Thesis for the degree of Master of Science in
Complex Adaptive Systems

Particle swarm optimization of artificial neural networks for autonomous robots

Mahmood Rahmani

Department of Applied Physics
Chalmers University of Technology
Göteborg, Sweden, 2008

Particle swarm optimization of artificial neural networks for autonomous robots

MAHMOOD RAHMANI

Department of Applied Physics
Chalmers University of Technology
412 96 Göteborg, Sweden
Telephone: +46 (0)31 772 1000

Chalmers reproservice
Göteborg, Sweden, 2008

Abstract

Artificial neural networks (ANNs), especially when they have feedback connections, are potentially able to produce complex dynamics, and therefore have received attention in control applications. Although ANNs are powerful, designing a network can be a difficult task, and the more complex desired dynamics are, the more difficult the design of the network becomes. Many researchers have sought to automate ANN design process using computer programs. The problem of finding the best parameter set for a network to solve a problem can be seen as a search and optimization problem. This thesis concerns a comparison of two widely used stochastic algorithms, genetic algorithms (GAs) and particle swarm optimization (PSO), applied to the problem of optimizing parameters of ANNs for a simulated autonomous robot. For this purpose, a mobile robot simulator has been developed. The neural network-based controller of the robot is optimized using GAs and PSO in order to enable the robot to accomplish complex tasks. The results show that both algorithms are able to optimize the network and solve the problem. PSO excels in smaller networks, while GAs perform better for larger networks.

Keywords : particle swarm optimization, genetic algorithm, neuroevolution, evolutionary robotics.

Acknowledgment

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I would like to thank my supervisor, Prof. Mattias Wahde, who shared his expertise and research insight with me. I owe inexpressible gratitude to my lovely wife, Fayan, for continuing to love me even through the many hours I sat at the computer ignoring her. I am also grateful to my friend Behrang Mahjani for his great support.

Contents

Chapter 1

Introduction

Autonomous mobile robots are electro mechanical systems capable of moving and carrying out complex tasks in an unstructured environment without continuous human intervention [5]. The essence of the autonomy comes from human-robot interaction and the nature of the tasks that are to be accomplished. There are many applications in which some degrees of autonomy are required. Space exploration, pipe inspection, and rescue applications are examples of efforts which are sometimes unsafe, difficult, or even impossible for human to carry out.

An autonomous robot has a processing unit, often called an (artificial) *brain*, that maps sensory information to actuator commands. The brain of a robot has to support complicated sensory-motor coordination in order to be able to perform complex tasks. An **artificial neural network (ANN)** -a simple model of the human brain- is a potentially powerful mechanism to produce complex dynamics. It can be used as a robot's brain while sensory signals are fed into the network and outputs of the network are considered as signals for actuators of the robot (e.g. [19, 17, 18, 14, 31, 47, 23]).

Although ANNs are powerful, designing a proper network can be a very tough task and the more complex a desired dynamics are, the more difficult the design of the network becomes. Many researchers (e.g. see [35, 24, 9, 1, 37, 40]) have sought to automate ANN design process by using computer programs. They have used algorithms that explore various combinations of network parameters (size, topology, connection weights, etc.) and find the most suitable networks.

Evolutionary algorithms (EAs) are optimization and search methods, based on Darwinian evolution. EAs are especially useful for finding global optima of functions which have many locally optimal solutions, because in comparison with traditional gradient-based search methods, EAs have more chances to escape from local optima. EAs are independent of gradient signals and are thus suitable for handling problems where such information is not available [53]. EAs have been frequently used to carry out various tasks regarding

optimization of ANNs, such as evolving connection weights (e.g. [42, 59, 16, 48, 54, 29]), architecture design (e.g. [20, 56, 35, 24, 25, 45, 30, 43, 23]), adaptation of learning rules (e.g. [10, 6, 7, 3]), etc. Genetic algorithms (GAs) are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology.

The evolutionary algorithm approach is not the only approach for stochastic optimization of ANNs. In 1995, Kennedy and Eberhart [33] introduced **particle swarm optimization (PSO)**, which is a stochastic population-based search method inspired by social behavior of animals such as birds and fish. It is known that PSO is also successful for optimization of ANNs and in this case it produces better results than GAs [13, 8, 44, 36, 57]. PSO excels in global search and compare to **backpropagation (BP)** algorithm, which is a very common gradient based method for training the connection weights of ANNs, PSO shows faster convergence (e.g. [26, 50, 39]). Compared to evolutionary algorithms, PSO is faster in approaching the optima but it is not able to adapt the step size of its velocity for fine tuning [2].

The term **evolutionary robotics (ER)** refers to methods that develop the body and controller of robots using an evolutionary approach. Evolutionary robotics in general and evolving *neurocontrollers* (neural network-based controllers) in particular have attracted many researchers over the last decade. The following are a few examples from pioneers of ER; Husband et al. [12] used GAs to evolve network architectures and develop a neurocontroller for a simple visually-guided robot to generate adaptive behaviors. Floreano and Mondada [46] have used GAs to evolve a neural network controller so that a miniature robot, Khepera [41], can navigate in an environment, avoid obstacles, and find light sources. They obtained the result of *homing behavior* for battery recharging as an emergent behavior. Angeline et al. [1] presented a simulated ant problem in a two-dimensional grid world in which the goal was finding the best agent that collects the most amount of food in a given period of time. They evolved an ANN brain for the ant to attain good foraging skills. Gomez et al. [23] evolved neural networks for a prey capture problem using an incremental learning approach. Gruau [24] used GAs to evolve a modular neural network for controlling a hexapod robot. Kodjabachian et al. [17] also evolved a neural controller for both simulated and real versions of a hexapod robot. In this experiment the robot learned to walk and avoid obstacles. Pasemann et al. [47] evolved a recurrent neural network for the Khepera robot in order to generate obstacle avoidance and gradient-following behaviors.

1.1 Aims

PSO is a successful algorithm for evolving artificial neural networks, particularly in order to solve problems that are non-gradient and have no explicit objective function. Numerous researchers have reported good results when using this method in various applications, but still little work has been done using PSO to develop a neurocontroller for a robot. The aim of this project is to investigate how PSO and GAs can be used to optimize a neural

network in connection with autonomous robots and compare the results obtained from both methods.

Obstacle avoidance and gradient following are frequently used in evolutionary robotics [18, 47, 17, 38, 32, 23]. In this project these tasks will be considered for the robot but in a more challenging environment in which some mobile obstacles are moving around.

Chapters 2, 3, and 4 give brief introductions to artificial neural networks, genetic algorithms, and particle swarm optimization, respectively. Chapter 5 sketches out an overview of the evolution of neural networks and evolutionary robotics. The experiments done in this project are described in chapter 6, along with the implementation of GAs and PSO algorithms and a description for the robot simulator developed in this project. The results and discussion are described in chapter 7. Finally, chapter 8 presents the conclusions and future work.

Chapter 2

Artificial neural networks

The human brain is an organ made up from a huge number of interconnected cells called *neuron*s. The number of neurons in a human brain is about 10^{11} and each neuron is, on average, connected to around ten thousand other neurons. The functions of neurons and how they interact determine the behavior of the entire brain. It is believed in neuroscience that once the individual and concerted actions of brain cells are fully understood, the origins of the mental ability of humans will be understood [4].

The neuron, as shown in Figure 2.1, has three main parts: *dendrites*, the *body*, and an *axon*. Dendrites are information providers for neurons. They collect electrical signals from other neurons and deliver them to the cell body. The body adds signals and if they exceed a threshold, the axon will be activated. The axon of each neuron is connected to dendrites of some other neurons. The tail of an axon branches into a number of *synapses*. Synapses release chemical substances, namely *neurotransmitters*, in response to the axonal signal. Neurotransmitters generate electrical signals into the dendrite [4]. The effect of neurons on other neurons through synaptic conjunction determines the behavior of the whole network, and it is thought that the process of *learning* is related to varying these synaptic effects [27].

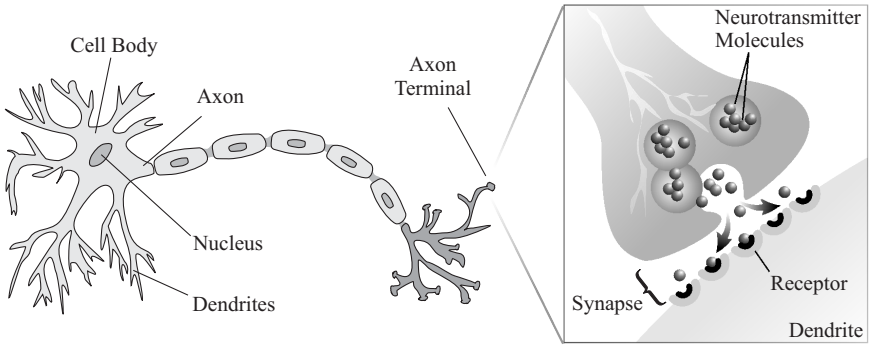

Figure 2.1: a sketch of a neuron presenting dendrites, axon, and some synaptic connections.

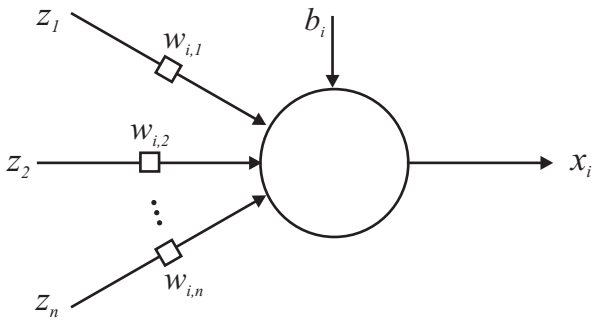

Figure 2.2: The artificial neuron.

2.1 Artificial neurons

An artificial neuron, also called a *neurod*, is a simplified model of a real neuron. As is the case with the real neurons, an artificial neuron has a number of inputs and one output. The neuron is modeled as a processing unit that produces an output based on its inputs. Figure 2.2 depicts the artificial neuron schematically. Although the function of a real neuron is a complex biochemical and bioelectrical reaction, it is believed that it simply adds the inputs and performs a threshold function [49].

The stimulation (s_i) of an artificial neuron (i) with n inputs is defined as the weighted sum of its inputs:

$$s_i = \sum_{j=1}^{n} w_{ij} z_j + b_i, \tag{2.1}$$

where w_{ij} resembles the strength of the synaptic connection between two neurons i and j, and z_j is the output of neuron j. A bias term b_i is sometimes added to the weighted sum.

The activation of a neuron (x_i) is defined as a function of its stimulation:

$$x_i = \sigma\left(s_i\right) = \sigma\left(\sum_{i=1}^{n} w_{ij} z_j + b_i\right). \tag{2.2}$$

The *activation function* σ is a *squashing* function. Two of the most common used activation functions are a logistic sigmoid function, $\sigma(x) = \frac{1}{1+e^{-cx}}$, and hyperbolic tangent, $\sigma(x) = \tanh(cx)$, where c is a positive constant.

2.2 Artificial neural networks

An artificial neural network is a simple model of the brain. It is a collection of interconnected artificial neurons.

Neural networks have been used to solve problems of many different kinds. Applications of neural networks include function approximation, face recognition, handwritten character recognition, speech processing, noise filtering, image compression, stock market prediction, mobile object path prediction, loan application scoring, automobile autopilot, soccer robot control, traveling salesman problem, medical diagnosis, and many others.

In order for the network to mimic a desired behavior, the parameters of the network should be optimized through the *learning process*. There are three major learning paradigms: *supervised*, *unsupervised*, and *reinforcement*. In supervised learning, there is a desired function that maps an input space to an output space, and the learning process changes the weights or topology of the network in order to make its behavior as close as possible to the desired function. Unsupervised learning is a learning process for cases in which the desired function is unknown. In this case there is usually a cost function to be optimized. Finally, reinforcement learning is used for cases where both input space and the desired function are not given, and the learner interacts with its surroundings to optimize its behavior.

2.2.1 Feedforward and recurrent networks

The topology of a network affects its behavior; hence, the way that neurons are connected to each other is important. If there exists a way to order neurons of a network in which all of them are only connected to the neurons with larger ordering number, the network is called a *feedforward* neural network (FFNN). If backward or lateral connections exist the network is called a *recurrent* neural network (RNN). The capability to have memory and generate periodical sequences are important characteristics of recurrent networks. A network in which any neuron is connected to all the others is called a *fully-recurrent* or *fully-connected* network. Figure 2.3 shows examples of the three network types.

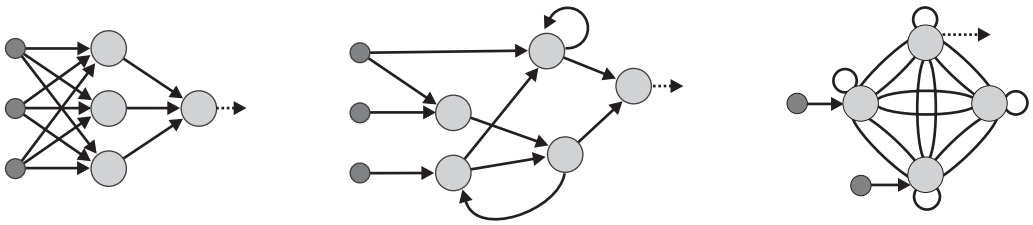

Figure 2.3: From left to right: a feedforward, a recurrent and a fully-connected network.

In recurrent networks the output of a neuron is given by:

$$\tau_i \dot{x}_i(t) + x_i(t) = \sigma\left(\sum_j w_{ij} x_j(t) + \sum_j w_{ij} I_j(t) + b_i\right), \tag{2.3}$$

where τ_i is a time constant coefficient, $x_i(t)$ is the output of neuron i-th at time t, and $I_j(t)$ is the value of input j at time t.

Backpropagation is a well-known method for optimizing weights of neural networks, first formalized by Rumelhart and McClelland [49]. It is a supervised method and requires existence of the desired output for any given input. This method is most useful for feed-forward networks.

Algorithm 1 Pseudocode for backpropagation learning algorithm

```
initialize the weights of the network (randomly)
repeat
  for all pairs of input-output of the training set do
    feed the input vector to the network and obtain the output
    error = difference between the output and the desired output
    backward pass I: compute Δw for all hidden-output connections
    backward pass II: compute Δw for all input-hidden connections
    update the weights in the network
  end for
until all pairs of input-output learned or termination criteria are met
```

Chapter 3

Genetic algorithms

In his 1859 book *On the Origin of Species by Means of Natural Selection*, Darwin formulated an idea that, despite its apparent simplicity, could explain the design of all biological organisms on Earth. This idea is known as evolutionary theory. The principle of the evolutionary theory, or *Universal Darwinism*, expresses that necessary and sufficient conditions for evolution are *variation*, *selection* and *heredity*. "*What Darwin offered the world was design out of chaos without the aid of mind.*"[15]

The term genetic algorithms (GAs) refers to a class of population-based adaptive algorithms inspired from Darwin's theory of evolution through the work of John Holland in the 1970s [28]. Each individual, or genome, in a population can be a composition of all variables of a problem, or in other words, each individual represents a particular position in the search space. Hence, GAs can be seen as a search method that is capable of finding the best set of variables that solve a problem. Major operators of genetic algorithms are *natural selection* and *crossover* (also called recombination). Individuals with advantageous qualities are more likely to be selected and become more common in the population. Because of the stochastic nature of GA, it is likely to have more than one solution for a particular problem.

3.1 Initialization

A number of individual solutions are randomly created at the beginning. Initial population size depends on the characteristics of the problem and it usually varies between several hundred to thousands of individuals. The initial population may be seeded in a particular region, where the probability of finding optimal solutions is large.

3.2 Selection

A number of individuals from the existing generation are selected to breed a new generation. The selection is typically based on fitness, thus the fitter an individual is, the more likely it

Algorithm 2 Pseudocode for a genetic algorithm

```
initialize population
repeat
  for all individuals do
    decode chromosome (create phenotype)
    evaluate individual
  end for
  if satisfactory result obtained then
    halt
  else
    place a copy of the best individual in new population (elitism)
    repeat
      select two individuals
      do crossover with probability P_c
      do mutation
      put the two new individuals in the new population
    until the new population is completed
    replace the old (parents) population with the new one (children)
  end if
until termination criteria are met
```

is to be selected. A weak individual still has a chance to be selected, and this helps to keep the diversity of the population high. Two popular selection mechanisms are *roulette-wheel selection* and *tournament selection*.

Roulette-wheel selection. Individual i with fitness f_i is selected with probability $p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$, where N is the size of the population. (Figure 3.1(left))

Tournament selection. An individual is selected if it wins a tournament among a few randomly chosen individuals. The behavior of the tournament selection depends on two parameters: tournament size (k) and the probability of selecting the best individual in the tournament (p). (Figure 3.1(right))

3.3 Crossover

Crossover is the process of varying DNA of chromosomes by exchanging some of their sections. This operation is applied for breeding offspring from selected individuals. There are different ways of applying crossover. Some of them are based on cutting points; the genome string of each parent splits from a number of cutting points, or *crossover points*, and a random swapping of the resulting sections breeds the children. The parents' string could have similar or different choices of cutting points. Accordingly, this can result in a

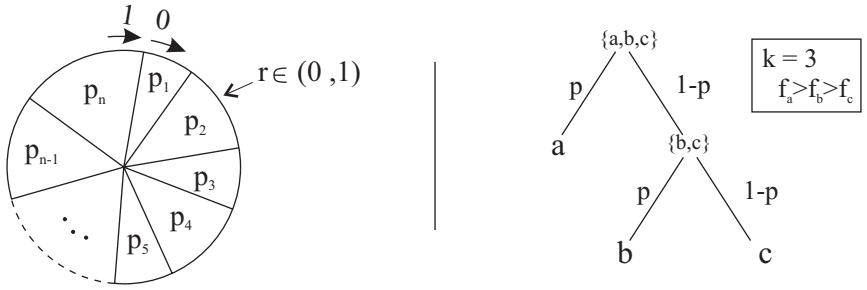

Figure 3.1: Selection methods: roulette wheel selection, p_i is the probability of selecting i-th individual and $r \in [0 \cdots 1]$ is a random number with uniform distribution (left). Tournament selection, tournament size $k = 3$, f_i is the fitness of i-th individual, and p is the probability of selecting the best individual in the tournament. In this example the probability of selecting individual a, b, or c is p, $p(1-p)$, and $(1-p)^2$ respectively (right).

change in the length of the children's string (Figure 3.2). In *uniform crossover* the number of cutting points is equal to the number of genes minus one, and each gene can be swapped with a fixed probability, normally 0.5. In some cases, such as ordered chromosomes, direct swapping does not create a valid order; hence, a particular type of crossover needs to be used that ends with a correct order. The crossover is usually applied with probability p_c (see Algorithm 2). This means some of the children may be formed only by mutation and only one parent is involved in reproduction (*asexual reproduction*). p_c is usually a number in the range of $[0.7, 0.9]$.

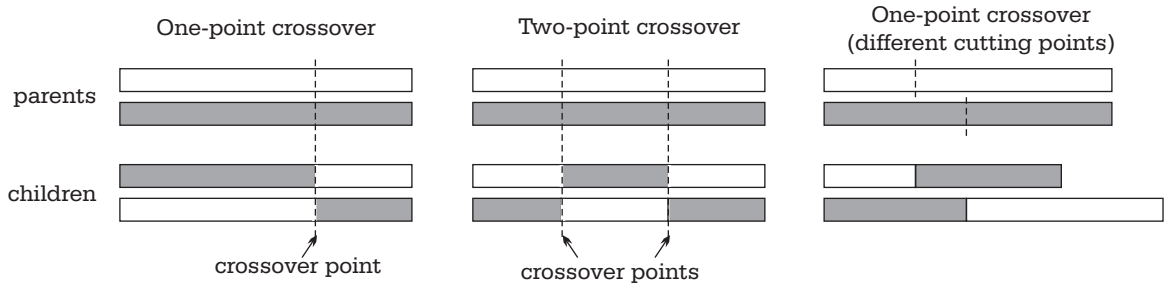

Figure 3.2: Three examples of crossover operator

3.4 Mutation

Mutation is the simplest genetic operator. It randomly flips bits in a binary string genome from zero to one or from one to zero. This operator improves the algorithm by introducing new solutions that do not exist in the population. The mutation rate has to be low to

prevent the algorithm from becoming a simple *random search*. Some types of mutation are deletion of genes, duplication of genes, inversion of a sequence of genes, and insertion of a portion of a chromosome into another chromosome.

Chapter 4

Particle swarm optimization

Particle swarm optimization (PSO) is a stochastic population-based search method inspired by the social behavior of animals such as birds and fish. This algorithm was first introduced by James Kennedy and Russell Eberhart in 1995 [33]. In PSO, each individual, called a particle, flies through the problem space and adjusts its position according to its own experience and the experience of its neighbors. A particle can fly either fast and far from the best positions to explore unknown areas (global search), or very slowly and close to a particular position (fine tune) to find better results. PSO is quite simple to implement and has few control parameters. Equations 4.1 and 4.2 are the two fundamental update rules of standard PSO:

$$v_i \leftarrow wv_i + c_1 r_1 (P_i - x_i) + c_2 r_2 (P_g - x_i), \tag{4.1}$$

$$x_i \leftarrow x_i + v_i, \tag{4.2}$$

where v_i and x_i are velocity and position vectors of particle i, respectively, P_i is the best local position found by particle i, and P_g is the best global position found in the whole population. The two parameters c_1 and c_2 are positive constants, called *learning factors*; $c1$ presents how much a particle is attracted to its best position, and c_2 is the same for the global position. Values of these two parameters vary depending on the nature of the problem but they are usually considered to be equal to 2.0. w is the inertia weight and controls the amount of freedom of the particles to explore. It has been shown, e.g. in [34] p. 342, that PSO performs better when w decays from 0.9 to 0.4 over time. r_1 and r_2 are uniform random variables providing the stochastic aspect of the algorithm.

Algorithm 3 Pseudocode for particle swarm optimization (continuous numbers)

```
initialize population (position and velocity of particles)
repeat
  evaluate all particles
  for all particle i do
    if current position of particle i, x_i, produces the best fitness in its history then
      P_i ← x_i
      if fitness of x_i is the best fitness in global then
        P_g ← x_i
      end if
    end if
  end for
  update velocity and position of particles according to the Equations 4.1 and 4.2
until termination criteria are met
```

PSO has many variations. the position of a particle can be limited to a range, e.g. $[x_{\min}, x_{\max}]$. The parameter $v_{\max}$ can be defined to limit the maximum amount of the particles' displacement in one iteration, and it usually equals to $|x_{\max} - x_{\min}|$. One might consider different types of population topologies such as square or ring topologies (Figure 4.1) in which each particle is attracted by every other particle in its neighborhood rather than being attracted by the global best individual.

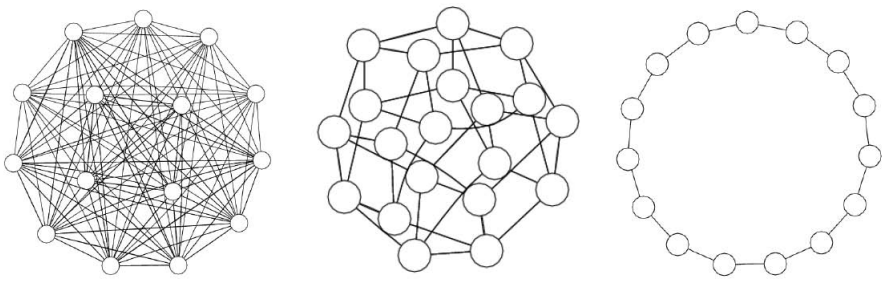

Figure 4.1: three population topologies for PSO; fully-connected (left), square (center), and ring (right).

Chapter 5

Evolution of artificial neural networks

Neural networks have been proved to be successful in mapping input patterns to particular outputs and producing complex dynamics. Finding a proper set of weights and topology for a network to produce a desired dynamic can be seen as an optimization problem. Evolutionary algorithms (EAs) have been shown to be good at optimization problem solving. The combination of these two methodologies, which is called *neuroevolution*, seems to be promising for developing hybrid computational tools which are more effective than either approach by itself. It is common to optimize connection weights of neural networks by using EAs, especially genetic algorithms (GAs). The topology and learning rules of a network have also been evolved by many researchers, but the number of these applications is not comparable with the huge number of applications for evolving network weights. Floreano et al. [20], Yao [56], Fogel [22], and Schaffer et al. [51] provided reviews of various methods that have been established for evolutionary development of neural networks.

Constructive and destructive approaches. Two general approaches for evolving the topology of networks are *constructive* and *destructive*. In the constructive approach, a network has initially a few neurons and connections, and while the algorithm proceeds, the network grows by adding new neurons and connections. Destructive algorithms start with a large network and evolve it by removing connections and neurons (Yao [56]).

Direct and indirect coding. A designer should decide how a network is encoded in a genotype; that is, determine which characteristics of the network are coded and what type of elements (binary or floating point numbers) are used. The modeler also determines the process of constructing a phenotype from the genotype. This process is called *genotype-to-phenotype mapping*. In general, there are two coding scheme approaches: *direct* and *indirect*. In direct encoding, the genotype contains all of the detailed information required to construct a particular network, including the number of layers, number of neurons in each layer, and the weight of connections. Having this information, one can always produce an identical phenotype from its genotype; hence, this approach is called *one-to-one mapping*. Because all of the details are kept in the genome, large networks have relatively large genomes in direct coding. In contrast, the indirect coding scheme is a reduced coding that

stores either a number of parameters or a set of deterministic developmental rules needed to specify the network topology, instead of encoding all the details. For example, instead of keeping the exact value of weights, a distribution of weights might be stored. In indirect encoding, reproducing an exact phenotype from a genome is usually impossible, because the genotype denotes a set of networks (*one-to-many mapping*). Considering the genome's size, the indirect coding is more suitable for evolution of large scale networks. But, it has its own drawbacks, such as the cost of converting genotypes to phenotypes and the noisy fitness that comes from nonsingularity of phenotypes.

In developmental encoding, either the phenotype might be completely constructed and then used for evaluation (phylogenetic approach), or the phenotype might continue to develope even during the evaluation time (ontogenetic approach).

Many researchers have developed neuroevolution methods using indirect encoding. Some of those methods are introduced below:

Matrix rewriting. Kitano [35] suggested a grammatical encoding which is based on matrix rewriting. The genome is a mixture of non-terminal and terminal symbols representing the development rules (Figure 5.1). Terminal symbols expand to 2×2 matrices of 0s and 1s. Non-terminal symbols expand to 2 by 2 matrices of symbols.

Cellular encoding. In this model, proposed by Gruau [24], the genome represents a grammar tree in which each node is an instruction selected from a set of 13 instructions. The instructions provide cellular duplication, connection adding and removal, and connection weight modification. The development of a phenotype starts with a single neuron connected to an input and an output. The tree is scanned from the root and each instruction is executed on the phenotype. In comparison with Kitano's coding scheme, cellular encoding is more compact and allows repeated subnetworks to be coded efficiently (modularity). The cellular encoding is used for some common control problem such as pole balancing [25] and walking behavior [24].

Growing encoding. Nolfi et al. [45] proposed a growing encoding scheme for neural network architectures. The genotype contains instructions that control axonal growth and branching. The development of a network in this model starts from a set of individual neurons distributed over 2-dimensional space (Figure 5.2). The axons of initial neurons grow and branch in the space and whenever an axon reaches another neuron, a connection between these two neurons will be established. This method is ontogenetic.

Husband et al. [30] proposed a similar scheme, called a force field development scheme, in which dendrites of neurons grow out according to ordinary differential equations on a 2D plane. The genotype in this scheme encodes the parameters of the equations in addition to the positions of neurons and the initial directions of dendrites.

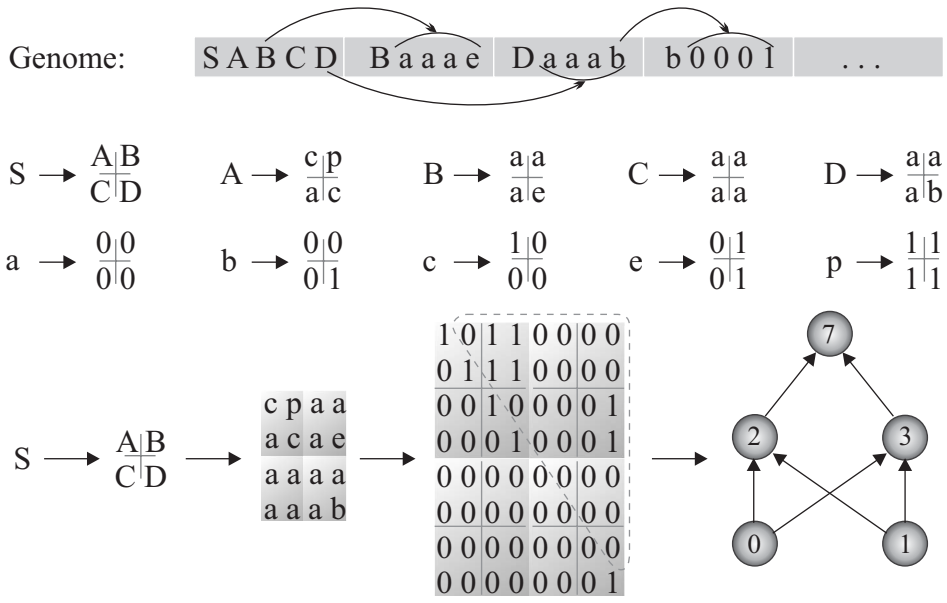

Figure 5.1: Kitano's grammar encoding. The genome consists of grammatic instructions. Starting from a non-terminal symbol S, develop S according to the rules to a 2×2 matrix of non-terminals, ABCD, then recursively apply other rules to get the adjacency matrix of the network. The strictly upper triangular matrix gives a feedforward network. (Redrawn from [35])

In all methods mentioned above, the genome encodes the whole network. There is another approach in which instead of the entire network, an individual neuron is encoded (i.e., each individual is a neuron). Moriarty and Miikkulainen [43] introduced a method—they call it *symbiotic, adaptive neuroevolution (SANE)*—for evolving individual neurons. This method evolves networks with one layer of neurons in which each neuron is connected to some input and output units. As shown in Figure 5.3, a genome in SANE stores all connection information for an individual neuron. A network is formed by random selection of a number of neurons from the population. The fitness of an individual neuron for one generation is the average fitness of all the networks containing the individual. SANE does not support evolving recurrent neural networks. Gomez and Miikkulainen [23] extended SANE by adding a subpopulation concept. In the new method, which they call *enforced subpopulation* (ESP), the individuals compete in their own subpopulation to form specialists for different aspects of the problem at hand. ESP is able to evolve recurrent networks.

One of the difficulties with evolution of networks is the *permutation* or *competing convention* problem. This problem is caused by a many-to-one genotype-phenotype mapping; i.e., the population might contain individuals with completely different genomes but (behaviorally) the same phenotype [51]. This problem causes the crossover operation to become

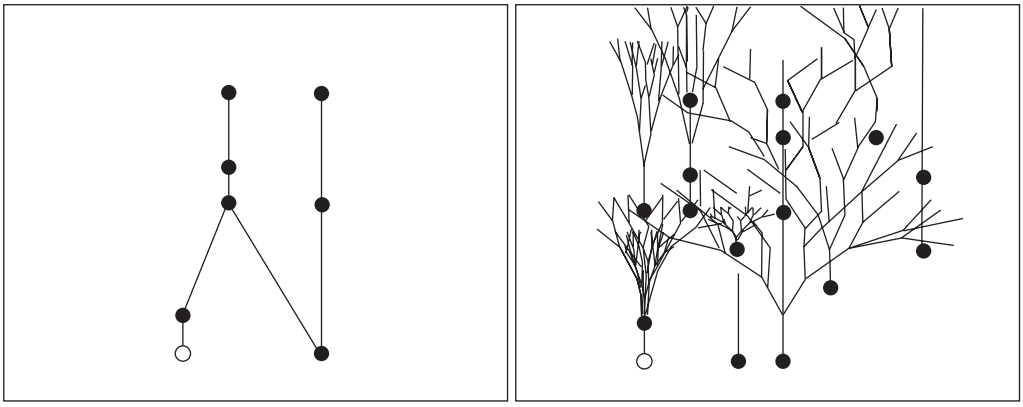

Figure 5.2: Growing encoding scheme. The neurons are positioned across the 2D plane with their grown and branched axons (right). The phenotypical network after pruning dangling axons and removing neurons that are not laid on any path between inputs and outputs (left). (After [45])

ineffective, so even two effective parents can produce very weak offspring (Figure 5.4).

Stanley and Miikkulainen [55] introduced a developmental method, namely *neuroevolution of augmenting topologies* (NEAT). It has a direct encoding, i.e., storing information about neurons and connections, along with the genetic historical record. Each new neuron is assigned an ordered, unique ID number (Figure 5.5). The inherited neurons in the offspring network keep the same ID as their parents. The ID is used to evaluate the similarity of genes, to find alignment points between genomes of different lengths, and to avoid the competing conventions problem. New individuals usually have low fitness and need time to be optimized and show better fitness. If such individuals compete with older high-fitness ones, they might be removed very soon from the population and will not have the opportunity to become developed. NEAT assigns an innovation number (which denotes how new the gene is) to permit new individuals to compete separately for reproduction. When they achieve advantages, they will be released to compete with other individuals.

5.1 Evolving ANNs using PSO

From the early days of particle swarm optimization (PSO), using this method for optimization of neural networks has been in the center of attention. Although neuroevolution means evolving networks and PSO is not known as a complete evolutionary algorithm, optimization of networks by PSO mimics evolution of networks; hence, it can be compared to other neuroevolution algorithms. Many researchers sought to use it as an alternative or complement of GAs. It is known that PSO succeeds in optimization of networks and produces better results than GAs [13, 8, 44, 36, 57]. PSO excels at global search and, when compared to a **backpropagation (BP)** algorithm, which is a very common gradient based method

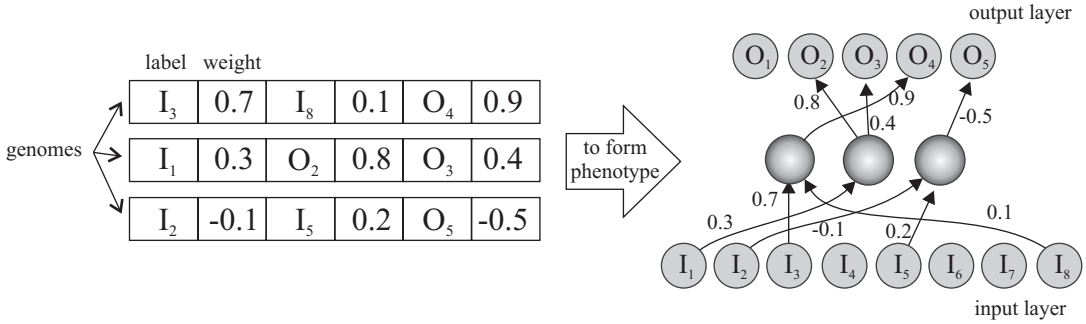

Figure 5.3: The symbiotic, adaptive neuroevolution (SANE) method. Each genome encodes connection information of an individual neuron. The labels refer to input or output units connected to the neuron. The weight fields denote weight of those connections. To construct a phenotype, a number of individual neurons are selected randomly from the population and combined with input and output units. (Redrawn from Moriarty and Miikkulainen [43])

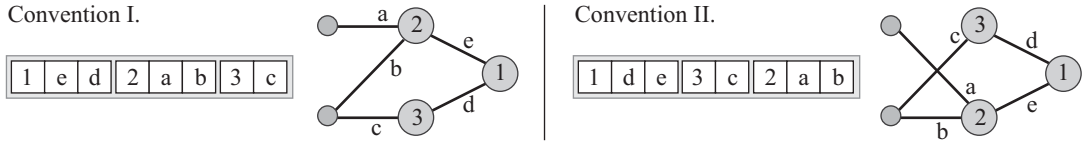

Figure 5.4: Competing conventions. Two networks with identical behavior, but completely different genomes.

for training connection weights of ANNs, PSO shows faster convergence (e.g., [26, 50, 39]). Particle swarm optimization's results present networks with good generalization[1] on the data sets. Compared to evolutionary algorithms, PSO is faster at getting close to optima, but it is not able to adapt its velocity step sizes for fine tuning [2].

5.2 Evolutionary robotics and neuroevolution

Evolutionary robotics (ER) refers to methods that develop the body and controller of robots using an evolutionary approach. Evolutionary robotics in general and evolving *neurocontrollers* (neural-network-based controllers) in particular have attracted many researchers over the last decade. The following are a few examples of pioneering research in ER: Husband et al. [12] used GAs to evolve recurrent network architectures and develop a neurocontroller for a simple visually-guided robot which generates adaptive behaviors.

[1]Generalization means a trained network could classify data from the same class as the learning data that it has never seen before. This is one of the major advantages of neural networks.

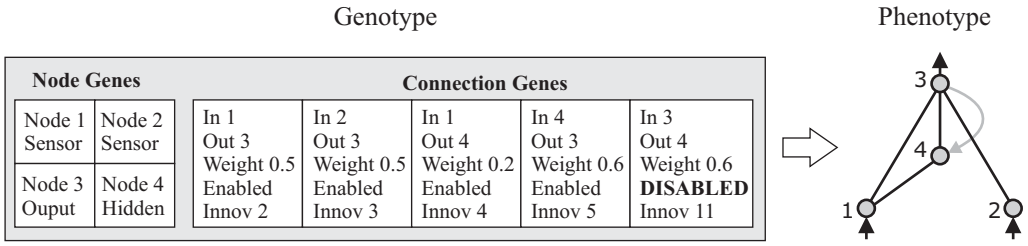

Figure 5.5: The neuroevolution of augmenting topologies (NEAT) algorithm. The Genes can be disabled or enabled. A gene gets an innovation number that shows its origin. (Redrawn from [55])

Their result also indicate that applying noise during evolution helps the networks to operate efficiently. Floreano and Mondada described in [18, 19, 46] how they have used genetic algorithms to evolve a neural network controller so that the Khepera robot can navigate in an environment, avoid obstacles, and find light sources. They obtained the result of *homing behavior* for battery recharging as an emergent behavior. Angeline et al. [1] presented a simulated ant problem in a two-dimensional grid world in which the goal is to find the best agent that collects the most amount of food in a given period of time. They evolved an RNN brain for the ant to attain good foraging skills. Gomez et al. [23] evolved neural networks for a prey capture problem using incremental learning approach. Kodjabachian et al. [17] also evolved a neural controller for both simulated and real versions of a hexapod robot. In this experiment the robot learned to walk and avoid obstacles. Pasemann et al. [47] evolved a recurrent neural network for the Khepera robot in order to generate obstacle avoidance and gradient-following behaviors.

Chapter 6

Implementation and experiments

The aim of this project is to study and compare GAs and PSO, which are used to optimize a neurocontroller for a simulated robot. To achieve this the following tasks must be carried out: constructing a robot simulation, encoding the parameters of the network to be optimized, implementing the two algorithms, defining complex tasks for the robot in an environment, and finally running the algorithms to optimize the brain of the robot.

6.1 Jarsim, a robot simulator

A simulator is designed by the author to allow users to experiment with evolutionary robotic techniques without requiring access to a real robot. The main idea of the simulator and its models of sensors and actuators have been borrowed from a simulator called *ARSim*, written by Mattias Wahde in Matlab. Since the new simulator is written in Java and its ancestor is ARSim, it is named *Jarsim*. Jarsim enables users to create simple environments of rigid objects, wandering obstacles, and light sources (as gradient providers). In addition, one can easily assemble a two wheeled robot using different numbers of infrared sensors, light detectors, and configuration of the robot. The only thing that users need to program is the brain of the robot. One can freely choose any kind of decision making mechanism and implement it as a subclass of class Brain. Jarsim is provided with a neural network library that allows users to implement a neural-based brain. The simulator supports multiple robot simulation. Appendix A presents more architectural and implementation details of the simulator.

6.2 GA and PSO configuration

Genotype encoding The coding is chosen to be direct and real-valued. The genome is a sequence of real numbers; each number corresponds to a connection weight of the network. The bias weights, the constant coefficient c of the activation function $\tanh(cx)$, and the time constant coefficient τ of Equation 2.3 are also included in the genome. All the neurons of the network use the same activation function and the same τ (i.e., $\forall i \ni \tau_i = \tau$).

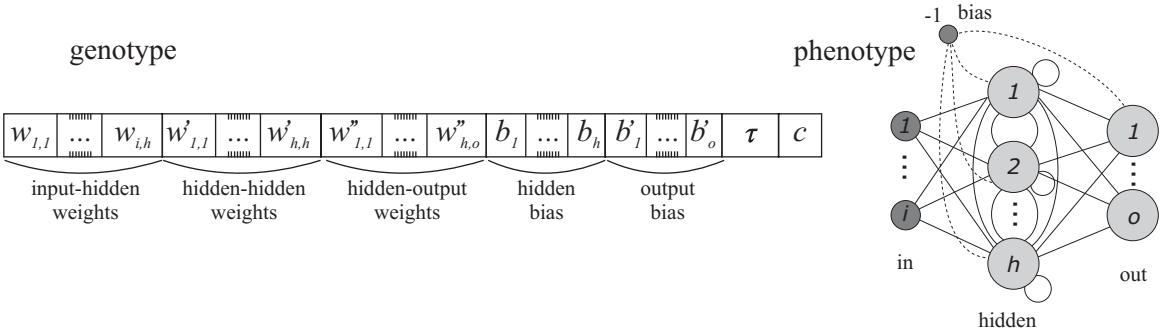

Figure 6.1: The genotype and phenotype used for the evolution of the neurocontroller of the simulated autonomous robot. τ is the time constant coefficient in Equation 2.1, and c is the constant coefficient in activation function $\sigma(x) = \tanh(cx)$

.

Particle structure. Since the genome is a vector of real values, it can be directly used as the data structure for a particle in the PSO algorithm.

Population initialization. Weights and biases are initialized by random real values in the range [-0.6, 0.6]. τ is set to a random number in the range [Δt,1], where $\Delta t = 0.01$ is the length of one time step of the simulator. c is a random positive real number less than 10.

Crossover. As shown in Figure 6.2, the genome is divided into seven parts. One-point crossover is applied to each of the first five parts. The offspring also inherit the τ and c from their parents as following:

$$\begin{aligned} \tau' &= p\tau_1 + (1-p)\tau_2, \\ c' &= qc_1 + (1-q)c_2, \end{aligned}$$

where τ' and c' are inherited parameters, τ_i and c_i are parameters of parent i, and p and q are normal random numbers.

Neural network. The architecture of the network is fixed during evolution, and only the weights will be optimized along with the constant coefficient of the activation function. The network has three layers. The number of input and output units are considered to be provided by the task and are immutable by the algorithm; thus, number of input units is equal to the number of sensors, and the number of output units is equal to the number of actuators the robot has. In this project, the robot has two DC motors, so the number of output units is two. All the neurons have been arranged in one hidden layer and are connected to both the input and output units.

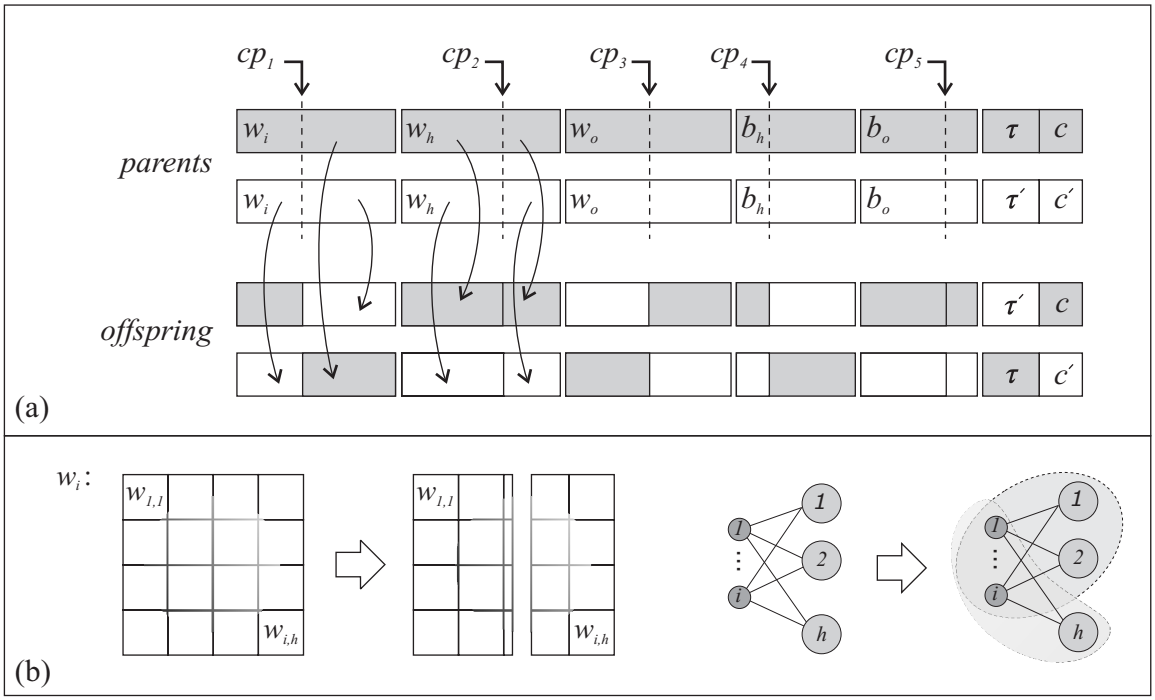

Figure 6.2: The crossover operator used for the neural network genome of Figure 6.1, where *cp* denotes a cross-point (a). A 2-dimensional matrix of w_i is cut vertically (b. left) and its phenotype after cutting (b. right)

6.3 Verification of implementations

The implementation of both algorithms needs to be verified. To achieve this, a simple function, exclusive OR (XOR), is used. XOR is a classic test problem in artificial neural networks studies. It is a binary logical operation that returns true if and only if its arguments are not the same. A fully-connected neural network with two inputs, three hidden neurons and one output is considered. Both algorithms were applied and were able to optimize the connection weights of the network to solve the XOR problem.

6.4 Experiments

For a robot to be autonomously mobile, it is necessary that it have skills to detect obstacles and try to avoid them. The complexity of the obstacle avoidance task varies from application to application and depends on the provided sensory information and complexity of the environment. Obstacle avoidance is a common case study in mobile robotics [18, 47, 17, 38, 32, 23]. In this project the same case study is used but in a more challenging environment in which there are some mobile obstacles that are moving around.

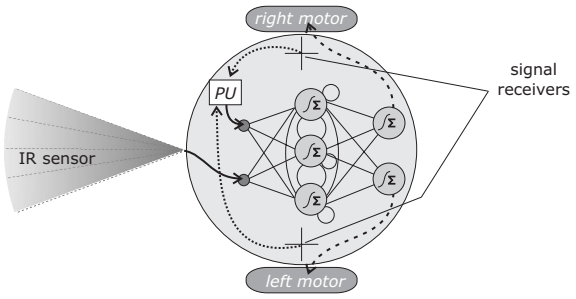

Figure 6.3: A two-wheeled robot with one infrared sensor in the front, two signal receivers (the cross signs) on the left and right hand sides, and one neural network as a brain. The PU is a hard-wired preprocessing unit which simply receives two inputs, a and b, and returns $k(a-b); k \in \Re$. The network has two input units: one is directly connected to the infrared sensor reading, and the other is connected to the output of the PU. The two outputs of the network are directly connected to motors. The bias connections of the network have been intentionally ommited in the figure to make it more readable.

Chapter 7

Results and discussion

7.1 Evolving obstacle avoidance and gradient following behaviors

In real applications, a mobile robot has to accomplish tasks while interacting with an environment made of stationary and mobile obstacles. In this experiment, the task for the simulated robot is to reach a target from an arbitrary starting point. There is a signal transmitter located at the target to help the robot find it. The obstacles are circular objects moving around with limited maximum speed that bounce off other objects like a billiard ball. The number of obstacles is fixed during a training episode. In this scenario, the robot needs to develop both obstacle avoidance and gradient following behaviors at the same time. Figure 7.1 depicts one of the configurations used in this experiment.

The simulated robot is equipped with one simulated IR-sensor on its front and two signal receivers on its left and right hand sides (Figure 6.3). A fully recurrent neural network with two inputs and two outputs is used for this problem. One input is connected to the IR-sensor. The other is connected to a hard-wired preprocessing unit that computes the difference between the readings from the two sideways sensors. Although the network can be developed in a way such that no preprocessing unit would be needed, in this project it is preferred to start from a simpler case that does use the preprocessing unit. The two outputs of the network are connected to the DC motors. The aim of this experiment is to investigate the performance of the two algorithms, GA and PSO, when they are used to optimize a neurocontroller for this robot. The number of hidden neurons varies from 2 to 30.

Fitness evaluation. The evaluation field in which the robot is examined is shown in Figure 7.1. There are four balls with random initial positions and velocities. The target, a signal transmitter, is located at the center of the field. The amount of signal (s) a sensor receives is a function of its distance to the target r: $s = \frac{1}{1+cr^2}$, where c is a constant

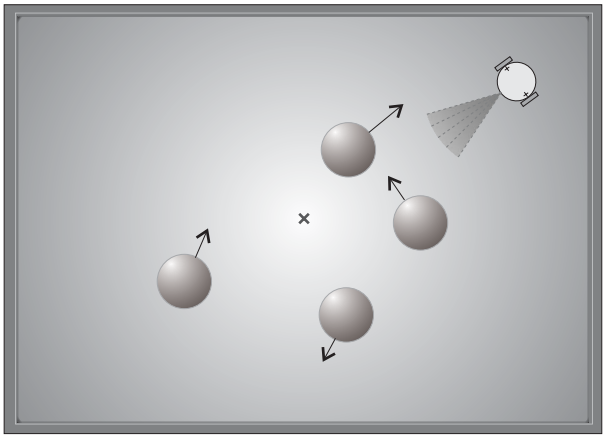

Figure 7.1: The arena used for robot-ball experiment in Jarsim. There are four balls moving around with different speeds. A signal transmitter (or one might say light source) is located at the center.

coefficient equal to $\frac{1}{l}$, and l is the diameter of the arena. The fitness function (f) is:

$$f = e^{-kd} + N, \tag{7.1}$$

where d is the distance between the robot and the target at the end of evaluation period, N denotes how many times the robot touches the target (the robot is automatically beamed to another starting position whenever it touches the target), and k is a constant coefficient equal to $\frac{2}{l}$.

Mechanical strategy is a common problem in evaluation of individuals; that is, the algorithm finds solutions which are good at a particular configuration but are not robust enough when the configuration deviates. To avoid this problem and achieve more robust and generalized solutions, one has to evaluate each individual a couple of times with different configurations and use the average fitness values instead of just a single value. In a robot-ball scenario, a set of initial configurations, say ξ, is needed:

$$\xi = \{X, X_b, V, V_b\}, b \in [1...B], \tag{7.2}$$

where X and V are initial position and velocity vectors of the robot respectively, X_b and V_b are position and velocity vectors of the b-th ball, and B is the total number of balls in the environment. Therefore, ξ determines a snapshot of the arena including positions and velocities of the balls and robot. Define C as a two dimensional matrix of snapshots:

$$C = \begin{bmatrix} \xi_{1,1} & \xi_{1,2} & \cdots & \xi_{1,m} \\ \xi_{2,1} & \xi_{2,2} & \cdots & \xi_{2,m} \\ \vdots & \vdots & \cdots & \vdots \\ \xi_{n,1} & \xi_{n,2} & \cdots & \xi_{n,m} \end{bmatrix}. \tag{7.3}$$

Each row of C has m different ξ-s which will be used for calculating one sub-fitness (f_i). The evaluator executes a sequence of tasks to calculate the fitness (Algorithm 4). By definition,

Algorithm 4 Pseudocode for the evaluation procedure in robot-ball scenario

```
for all i ∈ [1 ··· n] do
  reset the robot simulator
  for all j ∈ [1 ··· m] do
    set initial condition ξ_{i,j}
    reset internal state of the robot
    start simulator
    if robot reached the target then
      pause simulator
      f_i = f_i + 1
    end if
    if maximum simulation step is exceeded or the robot collides with an obstacle then
      f_i = f_i + e^{-kd}, (see description of Equation 7.1)
      break (and go to the next i)
    end if
  end for
end for
return the fitness F = (1/n) Σ_{i=1}^{n} f_i, (see Equation 7.4)
```

the fitness F assigned to an individual is the average of the sub-fitnesses (f_i-s):

$$F = \frac{1}{n} \sum_{i=1}^{n} f_i. \tag{7.4}$$

Parameter tuning. Each of the two algorithms should be used with its best parameter setting regarding for the problem in order to have a fair comparison between them. Table 7.1 shows the parameter tuning result for a GA. For small size networks, larger crossover rates perform better, and for the larger networks, smaller crossover rates, e.g. $P_c = 0.1$, produce better results. The best mutation rate found for this problem was $P_m = 0.05$. When the network is quite large, P_m can also be proportional to the number of connections. In this case, $P_m = \frac{1}{N^2}$ for a fully-connected neural network with N neurons.

Parameters of PSO are also tuned and the results in Table 7.2 show that the best parameter setting is $C_1 = \frac{1}{2}$, $C_2 = \frac{1}{2}$, and w decaying from 1.4 to 0.4.

Robot-ball experiment. Both algorithms with their best tuned parameters now are used to tackle the robot-ball problem. This time each configuration has been examined 50 times to take the average. Figure 7.2 depicts schematically all the loops happening in one run of the experiment.

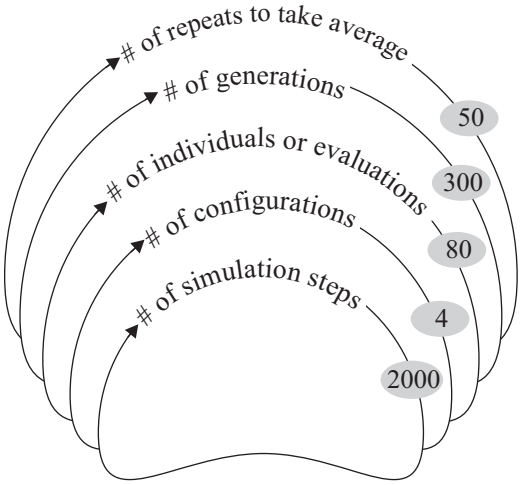

Figure 7.2: A scheme for inner loops of the optimization procedure in the robot-ball scenario. The given numbers denote arbitrary examples of the parameters. Assuming these values and one decision per simulation step, the total number of decision made (invoking the neural network) is *roughly* $50 \times 300 \times 80 \times 4 \times 2000 = 9.6 \times 10^9$. This number is just an estimation which might vary from the exact number for two reasons. First, the robot might reach the target and, according to the evaluation procedure, is allowed to examine more configurations, so the number of examined configuration might differ from 4. Second, the robot might collide before simulation step 2000, and in this case the number of passed simulation steps is less than 2000.

Table 7.1: GA parameter tuning for robot-ball problem (over 10 runs). P_c: crossover rate, P_m: mutation rate, and N: number of neurons

		$N = 2$	$N = 9$	$N = 15$	$N = 30$
$P_c = 0.7$	$P_m = 0.01$	4.11	6.06	4.87	3.48
$P_c = 0.5$	$P_m = 0.05$	**4.79**	5.58	6.11	3.60
$P_c = 0.5$	$P_m = 0.10$	4.29	6.10	5.63	3.43
$P_c = 0.1$	$P_m = 0.20$	4.03	6.30	6.16	3.50
$P_c = 0.1$	$P_m = 0.05$	4.01	**6.48**	**6.42**	**4.69**
$P_c = 0.1$	$P_m = \frac{1}{N^2}$	4.30	5.66	5.31	4.38

Table 7.2: PSO parameter tuning for robot-ball problem (over 10 runs).

C_1	C_2	$w = 0.0$	$w = 0.4$	$w = 0.9$	w decays [1.4 0.4]
	0.5	4.73	5.14	4.20	**5.58**
0.5	1.5	4.29	4.20	4.64	4.98
	2.0	4.07	3.84	5.01	5.04
	0.5	4.51	4.82	4.00	5.42
1.5	1.5	4.07	4.26	3.65	4.44
	2.0	4.57	4.18	3.68	5.20
	0.5	3.70	4.68	4.20	4.96
2.0	1.5	4.30	4.09	4.77	5.15
	2.0	4.20	4.45	4.35	5.15

The results (Figure 7.3) demonstrate that PSO has better performance for small-size networks (in this experiment where the number of connections is less than 100) compared to the GA. For medium-sized networks, which have between 100 and 500 connections, the fitness function for the GA grows rapidly, but the best fitness for PSO can catch up in long run. For large scale networks, neither of these two algorithms exhibit a phenomenal advantage to the other. The larger a network is, the larger the problem space becomes, so restricting the number of learning epochs to a low number (relative to the difficulty of the problem) prevents algorithms from finding a good solution.

7.2 Discussion and suggestions

Multiple evaluations. Evaluation of the behavior of a robot in an environment is a challenging task. A robot can simply learn to carry out a task for a particular initial condition but may fail if the initial condition deviates slightly. This special case solution, known as *mechanical strategy*, can be avoided by multiple evaluations in which a different configuration is considered for each run. The number of different configurations depends on the problem. While choosing a small number does not avoid the problem, using a large number makes the process very slow. Experience shows that using four to six different configurations is enough for the problem of evolving an autonomous robot in a dynamic environment.

Multiple runs. Adaptation is a stochastic process. The path that an algorithm takes to improve performance can be very different, depending on the initial conditions and other variables of the training process. There are many points in the fitness landscape that trap the algorithm. In order to compare two stochastic algorithms, it is important to have multiple runs, and use the average of many runs to have a more reliable comparison.

Table 7.3: Parameters of GA, PSO, and ANN used for robot-ball problem

general parameters	
population size	20
generations	300
GA	
selection	roulette wheel
mutation rate	$\{0.05, 0.10, 0.05, 0.05, 0.05\}$
crossover rate	$\{0.50, 0.20, 0.10, 0.10, 0.10\}$
elitism	2
PSO	
C_1	0.5
C_2	0.5
w	decay from 1.4 to 0.4
ANN	
# of input nodes	2
# of hidden neurons	$\{2, 5, 10, 20, 30\}$
# of output neurons	2
τ range	$[0.01, 1.0]$
c range	$[0.1, 10.0]$
weight range	$[-0.6, 0.6]$
activation function	$\tanh(cx)$

Evaluation period length. In the evaluation of a robot in a field, one important issue is the evaluation time, which is the length of time that the robot is allowed to act in the field. There are different ways to assign this time interval such as a fixed time interval, or an incremental interval starting from a small initial value. The latter option has some benefits. For the initial population, which might have many time-consuming, useless individuals, a short evaluation time avoids wasting time, and hence speeds up the algorithm. On the other hand, starting from a short evaluation time has its own drawback, which is observed for some fitness functions in this project. The problem is that the best individuals in this case would be the ones who act best in the short term, and they do not necessarily completely accomplish the task during extended evaluation time. The reason is that the potentially good individuals might not have the chance to achieve a significantly higher fitness in short time. The fitness function for the robot-ball problem is an example of this type of fitness function. Depending on the task, one needs to specify an initial value for the evaluation period length intuitively, and increase it during evolution.

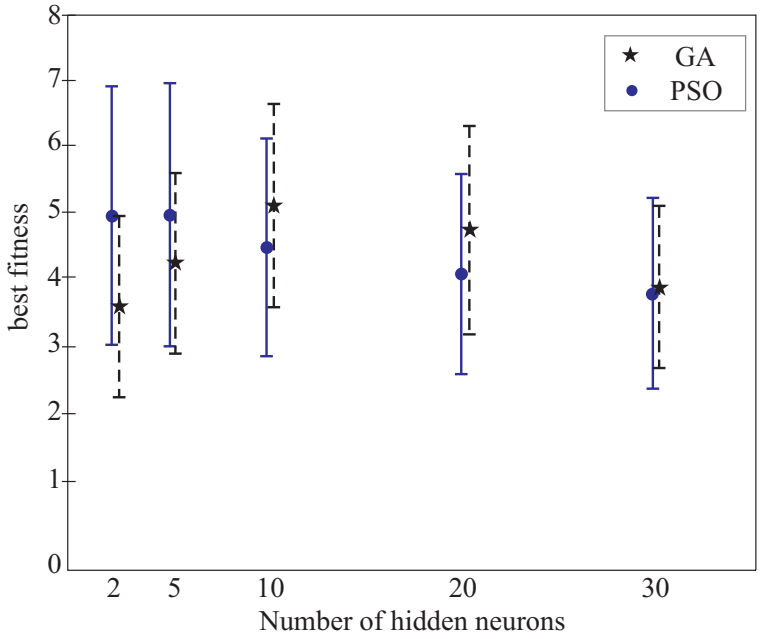

Figure 7.3: Comparison of the best fitnesses for a GA and PSO applied on the robot-ball problem for small, medium, and large networks. PSO shows better performance for small networks compared to the GAs. For medium sized networks, the fitness function for the GA grows rapidly, but the best fitness for PSO can reach the same level in long run. For large scale networks, neither of these two algorithms exhibit a phenomenal advantage to the other.

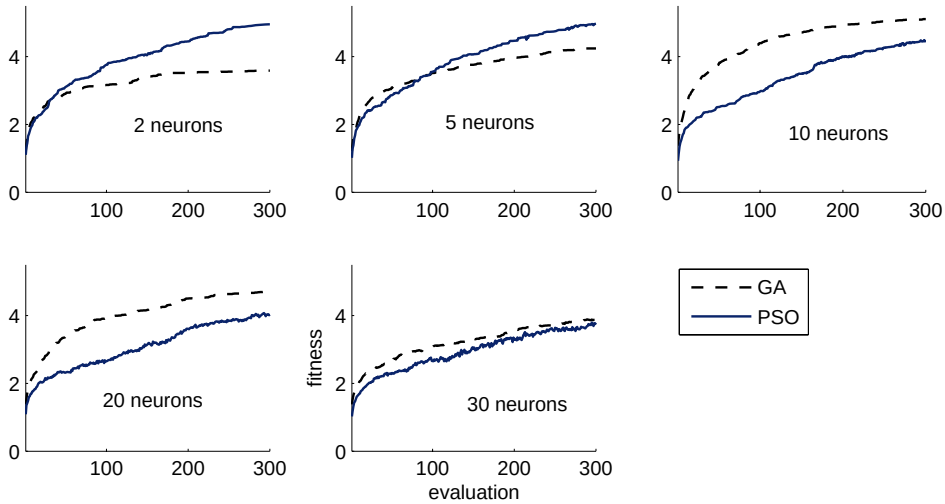

Figure 7.4: Comparison of the best fitnesses during evolution of the GA and PSO applied to the robot-ball problem for small, medium, and large networks.

Chapter 8

Conclusions and future work

The goal of this project was to compare genetic algorithms and particle swarm optimization while they are used to optimize connection weights of neural networks for a simulated autonomous robot. A simulator of a wheeled robot has been developed in this thesis. The brain of the robot is a recurrent neural network, and the aim of the robot is to learn how to accomplish a complex task in a dynamic environment. Connection weights of the network are optimized by both algorithms. Both the GA and PSO are capable of optimizing the neurocontroller. For smaller networks PSO produces better result than the GA, while the GA is better at optimizing larger networks.

8.1 Future work

The human brain can be seen as a *modular* neural network. The principle of modular design might result in a better network, especially when the network is used by a robot to accomplish a *composite* task. Although many genotype encoding schemas for neural networks have been introduced by researchers, small number of them support modularity, and even those methods are not efficient solutions for encoding large-scale modular networks. Developing a genotype encoding scheme that supports modularity for large neural networks could be an interesting idea to follow.

It is known that some control problems that can not be solved by an evolving network of conventional sigmoid neurons can be solved by using a spiking neuron model, or sometimes the spiking model can provide less complex solutions. One idea is to include various neuron models during evolution of the network.

Appendix A

The robot simulator

A.1 Modeling an infrared proximity sensor

The basic idea of infrared proximity sensors is to emit infrared light and receive the reflected light from obstacles. If an object exists in the intersection area of view of the emitter and the detector, reflected radiation from the object will be sensed by the detector. Therefore, the object is detected. In simulated IR sensors, a number of rays can be considered for the emitter (see Figure A.1-a), and if one or more of the rays intersect the object, the value of the sensor reading is simply the average of ray readings. The simulated IR sensor for this simulator has an identical emitter and detector which are installed at the same position and same direction. So, view areas of both the emitter and detector are completely overlapped, and this means the detector receives the reflected radiation if and only if a ray from the emitter intersects an object. Reading the value of each ray depends on its relative angle κ and the distance d_i between absolute position of the corresponding sensor and the closest intersecting point of the ray with any obstacle in front of it. In case there is no object inside the effective range of the sensor, the value of d_i will be considered too large, and this particular ray reads zero. The reading value (r) of such a sensor equals the average reading of its rays (r_i-s).

$$r_i = \cos(\kappa)\left(\frac{c_1}{d_i^2} + c_2\right), \tag{A.1}$$

where c_1 and c_2 are constants equal to 0.03 and 0.1 respectively.

$$r = \frac{\sum_{i=1}^{n} r_i}{n}. \tag{A.2}$$

A.2 Modeling a DC motor

A DC motor is an electro mechanical device that works by converting electric power into mechanical work. Table A.1 enumerates constant parameters of a DC motor and their default values in the Jarsim simulator.

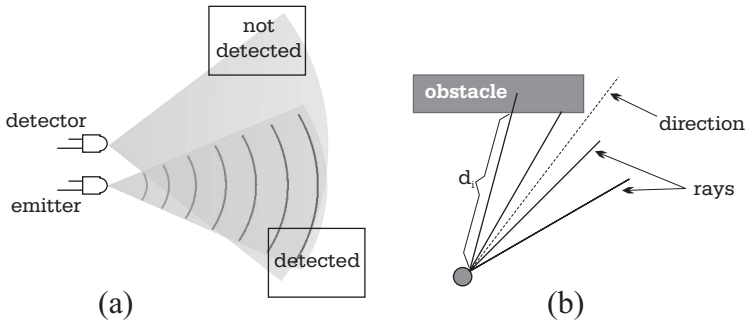

Figure A.1: infrared proximity sensor. The original technique for building an infrared proximity sensor using two identical LEDs (one IR emitter and one detector)(a), a simulated version of an infrared proximity sensor with four rays (b).

Table A.1: Constants of a DC motor

Variable	Name	Default value (in Jarsim)
k_t	Torque constant	0.0333
k_e	Back EMF constant	0.0333
r_a	Armature resistance	0.62
f_c	Coulomb friction	0.008
f_v	Viscous friction	0.02
T	Maximum torque	1.0
V	Maximum voltage	12
g_r	Gear ratio	2.0
g_e	Gear efficiency	1.0

With these constants values, the torque (τ) of the motor for a given signal ($s \in [-1 \cdots 1]$) can be calculated as follows:

$$\begin{aligned} \omega_s &= g_r \omega_a, \\ \tau &= \frac{k_t}{r_a} \left(sV - k_e \omega_s \right), \end{aligned} \tag{A.3}$$

where ω_a and ω_s are the *angular velocity* of the *axis* and the *shaft* respectively.

Applying friction. The coulombic and viscous friction should be subtracted from τ depending on the values of ω_s and τ itself (Table A.2).

Applying gear effects. τ is multiplied by the gear ratio and the gear efficiency, and finally minimum and maximum limitations of the torque are applied:

$$\tau = \min \left[\max \left(\tau \, \mathrm{g_e} \, \mathrm{g_r}, -\mathrm{T} \right), \mathrm{T} \right]. \tag{A.4}$$

Table A.2: Computing friction. $\epsilon = 10^{-6}$.

if $\lvert\omega_s\rvert > \epsilon$	$\tau = \tau - \text{sign}(\omega_a) f_c - \omega_s f_v$
if $\lvert\omega_s\rvert \leq \epsilon$ & $\lvert\tau\rvert \geq f_c$	$\tau = \tau - \text{sign}(\omega_a) f_c$
if $\lvert\omega_s\rvert \leq \epsilon$ & $\lvert\tau\rvert < f_c$	$\tau = 0$

A.3 Software architecture of Jarsim

For software to be flexible, extendable, and maintainable, advanced software design issues need to be considered. Jarsim is designed using object oriented methodology and utilizing architectural design patterns. Figure A.2 depicts a high level class diagram of the simulator. The main class of the application is called Jarsim, which is a container for all simulation objects. There are two types of objects in general, mobile objects and stationary objects. The robot is inherited from the mobile object and is composed of three classes: Brain, Sensor, and Motor. Three kinds of sensors are implemented: infrared proximity sensor, odometer, and compass (only IRSensor is used for the experiments in this project).

Jarsim has been written in Java and is easy to use with a quite simple API. Initially, the user creates an object of the class Robot and an arbitrary number of objects from classes Sensor and Motor, then adds these sensors and motors to the robot in proper positions. Finally, the user adds the robot to an instance of Jarsim. By varying the number of sensors and motors and their parameters and relative positions to the robot, one can construct different types of robots. This can be done also by the program itself at run-time; hence, it makes Jarsim a suitable platform for evolving both body and brain of the robot simultaneously.

The Brain class is designed to be abstract, so the user is able to implement his own version of the brain, which could be a neural-network-based brain (Figure A.3), a decision-tree-based brain, or any other kind of decision making process.

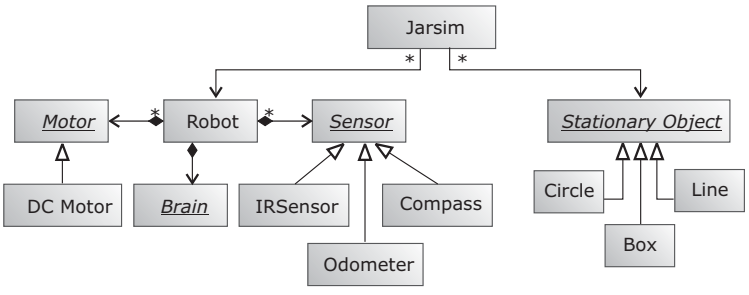

Figure A.2: high-level class diagram of Jarsim.

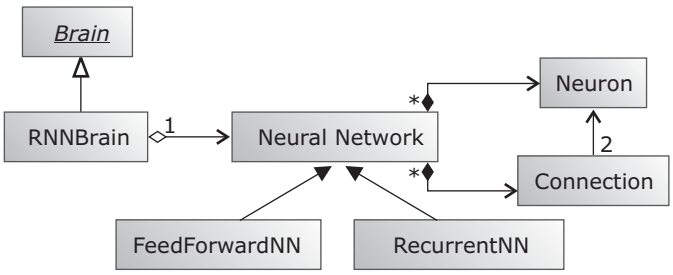

Figure A.3: a number of classes that make up the neural-based brain. The class Brain is an abstract class which here is realized by the class RNNBrain. RNNBrain aggregates a neural network, which could be a feedforward or recurrent neural network. One can also implement any other model of behavior (e.g. decision tree, state machine, etc.) by creating a subclass of Brain.

The two main implementation parts of the optimization process, algorithm and evaluator, are shown in Figure A.4. The classes PopulationBasedAlgorithm and Evaluator provide abstraction layers for the two parts. The class PopulationBasedAlgorithm can be PSO, GA, or potentially any other population-based algorithm. The Evaluator can be any optimization problem ranging from the XOR-problem to complicated robot control problems. Both PSO and GA are designed to be general, and one needs to implement them for his or her own problem by introducing the structure of particles/genome. The class GA collaborates with class GeneManipulator in order to manipulate the population of individuals. Having this level of abstraction provides enough flexibility for choosing any arbitrary pair of algorithm-evaluator (or algorithm-problem) to investigate.

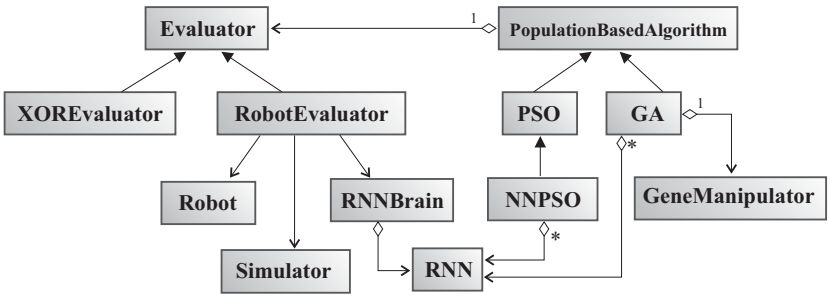

Figure A.4: Algorithm-evaluator class diagram (see the text for the description).

Bibliography

[1] P. J. Angeline, G. M. Saunders, and J. B. Pollack, *An evolutionary algorithm that constructs recurrent neural networks*, Neural Networks, IEEE, vol. 5, no. 1, pp. 54-65, 1994

[2] P. J. Angeline, *Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences*, pp. 601-610, 1998

[3] J. Baxter, *The evolution of learning algorithms for artificial neural networks*, in Complex Systems, Amsterdam, The Netherlands: IOS, 1992.

[4] M. F. Bear, B. Connors, and M. Paradiso, *Neuroscience: Exploring the Brain (Neuroscience)*. Lippincott Williams & Wilkins, February 2006.

[5] G. A. Bekey, *Autonomous Robots : From Biological Inspiration to Implementation and Control*, The MIT Press, 2005.

[6] Y. Bengio and S. Bengio, em Learning a synaptic learning rule, Dep. Informatique et de Recherche Operationelle, Univ. Montreal, Canada, Tech. Rep. 751, 1990.

[7] S. Bengio, Y. Bengio, J. Cloutier, and J. Gecsei, em On the optimization of a synaptic learning rule, in Preprints Conf. Optimality in Artificial and Biological Neural Networks, Univ. of Texas, Dallas, 1992.

[8] F. van den Bergh and A. Engelbrecht, *Cooperative learning in neural networks using particle swarm optimizers*, South African Computer Journal, 2000, pp. 84-90

[9] E. J. W. Boers and H. Kuiper, *Biological metaphors and the design of artificial neural networks*, Master's thesis, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands, 1992.

[10] D. J. Chalmers, em The evolution of learning: An experiment in genetic connectionism, in Proc. Connectionist Models Summer School, San Mateo, CA: Morgan Kaufmann, 1990.

[11] N. Chomsky, *Rules and Representations*, Columbia University Press, New York, 1980.

[12] D. Cliff, I. Harvey, and P. Husbands, *Incremental evolution of neural network architectures for adaptive behavior*, School of Cognitive and Computing Sciences, University of Sussex, Brighton, UK, Tech. Rep. CSRP256, 1992.

[13] A. Conradie, R. Miikkulainen, and C. Aldrich, *Adaptive control utilising neural swarming*,in Proceedings of GECCO 2002, pp. 60-67.

[14] I. Davis, *A modular neural network approach to autonomous navigation*, Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, 1996.

[15] D. C. Dennett, *Darwin's Dangerous Idea: Evolution and the Meanings of Life*, Simon & Schuster, June 1996

[16] F. A. Dill and B. C. Deer, *An exploration of genetic algorithms for the selection of connection weights in dynamical neural networks*, in Proc. IEEE 1991 National Aerospace and Electronics Conf. NAECON, vol. 3 1991.

[17] D. Filliat, J. Kodjabachian, and J. Meyer, *Incremental evolution of neural controllers for navigation in a 6-legged robot*, Connection Science, pp. 223-240, 1999.

[18] D. Floreano and F. Mondada, *Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot*, in Proceedings of the Conference on Simulation of Adaptive Behavior, 1994.

[19] D. Floreano and F. Mondada, *Evolutionary neurocontrollers for autonomous mobile robots*,Neural Networks, vol. 11, no. 7-8, pp. 1461-1478, October 1998.

[20] D. Floreano, P. Dürr, and C. Mattiussi, *Neuroevolution: from architectures to learning*, Evolutionary Intelligence, vol. 1, pp. 47-62, January 2008.

[21] J. Fodor, *The modularity of mind*, The MIT Press, Cambridge, MA, 1983.

[22] D. B. Fogel, *Evolutionary Computation: The Fossil Record*, Wiley-IEEE Press, 1998.

[23] F. Gomez and R. Mikkulainen, *Incremental evolution of complex general behavior*, vol. 5, no. AI96-248. Cambridge, MA, USA: MIT Press, January, 1997, pp. 317-342.

[24] F. Gruau, *Automatic definition of modular neural networks*, Adaptive Behaviour, vol. 3, no. 2, pp. 151-183, 1995

[25] F. Gruau, D. Whitley, and L. Pyeatt, *A comparison between cellular encoding and direct encoding for genetic neural networks*, in Genetic Programming 1996, MIT Press, 1996, pp. 81-89.

[26] V. G. Gudise and G. K. Venayagamoorthy, *Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks*, in Swarm Intelligence Symposium, Proceedings of the 2003 IEEE, 2003, pp. 110-117.

[27] G. E. Hinton, *How neural networks learn from experience*, Scientific American, vol. 267, no. 3, pp. 144-151, 1992.

[28] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan Press, 1975

[29] A. Homaifar and S. Guan, *Training weights of neural networks by genetic algorithms and messy genetic algorithms*, in Proc. 2nd IASTED Int. Symp. Expert Systems and Neural Networks, M. H. Hamza, Ed. Anaheim, CA: Acta, 1990.

[30] P. Husbands, I. Harvey, D. Cliff, and G. Miller, *The use of genetic algorithms for the development of sensorimotor control systems*, in From Perception to Action Conference, 1994, pp. 110-121.

[31] P. Husbands, I. Harvey, and D. Cliff, *Circle in the round: State space attractors for evolved sighted robots*, Robotics and Autonomous Systems, vol. 15, pp. 83-106, 1995.

[32] M. Islam, S. Terao, and K. Murase, *Incremental evolution of autonomous robots for a complex task*, in Evolvable Systems: From Biology to Hardware, ser. LNCS, Springer, pp. 182-191, 2001

[33] J. Kennedy and R. Eberhart, *Particle swarm optimization*, vol. 4, 1995, pp. 1942-1948

[34] R. C. Eberhart, Y. Shi, and J. Kennedy, *Swarm Intelligence*, (The Morgan Kaufmann Series in Artificial Intelligence). Morgan Kaufmann, 2001.

[35] H. Kitano, *Designing neural networks using genetic algorithms with graph generation system*, Complex Systems Journal, vol. 4, pp. 461-476, 1990.

[36] B. Liu, L. Wang, Y. Jin, and D. Huang, *Designing neural networks using pso-based memetic algorithm*, 2007, pp. 219-224.

[37] T. B. Ludermir and M. Carvalho, *Particle swarm optimization of neural network architectures and weights*, Hybrid Intelligent Systems, HIS 2007, pp. 336-339

[38] J.-A. Meyer, P. Husbands, and I. Harvey, *Evolutionary robotics: A survey of applications and problems*, ser. Lecture Notes in Computer Science. Springer, pp. 1-21, 1998

[39] R. Mendes, P. Cortez, M. Rocha, and J. Neves, *Particle swarms for feedforward neural network training*, in Neural Networks, IJCNN '02., pp. 1895-1899, 2002

[40] M. Mandischer, *Representation and evolution of neural networks*, in Artificial Neural Nets and Genetic Algorithms, Wien and New York, Springer, 1993.

[41] F. Mondada, E. Franzi, P. Ienne, T. Yoshikawa, and F. Miyazaki, *Mobile Robot Miniaturization: A Tool for Investigation in Control Algorithms*, Lecture Notes in Control and Information Sciences, pp. 501-13. Springer, London, 1994.

[42] D. Montana and L. Davis, *Training feedforward neural networks using genetic algorithms*, in Proc. 11th Int. Joint Conf. Artificial Intelligence. San Mateo, CA: Morgan Kaufmann, 1989.

[43] D. E. Moriarty and R. Miikkulainen, *Efficient reinforcement learning through symbiotic evolution*, Austin, TX, USA, Tech. Rep. AI94-224, 1994.

[44] U. Natarajan, V. Periasamy, and R. Saravanan, *Application of particle swarm optimisation in artificial neural network for the prediction of tool life*, The International Journal of Advanced Manufacturing Technology, vol. 31, 2007, pp. 871-876

[45] S. Nolfi and D. Parisi, *Evolution of Artificial Neural Networks*, M. A. Arbib (ed.), The Handbook of Brain Theory and Neural Networks, MIT Press, 2003, pp. 418-421

[46] S. Nolfi and D. Floreano, *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines (Intelligent Robotics and Autonomous Agents).* The MIT Press, 2004.

[47] F. Pasemann, U. Steinmetz, M. Hülse, and B. Lara, *Robot control and the evolution of modular neurodynamics*, Theory in Biosciences, vol. 120, 2001, pp. 311-326.

[48] D. L. Prados, *New learning algorithm for training multilayered neural networks that uses genetic-algorithm techniques*, Electron. Lett., vol. 28, 1992.

[49] D.E. Rumelhart and J.L McClelland (Eds.), *Parallel distributed processing*, Volume 1: Foundations. MIT Press, Cambridge, MA, 1986.

[50] J. Salerno, *Using the particle swarm optimization technique to train a recurrent neural model*, Tools with Artificial Intelligence, IEEE, pp. 45-49, 1997.

[51] J. D. Schaffer, D. Whitley, and L. J. Eshelman, *Combinations of genetic algorithms and neural networks: a survey of the state of the art*, COGANN-92, pp. 1-37, 1992.

[52] J. T. Schwartz, *The new connectionism: developing relationships between neuroscience and artificical intelligence*, The Artificial Intelligence Debate: False Starts, Real Foundations, MIT Press, pp. 123-141, 1988.

[53] R. S. Sexton, R. E. Dorsey, and J. D. Johnson, *Toward global optimization of neural networks: a comparison of the genetic algorithm and backpropagation*, Decis. Support Syst., vol. 22, no. 2, pp. 171-185, 1998.

[54] M. Srinivas and L. M. Patnaik, *Learning neural network weights using genetic algorithms-Improving performance by search-space reduction*, in Proc. IEEE Int. Joint Conf. Neural Networks, 1991.

[55] K. O. Stanley and R. Miikkulainen, *Evolving neural networks through augmenting topologies*, Evol. Comput., vol. 10, no. 1063-6560, pp. 99-127, 2002.

[56] X. Yao, *Evolving artificial neural networks*, Proceedings of the IEEE, vol. 87, no. 9, September 1999, pp. 1423-1447

[57] J. Yu, S. Wang, and L. Xi, *Evolving artificial neural networks using an improved pso and dpso*, Neurocomputing, 2008.

[58] C. Zhang, H. Shao, and Y. Li, *Particle swarm optimisation for evolving artificial neural network*, Systems, Man, and Cybernetics, IEEE, vol. 4, 2000.

[59] D. Whitley, T. Starkweather, and C. Bogart, *Genetic algorithms and neural networks: Optimizing connections and connectivity*, Parallel Comput., vol. 14, no. 3, 1990.

[60] Wikipedia, The world's fastest computer, http://en.wikipedia.org/wiki/IBM_Roadrunner, accessed 14 August 2008.