INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Evolution of Neural Networks Through Incremental Acquisition of Neural Structures

Yohannes Kassahun, Gerald Sommer

Bericht Nr. 0508
June 2005

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

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

Evolution of Neural Networks Through Incremental Acquisition of Neural Structures

Yohannes Kassahun, Gerald Sommer

Bericht Nr. 0508
June 2005

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

ABSTRACT

In this contribution we present a novel method, called Evolutionary Acquisition of Neural Topologies (EANT), of evolving the structures and weights of neural networks. The method introduces an efficient and compact genetic encoding of a neural network onto a linear genome that enables one to evaluate the network without decoding it. The method uses a meta-level evolutionary process where new structures are explored at larger time-scale and the existing structures are exploited at lower time-scale. This enables it to find minimal neural structures for solving a given learning task.

CONTENTS

1. INTRODUCTION

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 [28, 29]. The combination results in an evolutionary reinforcement learning system where each of the components of the learning system plays an important role.

Neural networks are useful for evolving the control system of an agent [39, 41, 28]. 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 (synaptic weights, architecture, etc) 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 [16, 57, 8, 40, 26, 50], in learning obstacle avoidance and navigation paths [22, 25], and in representing a value function while learning to play games without human expertise [53, 9].

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 [5, 52, 22]. 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 [41]. This property of reinforcement learning makes it important in evolutionary reinforcement learning. In reinforcement learning, an agent tries to estimate 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 drive 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 object recognition system [30].

There are many forms of evolutionary algorithms. The major ones are genetic algorithms [24, 56], genetic programming [35], evolution strategy [42, 45] and evolutionary programming [14]. 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 [11].

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 that combines the principles of neural networks, reinforcement learning and evolutionary algorithms is shown in figure 1.1. 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 will be 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.

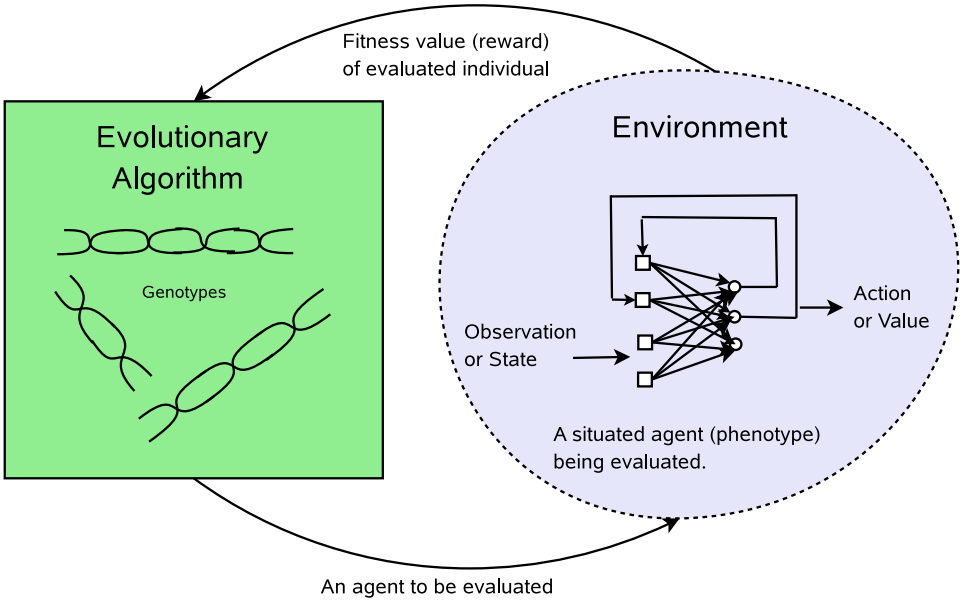


Fig. 1.1: Evolutionary reinforcement learning system. The agents, where the neural networks are embedded in, are evaluated in the environment and their fitness values are returned to the evolutionary algorithm as rewards.

1.1 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 [59]. In this section, we will give a review of evolutionary reinforcement methods that are applied to reinforcement learning tasks.

1.1.1 Evolution of Connection Weights

Wieland [58] 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.

An Evolutionary Programming (EP) to parameter optimization of feed-forward neural networks is used by Saravanan and Fogel for double pole balancing tasks [44]. The structure of the neural network especially the number of hidden units are determined a priori.

Moriarty and Miikkulainen [38] developed a method of evolving neural networks, called Symbiotic, Adaptive Neuroevolution (SANE), where the system evolves population of neurons instead of population of networks. A fully connected hidden layers of networks are formed by a combination of neurons selected randomly from population of neurons. A neuron individual receives an average fitness value of networks in which it takes part in.

The Enforced Subpopulations (ESP) [15, 16, 17] 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 is critically dependent upon the neurons to which it is connected.

Floreano and Urzelai [12, 13] 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 four different Hebbian rules and four different learning rates.

Igel [26] applied a specialized evolutionary strategy called CMA-ES [23] for evolving a fixed-topology neural network. The CMA-ES uses important concepts like *derandomization* and *cumulation*. The concept derandomization shows the deterministic 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.

1.1.2 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

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

Angeline et al. [1] 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 [50, 49] evolves both the structure and weights of neural networks using both crossover and mutation operators. It starts with networks of minimal structures and complexifies them 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.

Indirect Genetic Encoding

The methods that use indirect encoding specify 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 [34] grammar based encoding of neural networks use Lindenmayer systems (L-systems), which were introduced by Lindenmayer [37] and used 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. [47] 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 time-scale. Kitano and Sendhoff used their systems for evolution of feed-forward networks.

Gruau's Cellular Encoding (CE) method [19, 20, 21] 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.

Vaario et al. [55] 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. 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.

1.2 Contributions of the Work

The method presented in this work is closely related to the works of Angeline et al. [1] and to the works of Stanley and Miikkulainen [50, 49]. 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. But it has the following important features which makes it different from the earlier works:

1. 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. A meta-level evolutionary process where exploration of structures is executed at a larger time-scale and exploitation of existing structures is done at smaller time-scale.

2. EVOLUTIONARY ACQUISITION OF NEURAL TOPOLOGIES

Evolutionary Acquisition of Neural Topologies (EANT) is an evolutionary reinforcement learning system 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.

2.1 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 the 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. 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 to 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 2.1.

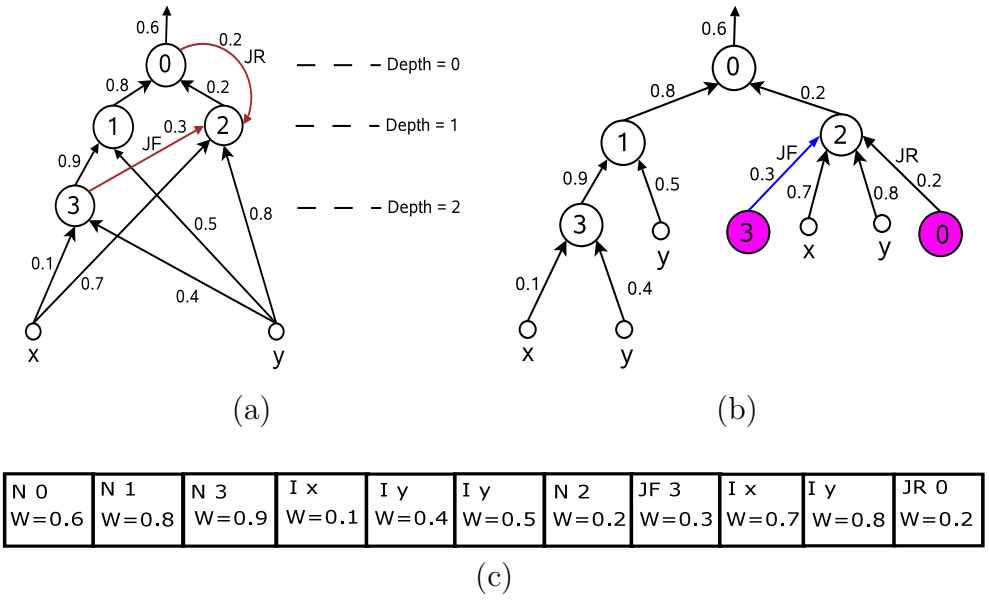


Fig. 2.1: An example of encoding a neural network using a linear genome. (a) The neural network to be encoded. It has one forward and one recurrent jumper connection. (b) The neural network interpreted as a tree structure, where the jumper connections are considered as terminals. (c) The linear genome 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 [4, 35]. 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 2.2 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.

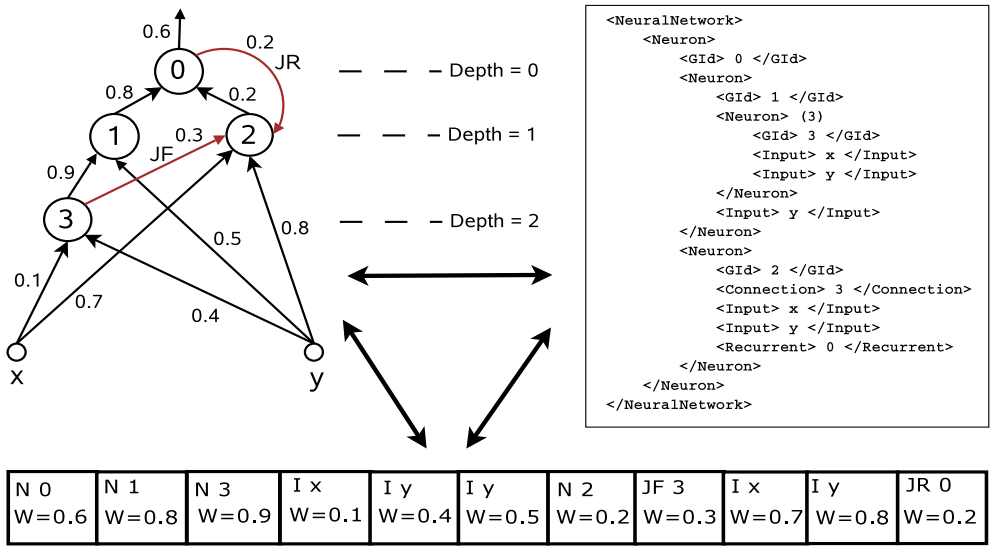


Fig. 2.2: The linear genome is equivalent to the neural network it encodes or a tree based program representing the neural network. One can generate the tree based program or the linear genome starting from the neural network or vise-versa. The tree-based program is coded in XML like commands. The commands `<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 important property 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 2.3.

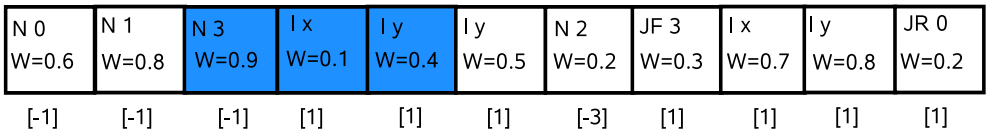


Fig. 2.3: An example of the use of assigning integer values to the nodes of the linear genome. The linear genome encodes the neural network shown in figure 2.1. The numbers in the square brackets below the linear genome show the integer values assigned to the nodes of the linear genome. Note that the sum of the integer values is *one* showing that the neural network encoded by the linear genome has only *one* output. The shaded nodes 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 [3, 27]. It is closed under structural mutation operator since every new linear genome produced by structural mutation is mapped into a valid phenotype network. 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 [49]. An example is shown in figure 2.4. The number of inputs to a neuron node which is common to the parent structures is updated using

$$n(s_1 \times s_2) = n(s_1) + n(s_2) - n(s_1 \cap s_2), \tag{2.1}$$

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. In addition to this, the genetic encoding is *scalable*. Scalability is defined by how decoding and genotype space complexity are affected by a single change in a phenotype [3, 27]. The encoding takes $O(1)$ time and space in node addition and deletion and no time is incurred during the genotype-phenotype mapping because of the equivalence of the linear genome to the neural network it encodes. Similarly, addition and deletion of node-to-node connection in phenotype will cost $O(1)$ space and time in genotype and again incurs no time in the genotype-phenotype mapping. Hence it can be said that the encoding scheme is $O(1)$ scalable with respect to the nodes and connectivity.

2.2 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 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). The first 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 neural network 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

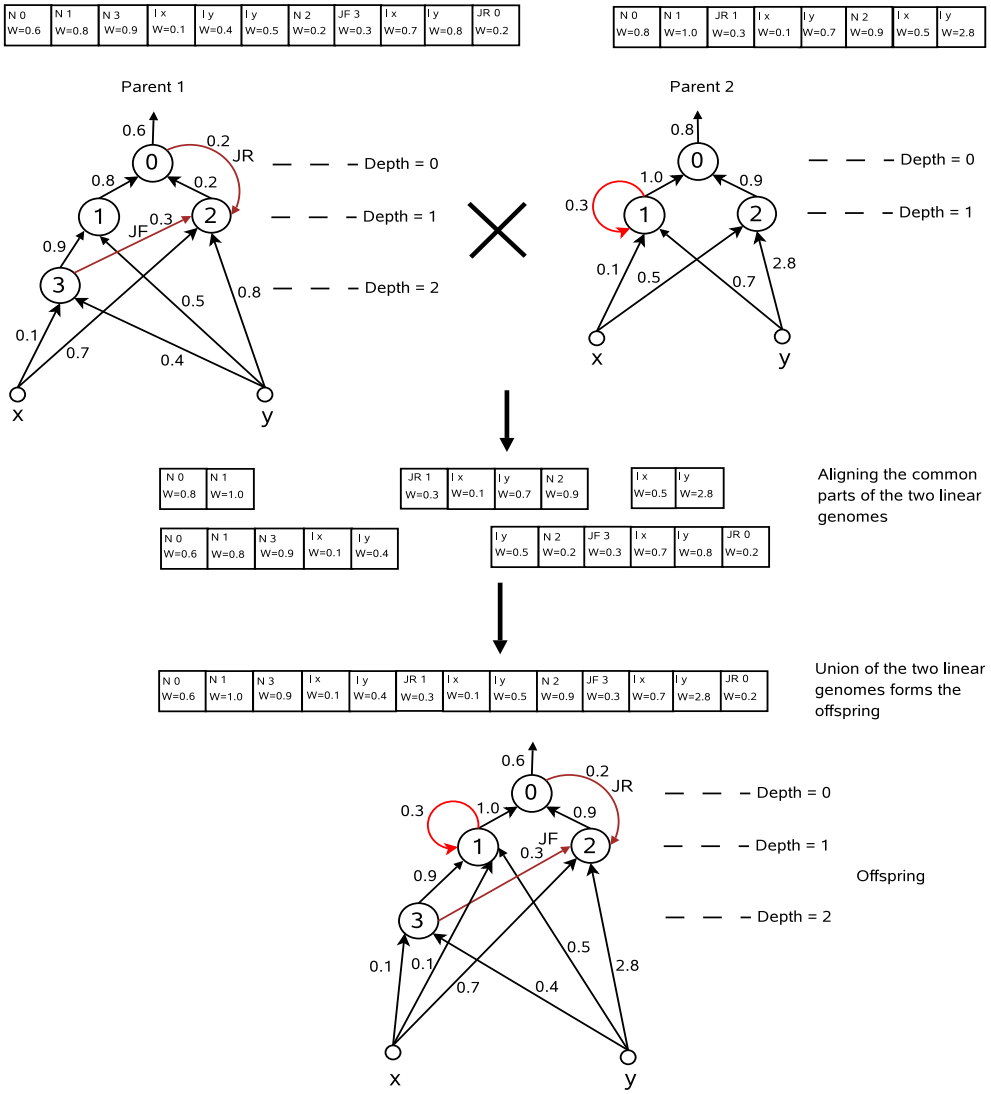


Fig. 2.4: Performing crossover between two linear genomes. The genetic encoding is closed under this type of crossover operator since the resulting linear genome maps to a valid phenotype network. The weights of the nodes of the resulting linear genomes are inherited randomly from both parents.

create a separate phenotype structure from genotype by some sort of ontological process [4]. In other words, it is not always necessary to decode the linear genome into a neural network for the purpose of evaluating it.

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

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

   $$O = g\left(\sum_{i=1}^{n} w_i\, a_i\right), \tag{2.2}$$

   where O is the result of computation for the current neuron node, v_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 2.5 shows an example of evaluating a linear genome encoding the neural network shown in figure 2.1. As can be seen from the figure, one does not need to decode the neural network in order to evaluate it.

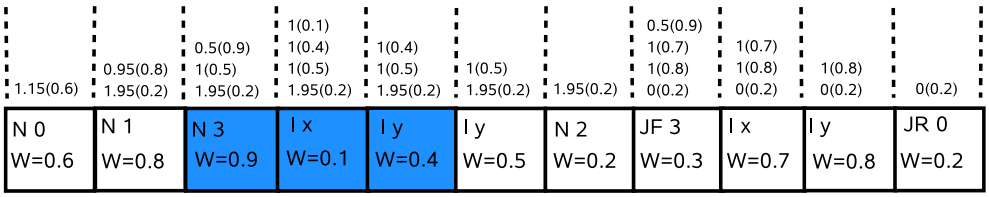


Fig. 2.5: An example of evaluating a linear genome without decoding the neural network encoded by it. The linear genome encodes the neural network shown in figure 2.1. For this example, the current values of the inputs to the neural network, x and y, are both set to 1. In the example, all neurons have 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

$$a_i(t) = g\left(\sum_{j=1}^{n_f} w_{ij}a_j(t) + \sum_{j=n_f+1}^{n} w_{ij}a_j(t-1)\right). \tag{2.3}$$

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.

2.3 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 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 [36, 4].

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 2.6 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.

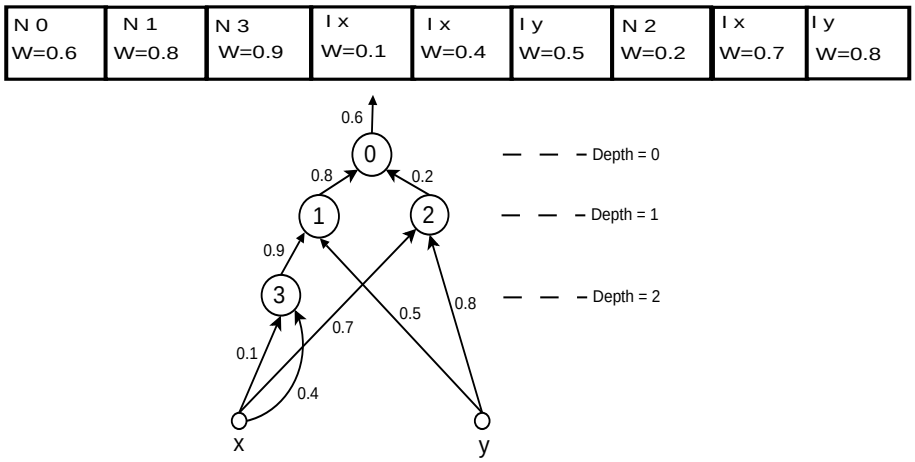


Fig. 2.6: An example of a linear genome generated by using the grow method and the neural network encoded by it. Note that the linear genome must be edited since the neural network encoded by it has a neuron with repeated inputs.

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 2.7 shows an example of a linear genome generated with full method.

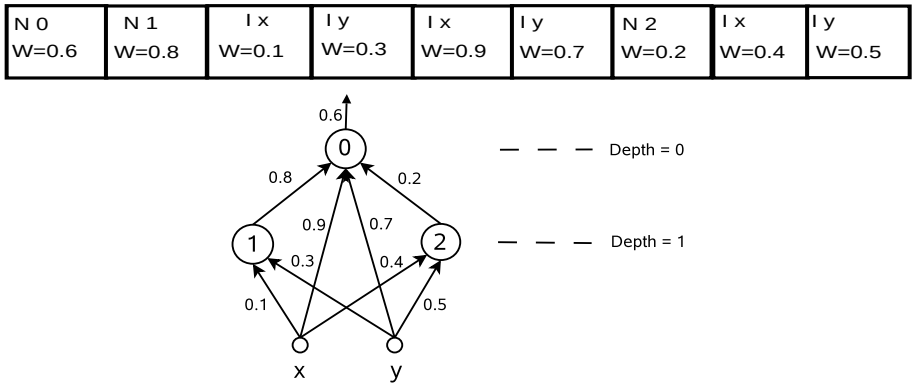


Fig. 2.7: An example of a linear genome generated by using the full method and the neural network encoded by it.

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, and hence reducing the number of evaluations necessary during optimization of the weights of the neural network. Figure 2.8 shows an example of editing a linear genome.

2.4 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 encoding a neural network. Figure 2.9 shows the case where the structural property of a neuron node `N3` is changed through the structural mutation. The neuron node losses an input `x` and gains a self-recurrent connection. In applying the structural mutation, each neuron node is tested if it is going to

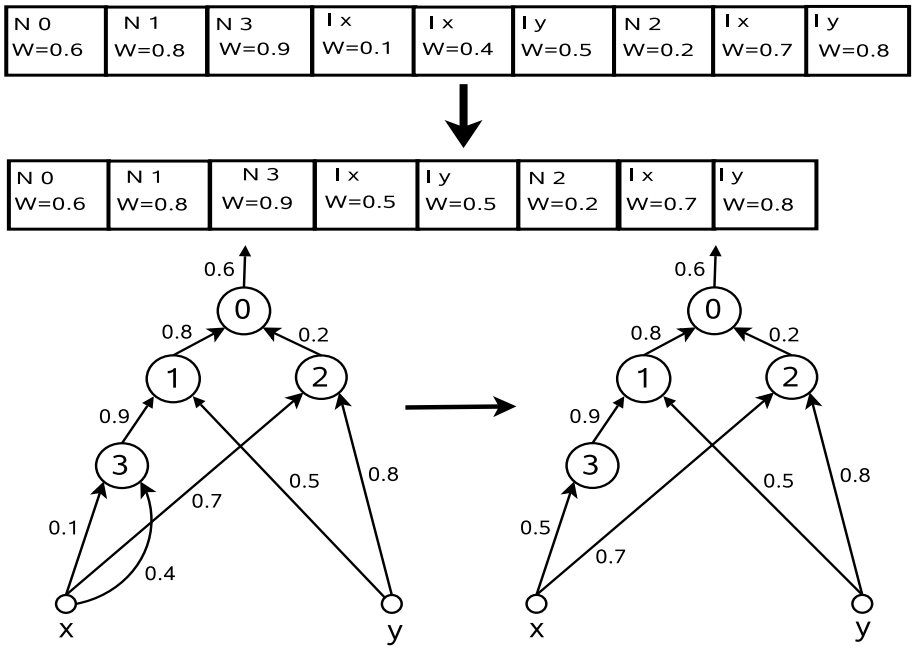


Fig. 2.8: An example of editing a linear genome. The editing replaces repeating inputs with non-repeating ones.

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, 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.

2.5 Variation Operator: Parametric Mutation

Parametric mutation is accomplished by perturbing the synaptic weights of the networks according to the uncorrelated mutation in evolution strategy or evolutionary programming [45, 11, 14]. In addition to the associated weight, each node in a linear genome encoding a neural network has an associated mutation step size or learning rate. Figure 2.10 shows a linear genome with n nodes where every node has in addition to weight a learning rate associated with it.

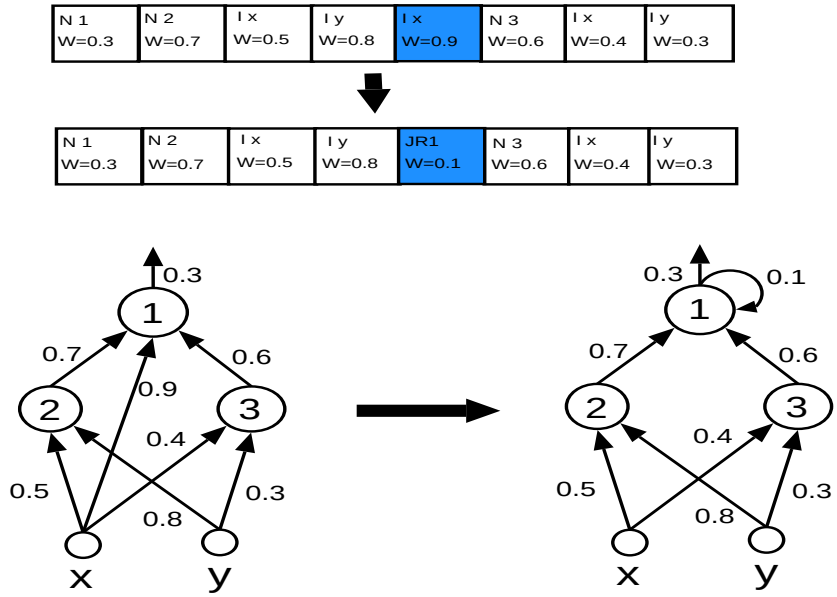


Fig. 2.9: An example of structural mutation. Note that the structural mutation deleted the input connection to `N1` and added a self-recurrent connection to it.

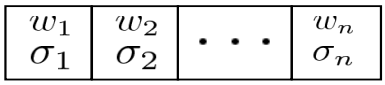


Fig. 2.10: Every node of a linear genome has in addition to weight an associated learning rate.

The mutation mechanism is specified as follows:

$$\sigma_i' = \sigma_i \, e^{\tau' \, N(0,1) + \tau \, N_i(0,1)}, \tag{2.4}$$

$$w_i' = w_i + \sigma_i \, N_i \, (0,1) \, , \tag{2.5}$$

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:

$$\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0. \tag{2.6}$$

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. Unlike genetic algorithms, evolution strategies and evolutionary programming perform search in the space of networks. Offspring created by mutation remain within a locus of similarity to their parents [1].

2. Self-adaptation of mutation step sizes of learning rates is inherent in both evolution strategies and evolutionary programming [45, 14, 2].

2.6 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 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. The initial structures are generated either with the grow or full method.

The structures that are already in the system are exploited. 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 [46], 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 certain number of generations N. This is an evolutionary process that occurs at smaller time-scale 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 2.11.

Exploration of structures is accomplished by structural mutation which is performed at larger time-scale. It is used to create new species or introduce new structures. From each of the existing structures, a new structure is

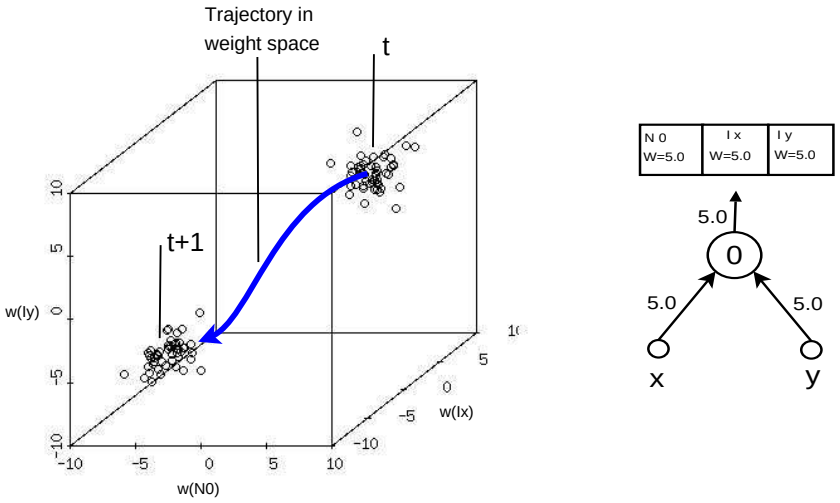


Fig. 2.11: The weight trajectory of the linear genome shown at the right side while it is being exploited. The quantities t and $t+1$ are time units with respect to the larger time-scale. The weights of the existing structures are optimized between two consecutive time units with respect to the larger time-scale. The point clouds at t and $t+1$ show populations of individuals from the same species.

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. [1].

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 2.12 summerizes 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 time-scale 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 time-scale, 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 the 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.

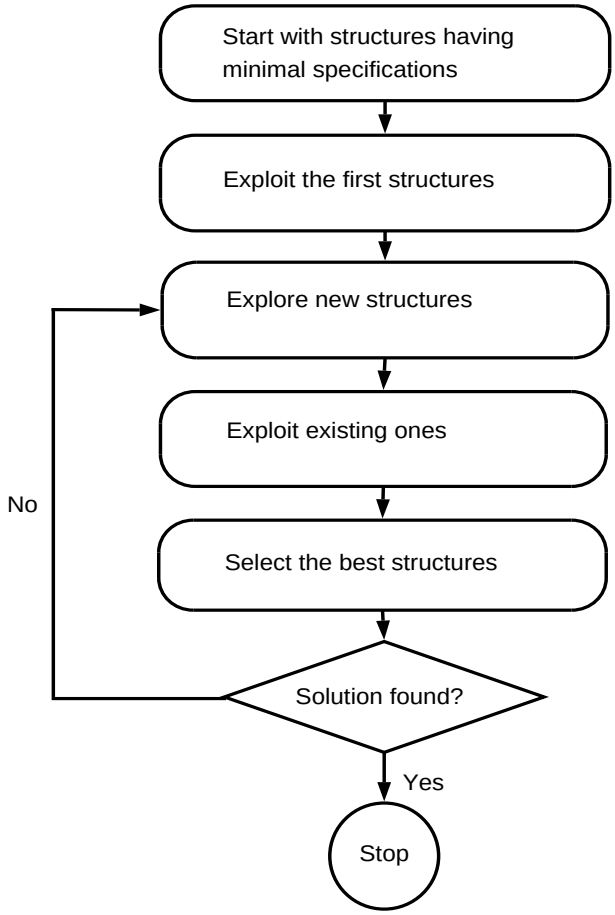


Fig. 2.12: EANT's evolutionary process at larger time-scale. Crossover does not occur between structures since they represent different species.

3. EXPERIMENTAL EVALUATION

3.1 Introduction

In this chapter the performance of EANT is examined. In the first part of the 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 part, an experiment on learning to move forward using a single legged robotic insect is considered. The third part of the experiment discusses the efficiency of EANT on the standard benchmark problem of balancing two poles attached to a moving cart. Comparison with other algorithms tested on the same problem is also given. Finally, the performance of EANT is investigated on the problem of reactive navigation with obstacle avoidance.

3.2 XOR Problem

The exclusive-OR (XOR) problem is a simple example of a data set which is not linearly separable [6]. It consists of four inputs which are divided into two classes. An example of exclusive-OR (XOR) is shown in figure 3.1. The inputs $(0, 0)$ and $(1, 1)$ belong to class C_1, while the inputs $(0, 1)$ and $(1, 0)$ belong to C_2.

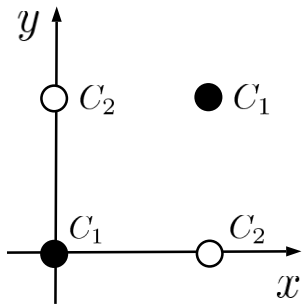


Fig. 3.1: The XOR problem. It is a simple example of a problem which is not linearly separable.

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 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.

The problem has been used to measure the performance of several other algorithms that evolve both the architecture and weights of the neural networks [34, 10, 49]. The NeuroEvolution of Augmenting Topology developed by Stanley [49] 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 or comparing the performance of the algorithm with other algorithms.

3.3 Crawling Robotic Insect

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 not. The robot was introduced and used by Tsuchiya and Kimura for reinforcement learning tasks [54, 33, 32]. They used the robot for learning to move forward through trial and error. The schematic diagram of the robot is shown in figure 3.2.

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 3.2. The angles ranges are chosen so that they are equivalent to the angle ranges chosen by Tsuchiya and Kimura. In the works of Tsuchiya and Kimura the first joint angle is measured from the horizontal

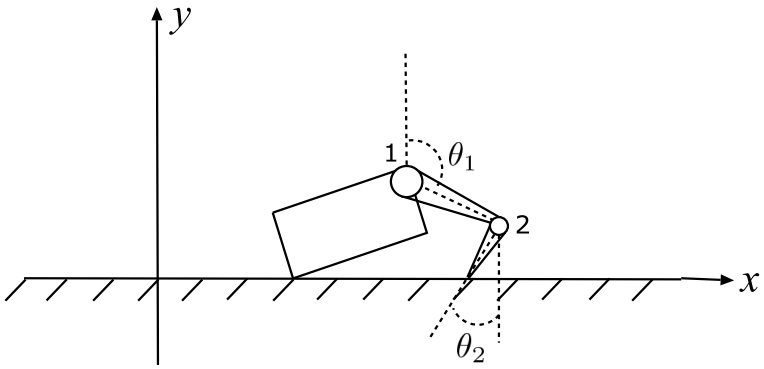


Fig. 3.2: The crawling robotic insect. The robot has one arm with two joints and a touch sensor for detecting whether the tip of the arm is touching the ground or not. The joint angles are measured from the vertical.

while the second joint angle is measured 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 $s_t = (x_0, y_0, x_2, y_2, \theta_1, \theta_2, \phi)$. 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 following equations define the state transition of the system.

$$
\begin{array}{rcll}
\theta_1(t+1) & = & \theta_1(t) + \delta_1 & \\
\theta_2(t+1) & = & \theta_2(t) + \delta_2 & \\
x_0(t+1) & = & x_0(t) & \\
y_0(t+1) & = & y_0(t) & \text{if } \phi(t) = 0 \\
x_2(t+1) & = & x_0(t+1) + l_1 \sin\theta_1(t+1) + l_2 \sin\theta_2(t+1) & \\
y_2(t+1) & = & y_0(t+1) + l_1 \cos\theta_1(t+1) - l_2 \cos\theta_2(t+1) &
\end{array}
$$

,

$$
\begin{array}{rcll}
\theta_1(t+1) & = & \theta_1(t) + \delta_1 & \\
\theta_2(t+1) & = & \theta_2(t) + \delta_2 & \\
x_2(t+1) & = & x_2(t) & \\
y_2(t+1) & = & y_2(t) & \text{if } \phi(t) = 1 \\
x_0(t+1) & = & x_2(t+1) - l_2 \sin\theta_2(t+1) - l_1 \sin\theta_1(t+1) & \\
y_0(t+1) & = & l_2 \cos\theta_2(t+1) - l_1 \cos\theta_1(t+1) &
\end{array}
\tag{3.1}
$$

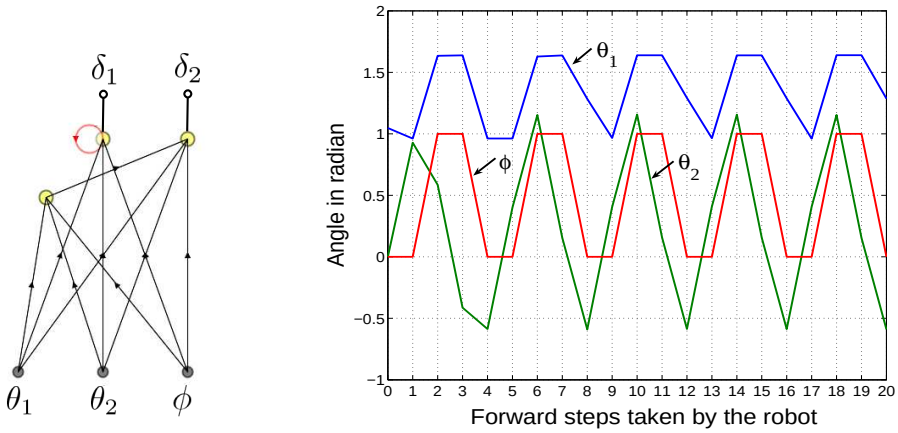
where δ_1 and δ_2 are the outputs of the neural controller, and l_1 and l_2 are the lengths of the first and the second link. The first link is between the first joint and the second joint while the second link is between the second joint and the tip of the arm. Care must be taken in using equation (3.1) especially when the tip touches the ground and loses contact with the ground. 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

$$f = \frac{1}{N}\sum_{t=1}^{N}(x_0(t) - x_0(t-1)), \tag{3.2}$$

where the difference $x_0(t) - x_0(t-1)$ is the velocity of the system at time $t+1$ 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.

Figure 3.3 shows the best controller found and the waveforms of the joint angles and the touch sensor as the robot moves forward.



(a) (b)

Fig. 3.3: Learning to move forward. (a) The best controller found by our algorithm that enables the robot to move forward. (b) The waveforms of the joint angles and the touch sensor as the robot moves forward.

Tsuchiya, Kimura and Kobayashi applied their policy learning by ge-

netic algorithm using importance sampling [54] 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 $\Theta = (\theta_{1,i}, \theta_{2,i}, \theta_{3,i}, \theta_{4,i}, \theta_{5,i}, \theta_{6,i}, \theta_{7,i})$ is used to select the action $a_i(t)$ from normal distribution with mean value $\mu_i = 1/(1 + \exp(-\sum_{k=1}^{6} \theta_{k,i} x_k))$ and standard deviation $\sigma_i = 1/(1 + \exp(-\theta_{7,i})) + 0.1$. 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.

Table 3.1 shows the performance comparison of our method with the method developed by Tsuchiya, Kimura and Kobayashi. The comparison is made with respect to the number of steps taken by the agents in learning to move forward.

Method	Average number of steps
GA-IS [54]	10000
EANT [31]	3520

Tab. 3.1: Performance comparison between genetic algorithm using importance sampling (GA-IS) and EANT.

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. The starting neural network has two output neurons and three input nodes.

3.4 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 [48]. The first reason of 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 [18]. 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 [15, 18]. To make the problem challenging, the basic pole balancing is extended in two ways [58]. 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 this case the controller is provided only with the cart position and the 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 state. Figure 3.4 describes the double pole balancing problem where the poles have 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

$$\begin{aligned}
\ddot{x} &= \frac{F - \mu_c \mathrm{sgn}\,(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{m_c + \sum_{i=1}^{N} \tilde{m}_i} \\
\ddot{\theta}_i &= -\frac{3}{4l_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_i \dot{\theta}_i}{m_i l_i}\right) \\
\tilde{F}_i &= m_i l_i \dot{\theta}_i^{\,2} \sin\theta_i + \frac{3}{4} m_i \cos\theta_i \left(\frac{\mu_i \dot{\theta}_i}{m_i l_i} + g\sin\theta_i\right) \\
\tilde{m}_i &= m_i\left(1 - \frac{3}{4}\cos^2\theta_i\right)
\end{aligned} \tag{3.3}$$

for $i = 1, \ldots, N$ [58]. 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

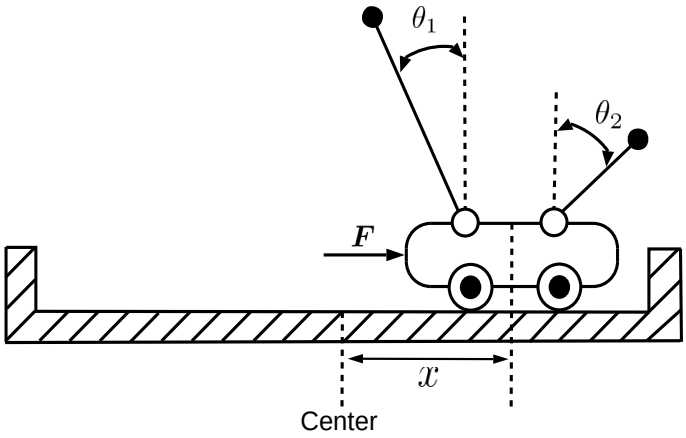


Fig. 3.4: The double pole balancing problem. The poles must be balanced simultaneously by applying a continuous force F to the cart. The parameters x, θ_1 and θ_2 are the offset of the cart from the center of the track and the angles from the vertical of the long and short pole, respectively.

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 the fourth-order Runge-Kutta integration with step size $\tau = 0.01$ s.

3.4.1 Experimental Setup

The experiments are setup in order to be comparable to the results reported in [38, 17, 50, 26]. 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 [17, 50, 26].

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. [21] together with a termination criterion is used in this task. The same fitness function is used by Gomez and Miikkulainen [16], Stanley and Miikkulainen [49], and Igel [26].

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

$$f_1 = t/1000, \tag{3.4}$$

$$f_2 = \begin{cases} 0 & \text{if} \quad t < 100 \\ \frac{0.75}{\sum_{i=t-100}^{t} \left(|x^i| + |\dot{x}^i| + |\theta_1^i| + |\dot{\theta}_i^i| \right)} & \text{otherwise,} \end{cases} \tag{3.5}$$

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. Each start state is chosen by giving each state variable $(x, \dot{x}, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2)$ each of the values 0.05, 0.25, 0.5, 0.75, 0.95, 0, 0, scaled to the range of input variable ($5^4 = 625$). 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 number of successful balances is a measure of the generalization performance of the best solution.

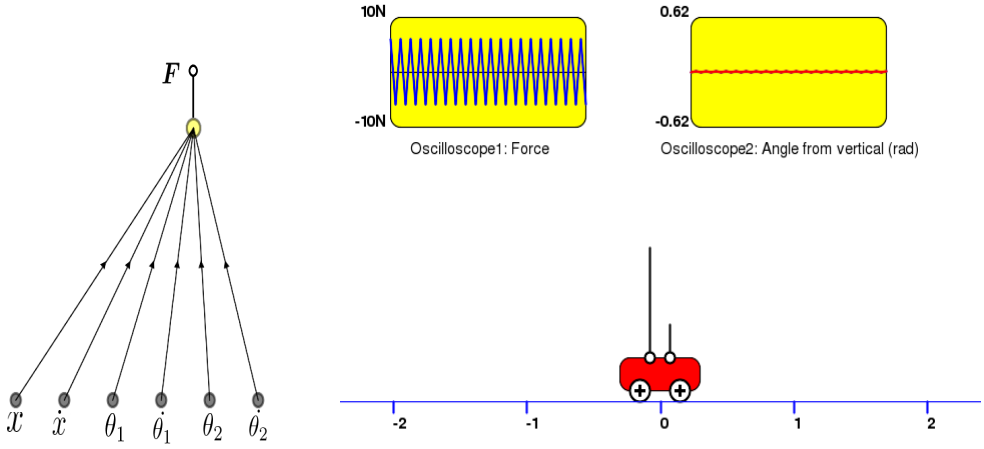
3.4.2 Results

Table 3.2 shows the average value of network evaluations needed by various methods in solving a given task. For our algorithm (EANT), the experiments are done for 120 times for both test scenarios.

Method	Double pole balancing with velocity	Double pole balancing without velocity	
	Evaluations	Evaluations	Generalization
CE [21]	34000	840000	300
ESP [18]	3800	169466	289
NEAT [51]	3600	33184	286
CMA-ES [26]	895	6061	250
EANT [31]	1580	15762	262

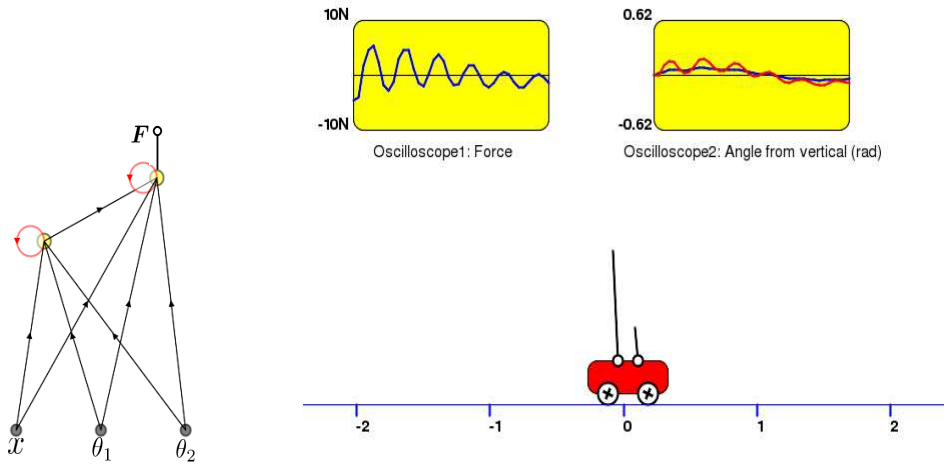
Tab. 3.2: The average network evaluations (trials) needed by various methods in solving the double pole balancing tasks. For CMA-ES, results for a neural network having 3 hidden nodes without a bias are shown.

From table 3.2 one can see that our algorithm is 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 3.5 and 3.6 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. For double pole balancing without velocities, the algorithm found a controller having one output neuron and one hidden neuron. Both neurons have a self-recurrent connection onto themselves. 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. The algorithms found both structures consistently for both test scenarios. The waveforms generated by the controllers depend on the connection 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 3.5 and 3.6 show one possible waveform that can be generated by the structure shown in the respective figures.



(a) (b)

Fig. 3.5: 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) Snapshot of a 2D real-time simulation of the controller in action. The left oscilloscope above the cart-poles system shows the waveform of the force exerted onto the cart and the right oscilloscope shows the waveforms of the angles from the vertical of both poles.



(a) (b)

Fig. 3.6: 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 onto themselves. (b) Snapshot of a 2D real-time simulation of the controller in action. The left oscilloscope above the cart-poles system shows the waveform of the force exerted onto the cart and the right oscilloscope shows the waveforms of the angles from the vertical of both poles.

3.5 Reactive Navigation with Obstacle Avoidance

The aim of this experiment is to demonstrate the automatic design of neural controllers for robots using EANT. We evolved the structure and weights of the neural controller which enables B21 robots [43] to autonomously explore the environment and avoid obstacles. The controller is expected to avoid dead lock situations where Braitenberg-like controllers [7] have difficulties of escaping them. In these situations, they either come to a rest or start to oscillate left to right.

We used the sonar sensors of the B21 robot for detecting the obstacles. 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. 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 (3.6) before feeding them to the neural controller.

$$V_n = \begin{cases} \frac{-V_s+1000}{1000} & \text{if } V_s < 1000 \\ 0 & \text{otherwise} \end{cases} . \tag{3.6}$$

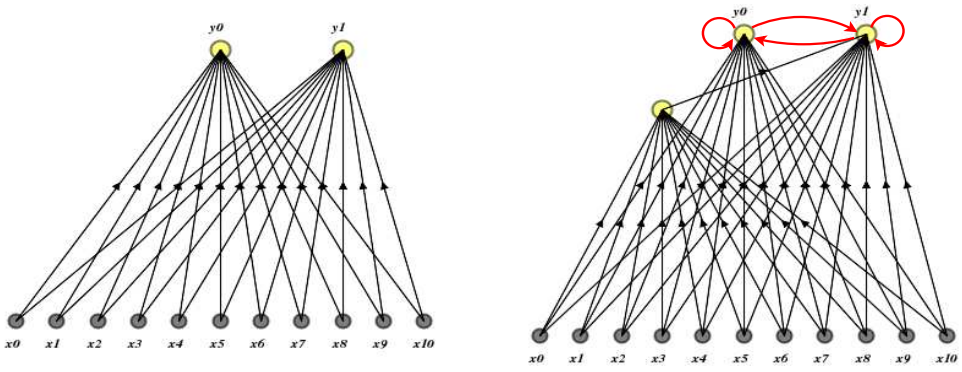
In the equation, V_n is the transformed and normalized sonar reading and V_s is the actual reading returned by a particular sonar sensor. The value V_n lies between 0 and 1 for obstacles which are located at a distance less than 1 m from the center of the robot.

The initial controller has two output neurons and each neuron is connected to all sensors. In addition to the sensor inputs, each neuron has a constant bias input connected to it. 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

$$F = \sum_{t=1}^{T} D(t) e^{-100(H(t)-H(t-1))^2} (1 - S_{max}(t)), \tag{3.7}$$

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 3.7 shows the initial neural controller and the final controller obtained by our algorithm. The ability of

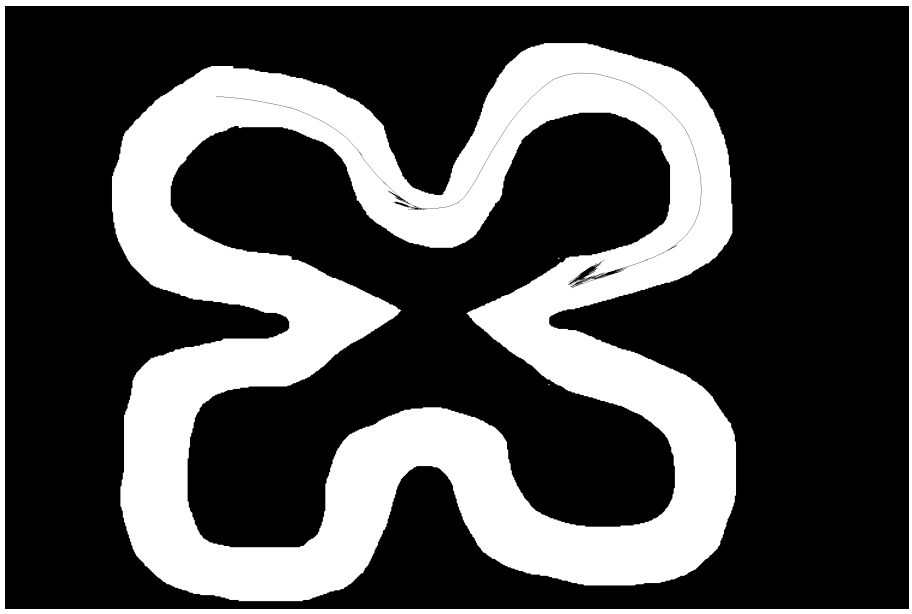
avoiding the deal lock situations comes because of the recurrent connections. The result is similar to that obtained by Nolfi and Floreano [39] and Hülse and Pasemann [25] but in both cases the structure of the neural controller is determined manually beforehand.



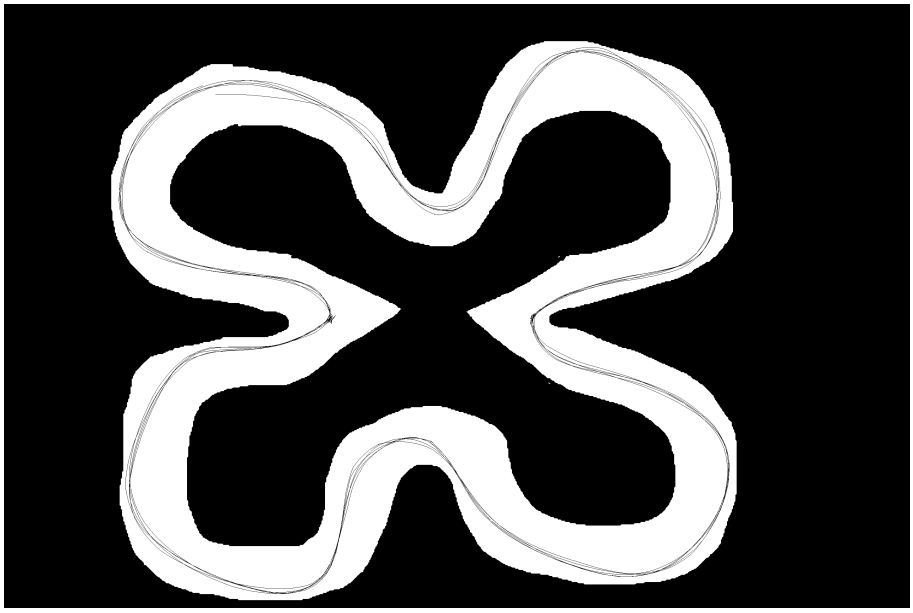
(a) (b)

Fig. 3.7: (a) The initial Braitenberg-like controller (b) The best neural controller found by EANT that is capable of avoiding dead lock situations.

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 3.8 shows an example of the performance of both controllers in a given environment. The best neural controlled found by EANT shows the behavior of avoiding dead lock situations and exploring the environment.



(a)



(b)

Fig. 3.8: (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.

4. CONCLUSION AND OUTLOOK

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 complexifies them along the evolution path. The self-organization process is an emergent property of the system.

The method introduces a compact genetic encoding that enables one to evaluate the neural network encoded by it without some type of ontological process of transforming the genotype into phenotype. In addition to this, a meta-level evolutionary process is introduced that is suitable to explore new structures incrementally and exploit the existing ones.

The system can be extended to handle the evolution of hierarchical structures and modular networks. In addition to this, ways of describing the search space as well as the final resultant networks can be included in order to direct the evolution.

In the future, we are planning to extend the system by using the principles of developmental biology. These principles are useful in evolving and representing very large networks and structures, and in designing a compact and efficient genetic encoding scheme to represent repetitive and recurrent structures.

BIBLIOGRAPHY

[1] 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.

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

[3] 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.

[4] 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, 1998.

[5] 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.

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

[7] V. Braitenberg. *Vehicles. Experiments in Synthetic Psychology.* The MIT Press, 1994.

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

[9] 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.

[10] 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.

[11] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer Verlag, 2003.

[12] 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'99)*, pages 13–17. Springer Verlag, 1999.

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

[14] D. B. Fogel. *Evolving Artificial Intelligence.* PhD thesis, University of California, San Diego, CA, U.S.A., 1992.

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

[16] 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.

[17] 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.

[18] F. J. Gomez and R. Miikkulainen. Robust non-linear control through neuroevolution. Technical report, Department of Computer Sciences, The University of Texas, Austin, TX 78712, U.S.A., 2002.

[19] 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.

[20] 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.

[21] 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.

[22] G. Hailu. *Towards Real Learning Robots*. PhD thesis, Christian Albrechts Universität zu Kiel, Report No. 9906, Institut für Informatik und Praktische Mathematik, Germany, October 1999.

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

[24] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Ann. Arbor, MI: Michigan Press, 1975.

[25] 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.

[26] 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.

[27] 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, Berlin Heidelberg, 2004.

[28] Y. Kassahun and G. Sommer. Learning and adaptation: A comparison of methods in case of navigation in an artificial robot world. Technical Report 0309, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, November 2003.

[29] Y. Kassahun and G. Sommer. Improving learning and adaptation capability of agents, November 2004.

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

[31] 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*, pages 259–266, Bruges, Belgium, April 2005. d-side publications.

[32] 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.

[33] H. Kimura and S. Kobayashi. Reinforcment learning by policy improvement making use of experiences of the other tasks. In *Proceedings of 8th Conference on Intelligent Autonomous Systems*, pages 385–394, Amsterdam, November 2004. IOS Press.

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

[35] J. R. Koza. Genetic programming: A paradigm for genetically breeding computer population of computer programs to solve problems. Technical report, Computer Science Department, Standford University, Stanford, CA, U.S.A., 1990.

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

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

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

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

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

[41] R. Pfeifer and C. Scheier. *Understanding Intelligence.* The MIT Press, 1999.

[42] I. Rechenberg. Evolutionstrategie: Optimierung technischer systeme nach prinzipien des biologischen evolution. *Fromman-Hozlboog Verlag, Stuttgart*, 1973.

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

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

[45] H. P. Schwefel. Numerische optimierung von computer-modellen mittels der evolutionsstrategie. volume Vol.26 of ISR. Birkhaeuser, 1977.

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

[47] 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.

[48] 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.

[49] K. O. Stanley. *Efficient Evolution of Neural Networks through Complexification*. PhD thesis, Artificial Intelligence Laboratory. The University of Texas at Austin., Austin, TX 78712, U.S.A., August 2004.

[50] 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.

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

[52] R. Sutton and A. Barto. *Reinforcement Learning. An Introduction*. The MIT Press, 1998.

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

[54] C. Tsuchiya, H. Kimura, and S. Kobayashi. Policy learning by ga using importance sampling. In *Proceedings of 8th Conference on Intelligent Autonomous Systems*, pages 385–394, Amsterdam, November 2004. IOS Press.

[55] 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.

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

[57] 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.

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

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