An Architecture for Representing and Learning Behaviors by Trial and Error

Pascal BLANCHET
CRIN-CNRS / INRIA Lorraine
BP 239, 54506 VANDOEUVRE-les-NANCY Cedex, FRANCE
E-mail: blanchet@loria.fr

Abstract

We propose an architecture for representing and learning behaviors by trial and error. This architecture is a network of automata which is dynamically built while the system evolves in its environment. It is presented as an alternative to rule based systems which learn through temporal differences algorithms. This system learns sequences of perceptions instead of perception-action rules. These sequences represent rules of the environment, and can also be considered as production rules. The learned sequences are used for learning other sequences. An induction mechanism is applied to produce general sequences from the different trials of the system. Different experiments have been carried out, including a block manipulation task which is described at the end of this paper.

1. Introduction

The question we are interested in, is the one of learning through interaction with the external world in a way that is as close as possible to human's one. In this paper, we focus on the following problem: given an environment with fixed rules, some goals, an input device, and a set of actions, how can a system learn by trying actions and receiving a positive feedback only when it reaches one of its goals?

The difficulty of such a problem is that the actions of the system are not always followed by an external feedback indicating whether it was good or not. The system must learn to estimate the utility of its actions from the feedback received occasionally, when a goal is achieved. This problem has been addressed with encouraging performances through Temporal Differences algorithms (TD) [11][12]. But these algorithms are limited to the task of learning reactive behaviors in markovian environments (i.e. environments where each state transition entirely depends on the current state). The problem of learning more complex tasks, through the same kind of interactions, remains unsolved.

A well known approach to this problem is the classifier system approach (CS) proposed by Holland [5]. CS are production systems which can support both knowledge representation [2], and rule acquisition through TD algorithms combined with genetic algorithms (GA). The utility of the rules created by GA is evaluated by the bucket brigade algorithm. Theoretically, any representation could be built by GA. But practically, problems occur when one tries to make CS learn a task which needs a short term memory of two consecutive states [10].

Despite the problems encountered by CS, it seems that a system capable of learning in real-world situations should also have the capability of learning through TD-like algorithms together with the capability of representing complex knowledge. But, of course, many other characteristics should be added to this same system to make it really work. Our work is an attempt to develop an architecture which must satisfy a set of competences that we think necessary for an animat. The competences we have taken into account are the following ones:

1) Learning rules with as few trials as possible: we suppose that in any environment there may exist many rules which can be learned by simple memorization and induction algorithms. These ones must be learned with few trials.

2) Consuming as few memory as possible: When a rule is never used it must be removed. When a task has been completely learned, only the useful rules must remain in memory.

3) Learning the effect of actions on the environment together with their utility: When the effects of the rules are known, they can be used for planning tasks.

4) Learning more than reactions to the current state: sometimes the condition which ensures the success of an action depends on a short sequence of several consecutive states instead of the single previous state.

The reasons for these choices can be discussed by comparison with other related approaches.

Several works only focus on the problem of learning the effect of actions on the environment, such as the connectionist system SLUG proposed by Mozer and Bachrach [8]. Other systems, such as the one studied by Moore and Atkinson [7], can use what they have learned about the effect of their actions, in order to chose the best possible action in each state. For each triple \((i,a,j)\), the system learns the probability that state \(i\) is followed by state \(j\) after the execution of
action $a$. This knowledge provides a way to compute the utility of each state $i$. As a matter of fact, the utility is related to the probability of reaching a goal state $g$ with a sequence of actions starting from $i$.

In the previous work, the system learns rules concerning transitions between states of a finite-state environment. But in real world problems any single state may be characterized by too numerous parameters, and the number of possible states can be considered as infinite compared to the memory capacity of our machines. An animat should be able to learn general rules concerning transitions between large classes of external states. Besides, in order to make the best use of its limited memory, it should also be able to select what is important to be learned and what can be neglected.

A system which consumes little memory in regard to the impressive results it obtains in the field of TD learning, is the one proposed by Tesauro [11] for learning to play backgammon. It is a neural network which learns to associate each state of the game to the probability of winning when this state has been observed. This system has the important generalization capabilities of perceptrons. Classes of states can be associated to the same probability.

The constraints in game playing problems are not the same as the ones encountered by animats in natural environments. One difference is that the goals of an animat may change. In a game, the final goal is fixed once for all. Therefore, it seems important for an animat to be able to use what has been learned for a particular goal, to solve other goals. This is not possible when the system only learns to associate states to the probability of reaching a fixed goal state. It seems more adequate to learn the effect of an action, then to decide if this effect is good with regard to the current goal.

Our work is also related to several other approaches, whose common goal is to explore the possibilities of representing and learning knowledge with connectionist architectures using complex automata [1][3][9]. Most of these works are inspired from the biological model proposed by Y. Burnod [6]. In this model the cortical column is presented as a complex automaton which relates perceptions to actions and goals. It is also the role of the automaton we have used in our system.

The following sections describe the architecture and the learning algorithms that we have used. The system has been applied to several learning problems, one of which is described at the end of this paper. This experiment is a block manipulation task which is similar to the one studied by Whitehead and Ballard [13].

2. Sequences of perceptions vs condition-action rules

We explain here the kind of representation we have used to take into account the following constraints:

- the same structure represents prediction rules together with production rules,
- the condition parts of rules can be sequences of states,
- rules concern classes of states instead of single states.

2.1 Perceptions and actions

Let $S$ be the set of all possible states of the environment where the system evolves. The time dimension is a discrete sequence of instants. The state of the environment at time $t$ is given by $s(t) \in S$.

Let $F$ be the set of binary feature-detectors which constitutes the perceptual device of the system. Each detector of $F$ represents a particular property of the environment, it takes the value $1$ (active) when the feature is observed, and $0$ (inactive) otherwise. $P(S)$ and $P(F)$ are the sets of parts of $S$ and $F$. $f(t)$ is the set of feature detectors activated at time $t$. Each element of $P(F)$ can be related to an element of $P(S)$. For each $f_i \in P(F)$, we define $E(f_i) \in P(S)$ as the set of external states which activate every detectors of $f_i$. The application $E$ satisfies the following rule:

$$f_1 \subseteq f_2 \Rightarrow E(f_2) \subseteq E(f_1)$$

We call $A$ the set of actions that the system can execute in order to modify the current state of the environment. We consider that the execution of an action is also a property of the external environment. In order to detect the execution of an action $a \in A$, $F$ contains an action-detector $d_a$ which takes the value $1$ at time $t+1$ if and only if $a$ is executed at time $t$. There is one action-detector for each action in $A$.

2.2 Rule representation

A production (and prediction) rule $r$ is represented by a couple of sets of feature-detectors:

$$r = (f_1, f_2),$$

where $(f_1, f_2) \in P(F) \times P(F)$. $f_1$ is the condition part of the rule. It is satisfied when the state of the environment belongs to $E(f_1)$. If $f_2$ contains one action-detector $d_a$ ($f_2$ could contain several action-detectors but we have only worked with rules containing one action), then the action part of the rule is $a$, the action which is detected by $d_a$. We say that $a$ is the action of $f_2$. If there is no action-detector in $f_2$, the action of $f_2$ is the empty action, the action which has no effect on the environment.

When $(f_1, f_2)$ is interpreted as a production rule, it corresponds to the rule:

$$E(f_1) \rightarrow a$$

which means that $a$ must be executed at time $t$ if $s(t) \in E(f_1)$.

When it is interpreted as a prediction rule, it corresponds
to the rule:

\[ E(f_1) \rightarrow_a E(f_2) \]

which means that if \( s(t) \in E(f_1) \) then \( s(t+1) \in E(f_2) \) if \( a \) is executed at time \( t \). In other words, the effect of action \( a \) when every detectors of \( f_1 \) are activated, is the activation of every detectors of \( f_2 \). (Though the execution of \( a \) occurs at time \( t \), the activation of \( d_a \) occurs at \( t+1 \). As a matter of fact, \( d_a \) is one of the effects of \( a \).

2.3 Sequences of perceptions

Since a rule can be represented by a sequence of two perceptions, it seems natural to consider how longer sequences of perceptions, such as \( (f_1, ..., f_n) \), should be treated.

Let \( a_i \) be the action of \( f_i \). In term of production device, \( (f_1, ..., f_n) \) means that for each \( i, 1 \leq i \leq n, a_i \) must be executed at time \( t \) if for each \( j, 1 \leq j \leq i \), we have \( s(t-j) \in E(f_j) \). In fact, the sequence represents a set of production rules whose conditions are sequences of classes of states. If we used a more classical notation to represent these rules, the sequence \( (f_1, ..., f_n) \) could be represented by this set of rules:

\[
\begin{align*}
(E(f_1)) & \to_a a_2 \\
(E(f_1), E(f_2)) & \to_a a_3 \\
& \ldots \\
(E(f_1), E(f_2), ..., E(f_{n-1})) & \to_a a_n
\end{align*}
\]

where a condition \( E(f_j) \ldots E(f_j) \) defines the sequence of classes of states which must be observed before triggering the action.

In term of prediction device, \( (f_1, ..., f_n) \) means that if for each \( j, 1 \leq j \leq i \), we have \( s(t-j) \in E(f_j) \), and if \( a_i \) is executed at time \( t \), then we will have \( s(t) \in E(f_i) \). It defines a set of prediction rules:

\[
\begin{align*}
(E(f_1)) & \to a_2 E(f_2) \\
(E(f_1), E(f_2)) & \to a_3 E(f_3) \\
& \ldots \\
(E(f_1), E(f_2), ..., E(f_{n-1})) & \to a_n E(f_n)
\end{align*}
\]

It is clear that such sequences can be used to deal with non-markovian environment where the condition of a prediction may depend on a succession of several states. But they can be useful also in markovian environment, if the perceptual device of the system is incomplete, so that the perception of one state is not sufficient to predict the next one.

3 An overview of the learning strategies

3.1 Learning markovian decision tasks in a finite-state environment.

In such an environment, it is possible to learn for each triple \( (i, a, j) \in S \times A \times S \), the probability that \( s(t) = j \) when \( s(t-1) = i \) and action \( a \) is applied. The notation \( q^a_{ij} \) is used for this probability. Each \( q^a_{ij} \) can be estimated with the trials made by the system. Once the system has an estimation of each \( q^a_{ij} \), it can estimate the reward-to-go from each state \( i \) (i.e. the utility of state \( i \), if the reward associated to each goal state is known (Moore & Atkeson, 1993):

\[
J_i = r_i + \max_{a \in actions(i)} \gamma \times \sum_{j \in succ(i, a)} q^a_{ij} J_j
\]

where \( J_i \) is an estimation of the optimal reward-to-go from state \( i \), \( r_i \) is the reward associated to state \( i \) (for example 100 for a goal state and 0 for any other state), \( succ(i, a) \) is the set of states which has already been observed after using the action \( a \) in state \( i \). \( \gamma \) is a discount factor which is usually a little bit lower than 1. This formula gives at the same time the best action to use when the state \( i \) is observed: it is the one which gives the maximum value.

3.2 Utility of sequences of perceptions

In our approach, the value \( q^a_{ij} \) has no reality since the system can only identify classes of states through its set of feature detectors \( F \). The probabilities that the system will have to learn, concern transitions from an element of \( P(F) \) to another one, with a given action. Such a probability value must be associated to each element of a sequence. For the sequence \( r = (f_1, f_2, ..., f_n) \), we can define \( q^r \) as the probability to observe \( s(t) \in E(f_r) \) when for each \( j, 1 \leq j \leq i \), there is \( s(t-j) \in E(f_j) \) and when \( a_i \), the action of \( f_i \), is executed at time \( t-1 \). It is the probability of the prediction

\[
(E(f_1), E(f_2), ..., E(f_{n-1})) \rightarrow_a E(f_n)
\]

There exists too many sequences of perceptions to learn these probabilities for every one of them. The system has to chose which sequence must be treated.

Despite this limitation, the utility (the expected reward-to-go) of a class of states can be computed with almost the same formula, except this important difference: the summation over the successors of a sequence cannot be used, since these successors are not necessarily independent. Therefore, when there are several classes of successors, the utility should be approximated with the \( \min \) operator instead of the \( \Sigma \) operator. In fact we have treated the problem of multiple
successors in a particular way (as it is discussed in subsection 4.4).

Another difference concerns the discount factor and the reward associated to the goal states. We fix them to $I$, and we compute independently the expected reward-to-go and the distance of a state from the goal. There is no discount factor like in equation (1), which enables the utility value to take into account the distance to the goal together with the probability to reach it.

For example, if the expected reward-to-go in $E(f_n)$ is $J_n$, we simply compute the expected reward-to-go in $E(f_i)$ as:

$$J_i = q_i J_n$$

or

$$J_i = (n-i) + D_n$$

and if $D_n$ is the distance of $E(f_n)$ to a goal state (according to the experiments of the system), then

$$D_i = (n-i) + D_n$$

or

$$D_i = 1 + D_{i+1}$$

We have made this choice because it is important in our algorithm to make the difference between rules which always work and rules which often work. In a deterministic environment, the first ones are real knowledge about the environment, while the second ones must be considered as short term solution until more reliable knowledge is discovered.

In this approach, the rules that have a utility value below $I$ are essentially used to «guide» the system when it learns: they are used as counter-examples.

### 3.3 Induction of sequences

The system uses an induction mechanism in order to create general sequences from its experimentations. In supervised learning problems the systems have to build general representations of given concepts from subsets of examples. The simplest induction mechanism consists in eliminating the features which are not in every examples. The final representation of the concept only keeps the common features of all the examples. The same induction mechanism can be applied to build general sequences.

Let $r_1$ and $r_2$ be two sequences of length $n$. We define the generalization of $r_1$ and $r_2$ as the sequence $r_3$

$$r_1 = (f_{11}, f_{12}, ..., f_{1n}),$$

$$r_2 = (f_{21}, f_{22}, ..., f_{2n}),$$

$$r_3 = (f_{31}, f_{32}, ..., f_{3n})$$

with $f_{3i} = f_{1i} \cap f_{2i}$

The problem is to know which couples of sequences must be generalized. We cannot generalize $r_1$ with $r_2$, if their actions are different. It would create a sequence with an empty action part, which is generally less useful than two sequences with real actions.

We distinguish two reasons for applying generalization to sequences:

a) discovering the general effect of a given action in a given context:

given $f_1, ..., f_{i-1} \in P(F)$ and $a \in A$, find $f_i \in P(F)$ so that if $a$ is executed at time $t-I$ and $\forall j \in \{1, ..., i-1\}$, $s(t-j) \in E(f_j)$ then $s(t) \in E(f_i)$

b) discovering the minimal condition in which a given action produces a general effect:

given $f_i \in P(F)$, and $a \in A$, find $(f_1, ..., f_{i-1}) \in P(F)^{i-1}$ so that if $a$ is executed at time $t-I$ and $\forall j \in \{1, ..., i-1\}$, $s(t-j) \in E(f_j)$ then $s(t) \in E(f_i)$, and $\forall (h_1, h_2, ..., h_i, j)$ if $\exists k \in \{1, ..., i-1\}, h_k \subseteq f_k$, then $\exists t$ which verifies

$$s(t-j) \in E(h_k),$$

$$a \in \text{executed at time } t-I$$

$$s(t) \notin E(f_i)$$

If a solution exists to the question (a), it is

$$f_i = t \cap f(t)$$

where $Ta$ is a set of instants so that $t \in Ta$ if $a$ is executed at $t-I$ and $\forall j \in \{1, ..., i-1\}, s(t-j) \in E(f_j)$.

A similar method can be applied to find the solution of question (b) when this solution is unique, and when we know its maximal length (the length is the number of successive perceptions). If the maximal length is $i-I$, the solution is $(f_1, ..., f_i, j)$, where

$$f_j = t \cap f(t-j)$$

where $t \in Tb$ if $a$ is executed at time $t-I$ and $s(t) \in E(f_j)$.

In a goal directed learning problem, the question (b) is the more useful. It is necessary to know what is the minimal condition for a given action to produce a given goal or subgoal. The role of the induction mechanism we have used is to find this solution.

The problem of the length of the solution is treated in the following manner. The first hypothesis is that the length is $1$ (the first induction steps are applied to single perceptions). Then if no solution is discovered, the hypothetical length becomes $2$, and the system applies induction to sequences of two successive perceptions. The length can grow until a limit, fixed to $5$ in our experiments.
When several solutions exist for (b), this induction algorithm can produce over-generalized rules. Such rules may fail when the system tries them. Then, they are used as counter-examples, in order to prevent the system from creating other rules, more general than these counter-examples.

3.4 An overview of the learning algorithm

At the beginning, the system only knows how to recognize goal states. It contains a set of goal perceptions $g_1, g_2, ..., g_m \in P(F)$ so that $E(g_1) \cup E(g_2) \cup ... \cup E(g_m)$ is the set of goal states.

The system randomly tries its actions until a goal state is recognized. If $t_1$ is the instant at which this event occurs ($s(t_1) \in E(g_i)$), the system memorizes the sequence of perceptions

$$(f(t_1-1), f(t_1)).$$

This new sequence is associated to the goal $g_i$, which has just been recognized. The sequence is considered as a production rule by the system:

if $s(t) \in f(t_1-1)$, the action of $f(t_1-1)$ must be executed.

We say that this sequence is a method of the goal $g_i$. If this method is efficient (if its execution is often followed by the recognition of $g_i$) then $f(t_1-1)$ is considered as a new goal (a subgoal). It means that new methods are created and associated to it, each time $s(t) \in f(t_1-1)$.

The efficiency of each sequence is updated after each trial according to the success or failure of its action.

When a new sequence $r_2$ is about to be created for a goal $g$, the system checks if $g$ already has an efficient method $r_1$ which uses the same action as $r_2$. If it is true, a general sequence $r_3$ is created by the induction mechanism described in the previous section.

$r_3$ will be associated to $g$ as one of its methods. But before, the system checks if $g$ has not an inefficient method, more specific than $r_3$, and containing the same action. If such a sequence exists, $r_3$ is destroyed, since it cannot be efficient if a more specific sequence is not.

Several criteria control the memorization process, so that new sequences are created only when it is necessary. For example, if $r_3$ is efficient, the system will never associate a method $r_4$ to $g$, if $r_4$ is less general than $r_3$.

Because of this criterion, it occurs that, after several memorization and generalization steps, no more methods need to be associated to a given goal. The system stops learning for such an easy goal.

For a more difficult goal $g$ (if the system always needs to memorize new sequences), the length of the methods which are associated to $g$, is increased. For example, if the sequence $s(t_i) \in E(g)$, the sequence

$$(f(t_i-2), f(t_i-1), f(t_i))$$

is memorized and associated to $g$. This sequence defines two production rules, one of which takes into account two successive states of the environment.

In order to save memory, the old sequences which are never used are periodically removed to make room for new ones. The «age» of a sequence is the number of time steps from its creation, or from its last successful execution.

4 The associative architecture

4.1 The units

The sequences of perceptions contained in the memory of the system are represented by sequences of automata (or internal units). We call $U$ the set of internal units contained in the memory. A sequence of perceptions is represented by a sequence of units:

$$r = (c_1, c_2, ..., c_n)$$

where each $c_i$ is an internal unit which contains the following parameters:

- a goal flag $gf(c_i) \in \{ 0, 1 \}$
- a utility value $u(c_i, t) \in [0,1]$  
- a probability of success $p(c_i) \in [0,1]$  
- a number of success $nsuc(c_i) \in \mathbb{R}$  
- a number of trials $ntry(c_i) \in \mathbb{R}$  
- a distance to a goal $dist(c_i) \in \mathbb{R}$  
- an age $age(c_i) \in \mathbb{R}$  
- a learning number $nlearn(c_i) \in \mathbb{R}$  
- a learning length $llearn(c_i) \in \mathbb{R}$  
- an activity value $act(c_i, t) \in \{ 0, 1 \}$  
- a global activity value $gact(c_i, t) \in \{ 0, 1 \}$  
- a contextual activity $cont(c_i, t) \in \{ 0, 1 \}$

The feature detectors are also represented by units. But these ones are more simple. A feature detector $d \in P(F)$ has only an activity value $act(d, t) \in \{ 0, 1 \}$.

Each internal unit is connected to several other units and feature detectors. Each unit $c_i$ can be associated to

- a set of feature detectors $f(c_i) \in P(F)$
- a successor $succ(c_i) \in U$
- a predecessor $pred(c_i) \in U$
- a subgoal $sg(c_i) \in U$
- a set of methods $m(c_i) \subset U$
The sequence of perceptions represented by \( r \) is \( (f(c_1), \ldots, f(c_n)) \).

The goals of the system are represented by a set of units with a goal flag set to 1. If \( gf(c) = 1 \), \( c \) is a goal unit, and \( E(f(c)) \) is a set of goal states.

4.2 The links

The feature detectors are associated to internal units through a particular kind of links called **instantaneous links** (because they propagate the activity of the feature detectors instantaneously to the internal units).

In order to represent the sequences of perceptions, the internal units are linked together through **sequential links**. In the sequence \( r \), there is a sequential link between each couple \( (c_i, c_{i+1}) \), and we have

\[
\text{succ}(c_i) = c_{i+1} \\
\text{pred}(c_{i+1}) = c_i \\
\text{succ}(c_n) = \text{nil} \\
\text{pred}(c_1) = \text{nil}
\]

The last unit of each sequence is associated to a subgoal unit (a unit which has a maximal utility value, this definition includes goal units). The subgoal of \( c_n \), \( sg(c_n) \), is associated to \( c_n \) through a **hierarchical link**. Each subgoal may be associated to several units. The set of units which are associated to a subgoal \( g \) is called \( m(g) \). We have

\[
sg(c) = g \iff c \in m(g)
\]

A sequence which terminates by \( c \in m(g) \) is a method of \( g \).

The figure 1 illustrates the kind of network we can build with such constituents.

![Figure 1: A typical connectivity (for a matter of clarity, only the instantaneous links of \( c \) and \( g \) has been drawn).](image)

4.3 The algorithm

This section gives a step by step description of the algorithm.

0) The programmer gives a set of units \( g_1, g_2, \ldots, g_m \) so that \( E(f(g_1)) \cup E(f(g_2)) \cup \ldots \cup E(f(g_m)) \) is the set of goal states. No units are associated to the goal units \( \text{pred}(g_i) = \text{succ}(g_i) = \text{sg}(g_i) = \text{nil}, m(g_i) = \emptyset \). The goal flags of the goal units are set to 1 \( (gf(g_i) = 1) \). The other parameters of \( g_i \) are initialized as follows: \( p(g_i) = 0 \), \( nlearn(g_i) = 0 \), \( llearn(g_i) = 2 \). The remaining parameters are computed while the system is running.

The programmer also defines the size of the memory (the maximum number of units, \( \text{MaxUnits} \)), and the age limit after which a unit must be destroyed \( (\text{AgeLimit}) \).

1) Compute the activity values of every feature detector, according to the current state of the environment.

2) Compute the different activity values of each internal unit \( c \)

\[
\text{act}(c, t) = \min_{d \in f(c)} (\text{act}(d, t)) \\
\text{gact}(c, t) = \text{cont}(c, t-1) \times \text{act}(c, t) \\
\text{cont}(c, t) = gact(\text{pred}(c), t) \\
\text{with} \quad gact(\text{nil}, t) = 1 \quad \text{(by convention)}
\]

3) Increase the age of each unit, and clear the memory

\[
\text{if} \quad gf(c) = 0 \quad \text{then} \quad \text{age}(c) = \text{age}(c) + 1 \\
\text{if} \quad \text{gact}(c_{\text{best}}, t) = 1 \quad \text{then} \quad \text{age}(c_{\text{best}}) = 0 \\
\text{if} \quad \text{age}(c) > \text{AgeLimit} \quad \text{then destroy} \ c \\
\text{if} \quad \text{Card}(U) > \text{MaxUnits} \quad \text{then destroy the oldest units so that Card}(U) \leq \text{MaxUnits}.
\]

4) Evaluate the unit \( c_{\text{best}} \) (\( c_{\text{best}} = \text{nil} \) at the beginning)

\[
\text{ntry}(c_{\text{best}}) = 1 + \text{ntry}(c_{\text{best}}) \\
\text{nsuc}(c_{\text{best}}) = \text{nsuc}(c_{\text{best}}) + \text{gact}(c_{\text{best}}, t) \\
\text{p}(c_{\text{best}}) = \text{nsuc}(c_{\text{best}}) / \text{ntry}(c_{\text{best}})
\]

5) Compute the utility values of each units:

\[
\text{if} \quad gf(c) = 1 \quad \text{then} \quad u(c, t) = 1 \\
\text{else if} \quad \text{succ}(c) = \text{nil} \quad \text{then} \quad u(c, t) = u(\text{sg}(c)) \times (1-\text{gact}(\text{sg}(c), t)) \\
\text{else} \quad u(c) = p(\text{succ}(c)) \times u(\text{succ}(c)) \times (1-\text{gact}(\text{succ}(c), t))
\]

6) Compute the distance to the goal:

\[
\text{if} \quad gf(c) = 1 \quad \text{then} \quad \text{dist}(c) = 0 \\
\text{else if} \quad \text{succ}(c) = \text{nil} \quad \text{then} \quad \text{dist}(c) = \text{dist}(\text{sg}(c))
\]
else \( \text{dist}(c) = 1 + \text{dist}(\text{succ}(c)) \)

7) Choose the best unit (\( c_{\text{best}} \)):

Let \( U_{\text{max}} \) be the set of units which verify:

\[
\forall c \in U_{\text{max}} \iff \text{cont}(c,t) \times u(c,t) \times p(c) = \max_{b \in U_{\text{max}}} (\text{cont}(b,t) \times u(b,t) \times p(b))
\]

The set of units which can be executed is \( U_{\text{best}} \), which is given by:

\[
c \in U_{\text{best}} \iff c \in U_{\text{max}} \text{ and } \text{dist}(c) = \min_{b \in U_{\text{max}}} (\text{dist}(b))
\]

The system chooses \( c_{\text{best}} \) in \( U_{\text{best}} \), so that the condition of \( c_{\text{best}} \) is the most general as possible (the number of feature detectors associated to its predecessors is minimal). Then a random value \( r_{\text{v}} \) is taken in \([0,1]\). If \( u(c_{\text{best}}) \times p(c_{\text{best}}) < r_{\text{v}} \), then \( c_{\text{best}} = \text{nil} \) (the choice is rejected, and a random action will be executed).

8) Learn new sequences

Let \( U_{\text{learn}} \) be the set of units to which new methods will be associated:

\[
c \in U_{\text{learn}} \text{ if and only if } \begin{cases} g(c) = 0 & \text{if } (\text{succ}(c) = c_{\text{best}} \text{ and succ}(c) \neq \text{nil}) \\ u(c,t) = 1 & \text{if } \text{cont}(c,t) \neq 0 \end{cases}
\]

for each \( c \in U_{\text{learn}} \)

\[
\{ \text{create a set } S_{\text{new}} \text{ of new sequences} \}
\]

\[
S_{\text{new}} = \emptyset
\]

for \( i = 2 \) to \( \text{llearn}(c) \)

create a sequence of units \( (c_{1}, ..., c_{i}) \)

so that \( f(c_{j}) = f(t-(i-j)) \)

and add it to \( S_{\text{new}} \)

for each \( b \) in \( m(c) \)

if \( p(b) = 1 \) and if the action of \( f(b) \) is the same as the action of \( f(c_{j}) \) then create the sequence

\[
h = (h_{1}, ..., h_{j}) \text{ with } f(h_{j}) = f(c_{j}) \land f(\text{pred}^{j-i}(b))
\]

and add \( h \) to \( S_{\text{new}} \).

\[
(\text{if } \text{pred}^{j-i}(b) = \text{nil} \text{ then } h_{j} = \text{nil})
\]

end \{ for each \( b \) \}

end \{ for \( i \) \}

\[
\{ \text{associate the useful sequences of } S_{\text{new}} \text{ to } m(c) \}
\]

for each \( r \) in \( S_{\text{new}} \)

\[
r = (r_{j}, ..., r_{k})
\]

if \( \exists b \) in \( m(c) \) so that

- \( p(b) = 1 \)

or if \( \exists b \) in \( m(c) \) so that

- \( p(b) < 1 \)

- \( \forall i \in \{1, ..., k\} , f(r_{i}) \subseteq f(\text{pred}^{k-i}(b)) \)

then destroy \( r \)

else add \( r_{j} \) to \( m(c) \)

end \{ for each \( r \) \}

if \( \text{nlearn}(c) = \text{nlearn}(c) + 1 \)

\[
\text{nlearn}(c) = \text{nlearn}(c) < 6
\]

if \( \text{nlearn}(c) = 3 \times \text{ilearn}(c) \) and \( \text{nlearn}(c) < 6 \) then

\[
\text{nlearn}(c) = 0, \text{ilearn}(c) = 1 + \text{llearn}(c)
\]

end \{ for each \( c \) \}

Each unit \( r_{j} \) of each new sequence \( r \) is initialized with the following values:

\[
\begin{align*}
& p(r_{j}) = 1, \text{gact}(r_{j}) = 0, \text{llearn}(r_{j}) = 2, \\
& \text{nlearn}(r_{j}) = 0, \text{age}(r_{j}) = 0.
\end{align*}
\]

9) Execute the action

\[
\text{if } c_{\text{best}} = \text{nil}, \text{execute a random action, }
\]

else

\[
\text{if } \exists d_{a} \text{ an action-detector so that } d_{a} \in f(c_{\text{best}}) \text{ then execute the action corresponding to } d_{a},
\]

10) update the state of the environment according to the executed action, and return to step 1.

4.4 Improvement

As it is described here, the system could not learn a problem where the same action in the same condition may have different benefic effects. As a matter of fact, each unit represents the effect of its execution: the set of feature detectors which must be activated after its execution. If this set is not always activated, the unit will have a low probability of success (\( p(c) \)), even if another goal or subgoal is recognized at each failure.

This problem is easy to solve with a slight modification of step 4:

\[
\text{if } \exists c \in U \text{ so that } gact(c) = 1, u(c,t) = 1 \text{ and dist}(c) \leq \text{dist}(c_{\text{best}})
\]

then \( \text{nsuc}(c_{\text{best}}) = 1 + \text{nsuc}(c_{\text{best}}) \)

else \( \text{nsuc}(c_{\text{best}}) = gact(c_{\text{best}},t) + \text{nsuc}(c_{\text{best}}) \)

The drawback of this solution, is that once this rule is applied, the unit \( c_{\text{best}} \) does not represent a rule of the environment anymore. But in all the experiments we have done, this problem occurs only for a minority of units.

5 The experiments

5.1 The problem

This system has been applied to several different problems (see [4] for their complete description). The experiment
described here is a simulation of a cube manipulation task. It is similar to the experiment described by Whitehead & Ballard [13]. The system must pick up a green cube among four cubes of different colors. The cubes are randomly stacked on a table so that the green cube can be under several other cubes. In such a case, the system must unstack several cubes before accessing the green one. There is a hand which can be moved laterally on the left or on the right. It can also be moved vertically to take or put one cube on the table or on a stack.

There is an active visual device which can be used to focus on a particular cube. Most of the perceptions are relative to the focus: the position of the hand, the number of cubes above the focus, and the position of other cubes.

In order to find a cube with a particular color, there is an action which selects the desired color. When a color is selected, the position detectors react to this color.

Finally, the problem is to learn to select the green color, to move the focus on a green cube, to unstack the cubes above the green one, and to take the green cube (figure 2).

For this task, the system has forty-three feature detectors, including eleven action-detectors:

- $P_0, P_1, \ldots, P_{10}$ detect the position of the hand relatively to the focus ($P_5$ is active when the focus and the hand are vertically aligned)
- $N_0, N_1, N_2, N_3$ detect the number of cubes above the focus.
- $D_0, D_1, \ldots, D_8$ detect the position of the selected color from the focus point. $act(D_0, t) = 1$ when the color is in the focus. The others detect the color in each of the height surrounding directions.
- $SR, SG, SB$ detect the selected color (red, green or blue).
- $HR, HG, HB$ detect the color of the cube in the hand.
- $HC, HN$ detect the state of the hand: $HC$ when it contains a cube, $HN$ otherwise.
- $MR$ and $ML$ detects the execution of the right and left move.
- $TAKE$ and $PUT$ are the action-detectors of the elementary actions take and put.
- $MFL, MFR, MFA, MFB$ are the action-detectors of the focus movements: left, right, above, or below.
- $ASR, ASG, ASB$ are the action-detectors of the color selections.

Two goals are given to the system:

a) the focus-goal: to focus on the green cube,
b) the take-goal: to take this cube.

They are represented by two goal-units. The focus-goal unit is associated to $D_0$ and $SG$. The take-goal unit is associated to $HG$. We can consider these two goals as two macro-actions: focus-on-green-block and take-green-block. The condition for the second one is the detection of the first one.

5.2 The results

The learning experiment consists in presenting a pile of blocks and waiting until the system takes the green cube, before presenting a new pile. The performance of the system is the number of actions used to get the green cube. As we can see in figure 3, for the first piles, the system may use several hundred of random actions (sometimes more than one thousand) before taking the desired cube. After the 359th pile, this number is definitely below 30. This is the last time a new sequence is created. As a matter of fact, our system learns a new method only when it is not already in its memory. Since the environment is quite simple, and because the system has some generalization capabilities, the set of necessary methods is rather small.

The system learns a complete representation of the problem using a network of 148 cells and 904 links, with no redundancy nor useless methods (other methods are

---

![Figure 2: The system focuses on the green cube (step 2, actions $MFB$ and $MFR$), then it uses its hand to unstack it (step 3 to 8, actions $ML$, $TAKE$, $ML$, $PUT$, $MR$, $TAKE$).](image-url)
destroyed). The 359 first problems are solved with a total of 18742 actions. As we can see on the curve, this last problem corresponds to an isolated peak. There are isolated peaks because the system cannot learn to solve a situation before it faces it for the first time. Since some situations are less frequent than others, the system may a great number of problems before facing a new one which does not match its knowledge.

As it was predictable, the system almost learns the whole set of methods which are necessary solve the **focus-goal**, before starting to learn methods for the **take-goal**. Once it focuses on the green cube, it starts using random actions to unstack it. Then, it often occurs that one of these random actions removes the focus from the green cube. Therefore, the system must solve the **focus-goal** again. When the **focus-goal** has enough methods, the bad effect of a random action is immediately canceled by one of them. Therefore, the only actions which have a real effect are the ones concerned with the **take-goal**. The learning rate is increased by the cooperation of the goals.

An interesting result is that the system learns long methods (with three units) to «refocus» on the green cube. As a matter of fact, when the system focuses on the green cube (which means that the selected color is green and that the position information is $D_0$), and randomly chose the action which selects the blue color, it looses the information about the position of the green cube (while it get information about the position of the blue ones). But with a long method, it is possible to «remember» that before selecting the blue color the green cube was in focus. In this case a simple selection of the green color is sufficient to focus on the green cube again. Here is one of the methods which has been learned:

$$(D_0, SG), [ASB, SB], [ASG, SG, D_0]$$

$D_0$ and $SG$ represent the **focus-goal**. The whole sequence means that if the selection of the blue color occurs while the focus is on the green cube, then the execution of the green color selection, $ASG$, will be followed by the **focus-goal**.

This method which is only used to «refocus» on the green cube, is no more necessary once the system has learned every unstacking methods. It is thus forgotten several hundred time steps after the last time the system has learned a sequence.

It is difficult to compare our system to the one used by Whitehead and Ballard: the perceptual and action devices are very different. They use a rule based system which maps perceptual states to actions. It does not create general rules. But its active perceptual device provides implicit generalization of external states. Their system learns to solve 95% (with less than 30 actions) of the presented pile of blocks after about 400 examples.

A second experiment has been carried out, with a slight change in the environment, so that longer methods are necessary to solve the **take-goal**. The problem is the same except that the table is smaller than in the previous experiment. For example, the green cube can be near the right border of the table, so that it is impossible to put a cube on its right side. In such a case, the action which moves the hand on the right is inhibited. Since the same problem occurs on the other side of the table, the system cannot learn an efficient method to put the block on the right side of the green cube. Moreover, there is no detector in the perceptual device to indicate whether there is room or not, on the right or on the left side of the focus. Then, the only solution is to learn a long method. The method learned by the system consists in trying to move the hand on one side, and if the action had no effect, to move the hand on the other side. Here is the method:

$$(HC, P5), [HC, P5, MR], [HC, P4, ML]$$

$HC$ indicates that the hand holds a cube, $P5$ that the hand is...
vertically aligned with the focus point. MR and ML are the right and left move of the hand. P4 indicates that the hand is on left side of the focus. The second perception, \{HC, P5, MR\}, indicates that the hand is at the same position after using MR. The third perception indicates that the hand is on the left side of the focus after using ML. To summary, this sequence means that if the left move has no effect then the right move will have. The first two units are necessary to detect the fact that the left move has no effect.

The results of this experiment are shown in figure 4. The system stops learning after 731 problems. They are solved with a total of 103844 actions. The number of units remaining in the memory is 218. There is 856 links. The number of actions decreases more slowly than in the previous experiment, but the convergence time is not prohibitive. It seems that learning methods with three units is not a so hard problem in this environment. But this is not true for methods with large number of units. As a matter of fact, a method can be learned only if the system can do it a first time. The probability of generating randomly a particular sequence of 3 or more actions in a particular context, may be very low if the number of actions is high. Then we cannot expect such a system to learn very long methods in a reasonable time. Despite this limitation, the experiments have shown that the capability of learning sequences with more than two units may be useful. The algorithm we propose has no problem to discover them when they are necessary.

6 Conclusions

We have described a reactive system which can learn a representation of the deterministic environment where it evolves. The behavior is represented as a succession of goals and subgoals related by sequences of perceptions representing the rules which must be used in order to get from one subgoal to another. The sequences represent the effect of their actions in the environment, and they concerns classes of sequences instead of single states. The system is also capable of learning rules which take into account successions of states. This seems to be more difficult in classifier systems [10].

The limits of the learning capabilities of our system can be characterized. Let us remember that it can learn to solve a subgoal only if it can be reached by few actions from a previous subgoal. Since a subgoal is represented by a set of feature detectors, the system can learn a given task if every property of each subgoal is detected by one of the given detectors. This capability is not sufficient for learning complex tasks such as those addressed by symbolic systems. But many improvement can be done from this first version of the architecture. One idea consists in using the sequences for planning tasks. We could also try to adapt the system to real world data, using perceptron learning rule instead of classical induction.

An important observation is that it is necessary to give a priori goals to the system when it has to learn a non-trivial task. It is not realistic to expect such a system to learn only one complex final goal, since the system begins to learn only when this goal is observed. Thus, it seems that an animat should have many a priori goals such as focusing on objects, touching objects, grasping objects, moving objects... The robot would have to learn to solve these goals before trying to use them as macro-actions to solve a more complex problem.

References