situation-action pair will be remembered. The secondary reinforcement mechanism takes place.

The behaviour of the CAA is examined facing it with the generalization of the classical maze learning problem, which we call life graph. It is a graph which each node is assigned a degree of diserability being there. Figure 2 shows such a graph represented by the technique of human faces. This particular environment has 20 nodes, and the CAA living in it has m=20, and n=3 since number of the alternatives from each node is maximum three. There is one input and four outputs, only one of which is desireable. One run of the CAA through the life graph consists of entering the life graph at the input and leaving it at any output. During the learning process consisting of num-ber of runs, CAA will chose a path through the life graph which leads to the output which gives sense of pleasure, Table I shows the results of one of the number of the experiments.

run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
steps	9	4	21	17	19	75	23	22	7	19	32	78	12	17	8	12	8	8
result	-	-	+	+	+	+	+	-	-	+	+	+	+	+	+	+	+	+

After 17 iterations (runs) the CAA exhibits deterministic behaviour. It finds the desireable exit in 8 steps.(The sign "-" in the table above shows that the exit was not associated with pleasure).

The CAA shows that a self-learning system, i.e. the system which receives no information from the environment besides the encountered situation (no teacher of any kind) can be effectively designed basing on two main assumptions: 1) some genetically knowledge 2) some kind of secondary reinforcement. This work also emphasizes the importance of the secondary reinforcement in the theory of self organization.

87

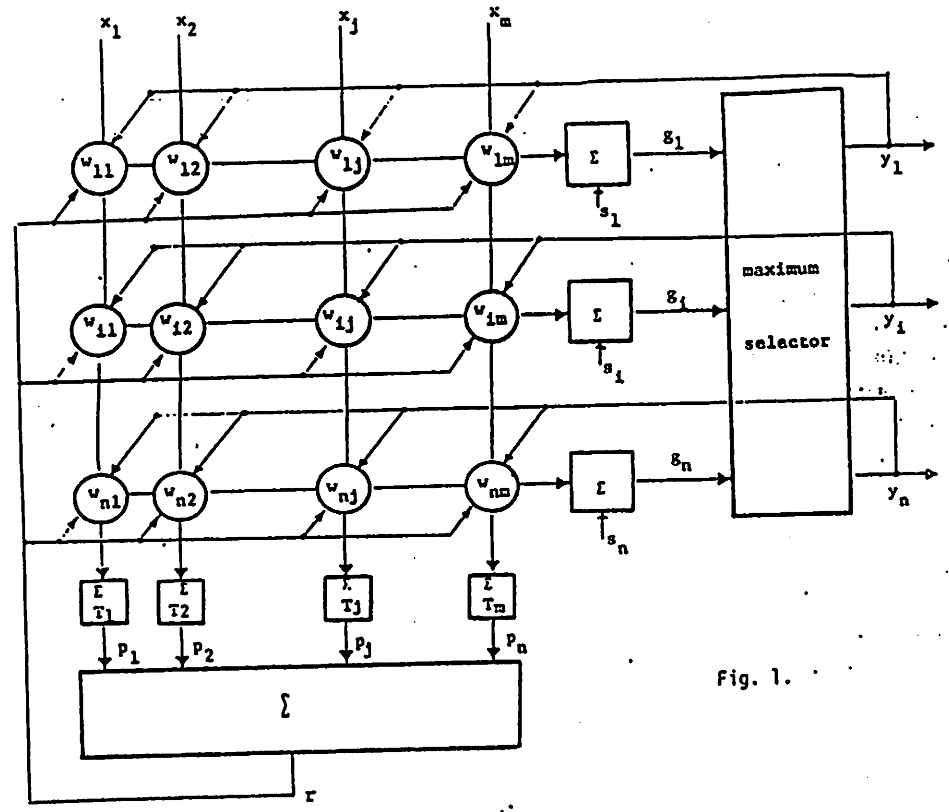


Fig. 1.

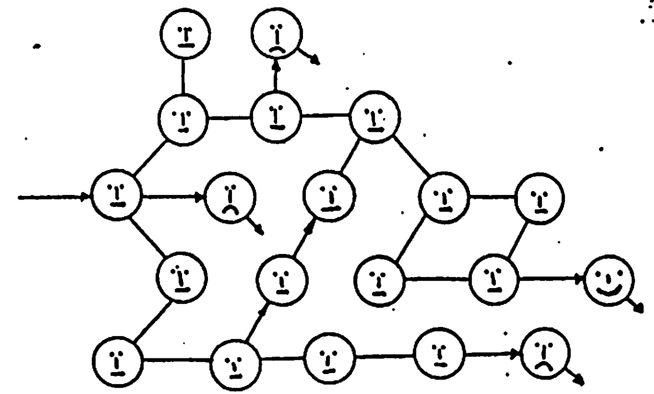


Fig. 2

Appendix E

Early papers by the Adaptive Networks Group, 1981-1983

1. A.G. Barto and R.S. Sutton. Landmark learning: An illustration of associative search. *Biological Cybernetics, 42:* 1-8, 1981

2. A.G. Barto, R.S. Sutton and P.S. Brouwer. Associative Search Network: A reinforcement learning associative memory. *Biological Cybernetics 40:* 201-211, 1981

3. S. Bozinovski. Teaching space: A representation concept for adaptive pattern classification. COINS Technical Report 81-28, University of Massachusetts, 1981

4. R.S. Sutton and A.G. Barto. An adaptive network that constructs and uses an internal model of its environment. *Cognition and Brain Theory, 4:* 217-246, 1981

5. R.S. Sutton and A.G. Barto. Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review 88*: 135-171

6. C.W. Anderson: Feature generation and selection by a layered network of reinforcement learning elements: Some initial experiments. COINS Technical Report 82-12, University of Massachusetts, 1982

7. A.G. Barto, C.W. Anderson and R.S. Sutton. Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics 43:* 175-185, 1982

8. A.G. Barto, and R.S. Sutton. Simulation of anticipatory responses in classical conditioning by a neuronlike adaptive element. *Behavioral Brain Research 4:* 221-235, 1982

9. S. Bozinovski. A self-learning system using secondary reinforcement. In R. Trappl (Ed.) *Cybernetics and Systems Research*: 397-402, North-Holland Publishing Company, 1982

10. A.G. Barto and R.S. Sutton. Neural problem solving. COINS Technical Report 83-03, University of massachusetts, 1983

11. S. Bozinovski and C. Anderson. Associative memory as controller of an unstable system: Simulation of a learning control. *Proceedings on the IEEE Mediteranean Electrotechnical Conference,* C5.11., Athens, May 1983

12. A.G. Barto, R.S. Sutton and C.W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics 13:* 835-846, 1983

REFERENCES

Amari S-I. "A mathematical approach to neural systems" In J.Metzler (Ed.) *Systems Neuroscience,* pp. 67-117, Academic Press, 1977

Arbib M.A., Lieblich I. "Motivational learning of spatial behavior" In J.Metzler (Ed.) *Systems Neuroscience,* pp. 221-239, Academic Press, 1977

Baird L., Klopf H. "A hierarchical network of provably optimal learning control systems: Extensions of the Associative Control Process (ACP) Network" *Adaptive Behavior 1,* 1993

Barto A., Sutton R., Brouwer P. "Associative Search Network: A reinforcement lerning associative memory" *Biological Cybernetics 40:* 201-211, 1981a

Barto A., Sutton R. "Landmark learning: An illustration of associative search" *Biological Cybernetics 42:* 1-8, 1981b

Barto A., Sutton R. "Goal seeking components for adaptive intelligence:An initial assessment" *Air Force Wright Aeronautical Laboratories, Avionics Laboratory Technical Report AFWAL-TR-81-1070,* Wright-Patterson AFB, Ohio, 1981c

Barto A., Sutton R., Anderson C. "Neuronlike elements that can solve difficult learning control problems" *IEEE Trans. Systems, Man, and Cybernetics 13:* 834-846, 1983

Barto A., Sutton R., Watkins C. "Learning and sequential decision making" In M.Gabriel, J. Moore (Eds.) *Learning and Computational Neuroscience: Foundations of Adaptive Networks"* MIT Press, pp. 539-602, 1990

Barto A. "Some learning tasks from a control perspective" In L.Nadel & D. Stein (Eds.) *Lectures in Complex Systems,* Addison-Wesley, 1991

Barto A. "Reinforcement learning and adaptive critic methods" In White D. & Sofge D. (Eds.) *Handbook of Intelligent Control,* Van Nostrand Reinhold, 1992

Barto A., Bradtke S., Singh S., "Learning to act using real-time dynamic programming", *Artificial Intelligence 72:* 81-138, 1995a

Barto A. "Reinforcement learning and dynamic programming" *Proc IFAC Man-Machine Systems Conference*, 1995b

Barto A., Sutton S. "Reinforcement Learning", Lecture Notes 791N, University of Massachusetts, Amherst, 1995

Bower G., Hilgard E. *Theories of Learning,* Prentice-Hall, 1981

Bozinovska L., Kovacev V., Nikodijevic O., Lazarevska M., Bozinovski S. "Learning experiments in computer controlled double-T maze" In M.Bajic (Ed.) *Neuron, Brain, and Behaviour* pp. 31-34, Pergamon Press, 1988

Bozinovski S. "Percetrons: Training for pattern classification" (In Croatian) Report for Student Research Contest, University of Zagreb, 1972a

Bozinovski S., Mesaric V. "Neural simulation of the conditioned reflexes" (In Croatian) Student research report, Electrical Engineering Department, University of Zagreb, 1972b

Bozinovski S. "Electrinic neural models" (In Croatian) Undergraduate Thesis, Electrical Engineering Department, University of Zagreb, 1972c

Bozinovski S. "Perceptrons and possibility of simulation of the teaching process" (In Croatian) Masters Thesis, University of Zagreb, Electrical Engineering Department, 1974

Bozinovski S., Fulgosi A. "The influence of pattern similarity ond the transfer of training upon the perceptron training" (In Croatian) Proc Symp Informatika, Bled, 1976

Bozinovski S., Bozinovska L. "Statistical properties of genetic languages and their role as evolutionary parameters" (In Macedonian) *Proc. Symp. Informatica,* 3.124, Bled, 1977a

Bozinovski S. "Normal training strategy in the process of pair-association learning in the case teacher:human - learner:machine" (In Croatian) Proc Conf ETAN, Banja Luka, 1977b

Bozinovski S. "Learning experiments with non biological systems" (In Macedonian) Proc Conf ETAN, Zadar, 1978

Bozinovski S. "A step toward the theory of instructing systems" *Unpublished report,* COINS, UMass, Amherst, 1981a

Bozinovski S. "The influence of pattern similarity upon the adaptive pattern classification teaching processes" (In Croatian) *PhD Thesis*, sent to University of Zagreb, Amherst, 1981b

Bozinovski S. "Teaching space: A representation concept for adaptive pattern classification" *COINS Technical Report, 81-28*, University of Massachusetts at Amherst, 1981c

Bozinovski S. "A model of adaptive detection of the surprising information" Unpublished report, COINS, UMass, 1981d

Bozinovski S., Chumbley J., "A measure of similarity based on T-matching functions", *Unpublished report,* COINS and Psychology Department, UMass, Amherst, 1981e

Bozinovski S. "A measure of similarity derived from the physiology of a pattern recognition system acting in a fuzzy world" *Unpublished report,* COINS, UMASS, Amherst, 1981f

Bozinovski S. "Inverted pendulum learning control" *ANW Memo*, December 10, Computer Science Department, University of Massachusetts, Amherst, 1981g

Bozinovski S. "A self-learning system using secondary reinforcement" *Published Abstracts of the Sixth European Meeting on Cybernetics and Systems,* Vienna, April 1982a

Bozinovski S. "A self-learning system using secondary reinforcement" In R. Trappl (Ed.) *Cybernetics and Systems,* North Holland, 1982b

Bozinovski S., Anderson C. "Associative memory as a controler of an unstable system: Simulation of a learning control" *Proc. IEEE Mediteranean Electrotechnical Conference, C5.11*, Athens, May 1983

Bozinovski S. "Adaptation and training: A viewpoint" *Automatika 26:* 137-144, Zagreb, 1985a

Bozinovski S., Cundeva K. "Linguistic properties of the training languages" *Automatica 26:* 152-158, 1985b

Bozinovski S., Spasovski K. "Control of dynamic system using controller which can learn to control" (In Macedonian) *Unpublished Technical Report,* Electrical Engineering Faculty, University of Skopje, 1985

Bozinovski S. "Implementation of multi-agent cooperation in programming for robot control" In P. Gabko, P. Kopacek, M. Voicu (Eds.) *Proc. Workshop on Computer Scinece Topics for Control Engineering Education,* Vienna, 1993

Bozinovski S. "Parallel programming for mobile robot control: Agent based approach" *Proc. Conf. Distributed Computing Systems,* IEEE Computer Society Press, pp 222-228, 1994

Bozinovski S., Beochanin D. "Pole balancing using neural network optimized by genetic algorithms" (In Macedonian) *Technical Report,* Laboratory for Intelligent Machines and Bioinformation Systems, LIMBIS 3, 1994, Electrical Engineering Department, University of Skopje, 1994

Bozinovski S. *The Artificial Intelligence,* (In Macedonian), Gocmar, 1994

Bozinovski S. "Experiments with CAA agent: Using entropy in the Markov Decision Model" ANW Memo, March 8, 1995

Bradtke S., Barto A., Ydstie E. "A reinforcement learning method for direct adaptive linear quadratic control" *Proc Eight Yale Workshop on Adaptive and Learning Systems,* pp. 85-90, 1994

Bush R., Mosteller F. *Stochastic Models of Learning,* John Wiley, 1955

Cannon R. *Dynamics of Physical Systems,* McGraw Hill, 1967

Dean T., Wellman M., *Planning and Control,* Morgan Kaufmann, 1991

Denardo E. "Contraction mappings in the theory underlying dynamic programming" *SIAM Review 9(2):* 165-177, 1967

Dietrich T., Flann N. "Explanation-based learning and reinforcement learning: An unified view" Unpublished report, 1995

Duda R., Hart P., *Pattern Recognition and Scene Analysis,* Willey Interscience, 1973

Eastwood E. "Control theory and the engineer" Proceedings IEE, 115, No 1, January 1968

Gambardella L., Dorigo M. "Ant-Q: A reinforcement learning approach to the traveling salesman problem" *Proc XII Intl Conference on Machine Learning,* Tahoe City, pp. 252-260, 1995

Grossberg S. "Some nonlinear networks capable of learning a spatial pattern of arbitrary complexity" *Proceedings of the National Academy of Sciences 59,* 368-372, 1968

Grossberg S. "Competitive learning: From interactive activation to adaptive resonance" *Cognitive Science 11:* 23-63, 1987

Heger M. "Consideration of risk in reinforcemet learning", *Machine Learning: Proceedings of the 11th International Conference,* pp. 105-111. Morgan Kaufmann Publishers, Inc. San Francisco, CA, 1994

Hinton G.E., Sejnowski T.J., Ackley D. *Boltzman Machines: Constraint Satisfaction Networks that Learn.* Tech. Tep. CMU CS 84, 111, Carnegie-Mellon University, Pittsburg, 1984

Hinton G.E., Sejnovski T.J. "Learning and relearning in Boltzman machines" in Rumelhart D. and McClelland J. *Parallel Distributed Processing,* MIT Press, 1986

Hinton G., Nowlan S. "How Learning can guide evolution" *Complex Systems 1,* 1987

Holland J. *Adaptation in Natural and Artificial Systems,* The University of Michigan Press, 1975

Hopfield J.J. "Neural networks and physical systems with emergent collective properties" *Proc. Nat. Acad. Sci. USA, 79:* 2554-8, 1982

Kaelbeing L. *Learning in Embedded Systems*, MIT Press, 1993

Keerthi S., Ravindan B. "A tutorial survey of reinforcement learning" Sadhana (Proceedings of the Indian Academy of Sciences) 19:_ 851-889, 1994

Kemeny J., Snell L. *Finite Markov Chains*, Van Nostrand, 1960

Kilmer W.L., McCulloch W.S., Blum J. "A model of the vertebrate central command system" *International Journal of Man-Machine Studies 1:* 279-309, 1969

Kilmer W. "Trainable recurrent nets of neural modules for computing concensus" *Proc. Seventh Yale Workshop on Adaptive and Learning Systems*, Yale University, 1992

Klopf H. "Brain functioning and adaptive systems" *Air Force Technical Report*, AFCRL 72-0164, 1972

Klopf H., Morgan J., Weaver S. "A hierarchical network of control systems that learn: Modeling nervous system function during classical and instrumental conditioning" *Adaptive Behavior 1*, 1993

Kimura H., Yamamura M., Kobayashi S. "Reinforcement learning by stochastic hillclimbing on discounted reward" *Proc XII Intl Conf on Machine Learning*, pp. 295-303, Tachoe City, 1995

Kohonen T. "An adaptive associative memory principle" *IEEE Trans. on Computers c-23:* 444-445, 1974

Lawton D. Personal communication, 1981

Lin L-J. "Self-improving reactive agents based on reinforcement learning, planning, and teaching" *Machine Learning 8*, 1992

Lin L-J. "Scaling up reinforcement learning for robot control" *Proc Tenth Int Conf on Machine Learning*, Amherst, pp. 182-185, 1993

Littman M., Cassandra A., Kaelbling L. "Learning policies for partially observable environments: Scalling up" *Proc XII Int'l Conf on Machine Learning*, pp. 362-370, 1995

McCulloch W. and Pitts W. "A logical calculus of the ideas immanent in nervous activity" *Bulletin of Mathematical Biophysics 5*: 115-133, 1944

McClelland J.L., Rumelhart D.E. *Explorations in Parallel Distributed Processing*, MIT Press, Cambridge, 1988

Michie D., Chambers R. "BOXES: An experiment in adaptive control" In E.Dale, D. Michie (Eds.) Edinburgh: Oliver and Boyd, 137-152, 1968

Minsky M. Steps toward artificial intelligence, *Proceedings of the IRE*, pp. 8-30, 1961

Minsky M.L. Theory of neural-analog reinforcement systems and its application to the brain-model problem. *Ph. D. Dissertation,* Princeton University, 1954

Minsky M., Papert S. *Perceptrons* MIT Press, 1969

Moore A., Atkeson C. "Prioritized sweeping:reinforcement learning with less data and less time", Machine Learning, 1993

Moore A. "Variable resolution reinforcement learning" *Proc Eight Yale Workshop on Adaptive and Learning Systems,* pp. 102-107, 1994

Morgan J., Patterson E., Klopf H. "Drive-reinforcement learning: a self-supervised model for adaptive control" *Network 1,* 1990

Najdovski B., Bozinovski S. "Self-learning control of inverted pendulum using neural network" (In Macedonian) *Proc Conf ETAI,* pp. 588-596, 1989

Newel A., Shaw J., Simon H. "A variety of intelligent learning in a general problem-solver" In M. Yovits, S. Cameron (Eds.) *Self-Organizing Systems,* Pergamon Press, pp. 153-189, New York, 1960

Pavlov I.P. *Conditioned reflexes,* Oxford University Press, 1927

Rescorla R.A., Wagner A.R. "A theory of Pavlovian conditioning: Variations of the effectiveness of reinforcement and non-reinforcement" In Blake A. & Procasy C. (Eds.) *Classical Conditioning II: Current Research and Theory.* Appleton Century Croffts, 1972

Rosenblatt F. "The perceptron: a probabilistic model for information storage and organization in the brain" *Psychological Review 65: 386-408, 1958*

Rosenblatt F. *Principles of Neurodynamics,* Spartan Book, 1962

Rumery G., Niranjan M. "On-line Q-learning using connectionist systems" *Technical Report CUED/F-INFENG/TR 166,* Cambridge University Engineering Department, 1994

Russel D., Rees S. "System control - a case study of a statistical learning automaton" In *Progress in Cybernetics and Systems Research,* Hemisphere Publishing Corporation, 1975

Russel S., Wefard E. "Principles of metareasoning" *Artificial Intelligence 49,* 1991

Samuel A. "Some studies in machine learning using the game of checkers" IBM Journal of Research and Development, 1959

Selfridge O. "Pandemonium - A paradigm of learning" HMSO Symposium on Thought Processes, 1959

Shannon C.E. "A mathematical theory of communication" *Bell Systems technical Journal 27:* 379-423, 1948

Shannon C.E. "Prediction and entropy of printed English" *Bell System Technical Journal,* 1951

Simpson P. *Artificial Neural Systems.* Pergamon Press, 1990

Singh S. "Learning to solve Markovian decision processes" CMPSCI Technical Report 93-77, Computer Science Department, University of Massachusetts, Amherst, 1993

Slagle J. *Artificial Intelligence: The Heuristic Programming Approach,* McGraw-Hill, 1971

Spinelli N. "OCCAM, A computer model for a content addressible memory in the central nervous system" In K. Pribram, D. Broadbent (eds.) *Biology of Memory,* pp. 293-306, 1970

Strain E.R. "Establishment of an avoidance gradient under latent learning conditions" *Journal of Experimental Psychology 46:* 391-399, 1953

Sutton R. "Temporal credit assignment in reinforcement learning" *Ph.D. Thesis*, University of Massachusetts, Amherst, MA, 1984

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

Sutton R. "Integrated architectures for learning, planning, and reacting based on approximate dynamic programming" In Porter and Mason (Eds.) *Machine Learning: Proceedings of the Seventh International Conference*, pp. 216-224, Morgan Kaufmann, 1990

Sutton R., Barto A., Wiliams R. "Reinforcement learning is direct adaptive optimal control" *Proceedings of the American Control Conference,* Boston, pp. 2143-2146, 1991

Schwartz A. "A reinforcement learning method for maximizing undiscounted rewards" *Proc. Tenth Int Conf on Machine Learning,* Amherst, pp. 298-302, 1993

Tash J., Russel S. "Control strategies for a stochastic planner" *Proc. Sixth National Conference on Artificial Intelligence,* Seatle, 1987

Tatman J., Shachtar R. "Dynamic programming and influence diagrams" *IEEE Trans. Systems, Man, and Cybernetics, 20,* 1990

Uttley A., *Information Transmittion in Nervous Systems,* Academic Press, 1979

Watkins C. "Learning from delayed rewards", *Ph. D. Thesis*, Cambridge University, Cambridge, England, 1989

Watkins C., Dayan P. "Q-learning" *Machine Learning 8,* 279-292, 1992

White D.S. *Markov Decision Processes,* John Willey and Sons, 1993

Whitehead S., Lin L-J. "Reinforcement learning of non-Markov decision processes" *Artificial Intelligence 73:* 271-306, 1995

Widrow B., Hoff G. "Adaptive switching circuits" Stanford Technical Report 1553-1-1968

Widrow B., Smith F. "Patern-recognizing control systems" In J. Tou and_ R. Wilcox (Eds.) *Computer and Information Sciences,* Spartan Books, 1964

Witten I. "An adaptive optimal controler for discrete-time Markov environments" *Information and Control 34*: 286-295, 1977

Zilberstein S. "Operational rationality through compilation of anytime algorithms" *AI Magazine,* 79-80, Summer 1995

Zilberstein S., Russel S. "Optimal composition of real-time systems" *Artificial Intelligence,* 1995