INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Learning and Adaptation: A Comparison of Methods in Case of Navigation in an Artificial Robot World

Yohannes Kassahun and Gerald Sommer

Bericht Nr. 0309
November 2003

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Learning and Adaptation: A Comparison of Methods in Case of Navigation in an Artificial Robot World

Yohannes Kassahun and Gerald Sommer

Bericht Nr. 0309
November 2003

e-mail: {yk,gs}@ks.informatik.uni-kiel.de

Dieser Bericht ist als persönliche Mitteilung aufzufassen

ABSTRACT

Neural networks, reinforcement learning systems and evolutionary algorithms are widely used to solve problems in real-world robotics. We investigate learning and adaptation capabilities of agents and show that the learning time required in continual learning is shorter than that of learning from scratch under various learning conditions. We argue that agents using appropriate hybridization of learning and evolutionary algorithms show better learning and adaptation capability as compared to agents using learning algorithms only. We support our argument with experiments, where agents learn optimal policies in an artificial robot world.

CONTENTS

1. INTRODUCTION

1.1 Background

When an infant learns how to go and how to stand, it has no explicit teacher, but it does have a direct sensori-motor connection to its environment. From this connection, the infant receives a wealth of information about cause and effect, about consequences of actions, and about what to do in order to achieve goals. This interaction is a major source of knowledge about our environment and ourselves. Learning from interaction is a fundamental idea underlying nearly all theories of learning and intelligence [14]. It is used by agents at the individual level. In this work, we investigate agents using learning from interaction. This type of learning is different from supervised learning, which is learning from examples provided by a knowledgeable external supervisor. Supervised learning is an important type of learning but alone it is not adequate for learning from interaction. Moreover, it is usually impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act and learn [10].

At the population level, it is clear that parents have inherited the infants the ability to learn and survive. This inherited ability is developed through evolution. A generation of an organism can only survive or continue to live if the population adapts itself to various situations in the environment. This shows that the learning and adaptation capabilities of agents is also affected by evolution.

An individual or a population of individuals learn and adapt to a situation in an environment either from scratch, that is without having any knowledge about the situation, or continually depending on the initial knowledge about the situation. At individual level, adaptation refers to the maximization of the satisfaction of the individual in its life time for different situations in the environment. At population level, adaptation refers to the survival of the individual. In other words, it is the ability of the individual to have offspring under new situation.

Natural evolution implies that organisms adapt to their environment. Evolution works over many generations, covering much longer periods than those of lifetime learning [8]. How could an individual learn to use its eyes if it hadn't been equipped with eyes through evolution? An organism without any organ of sight might not be able to react to visual stimuli, but it could be the ancestor of a species with eye-like organs. Therefore, evolution can be considered as a process of meta-learning on a generational level. Only evolutionary adaptations and innovations enable organisms to "optimally" react to environmental conditions. This involves an impressive potential for creativity and innovation.

Reinforcement learning [5] is one form of learning from interaction. It is learning what to do, how to map situations to actions so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover by itself which actions yield the most reward by trying them [14]. Like the infant, an agent using reinforcement learning learns and adapts itself through interaction with the environment. In this report, we use Q-learning [1], which is one form of reinforcement learning, to investigate the learning and adaptation of agents at individual level.

Evolutionary algorithms are, on the other hand, flavors of the well known machine learning method called beam search where the machine learning evaluation metric for the beam is called the "fitness function" and the beam of the machine learning is referred to as the "population" [2]. Like other machine learning systems, evolutionary algorithms have operators that regulate the size, contents and ordering of the beam (population). We use genetic algorithms (GA), which are one form of evolutionary algorithms, to investigate the learning and adaptation of agents at population level.

In this work, we try to answer the following important questions:

1. Is the learning time required by agents shorter in continual learning in comparison to learning from scratch at both individual and population levels, and under various learning conditions?

2. Is it possible to improve the learning and adaptation capability of agents by hybridizing learning and evolutionary algorithms?

We will use agents using Q-learning, hybrid of multi-layer perceptron (MLP) and genetic algorithm, hybrid of recurrent neural networks and genetic algorithms and hybrid of Q-learning and genetic algorithm respectively in answering the above questions. The agents live and operate in an artificial robot world.

1.2 The Robot World (Test Scenario)

A deterministic world of denumerable states is used as a test scenario to investigate the learning and adaption capability of an agent. The agent is assumed to be a point robot with simplified motor actions: `left, forward` and `right` [5]. All actions can be tried in all states. The robot world and its state of transitions as a function of the present state and action taken, are shown in figure 1.1. The arrows in the cells show the orientation of the point robot when the robot finds itself in these states.

The task of the agent is to reach a given goal state through the shortest path. For reinforcement learning agents, a reward function given any current state, next state and action, s_t, s_{t+1} and a, is given by equation (1.1).

$$R^a_{s_t,s_{t+1}} = \begin{cases} 0 & \text{if} \quad s_{t+1} \neq s_t \\ 1 & \text{if} \quad s_{t+1} = \text{goal state} \\ -1 & \text{if} \quad s_{t+1} = s_t \end{cases} . \tag{1.1}$$

The negative numerical reward in equation (1.1) discourages agents attempting an action against the world boundary. This action does not change the state of the environment. For genetic algorithm, a fitness function given by equation (1.2) is used.

$$f\left(n\right) = \gamma^n. \tag{1.2}$$

The quantity f represents the fitness value of an individual, γ is a constant laying in the interval $[0, 1)$, and n is the number of steps taken by the point robot from a given start state to a given goal state. Equation (1.2) encourages those individuals that go from the start state to the goal state through a shortest path. Figure 1.2 shows a fitness function for $\gamma = 0.8$. The dynamics of the robot world, which is described by the state transitions table and the reward function, is not known to the agents a priori.

The robot world is a very highly simplified scenario of a real robot world. First, it is impossible to think a dimensionless robot or completely distinguishable states. Second, it is not possible to throw the details of low level control and deal with only simplified motor actions. Even though these assumptions are unrealistic, we base our experiments on artificial robot world due to the following justifiable reasons:

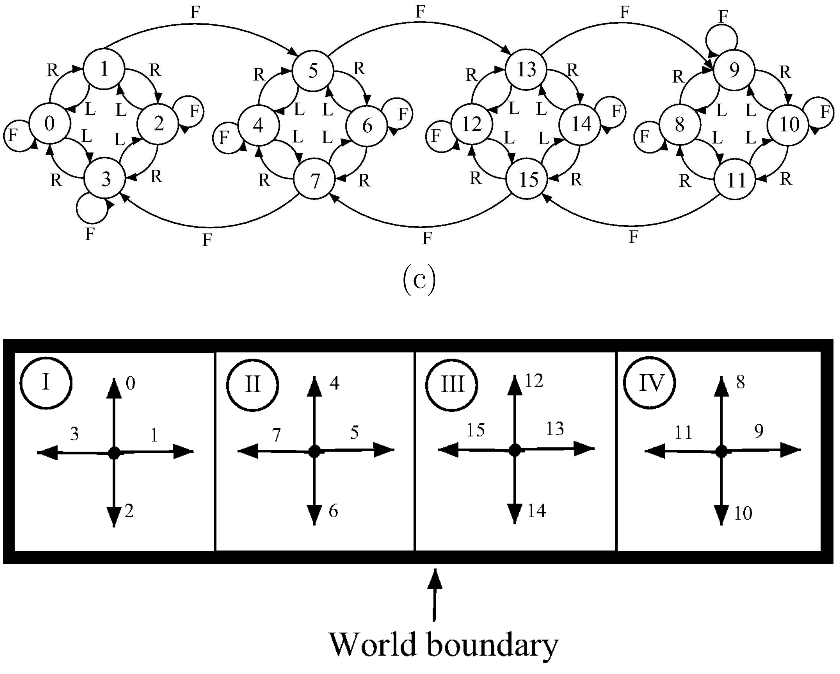
1. The experiments have to be carried out for a large number of times for different conditions of learning and adaptation experiments. This requires a lot of time and energy to execute all the experiments on real robot until one gets agents with satisfactory behaviors.

↑ 0	↑ 4	↑ 12	↑ 8
1 →	5 →	13 →	9 →
3 ←	7 ←	15 ←	11 ←
↓ 2	↓ 6	↓ 14	↓ 10

(a)

Start State	Resulting State		
	`Left`	`Forward`	`Right`
↑ 0 4 12 8	← 3 7 15 11	↑ 0 4 12 8	→ 1 5 13 9
→ 1 5 13 9	↑ 0 4 12 8	→ 5 13 9 9	↓ 2 6 14 10
← 3 7 15 11	↓ 2 6 14 10	← 3 3 7 15	↑ 0 4 12 8
↓ 2 6 14 10	→ 1 5 13 9	↓ 2 6 14 10	← 3 7 15 11

(b)



(c)

(d)

Fig. 1.1: A two-dimensional robot world (a) The robot world. The point robot must find the shortest path from any start state to the goal state. (b) The state transitions table that governs the motion of the point robot. (c) The state flow diagram of the state transitions table. The letters at the sides of the transition lines indicate the robot's motor actions F, R and L, which stand for `forward`, `right` and `left` motor actions, respectively. (d) The interpretation of the robot world. The robot world consists of four positions. In each of these positions, the robot can take one of the four orientations. A robot in state 0 or a robot in position I with orientation north will bump against the world boundary if it executes a `forward` action. In this case, the state of the robot world will not change. If it executes a `right` action, then it changes its orientation to east or goes to state 1.

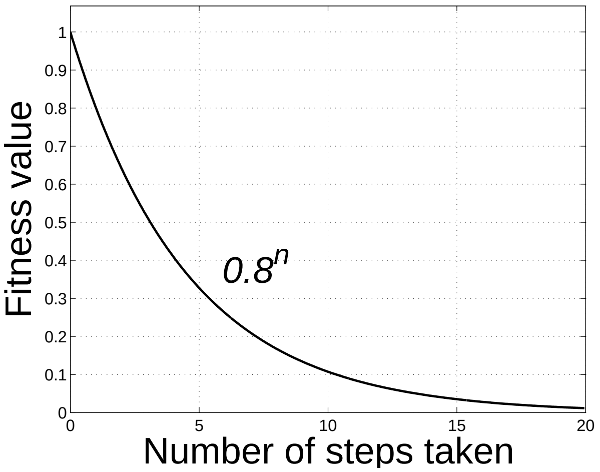


Fig. 1.2: The fitness function for $\gamma = 0.8$. The fitness value of an individual gets higher as the number of steps taken by the individual gets smaller.

2. There is a danger of coming up with wrong conclusions with experiments on real robots. This is because of the fact that noise and error makes certain parts of the agent's policy to fluctuate.

A more efficient and inexpensive method is, therefore, to run the experiments on an artificial robot world that needs much less experimental effort and yet to come up with domain free results with respect to our problem at hand. Of course, for other problems the context dependence of the behavior of the robots in real-world conditions are fundamental for their solution.

1.3 What to Learn?

In this work, the agent learns on-line through interaction with the environment either the optimal policy for perceived states or the model of the environment. A policy defines the learning agent's way of behaving at a given time. It is a mapping from perceived states of the environment to actions to be taken when in those states. The model of the environment is something that mimics the property of the environment. Given a state and an action, the model of the environment might predict the resultant next state and reward.

By optimal policy, we mean a policy that enables the agent to go from a given start state to a given goal state with minimum number of actions or steps. With genetic algorithm, the agent learns directly the optimal policy without having to learn the model of the environment. In Q-learning, the agent learns the model of the environment and saves it in a Q-table [5]. It can generate the optimal policy for perceived states from the Q-table.

1.4 Experimental Setup

The following test cases are selected for all experiments that are presented in this report. Each of the cases shows the level of the knowledge of the agent about what is going to be learned.

Test Case A

In this test case, we assume that the states of the policy that is going to be learned are completely contained in the previously learned optimal policy. For example, one of the optimal policies from start state *7* to goal state *15* contains the states *7, 4, 5, 13, 12, 15.* The sequences of actions that are contained in the policy are {*right, right, forward, left, left*}.

Assuming that the previously learned optimal policy is this policy, any policy with start state $s_{start} \in \{7, 4, 5, 13, 12, 15\}$ and goal state $s_{goal} = 15$ can be considered as a test policy, since it is known from Bellman optimality equation [5, 14] that an optimal policy with $s_{start} \in \{ 7, 4, 5, 13, 12, 15\}$ and goal state 15 has its states completely contained in one of the optimal policies with start state 7 and goal state 15.

Test Case B

Here it is assumed that the previously learned optimal policy and the policy which is going to be learned have common states. A policy with states *3, 0, 1, 5, 4, 7* generated by sequence of actions {*right, right, forward, left, left* } and a policy with states *2, 1, 5, 13, 12, 15* generated by actions {*left, forward, forward, left, left*} are good examples of policies having common states {*1, 5*}.

Test Case C

The previously learned optimal policy and the policy which is going to be learned have no common states. Examples of optimal policies which have no common states are *1, 5, 13, 9, 8, 11* generated by actions {*forward, forward, forward, left, left*} and *15, 7, 3, 0* generated by actions {*forward, forward, right*}.

For all the experiments in this report, the start and goal states {*7, 15*}, {*3, 11*} and {*15, 0*} are selected for the previously learned optimal policy for the test case A, B and C, respectively, and the start and goal states {*5, 15*}, {*7, 15*} and {*1, 11*} are selected for the optimal policy which is going to be learned for the test case A, B and C, respectively.

2. INTRODUCTION TO REINFORCEMENT LEARNING AND GENETIC ALGORITHM

2.1 Reinforcement Learning

Reinforcement learning is a type of machine learning that enables machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize their performance. A general signal measuring the quality of an action taken by an agent, called reward, is fed back to the learning algorithm. In other words, it is getting an agent to act optimally in its environment so as to maximize its rewards.

2.1.1 The Agent-Environment Interface

Reinforcement learning is one form of learning from interaction to achieve a certain predefined goal. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. The agent interacts with the environment continuously by selecting actions and the environment responds to those actions and presents the agent with a new situation. The environment also gives rise to rewards, special numerical values that the agent tries to maximize over time.

Figure 2.1 shows the agent-environment interaction. The current states, actions and rewards are represented by s_t, a_t and r_t respectively and the next states and rewards on the next time step are represented by s_{t+1} and r_{t+1}.

The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \cdots$.[1] At each time step t, the agent perceives a state of the environment. Depending on the state perceived, the agent executes an action and receives a corresponding reward.

At each time step, the agent tries to build a mapping from the states to probabilities of selecting each possible action. This mapping, which is

[1] The ideas for the discrete time can be extended to the continuous-time case.

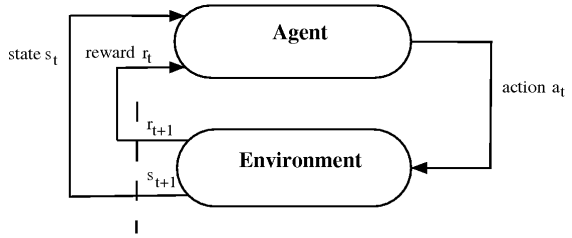


Fig. 2.1: The agent-environment interaction.

denoted by π_t, is called the agent's policy where $\pi(s, a)$ is the probability that $a_t = a$ and $s_t = s$. In reinforcement learning, the agent's goal is to maximize the reward it receives in the long run.

2.1.2 Returns

Assume that we have an agent that receives a sequence of rewards, denoted by $r_{t+1}, r_{t+2}, r_{t+3}, \cdots$, after time step t. The return, R_t, is defined as some specific function of the reward sequence. In reinforcement learning the objective of an agent is to maximize the expected return. In simplest case, the return is the sum of the rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T, \tag{2.1}$$

where T is the final time step. This is suitable for applications in which there is a natural notion of final time step. The agent-environment interaction breaks into subsequences which are called episodes (trials). Each trial ends in a special state called the terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states.

In many cases, however, the agent-environment interaction does not break naturally into distinct trials, but goes on continually without limit. For example, a control process of a robot with a long life span. These are called continuing tasks. For such tasks, the agent tries to maximize the expected discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.2}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate. It determines the present value of the future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma = 0$, the agent tries to maximize the immediate rewards: its objective in this case is to learn how to choose a_t so as to maximize only r_{t+1}. If γ approaches 1, the objective takes future rewards into account more strongly: the agent becomes more farsighted.

2.1.3 Markov Decision Process

In order to define the Markov property for a reinforcement learning problem, we assume that we have a finite number of states and reward values. The dynamics of an environment can be defined by specifying the complete probability distribution:

$$P\left\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \cdots, r_1, s_0, a_0\right\}, \tag{2.3}$$

for all s',r, and all possible values of the past events: $s_t, a_t, r_t, \cdots, r_1, s_0, a_0$. If the environment's response at $t+1$ depends only on the state and action representations at t, then the environment's dynamics can be defined by specifying only

$$P\left\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\right\}, \tag{2.4}$$

for all s', r, s_t and a_t. We say, a state signal has Markov property and a Markov state, if and only if equation (2.3) is equal to equation (2.4) for all s', r and histories, $s_t, a_t, r_t, \cdots, r_1, s_0, a_0$. A reinforcement learning task that satisfies the Markov property is called a Markov decision process or MDP. If the state and action spaces are finite, then it is called finite Markov decision process (finite MDP). A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. For a given state and action, s and a, the probability of each possible next state, s', is

$$P^a_{ss'} = P\left\{s_{t+1} = s' | s_t = s, a_t = a\right\}. \tag{2.5}$$

Equation (2.5) shows the state transition probabilities. The expected value of the next reward given any current state and action, s and a together with any next state, s', is

$$R^a_{ss'} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\}. \tag{2.6}$$

The quantities given by equation (2.5) and (2.6) completely specify the dynamics of a finite MDP. Since the point robot and the environment have a finite number of actions and states, all experiments in this report assume a finite MDP process [14, 9, 5].

2.1.4 Value Functions

Most of reinforcement learning algorithms are based on estimating value functions. The functions can be functions of states or functions of state-action pairs. They estimate how good it is for an agent to be in a given state or how good it is to perform a given action in a given state. The notion "how good" is defined in terms of the expected return. The rewards the agent expect to receive depend on what actions it will take. That means, value functions are defined with respect to particular policies.

The value of a state s under a policy π, which is defined by $V^{\pi}(s)$, is the expected return when starting in s and following π thereafter. For MDPs, it is given as

$$V^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}, \tag{2.7}$$

where E_{π} is the expected value given that the agent follows policy π. V^{π} is usually called the state-value function for policy π. The value of taking action a in state s under a policy π, denoted $Q^{\pi}(s,a)$, is the expected return starting from s, taking action a, and thereafter following policy π.

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_t | s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\}. \tag{2.8}$$

The value functions used in reinforcement learning and dynamic programming satisfy particular recursive relationships. For any policy π and state s, the following consistency condition holds between the value of s and the value of its possible successor states.

$$\begin{array}{lcl} V^{\pi}(s) & = & E_{\pi}\left\{R_t | s_t = s\right\} \\ & = & E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \\ & = & E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\right\} \\ & = & \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\right\}\right] \\ \Leftrightarrow V^{\pi}(s) & = & \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V^{\pi}(s')\right]. \end{array} \tag{2.9}$$

Similarly, it is possible to write the recursive relationship for the action-value function.

$$\begin{aligned}Q^{\pi}(s,a) &= E_{\pi}\left\{R_t|s_t=s,a_t=a\right\}\\ &= E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t=s,a_t=a\right\}\\ &= E_{\pi}\left\{r_{t+1}+\gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t=s,a_t=a\right\} \qquad (2.10)\\ &= \sum_{s'}P_{ss'}^{a}\left[R_{ss'}^{a}+\gamma E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1}=s''\right\}\right]\\ \Leftrightarrow Q^{\pi}(s,a) &= \sum_{s'}P_{ss'}^{a}\left[R_{ss'}^{a}+\gamma V^{\pi}(s')\right].\end{aligned}$$

Equations (2.9) and (2.10) are the Bellman equations for $V^{\pi}(s)$ and $Q^{\pi}(s,a)$ respectively. Figure 2.2 shows the backup diagrams for V^{π} and Q^{π}. They show the relationship between state value or action value of the current state and state values or action values of its successor states [5].

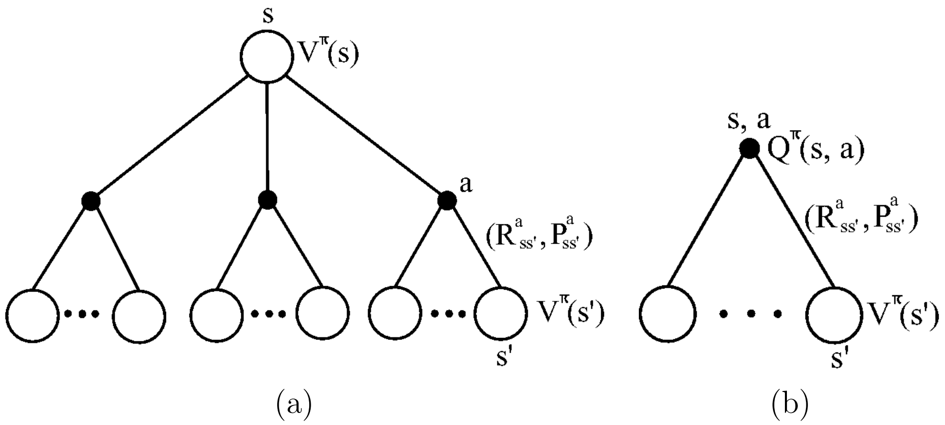


Fig. 2.2: Backup diagrams for (a) V^{π} and (b) Q^{π}.

2.1.5 Optimal Value Functions

A policy π is better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$ and $Q^{\pi}(s,a) \geq Q^{\pi'}(s,a)$ for all states and state-action pairs. There is at least one policy that is better than or equal to all other policies. This policy is the optimal policy. Although there may be more than one optimal policy, we denote all optimal policies by π^* since they share the same optimal state value function, denoted by V^*, and action-value function, denoted by Q^*. The optimal state-value function is given by

$$V^*(s) = \max_{\pi} V^{\pi}(s), \qquad (2.11)$$

for all states. Optimal policies share also the same optimal action-value function, which is given by

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \tag{2.12}$$

for all states and actions. It is possible to write equations (2.11) and (2.12) in recursive form using equations (2.9) and (2.10) as given by

$$V^*(s) = \max_{\pi} \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s')\right] \tag{2.13}$$

$$Q^*(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s',a')\right]. \tag{2.14}$$

The Bellman optimality equation (2.13) has a unique solution independent of the policy for a finite MDP problem. Actually, the Bellman optimality equation is a system of equations. If the dynamics of the environment are known ($R^a_{ss'}$ and $P^a_{ss'}$), then one can use one of the variety of methods of solving systems of nonlinear equations to solve the system of equations.

For V^*, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is attained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. This is similar to a one-step search. If we have the optimal state value function, V^*, then the actions that appear best after a one-step will be optimal actions. In other words, any policy that is greedy with respect to the optimal value function V^* is an optimal policy.

Choosing optimal actions for Q^* is still easier. The agent does not have to do a one-step-ahead search: for any state s, it can simply find any action that maximizes $Q^*(s,a)$. In other words, the agent does not need to know anything about the environment's dynamics in order to generate an optimal policy [9].

2.2 Dynamic Programming

If one has a perfect model of the environment as a Markov decision process (MDP), one uses a collection of algorithms referred to as dynamic programming [14]. We can apply dynamic programming to obtain the optimal value functions V^* and Q^*, which satisfy the Bellman optimality equations, and then the optimal policies.

2.2.1 Policy Evaluation

The process of computing the state-value function V^{π} for an arbitrary policy π is called policy evaluation. It is known that the Bellman equation (2.9) is a system of simultaneous linear equations. Its solution is straight forward, and can be found by one of the standard methods of solving a system of simultaneous linear equations.

The solution can also be found by generating a sequence of approximate value functions $V_0, V_1, V_2, \cdots$. The initial approximation, V_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for V^{π} as an update rule:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]. \tag{2.15}$$

It can be shown that $V_k \to V^{\pi}$ as $k \to \infty$. Figure 2.3 gives a complete algorithm for iterative policy evaluation.

```
Input $\pi$, the policy to be evaluated.
Initialize $V(s) = 0$ for all the states.
Repeat
    $\Delta \leftarrow 0$
    For each state $s$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^{\pi}$
```

Fig. 2.3: Iterative policy evaluation (taken from [14]).

2.2.2 Policy Improvement

Given a policy π, it is possible to find a better policy π' such that $V^{\pi'} \geq V^{\pi}$. This can be obtained by choosing deterministically an action at a particular state or by considering changes at all states and to all possible actions, selecting at each state the action that appears best according to $Q^{\pi}(s,a)$. A policy π' is greedy with respect to π if

$$\pi'(s) = \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^{\pi}(s') \right]. \tag{2.16}$$

In equation (2.16), $\arg\max_a$ denotes the value of a at which the expression that follows is maximized. The greedy policy takes the action that looks best in the short term; after one step of lookahead according to V^π. The greedy policy is as good as, or better than, the original policy.

The process of making a new policy that improves on an original policy, by making it greedy or nearly greedy with respect to the value function of the original policy, is called policy improvement.

2.2.3 Policy Iteration

We can start from a policy, π, and improve it using V^π to yield a better policy, π'. We can then compute $V^{\pi'}$ and improve it again to yield an even better policy, π''. As a result of this repeating process, we can obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_1^\pi \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*,$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). For a finite MDP, this process must converge to an optimal policy and optimal value function in a finite number of iterations. This way of finding an optimal policy is called policy iteration [14]. Figure 2.4 shows the algorithm for policy iteration.

2.2.4 Value Iteration

The value iteration algorithm follows from the recursive form of the Bellman optimal state value function (2.13). The equation that govern the value iteration is given by

$$V_{k+1}(s) = \max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V_k(s') \right]. \tag{2.17}$$

The sequence $\{V_k\}$ converges to the optimal state value V^*. Value iteration effectively combines both policy evaluation and policy improvement. The algorithm is shown in figure 2.5.

2.3 Monte Carlo Methods

Monte Carlo methods are suitable for learning from experience, which does not require prior knowledge of the environment's dynamics. These methods solve the reinforcement learning problem based on averaging sample returns.

```
1.  Initialization
    $V(s)$ and $\pi(s)$ arbitrarily for all $s$.

2.  Policy Evaluation
    Repeat
        $\Delta \leftarrow 0$
        For each state $s$:
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$ (a small positive number)

3.  Policy Improvement
    policy-stable $\leftarrow$ true
    For each state $s$:
        $b \leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right]$
        if $b \neq \pi(s)$, then policy-stable $\leftarrow$ false
    If policy-stable, then stop; else go to 2
```

Fig. 2.4: Policy iteration for V^* (taken from [14]).

There are two types of Monte Carlo methods that can be applied in estimating $V^\pi(s)$ or $Q^\pi(s,a)$: The every-visit MC method and the first-visit MC method. The every-visit MC Method estimates $V^\pi(s)$ as the average of returns following all visits to s in a set of episodes or trials. $Q^\pi(s,a)$ is estimated as the average return following all visits to the (s,a) pair in a set of episodes. On the other hand, the first-visit MC method averages just the return following the first-visit to s in estimating $V^\pi(s)$ and averages the first-visit to the (s,a) pair in estimating $Q^\pi(s,a)$. In this work, we use the first-visit MC method for estimating $V^\pi(s)$ or $Q^\pi(s,a)$. Both methods converge to $V^\pi(s)$ or $Q^\pi(s,a)$ as the number of visits to s, or (s,a) pair goes to infinity.

If the model of the environment is not available, then it is better to estimate the action values than the state values. With a model, state values are sufficient to determine a policy. It is not possible to use state values to determine a policy without having the model of the environment. Therefore, one estimates action values, which do not require the model of the environment

```
Initialize $V$ arbitrarily for all the states

Repeat
    $\Delta \leftarrow 0$
    For each state $s$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
$\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right)$
```

Fig. 2.5: Value iteration (taken from [14]).

in determining a policy.

For a deterministic policy, π, one will observe returns only for one of the actions for each state in following π. That is the Monte Carlo estimate of the other actions will not improve with experience. This is a series problem since the purpose of learning action values is to help in choosing among the actions available in each state. This implies that one needs to estimate values of all actions from each state, not the one we currently favor. To solve this problem, one can start each episode at a state-action pair, so that every such pair has a nonzero probability of being selected at a start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. This assumption is called "exploring starts".

For learning directly from real interactions with an environment, the assumption of exploring starts can not be in general relied upon. In this case, it is better to use stochastic policies with nonzero probability of selecting all actions. Figure 2.6 shows an algorithm for Monte Carlo method with exploring starts.

2.3.1 Recursive Implementation

Monte Carlo methods can be implemented recursively, on an episode-by-episode basis. This implementation enables Monte Carlo methods to process each new return recursively with no increase in computation or memory as

```
Initialize for all states $s$ and actions $a$:
    $Q(s,a) \leftarrow$ arbitrary
    $\pi(s) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list.

Repeat forever:
    (a) Generate an episode using exploring starts and $\pi$
    (b) For each pair $s$, $a$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s$, $a$
            Append $R$ to $Returns(s,a)$
            $Q(s,a) \leftarrow$ average($Returns(s,a)$)
    (c) For each $s$ in the episode:
            $\pi(s) \leftarrow \arg\max_a Q(s,a)$
```

Fig. 2.6: A Monte Carlo algorithm with exploring starts (taken from [14]).

the number of episodes increases. Suppose that we want to implement a weighted average, in which each return R_n is weighted by w_n, and we want to compute

$$Q_n(s,a) = \frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k}. \tag{2.18}$$

It is possible to write equation (2.18) in the form given by equation (2.19),

$$Q_{n+1}(s,a) = \frac{\frac{w_{n+1}}{W_n} R_n + Q_n(s,a)}{\frac{w_{n+1}}{W_n} + 1}, \tag{2.19}$$

where $W_n = \sum_{k=1}^{n} w_k$. Rewriting equation (2.19), we obtain

$$Q_{n+1}(s,a) = Q_n(s,a) + \frac{w_{n+1}}{W_{n+1}}\left[R_{n+1} - Q_n(s,a)\right]. \tag{2.20}$$

Equation (2.20) is an update rule for an action value function. In similar fashion, it is also possible to write an update rule for a state value function given by

$$V_{n+1}(s) = V_n(s) + \frac{w_{n+1}}{W_{n+1}}\left[R_{n+1} - V_n(s)\right]. \tag{2.21}$$

The quotient w_{n+1}/W_{n+1} can be considered as a step-size or learning rate, which is usually denoted by α. Replacing w_{n+1}/W_{n+1} by α in equations (2.21) and (2.20), one obtains the recursive forms of the Monte Carlo methods for $V(s)$ and $Q(s,a)$, which are given by

$$V_{n+1}(s) = V_n(s) + \alpha\left[R_{n+1} - V_n(s)\right] \text{ and} \tag{2.22}$$

$$Q_{n+1}\left(s,a\right) = Q_{n}\left(s,a\right) + \alpha\left[R_{n+1} - Q_{n}\left(s,a\right)\right]. \tag{2.23}$$

2.4 Genetic Algorithms

Genetic algorithms are computational models inspired by natural evolution. They encode a potential solution of a given problem on a simple chromosome-like data structure and apply genetic operators to these structures so as to preserve critical information.

A genetic algorithm (GA) starts with a population of chromosomes which are randomly generated. Chromosomes are then evaluated and they are given reproductive opportunities according to the result of their evaluations. Those chromosomes which represent a better solution to the target problem are given more chances to reproduce than those chromosomes which represent poorer solutions [3].

A chromosome is made up of genes. The values that can be assumed by a gene are called alleles. Genes code a specific property or component of a solution.

Genetic algorithms use two separated spaces: the search space and solution space. The search space is a space of coded solution to the problem and the solution space is the space of actual solutions. Coded solutions, or genotypes must be mapped onto actual solutions, or phenotypes before the quality of fitness of each solution can be evaluated.

2.4.1 The Algorithm

A simplest form of GA, the canonical or simple GA, is summarized in figure 2.7. A typical genetic algorithm starts with a population of randomly generated chromosomes. Each chromosome is decoded, one at a time, its fitness is evaluated, and three genetic operators, crossover, mutation and reproduction are applied followed by selection to generate a new population. This process is repeated until a desired individual is found, or until the best fitness value in the population stops increasing, or until a predefined number of generations have been produced.

2.4.2 Genetic Operators

Genetic algorithm uses its operators to find the best solution in the search space. Crossover and mutation operators maintain the variation between individuals so that children do not become identical copies of their parents.

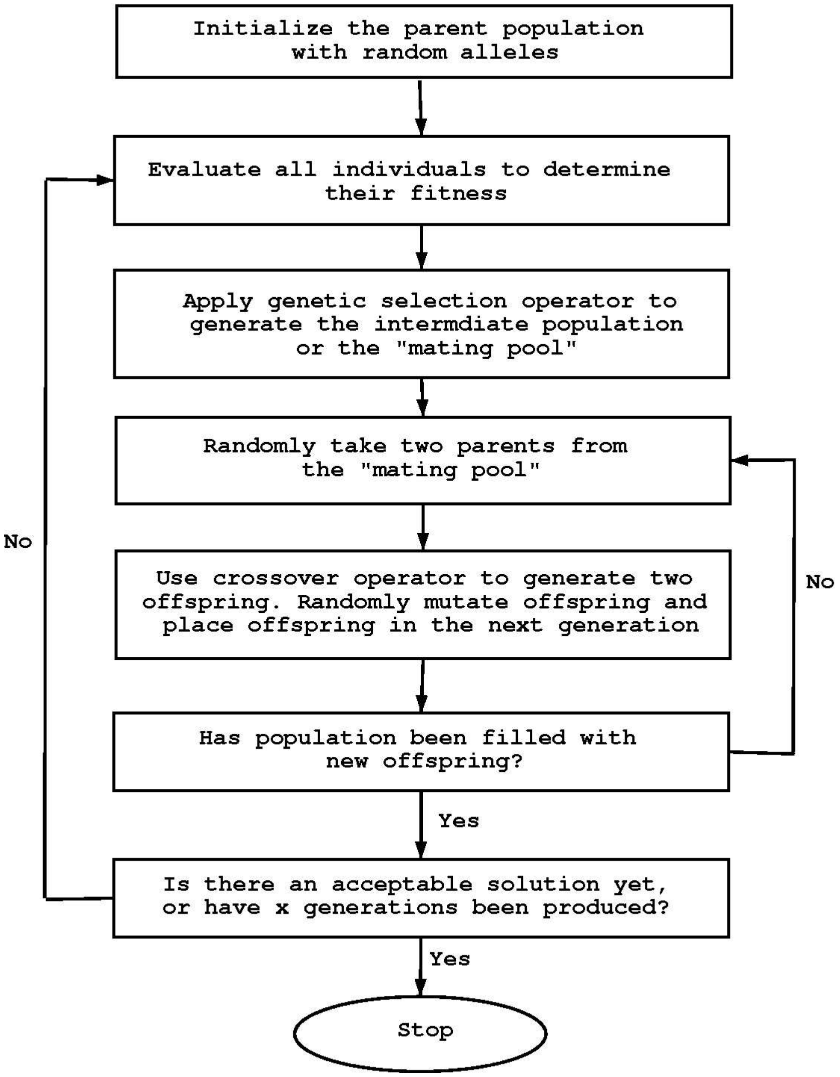

Initialize the parent population with random alleles
Evaluate all individuals to determine their fitness
Apply genetic selection operator to generate the intermdiate population or the "mating pool"
Randomly take two parents from the "mating pool"
Use crossover operator to generate two offspring. Randomly mutate offspring and place offspring in the next generation
Has population been filled with new offspring?
Yes
No
Is there an acceptable solution yet, or have x generations been produced?
Yes
No
Stop

Fig. 2.7: The simple genetic algorithm.

This variation between individuals helps the population to keep on improving from generation to generation.

Crossover Operators

These operators are used to exchange genetic material between two chromosomes. They are used to exploit the genetic material contained in the population of the chromosomes. The most common types of crossover operators are 1-point and 2-point crossover operators. With 1-point crossover operator, a crossover point is chosen randomly along the chromosomes and everything either before the crossover point or after the crossover point is exchanged between the chromosomes. With 2-point crossover operator, two crossover points are chosen along the chromosomes and everything between the crossover points, or everything before the first crossover point and after the second crossover point is exchanged. Figure 2.8 shows examples of the 1-point and 2-point crossovers.

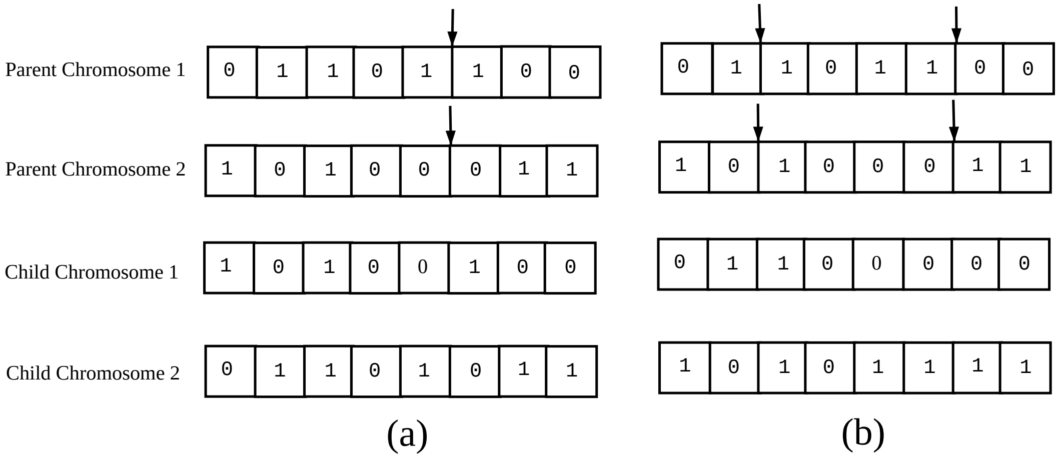


Fig. 2.8: 1-point (a) and 2-point (b) crossover operators. The arrows show crossover points.

Mutation Operators

Mutation operators are used to introduce a new genetic material into chromosomes. They help the genetic algorithm not to converge to a sub-optimal solution. They are used by the genetic algorithm to explore the search space. For binary chromosomes, the mutation operator flips bits contained in the genes of chromosomes. For chromosomes containing genes made up of real values, mutation is performed by adding normally distributed random numbers with expectation value 0. Figure 2.9 shows the effect of mutation operator on binary chromosomes.

Reproduction Operators

These operators are straightforward. They select an individual, copy it and place the copy into the mating pool.

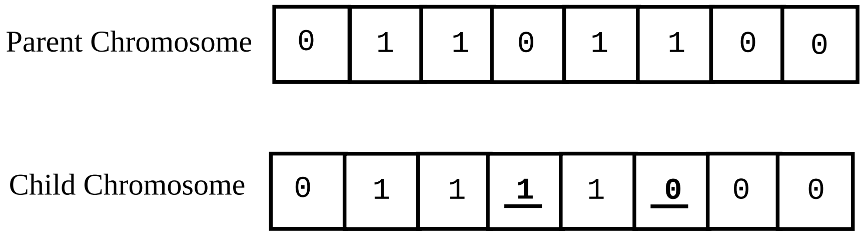


Fig. 2.9: Effect of mutation operator for binary chromosomes. The underlined bits show bits that are flipped.

2.4.3 The Selection Algorithm

Selection is a consequence of competition between individuals in a population. This competition results from an overproduction of individuals which can withstand the competition of varying degrees. The search for an optimal solution is directed by the "survival of the fittest" principle. This principle comes into play when we decide which chromosomes can join the mating pool and hence be parents for the next generation. This decision process is controlled by selection operators.

Fitness-Proportional Selection

Fitness-proportional selection specifies probabilities for individuals to be given a chance of passing offspring into the next generation. An individual i is given a probability of

$$p_i = \frac{f_i}{\sum_j f_j} \tag{2.24}$$

for being able to pass on traits. The value f is the fitness value of an individual. Following Holland [7], fitness-proportional selection has been the tool of choice for a long time in the GA community. It has been heavily criticized in recent times for attaching differential probabilities to the absolute value of fitness. Early remedies for this situation were introduced through fitness scaling, a method by which absolute fitness were made to adapt to the population average.

Truncation or (μ, λ) Selection

This is the second most popular method of selection. A number μ of parents are allowed to breed λ offspring, out of which the μ best are used as parents for the next generation. A variant of the (μ, λ) selection is $(\mu + \lambda)$ selection where, in addition to offspring, the parents participate in the selection process. The truncation selection methods are not dependent on the absolute fitness values of individuals in the population. The μ best will always be the best, regardless of the absolute fitness differences between individuals.

Tournament Selection

This type of selection is not based on competition within the full generation but in a subset of the population. A number of individuals, called the tournament size is selected randomly, and a selective competition takes place. The better individuals in the tournament are then allowed to replace those of the worse individuals. The tournament size is used to control the selection pressure. A small tournament size causes a low selection pressure and a large tournament size causes high selection pressure.

Tournament selection has recently become a mainstream method for selection, mainly because it does not require centralized fitness comparison between all individuals. This not only accelerates evolution considerably, but also provides an easy way to parallelize the algorithm [2].

Ranking Selection

Ranking selection is based on the fitness order, into which the individuals can be sorted. The selection probability is assigned to individuals as a function of their rank in the population. Mainly, linear and exponential ranking are used. For linear ranking, the probability is a linear function of the rank,

$$p_i = \frac{1}{N}\left[p^- + \left(p^+ - p^-\right)\frac{i-1}{N-1}\right], \tag{2.25}$$

where p^-/N is the probability of the worst individual being selected, and p^+/N is the probability of the best individual being selected, and

$$p^- + p^+ = 2 \tag{2.26}$$

should hold in order for the population size to stay constant.

For exponential ranking, the probability can be computed using a selection bias constant c,

$$p_i = \frac{c-1}{c^{N-1}}c^N - i, \tag{2.27}$$

with $0 < c < 1$.

2.5 Finding an Optimal Policy in an Artificial Robot World

In this section, we want to illustrate the concepts that are presented in this chapter taking as an example the test scenario discussed in section 1.2. We make the following assumptions before we start to find the optimal policy:

1. The point robot (the agent) has a predefined goal state. In our case we take state 15 as the goal state.
2. The optimal policy is determined off-line. That is, the dynamics of the environment is known a priori to the agent.

The dynamics of the environment in which the agent live and operate are determined using equations (2.5) and (2.6). For our test scenario, the dynamics of the environment is given by equation (2.28).

$$\begin{aligned} P^a_{ss'} &= \begin{cases} 1 & \text{if } s' \text{ is a valid next state} \\ 0 & \text{otherwise} \end{cases} \\ R^a_{ss'} &= \begin{cases} 0 & \text{if } s_{t+1} = s' \\ 1 & \text{if } s_{t+1} = \text{goal state} \\ -1 & \text{if } s_{t+1} = s_t \end{cases} \end{aligned} \tag{2.28}$$

We can apply the optimal Bellman equations to solve for the optimal state-value function V^* or the optimal action-value function Q^*. The optimal Bellman system of equations for the goal state 15 is

$$\begin{aligned} V^*(0) &= \max \begin{cases} \gamma V^*(3) & \text{left} \\ \gamma V^*(1) & \text{right} \\ -1 + \gamma V^*(0) & \text{forward} \end{cases} \\ V^*(1) &= \max \begin{cases} \gamma V^*(0) & \text{left} \\ \gamma V^*(2) & \text{right} \\ \gamma V^*(5) & \text{forward} \end{cases} \\ &\vdots \\ V^*(15) &= \max \begin{cases} \gamma V^*(14) & \text{left} \\ \gamma V^*(12) & \text{right} \\ \gamma V^*(7) & \text{forward} \end{cases} \end{aligned} \quad . \tag{2.29}$$

In equation (2.29), γ is the discounting factor. It is set to 0.8 for this example. The words left, right and forward show the possible motor actions of the point robot. The equation has a unique solution that is independent of a particular optimal policy. If one tries to apply exhaustive search for finding all policies which result in the same optimal state value function, one has to solve $3^{16} = 43046721$ systems of simultaneous equations to get the 32 optimal policies. Assuming that we need $1\mu s$ to solve one system of simultaneous equations, we need about 43 seconds to solve all systems of simultaneous equations in order to find all optimal policies. For a backgammon game, for example, which has about 10^{29} states, it would take millions of years on today's fastest computers to solve the Bellman equation for V^* [14]. In general, we can use dynamic programming methods (either value iteration or policy iteration) to solve MDPs with millions of states using today's computers.

We have applied the value iteration algorithm (dynamic programming) and found an optimal state value function shown in table 2.1 in only 20 iterations, for an absolute error of 10^{-50}. From the table, it is possible to get all the 32 optimal policies by using a one-step search. For example, for state 0 the optimal action is right since the action right will move the point robot to state 1, which is a valid next state with the largest state value. A state value of a state measures "how good" it is for an agent to be in that state. From the result obtained, we see that the state value of state 3 is worst for the goal state 15. This means that no matter which starting action the agent takes from this state, it needs the largest number of steps to reach the goal state as compared to starting from other states. One can also see that it is best for an agent to be in the states 11, 12 or 14, since the agent needs to execute only one optimal action (minimum number of actions) to reach the goal state.

$\uparrow^0$ (1.13778)	$\uparrow^4$ (1.42222)	$\uparrow^{12}$ (2.77778)	$\uparrow^8$ (2.22222)
$\rightarrow^1$ (1.42222)	$\rightarrow^5$ (1.77778)	$\rightarrow^{13}$ (2.22222)	$\rightarrow^9$ (1.77778)
$\leftarrow^3$ (0.91022)	$\leftarrow^7$ (1.13778)	$\leftarrow^{15}$ (2.22222)	$\leftarrow^{11}$ (2.77778)
$\downarrow^2$ (1.13778)	$\downarrow^6$ (1.42222)	$\downarrow^{14}$ (2.77778)	$\downarrow^{10}$ (2.22222)

Tab. 2.1: The optimal state value for the goal state 15.

We can also solve equation (2.29) using genetic algorithm. A chromosome containing 16 genes is defined, where a gene codes an action, which moves a point robot to the next state having the maximum state value. In other words, a chromosome codes directly a policy and we want to use a genetic algorithm to search for an optimal policy. An example of a chromosome coding a system of simultaneous equations (policy) is shown in figure 2.10. The index of a gene along the chromosome is the same as the corresponding state in the robot world.

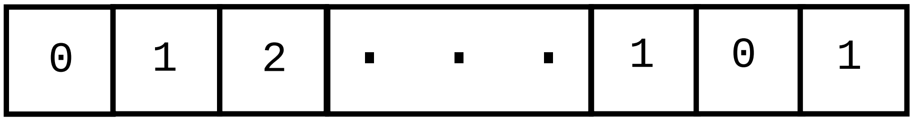


Fig. 2.10: A chromosome coding a system of simultaneous equations. A gene can take a value of 0, 1 or 2 representing an equation corresponding to left, right and forward motor actions respectively of the point robot.

In order to find a system of simultaneous equations whose solution is an optimal state value function, we have to define a fitness function evaluating an equation. We can use equation (1.2) for evaluating a system of simultaneous equations in such a way that the equation is used repeatedly for each starting state. The fitness function evaluating an equation is thus given as

$$f = \sum_{s=0}^{16} \gamma^{n_s}, \tag{2.30}$$

where n_s represents the number of steps taken by the point robot from a starting state s. Table 2.2 shows the parameters of the genetic algorithm used in finding the system of simultaneous equations, whose solution is the optimal state value function.

Number of individuals	50
Crossover probability	0.2
Mutation probability per gene	0.05
Selection method	Truncation selection
Number of generations	50

Tab. 2.2: Parameters of genetic algorithm used.

We have run the algorithm and found the best system of simultaneous equations (policy) after 11 generations. The best system of simultaneous equations found by the genetic algorithm is given by equation (2.31).

$$\begin{aligned} V^*(0) &= \gamma V^*(1)\,\text{right} \\ V^*(1) &= \gamma V^*(5)\,\text{forward} \\ &\vdots \\ V^*(15) &= \gamma V^*(14)\,\text{left} \end{aligned} \tag{2.31}$$

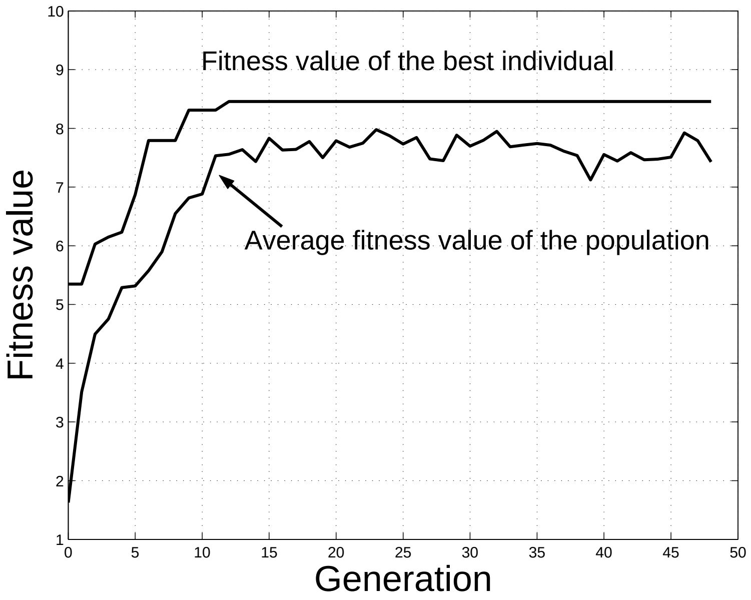


Fig. 2.11: A genetic algorithm run for finding the best system of simultaneous equations. The best equation is found after the 11^{th} generation.

The solution of equation (2.31) is the same as the solution found by applying value iteration algorithm, which is shown in table 2.1. As compared to the value iteration algorithm, the genetic algorithm is slower since it has solved $50 \times 11 = 550$ equations before it obtained an equation whose solution is the optimal state value function.

The Monte Carlo algorithm with exploring starts shown in figure 2.6 is also used to find the optimal action values. The algorithm needed about 10000 iterations to get the action values, from which one can generate one of the optimal policies for the goal state 15. As compared to the genetic algorithm used, Monte Carlo methods needed much longer time to get the optimal action values.

From this example, one can conclude that it is possible to solve the Bellman optimality equations in different ways. If the dynamics of the environment is known a priori, then dynamic programming can be used to get the solution faster than genetic algorithms or Monte Carlo methods. Genetic algorithms and Monte Carlo methods does not necessarily require the knowledge of the dynamics of the environment a priori. Genetic algorithm can directly search for the optimal policy in the space of policies. But Monte Carlo methods can estimate the action values (model of the environment) from experience. One can then generate the optimal policy from the estimated actions values. For an environment with a very large number of states such as backgammon, it is only possible to solve the optimal Bellman equation approximately in a given limited time.

3. LEARNING AND ADAPTATION AT INDIVIDUAL LEVEL

3.1 Introduction

Organisms, for example human beings, are always learning and adapting to their environment in their life time. Much of the learning is done through direct interactions with the environment. Consider a person who can not ride a bicycle. Let us say that this person wants to learn how to ride a bicycle. The first thing he does is he asks about how to ride a bicycle. But only telling him about how to ride a bicycle will not help him to ride the bicycle at the first trial. The only way to learn to ride a bicycle is, therefore, to try and have a real experience with the bicycle. This person has to do a lot of trials before he learns how to ride a bicycle. Of course, the number of trials made is dependent on the individual. Each of the trials made by the person, whether it is successful or not, can be evaluated by the person since he knows how well he has ridden the bicycle. Assuming that the bicycle is the environment and the person is the agent, the notion "how well" is the reward the person receives from the environment after having a trial. Each of the trials is made up of a sequence of actions that are executed by the person in riding the bicycle. The state of the bicycle can be the tilt angle and speed of the bicycle relative to the ground. Depending on the reward received and the state of the bicycle, the person has to execute sequence of actions to keep the bicycle upright and moving forward with a certain speed. In this chapter, we use Q-learning, which is one from of reinforcement learning, to investigate the learning and adaptation capability of agents that learn through interaction with the environment and from experience.

The following assumptions are made for experiments in this chapter and the following chapters.

1. The agent uses learning from interaction. That is, it uses an action-perception cycle.

2. The agent does not know the dynamics of the environment a priori.

Moreover, the agent tries to learn an optimal policy or the model of the environment only for perceived states.

3.2 Q-Learning

Q-learning is an on-line learning method, in which the agent learns from experience to act optimally in a given environment. The agent learns the model of the environment and saves it in the action-value function (Q-table). The agent uses the action-value function to generate the optimal policy for a given start and goal state.

Q-learning has the properties of both dynamic programming and Monte Carlo methods. It bases itself on the recursive implementation of the Monte Carlo method and uses the optimal Bellman equation to update the action-value of the current state. This can be shown as follows: The recursive implementation of the Monte Carlo method can be written as

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - Q\left(s_t, a_t\right)\right], \tag{3.1}$$

which is equivalent to

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} - Q\left(s_t, a_t\right)\right]. \tag{3.2}$$

With on-line learning, the agent can not receive all rewards. It can only receive the current reward for the current action. The term $\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}$ is the return for the next state and action. That is $Q\left(s_{t+1}, a_{t+1}\right) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}$. Replacing $\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}$ by $Q\left(s_{t+1}, a_{t+1}\right)$, we get equation (3.3).

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right] \tag{3.3}$$

If we want equation (3.3) to converge to the optimal action-value function, Q^*, then we have to select the maximum value of the action-values of the next state. This follows directly from the optimal Bellman equation for Q^*,

$$Q\left(s_{t+1}, a_{t+1}\right) = \max_a Q\left(s_{t+1}, a\right). \tag{3.4}$$

Using equation (3.4), we obtain an equation for the Q-learning algorithm,

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right]. \tag{3.5}$$

For a right convergence to the optimal action-value function, the agent has to update its action-vale function for all state-action pairs for the perceived states. In other words, the agent has to explore its environment and at the same time exploit what it has learned so far [1].

3.3 Exploration and Exploitation

With reinforcement learning, specially with on-line reinforcement learning, there is a problem of exploration and exploitation. On the one hand, the agent wants to explore the environment so as to find the optimal solution. On the other hand, the agent wants to minimize the cost of learning by exploiting the environment.

There are a lot of methods that balance the exploration and exploitation. The simplest and most popular form of balancing the exploration and exploitation is the so called ϵ-greedy-action selection method. In this method, an action is selected greedily most of the time. But every once in a while with small probability ϵ, an action is selected at random, uniformly, independently of the action-value estimates.

The other popular action selection mechanism is the softmax action selection method. The probability of executing an action is determined by a graded function of the estimated values. The greedy action is still given the highest selection probability. But all the others are ranked and weighted according to their value estimates. The Boltzmann distribution is used to calculate the action selection probability. Let A be a set of all actions. The probability of executing an action $a \in A$ is given by the following equation,

$$P\left(a\right) = \frac{e^{-Q(s,a)/\tau}}{\sum_{a' \in A} e^{-Q(s,a')/\tau}} \tag{3.6}$$

where τ is a positive parameter called temperature. High temperatures cause the action to be nearly equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates.

3.4 Experiments and Results

The experiments are done for all test cases mentioned in section 1.4. Table 3.1 shows the parameter of the Q-learning algorithm used. For balancing the exploration and exploitation of the environment, we have used a simple ϵ-greedy-action selection method. The reward function given by equation (1.1) is used to evaluate the actions executed by the agent.

α	γ
0.3	0.3

Tab. 3.1: Parameters of the Q-learning algorithm.

Test Case A

In this test case, the states of the policy which is going to be learned are completely contained in the previously learned optimal policy.

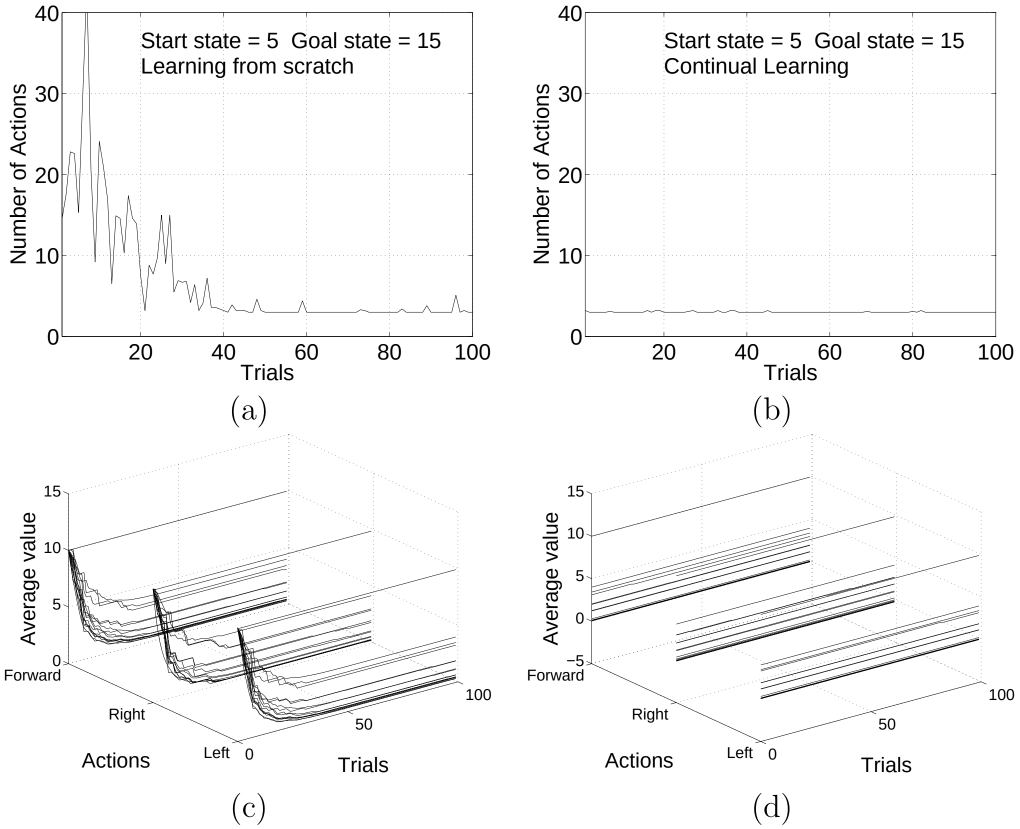
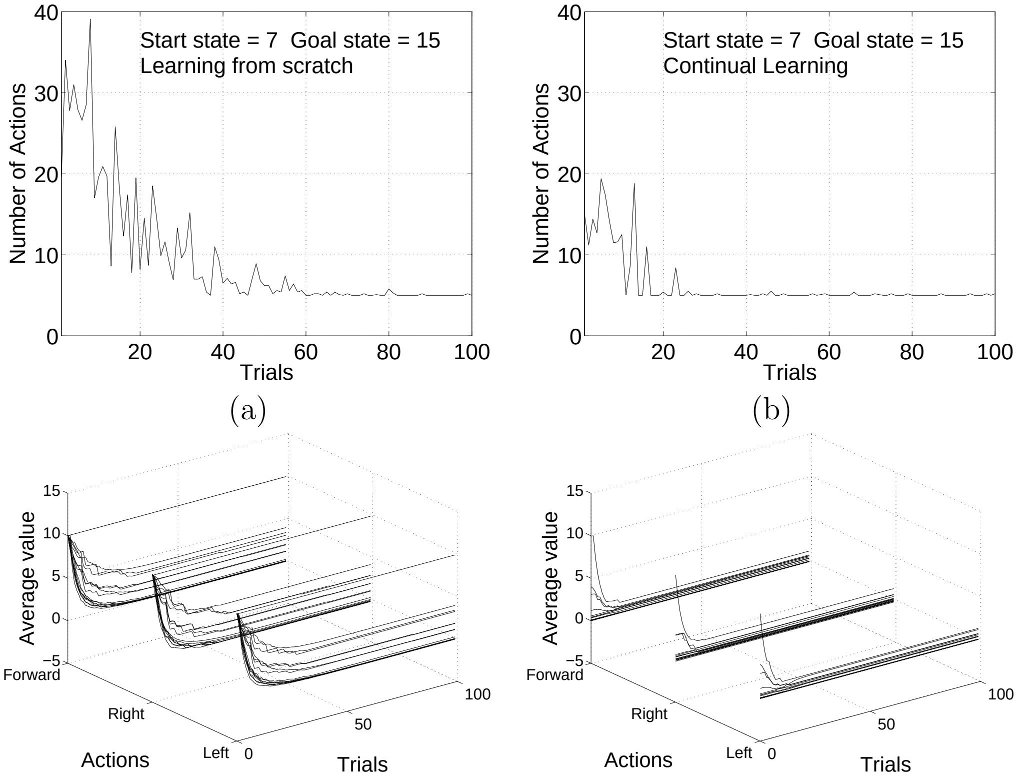


Fig. 3.1: Learning from scratch and continual learning. (a) Average number of actions taken versus trials in learning from scratch. (b) Average number of action taken versus trials in continual learning. (c) Average action values for each state in learning from scratch. (d) Average action values for each state in continual learning.

From figure 3.1, one can see that the agent does not need to learn the new optimal policy in continual learning. This is due to the fact that the states of the new optimal policy are completely contained in the previously learned optimal policy. One can also see that action values, that represent the learned model of the environment, remain the same in the continual learning.

Test Case B

In test case B, the previously learned optimal policy and the optimal policy that is going to be learned have some states in common.



(a) (b)

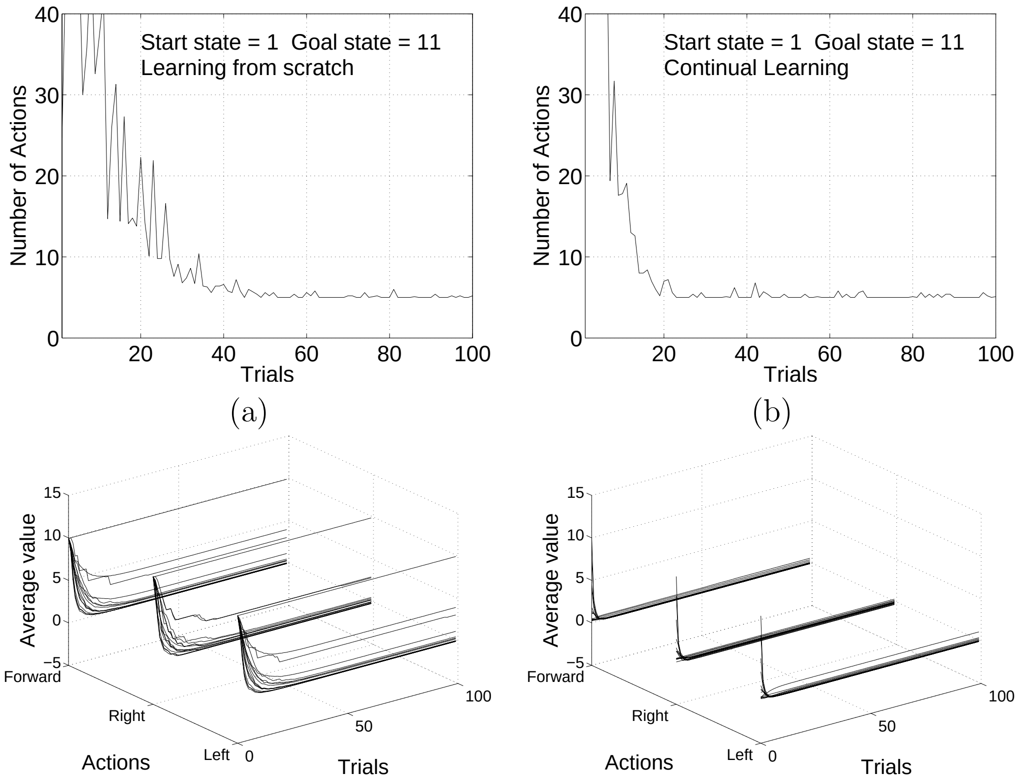
(c) (d)

Fig. 3.2: Learning from scratch and continual learning. (a) Average number of actions taken versus trials in learning from scratch. (b) Average number of action taken versus trials in continual learning. (c) Average action values for each state in learning from scratch. (d) Average action values for each state in continual learning.

Figure 3.2 shows that the learning time required by the agent in continual learning is shorter than that required in learning from scratch. The action values are adjusted by learning the action values for the new optimal policy in the continual learning accordingly.

Test Case C

Here, the previously learned optimal policy and the optimal policy that is going to be learned have no states in common.



(a) (b)

(c) (d)

Fig. 3.3: Learning from scratch and continual learning. (a) Average number of actions taken versus trials in learning from scratch. (b) Average number of action taken versus trials in continual learning. (c) Average action values for each state in learning from scratch. (d) Average action values for each state in continual learning.

As can bee seen in the figure 3.3, even though the previously learned optimal policy and the optimal policy that is going to be learned have no common states, the learning time in continual learning is shorter than the learning time in learning from scratch. This is possible due to the fact that the agent has collected experience about other states, which are not contained in the previously learned policy, while learning it.

From the experiments, we have concluded that the learning time in continual learning is shorter than the learning time in learning from scratch at an individual level and under different learning conditions. Moreover, the different test cases suggest how we may bias (put prior knowledge to) agents that learn from experience. If we bias an agent in such a way that the states of the policy that is going to be learned are completely contained in the optimal bias policy, then there is nothing left for the agent to learn and the bias is strong. If the bias policy and the policy that is going to be learned have no common states, a relatively large amount of information is left for the agent to learn. This shows that the amount of information that is going to be learned depends on the number of common states between the optimal bias policy and the policy that is going to be learned. The more common states the optimal bias policy and the policy that is going to be learned have, the less information is left for the agent to learn.

4. LEARNING AND ADAPTATION AT POPULATION LEVEL

4.1 Introduction

Populations of organisms have been adapting to their particular environmental conditions through evolutionary selection (survival of the fittest) and variability among them. Those members of organisms with specific advantageous abilities and features are able to cope with their environmental conditions. From the principles of adaptation in nature, we can drive a number of concepts and strategies for solving learning tasks and develop optimization problems for artificially intelligent systems [8]. An example of optimization problem that can be solved using the principles of evolution is a model based object recognition system.

In this chapter, we investigate learning and adaptation at population level, where the population is made up of neural networks. The neural networks are used to represent the optimal policy (control) that is going to be learned. The purpose of the genetic algorithm is to search for the best neural network (policy or controller) that controls the point robot in the artificial robot world. We use multi-layer perceptrons (MLP) and recurrent neural networks for our experiments. The genetic algorithm searches for the best neural network by directly determining the synaptic weights of the networks.

According to [10], the majority of experiments in evolutionary robotics have resorted to neural networks for the control system of an evolving robot due to the following justifiable reasons:

1. Neural networks provide a straightforward mapping between sensors and motors. That is, they can represent the policy (control) to be learned. They can accommodate continuous (analog) or discrete input signals and provide either continuous or discrete motor outputs, depending on the transfer function chosen. For example, the neural networks used in experiments run in this chapter have discrete input (states) and outputs (motor actions).

2. Neural networks offer a relatively smooth search space. Gradual changes to the parameters defining a neural network (synaptic weights, architecture, etc) will often correspond to gradual changes of its behavior.

3. Neural networks provide various levels of granularity. One can apply artificial evolution to the lowest level specification of a neural network, such as the connection strengths, or to higher levels, such as the coordination of predefined modules composed of predefined sub-networks.

4. Neural networks allow different levels of adaptation. For example, the blueprint of a network architecture may be evolved, its actual structure may develop during the initial stage of the robot "life," and its connection strengths may adapt in real time while the robot interacts with the environment.

5. Neural networks are robust to noise. Since their units are based upon a sum of several weighted signals, oscillations in the individual values of these signals do not drastically affect the behavior of the network. This is very useful property for physical robots with noisy sensors that interact with noisy environments.

6. Neural networks can be a biologically plausible metaphor of mechanisms that support adaptive behavior. They are a natural choice for understanding biological phenomena from an evolutionary perspective.

It was, however, suggested by some authors that evolution of neural networks is made difficult by the problem of "competing conventions"[12]. This is the case where the mapping between genotype and phenotype is many to one. In addition, crossover among competing conventions may produce offspring with very low fitness since they have duplicated structures. But experimental studies have shown that in practice this is not a problem. In general, it is wise to use small crossover rate when evolving neural networks [10].

4.2 Experiments and Results

The experiments are run for multi-layer neural networks (MLP) and for both architectures of the recurrent neural networks (Jordan and Elman). In the experiments, the weights of the neural networks are directly determined by the genetic algorithm.

4.2.1 Multi-layer Perceptron

In this section, we have used multi-layer perceptrons (MLPs) to represent the optimal policy. A population of MLPs with two layers forms a population of controllers. The structure of the networks and the number of hidden units is fixed but the weights are determined directly by the genetic algorithm. The MLP controls the point robot in the robot world. The genetic algorithm lets each individual to control the point robot and evaluates and selects an individual (controller) that moves the point robot from a given start state to a given goal state with minimal number of steps. It then applies genetic operators to generate the next population of MLPs for predefined number of trials.

Figure 4.1 shows an example of the MLP used in this experiment. Table 4.1 shows the encoding of states and actions, which are the input and output of the neural network, respectively.

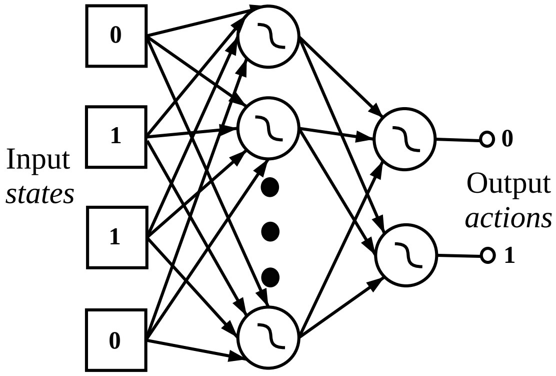


Fig. 4.1: The MLP used in the experiment. The input and output of the MLP are binary codes of the states and actions.

A fitness function given by equation (1.2) is used to evaluate the individuals. An example of a chromosome representing an MLP (an individual) is shown in figure 4.2. The parameters of the neural network and genetic algorithms are given in table 4.2.

We have run the experiment for all test cases and obtained the result shown in figure 4.3. As can be seen in the figure, the population attains a certain average fitness value. The average fitness value, which is controlled by the genetic operators, shows an equilibrium point of two ”forces”. One of the forces, which is controlled by selection operator, tries to pull the population towards the global maximum fitness value (fitness value of the best individual) and the other force, which is controlled by the crossover and mutation

State	**Code**
0	000
⋮	⋮
14	1110
15	1111

(a)

Action	**Code**
left	00
right	01
forward	10
don't care	11

(b)

Tab. 4.1: The encoding of the states (a) and actions (b).

$Wo_{1,1}$	···	$Wo_{1,N}$	$Wo_{2,1}$	···	$Wo_{2,N}$	$Wh_{1,1}$	···	$Wh_{1,4}$	$Wh_{N,1}$	···	$Wh_{N,4}$

Fig. 4.2: A chromosome encoding an MLP. Wo's show the synapses going to the output units and Wh's show synapses going from input to hidden units. N is the number of hidden units. In this experiment, we used two output and four input units, and six hidden nodes.

operators, tries to maintain the variation between individuals. The learning time, which is measured in number of generations, required in attaining a certain average fitness value is shorter in continual learning than the learning time in learning from scratch for all test cases.

4.2.2 The Jordan Recurrent Neural Network

The Jordan neural network shown in figure 4.4 is used in this experiment. In this neural network, the outputs of the network at previous time step are fed back to the network as an input. The number of inputs, outputs and hidden nodes is the same as that of the multi-layer perceptron used in the above experiment. The encoding of the states (inputs) and motor actions (outputs) is as shown in table 4.1.

The fitness function given by equation (1.2) is used to evaluate a neural network. The parameters of the genetic algorithm and the neural network are shown in table 4.2.

The experiment is run for all test cases and the result shown in figure 4.5 is obtained.

From figure 4.5, one can see that the learning time, which is measured in generations, in case of learning from scratch is shorter than the learning time in continual learning. Moreover, it is clear that the population does not need to learn the new optimal policy in test case A. This result is the same as

Number of individuals	50
Crossover probability	0.2
Mutation probability per bit	0.05
Selection method	Truncation selection
Number of hidden nodes	6
Number of bits per gene coding a synapse	8
Number of generations	100

Tab. 4.2: Parameters of the MLPs, Jordan and Elman neural networks and genetic algorithm used for all the experiments run in this chapter.

that obtained for the hybrid of the MLP and genetic algorithm. In test case C, even though the previously learned optimal policy and the policy that is going to be learned have no common states, the population has learned the new optimal policy in less number of generations in continual learning than in learning from scratch.

4.2.3 The Elman Recurrent Neural Network

In this experiment, we have used the same parameters as that of the Jordan neural networks that are shown in table 4.2. The Elman neural network used in this experiment is shown in figure 4.6. For this network, the activations of the hidden units at previous time step are fed back to the network as an input. Note that the memory units hold a copy of the activations of hidden units at the previous time step. The same fitness function given by equation (1.2) is used to evaluate the neural networks. The experiment is run for all test cases and results shown in figure 4.7 are obtained.

The results shown in figure 4.7 show that the learning time is shorter in continual learning than in learning from scratch. As can be seen from the figure, the population does not need to learn the new policy in continual learning for test case A. For test case C, the population has learned the new policy faster in continual learning even though the previously learned policy and the policy that is going to be learned have no common states.

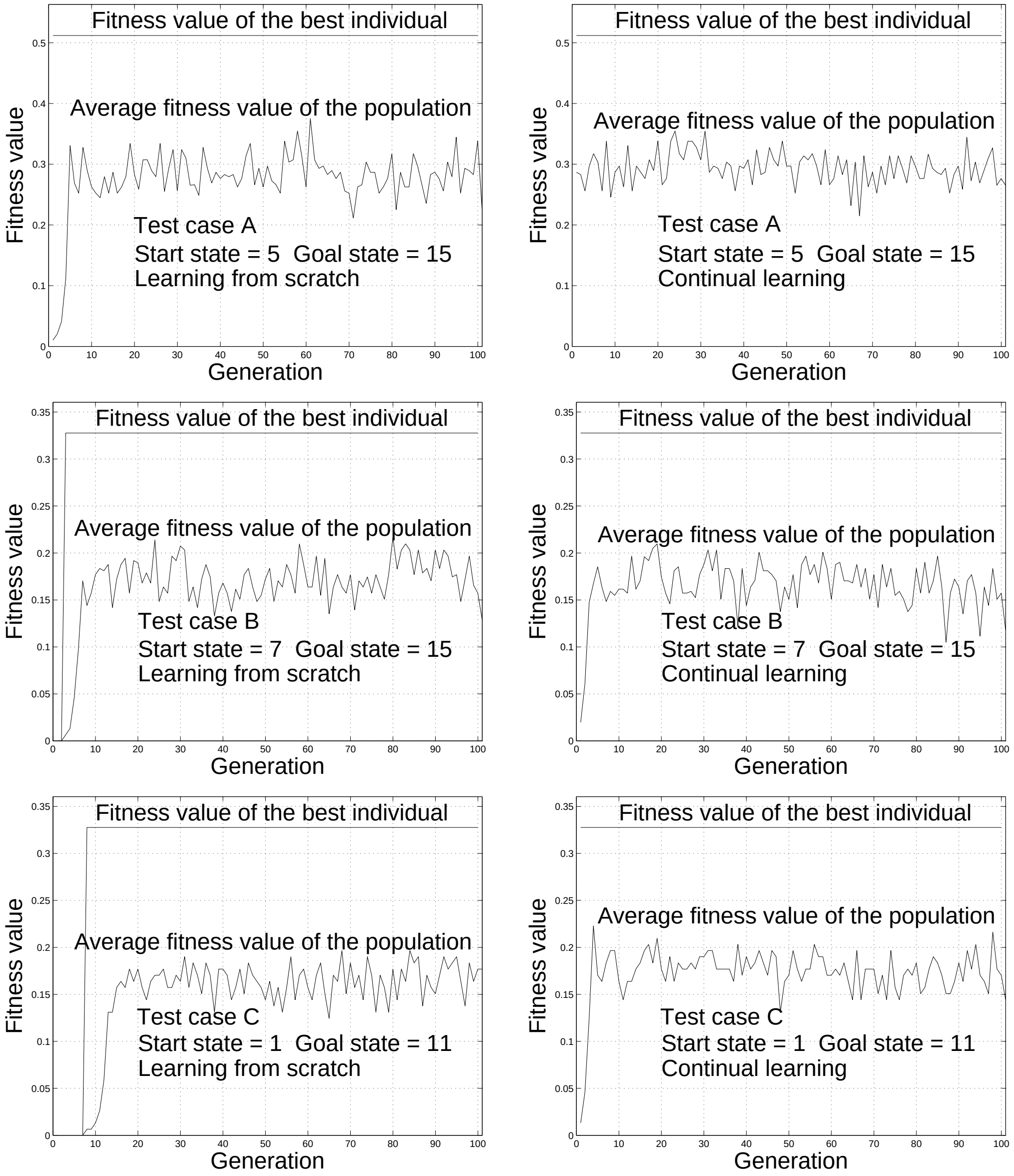


Fig. 4.3: Result obtained for a hybrid of MLPs and genetic algorithm. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

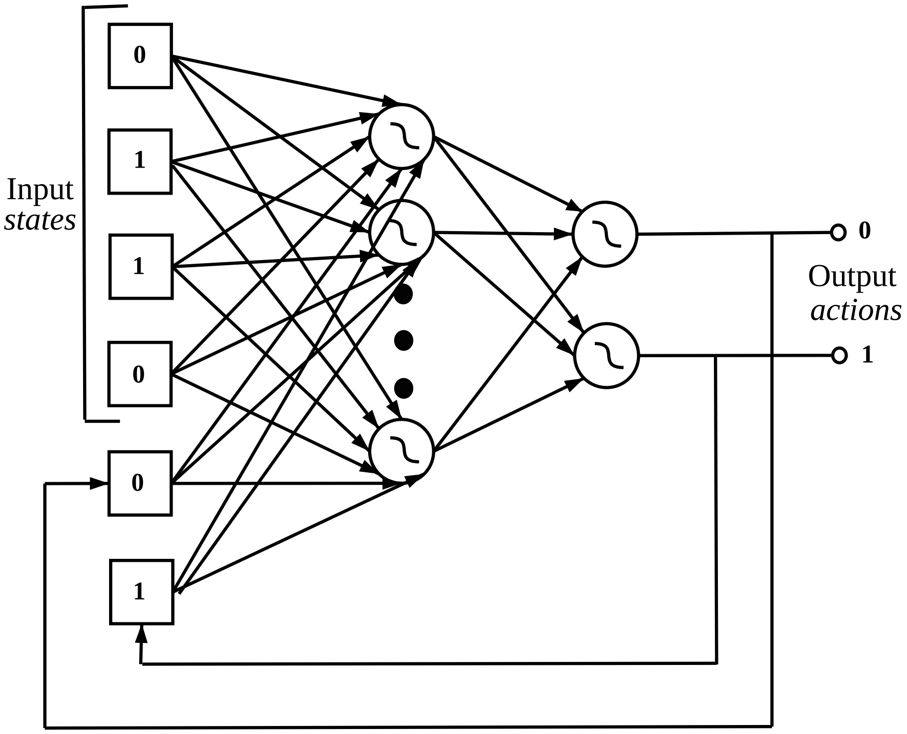


Fig. 4.4: The Jordan neural network used in the experiment.

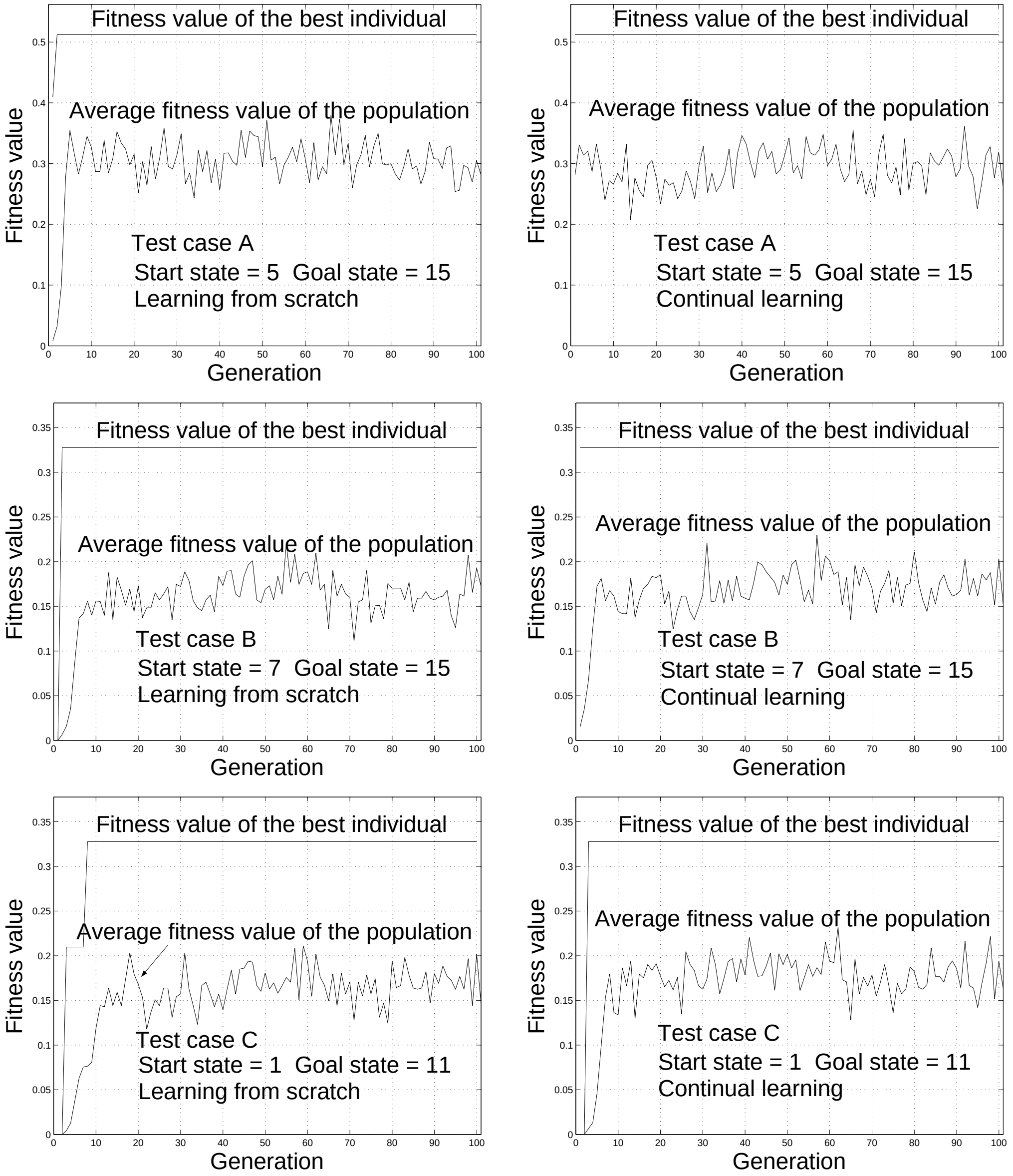


Fig. 4.5: Result obtained for a hybrid of Jordan neural networks and genetic algorithm. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

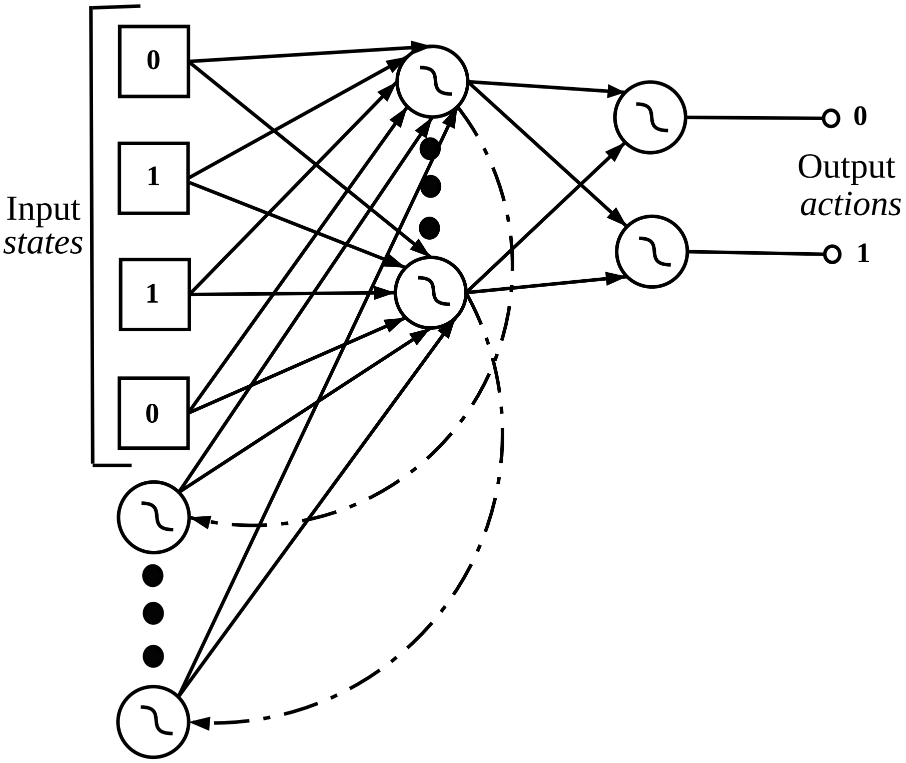


Fig. 4.6: The Elman neural network used in the experiment.

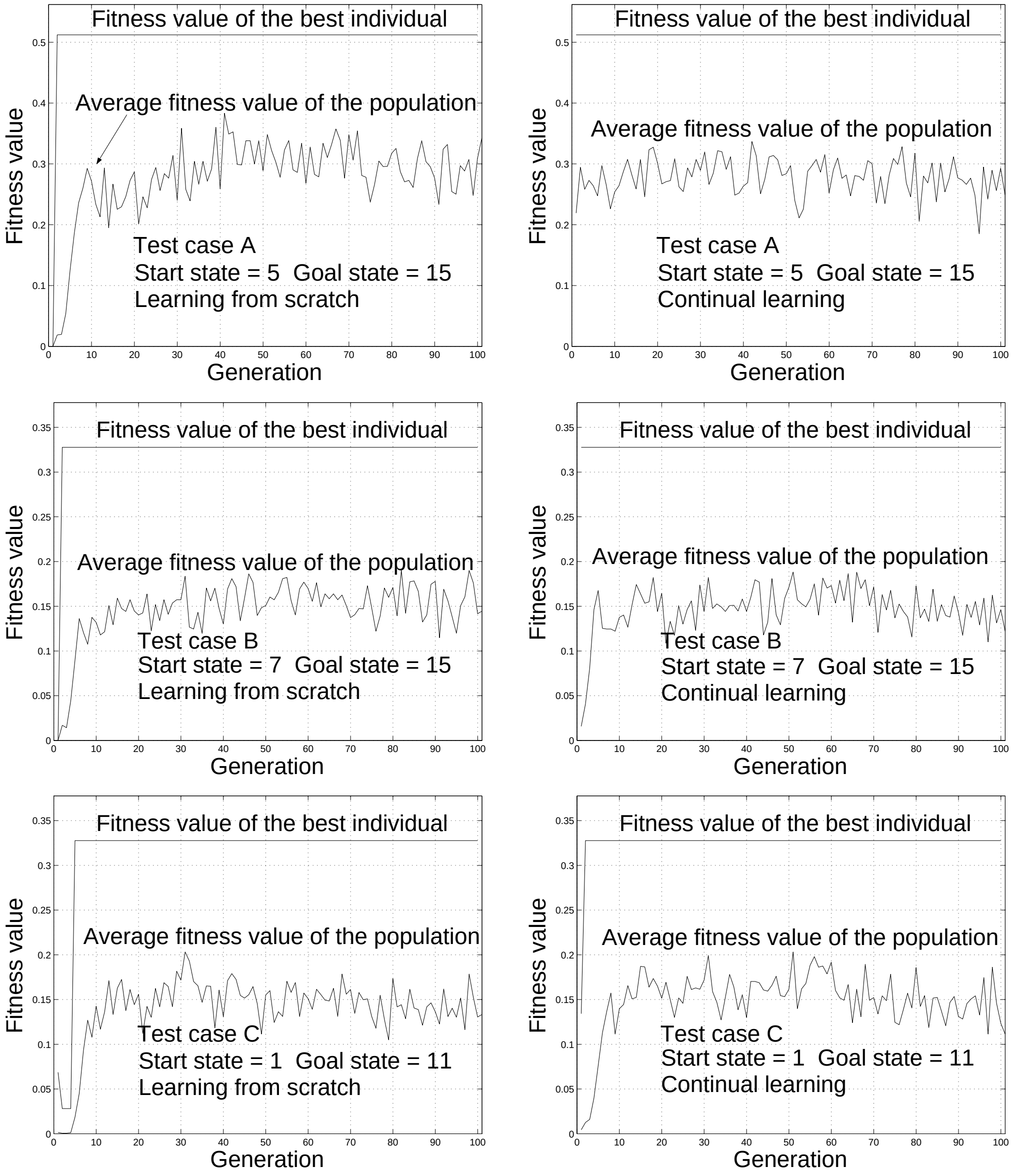


Fig. 4.7: Result obtained for a hybrid of Elman neural networks and genetic algorithm. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

5. HYBRID OF LEARNING AND EVOLUTIONARY ALGORITHMS

5.1 Introduction: Lamarckism and Darwinism

An ecosystem is populated by living organisms that have their own autonomy. The process of adaptation in these systems is made up of two phases. The first phase is learning that occurs at an individual level and the second phase is evolution occurring over successive generations of the population. An individual in a population of organisms performs a sequence of actions that maximize the reward it receives from the environment. The reward measures the degree of satisfaction of the individual. In its life time, the individual learns and adapts to its environment through interaction. The process of learning and adaptation enables the individual to select those actions which result in a higher satisfaction from those actions that cause danger or pain. It is clear that an individual is not born blank. That means it does not learn and adapt to its environment from scratch. The basic structures of the brain, which determines the behavior of the individual, as well as the entire body, is developed according to the genetic information inherited from its ancestors. The inherited genes in offspring are not exact copies of the genes in the parents because of the genetic mutation and recombination.

In evolutionary theory, there are two major ideas that give different explanations for the motive force of natural evolution and the phenomenon of genetic inheritance. These ideas are Lamarckism and Darwinism. The Lamarckian theory suggests that the motive of evolution is the effect of "inheritance of acquired characters." Individuals may undergo some adaptive changes through interaction with the environment or learning. These changes will be somehow be put in their genes and direct evolution. On the other hand, the central theory of the Darwinism is "non-random natural selection following on random mutation". Mutation itself has no direction, but some individuals with advantageous mutations will have more chance of survival through natural selection. The Darwinian theory claims that evolution is nothing but these commulative processes of natural selection. In summary, while the Lamarckian idea assumes the direct connection between learning and adaptation at the individual level and at the population level, the Darwinian idea clearly divides them from each other. It is known that the mainstream of today's evolutionary theory is Darwinism [11].

The main idea of this chapter is not to compare and contrast the Lamarckian and Darwinian strategies, but to investigate the learning and adaptation capability of agents under both strategies.

5.2 Experiments and Results

A population of reinforcement learning agents using Q-learning and whose performance is improved by a genetic algorithm are used to form the hybrid algorithm. In the experiments, we investigate agents that use the Lamarckian strategy and agents that use the Darwinian strategy. For both agents, the algorithm starts with genetic algorithm, which initializes the Q-tables of the agents. The agents learn through interaction in their lifetime and change the content of the Q-table as they learn about their environment. At the end of the life of an agent that uses the Lamarckian strategy, the collected knowledge which is stored in the Q-table will be written back to the chromosome which encodes it. In other words, the current generation will inherit to the next generation what it has learned about its environment. This is the same as inheritance of acquired characteristics. For agents using Darwinian strategy, the contents of the Q-table will not be written back to the chromosome at the end of the life of the agent. It means that the next generation will receive initial values of the Q-table that enables the agents to learn a given optimal policy as fast as possible. One can see clearly that the Q-table which is modified by an agent in its life time is not transferred to the next generation.

Figure 5.1 shows the Q-table and the chromosome that encodes it and which is used in this experiment.

	States				
Actions	0	1	$\cdots$	14	15
`left`	$Q_{0,0}$	$Q_{0,1}$	$\cdots$	$Q_{0,14}$	$Q_{0,15}$
`right`	$Q_{1,0}$	$Q_{1,1}$	$\cdots$	$Q_{1,14}$	$Q_{1,15}$
`forward`	$Q_{2,0}$	$Q_{2,1}$	$\cdots$	$Q_{2,14}$	$Q_{2,15}$

$Q_{0,0}$	$\cdots$	$Q_{0,15}$	$Q_{1,0}$	$\cdots$	$Q_{1,15}$	$Q_{2,0}$	$\cdots$	$Q_{2,15}$

Fig. 5.1: The Q-table and the chromosome that encodes it.

The reward function given by equation (1.1) is used for the reinforcement learning agents, and the fitness function given by equation (1.2) is used for the genetic algorithm. The parameters for the genetic algorithm and the reinforcement learning are shown in table 5.1.

Number of individuals in the population	50
Crossover probability	0.2
Mutation probability per bit	0.05
Selection method	Truncation selection
Learning rate of reinforcement learning	0.3
Discount rate of reinforcement learning	0.3
Number of bits coding a Q-value	8
Number of generations	100

Tab. 5.1: The parameters of genetic algorithm and reinforcement learning.

The experiment is run for all test cases and the results shown in figures 5.2 and 5.3 are obtained for learning and adaptation at the population level. As can bee seen from the figures, the learning time in continual learning is shorter than the learning time in learning from scratch for all test cases and for both strategies. In test case A, both populations of agents do not require to learn the new policy at population level. Moreover, there is an improvement in learning times in continual learning for both types of population of agents for test case B and C.

One of the advantages of hybridizing learning and evolutionary algorithms is that it enables one to generate effective initial values for the action values automatically. The determination of the initial values for the action values is one of the major problems in the reinforcement learning. One way to determine the initial values is to bias the agent with a goal directed built-in knowledge [6]. But this requires the knowledge of states that are perceived and the optimal actions at those perceived states. For a real environment, it is difficult to determine the optimal action for a given perceived state.

The other advantage of hybridizing learning and evolutionary algorithms is that it is also possible to determine the learning rate and discounting factor automatically. This will help agents to adapt to a new situation with minimum learning costs.

It is our believe that one can improve the learning and adaptation capability of agents by using both Lamarckian and Darwinian strategies. For agents which have explored the environment enough or for agents which have lived and operated in a given environment for a long time, it is advisable to use the Lamarckian strategy. For agents which have not explored the environment enough or for agents which are in a fast changing environment, it is better if one uses Darwinian strategy for improving the learning and adaptation capability of agents.

In comparison with learning and adaptation at population level, learning and adaptation at individual level is not computationally expensive, but its learning and adaptation capability depends on the initial knowledge of the individual about the situation that is going to be learned. It has been shown experimentally in chapter 4 that even though the individuals in the population have no learning and adaption capabilities, there is learning and adaption at population level. Note that the neural networks are used only to represent a policy or a controller for the point robot. The synaptic weights of the networks is directly determined by the genetic algorithm. That means the individuals (the neural networks) have no a capability of learning through interaction. It is natural, therefore, to think of individuals having learning and adaptation capabilities and which form a population. This will bring us to the hybrid of learning and evolutionary algorithms. The computational complexity of the hybrid of learning and evolutionary algorithms is much higher than both of learning and adaptation at individual and population levels. At the expense of this computational complexity, however, it is possible to learn and adapt to a more complex situation in the environment using an appropriate hybrid of learning and adaptation algorithms.

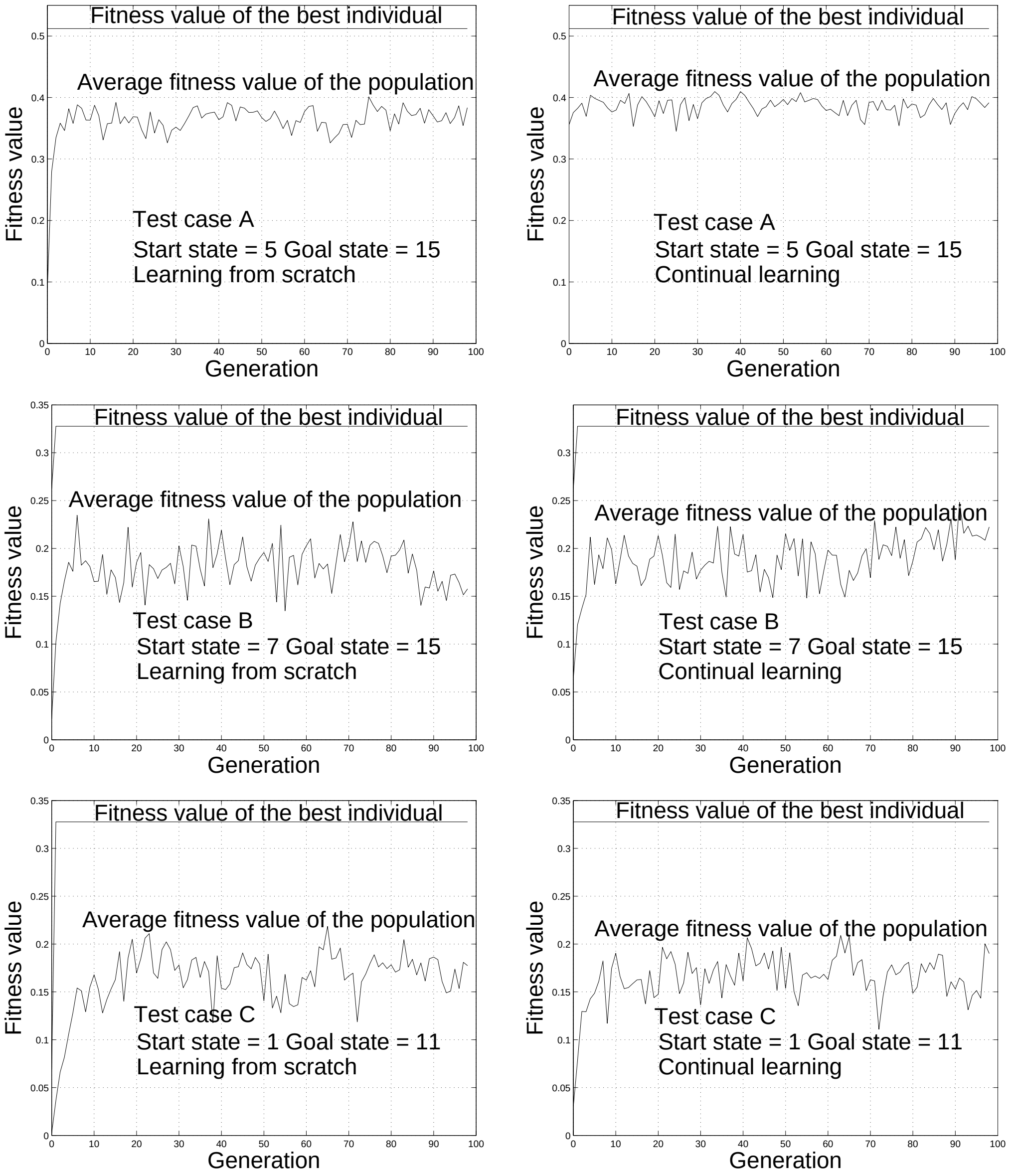


Fig. 5.2: Result obtained for agents using the Lamarckian strategy. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

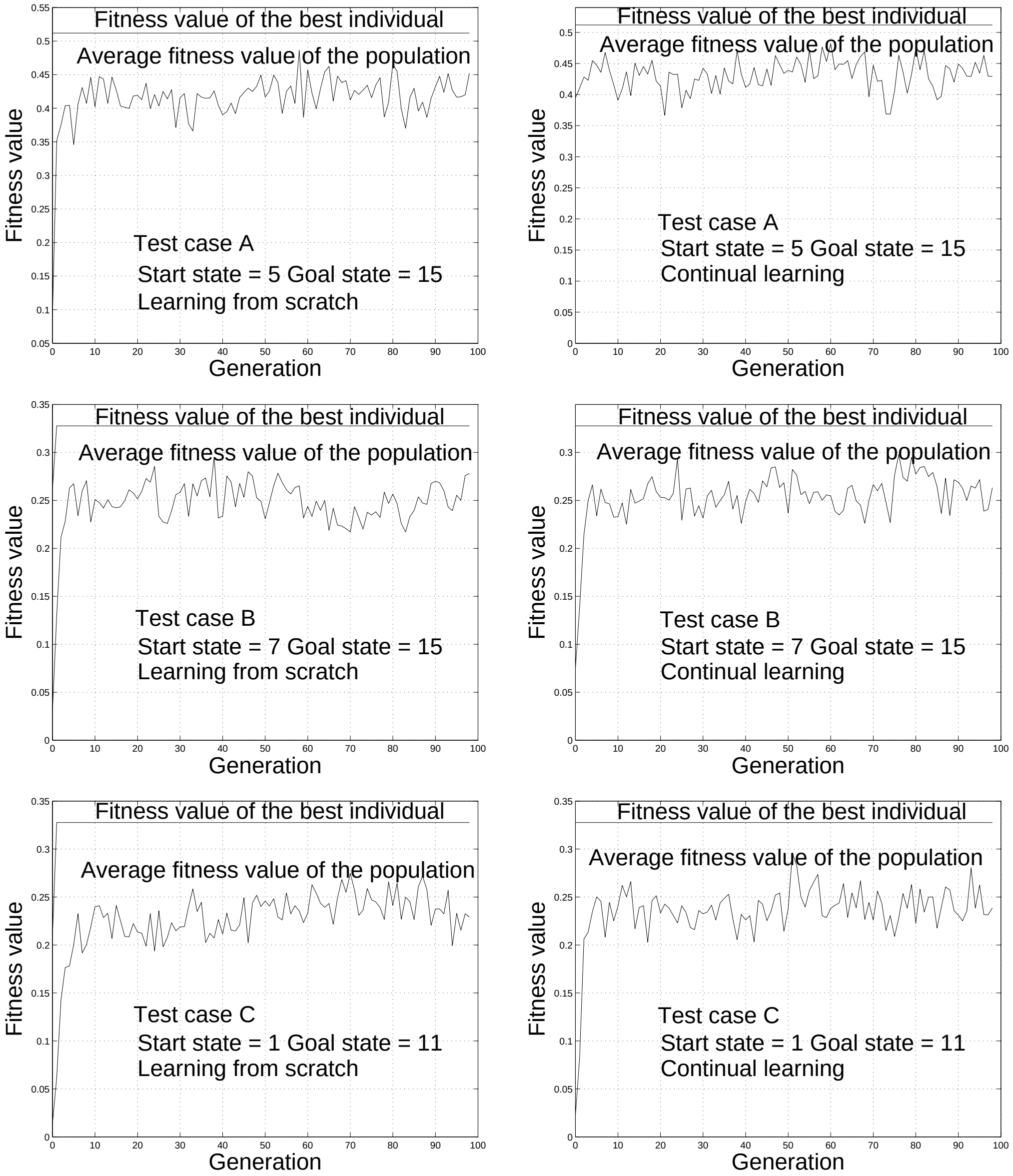


Fig. 5.3: Result obtained for agents using the Darwinian strategy. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

6. CONCLUSION AND OUTLOOK

For agents having knowledge about the dynamics of the environment, one can apply different techniques such as exhaustive search, dynamic programming, Monte Carlo methods and genetic algorithms to solve the Bellman optimality equation of an MDP problem. Methods using dynamic programming (DP) can solve the problem very efficiently. Dynamic programming methods are guaranteed to find an optimal policy in polynomial time even though the total number of policies is $n_s^{n_a}$ where n_s is the number of states and n_a is the number of actions [14]. Moreover, DP can be used to get both the optimal policy and action values at the same time. Genetic algorithms can also be used to solve a given MDP problem. They do that by directly searching for the optimal policy in space of policies. They are much slower as compared to the computational efficiency of dynamic programming. Monte Carlo methods are much slower as compared to genetic algorithms but can be used to find the optimal policy. Unlike dynamic programming, one can use genetic algorithms and Monte Carlo methods to find the optimal policy without a need for the knowledge of the dynamics of the environment.

Q-learning has the properties of both dynamic programming and Monte Carlo methods. It is suitable for learning from interaction at an individual level. The convergence rate of a Q-learning method to an optimal solution depends on the initial values of the Q-table, on the learning rate, on the discounting factor and on the way the exploration and exploitation strategies are balanced.

From the results of experiments that are presented in this report, one can conclude that the learning time required in continual learning is shorter than that required in learning from scratch at both individual and population levels and under various learning conditions. They also show that the learning time in continual learning depends on the number of states of a policy, which is going to be learned, that are contained in the previously learned optimal policy. The more states the two policies have in common, the shorter will be the time required in continual learning. For test case A, where the states of a policy are completely contained in the previously learned optimal policy, the agent does not need to learn the optimal policy in continual learning. It is also interesting to see that, even though the two policies have no common states (test case C), the time required in continual learning is shorter than the time required in learning from scratch. The different test cases of the experiments show that an agent can use a related knowledge to a new situation, which is going to be learned, to adapt itself faster and make the learning time shorter. Furthermore, the adaptation time required by an agent to adapt to a new situation depends on the amount of knowledge it has about the new situation.

Hybridization of various learning algorithms with evolutionary algorithms will give agents two levels of adaptation capabilities. The first is an individual level adaptation capability, and the second is a population level adaptation capability. The individual level adaptation capability depends on the learning algorithm used. At population level, the adaptation capability is contained in the variation between individuals.

With adequate hybridization of learning algorithms and evolutionary methods (like genetic algorithms, genetic programming and genetic strategies) it is our believe that, one can design better agents with better learning and adaptation capability for either lower or higher cognitive levels.

BIBLIOGRAPHY

[1] Barto A. G., Bradtke S. J. and Singh S. P. Learning to Act using Real-Time Dynamic Programming. *Department of Computer Science, University of Massachusetts, Amherst MA 01003*, 1993.

[2] Banzhaf W., Nordin P., Keller R.E. and Francone F.D. Genetic Programming. An Introduction on the Automatic Evolution of Computer Programs and its Applications. *Morgan Kaufmann Publishers*, 1998.

[3] Bentley P. J. and Corne D. W. Creative Evolutionary Systems. *Morgan Kaufmann Publishers, Academic Press*, 1999.

[4] Gerstner W. and Kistler W. Mathematical Formulation of Hebbian Learning. *Laboratory of Computational Neuroscience, EPFL, Lausanne EPFL and Department of Neuroscience, Ersamus University, Rotterdam*, 1994.

[5] Hailu G. Towards Real Learning Robots. *Institute of Computer Science and Applied Mathematics, Christian Albrechts University of Kiel*, Ph.D. Thesis, 1999.

[6] Hailu G. and Sommer G. On amount and quality of bias in reinforcement learning. *In Proceedings of International Conference on Systems, Man and Cybernetics, San Diego, California*, 1998.

[7] Holland J.H. Adaptation in Natural and Artificial Systems. *University of Michigan Press, Ann Arbor, Michigan*, 1975.

[8] Jacob C. Illustrating Evolutionary Computation with Mathematica. *Morgan Kaufmann Publishers, Academic Press*, 1993.

[9] Littman M. L. and Moore A. W. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research 4. 237-285*, 1996.

[10] Nolfi S. and Floreano D. Evolutionary Robotics: The Biology, Intelligence and Technology of Self-Organizing Machines. *A Brandford Book, The MIT Press*, 2000.

[11] Sasaki T. and Tokoro M. Adaptation toward changing environments: Why Darwinian in nature? *In Proceedings of 4th European Conference on Artificial Life (ECAL-97)*, 1997.

[12] Schaffer J.D., Whitley L.D. and Eshelman L.J. Combination of genetic algorithms and neural networks: A survey of the state of the art. *In Proceedings of an International Workshop on the Combination of Genetic Algorithms and Neural Networks, L.D Whitley and J.D. Schaffer (Editors), IEE Press, Los Alamitos*, 1992.

[13] Sobol I.M. Die Monte-Carlo-Methode. *VEB Deutscher Verlag der Wissenschaften, Berlin*, 1968.

[14] Sutton R. S. and Barto A.G. Reinforcement Learning. *A Brandford Book, The MIT Press*, 1998.

[15] Whitley D. A Genetic Algorithm Tutorial. *Technical Report CO 8052, Computer Science Department, Colorado State University Fort Collins*, 1993.