Pakistan Journal of Applied Sciences 2(3): 408-415, 2002

Automatic Tuning of Q-learning Algorithms' Parameters

M. R. Meybodi and S. Hodjat
Soft Computing Laboratory, Computer Engineering Department
Amir Kabir University of Technology, Tehran, Iran

Abstract: This paper describes a general approach for automatically tuning of reinforcement learning algorithms' parameters. In this approach a reinforcement learning agent's parameters are tuned by other more simple algorithms of reinforcement learning. We will explain this approach by tuning one of the parameters of a Q-learning and statistical clustering algorithm. The results of tuning this parameter will be described by some simple examples. Comparing the result of an algorithm using automatically tuned parameters and the algorithms with fixed parameters will show that the former is generally more flexible and capable of performing better in most cases.

Keywords: Reinforcement Learning, Hierarchical learning, Parameter Tuning, Q-Learning, Learning Automata, Statistical Clustering

Introduction

Reinforcement learning (Kaelbling *et al.*, 1996) studies how an agent can optimally choose an action by trial and error methods such that it maximizes over time a reward function measuring it's performance. Figure 1 shows the feedback connection of a reinforcement learning agent and it's environment. An interaction of the agent and it's environment goes as follows:
Step 1: the agent will choose an action a, in state S_1, and it applies the action.
Step 2: The agent will receive an immediate reward, r, from the environment.
Step 3: The agent will visit the next state S_2.
The above steps make one iteration of the agent's learning algorithm and takes place in one time step. An interaction of the agent and it's environment could be summarized with the tuple $<S_1,a,r,S_2>$. This tuple defines an experience of the agent.

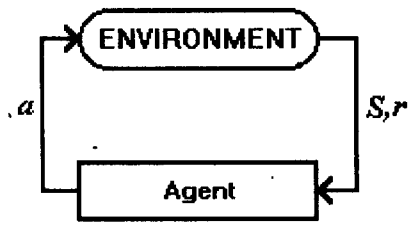


Fig. 1: The Feedback Connection of Agent and Environment

There are a variety of different reinforcement learning algorithms (Kaelbling *et al.*, 1996) (Harmon and S. Harmon, 1996). Almost each one of these algorithms have some sort of parameters, and the performance of the algorithms in any application highly depends on the right selection of these parameters. In the traditional approach the parameter values are chosen by the designer of the algorithm. The designer may choose conservative values which may not be the best but will work fairly well on most environments; or he may use some trial and error sessions and choose the parameters by comparing the performance of the algorithm using different values for it's parameters. The chosen parameters are then used as constants in the course of learning. It is obvious that choosing the parameters in this manner is inaccurate, time consuming and not flexible (specially in more dynamic environments). There has been some approaches where the parameters are tuned during the course of learning. For instance some of the undirected exploration techniques used in the reinforcement learning algorithms have a temperature parameter (Kaelbling *et al.*, 1996). This parameter controls the exploration vs. exploitation factor. In some approaches this parameter is initially set to a high value and it is decreased over time to decrease exploration. The idea is that as time goes by the agent will need less exploration. But there are some problems with this approach; an appropriate rate of changes in exploration could not always be determined prehand, and in dynamic environments we usually need a more controlled way of changing the exploration rate (for example, the agents may need to decrease and increase their exploration based on the familiarity of the current state of the environment). Another obvious approach is to manually tune the algorithm parameters. But tuning the parameters is not a simple task even for a human observer. Having a human supervisor for tuning the parameters is also in contradiction with one of the fundamental properties of reinforcement learning algorithms which is an unsupervised learning.

The approach taken here is to use some hierarchical learning method where an agent parameters are tuned by some sub-agents (Hodjat and Meybodi, 1996) (Hodjat, 1997). The agent acts in the environment and the sub-agents act over the agent's parameters and are reinforced by the agent's performance and status in the environment (thus the agent will be the environment for the sub-agents). Our hierarchical approach could be compared with the gated behavior method (Maes and Brooks, 1990) (Mahadevan and Connel, 1992). In the gated behaviors approach there is a collection of behaviors that map environment states into low-level actions and a gating function that decides what behavior should be executed. An instance of the gated behavior approach called the Feudal Q-learning (Dayan and Hinton, 1993) (Watkins, 1989) is the most similar to our approach. Feudal Q-learning, in the simplest case, has a high-level master and a low-level slave. The master receives reinforcement from the external environment. It's actions consist of commands that it can give to the low-level learner. When the master generates a particular command to the slave, it must reward the slave for taking actions that satisfy the command, even if they do not result in

external reinforcement (Kaelbling *et al.*, 1996). The difference between our hierarchical approach and the gated behavior approaches is depicted in Fig. 2. In the gated approaches the high-level learner maps environment states to commands and the low-level learners map commands to actions. But in our approach the high-level learner is responsible for mapping the environment states to actions and low-level learners are responsible for the changes in the internal state of the high-level learner (so they are placed inside the high-level learner and they only interact with this learner).

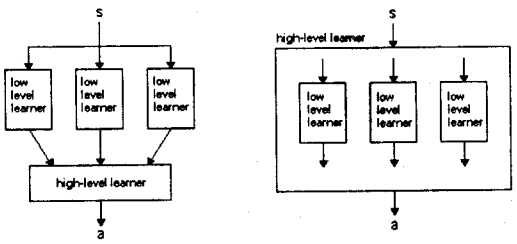


Fig. 2: Comparing Our Hierarchical Approach with Gated Behavior Approach

In this paper we will examine the behavior of an agent using the Q-learning and statistical clustering algorithms. These algorithms are borrowed from the (Mahadevan and Connel, 1992) article. Section 2 gives a brief description of Q-learning and statistical clustering. Section 3 gives an introduction to learning automata (Narendra and Thathachar, 1989) and describes some different learning automatons that we have used for tuning our agent's parameters. In section 4 the proposed model for tuning the parameters of a reinforcement learning algorithm called Q-Learning is presented. Section 5 contains the description and the results of some simple experiments used for evaluating the proposed tuning method. Finally, the last section concludes and suggests directions for further research.

Learning and Statistical Clustering

Q-learning: The most important problem of a learning agent in reinforcement learning is to determine the consequence of applying an action in the environment. This is called the temporal credit assignment problem. Q-learning is a well known learning algorithm for temporal credit assignment. The Q-learning algorithm is based on a data structure named *Q*. This data structure evaluates the utility of applying an action *a* in a state *S* and it is represented by *Q(S,a)*. *Q(S,a)* is initialized to zero for every action *a* and every state *S*. The *Q(S,a)* is then updated after each new experience of the agent. If the agent makes the $<S_1,a,r,S_2>$ experience, the $Q(S_1,a)$ structure is updated using the following formula:

$$Q(S_1,a) \leftarrow Q(S_1,a) + \lambda(r + \gamma \underset{x=a_1..a_m}{Max}\, Q(S_2,x) - Q(S_1,a)) \quad (1)$$

Max $Q(S_2,x)$ represents the utility of state S_2. The λ parameter ($0<\lambda<1$) controls the error correction rate. Setting λ to 1 results in one-shot learning and setting it to zero results in a no learning situation. The γ ($0<\gamma<1$) parameter controls the rate which the utility of the next state is discounted. When there are no delayed rewards this parameter should be set to zero.

Statistical Clustering: Another problem that an agent has to face in reinforcement learning is the task of determining the similar states of the state space. This problem is specially important when the agent deals with large state spaces. Statistical clustering (Schalkoff, 1991) tackles this problem. This is a technique for creating and updating clusters that represent similar experiences. In this technique every new experience is compared with existing clusters. If there are matching clusters the new experience would be merged with them, otherwise a new cluster will be generated based on the newly encountered experience. Suppose that an agent's state *S* is represented by the following n bits: $S=<s_1,s_2,...,s_n>$, then the tuple $C=<(o_1,z_1),(o_2,z_2),...(o_n,z_n),Q_c,m_c>$ represents a cluster. Here O_i (or Z_i) is the decayed count of zeros (or ones) in the *i*th bit of a state S matching cluster *C*, Q_c is the *Q* value of the cluster; and m_c is the number of experiences matched with this cluster. Suppose that we could estimate the probability of a state matching a cluster: $P(S \in C)$ (an estimate of this could be found in [8]), then we could assume a state S_1 in the experience $<S_1,a,r,S_2>$ matches a cluster c if:

$$\begin{cases} \text{Cluster} C \text{ is under action} a \\ P(S_1 \in C) > \varepsilon \\ |Q_c - Q_s| < \delta \end{cases} \quad (2)$$

Where ε and δ are usually a fixed parameter input to the clustering algorithm. The value of these parameters directly influences the type and number of clusters formed by the algorithm (high values for ε and low values for δ result in more and similar clusters and low values of ε and high values for δ result in fewer and dissimilar clusters).

If a state *S* matches a cluster *C*, then *C* should be updated. The following computations are used for updating the *C* elements:

$Z_i \leftarrow \mu Z_i$, $O_i \leftarrow \mu O_i + 1$,if $S_i=1$ (3)

$Z_i \leftarrow \mu Z_i + 1$, $O_i \leftarrow \mu O_i$,if $S_i=0$ (4)

$$Q_c \leftarrow Q_c\left(\frac{m_c}{m_c+1}\right) + Q(S,a)\left(\frac{1}{m_c+1}\right) \quad (5)$$

$m_c \leftarrow m_c + 1$ (6)

Here μ is a real number between 0 and 1 that controls the importance of new experiences (if $\mu=1$ then the new experiences have no significance over the older experiences). μ is usually estimated with the formula $\mu=(2K-1)/2K$, where K is an integer (the K parameter will be also used in creating new clusters).

If a state does not match any clusters then a new cluster should be created from that state. This is done by first creating an empty cluster. The elements of an empty cluster are initialized as follows:

$Z_i = O_i = K$ (7)

$Q_c = 0$ (8)

$m_c = 0$ (9)

Then the state *S* should be merged with the empty cluster using the equations 3 to 6.

If two clusters are 'close' enough they should be merged. Two clusters C_1 and C_2 are considered close if:

$$\begin{cases} C_1 \text{ and } C_2 \text{ are both under action } a \\ \text{distance}(C_1, C_2) < \rho \\ |Q_c - Q_s| < \delta \end{cases} \quad (10)$$

distance(C_1,C_2) is used to measure how close two clusters are (Mahadevan and Connel, 1992) shows a way to estimate this function), and ρ is another parameter usually fixed to the clustering algorithm (ρ and δ are also important in the type and number of clusters generated by the algorithm). If the above equations hold the clusters are merged into a new cluster, C. The elements of this cluster are formed as follows:

$$Z_i \leftarrow Z_{C_1,i}\left(\frac{m_{C_1}}{m_{C_1}+m_{C_2}}\right)+Z_{C_2,i}\left(\frac{m_{C_2}}{m_{C_1}+m_{C_2}}\right)$$

$$Q_i \leftarrow Q_{C_1,i}\left(\frac{m_{C_1}}{m_{C_1}+m_{C_2}}\right)+Q_{C_2,i}\left(\frac{m_{C_2}}{m_{C_1}+m_{C_2}}\right)$$

$$Q_c \leftarrow Q_{C_1}\left(\frac{m_{C_1}}{m_{C_1}+m_{C_2}}\right)+Q_{C_2}\left(\frac{m_{C_2}}{m_{C_1}+m_{C_2}}\right)$$

$$m_c \leftarrow m_{C_1}+m_{C_2}$$

The Q-learning and statistical clustering algorithm, chooses an action a in each state S such that $Q(S,x)$ would be maximized over all actions x=$a_1,a_2,\ldots,a_m$. The following formula is used to compute $Q(S,x)$ over all clusters:

$$Q(S,x) = \frac{\sum_{C \in C_x} (Q_c \times P(S \in C))}{\sum_{C \in C_x} P(S \in C)}$$

In the above equation C_x is the set of clusters under action x.

But choosing actions in the approach described above would not lead to a policy that could explore new actions. Using random walks is one way for ensuring exploration in the algorithm. For this purpose a new parameter, θ, should be used. This parameter controls the percentage of random action selection in the algorithm. Increasing the value of this parameter will increase the exploration rate of the agent (and vice-versa).

Summarizing the Q-learning and Statistical Clustering

- Assign appropriate values for the Q-learning and statistical clustering parameters ($\theta,\gamma,\lambda,\varepsilon,\delta,K,\rho$)
- Do forever:

A: In θ percent of time select an action randomly. In other times select an action, a, that would maximize $Q(S,x)$ for 1 all actions x.

B: Apply action a in the environment. Sense the resulting state S_2 and receive the immediate reward r.

C: Update $Q(S_1,a)$ using equation 1.

D: Merge S_1 in all matching clusters; if there isn't any matching clusters then create a new cluster from state S_1.

E: Merge every two clusters C_1 and C_2 that satisfy equation 10.

In this paper we would refer to the Q learning and statistical clustering algorithm as the Q algorithm.

Learning automata: In Learning automata the agent (automaton) does not receive any other input other than the immediate reward (Fig. 3). Here the automaton must adapt an action selection strategy based on only the reward it receives from the environment. A Learning automata could be represented by five elements [Narendra89]: The reward, r, the action, a, the agent's internal state, φ, and two functions $f: r \rightarrow \varphi$, $g: \varphi \rightarrow a$. When the f and g mappings are constant the Learning automata is said to be fixed structured; otherwise the Learning automata is variable structured.

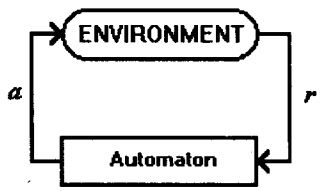


Fig. 3: The Feedback Connection of Learning Automaton and Environment

Fixed structured automatons

FA Automaton: A simple action selection strategy for an agent is to keep selecting an action when it is successful and to select another action when it is not. Fig. 4 depicts this strategy (m is the number of actions). To implement this strategy we need a way to determine favorable and unfavorable responses to actions. Suppose that the automatons immediate reward is a real number between 0 and 1; then a simple approach for determining favorable and unfavorable responses could be:

IF $r(t) > 0.5$ *THEN* response = favorable *ELSE* response = unfavorable

In this paper we would represent this algorithm with the FA symbol.

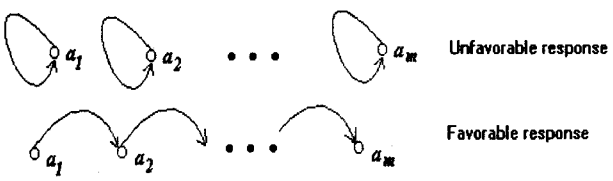


Fig. 4: The State Transition Graph for FA Automaton

Tseltsin's (1962) First Automaton (L): Tseltsin's first automaton keeps an account of the number of successes and failures received for each action. It is only when the number of failures exceeds the number of successes (or some maximum value N) by one that the automaton switches from one action to another (Narendra and Thathachar, 1989). N is the depth of the automaton and it could be seen as the memory of

the agent. φ is the agents internal state, the φ_{iN+1}, φ_{iN+2}, ..., φ_{2iN} states correspond to the action a_i (where i could be any integer between 0 and m-1). This automaton will be referred to as the L automaton.

Tseltsin's Second Automaton (G): The L automaton switches from state φ_{iN} to state $\varphi_{(i+1)N}$ when it encounters an unfavorable response from the environment. Since $\varphi_{(i+1)N}$ corresponds to action a_{i+1}, this action is now performed. If this again results in an unfavorable response, the automaton immediately switches to another action. Unlike the L automaton, Tseltsin's second automaton (or the G automaton) performs any action a_i at least N times (resulting in N consecutive penalties) before choosing another action (Narendra and Thathachar, 1989).

The Krinsky (1964) Automaton (K_1): This automaton behaves exactly like the L automaton when the response of the environment is unfavorable. However, for a favorable response, any state φ_{iN+j} ($0 \le i \le m-1$ and $1 \le j \le N$) passes to state φ_{2iN}. This in turn implies that a string of N consecutive unfavorable responses are needed, in general, to change from one action to another.

The Krylov (1964) Automaton (K_2): This automaton has state transitions that are identical to the L automaton when the output of the environment is favorable. However, when the response of the environment is unfavorable, a state φ_{iN+j} ($0 \le i \le m-1$ and $1 < j < N$) passes to a state φ_{iN+j+1} with probability 1/2 and to a state φ_{iN+j-1} with the same probability. When $j=1$, φ_{iN+1} stays in the same state with the probability 1/2 and moves to φ_{iN+2} with the same probability. When $j=N$, $\varphi_{(i+1)N}$ stays in the same state with the probability 1/2 and moves to $\varphi_{(i+2)N}$ (or $\varphi_{(i-2)N}$ if $i=m$) with the same probability.

Variable structure automatons: In variable structure automatons a probability P_a is assigned to each action a. When the automaton selects the action, a, in time step t, and receives a favorable response, then P_a will increase and the probability of selecting other actions (P_i for all actions $i \neq a$) will decrease (The opposite occurs if it receives an unfavorable response). The following equations show how the action probabilities are updated when an action a is executed in the environment and the reward r is received:

$$p(i) \leftarrow p(i) + \frac{(1-r)\times\beta\times p(a)}{m-1} - \frac{r\times a\times(1-p(a))}{m-1} \quad \text{, for all } i \neq a$$

$$p(a) \leftarrow p(a) - (1-r)\times\beta\times p(a) + r\times a\times(1-p(a))$$

Where m is the number of actions, and α and β are reward and punishment parameters ($0 \le \alpha \le 1$ and $0 \le \beta \le 1$). The reward parameter controls the rate which probability of an action is increased when: 1)The action is selected and the agent has received a favorable response; or 2) When the action is not selected and the agent has received an unfavorable response (similarly the punishment parameter controls the rate which the probability of an action is decreased when the opposite occurs). In the above equations the immediate reward, r, should be a real number between 0 and 1.

Variable structure learning automatons are classified into 3 different schemes based on the value of α and β parameters: If $\alpha \neq 0$ and $\beta = 0$, then the automaton is called linear reward inaction (LRI). If $\alpha=\beta\neq0$ the automaton is called linear reward-penalty (LRP); and if $\alpha\neq0$ and β is sufficiently small the automaton is called linear reward-ε-penalty (LRεP). Suppose that the probability of selecting actions $a_1, a_2, \ldots, a_m$ in time t is represented with the vector $P(t)=\{P_{a_1}, P_{a_2}, \ldots, P_{a_m}\}$. $P(n)$ is called an absorbing state if for each $k \ge n$ we would have $P(k)=P(n)$. An automaton with an absorbing state is said to be an absorbing automaton. It could be proved that in a stochastic environment the LRI automaton is always absorbing (Meybodi and Lakshmivarahan, 1982) (Narendra and Thathachar, 1989).

4. Fine Tuning Q Algorithm Parameters Using Learning Automata: The general schema that we would use for tuning the Q algorithm parameters is summarized below (Fig. 5):

Do forever:

- The automatons choose and apply actions that would select a value for the Q algorithm parameters they tune.
- The Q algorithm uses the selected value of the parameters for a fixed *period* of time.
- After the period is over the efficiency of each parameter is reinforced to the automatons responsible for tuning them. The efficiency is evaluated from the performance of the Q algorithm in the environment.
- The automatons update their action selection strategy based on the feedback.

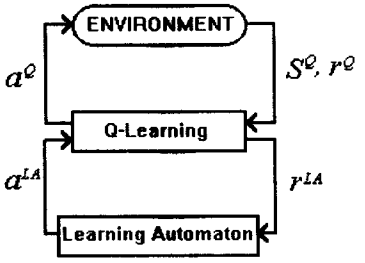


Fig. 5: The Hierarchical Feedback Connection

As the above schema indicates, the automatons should be equipped with a set of actions for changing the values of the Q algorithm's parameters. We have assigned a set of values to each parameter for this purpose. We will refer to this set with ξ. Each automaton is equipped with actions for selecting an element of the ξ. All the values that might be appropriate for the parameter in an environment or different situations in an environment should be included in ξ. But a large ξ would make a large state space and will slow down the automatons. So we have partitioned the intervals of appropriate values for each parameter into some sub-ranges, and only one candidate for each sub-range is included in ξ (most experiments show that the Q algorithm performs nearly the same when it's parameters change in a relatively large range). We have followed some very simple rules to select the elements of ξ for each parameter: First, each element should represent a unique interval of the accepted values. Second the intervals should cover the whole domain of the values that are logically acceptable in an unknown environment, and finally the number of the elements in the set should not be too much.

Tuning The θ Parameter Using Learning automatons: As we have already mentioned, the θ parameter is the percent of time that the Q-algorithm selects it's actions randomly. This parameter could accept any value between 0 and 1. We have partitioned the [0,1] interval to 10 sub-intervals with length 0.1, and we have selected the smallest number in each sub-interval as an element of the $\xi(\theta)$, that is: $\xi(\theta)$ = {0,0.1,0.2,...0.9}. The automaton that tunes θ has to select an element of this set for the Q-algorithm's θ parameter. This means that the automaton should have 10 actions for selecting each element. The automaton also needs to be reinforced by an evaluation function. We have defined the θ parameter's evaluating function with the following rules: (These rules generate the automaton's feedback in time t, it is also assumed that the automaton has had received the most rewards in the interval [*T-period,T*])

$$\text{IF} \sum_{i=t-period}^{t} r_i^Q \geq \sum_{i=T-period}^{T} r_i^Q \text{ THEN } r^{LA} = 1$$

$$\text{ELSE IF} \sum_{i=t-period}^{t} \frac{r_i^Q}{p} \leq \sum_{i=1}^{t} \frac{r_i^Q}{t} \text{ THEN } r^{LA} = 0$$

$$\text{ELSE } r^{LA} = \frac{\sum_{i=t-period}^{t} \frac{r_i^Q}{p} - \sum_{i=1}^{t} \frac{r_i^Q}{t}}{\sum_{i=T-period}^{T} \frac{r_i^Q}{p} - \sum_{i=1}^{t} \frac{r_i^Q}{t}}$$

The first rule will be true when the Q-algorithm has performed better or equal to it's best period in the environment. In this case the parameter is considered to be working good enough and the automaton will receive the maximum reward. The second rule will be true when the Q-algorithm has performed worse than it's average performance in the environment, in this case the parameter is considered to be inappropriate and the automaton will receive the minimum reinforcement. Finally, if the Q-algorithm performs better than it's average but worse than it's best then the third rule will generate a reinforcement for the automaton.

Tuning The Statistical Clustering Parameters Using Learning automatons: We did also tune some of the statistical clustering parameters in our experiments: namely the ε, σ and ρ parameters. We used the following sets to select the values for these parameters: $\xi(\varepsilon)$ = {0, 0.0000001,0.000001, 0.00001, 0.001, 0.2, 0.4, 0.6, 0.8}, $\xi(\sigma)$ = {0, 1, 2, 3, 4} and $\xi(\rho)$ = {1, 5, 10, 15, ..., 45}. The statistical clustering parameters influence the count and type of the clusters of the Q algorithm, This in turn will influence the performance of the Q-algorithm. In general we would like our agent to perform the best with minimum number of clusters. The evaluating function we used for these parameters are the same as θs evaluating function when the number of clusters are not changed in a period. But if the number of clusters increase and the average rewards received by the Q-algorithm does not improve in that period, then the automaton would receive r=0. If the number of clusters decrease in a period and the Q-algorithm's average reward received in that period doesn't decrease then the automaton would receive r=1.

Experiments and Results: We have used the LRP, LRεP, LRI, FA, L, G, K1 and K2 automatons for tuning the Q-algorithm parameter θ and we have named the resulting agents Q_LRP, Q_LRεP, Q_LRI, Q_FA, Q_L, Q_G, Q_K1 and Q_K2, respectively. In this section some experiments in tuning the θ are presented. We would refer to the agent using the Q-algorithm with fixed parameters (without tuning automatons) simply as the Q agent. For a discussion about tuning other parameters of Q-Learning refer to(Hodjat, 1997).

The environments in our experiments are defined by a set of rules governing the interactions of objects in an infinite grid. We have defined two different environments for our experiments. The first environment is governed with very simple rules, and we will call it the *simple* environment. We will use different configurations of this environment to analyze an agent's performance when the agent's θ parameter is tuned by different automata.

The agents in our simple environment have 4 actions and 4 sensors. The actions could move an agent to the four neighboring cells (the up, right, down and left cells), and the sensors point to the color of object's in these neighboring cells (the color of object's are represented with three bits and are unique for each of the object types in this environment). There are six type of objects in this environment (as indicated in Fig. 6). The environment rules are such that an agent could move to a

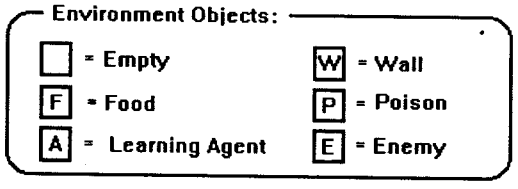


Fig. 6: The Object Types

cell only if that cell contains an empty object, in this case the agent will receive a zero reward (r=0). If an agent tries to move to a cell that contains one of the wall, food, poison or enemy objects it would receive the rewards -40, +100. -100 and -100 respectively. The wall, food and poison objects could not move in the environment but the enemy is a random-walker that would randomly choose an action to move to one of its four neighboring cells in every time step.

Performance Criteria: Performance of an agent could be defined in many ways. Here, we will define an agent's performance at time t as the average reward received by the agent from time 0 to time t. We have used the configuration depicted in Fig. 7 to examine the performance of our agents in the simple environment. In this configuration an agent (A) and an enemy (E) are placed within eight wall objects. So the movement of the agent is very limited. It is obvious that the agent should avoid hitting the walls and the enemy to receive the maximum rewards. The value of the parameters for an agent with fixed parameters (the Q agent) are set to: θ=0.1, k=5, δ=10, ε=0.000001, ρ=2, γ=0.9. These are the values that are usually recommended for these parameters (Mahadevan and connel, 1992). The agents with the tuned θ parameter also use the same values for their parameters except that their θ parameter is tuned as explained in section 4.1. Here we have set the period

of the automatons update intervals to 1. In this section we have carried 3 runs for each of our experiments and have used the result of the run with the average result in the text.

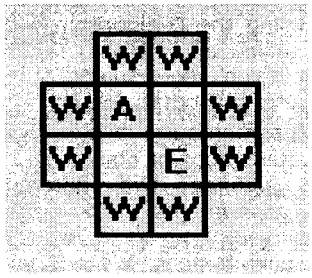


Fig 7: The Simple Environment Configuration Used In Performance Test

Fine Tuning With Variable Structure Automatons: In this experiment we have used the Q, Q_LRI(T), Q_LRP(T) and Q_LRεP(T) agents in the environment described in section 5.1.1.1. Here, we have set the α and β parameters of the LRI, LRP and LRεP automatons as follows:

Table 1: The Values Used for α and β

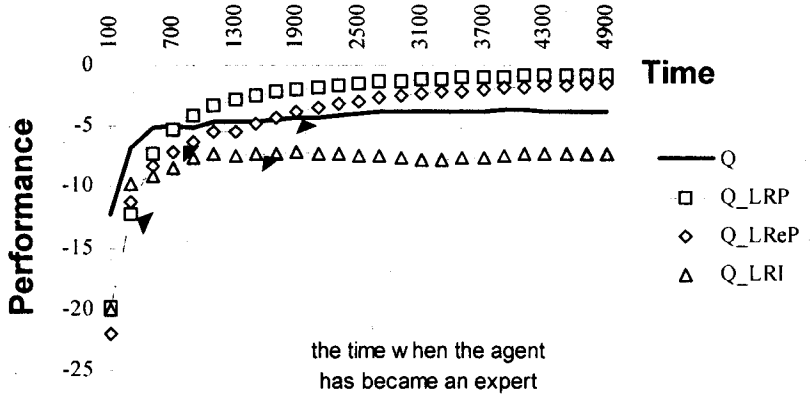
	α	β
LRI	0.1	0
LRP	0.1	0.1
LRεP	0.1	0.01

The values in Table 1 are the conventional values usually set for these automatons.

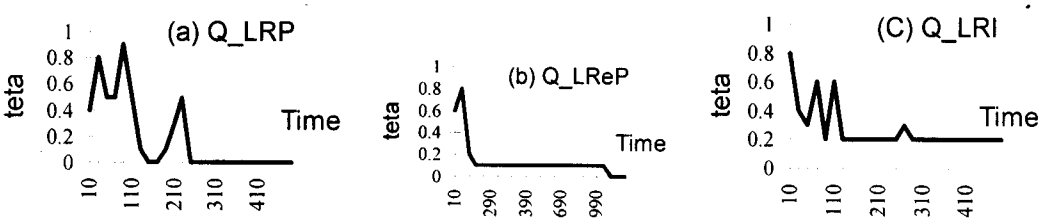
Graph 1 compares the performance of agents over 5000 time units. The arrows in this graph point to the time needed for each agent to become an expert in the environment. Here, we assume that an agent becomes an expert in the environment when it would find all the clusters needed to select it's actions optimally. We have used the time when an agent stops adding new clusters to estimate this. The graph shows that the Q, Q_LRI(T), Q_LRP(T) and Q_LRεP(T) agents become an expert after 1900, 300, 800 and 1600 time units respectively.

The Q_LRP(T) and Q_LRεP(T) agents show the best performance in the environment. The reason is: First, these agents use more random action selections at the beginning of the run, because their automaton sets the θ parameter to high values, this will result in more exploration and so the agents will become experts sooner than the agents with less random selections. Second, these agents do not use random selection after they become experts in the environment, because their automaton sets the θ parameter to 0 after they expertise. The graphs 2-a, 2-b and 2-c show how the Q_LRP(T), Q_LRεP(T) and Q_LRI(T) agents' θ parameter changes over time. As the 2-c graph shows the Q_LRI(T) agent's θ parameter has converged to the value 0.2 very soon, so it's performance is very similar to a Q agent with θ=0.2. The Q and Q_LRI(T) agents use 0.1 and 0.2 random selections respectively even after the become an expert, so they show a weak performance compared to the Q_LRP(T) and Q_LRεP(T) agents.

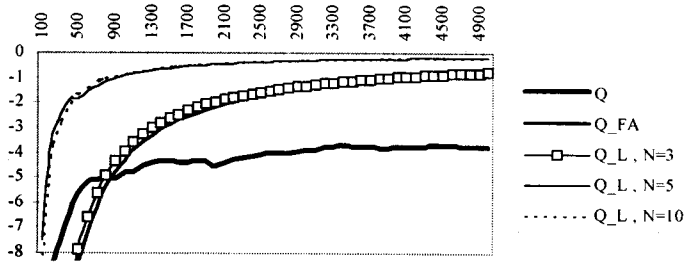
Fine Tuning With Fixed Structure Automatons: The Q_L(T), Q_G(T), Q_K1(T) and Q_K2(T) agents have been also tested on the environment described in section 5.1.1.1. Graphs 3 to 6 compare the performance of each agent using automatons with different memory depths with the Q agent. The graphs indicate that: First, tuning the parameters increase the agents performance (with the same reasons described in the previous section) and Second, the growth of memory depth also increases the performance of the agents to some extent. Graph 7 shows how the performance of the Q_L agent changes with different memory depths. As the graph indicates the memory depth between 5 to 10 has the best result in this environment. The results show that the performance of the Q_L(T), Q_G(T), Q_K1(T) and Q_K2(T) agents in this environment were relatively similar.



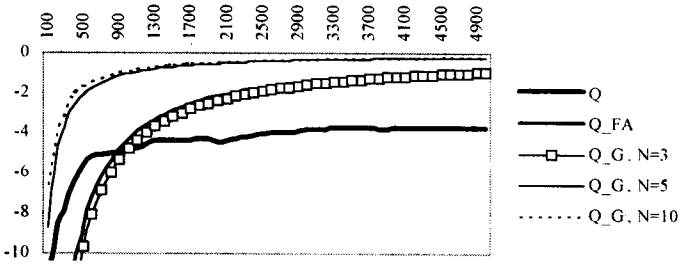
Graph 1



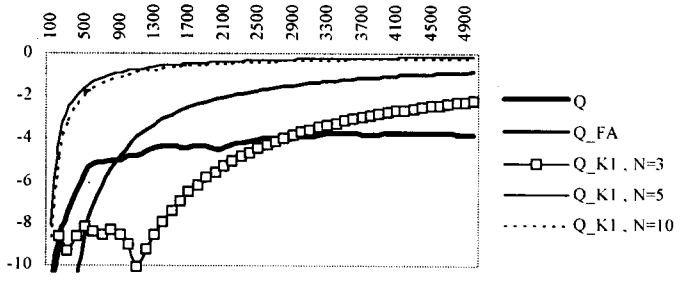
Graph 2



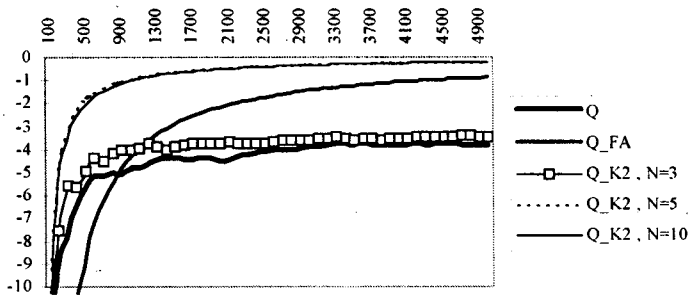
Graph 3



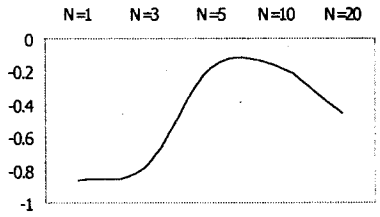
Graph 4



Graph 5



Graph 6



Graph 7

Conclusion

Automatically tuning the values of learning algorithms parameters removes the burden of finding the appropriate values for them. Most importantly it gives the algorithms more flexibility. Having fixed parameter values in the course of learning process is not appropriate in most cases. For instance a learning agent needs to explore more when it starts experimenting in a new environment, it needs less exploration after becoming expert in the environment, and it would again need more exploration whenever it encounters an unexpected situation. This article showed how tuning the parameters could increase the performance, flexibility and ability of learning sequences of actions in a learning agent. We also saw how tuning the clustering parameters could result in a better clustering of information. In our simple approach we used the same value parameter for all the clusters, a better approach would have been to assign different parameters for each cluster and tune each cluster's parameters independently.

In our proposed approach learning takes place in two different layers. The first layer deals with the environment and the second layer deals with parameters of the learning algorithms in the first layer. But the learning algorithms of the second layer may also have parameters of their own (the α and β parameters in the variable structure automatons and the memory depth parameter in the fixed structure automatons). The period of updating the second level algorithms should also be determined. But the experiments suggest that the impact of having imprecise values in the second level algorithm parameters are not as important as the first level parameters. In order to decrease the effect of imprecise parameter values of the second level algorithms new levels could be added to the learning hierarchy.

Our approach also requires us to define a set of acceptable values (ξ) for each tuned parameter. In general there should be infinite number of acceptable members for a typical set, but the tuning would work the best when the set would be limited to few members. These should be the members we expect to be optimal in large periods of time. But determining this set is also difficult or impossible in most cases. That is why we used a simple heuristic for this problem. The heuristic has been working well enough in most experiments but we expect that a better heuristic could be found if more work would be done in this area.

References

Dayan, P. and G. E. Hinton, 1993. Feudal reinforcement learning, In S.J. Hanson, J.D. Cowan and C. L. Giles (Eds.), Advances in neural information processing Systems 5, San Mateo, CA, Morgan Kaufmann.

Harmon, M. E. and S. S. Harmon, 1996. Reinforcement learning: tutorial, http://www-anw.cs.umass.edu./~mharmon/rltutorial.

Hodjat, S. and M.R. Meybodi, 1996. Fine tuning of Q-learning parameters using learning automata (in Farsi), In proceedings of the second annual conference of computer society of Iran, Tehran, Iran, 209-220

Hodjat, S., 1997. An artificial lab for creating and comparing learning algorithms (in Farsi), M.S. thesis, Computer Engineering Department, Amirkabir Uni., Tehran, Iran.

Kaelbling, L. P. , M. L. Littman and A.W. Moore, 1996. Reinforcement learning, Artificial Intelligence J. 4: 237-285

Krinsky, V. I., 1964. An Asymptotically optimal automaton with exponential convergence, Biofizica, 9: 99-105

Krylov, V. Yu., 1964, One stochastic automaton which is asymptotically optimal in random medium, Automata and Remote Control, 24: 1114-16

Maes, P. and R. A. Brooks, 1990. Learning to coordinate behaviors, In proceedings of eighth national conference on artificial intelligence, pp 796-802

Mahadevan, S. and J. Connel, 1991. Scaling reinforcement learning to robotics by exploiting the subsumption architecture, In proceedings of the eighth international workshop on machine learning, 328-332

Mahadevan, S. and J. Connel, 1992. Automatic programming of behavior-based robots using reinforcement learning, Artificial Intelligence J. 55: 311-365

McCallum, A. K., 1996. Efficient exploration in reinforcement learning with hidden state, Uni. of Rochester.

Meybodi, M. R. and S. Lakshmivarahan, 1982. ε-Optimality of a general class of absorbing barrier learning algorithms", Information Sciences, 28: 1-20

Narendra, K. S. and A. L. Thathachar, 1989. Learning automata, Prentic Hall, NJ, USA.

Schalkoff, R., 1991. Pattern recognition, Wiley International Editions. NY, USA.

Tseltsin, M. L. 1962. On the behavior of finite automata in random media, Automata and Remote Control, 22: 1345-54

Watkins, C., 1989. Learning from delayed rewards, PhD. Thesis, Kings College.