Towards a Unified Approach to Learning and Adaptation

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.–Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Yohannes Kassahun

Kiel
Februar 2006

1. Gutachter	Prof. Dr. Gerald Sommer (Kiel)
2. Gutachter	Prof. Dr. Anand Srivastav (Kiel)

Datum der mündlichen Prüfung: 31.01.2006

ABSTRACT

The aim of this thesis is to develop a system that enables autonomous and situated agents to learn and adapt to the environment in which they live and operate. In doing so, the system exploits both adaptation through learning and evolution. A unified approach to learning and adaptation, which combines the principles of neural networks, reinforcement learning and evolutionary methods, is used as a basis for the development of the system. In this regard, a novel method, called Evolutionary Acquisition of Neural Topologies (EANT), of evolving the structures and weights of neural networks is developed. The method introduces an efficient and compact genetic encoding of a neural network onto a linear genome that encodes the topology of the neural network implicitly in the ordering of the elements of the linear genome. Moreover, it enables one to evaluate the neural network without decoding it. The presented genetic encoding is complete in that it can represent any type of neural network. In addition to this, it is closed under both structural mutation and a specially designed crossover operator which exploits the fact that structures originating from some initial structure have some common parts. For evolving the structure and weights of neural networks, the method uses a biologically inspired meta-level evolutionary process where new structures are explored at larger timescale and existing structures are exploited at smaller timescale. The evolutionary process starts with networks of minimal structures whose initial complexity is specified by the domain expert. The introduction of neural structures by structural mutation results in a gradual increase in the complexity of the neural networks along the evolution. The evolutionary process stops searching for the solution when a solution with the necessary minimum complexity is found. This enables EANT to find optimal neural structures for solving a given learning task. The efficiency of EANT is tested on couple of learning tasks and its performance is found to be very good in comparison to other systems tested on the same tasks.

ACKNOWLEDGMENTS

This thesis would not have been brought about without the support of so many individuals. These individuals have contributed a lot to the great years I have had at Christian-Albrechts-University (CAU) of Kiel. It is my great pleasure to thank them here.

My unlimited thanks go to my supervisor Prof. Dr. Gerald Sommer who has been a major source of inspiration and motivation throughout the work. I learned a lot from the discussions I had with him. I appreciate his unreserved support, encouragements and helpful suggestions.

Deepest thanks to Dr. Getachew Hailu for introducing me to Prof. Dr. Gerald Sommer. I had important discussions with him about machine learning at the beginning of my work.

I thank all members of the Cognitive Systems Group, Kiel for their support and help. My special thanks go to Oliver Granert for being a wonderful friend and source of great discussion. I thank Di Zang, Stephan Al-Zubi, Sven Buchholz and Nils Siebel for proofreading different chapters of this thesis. I further thank my colleagues Marco Chavarria, Christian Gebken, Florian Hoppe, Christian Perwass, Herward Prehn and Antti Tolvanen. They have been always helpful all the times.

Wholehearted and deepest thanks to our technical staff Henrik Schmidt and Gerd Diesner for their unreserved support during the experiments conducted in our laboratory. I would like to thank also our secretary Francoise Maillard for her friendly help in administrative matters.

I acknowledge the German Academic Exchange Service (DAAD) under grant code R-413-A-01-46029 for sponsoring the research presented in this thesis. “Danke Deutschland!”

Last, but not least, I thank all members of my family, to whom I dedicate this thesis.

CONTENTS

Chapter 1

INTRODUCTION

It is the dream of human kinds to create machines that are as intelligent as we are. We humans want to have machines that can recognize the people around us, talk to each other or to us in the most natural way, read a newspaper, drive a car, make and serve a tea, help the disabled and so on.

Since learning is considered as the main component of an intelligent system, scientists have tried, with varying degrees of success, to give machines the ability to learn. Machine learning is the term used for this field of study and is coined by Samuel, who made a computer for the first time to perform learning task [84]. There are three major approaches that have been used in machine learning:

- **Supervised learning**: In supervised learning, the trainer supplies the learning system example training sequences containing inputs accompanied with correct outputs. The system is expected to generalize so as to correctly classify or recognize unseen data that is not included in the training sequence.

- **Reinforcement learning**: In reinforcement learning [7, 97], the learning system is not told the correct output for a given input but is fed back with a reward signal that measures the quality of the output signal.

- **Unsupervised learning**: In this type of learning, the system is supplied only with input sequence and is not told what the correct output sequence is. Rather, the system has to look for input patterns in the input data. A very good example of unsupervised learning is the operation of a Kohonen neural network [65].

For most of the intelligent systems that are developed so far, it is extremely hard to recognize objects, to talk in the most natural way, or to make and serve a tea. This is because the design of these systems is not

based on behavioral diversity and survival of the fittest. Evolutionary theory teaches us that the brain of human beings, or in general animals has evolved to ensure the survival of the species and to control its behavior. The design of intelligent agents requires a system that enables the evolution of "artificial brain" starting from simple structures and increasing its complexity over generations based on evolutionary theory which results in the behavioral diversity. The complexity of the brain has to match the complexity of the task environment. This thesis tries to give a contribution in this direction.

1.1 Motivation

The term intelligence is too complex to give a full definition describing it. But in many fields of studies of intelligence, the key concept is the generation of behavioral diversity while complying with the rules [78]. One of the important properties of an intelligent agent is that it should be autonomous in the sense that the degree of external control is minimal. Autonomy implies the agent's ability to *learn from experience* through interaction with the environment. An agent that has the ability to learn through interaction with the environment is autonomous since it can acquire its own knowledge over time. Learning from experience requires automatically the agent's ability to *learn continuously*. Moreover, intelligent agents exhibit emergent behaviors. Such behaviors are not programmed into the agents by the designer but rather are the result of the interaction of the agents with their environment.

An intelligent agent should also be a situated agent in that it acquires information about its environment only through its sensors while interacting with the environment. Situatedness like autonomy requires us to have agents with learning components. The capacity to learn increases the agent's autonomy. If intelligent agents are situated, they can learn and adapt to their environment continually.

In addition, an intelligent system should learn to perform a task *without being told precisely how to do it*, but it should be told *precisely how to learn*. This thesis is concerned with the design of a learning system for intelligent agents which enables them to learn to accomplish a task without being told precisely how to do it.

The main motivation behind the design of the learning system comes from the hypotheses proposed by Charles Darwin [25] on which the theory of evolution is based upon. The first hypothesis says:

All organisms adapt to their environments.

If a species is to sustain over an extended periods of time in a continuously changing, unpredictable environment, then it must be adaptive. The term

adaptation has various interpretations and meanings but we considered the following types of adaptations in the agents to be designed.

- **Evolutionary adaptation**: We see different kinds of adaptations in nature. For example, the light-skinned form of the peppered moth (*Biston betularia*), called *typica* was the predominant form in England prior to the beginning of the industrial revolution. It would be difficult to pick it out against the light-colored bark of many trees common in England. In regions that became industrialized, industrial smoke darkend the tree trunks. Gradually the peppered moth population in industrial areas became predominantly composed of a dark variety (*Carbonaria*), which would be almost invisible against a dark background. This is a good example of adaptation of an organism to changes in the environment.

- **Adaptation by learning**: Animals adapt to their environment through learning in their lifetimes. We can take human beings as an example. Human beings have acquired through evolution some instincts and capabilities of learning. In other words, they are not born blank and therefore they do not learn from scratch. They use these instincts and capabilities as a starting knowledge to learn about their environment.

It is important to note that evolutionary and learning based adaptations work on different timescales. Typically, evolutionary adaptation takes many generations to happen while adaptation by learning happens in the lifetime of an organism.

Based on the two types of adaptations, a meta-level evolutionary process is developed that searches for new species (structures) at larger timescale and adapts existing structures at smaller timescale. That means, agents learn and adapt to their environment ***through evolution and by learning***. The second hypothesis of Darwin proposes:

All organisms descend from common ancestors.

A strong evidence for common descent may be found in traits shared between all living organisms. It is known that every living thing makes use of nucleic acids as its genetic material, and uses the same twenty amino acids as the building blocks for proteins [6]. All organisms use the same genetic code with some extremely rare and minor deviations to translate nucleic acid sequences into proteins. In addition to this, observable morphological

similarities give us a good evidence for common descent. For example, all birds even those which do not fly have wings.

There are heritable changes in phenotype (and genotype) of a species, resulting in a transformation of the original species into a new species similar to, but distinct from, its parent species.

According to this hypothesis, the developed learning system starts with agents with minimum structures (species) whose initial complexity is specified by the domain expert. New structures are introduced by mutation along the evolution. That means, all other structures originate from the initial structure. The introduction of new structures results in a gradual increase in the number of species and their complexity. Through ***complexification***, the agents will have the ability of continual learning and are able to extend their competence, or acquire new competences. The third and the most well known Darwin's hypothesis states:

All organisms are not equally well adapted to their environment, some will survive and reproduce better than others.

This hypothesis is known as the natural selection or the survival of the fittest. Natural selection can be subdivided into two categories: ecological selection and sexual selection. Ecological selection occurs when organisms which survive and reproduce increase the frequency of their genes in the gene pool over those which do not survive. Sexual selection occurs when organisms which are more attractive to the opposite sex because of their features reproduce more and thus increase the frequency of those features in the gene pool.

In developing our system, we assumed only ecological natural selection. In the system, one can identify two types of competitions. The first is the competition within a species (individuals having the same structure) and the other is the competition between species. Competition within a species occurs at smaller timescale while competition among species occurs at larger timescale. Species which are strong enough survive and continue to live while others get extinct. The fourth hypothesis asserts the fact:

All organisms are variable in their traits.

The transfer of genetic material from parent to child must be accomplished so that the parental traits pass to the child with high probability. There must also be a chance of passing useful new or different traits to the child. Without the possibility of new or different traits, there would be no

variability for natural selection to act upon. The variability among individuals within a species or among species is an important factor for the adaptation capabilities of organisms.

In the system presented, the variablity is achieved through structural mutation and crossover operators that occur at the larger timescale, and parametric mutation and recombination operators that occur at smaller timescale.

In summary, the thesis is concerned with the design of a learning system for autonomous intelligent agents that are situated. The agents learn and adapt to their environment *through evolution and by learning*. The learning system is realized using a *nature inspired* meta-level evolutionary process where new structures are explored at larger timescale and existing structures are exploited at smaller timescale. With respect to the goal of self-organizing learning machines which start from minimal specification and rise to great sophistication, the system starts with agents of minimal structures, and increases their *complexity* along the evolution path.

1.2 Closely Related Works

Learning and adaptation through evolution of the structure and weights of the neural networks is found to be successful in solving reinforcement learning tasks. Two major types of encoding the neural structures are mentioned in the literature: the direct and indirect encoding. In the direct encoding, every connection and node of a neural structure is explictly encoded [2, 29, 34, 51, 73, 85, 95, 105] in the genome while in the indirect encoding the genome contains rules that are used in constructing the neural structure [39, 64].

Our work is closely related to the works of Angeline et al. [2] and to the works of Stanley and Miikkulainen [94, 95]. It is related to the works of Angeline et al. in that the method uses structural mutation as a main search operator for structural discoveries, and parametric mutation that is based on evolution strategies or evolutionary programming with adaptive step sizes for optimization of the weights of the neural networks. Complexification of structures along the evolution path starting from a minimum structure makes it related to the works of Stanley and Miikkulainen. In this section, the two most closely related works are mentioned. A detailed review of related works is given in Section 5.1.

The work presented in this thesis introduces a novel genetic encoding where the construction rules of the neural structure are implicitly encoded in the ordering of the nodes of the genome. Unlike other encoding systems in the area of evolution of neural networks, the genome enables one to evaluate the neural structure without reconstructing it from the linear genome. For

evolution of neural structures, the presented method uses a meta-level evolutionary process where new structures are explored at larger timescale and existing structures are exploited at smaller timescale.

1.3 Outline of the Thesis

This section outlines the structure of the thesis and gives a short summary for each of the chapters.

Chapter two gives a brief overview of the basic components used in the design of the learning system. Artificial neural networks, reinforcement learning, evolutionary methods and behavior based systems are discussed. The two building blocks of reinforcement learning, dynamic programming and Monte Carlo methods, are also introduced. Very brief introductions to evolutionary methods namely genetic algorithms, genetic programming, evolution strategy and evolutionary programming is also given.

The third chapter introduces an application example of an evolutionary method in model based object recognition systems. The system recognizes 2D planar objects in 3D and determines their pose simultaneously. The recognition is independent of translation, rotation and scaling. Models of new objects which are not in the system are automatically acquired and stored in a database of models to be recognized.

Chapter four proposes methods of improving the learning and adaptation capability of autonomous agents by taking as an example a navigation problem in an artificial robot world. Learning and adaptation at both individual and population levels are considered. Moreover, the advantages of continual learning over learning from scratch is discussed for both learning at individual and population levels and under different learning conditions. A recommendation is given at the end of the chapter that is used as a design guideline for the learning system that is introduced in chapter five.

The main contributions of this thesis is given in chapter five. The chapter starts with a discussion of evolutionary reinforcement learning that combines concepts from artifical neural networks, reinforcement learning and evolutionary methods. Then it introduces a novel method of evolving artifical neural networks, called evolutionary acquisition of neural topologies (EANT). It combines meaningfully the concepts of neural networks, reinforcement learning and evolutionary methods. The method introduces a new genetic encoding of neural network that enables one to evaluate it without decoding it, and a meta-level evolutionary method that exploits neural structures at smaller timescale and explore new ones at larger timescale. The chapter gives performance evaluation of the method by taking benchmark problems. It also

gives performance comparison of the method with related algorithms tested on the same benchmark problems.

The application of EANT to the problem of robot navigation with obstacle avoidance is considered in chapter six. The main purpose of the chapter is to demonstrate the automatic design of neural controllers for robots using EANT. The chapter begins with the development of a controller for sonar based navigation. Then a vision module which is equivalent to the sonar sensors in detecting obstacles is developed. This enables one to use the controller developed for sonar based navigation for vision based navigation. The results obtained by using EANT are compared to related algorithms tested on the same application.

Finally, a conclusion and outlook is given in chapter seven. The chapter gives analysis of the overall results obtained and discusses the possible future directions of research.

Chapter 2

BACKGROUND MATERIAL

This chapter gives very brief introductions to the basic components used in the development of a learning system for intelligent agents. In particular, artificial neural networks, reinforcement learning, evolutionary methods and behavior based systems are discussed. The two basic components of reinforcement learning namely dynamic programming and Monte Carlo methods are also presented. Moreover, a discussion on major types of evolutionary methods such as genetic algorithms, genetic programming, evolution strategy and evolutionary programming is given.

2.1 Artificial Neural Networks

An artificial neural network is an interconnected assembly of simple computational units or nodes whose functionality is loosely based on the animal neuron. The processing or computational ability of the neural network is stored in the inter-unit connection strengths, or *weights*, obtained by the process of adaptation or learning. A neural network has input and output units which receive and give out signals to the environment, respectively. Computational units which are not directly connected to the environment are usually called hidden units. A neural network is a parallel computational system since signals travel independently on weighted channels and the units can update their state in parallel. Figure 2.1 shows a simple neuron model.

In a simple neuron model, the output y of a neuron node is calculated by taking the sum of all signals x_i weighted by a connection strength w_i and transforming the sum by the activation function g.

$$y = g\left(\sum_{i}^{n} w_i x_i\right) \tag{2.1}$$

McCulloch and Pitts [72] used only binary weight values and binary inputs. Rosenblatt proposed the perceptron [81], a more general computation

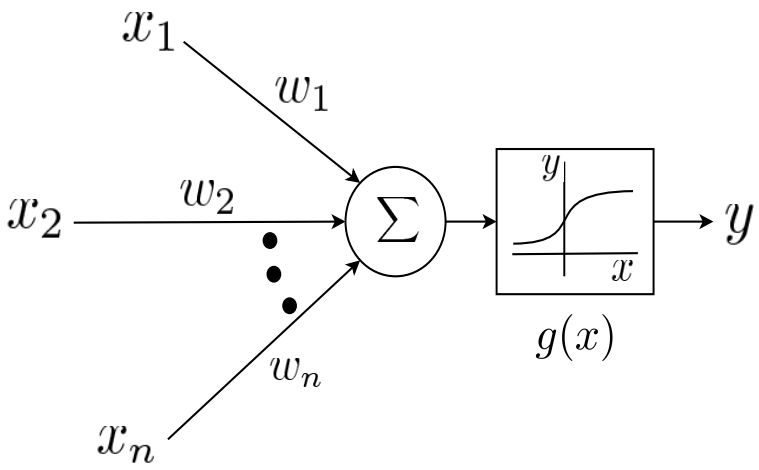


Fig. 2.1: A neuron model. In computing the output of a neuron, each input signal is multiplied by the corresponding connection weight and the results are added together. The neuron output is found after transforming it with a function $g(x)$ which is usually called activation function.

model than McCulloch and Pitts units, where the inputs and the weights can assume real values. The most commonly used activation functions include the step function, the linear function and the sigmoid function. The step function used by McCulloch and Pitts, and Rosenblatt returns only one bit of information, whether the unit is on or off. It is given by

$$g(x) = \begin{cases} 0 & \text{if} \quad x > \theta \\ 1 & \text{if} \quad \text{otherwise} \end{cases}, \tag{2.2}$$

where θ is the threshold value. The unit is on when the sum of all signals x_i weighted by a connection strength w_i is greater than θ. The linear function can transmit more information as compared to the step function with its graded output,

$$g(x) = ax, \tag{2.3}$$

where a is a constant that controls the inclination. Out of the three activation functions, the sigmoid function is the most commonly used type of activation function. The sigmoid function is given by

$$g(x) = \frac{1}{1 + \exp(-ax)}, \tag{2.4}$$

where the constant a controls the slope of the response. As $a \to \infty$, the sigmoid function approximates the step function. The translated version of the sigmoid function $\tanh(ax)$ which is bounded between -1 and 1 is also commonly used as an activation function.

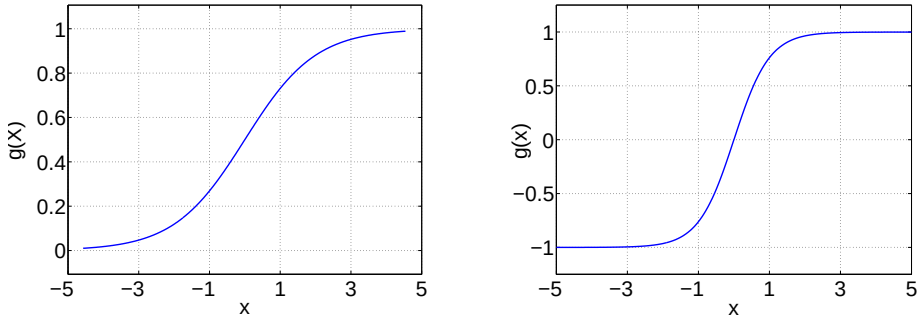


Fig. 2.2: The sigmoid and $\tanh(x)$ activation functions.

The behavior of a neural network is not only captured in the connection weights but also in how the nodes are interconnected with each other. In other words, the architecture of a neural network plays an important role in the determination of the behavior of the neural network. The architecture of a neural network is defined by the number of neurons and their pattern of connectivity. Most of the time units are organized in layers: input layer, hidden layer(s) and output layer. There are two large categories of architectures: feed-forward and recurrent. Signals travel from the input unit forward to the output units in forward architectures. The state of the hidden units are updated before the output units. In recurrent architectures, there may be recurrent connection from a neuron in the upper layer to a neuron in the lower layer, or recurrent connection from the same neuron. Recurrent architectures can have quite complex time dependent dynamics. A general form of learning in systems with neural networks comprises both the acquisition of the network architecture and the modification of the synaptic weights of the acquired network architecture.

For a given architecture, learning is achieved by repeatedly updating the weight values. In supervised learning, synaptic strengths are modified using the discrepancy between the desired output and the output given by the network for a given input pattern. In unsupervised learning, the network updates the weights on the basis of the input patterns only. The way the network self-organizes its behavior is determined by the learning rule and architecture chosen. Unsupervised learning is usually mainly applied

in feature extraction, categorization and data compression. In evolutionary reinforcement learning, the synaptic strengths are modified based on the performance of the network in solving a given learning task. The performance is measured using a fitness value set by the domain expert. In most cases, initial weights are set to zero or to small random values centered around zero. Learning takes place by repeatedly presenting pairs of input-output patterns for supervised learning and only input patterns for unsupervised learning. In evolutionary reinforcement learning, learning takes place after the individual has lived and operated in the environment. The synaptic strength of a unit is modified using

$$w_i(t+1) = w_i(t) + \sigma \Delta w_i(t) \qquad 0 < \sigma \leq 1, \tag{2.5}$$

where $w_i(t)$ and $w_i(t+1)$ are the synaptic strengths of the unit i before and after modification, respectively. In order to prevent wide oscillations of the weights from one change to the next, only a small rate of modification is added to the previous weights. The rate of modification σ is known as the learning rate. Learning algorithms are concerned with the computation of Δw. A detailed introduction to neural networks is given by Bishop [13] and Rojas [80].

Artificial neural networks are useful in the design of control architectures of autonomous agents because they are robust and are excellent generalization models [75]. In order to use a neural network for controlling autonomous agents, it must be connected to the robot. In order to understand the behavior of the neural network, one must know how the network is embedded in the robot, and one must know about the physics of the sensory and motor systems [78].

2.2 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.

2.2.1 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 2.3 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}.

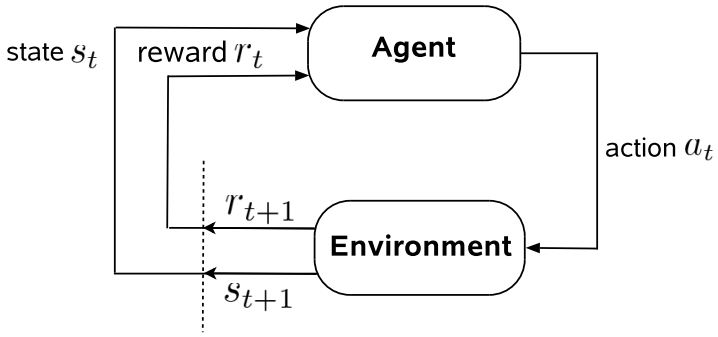


Fig. 2.3: The agent-environment interaction.

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 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.

2.2.2 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

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

return is the sum of the rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T, \tag{2.6}$$

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. As an example one can consider 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:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.7}$$

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.

2.2.3 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:

$$P\left\{s_{t+1} = s', r_{t+1} = r' | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \cdots, r_1, s_0, a_0\right\}, \tag{2.8}$$

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

$$P\left\{s_{t+1} = s', r_{t+1} = r' | s_t, a_t\right\}, \tag{2.9}$$

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 (2.8) is equal to equation (2.9) 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

$$P_{ss'}^{a} = P\left\{s_{t+1} = s' | s_t = s, a_t = a\right\}. \tag{2.10}$$

Equation (2.10) 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

$$R_{ss'}^{a} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\}. \tag{2.11}$$

The quantities given by equation (2.10) and (2.11) completely specify the dynamics of a finite MDP.

2.2.4 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

$$V^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}, \tag{2.12}$$

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.

$$Q^{\pi}(s, a) = E_{\pi}\left\{R_t | s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\}. \tag{2.13}$$

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.

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{R_t|s_t = s\} \\ &= E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s\right\} \\ &= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s'\right\}\right] \\ \Leftrightarrow V^{\pi}(s) &= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right]. \end{aligned} \tag{2.14}$$

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

$$\begin{aligned} Q^{\pi}(s,a) &= E_{\pi}\{R_t|s_t = s, a_t = a\} \\ &= E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s, a_t = a\right\} \\ &= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s''\right\}\right] \\ \Leftrightarrow Q^{\pi}(s,a) &= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right]. \end{aligned} \tag{2.15}$$

Equations (2.14) and (2.15) are the Bellman equations for $V^{\pi}(s)$ and $Q^{\pi}(s,a)$ respectively. Figure 2.4 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 [42].

2.2.5 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, $\pi \geq \pi'$

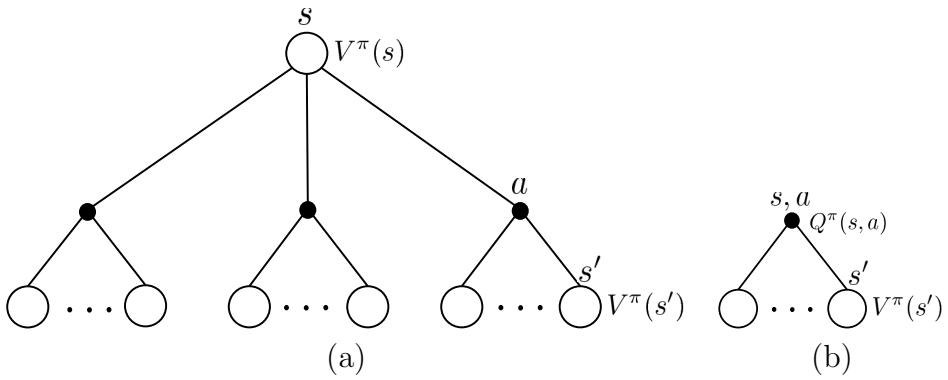


Fig. 2.4: Backup diagrams for (a) V^π and (b) Q^π.

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

$$V^*(s) = \max_\pi V^\pi(s), \tag{2.16}$$

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

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \tag{2.17}$$

for all states and actions. It is possible to write equations (2.16) and (2.17) in recursive form using equations (2.14) and (2.15) as given by

$$V^*(s) = \max_\pi \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s')\right] \tag{2.18}$$

$$Q^*(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s',a')\right]. \tag{2.19}$$

The Bellman optimality equation (2.18) 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 ahead search. If we have the optimal state value function, V^*, then the actions that appear best after a one-step ahead search 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 [70].

2.3 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 [97]. 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.

2.3.1 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 (2.14) 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:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]. \tag{2.20}$$

It can be shown that $V_k \to V^\pi$ as $k \to \infty$. Figure 2.5 gives a complete algorithm for iterative policy evaluation.

Input π, 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. 2.5: Iterative policy evaluation (taken from [97]).

2.3.2 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

$$\pi'(s) = \arg\max_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^\pi(s')]. \tag{2.21}$$

In equation (2.21), $\arg\max_a$ 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.

2.3.3 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:

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

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 [97]. Figure 2.6 shows the algorithm for policy iteration.

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. 2.6: Policy iteration for V^* (taken from [97]).

2.3.4 Value Iteration

The value iteration algorithm follows from the recursive form of the Bellman optimal state value function (2.18). The equation that governs the value iteration is given by

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s')\right]. \tag{2.22}$$

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 2.7.

Initialize V arbitrarily for all the states

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

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

Fig. 2.7: Value iteration (taken from [97]).

2.4 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.

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 the model of the environment at hand, 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 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 serious 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 2.8 shows an algorithm for Monte Carlo method with exploring starts.

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 π
(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. 2.8: A Monte Carlo algorithm with exploring starts (taken from [97]).

2.4.1 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 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

$$Q_n(s,a) = \frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k}. \tag{2.23}$$

It is possible to write equation (2.23) in the form given by equation (2.24),

$$Q_{n+1}(s,a) = \frac{\frac{w_{n+1}}{W_n} R_n + Q_n(s,a)}{\frac{w_{n+1}}{W_n} + 1}, \tag{2.24}$$

where $W_n = \sum_{k=1}^{n} w_k$. Rewriting equation (2.24), we obtain

$$Q_{n+1}(s,a) = Q_n(s,a) + \frac{w_{n+1}}{W_{n+1}} \left[R_{n+1} - Q_n(s,a) \right]. \tag{2.25}$$

Equation (2.25) 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

$$V_{n+1}(s) = V_n(s) + \frac{w_{n+1}}{W_{n+1}} \left[R_{n+1} - V_n(s) \right]. \tag{2.26}$$

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 (2.26) and (2.25), one obtains the recursive forms of the Monte Carlo methods for $V(s)$ and $Q(s,a)$, which are given by

$$V_{n+1}(s) = V_n(s) + \alpha \left[R_{n+1} - V_n(s) \right] \text{ and} \tag{2.27}$$

$$Q_{n+1}(s,a) = Q_n(s,a) + \alpha \left[R_{n+1} - Q_n(s,a) \right]. \tag{2.28}$$

2.5 Genetic Algorithms

Genetic algorithms [48] 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 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 [10].

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.

2.5.1 The Algorithm

A simplest form of GA, the canonical or simple GA, is summarized in Figure 2.9. 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.

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

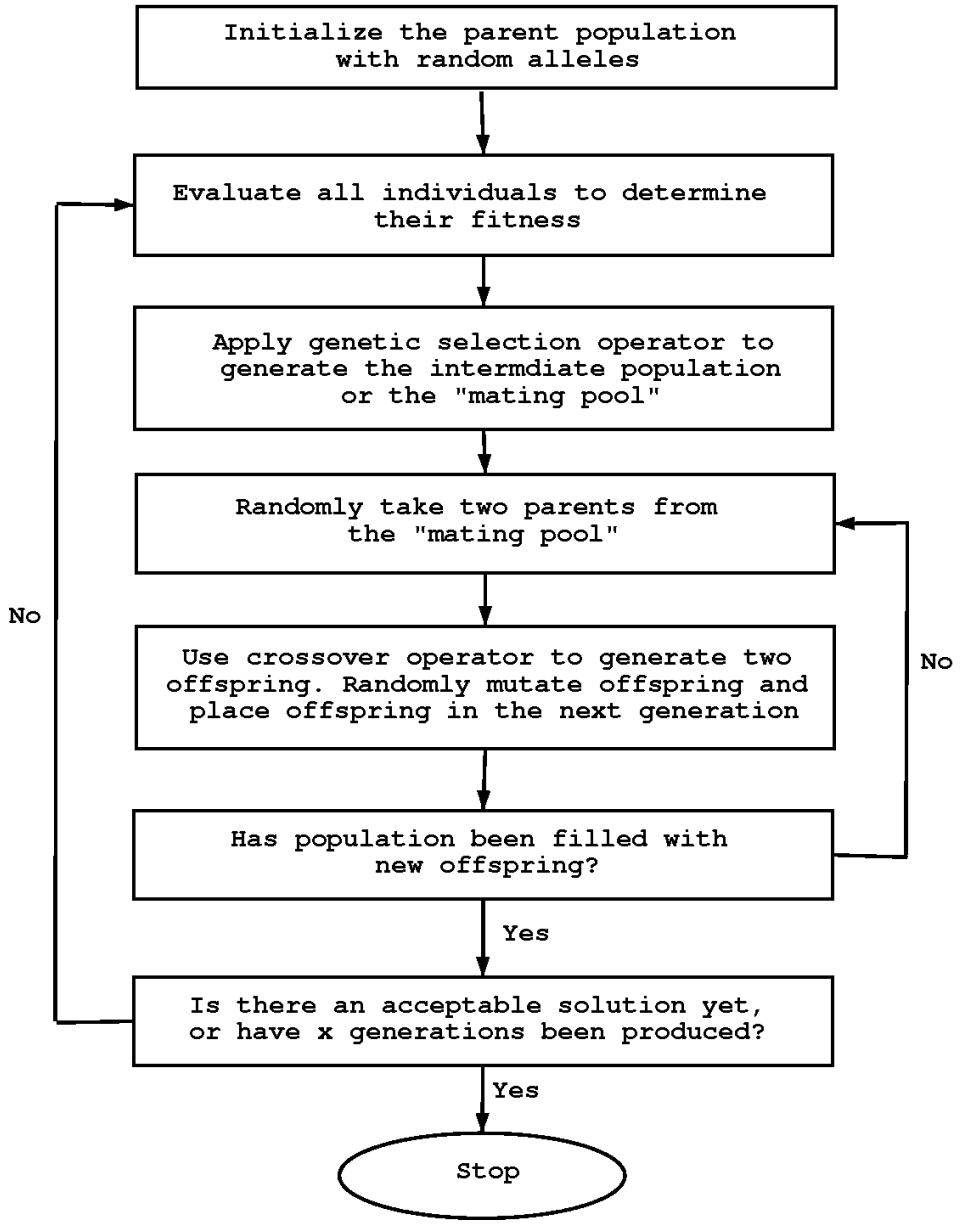


Fig. 2.9: The simple genetic algorithm.

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 op-

erators 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 2.10 shows examples of the 1-point and 2-point crossovers.

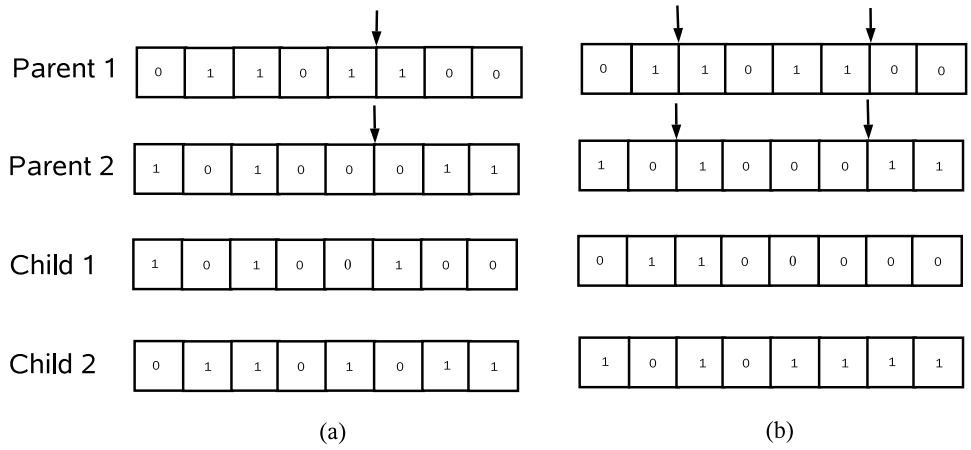


Fig. 2.10: 1-point (a) and 2-point (b) crossover operators. The arrows show crossover points.

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 2.11 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.

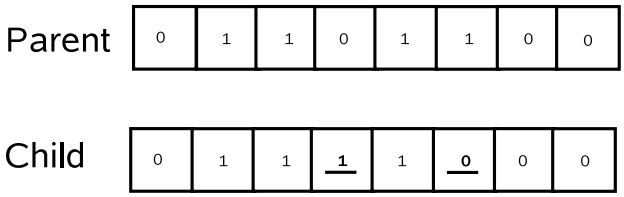


Fig. 2.11: Effect of mutation operator for binary chromosomes. The underlined bits show bits that are flipped.

2.5.3 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

$$p_i = \frac{f_i}{\sum_j f_j} \tag{2.29}$$

for being able to pass on traits. The value f is the fitness value of an individual. Following Holland [48], 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 [6].

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,

$$p_i = \frac{1}{N}\left[p^- + \left(p^+ - p^-\right)\frac{i-1}{N-1}\right], \tag{2.30}$$

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

$$p^- + p^+ = 2 \tag{2.31}$$

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,

$$p_i = \frac{c-1}{c^{N-1}}c^N - i, \tag{2.32}$$

with $0 < c < 1$.

2.6 Genetic Programming

Genetic programming (GP) is the evolution of computer programs [67, 68], where the genetic strings encode a program that solves a given problem. Koza and Cramer [24] suggested the use of tree structure as program representation in a genome. A program in genetic programming is built with functions and terminals. A terminal set is composed of the inputs to the GP program, the constants supplied to the GP program, and the zero argument functions, while a function set is composed of statements, operators, and functions available to the GP program [6]. Together, terminals and functions are referred to as nodes.

The function set is selected to fit the problem domain. The range of available functions is very broad. A GP program may use any programming construct that is available in any programming language. Examples of functions and constructs that can be used in GP include boolean functions, arithmetic functions, transcendental functions, conditional statements, loop statements and subroutines.

The first step in performing a GP run is the definition of the initial set of functions and terminals. An example of initial set of functions and terminals is give by

$$\begin{array}{rcl} F & = & \{+, -, \times, \div, \cos\} \\ T & = & \{x, y, z\} \end{array} .$$

The second step consists of creating an initial population of random programs by randomly choosing the branching nodes from the function and terminal set within a predefined maximum depth level. The depth of a node is the minimal number of nodes that must be traversed to get from the root node of the tree to the selected node. Unlike genetic algorithms, the individuals that are generated can have different lengths. As usual the fitness value of an individual is computed after the individual is evaluated on a given data set, or after the individual has lived and operated in a certain environment.

Crossover operator combines the genetic material of two parents by swapping a part of one parent with a part of the other. It exchanges subtrees between parents. It is important that the function set and terminal set are closed so that all possible combinations of subtrees correspond to legal programs. An example of crossover operator applied on tree-based GP programs is shown in Figure 2.12.

Mutation operates on only one individual. It deletes a randomly selected node or creates a new random subtree in its place. In GP, after crossover has occurred, each child produced undergoes mutation with a low probability.

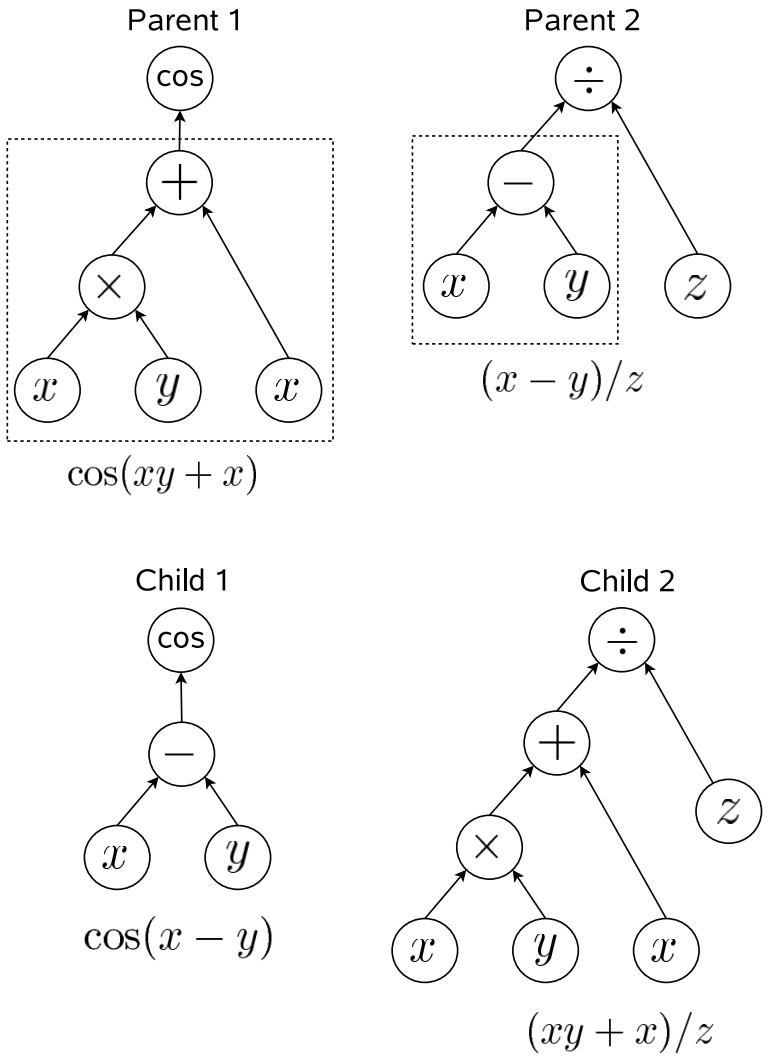


Fig. 2.12: Tree-based crossover. The crossover operator swaps a randomly selected subtree of one parent with a randomly selected subtree of the other. In the figure the subtrees to be swapped are enclosed by rectangles.

The other operator used in GP is the reproduction operator which simply produce a copy of an individual.

2.7 Evolution Strategy

Evolution Strategy (ES) was developed at Berlin Technical University by Ingo Rechenberg [79] and Hans Peter Schwefel [87]. Rechenberg and Schwefel were working with hydrodynamic problems when they hit upon the idea of using random events by throwing dice to decide the direction of an optimization process. Initially, evolution strategy experiments were restricted to variables with integer values, which represent parameters of an experimental setup to be optimized [52], and the population consisted of one individual only. In today's computer implementation, an ES algorithm uses vectors of real numbers and the population consists of many individuals.

For n parameters to be adapted, an ES chromosome is represented by a vector pair $\mathbf{c}$ given by

$$\mathbf{c} = (\mathbf{w}, \boldsymbol{\sigma}) = ((w_1, w_2, \ldots, w_n), (\sigma_1, \sigma_2, \ldots, \sigma_n)), \tag{2.33}$$

where $\mathbf{w}$ is the decision or object parameters vector and σ is the strategy parameters or learning rates vector. The adaptation of the strategy parameters is an integral part of the optimum search for the object parameters. The extension of the representation of individuals to include strategy parameters has introduced a distinction between phenotype and genotype. Strategy parameters are subjected to the same variation policy as are the object parameters. Selection indirectly favors the strategy parameter settings that are beneficial to make progress in the given problem domain, thus developing an internal model of the environment constituted by the problem. Self-adaptation of strategy parameters is the key aspect of ES which is the result of selecting better adapted individuals in both the domain of object variables and the domain of strategy parameters.

The mutation operator acts on both the object and strategy parameters and is given by

$$\sigma_i' = \sigma_i \, e^{\tau' \, N(0,1) + \tau \, N_i(0,1)}, \tag{2.34}$$

$$w_i' = w_i + \sigma_i \, N_i \, (0,1) \, , \tag{2.35}$$

where $\tau' = 1/\sqrt{2n}$ and $\tau = 1/\sqrt{2\sqrt{n}}$, and $N\,(0,1)$ is a random number drawn from a Gaussian distribution of zero mean and unity standard deviation. A boundary rule given by the following equation is used to force learning rates not to be smaller than a threshold value:

$$\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0. \tag{2.36}$$

Causality is emphasized in evolution strategy so that strong causes would generate strong effects. That means, large mutations should result in large jumps in fitness, and small mutations should result in small changes in fitness.

The recombination operator of two ES chromosomes $\mathbf{c}_1 = (\mathbf{w}_a, \boldsymbol{\sigma}_a)$ and $\mathbf{c}_2 = (\mathbf{w}_b, \boldsymbol{\sigma}_b)$ denoted by $\times_{rec}$ is defined as

$$\times_{rec}(\mathbf{c}_1, \mathbf{c}_2) = (\mathbf{w}', \boldsymbol{\sigma}') = ((w'_1, w'_2, \ldots, w'_n), (\sigma'_1, \sigma'_2, \ldots, \sigma'_n)), \tag{2.37}$$

where $w'_i = f_w(w_{a,i}, w_{b,i})$ and $\sigma'_i = f_\sigma(\sigma_{a,i}, \sigma_{b,i})$ for all $i \in \{1, 2, \ldots, n\}$. The functions f_w and f_σ are the recombination functions. In evolution strategies, two recombination functions play a prominent role: discrete and intermediate recombination. With discrete recombination one of the two respective components is randomly chosen,

$$f(x_1, x_2) = \begin{cases} x_1 & \chi \leq 0.5, \\ x_2 & \chi \geq 0.5, \end{cases} \tag{2.38}$$

where the function χ returns a uniformly distributed random number from interval $[0, 1]$. The components x_1 or x_2 are chosen with equal probability of 50%. For intermediate recombination the mean values of the respective components are computed using

$$f(x_1, x_2) = \frac{x_1 + x_2}{2}. \tag{2.39}$$

The intermediate operator is used for recombinations in particular among strategy parameters.

The selection operator in evolution strategy is truncation or (μ, λ) selection. It is a deterministic operator, which chooses the $\mu < \lambda$ individuals to constitute the population in the next generation. The symbol λ stands for the number of offspring and μ for the number of parents. The selection in ES is nearer to what Darwin called "natural selection" [89].

There are two important ways in which ES differs from Genetic algorithms (GA). First, there is no constraint on the representation. The typical GA approach involves encoding the problem solutions as a string of representative tokens. In ES, the representation follows from the problem. Second, the mutation operator simply changes aspects of the solution according to a statistical distribution which weights minor variations in the behavior of the offspring as highly probable and substantial variations.

2.8 Evolutionary Programming

Evolutionary programming (EP) is created in the early 1960s by Fogel, Owens, and Walsh [32]. They conducted experiments dealing mainly with the question of how to evolve computer programs, which are implemented as finite state machines, for prediction, control and pattern classification tasks.

A chromosome of length n in evolutionary programming is a real valued vector $\mathbf{c}$ given by

$$\mathbf{c} = (\mathbf{w}) = (w_1, w_2, \ldots, w_n), \tag{2.40}$$

where $\mathbf{w}$ is a vector of object variables. For such types of chromosomes used in EP, the mutation step-sizes are not self-adaptive. But as the optimal value for fitness is approached, the mutation rate is decreased. The fitness is made to influence the spread of mutations, for example, by tying it to the variance of the Gaussian distribution. The nearer the optimum, the sharper the distribution becomes around zero. An example of mutation operator [33] which depends on the fitness of a chromosome $\mathbf{c}$ is given by

$$w_i' = w_i + \left(\sqrt{k \cdot \text{fitness}(\mathbf{c}) + z}\right) \cdot N(0,1), \tag{2.41}$$

where $N(0,1)$ is a random number drawn from a Gaussian distribution of zero mean and unity standard deviation. The parameters k and z are determined by the domain expert and are tuned to the problem at hand. But usually, k is set 1 and z is set to zero so that the mutation operator depends on the fitness of the chromosome. The problem in using the above type of mutation operator is that for fitness functions whose global optimum is not zero, the spread of mutation will not come to zero.

In most of practical applications, the optimum value of the fitness function is not known. This implies that, it is not always possible to give a guarantee that the distribution of mutation step-sizes will come to zero. In order to avoid such problems, Fogel [31] developed a method called Meta-EP, where the original chromosome used in evolution programming is extended to included strategy parameters or adaptable mutation step-sizes. For the extended chromosome $\mathbf{c}$ given by

$$\mathbf{c} = (\mathbf{w}, \boldsymbol{\sigma}) = ((w_1, w_2, \ldots, w_n), (\sigma_1, \sigma_2, \ldots, \sigma_n)), \tag{2.42}$$

the mutation operator in EP is

$$\begin{array}{c} \sigma_i' = \sigma_i \cdot (1 + \alpha N_i(0,1)) \\ w_i' = w_i + \sigma_i' \cdot N_i(0,1) \end{array}, \tag{2.43}$$

where α is usually set to 0.2. Like evolution strategy, a boundary rule given by the following equation is used to force mutation step-sizes not to be smaller than a threshold value:

$$\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0. \tag{2.44}$$

Other variants of mutation operator are also used and tried. Mutation operators that used the lognormal scheme as in ES, and those that use Cauchy distribution instead of Gaussian are good examples. For modern EP, self-adaptation of mutation step-sizes is one specific feature. But unlike evolution strategy, EP still refrains from using recombination as a major operator for generating variants [31].

In order to generate the next population $P(t+1)$, first a reproduction operator is used to generate a copy of the current population $P(t)$ of size N. Then the mutation operator is applied on copy of $P(t)$ and is combined with $P(t)$ to form an intermediate population $P'(t+1)$, whose size is twice as large as the size of $P(t)$. Every individual in the intermediate population $P'(t+1)$ is then evaluated. A probabilistic (N, N) selection is applied on the intermediate population to form the next generation.

2.9 Behavior-Based Robotics

Behavior-based approach (embodied cognitive science) assumes the development of autonomous agents which can operate in an environment where the boundary conditions are changing rapidly [18, 66]. Brook's subsumption architecture was the first approach towards a new paradigm in the study of intelligence. It is a method of decomposing a robot's control architecture into a set of task achieving behaviors or competences. The approach taken by Brooks is purely reactive behavior-based method. He argued that the traditional *sense-plan-act* paradigm used in some of the first autonomous robots was in fact detrimental to the construction of real working robots [3]. In contrast to the traditional approach, the subsumption architecture builds control architectures by incrementally adding task-achieving behaviors on top of each other. He further argued that building world models and reasoning using explicit symbolic representational knowledge at best was an impediment to timely response and at worst actually led robotic researchers in the wrong direction. Figure 2.13 illustrates the conventional *sense-plan-act* and the new decomposition of a mobile robot control system based on task achieving behaviors.

The design decisions of the mobile robots in subsumption architecture are based on the following dogmatic principles:

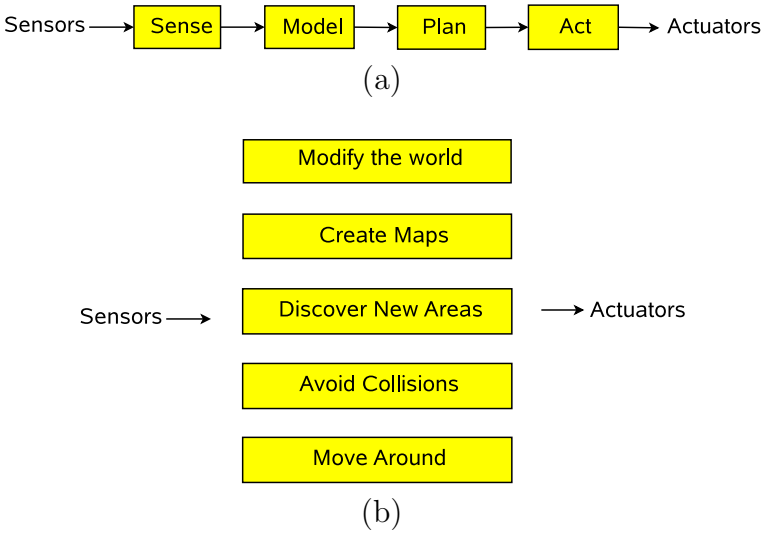


Fig. 2.13: The traditional *sense-plan-act* (a) and the subsumption architecture (b).

1. Complex behavior need not necessarily be a product of a complex control system [18]. It may be an observer who ascribes complexity to an organism. Complex behavior may be a reflection of a complex environment. Intelligence is in the eye of the observer [20].

2. There is no need for representation when intelligence is approached in incremental manner [20].

3. Robots should be cheap [17]. The term cheap is meant essentially three things. First, it implies exploiting the physics of the system-environment interaction. Second, it means exploiting the constraints of the ecological niche and third it means the system must be parsimonious. An ecological niche is defined by Wilson [106] as the range of each environmental variable such as temperature, humidity, and food items, within which a species can exist and reproduce.

4. The world is its own best model [20].

5. Robustness in the presence of noise or when one or more of its sensors fails or starts is the design goal.

6. Intelligence is emergent from an agent-environment interaction based on a large number of parallel, loosely coupled processes that run asyn-

chronously and are connected to the agent's sensory-motor apparatus [78, 19].

7. The control system of autonomous intelligent agents should be built incrementally [16].

In subsumption architecture, a class of desired behaviors that the robot should be able to perform in the environment in which it will have to operate is called level of competence. For example, one of the basic things a mobile robot should be able to do is to avoid objects on its way. That means, it should be equipped with the competence to avoid obstacles. Since obstacle avoidance is the most basic one, it is designated as a level 0 competence. Next there could be a move around or level 1 competence or any other sort of more complex competences like create maps.

Each level of competence is implemented as a layer of the control architecture. These layers can be built incrementally. This naturally leads to extendable designs in which new competences can simply be added to the already existing and functioning control system. Once each layer has been built and debugged, it never has to be changed again. The statement that the layers, once tested and debugged, never have to be changed is not always true. Sometimes links are added between layers to inhibit certain behaviors. Higher-level layers build and rely on lower-level layers. This is analogous to the evolutionary idea. In behavior-based robotics, we do not have to wait until all the layers have been designed and put together in order to operate the robot.

Each layer consists of a set of modules that asynchronously send messages to each other over connecting wires. In subsumption architecture, each layer is an augmented finite state machine. Other modules can suppress input to modules and inhibit outputs from modules. There is a certain amount of interaction between the modules but in order to achieve emergence and incremental complexity, the interaction between modules has to be minimized. The design of a subsumption architecture involves the following steps: (1) identification of behaviors of the robot that should be performed (2) the organization of the level of competences. A very good example of designing real agent using subsumption architecture that follows an object is described by Bunten [22].

Sommer [93] proposed the algebraic aspects of designing behavior-based systems. He suggested that a common theoretical framework is necessary for combining robotics, computer vision, neural computation and signal theory. The framework consists of a global algebraic frame for embedding the perceptual and motor categories, a local algebraic framework for bottom-up construction of necessary information, and a framework for learning and

self-control that is based on the equivalence of perception and action. He identified Geometric algebra [47] as an adequate global algebraic frame and the Lie theory [99] as adequate local algebraic frame.

Pfeifer [78] suggested design principles of behavior based systems that base themselves on embodied cognitive science. The design principles are different from engineering designs since engineering designs require no behavioral diversity in their methods while the design principles in cognitive science require behavioral diversity. A brief summary of the design principles is given as follows:

1. **The three-constituents principle**: The design of autonomous intelligent agent constitute the following three fundamental components.
   - Definition of ecological niche
   - Desired behaviors and tasks
   - Agent design: Steps 2-7 explain the design of the agent itself.
2. **The complete agent principle**: This design principle implies that agents should be autonomous, self-sufficient, embodied and situated. Even though this principle is extremely powerful, it is not usually considered explicitly. All the artificial agents designs to date do not entirely fulfill the complete agent principle. In contrast, all natural systems fulfill the complete agent principle.
3. **The principle of parallel, loosely coupled processes**: This principle assumes that intelligence is emergent from an agent-environment interaction, and from a large number of processes that run asynchronously and connected to the agent's sensory-motor apparatus. The term "loosely coupled" refers to the coupling through the interaction with the environment.
4. **The principle of sensory-motor coordination**: The sensory-motor coordination enables the agents to interact efficiently with the environment. It also serves the purpose of structuring the sensory input.
5. **The principle of cheap design**: Designs must be parsimonious and exploit the physics and constraints of the ecological niche.
6. **The redundancy principle**: Applying this principle to the design of agents results in agents that are robust whenever some of their part stops to function. The redundancy of the sensors will give the agent the ability to function for example in a dark room.

7. **The value principle**: Autonomous systems should have a value system that guides the mechanisms of self-supervised learning employing principles of self-organization. The value system modulates the learning process by increasing the probability that an agent gets into a situation.

2.10 Summary

The basic components used in the design of the learning and adaptation system for autonomous intelligent agents are briefly discussed. In the system presented in this thesis, neural networks represent the optimal value function to be learned. The learning and adaptation concepts in reinforcement learning and evolutionary methods are used for the evolution of the structures and connection weights of neural networks. The design principles of behavior based systems are employed in the complexification process of neural networks.

Chapter 3

MODEL BASED EVOLUTIONARY OBJECT RECOGNITION SYSTEM

In this chapter, an application example of evolutionary methods to the problem of object recognition is presented. A novel evolutionary object recognition system, which is independent of translation, rotation and scaling, for recognizing 2D plane objects in 3D and determining their 3D poses simultaneously is proposed.

Object recognition is one of the most important, yet least understood, aspect of visual perception [101]. For many biological vision systems, the recognition and classification of objects is spontaneous, natural activity. In contrast, the recognition of common objects is still way beyond the capability of artificial systems, or any recognition system proposed so far.

The proposed system has the following features which makes it different from other recognition systems:

1. A common unifying approach for both recognition and pose estimation, treating both subtasks at the same time.

2. Finding a global optimal solution in space of solutions. The space of solutions is made up of the Cartesian product of objects to be recognized and their poses.

By pose, we mean the transformation needed to map an object model from its own inherent coordinate system into agreement with the sensory data [38]. The system extracts and saves contour points of objects from training images as their contour models.

3.1 Previous Work

There are different approaches to object recognition that have been proposed. One can divide them in general in two groups: non-correspondence or global

matching and correspondence or feature matching. A global matching involves finding a transformation that fits a model to an image without first determining the correspondence between individual parts or features of the model and the data. Several of this approaches base themselves on a simple geometric parameters such as area, perimeter, Euler number, moments of inertia, Fourier or other spatial frequency descriptions, tensor measures and so on. Representative examples include works of Hu, Murase and Nayar, and Zahn and Roskies [49, 74, 109]. These methods are efficient but are sensitive to occlusion [38]. On the other hand, feature matching procedures try to find the correspondence between local features of the model and the data, and then determining the transformation for a given correspondence between the model and the data. These methods are robust against occlusion but are not as efficient as global matching procedures [38, 82, 101] since they have to solve the correspondence problem for every model that is going to be assumed.

Genetic algorithms are mostly employed in searching the best correspondence between a given model and the data, or in searching the best geometric transformation that brings a large number of model points into alignment with the scene [9, 12]. In our implementation, we use genetic algorithm to search for a model that best fits a given scene and simultaneously determines the transformation that brings the model into alignment with the scene. The presented system belongs to the global matching group.

3.2 Object Recognition System

Our system has three parts. The first part performs a simple visual grouping based on predefined colors of an object. The second part does the recognition and pose estimation of the perceived object and the third part is used to acquire new models of new objects that are not known to the system.

3.2.1 Visual Grouping

There is a considerable evidence that prior to the recognition, the early processing stages in the visual cortex are involved in grouping and segmentation operations on the base of image properties, such as proximity, collinearity, similarity of contrast, color, motion, texture, etc [101]. The grouping and segmentation process attempts to organize the image into coherent units, and to decide what parts of an image belong together.

Our visual grouping algorithm tries to group objects based upon predefined colors of an object. The visual grouping is done as follows:

1. Read in an RGB color image.
2. Convert the image into its HSV (Hue, Saturation, Value) image format.
3. Generate a binary image with pixels marked for the object of interest with predefined colors.
4. Use graph-search or an equivalent algorithm to group pixels that come from the same object, and extract image regions (components).
5. Get the centroid and number of pixels of each of the components in the image.

3.2.2 Recognition and Model Acquisition Systems

The recognition system starts by generating a binary contour image from the components image using a standard contour following algorithm. Then it generates the complement image of the contour image and calculates the distance transform [45, 92] of it. The distance transform assigns a distance value of zero to all contour points and a positive distance value to all non-contour points as the distance to the closest contour point depending on the distance metric used. In calculating the distance transform, we have assumed that the complement of the contour image is toroidal rather than planar. Figure 3.1 shows the steps involved in getting the distance transform of an input image to the recognition system. The distance transform is used in determining the fitness value of an individual.

(a) (b) (c) (d) (e)

Fig. 3.1: The steps involved in getting the distance transform taking letter A as an example. (a) The input image. (b) The components image which is the result of the visual grouping algorithm. (c) The contour image generated by the standard contour following algorithm. (d) The complement of the contour image. (e) The distance transform of the complement of the contour image.

In this work, we have used a chromosome (an individual) shown in Figure 3.2. The chromosome has four genes. The first gene codes the index of an object in the database of the contour models, which are already acquired

by the system. The second, third and fourth genes code the rotation of the object about z-axis, y-axis and x-axis respectively relative to the non-rotated model of the object. The centroid of the model is taken as the origin of the coordinate system and the z-axis is assumed to be perpendicular to the image plane of the camera. The length of the chromosome is determined by the number of contour models in the system and the number of bits used in representing the orientation of an object. The number of bits that code the rotation of an object about an axis determines the resolution of the rotation angle coded by the chromosome.

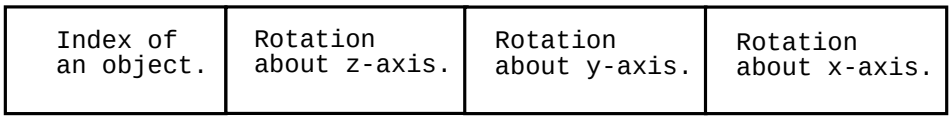


Fig. 3.2: A chromosome: The first gene codes the object to be recognized. The second, third and fourth genes code the orientation of the object.

A fitness function of a chromosome given by

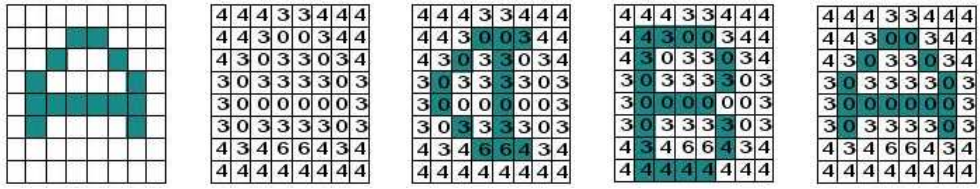
$$f = -\sum_i d_i, \tag{3.1}$$

is used to determine the fitness value of an individual. It is defined as the negative sum of the distance values, d_i, in the distance transform of the complement image of the contour image. The index i runs for all points of the translated, scaled and rotated contour model of an object coded by the chromosome. A zero value of the fitness function means a perfect fit of the model to a particular component in the image since the distance values for the contour points in the distance transform are zero.

Figure 3.3 shows a contour image of an 8×8 hypothetical input image, the distance transform of the complement of the contour image and contour models coded by three different chromosomes which are projected onto the distance transform. The fitness value of the chromosomes coding the contour models shown in Figure 3.3(c), 3.3(d) and 3.3(e) are -31 , -39 and 0 respectively. Figure 3.3(e) shows a case where the model fits to the component perfectly.

The following steps are used in evaluating an individual:

1. Decode the individual. That is, get the type of the contour model with its orientation coded by the chromosome.



(a) (b) (c) (d) (e)

Fig. 3.3: Determination of the fitness value of a chromosome. (a) The contour image. (b) The distance transform of the complement of the contour image. (c), (d) and (e) Contour models coded by different chromosomes and projected onto the distance transform.

2. Calculate the scale factor between the rotated contour model and the component using,

$$s = \sqrt{\frac{A_i}{A_m}} \tag{3.2}$$

where A_i is the area of the component in the image and A_m is the area of the orthogonal projection of the rotated contour model onto the plane parallel to the image plane of the camera.

3. Translate the contour model to the centroid of the component. Then rotate it using the orientation angle obtained when decoding the chromosome and scale it using the scale factor calculated above.

4. Determine the fitness value of the individual using equation (3.1).

The position (centroid) of a component and the number of pixels of the component are determined by the visual grouping algorithm. The position of a contour model and the scale factor between the contour model and the component are determined while an individual (chromosome) is being evaluated. The search for the best model with best orientation that fits a given component in the image is done by the genetic algorithm. One can see that the recognition and pose estimation problems are solved simultaneously by the recognition system.

The main purpose of the genetic algorithm is to solve the optimization problem of finding the maximum value on the fitness landscape. Unlike traditional search algorithms like gradient-following strategy, genetic algorithms are not trapped in a local minima. They usually find the global maximum of a multimodal function having many local maxima and minima. Figure 3.4 shows an example of a fitness landscape of an input image containing letter

W, which is rotated by 180 degrees about z-axis. The coordinate at which the global maximum value of the fitness function occurs gives the index of the object recognized with its rotation angles. The genetic algorithm uses its genetic operators to move on the fitness landscape.

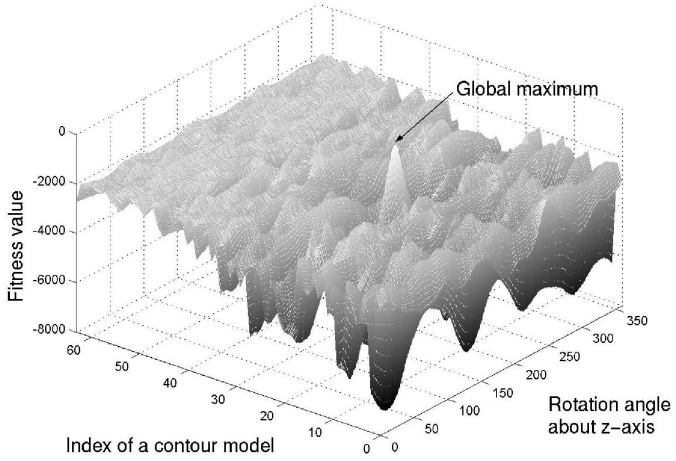


Fig. 3.4: A fitness landscape of an input image containing letter W rotated by 180 degrees about z-axis taking the centroid of letter W as the origin of the coordinate system. The global maximum of the landscape occurs at the index of the contour model of letter W in the database of the contour models and at rotation angle of 180 degrees.

The recognition system executes the following two important steps while recognizing an object:

1. Perform a complete genetic run for a component until the best individual is found.

2. Decode the best individual and return the object recognized with its pose.

The main advantage of the recognition system is that it can be easily parallelized. This can be done in three different ways. First, we can have copies of the algorithm that operate in parallel on different components detected in the input image so that the recognition of all detected components can be done simultaneously. Second, we can run a parallel implementation of the algorithm for one component and apply it to the rest of the components

sequentially. The algorithm can be parallelized since genetic algorithms let themselves easily parallelized [103]. This is specially useful if we want to increase the recognition rate of our system. Third, we can combine the above types of parallelizations and benefit an increase in both the recognition rate and speed of recognition.

The model acquisition system is used to acquire and save a new model for a new object for which the model does not exist in the system. It uses the above visual grouping algorithm to identify and locate the object. Then it uses a contour following algorithm to extract the contour points of the object and samples the contour points evenly so that the resulting model has the same number of points as the other models in the system. This is important because the fitness value of an individual depends on the number of contour points of a model coded by the chromosome (individual).

3.3 Experiments and Results

For all the experiments, we have set the parameters of the genetic algorithm as shown in Table 3.1. The system is made to acquire models of 64 objects some of which are shown in Figure 3.5. The objects include the English alphabets, digits, free hand drawn objects and some German traffic signs. The resolution of the system in coding the rotation angle of the object about an axis is 0.7045 degrees.

No. of individuals in the population	500
Crossover probability	0.2
Mutation probability per bit	0.05
Selection scheme	Truncation
No. of bits per gene coding the index of an object	6
No. of bits per gene coding a rotation angle about z-axis	9
No. of bits per gene coding a rotation angle about y-axis	9
No. of bits per gene coding a rotation angle about x-axis	9

Tab. 3.1: Parameters of genetic algorithm used

Experiment on Artificial Images

In this experiment, we study the effect of random noise on the recognition rate and pose estimation of the system. Each input image is subjected to

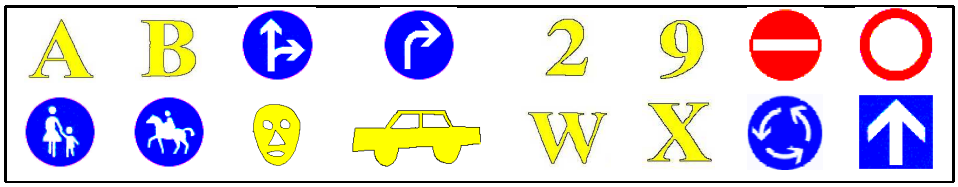


Fig. 3.5: Some of the objects whose models are acquired by our system.

different levels of random corruption between 0% to 50%. The percentage of the noise levels shows the ratio of the number of pixels that are corrupted by the random noise to the total number of pixels in the image. A uniform random function generator is used to select a pixel in the input image. The color of the selected pixel is made to change to some other color which is not used in the visual grouping algorithm.

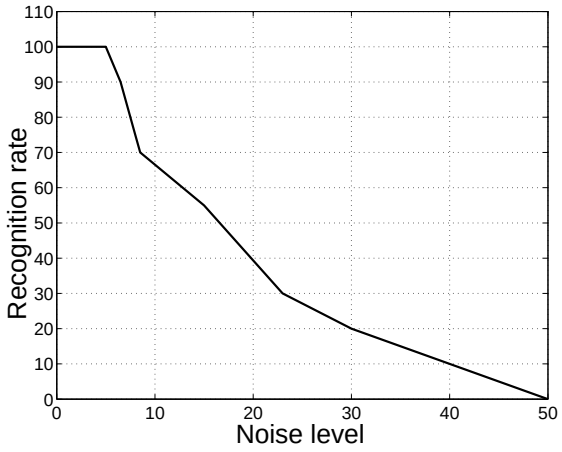


Fig. 3.6: Recognition rate obtained for different random noise levels.

Figure 3.6 shows the result obtained for different noise levels added to the test images. As the test images, we have used translated, scaled and rotated versions of all images whose models are acquired by our system. The translation, scale and rotation parameters of each image is randomly selected. As can be seen in Figure 3.6, the system has 100 percent recognition rate for noise level up to 5%. This makes the system robust to be used for real life applications.

In order to investigate the pose (rotation) estimation capability of our system, we have generated rotated versions of one of the German traffic signs shown in Figure 3.7. The non-rotated version of this traffic sign was already

acquired by our system. We have measured the average pose estimation error of 10 experiments, which are run for 0%, 1%, 5% and 10% noise levels and for all rotation angles shown in Figure 3.7.

Image									
Angle	0	20	40	60	80	100	120	140	160
Image									
Angle	180	200	240	260	280	300	320	340	360

Fig. 3.7: Rotated versions of the German traffic sign used to indicate a pedestrian path.

Each of the experiments are done by starting the recognition system with different random seed values. As can be seen from Figure 3.8, the system is able to estimate the pose of an object with absolute maximum pose estimation error of 5.5 degrees for noise level of 10%. This makes the system again robust for estimating the pose of an object.

Figure 3.9 shows sample recognition results obtained on artificial images. As can be seen from the figure, the recognition system has found in (d), (e), (f) and (h) equivalent rotation angles, which will result in the same image projected onto the image plane of the camera.

Experiment on Real Images

Figure 3.10 shows sample recognition results obtained in recognizing real 2D images. For the experiments, we have used plane images printed on sheet of papers and some real German traffic signs.

For real images, we have obtained a recognition rate of 95%. This recognition rate can be increased even to a higher level if one uses a parallel implementation of the recognition system as stated in Section 3.2, where different recognizers work on one component at the same time.

We have tested and implemented our system on a standard 800 MHz computer running the Linux operating system. On the system, our algorithm was able to process 2 input images per second if the image contains one component. The recognition time of the system increases linearly with the number of components detected in the input image.

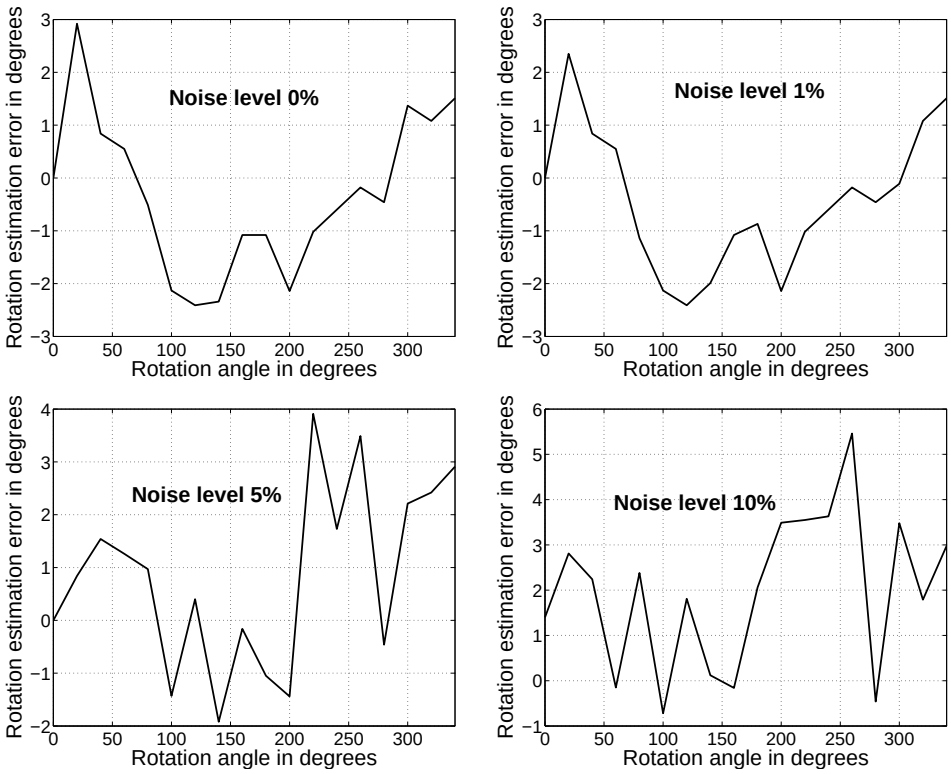


Fig. 3.8: Pose estimation error for 0, 1, 5 and 10 percent noise levels.

3.4 Summary

A novel evolutionary object recognition system, which is independent of translation, rotation and scaling, for recognizing 2D plane objects in 3D and determining their 3D poses simultaneously is presented. The system has three important components. The first part performs the visual grouping for detecting pixels that come from the same object. The second part does the recognition and pose estimation using genetic algorithm and the third part is used to acquire a model of new object whose model is not already in the system.

From the experiments, it can be concluded that one can use the power of genetic algorithms to search for the best fit in the space of objects and their pose. The presented system is suitable for applications requiring the recognition of navigation symbols such as traffic signs.

The system has one limitation. It is not robust with respect to occlusions. If an object is occluded, the system may not recognize the object correctly. The system can be extended to handle the occlusion problem by redesigning

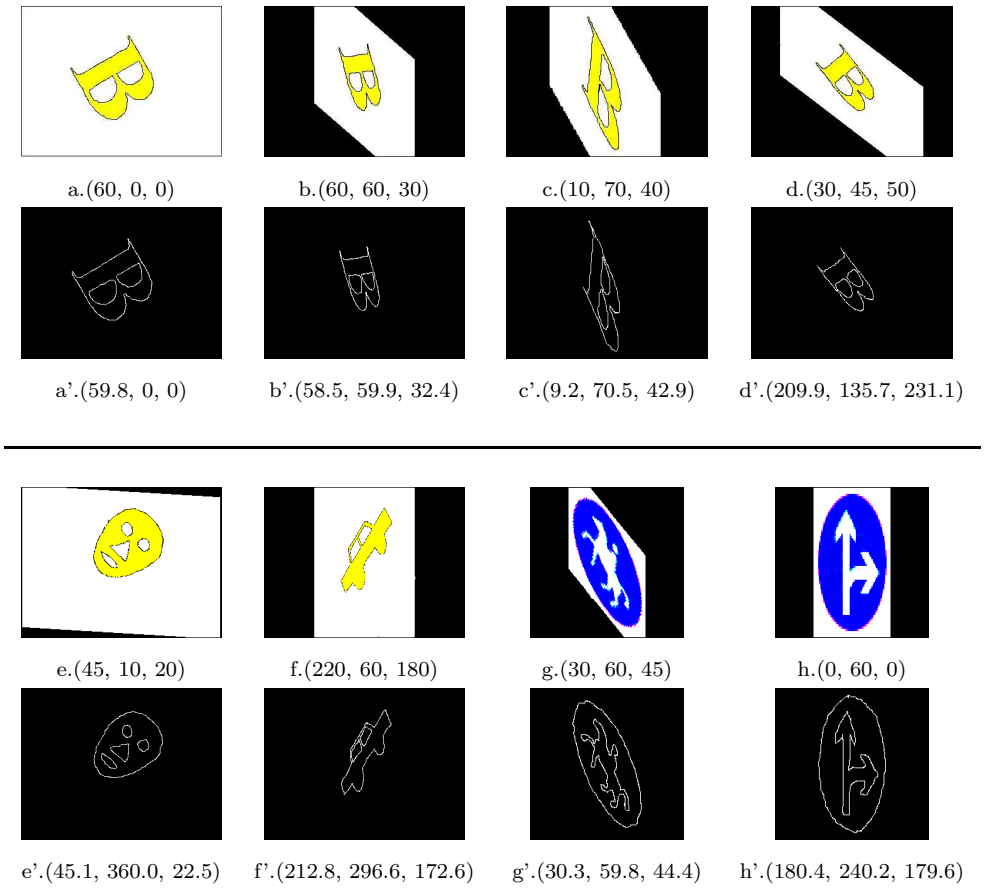


Fig. 3.9: Sample recognition results obtained from experiments on artificial images. The numbers in the brackets below each of the images show the rotation angles of the images about z, y and x-axis respectively. The contour images show the recognition result obtained.

the visual grouping algorithm to use other features such as object boundaries.

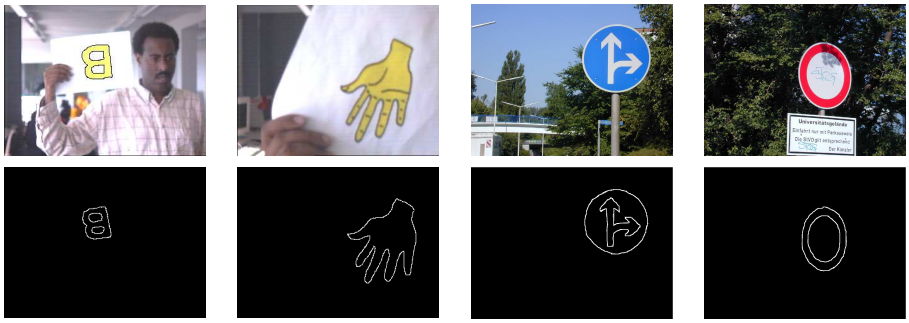


Fig. 3.10: Sample recognition results obtained in recognizing real 2D images. The contour images below each of the images show the type of the object that is recognized.

Chapter 4

IMPROVING LEARNING AND ADAPTATION CAPABILITIES OF AGENTS

When an infant learns how to go and how to stand, it has no explicit teacher, but it does have a direct sensory-motor connection to its environment. Through 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 [97]. It is used by agents at the individual level. In this chapter, 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 on its own 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 [75].

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 lifetime 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 in a 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 [52]. How could an individual learn to use its eyes if it had not 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 [42, 97] 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 [97]. Like the infant, an agent using reinforcement learning learns and adapts itself through interaction with the environment. In this chapter, we use Q-learning [8], 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" [6]. 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.

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 Q-learning and genetic algorithm in answering the above questions. The agents live and operate in an artificial robot world.

4.1 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` [42]. 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 4.1. 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 (4.1).

$$R^a_{s_t,s_{t+1}} = \begin{cases} 0 & \text{if} \quad s_{t+1} \neq s_t \\ 1 & \text{if} \quad s_{t+1} = \text{goal state} \\ -1 & \text{if} \quad s_{t+1} = s_t \end{cases} . \tag{4.1}$$

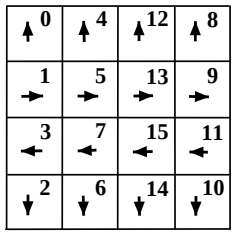
The negative numerical reward in equation (4.1) discourages agents attempting an action against the world boundary. This action does not change the state of the environment. For genetic algorithms, a fitness function given by equation (4.2) is used.

$$f(n) = \gamma^n. \tag{4.2}$$

The quantity f represents the fitness value of an individual, where $\gamma \in [0, 1)$ is a constant, and n is the number of steps taken by the point robot from a given start state to a given goal state. Equation (4.2) encourages those individuals that go from the start state to the goal state through a shortest path. Figure 4.2 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:

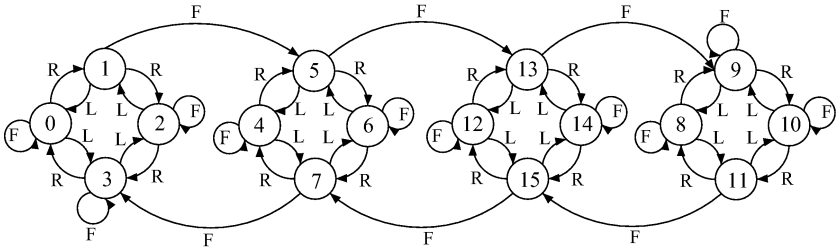
1. The experiments have to be carried out for a large number of times for different conditions of learning and adaptation experiments. This requires a lot of time and energy to execute all the experiments on real robot until one gets agents with satisfactory behaviors.



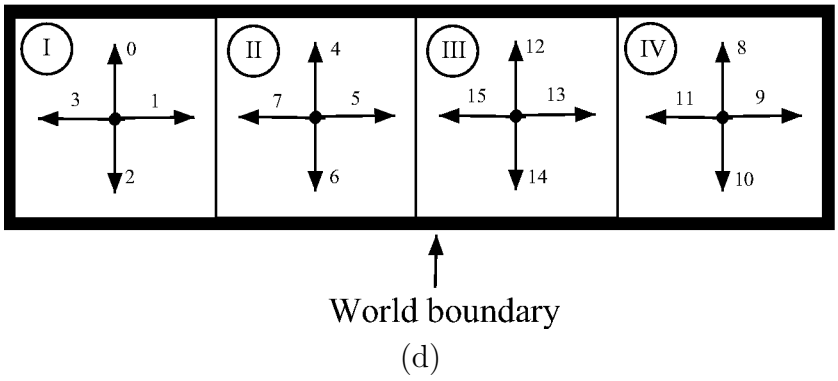
(a)

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. 4.1: 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.

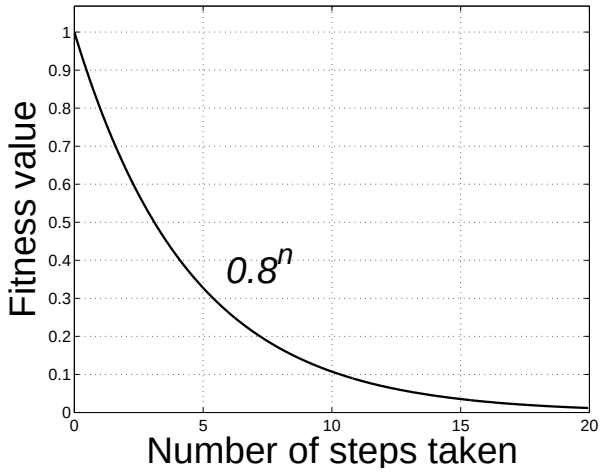


Fig. 4.2: The fitness function for $\gamma = 0.8$. The fitness value of an individual gets higher as the number of steps taken by the individual gets smaller.

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 causes 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.

4.2 What to Learn?

The agent learns on-line through interaction with the environment either the optimal policy for perceived states or the action values of the states 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. An action value of a state shows "how good" it is for an agent to perform a given action in a given state.

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 algorithms, the agent learns directly the optimal policy without having to learn the model of the environment. In Q-learning, the agent learns the action values and saves them in a Q-table [42]. It can generate the optimal policy for perceived states from the Q-table.

4.3 Experimental Setup

The following test cases are selected for all experiments that are presented in this thesis. 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 [42, 97] 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 chapter, 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.

4.4 Offline Solution to the Optimal Policy in the Artificial Robot World

In this section, a discussion is given on how to obtain the optimal solution using different methods. The following assumptions are made before finding 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 (2.10) and (2.11). For our test scenario, the dynamics of the environment is given by equation (4.3).

$$P^a_{ss'} = \begin{cases} 1 & \text{if } s' \text{ is a valid next state} \\ 0 & \text{otherwise} \end{cases}$$
$$R^a_{ss'} = \begin{cases} 0 & \text{if } s_{t+1} = s' \\ 1 & \text{if } s_{t+1} = \text{goal state} \\ -1 & \text{if } s_{t+1} = s_t \end{cases} \tag{4.3}$$

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

$$\begin{aligned} V^*(0) &= \max \begin{cases} \gamma V^*(3) & \text{left} \\ \gamma V^*(1) & \text{right} \\ -1 + \gamma V^*(0) & \text{forward} \end{cases} \\ V^*(1) &= \max \begin{cases} \gamma V^*(0) & \text{left} \\ \gamma V^*(2) & \text{right} \\ \gamma V^*(5) & \text{forward} \end{cases} \\ &\vdots \\ V^*(15) &= \max \begin{cases} \gamma V^*(14) & \text{left} \\ \gamma V^*(12) & \text{right} \\ \gamma V^*(7) & \text{forward} \end{cases} \end{aligned} \tag{4.4}$$

In equation (4.4), 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^* [97]. 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 4.1 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 ahead 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. 4.1: The optimal state value for the goal state 15.

We can also solve equation (4.4) using a 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 4.3. The index of a gene along the chromosome is the same as the corresponding state in the robot world.

Fig. 4.3: 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 (4.2) 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

$$f = \sum_{s=0}^{16} \gamma^{n_s}, \tag{4.5}$$

where n_s represents the number of steps taken by the point robot from a starting state s. Table 4.2 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. 4.2: 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 (4.6).

$$\begin{aligned} V^*(0) &= \gamma V^*(1)\,\text{right} \\ V^*(1) &= \gamma V^*(5)\,\text{forward} \\ &\vdots \\ V^*(15) &= \gamma V^*(14)\,\text{left} \end{aligned} \tag{4.6}$$

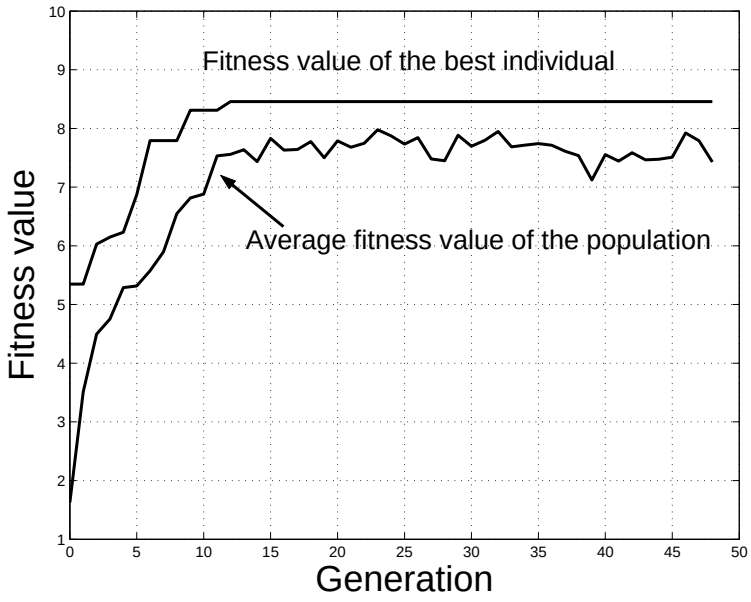


Fig. 4.4: A genetic algorithm run for finding the best system of simultaneous equations. The best equation is found after the 11^{th} generation.

The solution of equation (4.6) is the same as the solution found by applying value iteration algorithm, which is shown in Table 4.1. 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 2.8 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.

Conclusion: 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 do not necessarily require the knowledge of the dynamics of the environment a priori. Genetic algorithms 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.

4.5 Learning and Adaptation at Individual Level

Organisms, for example human beings, are always learning and adapting to their environment in their lifetime. Much of the learning is done through direct interactions with the environment. Consider a person who cannot 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 large number 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 forward speed of the bicycle relative to the ground. Depending on the reward received and the state of the bicycle, the person has to execute a sequence of actions to keep the bicycle upright and moving forward at 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 section and the following sections.

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 only for perceived states.

4.5.1 Q-Learning

Q-learning is an online 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

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - Q\left(s_t, a_t\right)\right], \tag{4.7}$$

which is equivalent to

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} - Q\left(s_t, a_t\right)\right]. \tag{4.8}$$

With online learning, the agent cannot 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 (4.9).

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right] \tag{4.9}$$

If we want equation (4.9) 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^*,

$$Q\left(s_{t+1}, a_{t+1}\right) = \max_a Q\left(s_{t+1}, a\right). \tag{4.10}$$

Using equation (4.10), we obtain an equation for the Q-learning algorithm,

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right]. \tag{4.11}$$

For a correct 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 [8].

4.5.2 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,

$$P(a) = \frac{e^{-Q(s,a)/\tau}}{\sum_{a' \in A} e^{-Q(s,a')/\tau}} \tag{4.12}$$

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.

4.5.3 Experiments and Results

The experiments are done for all test cases mentioned in Section 4.3. Table 4.3 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 (4.1) is used to evaluate the actions executed by the agent.

α	γ
0.3	0.3

Tab. 4.3: 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.

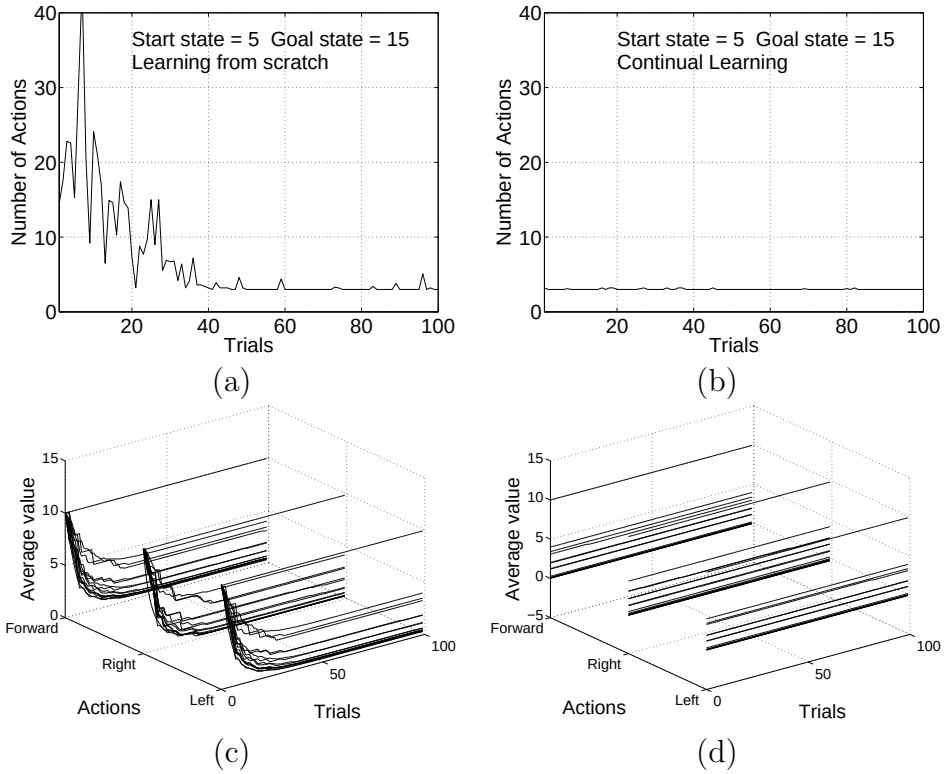


Fig. 4.5: Learning from scratch and continual learning. (a) Average number of actions taken versus trials in learning from scratch. (b) Average number of actions taken versus trials in continual learning. (c) Average action values for each state in learning from scratch. (d) Average action values for each state in continual learning.

From Figure 4.5, 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.

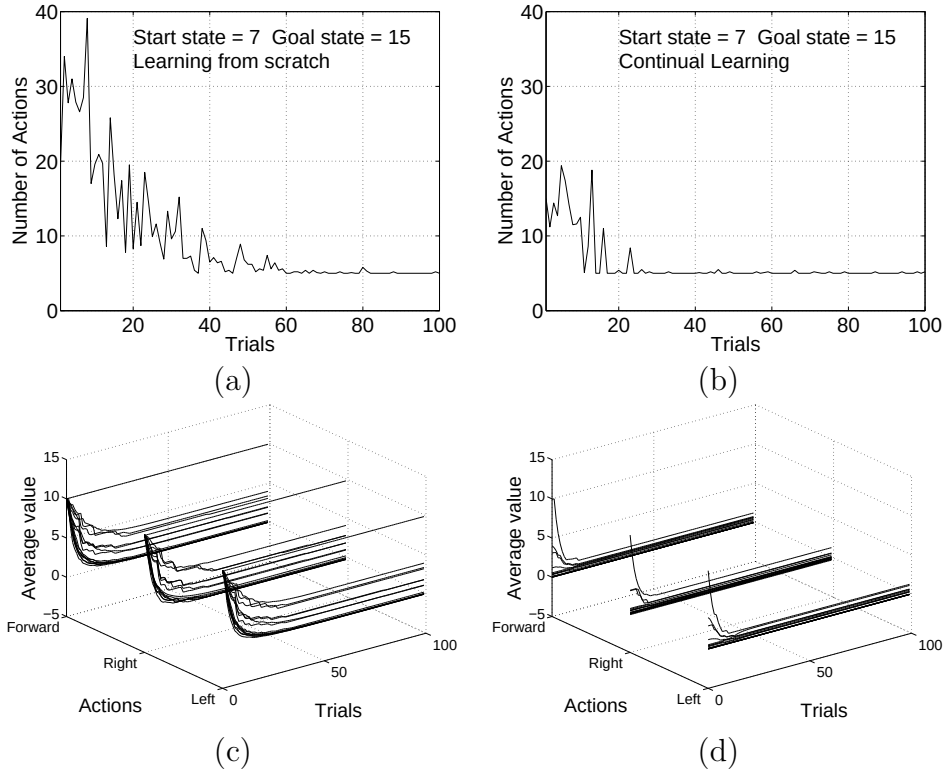


Fig. 4.6: Learning from scratch and continual learning. (a) Average number of actions taken versus trials in learning from scratch. (b) Average number of actions taken versus trials in continual learning. (c) Average action values for each state in learning from scratch. (d) Average action values for each state in continual learning.

Figure 4.6 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.

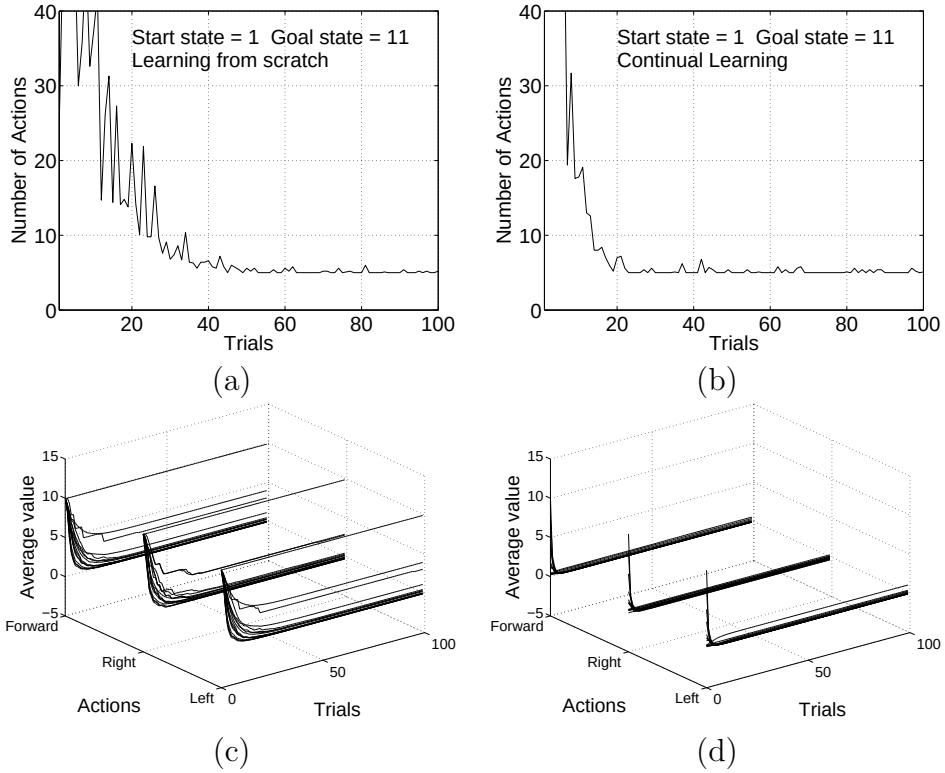


Fig. 4.7: Learning from scratch and continual learning. (a) Average number of actions taken versus trials in learning from scratch. (b) Average number of actions taken versus trials in continual learning. (c) Average action values for each state in learning from scratch. (d) Average action values for each state in continual learning.

As can bee seen in Figure 4.7, 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 (give 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.

4.6 Learning and Adaptation at Population Level

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.

In this section, learning and adaptation at population level where the population is made up of neural networks is investigated. 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. The genetic algorithm searches for the best neural network by directly determining the synaptic weights of the networks.

4.6.1 Experiments and Results

In the experiments the weights of the neural networks are directly determined by the genetic algorithm. That means the evolutionary method searches for the optimal policy directly in the space of policies represented by the neural network. 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. In this experiment, we used MLPs having two outputs, four input units and six hidden nodes. The MLP controls the point robot in the robot world. The genetic algorithm lets each individual 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 a minimal number of steps. It then applies genetic operators to generate the next population of MLPs for

a predefined number of trials.

Figure 4.8 shows an example of the MLP used in this experiment. Table 4.4 shows the encoding of states and actions, which are the input and output of the neural network, respectively.

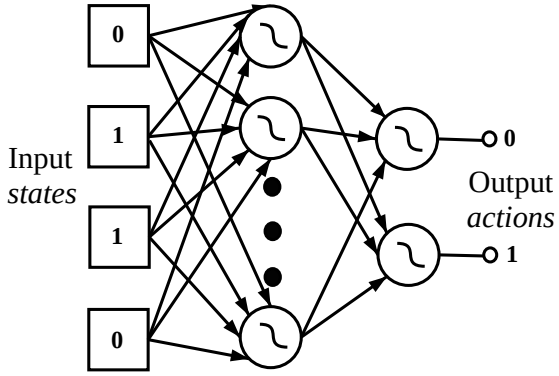


Fig. 4.8: The MLP used in the experiment. The input and output of the MLP are binary codes of the states and actions.

State	Code
0	0000
⋮	⋮
14	1110
15	1111

(a)

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

(b)

Tab. 4.4: The encoding of the states (a) and actions (b).

A fitness function given by equation (4.2) is used to evaluate the individuals. An example of a chromosome representing an MLP (an individual) is shown in Figure 4.9. The parameters of the neural network and genetic algorithms are given in Table 4.5.

We have run the experiment for all test cases and obtained the result shown in Figure 4.10. 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

$Wo_{1,1}$	$\cdots$	$Wo_{1,N}$	$Wo_{2,1}$	$\cdots$	$Wo_{2,N}$	$Wh_{1,1}$	$\cdots$	$Wh_{1,4}$	$Wh_{N,1}$	$\cdots$	$Wh_{N,4}$

Fig. 4.9: 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.

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. 4.5: Parameters of the MLPs and genetic algorithm used for the experiments.

individual) and the other force, which is controlled by the crossover and mutation operators, tries to maintain the variation between individuals. The learning time, which is measured in number of generations, required to attain a certain average fitness value is shorter in continual learning than the learning time in learning from scratch for all test cases.

4.7 Hybrid of Learning and Evolutionary Algorithms

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 lifetime, 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

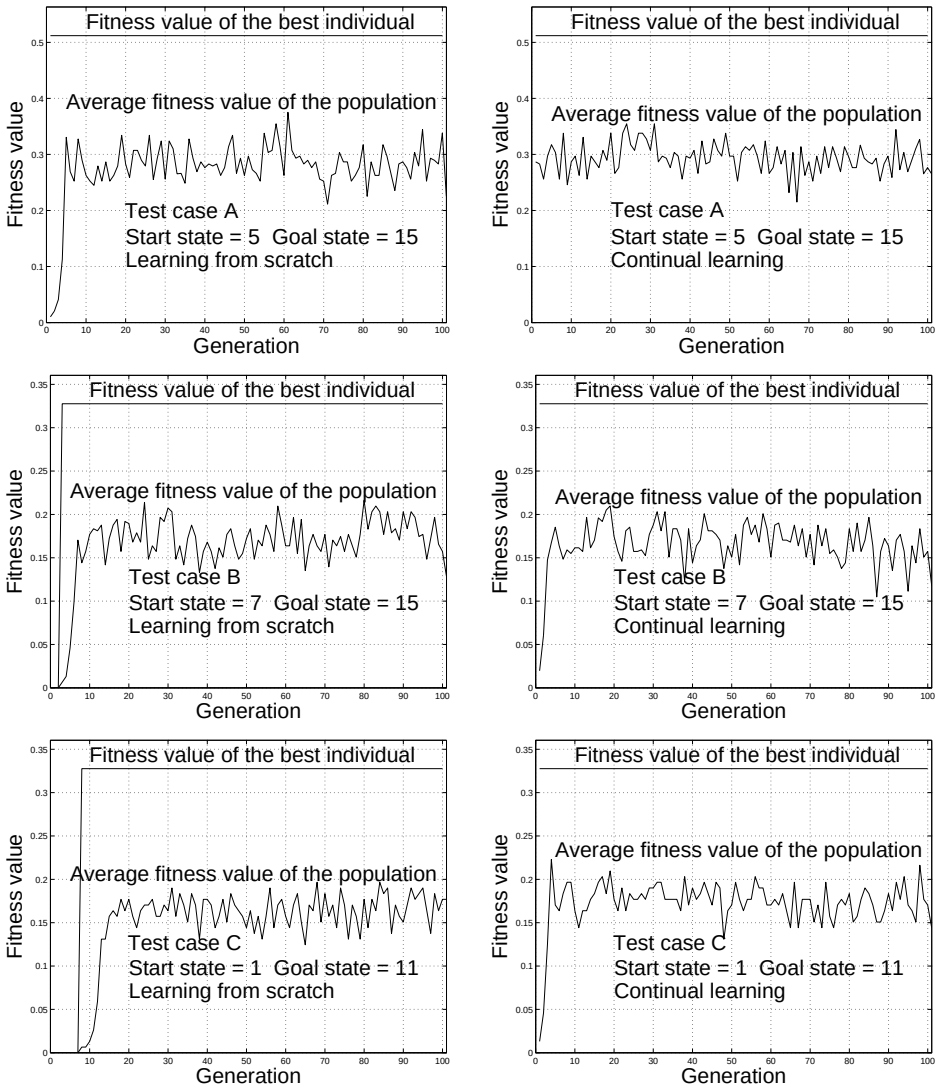


Fig. 4.10: Result obtained for a hybrid of MLPs and genetic algorithm. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

ancestors. The inherited genes in offspring are not exact copies of the genes in the parents because of the genetic mutation and recombination operators.

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 Darwinism is "non-random natural selection following 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 cumulative 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 [86].

4.7.1 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 lifetime is not transferred to the next generation.

Table 4.6 shows the Q-table and the chromosome that encodes it and which is used in this experiment.

The reward function given by equation (4.1) is used for the reinforcement learning agents, and the fitness function given by equation (4.2) is used for the genetic algorithm. The parameters for the genetic algorithm and the reinforcement learning are shown in Table 4.7.

The experiment is run for all test cases and the results shown in Figures 4.11 and 4.12 are obtained for learning and adaptation at the population

	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`	$Q_{2,0}$	$Q_{2,1}$	$\cdots$	$Q_{2,14}$	$Q_{2,15}$

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

Tab. 4.6: The Q-table and the chromosome that encodes it.

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. 4.7: The parameters of genetic algorithm and reinforcement learning.

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 [43]. However, 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 cost.

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 previous section 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 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.

4.8 Summary and Analysis of Results

Table 4.8 summarizes the results of the experiments. All experiments were run 50 times and the result in Table 4.8 are the average number of trials to accomplish a certain learning task. For evolutionary experiments all individuals have exactly one trial per generation.

The results of experiments clearly show 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

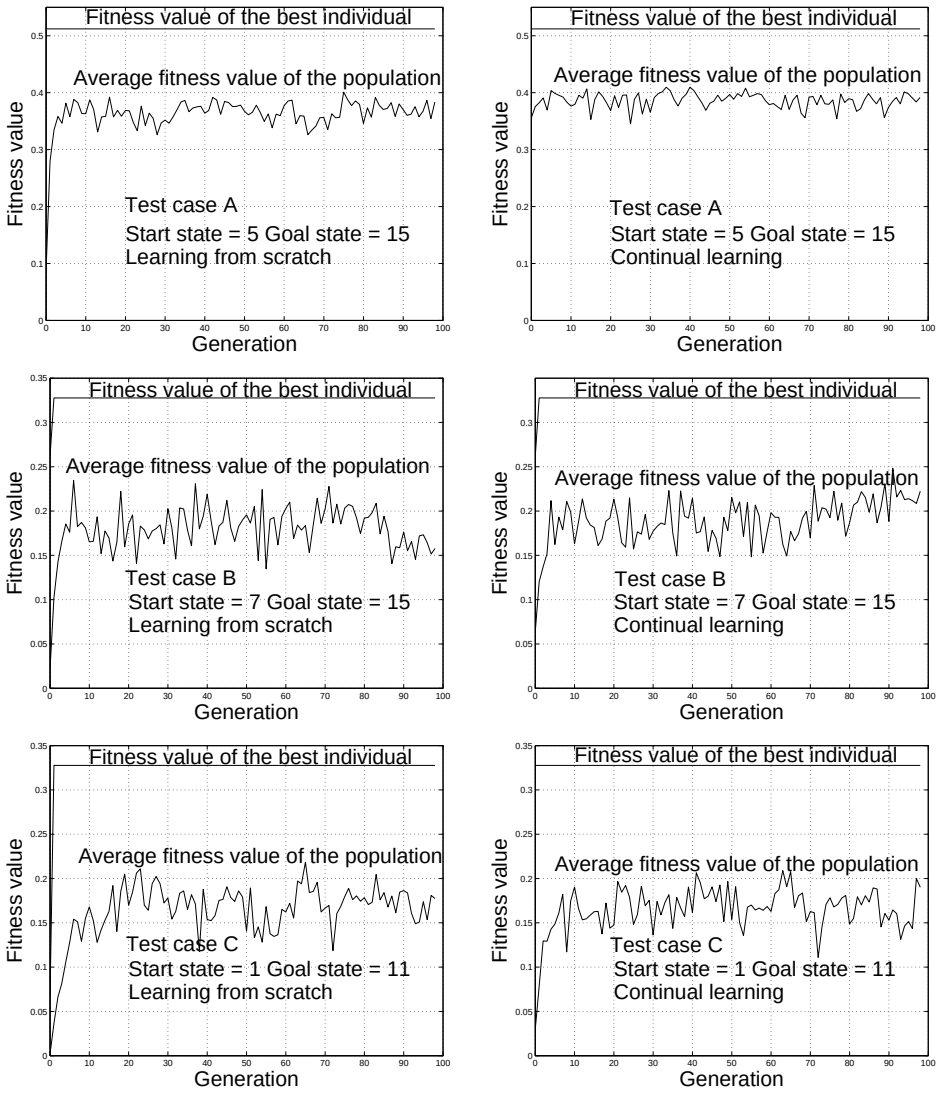


Fig. 4.11: Result obtained for agents using the Lamarckian strategy. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

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

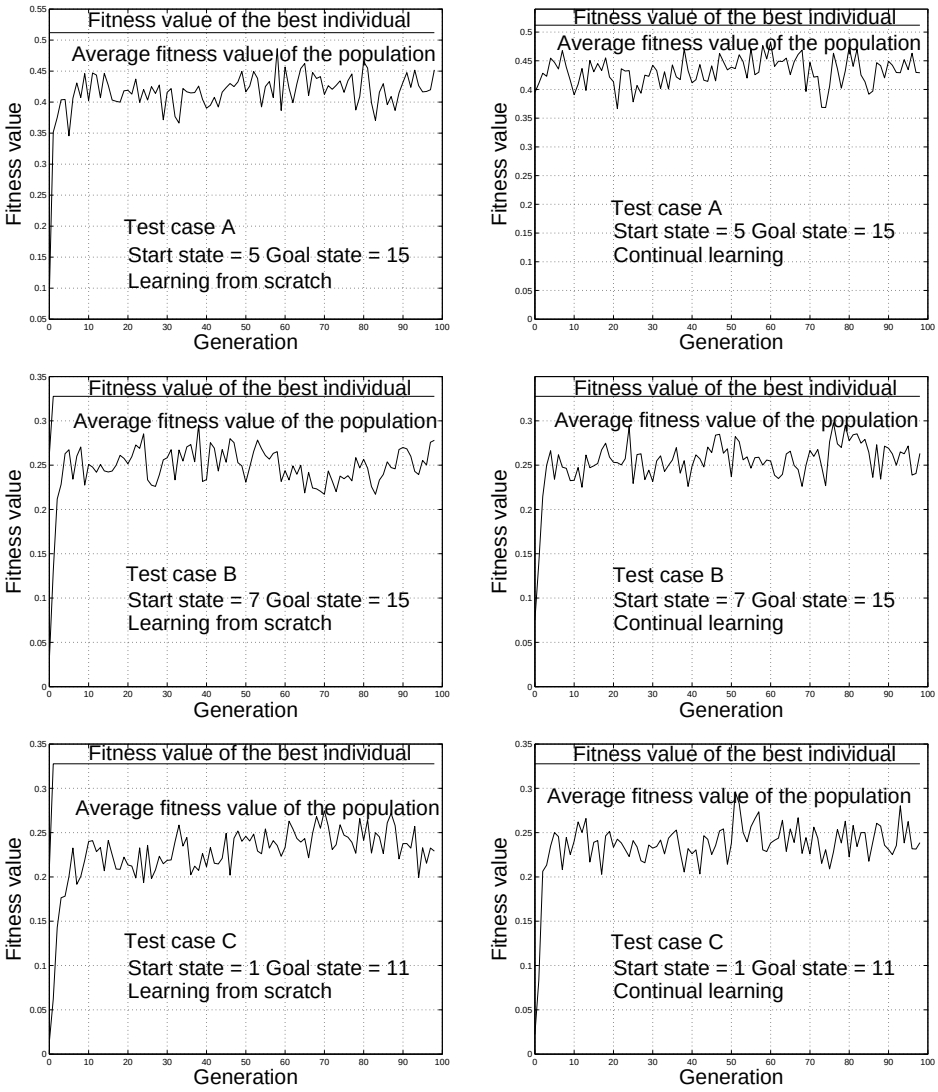


Fig. 4.12: Result obtained for agents using the Darwinian strategy. The left column shows from top to down results of learning from scratch for test case A, B, and C, respectively. The right column shows the corresponding result in continual learning.

required in continual learning is shorter than the time required in learning from scratch.

By comparing the results of agents using Q-learning and agents using a hybrid of Q-learning and GA, one can see that agents using a hybrid of Q-learning and GA required a smaller number of trials to learn the action

Learning and evolutionary methods used by agents	**# of trials in learning from scratch**			**# of trials in continual learning**		
	A	**B**	**C**	**A**	**B**	**C**
Q-Learning	40	50	40	0	14	20
Hybrid of MLP and GA	4	9	8	0	4	4
Hybrid of Q-Learning and GA (Lamarckian)	10	5	7	0	3	4
Hybrid of Q-Learning and GA (Darwinian)	5	5	6	0	3	4

Tab. 4.8: Summary of results obtained after running the experiments for test case A, B and C respectively.

values of the states of the environment in both learning from scratch and continual learning than agent using Q-learning algorithm only. This supports our argument that agents using appropriate hybridization of learning and evolutionary algorithms show better learning and adaptation capability as compared to agents using learning algorithms only.

4.9 Conclusion Drawn and Recommendation

We have shown that continual learning requires shorter learning time as compared to learning from scratch under various learning conditions. 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 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 evolution strategies) *it is possible to design better agents with better learning and adaptation capability for either lower or higher cognitive levels.*

Chapter 5

EVOLUTIONARY ACQUISITION OF NEURAL TOPOLOGIES (EANT)

A meaningful combination of the principles of neural networks, reinforcement learning and evolutionary computation is useful for designing agents that learn and adapt to their environment through interaction [56, 57, 94]. The combination results in an evolutionary reinforcement learning system where each of the components plays an important role.

Neural networks are useful for evolving the control system of an agent [56, 75, 78]. They provide a straight forward mapping between sensors and motors and this enables them to represent directly the policy (control) or the value function to be learned. Moreover, they can accommodate continuous (analog) or discrete input signals and provide either continuous or discrete motor outputs, depending on the transfer function chosen. Gradual changes to the parameters defining a neural network will often correspond to gradual changes of its behavior i.e they offer a relatively smooth search space. In addition to this, they 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. They have been used in combination with other methods in solving inherently unstable control tasks [21, 35, 51, 77, 95, 104], in learning obstacle avoidance and navigation paths [42, 50], and in representing a value function while learning to play games without human expertise [23, 98].

Reinforcement learning is useful as a type of learning where the agent is not told directly what to do but fed with a signal (reward) that measures the quality of executing an action in a given state [8, 42, 97]. The purpose of the agent is to act optimally in its environment so as to maximize its rewards. It is one form of learning through interaction. Learning through interaction underlines nearly all the principles of intelligence [78]. This property of reinforcement learning makes it important in evolutionary reinforcement learning system. In reinforcement learning, an agent tries to estimate the value function. This function shows how good it is for an agent to be in

a given state or how good it is for an agent to execute a given action in a given state. It is possible to generate the policy directly from value function. In reinforcement evolutionary learning, the policy or the value function is represented by a neural network.

Like neural networks, evolutionary algorithms are inspired from biology. Populations of organisms have been adapting to their particular environmental conditions through evolutionary selection (survival of the fittest) and variablity among them. From these principles of adaptation in nature, it is possible to derive a number of concepts and strategies for solving learning tasks and develop optimization strategies for artificial intelligent systems. An example of an optimization problem that can be solved using the principles of evolution is a model based evolutionary object recognition system [58].

There are many forms of evolutionary algorithms. The major ones are genetic algorithms [48, 103], genetic programming [67], evolution strategy [79, 87] and evolutionary programming [31]. Most of the evolutionary algorithms have the following important components: representation (definition of individuals), evaluation function (fitness function), population, parent selection mechanism, variation operators (recombination and mutation) and survivor selection mechanism [28].

Evolutionary algorithms can be considered as a kind of reinforcement learning. In evolutionary algorithms, the fitness function is a kind of a reward signal of an agent that has operated and lived in a given environment. But reinforcement learning algorithms and evolutionary algorithms have the following major differences:

1. Reinforcement learning algorithms have only one agent while evolutionary algorithms have population of agents at a time.

2. In reinforcement learning, signals (rewards) are provided after each action is executed by the agent. In evolutionary algorithms, fitness values (rewards) are provided to the agent at the end of the life of the agent or after the individual has performed or operated in the environment.

3. Reinforcement learning algorithms update the policy or value function of an agent while the agent is operating in the environment. Evolutionary algorithms, however, update the policy of an agent after the agent has lived and operated in the environment. That means, evolutionary algorithms search for optimal value functions or optimal policies directly in space of value functions or policies.

The evolutionary reinforcement learning system that combines the principles of neural networks, reinforcement learning and evolutionary algorithms is

shown in Figure 5.1. The evolutionary algorithm contains genotypes of neural networks to be evaluated in a given environment. Each neural network is evaluated and assigned a given fitness value (reward). Through genetic operators of the evolutionary algorithm, the agents are improved and evaluated in the environment. The process continues until a certain number of generations or until an agent is found that solves a given task. The neural network may represent a policy or a value function depending on the task that is going to be solved. It may even represent a regression or classification function for supervised training of neural networks with evolutionary algorithms.

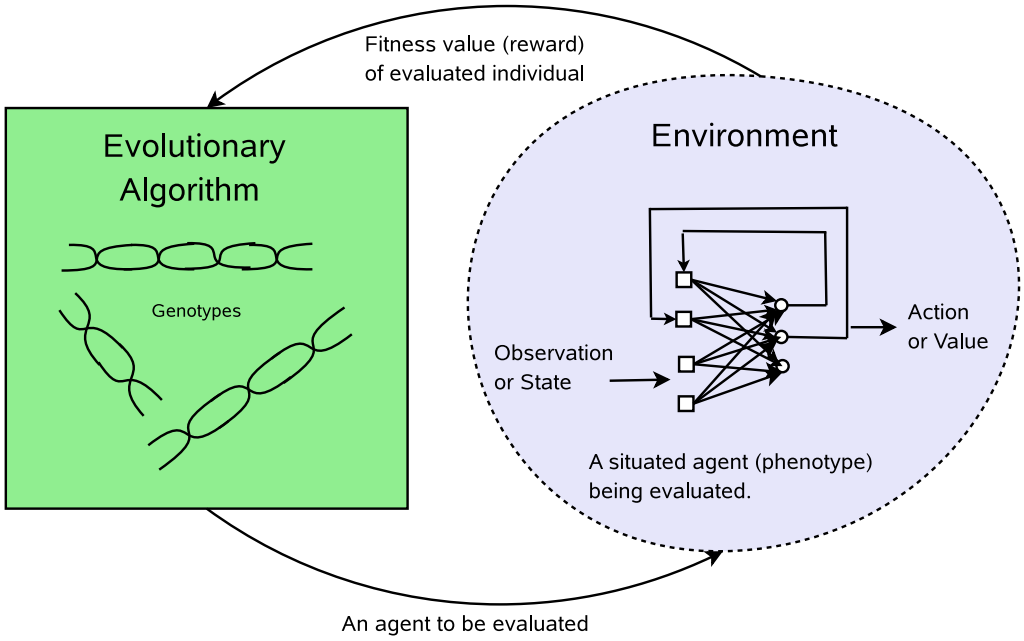


Fig. 5.1: Evolutionary reinforcement learning system. The agents, where the neural networks are embedded in, are evaluated in the environment and their fitness values are returned to the evolutionary algorithm as rewards.

5.1 Related Works

The evolution of neural networks can be divided into two major categories: the evolution of connection weights and the evolution of both the structure and connection weights. In the first category, the structure of neural networks

is fixed and is determined by the domain expert before the evolution begins. In the second category, both the structure and the connection weights are determined automatically by the evolutionary process. A detailed review of the evolution of neural networks is given by Yao [108]. Our work falls in the second category but in addition to the review of related works in the second category, we will give a review of works in the first category that are applied to reinforcement learning tasks.

5.1.1 Evolution of Connection Weights

Wieland [105] studied the evolutionary optimization of a fully connected recurrent neural networks on different pole balancing problems. He encoded the weights of the neural networks by eight bits and used genetic algorithm for optimization. The structure of the recurrent neural network is determined manually.

Evolutionary Programming (EP) for parameter optimization of feed-forward neural networks is used by Saravanan and Fogel for double pole balancing tasks [85]. The structure of the neural network especially the number of hidden units are determined a priori.

Moriarty and Miikkulainen [73] developed a method of evolving neural networks, called Symbiotic Adaptive Neuroevolution (SANE), where the system evolves population of neurons instead of population of networks. Fully connected hidden layers of networks are formed by a combination of neurons selected randomly from a population of neurons. A neuron individual receives an average fitness value of networks in which it takes part in.

Enforced Subpopulations (ESP) [34, 35, 36] is based on SANE, but it specializes neurons to specific tasks. Each non-input unit of the neural network is assigned to a separate subpopulation and a neuron is recombined with the members of its own subpopulation. Unlike SANE, the networks formed by ESP consists of a representative from each evolving specialization and this allows it to evolve recurrent networks since a neuron's behavior in a recurrent network critically depends on the neurons to which it is connected.

Floreano and Urzelai [29, 30] evolved a fully connected recurrent neural network for learning a light-switching task. The genotype is made up of genes that either code the synaptic strength of the connections, or the learning rate and learning rule that may be used in modifying the synaptic strengths of the connections while the agent is operating in the environment. In the latter case, they used different Hebbian rules and different learning rates.

Igel [51] applied a specialized evolutionary strategy called CMA-ES [44] for evolving a fixed-topology neural network. The CMA-ES uses important concepts like *derandomization* and *cumulation*. Derandomization is a deter-

ministic way of altering the mutation distribution such that the probability to reproduce steps in search space that have led to better population is increased. Moreover, the algorithm detects correlations between object variables and is invariant under orthogonal transformation of the search space. The search path of population over a number of past generations is used in order to use the information from previous generations more efficiently. In CMA-ES this is known as cumulation.

5.1.2 Evolution of Structure and Connection Weights

These methods evolve both the connection weights and the structure of the neural networks. They are divided into two major groups depending on the type of genetic encoding used. The two types of genetic encoding are the direct and indirect encoding types.

Direct Genetic Encoding

Methods that use direct encoding scheme must specify explicitly every connection and nodes that will appear in the phenotype.

Angeline et al. [2] developed a system called GNARL (GeNeralized Acquisition of Recurrent Links) that uses only structural mutation on the topology, and parametric mutations on the weights as genetic search operators. The system is based on evolutionary programming where crossover operator is not used as a search operator. The system tries to maintain the behavior of the network in order to avoid a radical jump from parent to offspring. New links are initialized with zero weight, leaving the behavior of the modified network unchanged and hidden nodes are added to the network without any incident connections. The main problem of this method is that genomes may end up in many extraneous disconnected structures that do not have any contribution to the solution.

The Neuroevolution of Augmenting Topologies (NEAT) developed by Stanley and Miikkulainen [94, 95] evolves both the structure and weights of neural networks using both crossover and mutation operators. It starts with networks of minimal structures and increases their complexity along the evolution path. Every node and connection of the phenotype is encoded by the genotype. The algorithm keeps track of the historical origin of every gene that is introduced through structural mutation. The history is used by a specially designed crossover to match up genomes encoding different network topologies, and to create a new structure that combines the overlapping parts of the two parents as well as their different parts. Structural discoveries of the evolutionary process are protected by niching (speciation). The

speciation in NEAT is achieved by explicit fitness sharing, where organisms in the same species share the fitness of their niche. Unlike GNARL, NEAT does not use self-adaptation of mutation step-sizes. Each connection weight is perturbed with a fixed probability by adding a floating number chosen from a uniform distribution of positive and negative values.

Indirect Genetic Encoding

In indirect encoding, one specifies rules that are used in constructing the phenotype. Every connection and node is not specified in the genome but can be derived from it.

Kitano's [64] grammar based encoding of neural networks use Lindenmayer systems (L-systems) [69] to describe the morphogenesis of linear and branching structures in plants. L-systems are parallel string rewriting systems that rewrite a starting string into a new string by applying a set of production rules to all symbols of the string in parallel. Sendhoff et al. [90] extended Kitano's grammar encoding with their recursive encoding of modular neural networks. Their system provides a means of initialization of the network weights. Networks are trained using the standard back-propagation and the encoding itself is variable and optimized on a larger timescale. Both Kitano and Sendhoff used their systems for evolution of feed-forward networks. In Kitano's grammar based encoding, there is no direct way of representing the connection weights of neural networks in the genome.

Gruau's Cellular Encoding (CE) method [39, 40, 41] is a language for local graph transformations that controls the division of cells which grow into artificial neural network. Through cell division, one cell called the parent cell is replaced by two cells called child cells. During division, a cell must specify how the two child cells must be linked. The genetic representations in CE are compact because genes can be reused multiple times during the development of the network and this saves space in genome since every connection and node does not need to be explictly specified in the genome. Defining a crossover operator for CE is still difficult, and it is not easy to analyze how crossover affects the sub-functions in CE encoding since they are not represented explicitly. Moreover, it suffers from the same problem as that of Kitano's grammar based encoding since there is no direct way of representing the connection weights of neural networks in the genome. Luke and Spector [71] proposed *edge encoding* as a solution to these problems. Unlike CE, the grammar trees are traversed in depth-first rather than breadth-first order, and networks are grown by modifying the edges in a graph rather than the nodes. Currently, there is no experimental comparison between CE and *edge encoding* and therefore nothing can be said on suitability and efficiency of

edge encoding over CE in evolving neural networks.

Nolfi and Parisi [76] modeled biological development at the chemical level using reaction-diffusion model. Diffusion is modeled at the level of axon growth and reaction is modeled as the interaction between axon and cell bodies. This method utilizes growth to create connectivity without explictly describing each connection in the phenotype. Axon which do not hit other cells during development are pruned. The complexity represented by the genome is limited since every neuron is directly specified in the genome.

Vaario et al. [102] have developed a biologically inspired neural growth based on diffusion field modeling combined with genetic factors for controlling the growth of the network. The neural structures are grown in either a two-dimensional or three-dimensional grid resulting in a two-dimensional or three-dimensional tree-based neural structure, respectively. One weak point of the method is that it can not generate networks with recurrent connections or networks with connections between neurons on different branches of the resulting tree structure.

Other works in indirect encoding simulated the genetic regulatory networks (GRN) in biology where genes produce signals that either activate or inhibit other genes in the genome. The interaction of all genes forms a network that produces a phenotype. Typical works that used GRN include works of Dellaert and Beer [27] which use boolean functions called *operons* to implement GRN, Morphogenesis of Jakobi [54], Artificial Ontogeny of Bongard and Pfeifer [14], Computational Embryogeny of Bentley and Kumar [11]. The main problem of these methods like other indirect encoding methods is that they do not search for the optimal solution starting from some initial structure and increasing its complexity along the evolution.

5.2 Contributions of the Work

The method presented in this work is closely related to the already mentioned works of Angeline et al. [2] and to the works of Stanley and Miikkulainen [95, 94]. It is related to the works of Angeline et al. in that the method uses parametric mutation that is based on evolution strategies or evolutionary programming with adaptive step sizes for optimization of the weights of the neural networks. Complexification of structures along the evolution path starting from a minimum structure makes it related to the works of Stanley and Miikkulainen. But it has the following important features which makes it different from the earlier works:

1. It introduces a compact genetic encoding of a neural network that enables one to evaluate the neural network without decoding it. The

topology of the network is implicitly encoded in the order of genes in the linear genome.

2. For evolving the structures and weights of neural networks, a nature inspired meta-level evolutionary process is used, where exploration of structures is executed at a larger timescale and exploitation of existing structures is done at smaller timescale.

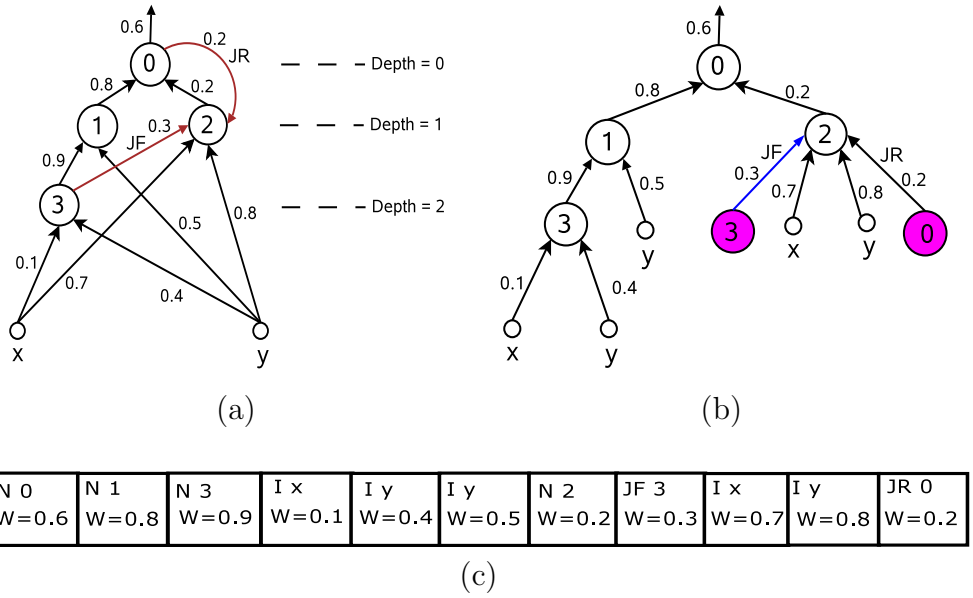
5.3 The Proposed Method

In this section, we present our evolutionary reinforcement learning system, called Evolutionary Acquisition of Neural Topologies (EANT). It is a novel method that is suitable for learning and adaptation to the environment through interaction. The system evolves both the structures and weights of neural networks. With respect to the goal of self-organizing learning machines which start from minimal specification and rise to great sophistication, EANT starts with neural networks of minimal structures, and increases their complexity along the evolution path.

5.3.1 Genetic Encoding

A flexible encoding method enables one to design an efficient evolutionary method that can evolve both the structures and weights of neural networks. The genome in EANT is designed by taking this fact into consideration. A genome in EANT is a linear genome consisting of genes (nodes) that can take different forms (alleles). The forms that can be taken by a gene can either be a neuron, or an input to the neural network, or a jumper connecting two neurons. The jumper genes are introduced by structural mutation along the evolution path. A jumper gene can either encode a forward or a recurrent connection. A jumper gene encoding a forward connection represents a connection starting from a neuron at a higher depth and ending at a neuron at a lower depth. The depth of a neuron node in a linear genome is the minimal number of neuron nodes that must be traversed to get from the output neuron to the neuron node, where the output neuron and the neuron node lie within the same sub-network that starts from the output neuron. On the other hand, a jumper gene encoding a recurrent connection represents a connection between neurons having the same depth, or a connection starting from a neuron at a lower depth and ending at a neuron at a higher depth. Every node in a linear genome has a weight associated with it. The weight encodes the synaptic strength of the connection between the node coded by the gene and the neuron to which it is connected. Moreover, every node can

save the results of its current computation. This is useful since the results of signals at recurrent links are available at the next time step. In addition to the synaptic weight, a neuron node has a unique global identification number and number of input connections associated with it. A jumper node has also additionally a global identification number, which shows the neuron to which it is connected. An example of a linear genome encoding a neural network is shown in Figure 5.2.



N 0 W=0.6	N 1 W=0.8	N 3 W=0.9	I x W=0.1	I y W=0.4	I y W=0.5	N 2 W=0.2	JF 3 W=0.3	I x W=0.7	I y W=0.8	JR 0 W=0.2

(c)

Fig. 5.2: An example of encoding a neural network using a linear genome. (a) The neural network to be encoded. It has one forward and one recurrent jumper connection. (b) The neural network interpreted as a tree structure, where the jumper connections are considered as terminals. (c) The linear genome encoding the neural network shown in (a). In the linear genome, `N` stands for a neuron, `I` for an input to the neural network, `JF` for a forward jumper connection, and `JR` for a recurrent jumper connection. The numbers beside `N` represent the global identification numbers of the neurons, and `x` or `y` represent the inputs coded by the input gene (node).

The linear genome can be interpreted as a tree based program if we consider all the inputs to the network and all jumper connections as terminals. Terminals are sources of signals either from the inputs or from other parts of the neural network. On the other hand, neurons are processing units that

map the signals at their inputs to signals at their outputs. Terminals are analogous to the terminal set and neurons are analogous to the function set in a standard GP program [6, 67]. The linear genome is a prefix ordering of genes (nodes) where the ordering implicitly represents the topology of the neural network encoded by it. The term prefix ordering stands for the fact that in the ordering the neuron nodes (operators) come before the inputs and jumper connections (operands). Figure 5.3 shows the equivalence between a neural network, a linear genome representing it, and a tree-based program representing the neural network. Starting from a neural network it is possible to generate a linear genome that encodes it or a tree-based program representing it. The converse is also true; starting from a linear genome or a tree-based program, it is possible to generate the neural network.

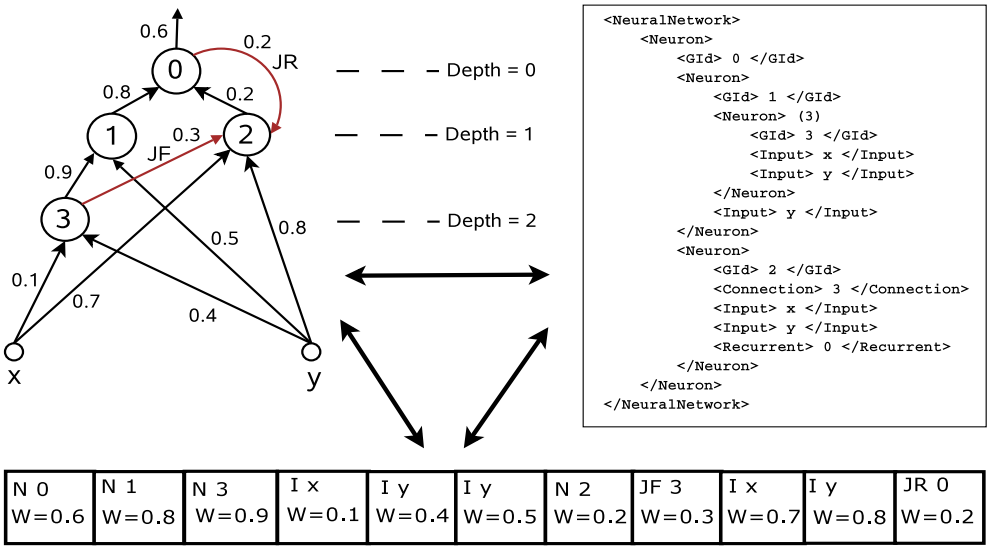


Fig. 5.3: The linear genome is equivalent to the neural network it encodes or a tree based program representing the neural network. One can generate the tree based program or the linear genome starting from the neural network or vise-versa. The tree-based program is coded in XML like commands. The commands `<NeuralNetwork>` and `</NeuralNetwork>`, `<Neuron>` and `</Neuron>`, `<Input>` and `</Input>`, `<Connection>` and `</Connection>`, `<Recurrent>` and `</Recurrent>`, and `<GId>` and `</GId>` stand for the start and end of a program representing a neural network, a neuron, an input, a forward jumper connection, a recurrent jumper connection, and global identification number, respectively.

The linear genome has some interesting properties that makes it useful for evolution of the structure of neural networks. Assume that integer values are assigned to the nodes of a linear genome encoding a neural network such that the integer values show the difference between the number of outputs of the nodes and the number of arguments of the nodes (inputs to the nodes). Note that every node in the linear genome has only *one* output. If a node is an input to the neural network, the integer assigned to it is 1 since an input to a neural network has only one output and no arguments (inputs) at all. An integer value of 1 is also assigned to the forward and recurrent jumper nodes since they are sources of signals from other neurons in the neural network encoded by the linear genome. A neuron node will take an integer value which is the same as one minus the number of inputs to the neuron. In EANT, since there is no neuron without an input, the maximum value of an integer assigned to a neuron node is zero. This is true for all neurons with only one input. One of the important properties of a linear genome is that the sum of the integer values assigned to each of the nodes in a linear genome encoding a neural network is the same as the number of outputs of the neural network.

After assigning the integer values to the nodes of the linear genome, it is possible to detect a sub-linear genome (sub-network) of a linear genome. A sub-linear genome (sub-network) in EANT is defined as a collection of nodes starting from a neuron node and ending at a node where the sum of integer values assigned to the nodes between and including the start neuron node and the end node is *one*. An example is shown in Figure 5.4.

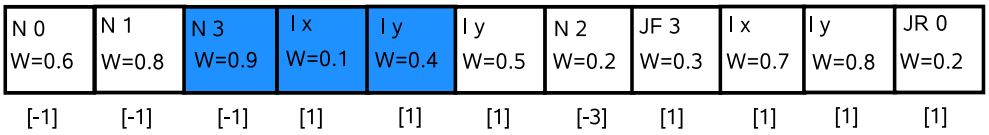


Fig. 5.4: An example of the use of assigning integer values to the nodes of the linear genome. The linear genome encodes the neural network shown in Figure 5.2. The numbers in the square brackets below the linear genome show the integer values assigned to the nodes of the linear genome. Note that the sum of the integer values is *one* showing that the neural network encoded by the linear genome has only *one* output. The shaded nodes in the linear genome form a sub-network. Note also that the sum of the integer values assigned to a sub-network is always *one*.

The linear genome is *complete* in that it can be used to represent any type

of neural network. It is also a *compact* encoding of neural networks since the length of the linear genome is the same as the number of synaptic weights in the neural network. Moreover, the encoding scheme used is *closed*. An encoding scheme is said to be closed if all genotypes produced are mapped into a valid set of phenotype networks [5, 55]. It is closed under structural mutation operator since every new linear genome produced by structural mutation is mapped into a valid phenotype network. See Section 5.3.4. It is also closed under a special crossover where the crossover operator exploits the fact that structures which originate from the initial minimal structures have some parts in common. By aligning the common parts of two randomly selected structures, it is possible to generate a third structure which contains the common and disjoint parts of the two mother structures. The resulting structure formed in this way maps to a valid phenotype network. This type of crossover is introduced and used by Stanley [94]. An example is shown in Figure 5.5. The number of inputs to a neuron node which is common to the parent structures is updated using

$$n(s_1 \times s_2) = n(s_1) + n(s_2) - n(s_1 \cap s_2), \tag{5.1}$$

where $n(s_1 \times s_2)$ is the number of inputs to the neuron node in the offspring, $n(s_1 \cap s_2)$ is the number of input nodes to the neuron node which are common to both structures, and $n(s_1)$ and $n(s_2)$ are the number of input nodes to the neuron node in the parent structure s_1 and s_2, respectively.

5.3.2 Evaluating a Linear Genome

There are two methods of evaluating a linear genome. In the first method, one decodes the linear genome into a neural network that it represents and then evaluates the neural network directly. In other words, in this method there is a physical difference between the genotype (the linear genome) and the phenotype (the neural network encoded by the linear genome). This method is especially useful if one wants to evaluate the network using some type of parallel computation. In the second method, it is not necessary to decode the linear genome into the neural network but one can use the linear genome directly to evaluate the neural network represented by the genome. The second method emphasizes the fact that it is not always necessary to create a separate phenotype structure from genotype by some sort of ontological process [6]. In other words, it is not always necessary to decode the linear genome into a neural network for the purpose of evaluating the network encoded by it.

For the purpose of evaluating or computing the output of the neural

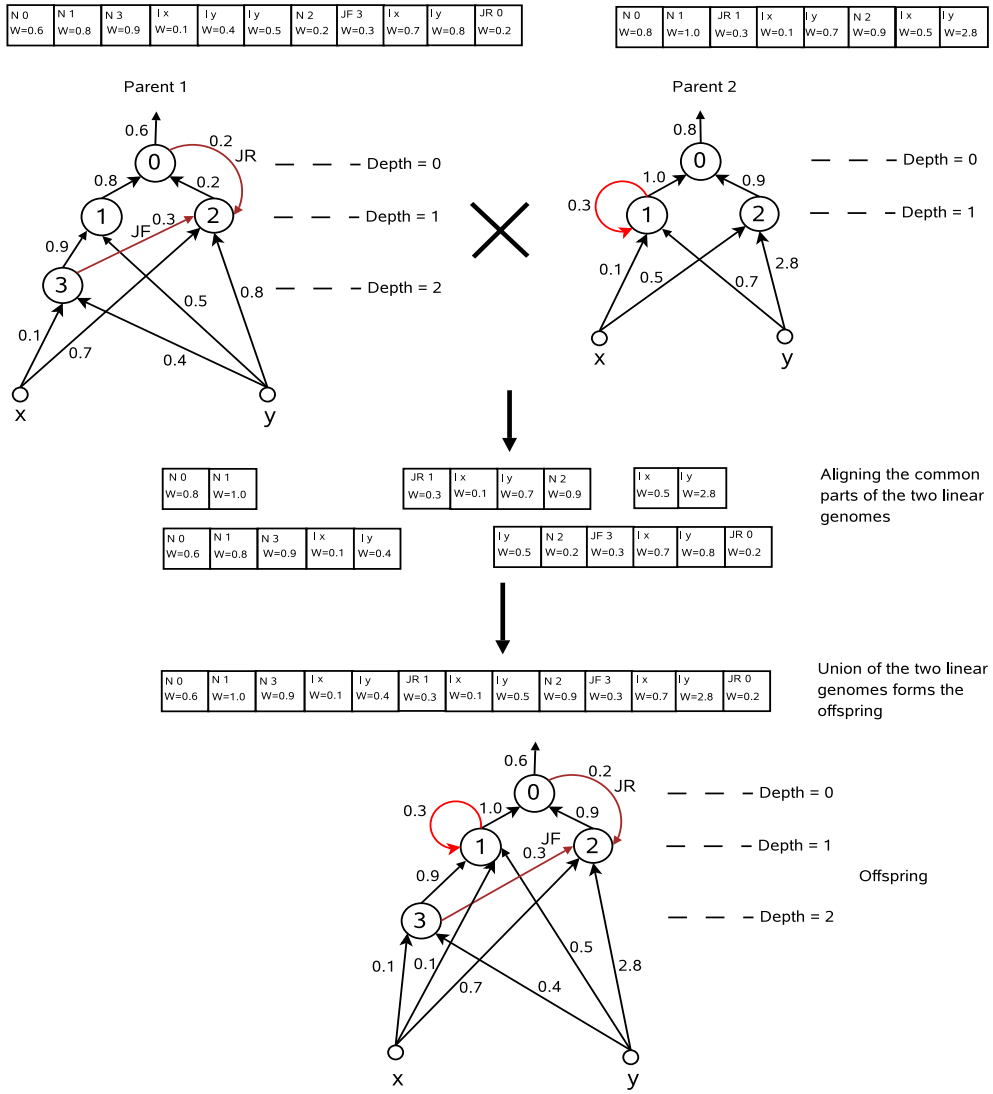


Fig. 5.5: Performing crossover between two linear genomes. The genetic encoding is closed under this type of crossover operator since the resulting linear genome maps to a valid phenotype network. The weights of the nodes of the resulting linear genomes are inherited randomly from both parents.

network without decoding the genome, we use a *first in last out* stack and the following rules:

1. Start from the right most node of the linear genome.

2. Move from right to left in computing the output of the neural network. This is the same as incrementing a program counter while running a program.

3. If the current node is an input node, push its current value and the weight associated with it onto the stack.

   If the current node is a neuron node, pop n values with their associated weights from the stack and push the result of computation with its associated weight onto the stack, where n is the number of inputs of the neuron being evaluated. The output of a neuron node is computed using

   $$O = g\left(\sum_{i=1}^{n} w_i\, a_i\right), \tag{5.2}$$

   where O is the result of computation for the current neuron node, a_i and w_i are the popped values and their associated weights. $g(.)$ is the activation function of the neuron node.

   If the current node is a recurrent jumper node, get the *last value* of the neuron node whose global identification number is the same as the global identification number of the recurrent jumper node. Then push the value obtained with the weight associated with jumper node onto the stack.

   If the current node is a forward jumper node, first copy the sub-linear genome (sub-network) starting from a neuron whose global identification number is the same as the global identification number of the forward jumper node. Then compute the response of the sub-linear genome in the same way as that of the linear genome. Finally, push the result of computation with the weight associated with forward jumper node onto the stack. This is analogous to calling a function in a program or jumping to an interrupt service routine.

4. After traversing the genome from right to left completely, pop the resulting values from the stack. The number of the resulting values is the same as the number of outputs of the neural network coded by the linear genome.

Figure 5.6 shows an example of evaluating a linear genome encoding the neural network shown in Figure 5.2. As can be seen from the figure, one does not need to decode the neural network in order to evaluate it.

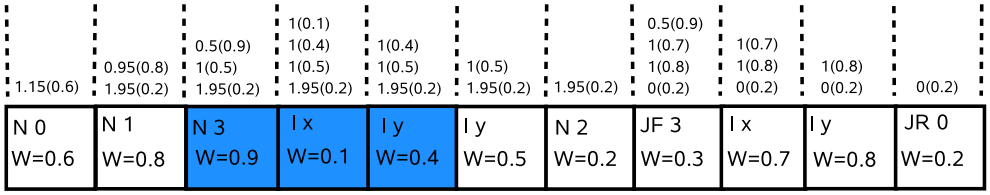


Fig. 5.6: An example of evaluating a linear genome without decoding the neural network encoded by it. The linear genome encodes the neural network shown in Figure 5.2. For this example, the current values of the inputs to the neural network, x and y, are both set to 1. In the example, all neurons have a linear activation function of the form $z = a$, where a is the weighted linear combination of the inputs to a neuron. The overlapped numbers above the linear genome show the status of the stack after computing the output of a node. The numbers in brackets are the weights associated with the nodes.

The evaluation of a linear genome discussed above is equivalent to the evaluation of a decoded neural network represented by the genome, where the activation of a neuron of the network is given by

$$a_i(t) = g\left(\sum_{j=1}^{n_f} w_{ij}a_j(t) + \sum_{k=n_f+1}^{n} w_{ik}a_k(t-1)\right). \tag{5.3}$$

In the equation, g is the activation function of the neuron and n is the number of input connections to the neuron. The number of forward connections and the number of recurrent connections to the neuron are n_f and $n - n_f$, respectively.

5.3.3 Generating the Initial Linear Genome

The first step in generating the initial genome is to determine the number of outputs and number of inputs of the neural network required for a given task. The initial linear genome contains only output neuron nodes and input nodes. The forward and recurrent connection nodes are introduced by the structural mutation operator and added to the linear genome along the evolution path. Two methods are used in the initialization of the initial genome. These are the *grow* and *full* methods [6, 68].

Grow Method

For a given maximum depth, the grow method produces linear genomes encoding neural networks of irregular shape because a node is assigned to a randomly generated neuron node having a random number of inputs or to a randomly selected input node. Figure 5.7 shows an example of a linear genome generated using the grow method. The neural network encoded by the linear genome has a neuron with repeated inputs.

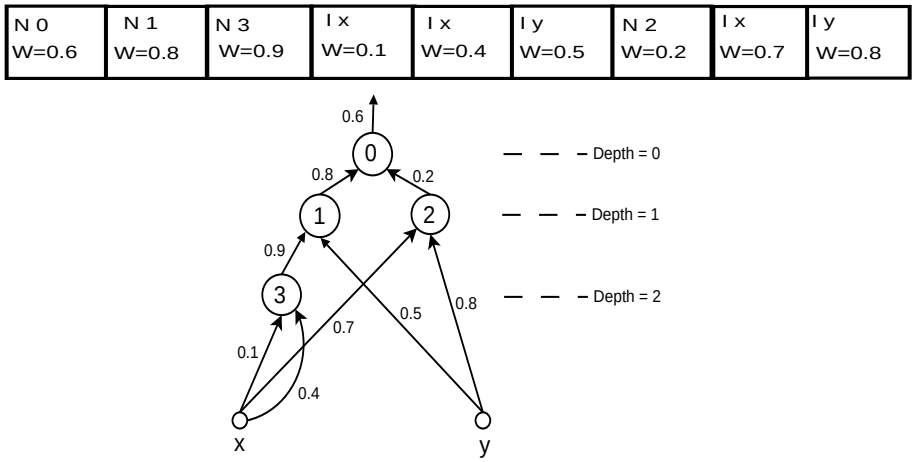


Fig. 5.7: An example of a linear genome generated by using the grow method and the neural network encoded by it. Note that the linear genome must be edited since the neural network encoded by it has a neuron with repeated inputs.

Full Method

This method adds to the linear genome randomly generated neurons connected to all inputs until a node is at the maximum depth and then adds only random input nodes. This results in neural networks with symmetric structures where every branch of a tree-based program equivalent of the linear genome goes to the full maximum depth. In this method, except neurons at the maximum depth, all neurons are connected to a fixed number of neuron nodes. Figure 5.8 shows an example of a linear genome generated with full method.

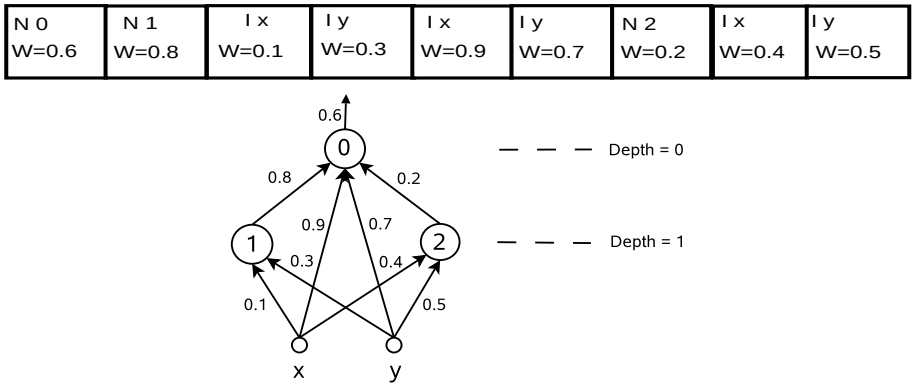


Fig. 5.8: An example of a linear genome generated by using the full method and the neural network encoded by it.

Editing a Linear Genome

An initially generated genome with either the grow or full method should be edited so that it has no neuron nodes which have repeated inputs. It is obvious that repeated inputs can be represented by a single input connection to the neuron. Editing a linear genome avoids the unnecessary addition of parameters in the weight space due to repeated inputs connected to a neuron node, and hence reducing the number of evaluations necessary during optimization of the weights of the neural network. Figure 5.9 shows an example of editing a linear genome.

5.3.4 Variation Operator: Structural Mutation

The structural mutation used by EANT adds or removes a forward or a recurrent jumper connection between neurons, or adds a new sub-network to the linear genome. The initial weight of a newly added jumper connection or the initial weight of the first node of a newly added sub-network is set to zero so as not to disturb the performance or behavior of the neural network. The structural mutation operator does not remove a sub-network because removing a sub-network results in a removal of all jumper connections that are coming to or going out of the sub-network. This would cause a tremendous loss of the performance of the neural network.

The structural mutation operates only on neuron nodes of a linear genome. Figure 5.10 shows an example where the structural property of a neuron node `N3` is changed through the structural mutation. The neuron node losses a connection to an input `x` and gains a self-recurrent connec-

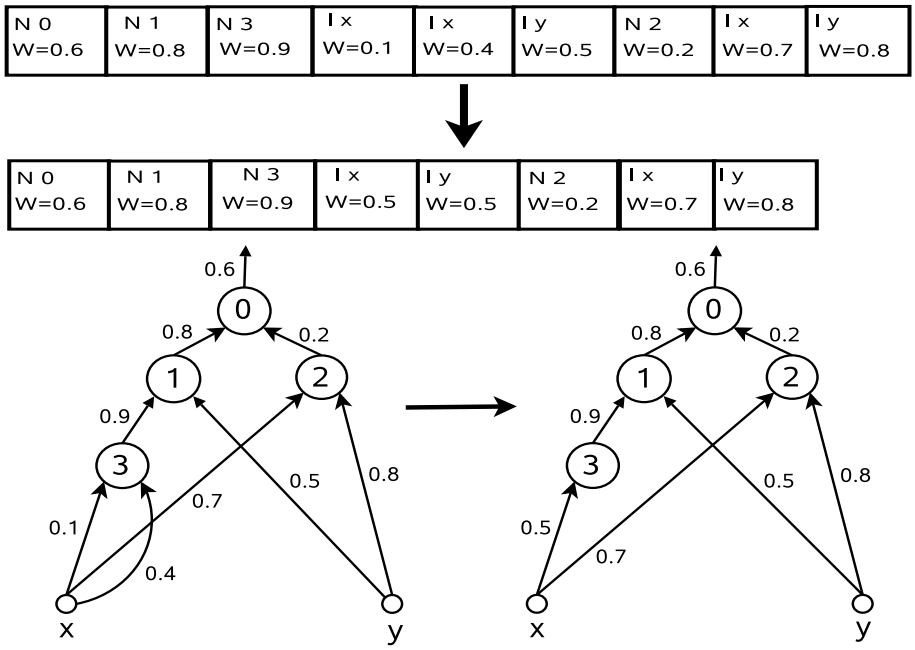


Fig. 5.9: An example of editing a linear genome. The editing replaces repeating inputs with non-repeating ones.

tion. In applying the structural mutation, each neuron node is tested if it is going to be mutated or not by drawing a random number from a uniform distribution between 0 and 1. If the currently drawn random number is less than the structural mutation probability p_m, which is usually set between 0.05 and 0.1, the neuron node will be mutated. Once it is known that the neuron node is going to be mutated, a random number is again drawn from a uniform distribution between 0 and 1 for determining the kind of structural mutation to execute. Adding connections, adding sub-networks, and removing connections are all given equal probabilities of execution.

5.3.5 Variation Operator: Parametric Mutation

Parametric mutation is used for optimization of the weights of a given structure. It is accomplished by perturbing the synaptic weights of the networks according to the uncorrelated mutation in evolution strategy or evolutionary programming [28, 31, 87]. In addition to the associated weight, each node in a linear genome has an associated mutation step size or learning rate. Figure 5.11 shows a linear genome with n nodes where every node has a learning rate associated with it.

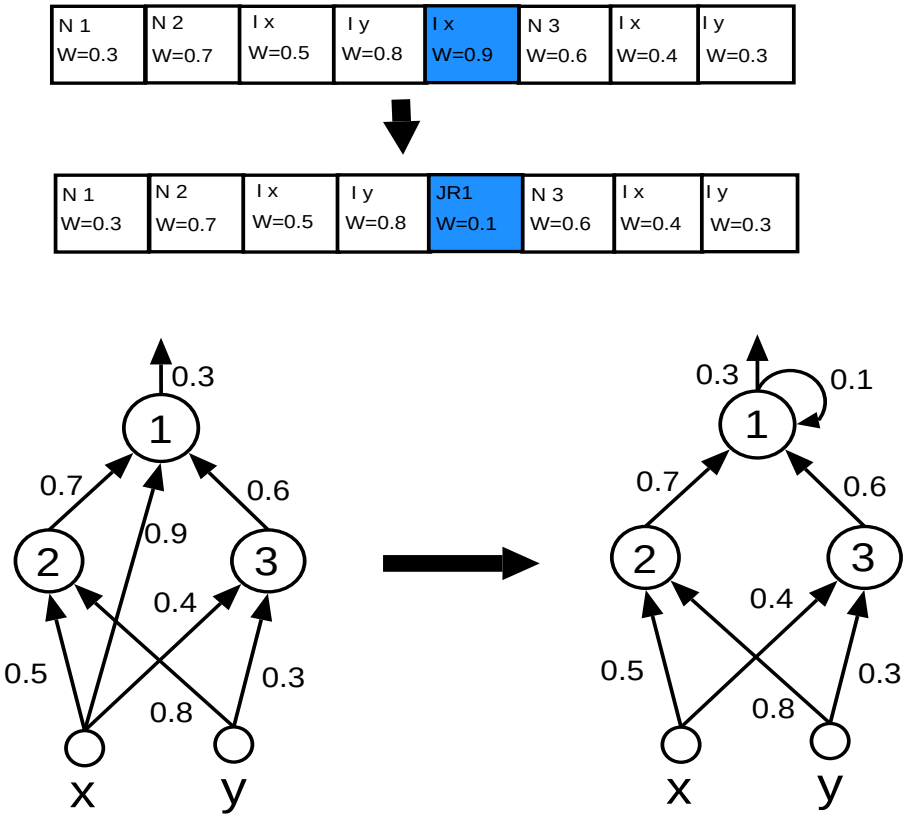


Fig. 5.10: An example of structural mutation. Note that the structural mutation deleted the input connection to `N1` and added a self-recurrent connection to it.

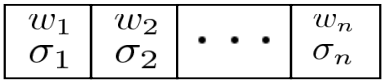


Fig. 5.11: Every node of a linear genome has in addition to weight an associated learning rate.

The mutation mechanism is specified as follows:

$$\sigma_i' = \sigma_i \, e^{\tau' \, N(0,1) + \tau \, N_i(0,1)}, \tag{5.4}$$

$$w_i' = w_i + \sigma_i \, N_i \, (0,1) \, , \tag{5.5}$$

where $\tau' = 1/\sqrt{2n}$ and $\tau = 1/\sqrt{2\sqrt{n}}$, and $N\,(0,1)$ is a random number drawn

from a Gaussian distribution of zero mean and unity standard deviation. A boundary rule given by the following equation is used to force learning rates not to be smaller than a threshold value:

$$\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0. \tag{5.6}$$

The main advantages of using the parametric mutations of synaptic weights of the neural networks in the style of evolution strategies or evolutionary programming are:

1. Both evolution strategy and evolutionary programming perform search in the space of networks. Offspring created by mutation remain within a locus of similarity to their parents [2].

2. Self-adaptation of mutation step sizes of learning rates is inherent in both evolution strategy and evolutionary programming [4, 31, 87].

5.3.6 Exploitation and Exploration of Structures

The algorithm starts with networks of minimal structures whose initial complexity is specified by the domain expert through the maximum depth that can be assumed by the initial structures. The initial structures are generated either with the grow or full method.

The system starts with exploitation of structures that are already in the system. By exploitation, we mean the optimization of the weights of the existing structures. At the beginning of the exploitation of structures, each of the existing structures is parametrically perturbed to form a population of μ individuals. Then each of the individuals is evaluated to determine its fitness value. After that, the standard survivor selection, the truncation or (μ, λ) selection [88], is used for generating the individuals of the next generation. In truncation selection, μ parents are allowed to breed λ offspring, out of which the μ best are used as parents for the next generation. The (μ, λ) selection does not depend on the absolute fitness values of individuals in the population. The first μ best individuals remain best, regardless of the absolute fitness differences between individuals. The process of generating new individuals through parametric mutation and using the (μ, λ) selection continues for a certain number of generations N. This is an evolutionary process that occurs at smaller timescale for optimization of the weights of a particular structure. The number of evaluations that is necessary per structure is μN. An example of the exploitation process is shown in Figure 5.12.

Exploration of structures is accomplished by structural mutation which is performed at larger timescale. It is used to create new species or introduce

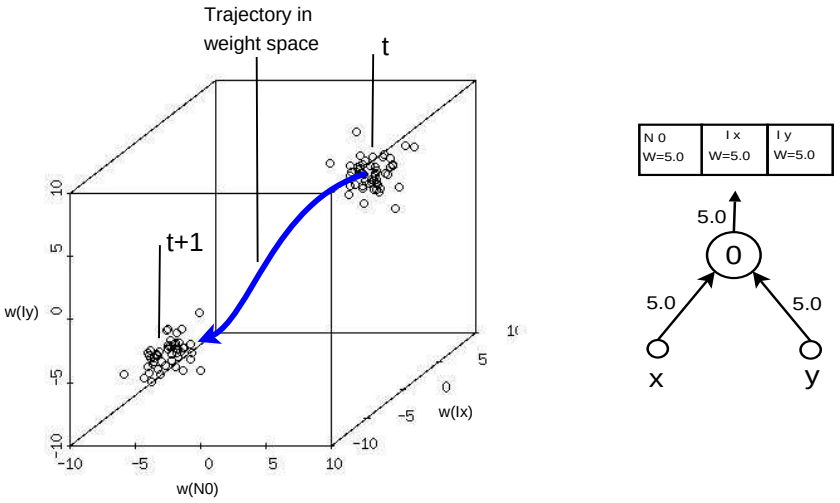


Fig. 5.12: The weight trajectory of the linear genome shown at the right side while being exploited. The quantities t and $t+1$ are time units with respect to the larger timescale. The weights of the existing structures are optimized between two consecutive time units with respect to the larger timescale. The point clouds at t and $t+1$ show populations of individuals from the same species.

new structures. From each of the existing structures, a new structure is formed and added to the existing ones. The weights of the newly acquired structural parts of the new structure are initialized to zero so as not to form (get) a new structure whose fitness value is less than its parent. This type of initialization scheme for newly acquired structures is also used by Angeline et al. [2].

The structural selection operator begins by sorting the exploited structures in descending order according to their fitness value. Then the first half of the population are selected. Young structures which are less than M generations old with respect to the larger time scale and which are not selected are carried on along the evolution regardless of the results of the selection operator. This will give them time to optimize their newly acquired structures before they compete with other individuals globally. This way it is possible to maintain the new structural discoveries of the evolution before they get extinct pretty much earlier. The number of structures in any given generation is not allowed to be larger than some pre-specified number. The

limit will keep the number of structures being entertained not to explode as the evolution proceeds.

Figure 5.13 summarizes the evolutionary process of EANT at larger time scale. As can be seen in the diagram, the evolutionary process continues until a structure is found that solves a given task. The flow diagram reflects the philosophy behind EANT in that different structures represent different species and all species compete for resources in an environment in which they live and operate. One can identify two types of competitions. The first is the competition within a species and the other is the competition between species. Competition within a species occurs between individuals having the same structure while optimizing the weights of a structure. Species which are strong enough survive and continue to live while others get extinct.

The main search operators at larger timescale are the structural mutation and structural crossover. The structural operator used in EANT exploits the fact that structures (species) which originate from the initial structure (species) have some genetic material in common. By aligning the common parts of two randomly selected species, it is possible to generate a third species which contains the common and disjoint parts of the two mother species. Structural mutation operates on a single species and creates a new species by changing the structure of the mother species. At smaller timescale, parametric mutation and recombination between individuals of the same species are used as search operators.

New structures are introduced through structural mutation and those structures that are better according to fitness evaluations survive and continue to exist in the population. Since sub-networks that are introduced by structural mutation are not removed, there is a gradual increase in the complexity of the structures along the evolution. This allows EANT to search for a solution starting from a minimum structural complexity specified by the domain expert. The search stops when a structure with the necessary minimal structure that solves a given task is obtained. In EANT complexification is an emergent property that depends on the task to be solved.

5.4 Experimental Evaluation

In this section the performance of EANT is examined. In the first experiment, EANT's ability of evolving the necessary minimal structure for a given task is considered. The standard XOR problem is used for this case. In the second experiment, we consider the problem of learning to move forward using a single legged robotic insect. The third experiment discusses the efficiency of EANT on the standard benchmark problem of balancing two poles attached

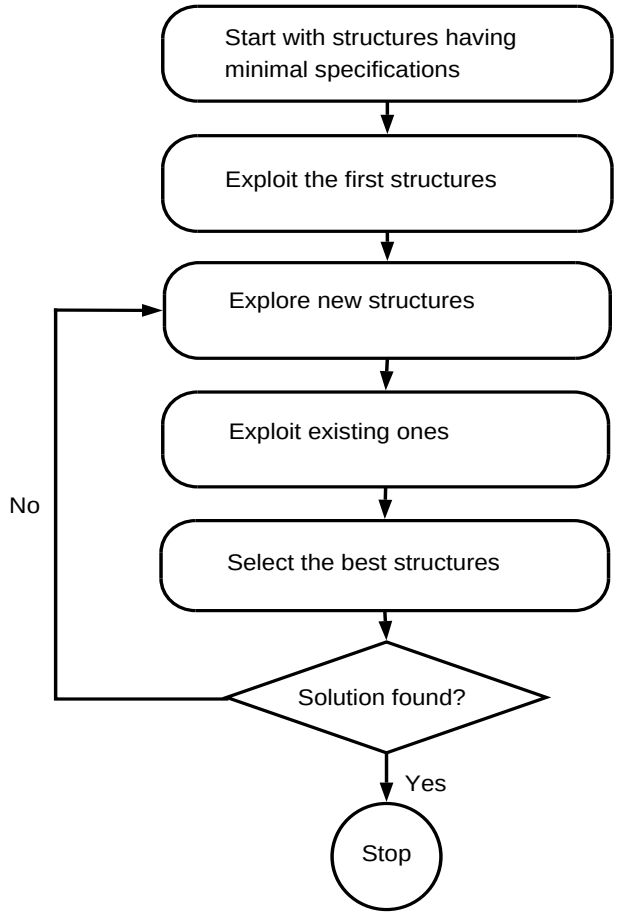


Fig. 5.13: EANT's evolutionary process at larger timescale.

to a moving cart. Comparison with other algorithms tested on the same problem is also given.

5.4.1 XOR Problem

The exclusive-OR (XOR) problem is a simple example of a data set which is not linearly separable [13]. It consists of four inputs which are divided into two classes. An example of exclusive-OR (XOR) is shown in Figure 5.14. The inputs $(0, 0)$ and $(1, 1)$ belong to class C_1, while the inputs $(0, 1)$ and $(1, 0)$ belong to C_2.

At least one hidden node is required to solve the problem since there is no linear decision boundary which can classify all four points correctly. That means the minimal neural structure for solving the XOR problem has at least

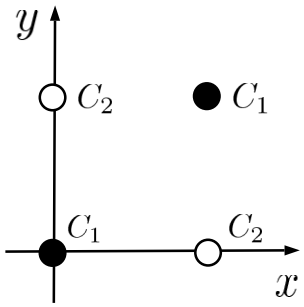


Fig. 5.14: The XOR problem. It is a simple example of a problem which is not linearly separable.

one hidden node. The aim of this experiment is to prove if EANT is able to find the necessary minimal neural structure required to solve the XOR problem starting from a neural structure having only one output neuron.

The experiment is run for 100 times and EANT is able to find networks having on the average 1.52 hidden nodes, and it takes the algorithm 1234 network evaluations to get a solution. Moreover, EANT has found a solution all the time. From the results of the experiment, one can say that EANT is consistent in finding the minimal neural structure required to solve the XOR problem. Figure 5.15 shows a sample evolutionary run in solving the XOR problem. If one decodes the best structure of the last generation, one obtains a neural network with exactly one hidden neuron.

The problem has been used to measure the performance of several other algorithms that evolve both the architecture and weights of the neural networks [26, 64, 94]. The NeuroEvolution of Augmenting Topology developed by Stanley [94] found the solution to the XOR problem after 4755 network evaluations and on the average found a solution network that has 2.35 hidden nodes. It is clear to see that our algorithm performs better in this simple problem. However, the XOR problem is such a simple task that it is not a good benchmark for measuring the performance of an algorithm.

5.4.2 Crawling Robotic Insect

The crawling robotic insect introduced by Kimura and Kobayashi [63] is used for this experiment. It is used for reinforcement learning task where the agents learn to move forward as fast as possible through interaction with the environment. The crawling robotic insect has one arm having two joints where the joints are controlled by two servo motors. It has also a touch sensor which detects whether the tip of the arm is touching the ground or

```
Generation 0
Species 0 → N1:3 I1 I0 I2

Generation 1
Species 0 → N1:4 JR1 I1 I0 I2
Species 1 → N1:3 I1 I0 I2

Generation 2
Species 0 → N1:4 N3:3 I2 I0 I1 I1 I0 I2
Species 1 → N1:4 JR1 I1 I0 I2
Species 2 → N1:5 N2:3 I0 I1 I2 JR1 I1 I0 I2
Species 3 → N1:3 I1 I0 I2
```

Fig. 5.15: A sample evolutionary run for the XOR problem. The species are sorted in descending order according to their fitness values. The first four best solutions are shown for each generations. For a neuron node, the number after the colon shows the number of input connections to it. The input node `I0` is connected to a constant bias.

not. The schematic diagram of the robot is shown in Figure 5.16.

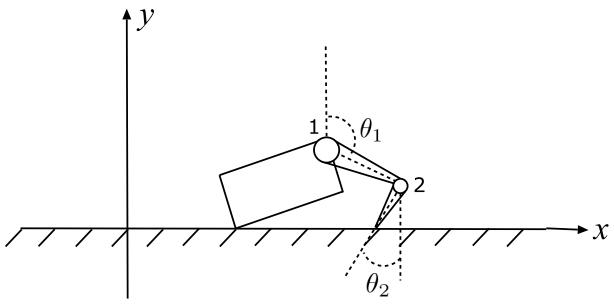


Fig. 5.16: The crawling robotic insect. The robot has one arm with two joints and a touch sensor for detecting whether the tip of the arm is touching the ground or not.

The robot has bounded continuous and discrete state variables. The continuous state variables are the joint angles and the discrete state variable is the state of the touch sensor. The controller observes the joint angles

and the state of the touch sensor. Depending on the state it perceives, the controller is expected to change the angles of the joints appropriately so that the robot can move forward as fast as possible. The first joint angle θ_1 is bounded between 55° and 94°, and the second joint angle θ_2 lies in the range $[-34^\circ, 135^\circ]$. For both of the joints, the angles are measured from the vertical as shown in Figure 5.16. The angle ranges are chosen so that they are equivalent to the angle ranges chosen by Kimura and Kobayashi. They measured the first joint angle from the horizontal and the second joint angle from the first link. The touch sensor ϕ takes the value 0 for non-touch state and 1 for touch state.

Let the coordinates of the first and the second joints be (x_0, y_0) and (x_1, y_1), respectively and let the coordinate of the tip of the arm be (x_2, y_2). The state of the robot at each time step $t = 0, 1, \ldots$ is given by $s_t = (x_0, y_0, x_2, y_2, \theta_1, \theta_2, \phi)$. Since the coordinate (x_1, y_1) can be calculated given a state s, it is not listed in the definition of the state of the robot. The state transition of the system is governed by equations (5.7) and (5.8). If the tip of the arm is not touching the ground ($\phi(t) = 0$), then the state transition equation is given by

$$\begin{array}{rcl} \theta_1(t+1) & = & \theta_1(t) + \delta_1 \\ \theta_2(t+1) & = & \theta_2(t) + \delta_2 \\ x_0(t+1) & = & x_0(t) \\ y_0(t+1) & = & y_0(t) \\ x_2(t+1) & = & x_0(t+1) + l_1 \sin\theta_1(t+1) + l_2 \sin\theta_2(t+1) \\ y_2(t+1) & = & y_0(t+1) + l_1 \cos\theta_1(t+1) - l_2 \cos\theta_2(t+1) \end{array} , \qquad (5.7)$$

and if the tip of the arm is touching the ground ($\phi(t) = 1$), then the state transition equation takes the form

$$\begin{array}{rcl} \theta_1(t+1) & = & \theta_1(t) + \delta_1 \\ \theta_2(t+1) & = & \theta_2(t) + \delta_2 \\ x_2(t+1) & = & x_2(t) \\ y_2(t+1) & = & y_2(t) \\ x_0(t+1) & = & x_2(t+1) - l_2 \sin\theta_2(t+1) - l_1 \sin\theta_1(t+1) \\ y_0(t+1) & = & l_2 \cos\theta_2(t+1) - l_1 \cos\theta_1(t+1) \end{array} . \qquad (5.8)$$

The quantities δ_1 and δ_2 are the outputs of the neural controller to be designed, and l_1 and l_2 are the lengths of the first and the second link. The first link is between the first joint and the second joint while the second link is between the second joint and the tip of the arm. For the experiment, $l_1 = 34$ cm and $l_2 = 20$ cm are chosen. The first joint is located at right upper corner of the rectangular body of the robotic insect which has a height of 18 cm

and width of 32 cm. A trial contains 50 time steps and at the beginning of a trial the robot is placed at the origin. The fitness function used to evaluate a neural controller is given by

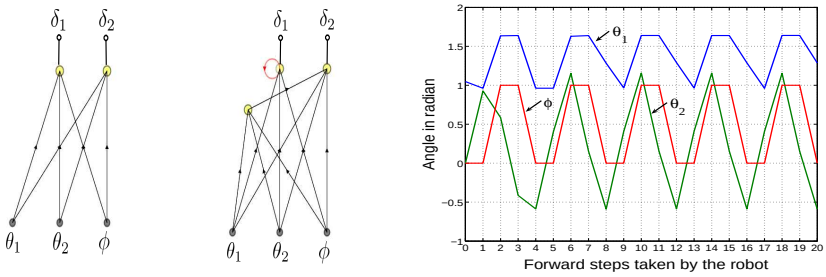
$$f = \frac{1}{N}\sum_{t=1}^{N}(x_0(t) - x_0(t-1)), \tag{5.9}$$

where the difference $x_0(t) - x_0(t-1)$ is the velocity of the system at time t in the direction of the $x-$axis and f is the average velocity of the robot for a trial. The number of time steps used per trial is represented by N.

Tsuchiya et al. [100] applied their policy learning by genetic algorithm using importance sampling (GA-IS) for learning to move forward. They defined a three dimensional vector $X = (x_1, x_2, x_3)$ for representing the state space. The dimensions of the state space is made up of the joint angles and the state of the touch sensor. The policy used in their experiment is a 7 dimensional feature vector $F = [x_1, x_2, x_3, x_4\,(= 1 - x_1)\,, x_5(= 1 - x_2), x_6(= 1 - x_3), 1.0]$. A weight vector $\Lambda = (\lambda_{1,i}, \lambda_{2,i}, \lambda_{3,i}, \lambda_{4,i}, \lambda_{5,i}, \lambda_{6,i}, \lambda_{7,i})$ is used to select the action $a_i(t)$ from normal distribution with mean value $\mu_i = 1/(1 + \exp(-\sum_{k=1}^{6}\lambda_{k,i}x_k))$ and standard deviation $\sigma_i = 1/(1 + \exp(-\lambda_{7,i})) + 0.1$. If the selected action is out of range then it is resampled. The number of the policy parameters is 14 and hence the search space for the genetic algorithm has 14 dimensions.

In our experiment, the structure shown in Figure 5.17 (a) containing two output neurons connected to three input nodes (θ_1, θ_2, ϕ) is used as the initial controller. The best controller shown in Figure 5.17 (b) is found after running EANT. Note that the best controller is more complex than the initial structure. Figure 5.17 (c) shows the waveforms of the joint angles and the touch sensor for the first 20 time steps as the robot moves forward and being controlled by the best controller, and Figure 5.18 shows the sample evolutionary run in obtaining the neural controller.

The method introduced by Tsuchya et al. (GA-IS) needed on the average 10000 interactions with the environment for learning to move forward. We have run EANT 50 times and obtained on the average 3520 interactions for learning the task. As compared to the GA-IS, EANT has reduced the number of interactions with the environment necessary to learn to move forward. The reduction in the number of interactions is due to the direct search for an optimal policy in the space of policies, starting minimally and increasing the complexity of the neural network that represents the policy.



(a) (b) (c)

Fig. 5.17: Learning to move forward. (a) The initial structure. (b) The best controller found by our algorithm that enables the robot to move forward. (c) The waveforms of the joint angles and the touch sensor as the robot moves forward.

5.4.3 Pole Balancing

The inverted pendulum or the pole balancing system has one or several poles hinged to a wheeled cart on a finite length track. The movement of the cart and the poles are constrained within a vertical plane. The objective is to balance the poles indefinitely by applying a force to the cart at regular time intervals such that the cart stays within the track boundaries. A trial to balance the poles fails if either the angle from vertical of any pole exceeds a certain threshold or the cart leaves the track.

The problem has been a standard benchmark for the design of controllers for unstable systems over 30 years [91]. The first reason for using the problem as a standard benchmark is that it is a continuous real-world task that is easy to understand and visualize. Moreover, it can be performed manually by humans and implemented on a physical robot. The second reason is that it embodies many essential aspects of a whole class of learning tasks that involve temporal credit assignment [37]. The controller is expected to discover its own strategy based on the reinforcement signal it receives every time it fails to control the system.

For modern reinforcement learning methods, the basic pole balancing problem, which has only one pole hinged to a wheeled cart, is obsolete since especially for those methods that evolve neural networks the solution is often found in the initial random population [34, 37]. To make the problem challenging, the basic pole balancing is extended in two ways [105]. The first extension is the addition of a second pole next to the other and the second one is the restriction of the state information received by the controller. In the latter case the controller is provided only with the cart position and the

```
Generation 0
Species 0 → N1:3 I2 I0 I1 N2:3 I1 I0 I2

Generation 1
Species 0 → N1:3 I2 I0 I1 N2:4 N3:3 I2 I0 I1 I1 I0 I2
Species 1 → N1:3 I2 I0 I1 N2:3 I1 I0 I2

Generation 2
Species 0 → N1:4 JR1 I2 I0 I1 N2:3 I1 I0 I2
Species 1 → N1:3 I2 I0 I1 N2:4 N3:3 I2 I0 I1 I1 I0 I2

Generation 3
Species 0 → N1:4 JR1 I2 I0 I1 N2:4 N4:3 I0 I2 I1 I1 I0 I2
Species 1 → N1:4 JR1 I2 I0 I1 N2:3 I1 I0 I2
```

Fig. 5.18: A sample evolutionary run for the learning to move forward problem. The species are sorted in descending order according to their fitness values. The first two best solutions are shown for each generations. The input nodes `I0, I1` and `I2` are connected to θ_1, θ_2 and ϕ, respectively.

angles from the vertical of both poles. The first extension makes the task more difficult by introducing non-linear interactions between the poles. The second makes the task non-Markovian which forces the controller to employ short term memory to disambiguate underlying process states. Figure 5.19 describes the double pole balancing problem for poles having unequal lengths. This is the most challenging of the pole balancing versions.

The equations of motion of a cart with N poles are given by [105]

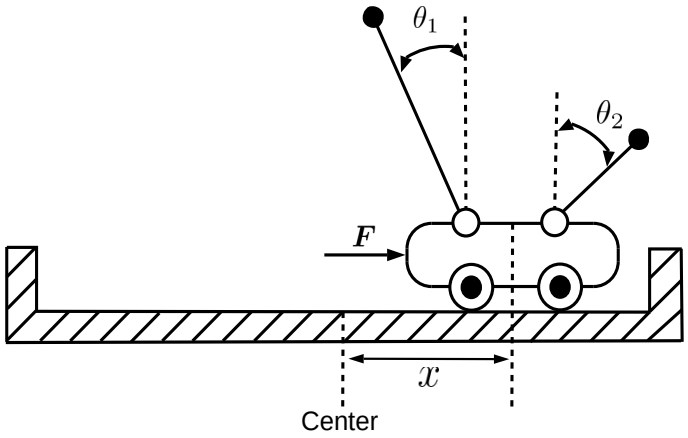


Fig. 5.19: The double pole balancing problem. The poles must be balanced simultaneously by applying a continuous force F to the cart. The parameters x, θ_1 and θ_2 are the offset of the cart from the center of the track and the angles from the vertical of the long and short pole, respectively.

$$
\begin{aligned}
\ddot{x} &= \frac{F - \mu_c \mathrm{sgn}\,(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{m_c + \sum_{i=1}^{N} \tilde{m}_i} \\
\ddot{\theta}_i &= -\frac{3}{4l_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_i \dot{\theta}_i}{m_i l_i}\right) \\
\tilde{F}_i &= m_i l_i \dot{\theta}_i^{\,2}\sin\theta_i + \frac{3}{4} m_i \cos\theta_i \left(\frac{\mu_i \dot{\theta}_i}{m_i l_i} + g\sin\theta_i\right) \\
\tilde{m}_i &= m_i\left(1 - \frac{3}{4}\cos^2\theta_i\right)
\end{aligned}
\tag{5.10}
$$

for $i = 1, \ldots, N$. In the equation, F is the force applied to the cart, x is the offset of the cart from the center of the track, and g is the acceleration due to gravity. The quantities m_i, l_i, θ_i and μ_i stand for the mass, the half of the length, the angle from the vertical, and the coefficient of friction of the i^{th} pole, respectively. The mass and coefficient of friction of the cart are denoted by m_c and μ_c, respectively. The effective force from pole i on the

cart is denoted by $\tilde{F}_i$ and its effective mass is given by $\tilde{m}_i$.

For our benchmark double pole experiments, $N = 2$, $m_c = 1$ kg, $m_1 = 0.1$ kg, $l_1 = 0.5$ m, $l_2 = 0.1l_1$, $m_2 = 0.1m_1$, $\mu_c = 5.10^{-4}$ and $\mu_1 = \mu_2 = 2.10^{-6}$. The length of the track is set to 4.8 m. The parameters are the most common choices for the double pole experiments. The dynamical system is solved using fourth-order Runge-Kutta integration with step size $\tau = 0.01$ s.

Experimental Setup

The experiments are setup in order to be comparable to the results reported in [36, 51, 73, 95]. The controllers perceive continuous states and produce continuous control signals rather than jerk left-right or "bang-bang". Two balancing configurations with and without complete state information are used in the experiments.

Double Pole Balancing with Velocities

In this experiment the controller is provided with full state information $(x, \dot{x}, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2)$ and the initial state of the long pole is set to $\theta_1 = 1^0$. The controller is expected to balance the poles for 10^5 time steps so that the angles of the poles from the vertical lie in the range $[-36^\circ, 36^\circ]$. Each time step corresponds to 0.01s. The descriptions of the experiment are based on that reported in [36, 51, 95].

Double Pole Balancing without Velocities

In this setup, the controller observes only x, θ_1 and θ_2. A fitness function introduced by Gruau et al. [41] together with a termination criterion is used in this task. The same fitness function is used by Gomez and Miikkulainen [35], Stanley and Miikkulainen [94], and Igel [51].

The fitness function is the weighted sum of two separate fitness measurements $0.1f_1 + 0.9f_2$ taken over 1000 time steps:

$$f_1 = t/1000, \tag{5.11}$$

$$f_2 = \begin{cases} 0 & \text{if} \quad t < 100 \\ \dfrac{0.75}{\sum_{i=t-100}^{t} \left(|x^i| + |\dot{x}^i| + |\theta_1^i| + |\dot{\theta}_i^i| \right)} & \text{otherwise,} \end{cases} \tag{5.12}$$

where t is the number of time steps the pole is balanced starting from a fixed initial position. In the initial position, all states are set to zero except $\theta_1 = 4.5^\circ$. The angle of the poles from the vertical must be in the range $[-36^\circ, 36^\circ]$. The fitness function defined favors controllers that can keep the poles near the equilibrium point and minimize the amount of oscillation. The first fitness measure f_1 rewards successful balancing while the second measure f_2 penalizes oscillations.

The evolution of the neural controllers is stopped when a champion of a generation passes two tests. First, it has to balance the poles for 10^5 time steps starting from the 4.5° initialization. Second, it has to balance the poles for 1000 steps starting from at least 200 out of 625 different initial starting states. The ranges of the input variables are ± 2.16 m for x, ± 1.35 m/s for $\dot{x}$, $\pm 3.6^\circ$ for θ_1, and $\pm 8.6^\circ$ for $\dot{\theta_1}$. The starting states are generated using

$$\begin{array}{rcl} x & = & 4.32\ k_i - 2.16 \\ \dot{x} & = & 2.70\ k_j - 1.35 \\ \theta_1 & = & 0.123\ k_m - 0.062 \\ \dot{\theta_1} & = & 0.3\ k_n - 0.15 \\ \theta_2 & = & 0 \\ \dot{\theta_2} & = & 0 \end{array} \quad , \tag{5.13}$$

where $i, j, m, n \in \{0, 1, 2, 3, 4\}$, and $k_0 = 0.05$, $k_1 = 0.25$, $k_2 = 0.5$, $k_3 = 0.75$ and $k_4 = 0.95$. The number of successful balances is a measure of the generalization performance of the best solution.

Results

Table 5.1 shows the average value of network evaluations needed by various methods in solving the double pole balancing task. For our algorithm (EANT), the experiments are done for 120 times for both test scenarios.

From Table 5.1 one can see that our algorithm performs better than other algorithms that evolve both the structure and weights of a neural network (CE, ESP, NEAT). The results are highly significantly better ($p < 0.001$) than the best algorithms which evolve both the network structure and weights of the neural networks. The CMA-ES has outperformed EANT on both double pole balancing tasks. But for CMA-ES the topology (structure) of the neural network has to be chosen manually before optimizing the weights of the network. Figures 5.22 and 5.23 show examples of the results obtained for both test scenarios. For double pole balancing with velocities, our algorithm found a controller having only one output node. Since the evolution starts with neural controllers of minimal structures, the algorithm was able to find this minimal structure for most of the experiments. Figure 5.20 shows a

Method	Double pole balancing with velocity	Double pole balancing without velocity	
	Evaluations	Evaluations	Generalization
CE [41]	34000	840000	300
ESP [37]	3800	169466	289
NEAT [96]	3600	33184	286
CMA-ES [51]	895	6061	250
EANT	1580	15762	262

Tab. 5.1: The average network evaluations (trials) needed by various methods in solving the double pole balancing tasks. For CMA-ES, results for a neural network having 3 hidden nodes without a bias are shown.

```
Generation 0
Species 0 → N1:6 I0 I3 I1 I4 I5 I2

Generation 1
Species 0 → N1:6 I0 I3 I1 I4 I5 I2
Species 1 → N1:7 N2:6 I4 I3 I2 I0 I5 I1 I0 I3 I1 I4 I5 I2
```

Fig. 5.20: A sample evolutionary run for the double pole balancing with velocity. The species are sorted in descending order according to their fitness values. The first four best solutions are shown for each generations. The input nodes `I0, I1, I2, I3, I4` and `I5` are connected to x, $\dot{x}$, θ_1, $\dot{\theta_1}$, θ_2 and $\dot{\theta_2}$, respectively.

sample evolutionary run in solving the double pole balancing problem with velocity. As can be seen from the sample run the initial structure is complex enough in solving the problem.

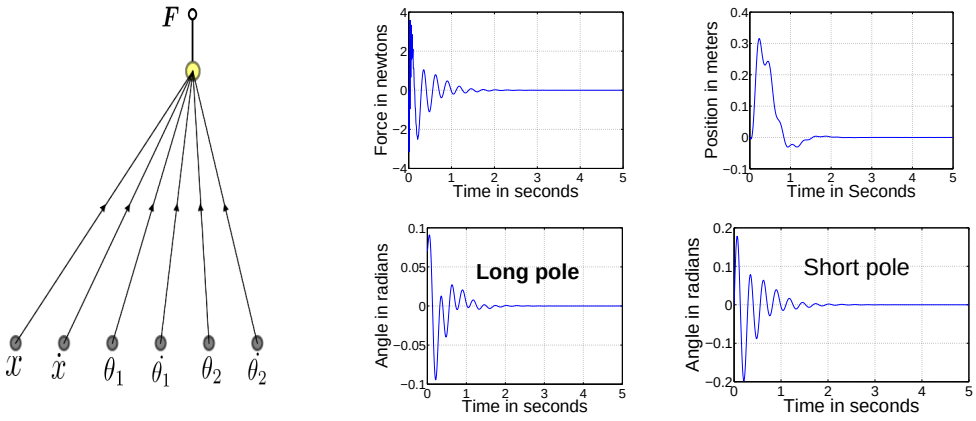
For double pole balancing without velocities, our algorithm found a controller having one output neuron and one hidden neuron. Both neurons have a self-recurrent connection. Once again, one can see that because of starting minimally, it is possible to obtain compact, efficient and clever solutions in the design of controllers. Figure 5.21 shows a sample evolutionary run for the double pole balancing without velocity information.

Our algorithm found both structures consistently for both test scenarios. The waveforms generated by the controllers depend on the connection

```
Generation 0
Species 0 → N1:3 I1 I2 I0

Generation 1
Species 0 → N1:4 JR1 I1 I2 I0
Species 1 → N1:3 I1 I2 I0

Generation 2
Species 0 → N1:5 N2:3 I1 I2 I0 JR1 I1 I2 I0
Species 1 → N1:4 JR1 I1 I2 I0
Species 2 → N1:3 I1 I2 I0
Species 3 → N1:4 N3:3 I0 I2 I1 I1 I2 I0

Generation 3
Species 0 → N1:5 JR1 N3:3 I0 I2 I1 I1 I2 I0
Species 1 → N1:6 N4:4 JF:2 I0 I1 I2 N2:3 I1 I2 I0 JR1 I1 I2 I0
Species 2 → N1:5 N2:3 I1 I2 I0 JR1 I1 I2 I0
Species 3 → N1:4 JR1 I1 I2 I0

Generation 4
Species 0 → N1:5 JR1 N3:4 JR3 I0 I2 I1 I1 I2 I0
Species 1 → N1:5 JR1 N3:3 I0 I2 I1 I1 I2 I0
Species 2 → N1:6 N4:4 JF:2 I0 I1 I2 N2:3 I1 I2 I0 JR1 I1 I2 I0
Species 3 → N1:5 N2:3 I1 I2 I0 JR1 I1 I2 I0
```

Fig. 5.21: A sample evolutionary run for the double pole balancing without velocity. The species are sorted in descending order according to their fitness values. The first four best solutions are shown for each generations. The input nodes `I0, I1` and `I2` are connected to x, θ_1 and θ_2, respectively.

weights. Since for the same structure there are many weight combinations that solves a given task, the waveforms of the force exerted or the waveforms of the angles from the vertical could be different. Figures 5.22 and 5.23 show one possible waveform that can be generated by the structure shown in the respective figures.



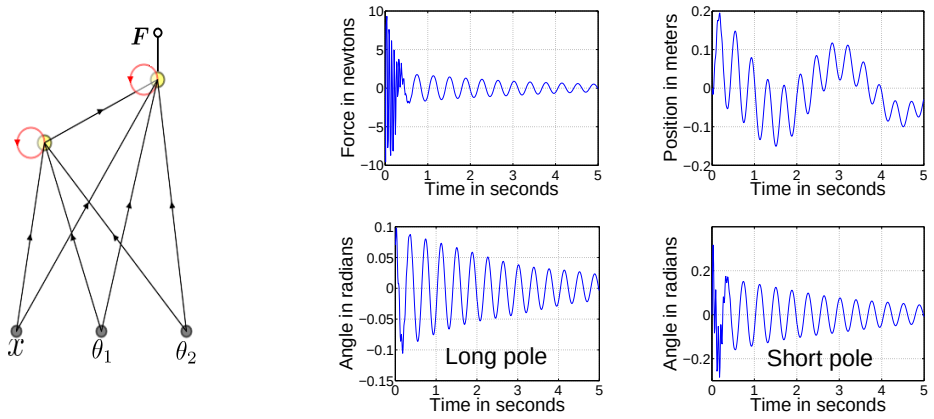
(a) (b)

Fig. 5.22: Double pole balancing with velocities. (a) The best controller found by our algorithm. Note that the minimum neural structure necessary to balance double poles with velocities has only one output neuron. (b) Waveforms of the force exerted, position of the cart on the track and the angular positions from the vertical of both poles.

5.5 Summary

A system that enables autonomous and situated agents to learn and adapt to the environment in which they live and operate is developed. The system exploits both types of adaptations: namely evolutionary adaptation and adaptation through learning. Moreover, self-organization is inherent in the system in that the system starts with networks of minimal structures and increases their complexity along the evolution path. The self-organization process is an emergent property of the system.

EANT introduces a novel compact genetic encoding that encodes a neural network implicitly in the ordering of the elements of the linear genome representing the neural network. The linear genome enables one to evaluate



(a) (b)

Fig. 5.23: Double pole balancing without velocities. (a) The best controller with minimum neural structure found by our algorithm. The controller has one output neuron and one hidden neuron where both neurons have a self-recurrent connection. (b) Waveforms of the force exerted, position of the cart on the track and the angular positions from the vertical of both poles.

the neural network encoded by it without some type of ontological process of transforming the genotype into phenotype. The presented genetic encoding is complete in that it can encode any type of neural network, and closed under structural mutation and a specially designed crossover operator. The crossover operator exploits the fact that neural structures that come from some initial neural structures have some parts in common. By aligning the common parts of two randomly selected structures, it is possible to generate a valid third structure which contains the common and disjoint parts of the two mother structures.

In addition to this, a nature inspired meta-level evolutionary process is introduced that is suitable to explore new structures incrementally and exploit the existing ones. Structural mutation and crossover operators are used to search for structures at larger timescale, and parametric mutation and recombination operators are used to optimize the weights of the existing structures at smaller timescale.

Structural discoveries or innovations of the evolutionary process are protected by carrying on young structures, which are few years old with respect to larger timescale, along the evolution regardless of the selection operator. This gives the young species to optimize their newly acquired structures be-

fore they compete with other individuals globally.

Through introduction of neural structures by structural mutation, there is a gradual increase in the complexity of structures along the evolution. Those structures that are better according to fitness evaluations survive and continue to exist in the population. The complexification process enables EANT to search for an optimal solution starting from a minimum structural complexity specified by the domain expert.

Chapter 6

VISION BASED ROBOT NAVIGATION

The aim of this chapter is to demonstrate the automatic design of neural controllers for robots using EANT. The problem of robot navigation with obstacle avoidance is chosen as a test bed where controllers are expected to give the robot the ability of exploring the environment.

We start with sonar based robot navigation for developing the controllers in simulation and then transfer the developed controllers to real robot. After having a controller that can control the robot in the environment, the inputs to the controller are exchanged with the inputs from the camera system of the robot. Based on stereo matching technique, we give a method of detecting obstacles which is equivalent to the obstacle detection using sonar sensors.

6.1 The Physical Robot

The B21 robot from Real World Interface (RWI) [83] is used for experiments in this chapter. The robot has a cylindrical body with two parts: a base and an enclosure. It uses its four-wheeled synchronous drive to move on indoor flat platform. The base has 32 infra-red and 32 tactile sensors, whereas the enclosure has a belt of 24 tactile, 24 infra-red and 24 sonar sensors each evenly placed around the robot's perimeter. On the top of the enclosure a two finger Scara robot manipulator with 6 DOF and a binocular CCD camera are mounted. In addition, the robot is equipped with two on board computers running a Linux operating system. One of the computers controls the base, the manipulator and the pan-tilt unit carrying the cameras of the robot. Moreover, it is used to acquire data from the sonar, tactile and infra-red sensors. The other computer is connected to the CCD cameras and is used to acquire images that are perceived by the cameras. The computers are connected to each other by a local LAN network. Figure 6.1 show the B21 experimental robot.

Fig. 6.1: The B21 experimental mobile robot. The robot is equipped with 24 sonar, 56 infra-red, 32 tactile sensors, and a binocular CCD camera.

6.2 Reactive Navigation with Obstacle Avoidance

We evolved the structure and weights of the neural controller which enables B21 robots to autonomously explore the environment and avoid obstacles. The controller is expected to avoid dead lock situations where Braitenberg-like controllers [15] have difficulties of escaping them. In these situations, they either come to a rest or start to oscillate left to right.

We first used the sonar sensors of the B21 robot for detecting the obstacles. This helps us to design the controller in simulation and transfer it to real robot. After testing the controller on the real robot, we introduce a vision module which detects obstacles using stereo matching technique and feed the neural network with information obtained from the vision module for controlling the robot in the environment.

The B21 robot has 24 sonar sensors which are symmetrically distributed around its cylindrical body. We used the 8 in front and 2 in the rear sonar sensors as inputs to the neural controller. Figure 6.2 shows the 8 frontal

sonar sensors used in the experiment.

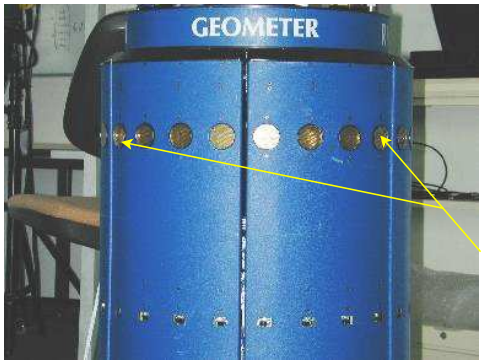


Fig. 6.2: The 8 frontal sensors used in the experiment. The 2 in the rear sonar sensors not shown in the figure are at the opposite side of the two frontal sensors.

The sonar sensors give the distance of obstacles in millimeters measured from the center of the robot. The values returned by the sonar sensors are transformed using equation (6.1) before feeding them to the neural controller.

$$V_n = \begin{cases} \dfrac{-V_s + 2000}{2000} & \text{if } V_s < 1000 \\ 0 & \text{otherwise} \end{cases} . \tag{6.1}$$

In the equation, V_n is the transformed and normalized sonar reading and V_s is the actual reading returned by a particular sonar sensor. The value of V_n lies between 0 and 1 for obstacles which are located at a distance less than 2 m from the center of the robot. The value of V_n increases as the obstacle come near to the center of the robot.

The initial controller has two output neurons and each neuron is connected to all sensors. The outputs of the neurons are connected to the motor apparatus of the robot. In addition to the sensor inputs, each neuron has a constant bias input connected to it. The forward translational velocity and rotational velocity of the robot is given by

$$\begin{aligned} V_t &= 0.5(O_1 + O_2) \\ R_t &= O_1 - O_2 \end{aligned} , \tag{6.2}$$

where O_1 and O_2 are the outputs of the neural network, and V_t and R_t are the translational and rotational velocity of the robot. Since the output of the neurons is between -1 and 1, the maximum and minimum forward velocity

of the robot is 1 m/s and −1 m/s, respectively. The rotational velocity is bounded between 2 rad/s and −2 rad/s.

The initial controller is similar to Braitenberg-like controller and is not capable of avoiding dead lock situations. The algorithm is expected to find a controller which is complex enough for solving the navigation problem with the ability of avoiding dead lock situations. The fitness function used to evaluate the controllers is given by

$$F = \sum_{t=1}^{T} D(t)e^{-100(H(t)-H(t-1))^2}(1 - S_{max}(t)), \tag{6.3}$$

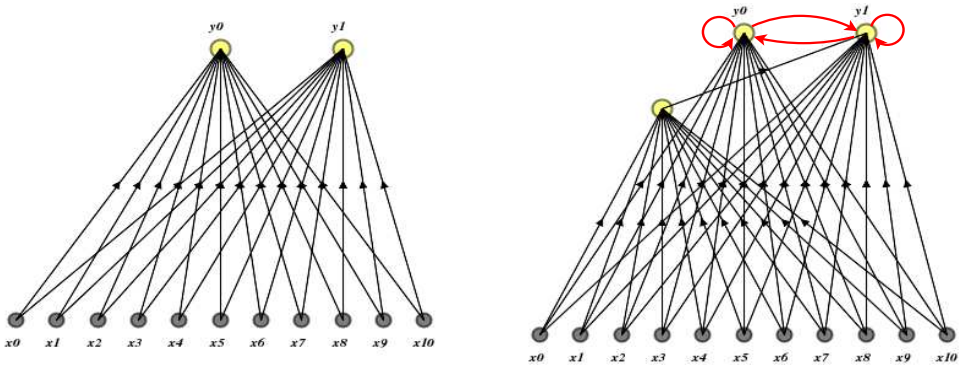
where $D(t)$, $H(t)$ and $S_{max}(t)$ are the distance traveled, the heading of the robot, and the maximum value of the currently perceived normalized sonar readings respectively. The fitness function favors controllers that move straight as long and as fast as possible and controllers that give the robot the maximum distance from the obstacles.

Figure 6.3 shows the initial neural controller and the final controller obtained by our algorithm. The ability of avoiding the dead lock situations comes because of the recurrent connections. The result is similar to that obtained by Nolfi and Floreano [75] and Hülse and Pasemann [50] but in both cases the structure of the neural controller is determined manually beforehand.

Ahrns et.al [1] designed a fuzzy-neuro controller for solving the robot navigation with obstacle avoidance. They solved the dead lock situations problem by designing a feature extraction mechanism that extracts a free space direction closest to the heading of the vehicle. They further stored the sonar readings in a short time memory to extract the coarse model for the direct robot surroundings. They used the feature extraction mechanism since the fuzzy-neuro controller does not have recurrent connections.

It is difficult to make performance comparisons between the algorithms tested on the obstacle avoidance problem. This is because the controllers used are manually designed in case of other algorithms but in our case it is evolved automatically. However, we made 50 independent runs and averaged the number of evaluations necessary to get a neural controller for solving the obstacle avoidance problem using our method. The result shows that our algorithm needed on average about 4500 evaluations to solve the task. Figure 6.4 shows a sample evolutionary run while evolving the neural controller.

The initial Braitenberg-like controller and the best neural controller found are tested in an environment with sharp corners. The sharp corners form dead lock situations where Braitenberg-like controllers have difficulties of escaping them. Figure 6.5 shows an example of the performance of both controllers in



(a) (b)

Fig. 6.3: (a) The initial Braitenberg-like controller (b) The best neural controller found by EANT that is capable of avoiding dead lock situations.

a given environment. The best neural controlled found by EANT shows the behavior of avoiding dead lock situations and exploring the environment.

After transferring and testing the neural controller on real robot, we developed a vision model based on stereo matching which is equivalent to the sonar sensors on the robot. The only difference is that the vision module can not perceive obstacles which lie in opposite direction to the current heading of the robot.

6.3 Vision Module

The vision module uses stereo matching and the fact that the indoor platform on which the robot navigates is flat for the purpose of detecting obstacles. Anything that is above the floor is considered as an obstacle. The cameras are oriented in parallel to each other but both have the same tilt angle measured from horizontal. Figure 6.6 shows the side and top view of the cameras on the B21 robot.

The forward projective mapping [107] of a point with coordinates (x, y, z) with respect to the world coordinate system onto the plane image of a camera with coordinate (u, v) is given by

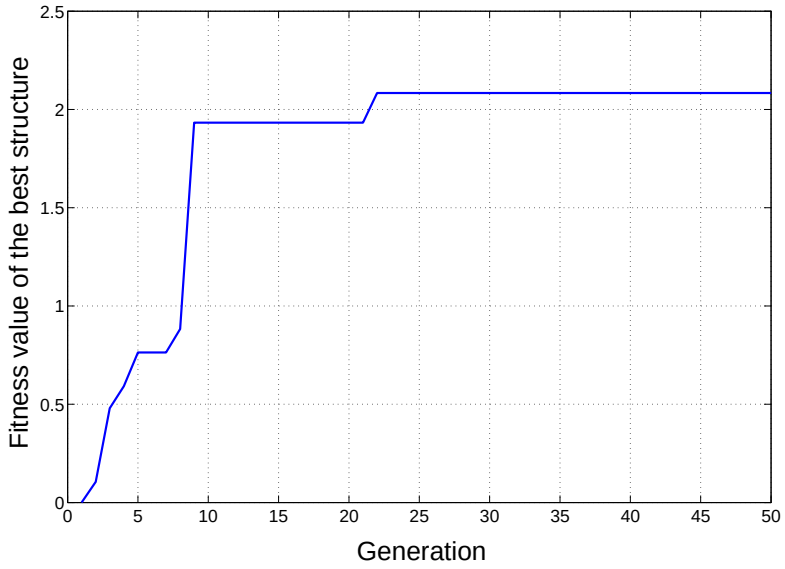


Fig. 6.4: Sample evolutionary run while evolving the neural controller. Note that as the fitness value of the best structure increases, its topology assumes the optimal structure.

$$\begin{pmatrix} s \cdot u \\ s \cdot v \\ s \end{pmatrix} = H \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{6.4}$$

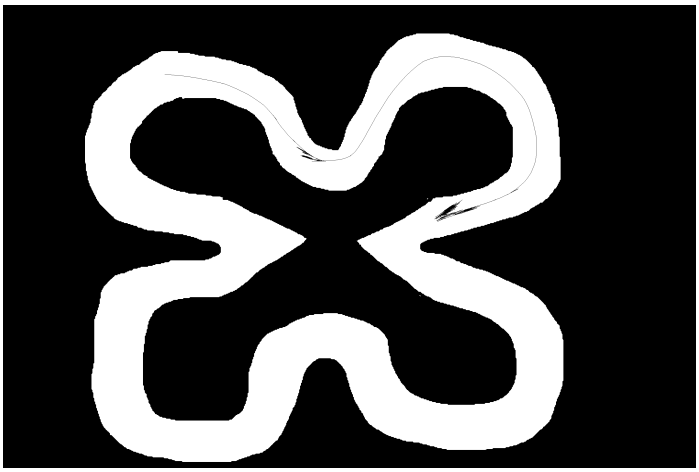
where s is a scaling factor and H is a 3×4 projection matrix. The projection matrix can be defined to take the form

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & 1 \end{pmatrix}. \tag{6.5}$$

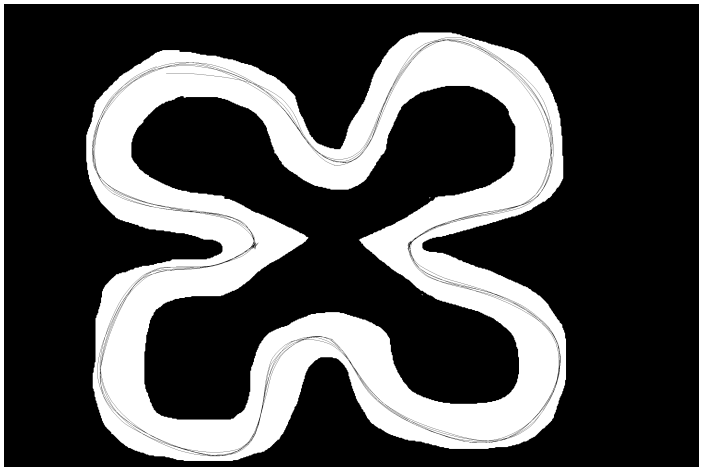
Inserting equation (6.5) into equation (6.4) we obtain

$$\begin{pmatrix} s \cdot u \\ s \cdot v \\ s \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{6.6}$$

If we eliminate the unknown s in equation (6.6), we get



(a)



(b)

Fig. 6.5: (a) Trajectory of the robot controlled by the initial Braitenberg-like controller in a simulated environment. Note that the controller can not escape the sharp corner. (b) Trajectory of the robot controlled by the best neural controller. The controller found is capable of avoiding dead lock situations.

$$\begin{aligned} u &= \frac{xh_{11}+yh_{12}+zh_{13}+h_{14}}{xh_{31}+yh_{32}+zh_{33}+1} \\ v &= \frac{xh_{21}+yh_{22}+zh_{23}+h_{24}}{xh_{31}+yh_{32}+zh_{33}+1}. \end{aligned} \tag{6.7}$$

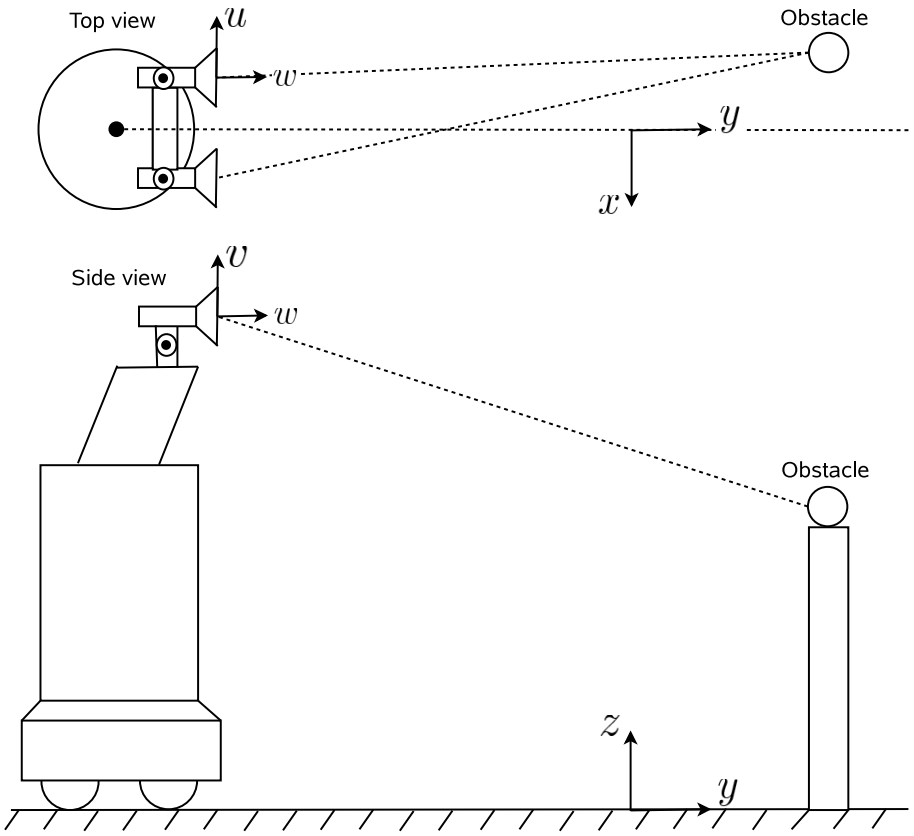


Fig. 6.6: The top and side views of the camera system used to detect obstacles above the ground. The axes of world coordinates are designated by x, y and z. The axes u and v are parallel to the image plane of a camera while w is perpendicular to it.

If we define

$$A = \begin{bmatrix} x & y & z & 1 & 0 & 0 & 0 & 0 & -xu & -yu & -zu \\ 0 & 0 & 0 & 0 & x & y & z & 1 & -xv & -yv & -zv \end{bmatrix},$$

$$B = \begin{bmatrix} u \\ v \end{bmatrix}$$

and

$$V = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{21} & h_{22} & h_{23} & h_{24} & h_{31} & h_{32} & h_{33} \end{bmatrix}^t,$$

equation (6.7) can be compactly written in matrix form as:

$$A \cdot V = B. \tag{6.8}$$

In order to calibrate a camera, one needs at least six pairs of $\{(u, v), (x, y, z)\}$, where each pair provides two constraints. Assume that a set of n pairs of $\{(u, v), (x, y, z)\}$ are available ($n \geq 6$), the optimal solution which minimizes the squared error $\| A \cdot V - B \|^2$ given by

$$V = (A^t A)^{-1} \cdot (A^t B) \tag{6.9}$$

contains the elements of the projection matrix H.

Using the above calibration method, both cameras are calibrated with respect to a common world coordinate system. As shown in Figure 6.6, the x and y coordinates of the common world coordinate system lie on the flat indoor platform.

The detection of obstacles begins with the detection of corner points using the Harris corner detector [46] on left image. We assume that the obstacles have some corner points which are going to be detected by the corner detector algorithm. Figure 6.7 shows the corner points of the obstacles detected on the image perceived by the left camera.

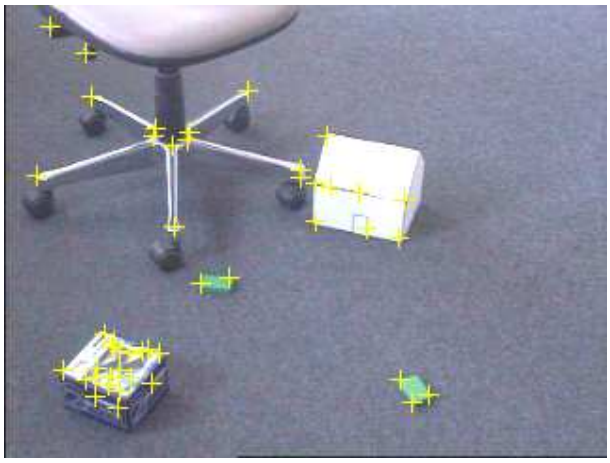


Fig. 6.7: Example of corner points detection using the Harris corner detector.

After the corner points are detected on the left image, 3D imaginary projection lines which pass through the corner points detected on the image plane and intersect the flat indoor platform are drawn. The equation of a projection line that passes through a corner point with image coordinate (u_c, v_c) is given by

$$\boldsymbol{p} \times \boldsymbol{r} - \boldsymbol{m} = \boldsymbol{0}, \tag{6.10}$$

where $\boldsymbol{p}$ is a 3D point on the projection line, $\boldsymbol{r}$ is direction vector along the line and $\boldsymbol{m}$ is the moment of the line. The direction vector and the moment of the line are calculated as follows. Assuming that the calibration matrix H_L of the left camera is given by

$$H_L = \begin{pmatrix} h'_{11} & h'_{12} & h'_{13} & h'_{14} \\ h'_{21} & h'_{22} & h'_{23} & h'_{24} \\ h'_{31} & h'_{32} & h'_{33} & 1 \end{pmatrix}. \tag{6.11}$$

The direction vector in the direction of the focal point of the left camera from the indoor platform is calculated as follows. First a direction vector $\boldsymbol{r'}$ given by

$$\begin{aligned} R &= (h'_{21}h'_{32} - h'_{22}h'_{31})u_c + (h'_{31}h'_{12} - h'_{32}h'_{11})v_c + h'_{22}h'_{11} - h'_{21}h'_{12} \\ r'_x &= \frac{(h'_{32}h'_{23}+h'_{33}h'_{32})u_c+(h'_{13}h'_{31}-h'_{12}h'_{33})v_c+h'_{12}h'_{23}-h'_{13}h'_{23}}{R} \\ r'_y &= \frac{-\left((h'_{21}h'_{33}-h'_{23}h'_{31})u_c+(h'_{31}h'_{13}-h'_{33}h'_{11})v_c+h'_{23}h'_{11}-h'_{21}h'_{13}\right)}{R} \\ r'_z &= 1.0 \end{aligned} \tag{6.12}$$

is determined. Then the direction vector is calculated using

$$\boldsymbol{r} = \frac{\boldsymbol{r'}}{\| \boldsymbol{r'} \|}. \tag{6.13}$$

In order to calculate the moment of the projection line, one has to get at least one point that lies on the line. The intersection point between the line and the indoor platform given by

$$\begin{aligned} p'_x &= \frac{(h'_{34}h'_{22}-h'_{32}h'_{24})u_c+(h'_{14}h'_{32}-h'_{12}h'_{34})v_c+h'_{12}h'_{24}-h'_{14}h'_{22}}{R} \\ p'_y &= \frac{-\left((h'_{24}h'_{31}-h'_{21}h'_{34})u_c+(h'_{31}h'_{14}-h'_{34}h'_{11})v_c+h'_{24}h'_{11}-h'_{21}h'_{14}\right)}{R} \\ p'_z &= 0.0 \end{aligned} \tag{6.14}$$

can be used in calculating the moment of the projection line which can be calculated using $\boldsymbol{m} = \boldsymbol{p'} \times \boldsymbol{r}$.

Once the equations of the projection lines are determined, it is possible to search for the corresponding point on the right image for every corner point detected on left image. The search is accomplished as follows. We start from an intersection point of the projection line with the indoor platform which can be calculated using equation (6.14) for a given detected corner point with image coordinates (u_c^L, v_c^L). We move along the projection line in small steps until the height of the point above the indoor platform is 1 m. The position of the point along the line is updated using

$$\boldsymbol{p'} = \boldsymbol{p} + h\boldsymbol{r} \tag{6.15}$$

where h is the step size, $\boldsymbol{p}$ is the old position, and $\boldsymbol{p'}$ is the current position on the line. The value of h is determined by

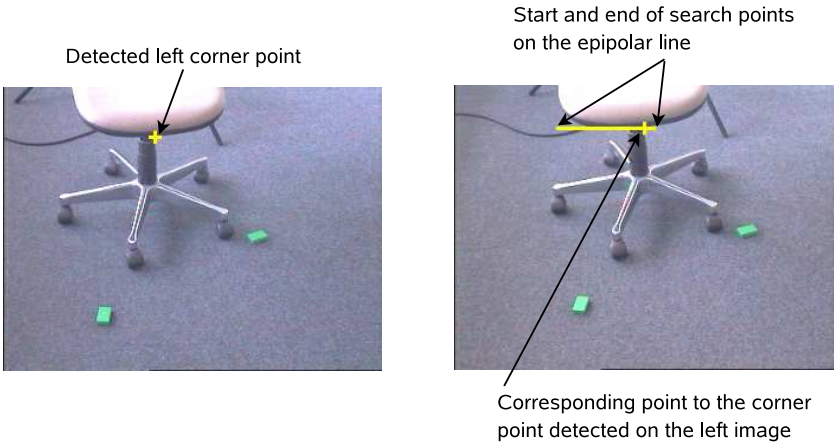
$$\frac{1}{kr_z} \tag{6.16}$$

where k is the number of steps needed to traverse the projection line from the indoor platform to a height of 1 m, and r_z is the z-component of the direction vector of the projection line. While moving along the line, the point is projected onto the image plane of the right camera using its projection matrix. Moving along the imaginary projection lines is equivalent to moving on epipolar lines in the image plane of the right camera. Let the coordinate of the projected point in the right image be (u_c^R, v_c^R). A measure of similarity between two regions centered around (u_c^L, v_c^L) and (u_c^R, v_c^R) called the correlation coefficient [53] defined by

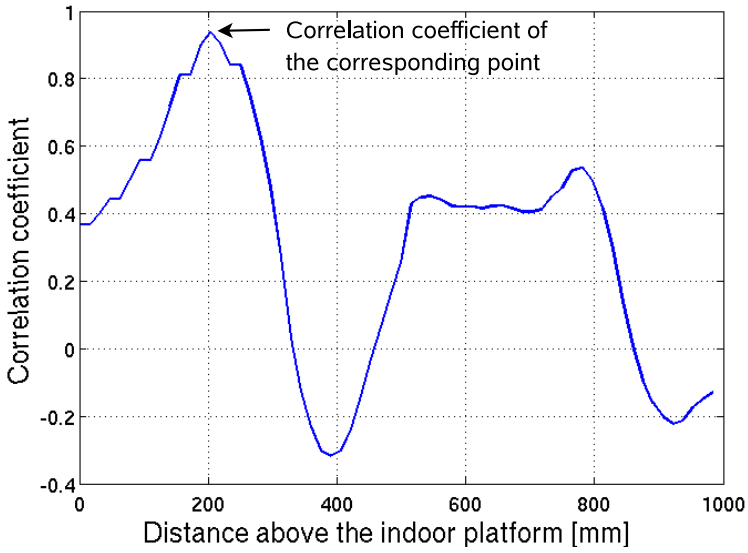
$$r = \frac{\sum\limits_{(x,y)\in S} \left[f_L(x,y) - \overline{f}_L\right]\left[f_R(x+d_x, y+d_y) - \overline{f}_R\right]}{\left\{\sum\limits_{(x,y)\in S} \left[f_L(x,y) - \overline{f}_L\right]^2 \sum\limits_{(x,y)\in S} \left[f_R(x+d_x, y+d_y) - \overline{f}_R\right]^2\right\}^{\frac{1}{2}}} \tag{6.17}$$

is calculated. In the equation, f_L and f_R are the left and right images respectively, and (d_x, d_y) is the disparity between the corner point and the point projected on the right image plane of the camera.

The correlation coefficient is calculated for every point along the line selected using equation (6.15). The point at which the maximum correlation coefficient occurs gives us the 3D coordinates of the corner point relative to the common world coordinate system. The projection of the point onto the image plane of the right camera, at which the maximum correlation coefficient



(a)



(b)

Fig. 6.8: (a) There is no need to search the whole epipolar line to find the corresponding point to the corner point detected on the left image. (b) The corresponding point on the right image occurs where the correlation coefficient becomes maximum.

occurs, gives us the coordinate of the corresponding point to the corner point detected on the left image. Figure 6.8 illustrates the idea.

After the corresponding points for every left corner detected in the left image are found, the position of obstacles relative to the world coordinate

system is determined. The most important coordinates of the obstacles are the x and y coordinates. The x-coordinate of the world coordinate system is perpendicular to the heading of the robot while the y-coordinate is parallel to it. The origin of the world coordinate system is translated to the center of the robot. That means the coordinates of the obstacles will be adjusted according to

$$\begin{aligned} x' &= x \\ y' &= y + d \\ z' &= z \end{aligned} \tag{6.18}$$

where d is the distance between the common world coordinate system and the center of the robot. Figure 6.9 shows the vision module used to perceive the distance of obstacles from the center of the robot. Radial imaginary lines that start from the center of the robot are constructed to divide the perception field of the vision module into perception regions. The number of perception regions is equal to the number of the in front sonar sensors used for designing the neural controller.

If more that one corner points of an obstacle (obstacles) are detected in a perception region of the vision module, the corner point of an obstacle which is nearer to the center of the robot is taken. The distance of a corner point of an obstacle from the center of the robot falling in one of the perception regions n is calculated by

$$d_n = \sqrt{x'^2 + y'^2} \tag{6.19}$$

where x' and y' are the coordinates of the corner point relative to the world coordinate system translated to the center. Before feeding the distance to the corresponding input of the neural network, it is scaled using

$$d'_n = \begin{cases} 0 & \text{if } d_n \leq 2 \\ \dfrac{d_n - 2}{3} & \text{if } d_n \text{ is in the interval } (2, 5) \\ 0 & \text{otherwise} \end{cases} . \tag{6.20}$$

where all the distances are in meters. The scaled distance lies between 0 and 1 for corner points which are at a distance between 2 m and 5 m from the center of the robot.

Figure 6.10 shows a result of an experiment in detecting obstacles using the vision module developed. In the experiment, three objects namely office chair, a small black box and a flat paper with a white object drawn on it

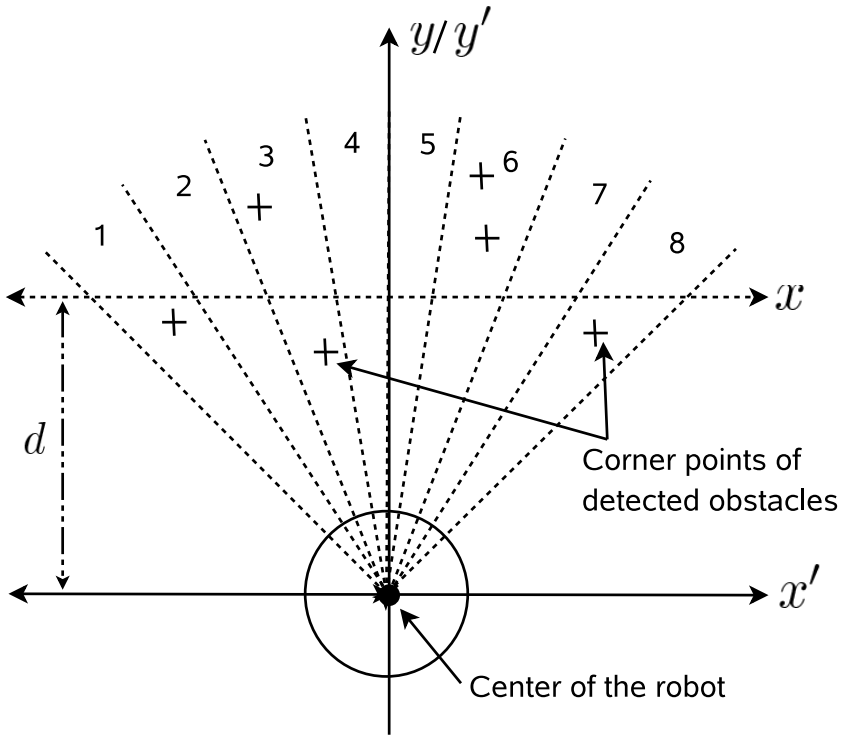


Fig. 6.9: Perception regions of the vision module developed. Note that the perception regions are equivalent to sonar sensors.

are placed in front of the robot at a distance between 2 m and 5m from the center of the robot. One can see in Figure 6.10 (b) that only the chair and the box are considered as obstacles. The paper is not considered as an obstacle eventhough a corner point is detect on it. This is because it has no significant part that lies above the indoor platform.

The main problem of the vision module is the detection of corner points of obstacles that are not texturized. A very good example is a white colored wall which is common in indoor platforms. To avoid collisions with obstacles having no texture, the neural controller switches to sonar sensors if one of the sonar sensors detects an obstacle which is at a distance less than 2 m from the center of the robot. That means if one of the sonar sensors is active, the controller uses inputs from the sonar sensors, otherwise it uses the information obtained from the vision module.

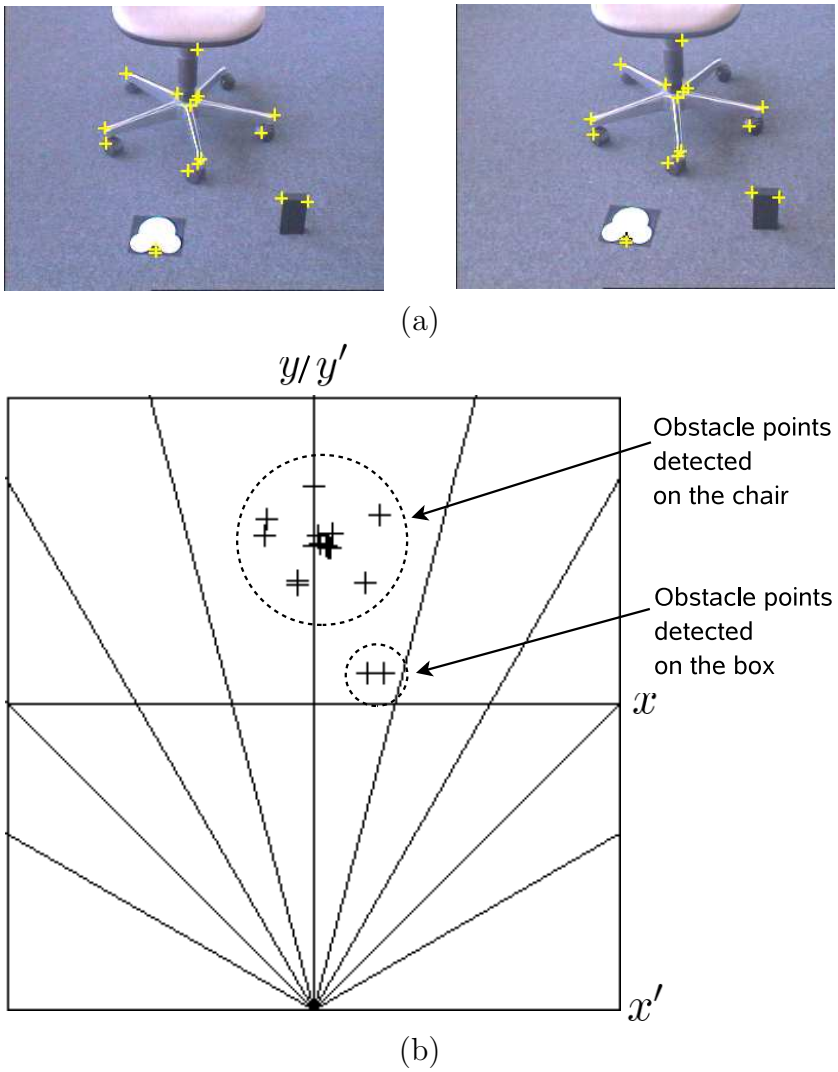


Fig. 6.10: (a) The corner points detected in the left image with their corresponding points in the right image. (b) Result of obstacle detection. Note that the paper lying on the flat platform is not detected as an obstacle.

6.4 Summary

In this chapter we have demonstrated the automatic design of controllers for real robots using EANT taking as an example robot navigation with reactive obstacle avoidance. EANT found a clever solution that gives the robot the ability to explore the environment without being trapped in dead-lock sit-

uations. Simple designs like Braitenberg-like controllers have difficulties of escaping dead-lock situations.

A vision module based on stereo matching is developed for detecting obstacles above the flat indoor platform for vision based robot navigation. The perception field of the vision module is designed so that it is equivalent to the sonar sensors on the B21 robot. This helps us to directly use the controller developed for the sonar sensors.

Chapter 7

CONCLUSION

In this chapter a summary of the main ideas raised throughout the previous chapters and an outline of possible future directions of research are presented.

7.1 Summary

In the thesis a system that enables autonomous and situated agents to learn and adapt to the environment in which they live and operate is developed. The system exploits both types of adaptations: namely evolutionary adaptation and adaptation through learning. Self-organization is inherent in the system in that the system starts with networks of minimal structures and increases their complexity along the evolution path.

The thesis started with ways of improving the learning and adaptation capabilites of agents using a navigation problem in an artificial robot world. The agents are expected to learn through interaction with the environment, and are not provided with the dynamics of the environment a priori. From the results of the experiments, it is concluded that the learning and adaptation time required in continual learning is shorter than that required in learning from scratch and under various learning conditions and at both individual and population levels. The learning time 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. Hybridization of learning algorithms such as reinforcement learning with evolutionary methods give the agents two levels of learning and 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. The most important conclusion derived from the experiments is that with adequate combination of learning and adaptation algorithms which occur at individual and population levels, it is possible to design better agents with better learning and adaptation

capabilities. This motivated the realization of a unified approach to learning and adaptation.

The unified approach to learning and adaptation which combines the concepts from neural networks, reinforcement learning and evolutionary algorithms is used for the design of the learning system for autonomous intelligent systems. Neural networks are used to represent value functions or policies. The structure and weights of neural networks are altered through genetic operators. The agents which embed the neural networks are tested in the environment and performance measures (rewards) are given to the neural structures.

Based on the unified approach to learning and adaptation, a novel method of evolving the structure and weights of neural structures (EANT) is developed. The method starts with neural networks of minimal structures, whose initial complexity is specified by the domain expert. The complexity of the neural structures increases along the evolution path until a solution is found. A new genetic encoding of neural networks that is suitable for evolving the structure and weights of neural networks is introduced. The genetic encoding uses linear genomes of genes that encodes the topology of the neural network implicitly in the ordering of the elements of the linear genome. This type of encoding enables one to evaluate the linear genome without having some ontological process of developing the neural network encoded by it. Furthermore, the genetic encoding has some interesting properties that makes it useful for the evolution of neural networks:

1. The genetic encoding is *complete* in that it can encode any type of neural network.

2. It is a *compact* encoding of neural networks since the length of linear genome is the same as the number of synaptic weights of the neural network.

3. It is *closed* under structural mutation and under specially designed crossover operator, where the crossover operator exploits the fact that structures which originate from the initial minimal structure have some parts in common. By aligning the common parts of two randomly selected structures, it is possible to generate a third structure which contains the common and disjoint parts of the two mother structures. The resulting structure formed in this way maps to a valid phenotype network.

4. Genes that make up a sub-network of the network encoded by a linear genome can be easily identified. This feature can be used in designing the evolution of hierarchical and modular neural networks.

The main search operators used in EANT are the structural mutation, structural crossover and parametric mutations. The structural mutation adds or removes forward or a recurrent jumper connection between neurons, or adds a new sub-network to the linear genome. It operates only on neuron nodes. The weights of a newly acquired topology are initialized to zero so as not to disturb the performance of the network. Structural crossover operates on two randomly selected mother species to generate a new species. The parametric mutation is accomplished by perturbing the weights of the neural networks.

The developed method introduces a biologically motivated meta-level evolutionary process where each structure represents a particular species. All species start to develop from an initial species or structure. The initial structure is generated using either the grow or full method. Its initial complexity is determined by the domain expert and is specified by the maximum depth that can be assumed by the initial structure. At each generation, existing species are exploited. Exploitation implies optimization of the weights of the structures that is accomplished by an evolutionary process that occurs at smaller timescale. The evolutionary process at smaller timescale uses parametric mutation and recombination as search operators. Exploration of structures or creation of new species is done through structural mutation and crossover operators. The structural selection operator that occurs at larger timescale selects the best individuals to form the next generation.

In order to protect the structural innovations or discoveries of the evolution, young structures that are few generations old with respect to the larger timescale are carried over along the evolution regardless of the results of the selection operator. This gives them time to optimize their newly acquired structures before they compete with other individuals globally.

New structures that are introduced through structural mutation and which are better according to the fitness evaluations survive and continue to exist. Survival of the fittest results from both intraspecies competition which occurs during exploitation of species and interspecies competition which occurs at larger timescale. Since sub-networks that are introduced are not removed, there is a gradual increase in the complexity of structures along the evolution. This allows EANT to search for a solution starting from a minimum structural complexity specified by the domain expert. The search stops when a neural network with the necessary minimal structure that solves a given task is obtained.

From the result of experiments, one concludes that the developed system is suitable for reinforcement learning tasks. Moreover, the results show that the solution obtained by the system is efficiently found with higher repeatability rate and has the optimal structure necessary for solving a given learning task.

Parts of this thesis are published in [56, 57, 58, 59, 60, 61, 62].

7.2 Outlook

The work presented in this thesis is a step forward in the unified approach to learning and adaptation. But still there are open questions and therefore the work can be directed to many different aspects. In this section, we try to point out the main possible future directions of research.

An immediate logical extension of the genetic encoding is the introduction of input delay genes. In addition to recurrent connection gene, the input delay gene will help the neural network to capture the dynamics of the environment, where the system has the ability of predicting the next state of the environment based on the current perceived state.

The development of behavior based evolutionary robotics using the learning system developed is a very interesting research area. For such systems, the learning system should handle the evolution of hierarchical structures and modular networks. In order to direct the evolution efficiently, ways of describing the search space as well as the final resultant networks should be developed.

The genetic encoding can also be extended to the areas of indirect encoding where developmental rules that are used in constructing the phenotype are encoded. An indirect encoding based on the principles of developmental biology enables the system in evolving and representing very large neural networks and complex structures. Moreover, it enables one in designing compact and efficient genetic encoding schemes for representing repetitive and recurrent structures.

Since the neuron nodes of the linear genome can assume any function, non-neural structure evolution is another area of research where the developed genetic encoding can be used. Application areas could be evolution of structures for image processing and object recognition, robot body morphologies, electrical circuits and finite automata.

We hope that the developed genetic encoding gives a good basis for the design and development of evolutionary systems that evolve the necessary optimal structure starting from some initial structures.

BIBLIOGRAPHY

[1] I. Ahrns, J. Bruske, G. Hailu, and G. Sommer. Neural fuzzy techniques in sonar-based collision avoidance. In L.C. Jain and T. Fukuda, editors, *Soft Computing for Intelligent Robotic Systems*, Studies in Fuzziness and Soft Computing, pages 185–214. Physica-Verlag (Springer), 1998.

[2] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65, 1994.

[3] R. C. Arkin. *Behaviour-Based Robotics.* MIT Press, Massachusetts, London, 1998.

[4] T. Bäck. Evolution strategies: An alternative evolutionary algorithm. In *Artificial Evolution*, pages 3–20, 1995.

[5] K. Balakrishnan and V. Honavar. Properties of genetic representations of neural architerctures. In *Proceedings of the World Congress on Neural Networks*, pages 117–146, 1995.

[6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann, San Francisco, CA, 1998.

[7] A. Barto, R. Sutton, and C. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.

[8] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *AI Journal on Special Volume on Computational Research on Interaction and Agency*, 72:81–138, 1995.

[9] G. Bebis, S. Louis, Y. Varol, and A. Yfantis. Genetic object recognition using combinations of views. *IEEE Transactions on Evolutionary Computation*, 6(2):132–146, 2002.

[10] P. J. Bentley and D. W. Corne. *Creative Evolutionary Systems.* Academic Press, San Diego, CA, 2002.

[11] P. J. Bentley and S. Kumar. The ways to grow designs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 35–43, 1999.

[12] R. Beveridge and M. Riseman. How easy is matching 2D line models using local search? *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 6(1), 1997.

[13] C. M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[14] J. C. Bongard and R. Pfeifer. Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, pages 829–836, 2001.

[15] V. Braitenberg. *Vehicles. Experiments in Synthetic Psychology.* MIT Press, Massachusetts, London, 1994.

[16] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[17] R. A. Brooks and A. M. Flynn. Fast, cheap and out of control: A robot invasion of the solar system. *Journal of the British Interplanetary Society*, 42:478–485, 1989.

[18] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[19] R.A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural Computation*, 1(2):253–262, 1989.

[20] R.A. Brooks. Intelligence without representation. *Journal of Artificial Intelligence*, 47:139–159, 1991.

[21] J. Bruske, I. Ahrns, and G. Sommer. Practicing Q–learning. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, pages 25–30, Brugges, 1996.

[22] A. Bunten. Ein Autonomes Robotersystem zum Folgen einer Person. Diplomarbeit, Institute of Computer Science and Applied Mathematics, Christian-Albrechts University, Kiel, Germany, January 2002.

[23] K. Chellapilla and D. B. Fogel. Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5:422–428, 2001.

[24] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Applications*, pages 183–187, 1985.

[25] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. John Murray, 1859.

[26] D. Dasgupta and D. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96, 1992.

[27] F. Dellaert and R. D. Beer. A developmental model for the evolution of complete autonomous agents. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, 1996.

[28] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, Berlin, Heidelberg, New York, 2003.

[29] D. Floreano and J. Urzelai. Evolution of neural controllers with adaptive synapses and compact genetic encoding. In *Proceedings of the 5th European Conference on Artificial Life (ECAL)*, pages 13–17. Springer Verlag, 1999.

[30] D. Floreano and J. Urzelai. Evolutionary robots with on-line self-organization and behavioral fitness. *Neural Networks*, 13:431–443, 2000.

[31] D. B. Fogel. *Evolving Artificial Intelligence*. PhD thesis, University of California, San Diego, CA, USA, 1992.

[32] L. Fogel, A. Owens, and M. Walsh. Artificial intelligence through a simulation of evolution. In *Biophysics and Cybernetic Systems*, pages 131–155. 1965.

[33] I. Gerdes, F. Klawonn, and R. Kruse. *Evolutionäre Algorithmen*. Vieweg-Verlag, Wiesbaden, Germany, 2004.

[34] F. J. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

[35] F. J. Gomez and R. Miikkulainen. 2-D pole balancing with recurrent evolutionary networks. In *Proceedings of the International Conference on Artificial Neural Networks*, Skovde, Sweden, 1998.

[36] F. J. Gomez and R. Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, 1999.

[37] F. J. Gomez and R. Miikkulainen. Robust non-linear control through neuroevolution. Technical Report AI-TR-03-303, Department of Computer Sciences, The University of Texas, Austin, USA, 2002.

[38] W. E. L. Grimson. *Object Recognition by Computer*. MIT Press, Massachusetts, London, 1990.

[39] F. Gruau. Genetic synthesis of modular neural networks. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 318–325. Morgan Kaufmann, 1993.

[40] F. Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Superieure de Lyon, Laboratoire de l'Informatique du Parallelisme, France, January 1994.

[41] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming: Proceedings of the First Annual Conference*, pages 81–89, Standford University, CA, USA, 1996. MIT Press.

[42] G. Hailu. *Towards Real Learning Robots*. PhD thesis, Technical Report 9906, Institute of Computer Science and Applied Mathematics, Christian-Albrechts University, Kiel, Germany, October 1999.

[43] G. Hailu and G. Sommer. On amount and quality of bias in reinforcement learning. In *IEEE International Conference of System, Man, and Cybernetics, Tokyo, Japan*, pages 728–733, 1999.

[44] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[45] M. Haralick and G. Shapiro. *Computer and Vision*, volume II. Addison-Wesley Publishing Company, 1993.

[46] C. G. Harris and M. Stephens. A combined corner and edge detector. In *4th Alvey Vision Conference*, pages 147–151, 1995.

[47] D. Hestens and G. Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics*. Springer-verlag, Berlin, Heidelberg, New York, 1987.

[48] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Massachusetts, London, 1975.

[49] M. K. Hu. Visiual pattern recognition by moments invariants. *RE Transaction on Information Theory*, 8:179–187, 1962.

[50] M. Hülse and F. Pasemann. Dynamical neural schmitt trigger for robot control. In *Proceedings of International Conference on Artificial Neural Networks (ICANN 2002)*, pages 783–788. Springer-Verlag, 2002.

[51] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Congress on Evolutionary Computation (CEC2003)*, volume 4, pages 2588–2595. IEEE Press, 2003.

[52] C. Jacob. *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, San Francisco, CA, 2001.

[53] R. Jain, R. Kasturi, and B. G. Schunck. *Machine Vision*. McGraw-Hill, New York, Tokyo, Toronto, 1995.

[54] N. Jakobi. Harnessing morphogenesis. In *Proceedings of Information Processing in Cells and Tissues*, pages 29–41, 1995.

[55] J. Jung and J. Reggia. A descriptive encoding language for evolving modular neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 519–530. Springer-Verlag, 2004.

[56] Y. Kassahun and G. Sommer. Learning and adaptation: A comparison of methods in case of navigation in an artificial robot world. Technical Report 0309, Institute of Computer Science and Applied Mathematics, Christian-Albrechts University, Kiel, Germany, November 2003.

[57] Y. Kassahun and G. Sommer. Improving learning and adaptation capability of agents. In *Proceedings of 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 472–481, Amsterdam, November 2004.

[58] Y. Kassahun and G. Sommer. Model based evolutionary object recognition system. In *Proceedings of 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 925–934, Amsterdam, November 2004.

[59] Y. Kassahun and G. Sommer. Automatic neural robot controller design using evolutionary acquisition of neural topologies. In *19. Fachgespräch Autonome Mobile Systeme (AMS 2005)*, pages 259–266, Stuttgart, Germany, December 2005.

[60] Y. Kassahun and G. Sommer. Efficient reinforcement learning through evolutionary acquisition of neural topologies. In *Proceedings of the 13th European Symposium on Artificial Neural Networks (ESANN 2005)*, pages 259–266, Bruges, Belgium, April 2005.

[61] Y. Kassahun and G. Sommer. Evolution of neural networks through incremental acquisition of neural structures. Technical Report 0508, Institute of Computer Science and Applied Mathematics, Christian-Albrechts University, Kiel, Germany, June 2005.

[62] Y. Kassahun and G. Sommer. Evolutionary reinforcement learning for simulated locomotion of a robot with a two-link arm. In *Proceedings of the 9th Conference on Intelligent Autonomous Systems (IAS-9)*, 2006.

[63] H. Kimura and S. Kobayashi. Reinforcement learning for locomotion of a two-linked robot arm. In *Proceedings of the 6th European Workshop on Learning Robots*, pages 144–153, 1997.

[64] H. Kitano. Designing neural networks using genetic algoithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[65] T. Kohonen. *Self-organization and Associative Memory*. Springer-Verlag, Berlin, Heidelberg, New York, 1989.

[66] R. Kortmann. Embodied cognitive science. In *Proceedings of Robo Sapiens: the First Dutch Symposium on Embodied Intelligence*, pages 173–182. Springer Verlag, 1999.

[67] J. R. Koza. Genetic programming: A paradigm for genetically breeding population of computer programs to solve problems. Technical Report STAN-CS-90-1314, Computer Science Department, Standford University, Stanford, CA, USA, 1990.

[68] J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Massachusetts, London, 1992.

[69] A. Lindenmayer. Mathematical models for cellular interactions in development, parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

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

[71] S. Luke and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. In *Late-breaking papers of Genetic Programming 1996*. Stanford, CA, 1996.

[72] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[73] D. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–33, 1996.

[74] H. Murase and S. Nayar. Visiual learning and recognition of 3d objects from appearance. *International Journal of Computer Vision*, 14:5–24, 1995.

[75] S. Nolfi and D. Floreano. *Evolutionary Robotics. The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT Press, Massachusetts, London, 2000.

[76] S. Nolfi and D. Parisi. Growing neural networks. Technical Report PCIA-91-15, Institute of Psychology, Rome, 1991.

[77] F. Pasemann. Evolving neurocontrollers for balancing an inverted pendulum. *Network: Computation in Neural Systems*, 9:495–511, 1998.

[78] R. Pfeifer and C. Scheier. *Understanding Intelligence.* MIT Press, Massachusetts, London, 1999.

[79] I. Rechenberg. Evolutionstrategie: Optimierung Technischer Systeme nach Prinzipien des Biologischen Evolution. *Fromman-Hozlboog Verlag, Stuttgart*, 1973.

[80] R. Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[81] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:41–43, 1958.

[82] B. Rosenhahn. *Pose Estimation Revisited*. PhD thesis, Technical Report 0308, Institute of Computer Science and Applied Mathematics, Christian-Albrechts University, Kiel, Germany, September 2003.

[83] RWI. Homepage of real world interface. http://www.irobot.com/rwi/index.asp.

[84] A. Samuel. Some studies in machine learning using the game of checkers. In *Computers and Thought*. McGraw-Hill, New York, 1963.

[85] N. Saravanan and D. B. Fogel. Evolving neural control systems. *IEEE Expert*, 3:23–27, 1995.

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

[87] H. P. Schwefel. Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie. volume 26 of Interdisciplinary Systems. Birkhaeuser, Basel, 1977.

[88] H. P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, New York, 1995.

[89] H. P. Schwefel and G. Rudolph. Contemporary evolution strategies. In *Advances in Artificial Life*, pages 893–907. Springer-Verlag, 1995.

[90] B. Sendhoff and M. Kreutz. Variable encoding of modular neural networks for time series prediction. In *Congress on Evolutionary Computation (CEC'99)*, pages 259–266, 1999.

[91] J. Shaffer and R. Cannon. On the control of unstable mechanical systems. In *Proceedings of the Third Congress of the International Federation of Automatic Control*, 1966.

[92] P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[93] G. Sommer. Algebraic aspects of designing behavior based systems. In G. Sommer and J.J. Koenderink, editors, *Algebraic Frames for the Perception-Action Cycle (AFPAC 97)*, pages 1–28. Lecture Notes in Computer Science 1315, 1997.

[94] K. O. Stanley. *Efficient Evolution of Neural Networks through Complexification*. PhD thesis, Artificial Intelligence Laboratory. The University of Texas at Austin., Austin, USA, August 2004.

[95] K. O. Stanley and R. Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference*, San Francisco, CA, 2002.

[96] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

[97] R. Sutton and A. Barto. *Reinforcement Learning. An Introduction*. MIT Press, Massachusetts, London, 1998.

[98] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[99] P. Tondeur. *Introduction to Lie Groups and Transformation Groups*. Springer Lecture Notes, Berlin, Heidelberg, New York, 1965.

[100] C. Tsuchiya, H. Kimura, and S. Kobayashi. Policy learning by GA using importance sampling. In *Proceedings of 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 385–394, Amsterdam, November 2004.

[101] S. Ullman. *High-level Vision*. MIT Press, Massachusetts, London, 1996.

[102] J. Vaario, A. Onitsuka, and K. Shimohara. Formation of neural structures. In *Proceedings of the Fourth European Conference on Articial Life, ECAL97*, pages 214–223, 1997.

[103] D. Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Journal of Information and Software Technology*, 43:817–831, 2001.

[104] D. Whitley, F. Gruau, and L. Pyeatt. Cellular encoding applied to neurocontrol. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 460–467, Pittsburgh, PA, USA, 1995. Morgan Kaufmann.

[105] A. Wieland. Evolving controls for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks*, pages 667–673, 1991.

[106] E.O. Wilson. *Sociobiology*. The Belknap Press of Harvard University Press, Cambridge, Massachusettes, 1975.

[107] M. Xie. *Fundamental of Robotics: Linking Perception to Action*. World Scientific Publishing, New Jersey, London, Singapore, Hong Kong, 2003.

[108] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[109] C. T. Zahn and R. Z. Roskies. Fourier descriptors for plane closed curves. *IEEE Transactions on Computers*, 21(3):269–281, 1972.