ESANN'2005 proceedings - European Symposium on Artificial Neural Networks
Bruges (Belgium), 27-29 April 2005, d-side publi., ISBN 2-930307-05-6.

Efficient Reinforcement Learning Through Evolutionary Acquisition of Neural Topologies

Yohannes Kassahun and Gerald Sommer *

Christian Albrechts University, Department of Cognitive Systems,
Olshausenstr. 40, D-24098 Kiel, Germany

Abstract. In this paper we present a novel method, called Evolutionary Acquisition of Neural Topologies (EANT), of evolving the structure 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 explores new structures whenever it is not possible to further exploit the structures found so far. This enables it to find minimal neural structures for solving a given learning task. We tested the algorithm on a benchmark control task and found it to perform very well.

1 Introduction

A meaningful hybridization of neural networks and evolutionary methods has been proven to be effective in solving reinforcement learning problems [1, 2, 3, 4, 5, 6]. A very good review of hybridizing neural networks with evolutionary methods is given in [7].In evolutionary reinforcement learning, the neural networks are used to represent either a value function or a policy and evolutionary methods are used to search for the optimal value function or optimal policy directly in space of value functions or policies respectively.

The work presented in this paper is related to *NeuroEvolution of Augmenting Topologies* (NEAT) [4] and *NeuroEvolution for Reinforcement Learning using Evolution Strategies* [3]. The presented work starts with networks of minimal structures and complexifies them along the evolution path. It optimizes the weights of the neural networks using an efficient evolution strategy called CMA-ES [8]. But it has the following important features which makes it different from the earlier works:

1. A compact and efficient encoding of a neural network that enables one to evaluate the neural network without decoding it.

2. Exploitation of structures found so far and automatic exploration of new structures whenever it is not possible to further exploit the existing structures.

*This work is sponsored by the German Academic Exchange Service (DAAD) under grant code R-413-A-01-46029 and the COSPAL project under the EC contract no. FP6-2003-IST-2004176, which are duly acknowledged. We would like to thank N. Hansen and A. Ostermeier for making their MATLAB implementation of their CMA-ES algorithm publicly available on the web.

2 Evolutionary Acquisition of Neural Topologies (EANT)

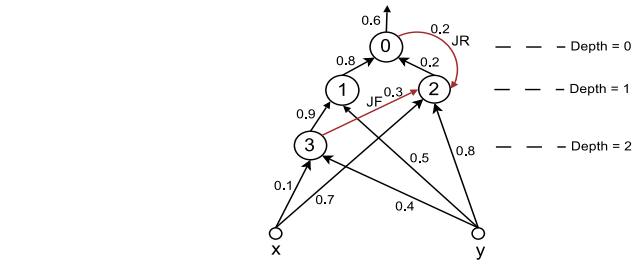
The above two features are considered while designing EANT. We begin by describing the genetic encoding used and continue with the explanation of evaluating a linear genome, structural mutation and exploitation and exploration of structures.

2.1 Genetic Encoding

A flexible encoding enables one to design an efficient evolutionary method that can evolve both the structure and weights of neural networks. The genome in EANT is developed by taking this fact into consideration. A genome in EANT is a linear genome 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 a neural network, or a jumper gene connecting two neurons. The jumper genes are introduced by structural mutation along the evolution path. They encode either a forward or a recurrent jumper connections. 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 of 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 1.

EANT's linear genome can be interpreted as a linear program encoding a tree-based program if one assumes that all the inputs of the neural network and all jumper connections are terminals, and the neurons are functions. A tree-based program (a neural network) can be stored in an array (linear genome) where the tree structure (topology of the network) is implicitly coded in the ordering of the elements of the array. This results in a compact encoding of neural structures using linear genomes.

The linear genome has some interesting properties that makes it useful for evolution of the structure of neural networks. If one assigns integer values to each of the nodes of a linear genome such that an integer value represents the difference between the number of outputs of a node, which is always *one*, and the number of inputs to the node, one can easily see that the sum of the integer values is the same as the number of outputs of the neural network encoded by the linear genome. An integer value of *one* is assigned to all input nodes and forward/recurrent jumper nodes since input nodes and forward/recurrent



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

Fig. 1: An example of encoding a neural network using a linear genome. 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).

jumper nodes are considered as source of signals. An integer value of $1 - n$, where n is the number of inputs to a neuron, is assigned to a neuron node. A sub-network starting from any neuron node on a linear genome can be easily identified using a rule which states that the sum of integer values assigned to the nodes between and including the start neuron node and the end node of a sub-network is *one*.

2.2 Evaluating a Linear Genome

In addition to the compact representation of neural structures, the linear genome offers a possibility of evaluating the neural network encoded by the linear genome without decoding it. The process of evaluating a linear genome without decoding the neural network encoded by it is performed as follows. One starts from the right most node of the linear genome and then moves to the left in computing the output of the nodes. If the current node is an input node, we push its current value and the weight associated with it onto the stack. If the current node is a neuron, we pop n values with their associated weights from the stack and push the result of computation with the weight associated with the neuron node onto the stack. If the current node is a recurrent jumper node, we get the *last value* of the neuron node whose global identification number is the same as that of the jumper node. Then we push the value obtained with the weight associated with the jumper node onto the stack. If the current node is a forward jumper node, we first copy the sub-linear genome (sub-network) starting from a neuron whose global identification number is the same as that of the forward jumper node. We compute the response of the sub-linear genome in the same way as that of the linear genome. Finally, we push the result of computation with the

weight associated with the forward jumper node onto the stack. After traversing the genome from right to left completely, we 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 encoded by the linear genome. An example of evaluating a linear genome is shown in figure 2.

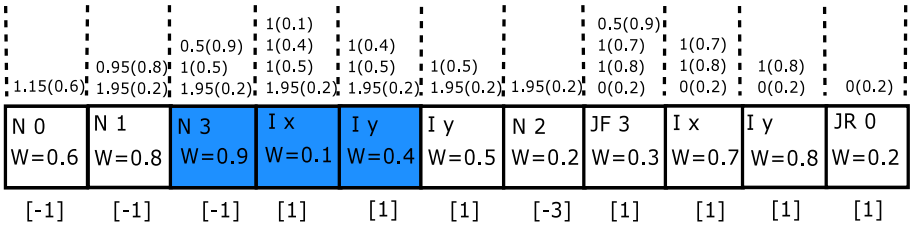


Fig. 2: An example of evaluating a linear genome without decoding the neural network encoded by it. The linear genome encodes the neural network shown in figure 1. For this example, the current values of the inputs to the neural network, x and y, are both set to 1. In the example, all neurons have a 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 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 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

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

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 recurrent connections to the neuron are n_f and $n - n_f$ respectively.

The evaluation of a linear genome is closely related to executing a linear program using a postfix notation. In the genetic encoding the operands (inputs and jumper connections) come before the operator (the neural network).

2.3 Structural Mutation

The structural mutation in EANT adds or removes a forward or recurrent jumper connection between neurons, or adds a new sub-network to the linear genome. The structural mutation does not remove a sub-network because removing a sub-network results in a removal of all jumper connections that are coming or going out of the sub-network. This causes a tremendous loss of the performance of the neural network. The structural mutation operates only on neuron nodes. Assuming that we have currently N neurons in the population, the number of neuron nodes whose structural property will be changed by the structural mutation is given by $n_m = p_m N$ where n_m is the number of neuron nodes whose structural property is changed through structural mutation and p_m is the probability of executing a structural mutation. In applying the structural mutation, each neuron node is tested if it is going to be mutated or not by drawing a random number from a uniform distribution between 0 and 1. If the currently drawn random number is less than 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. Even though it is relatively easy to define a crossover operator for the linear genome, it is not used as a structural search operator since it results in another structure which can be achieved through structural mutation.

2.4 Exploitation and Exploration of Structures

The algorithm starts with networks of minimal structures and complexifies them along the evolution path. The exploration of new structures is initiated whenever it is not possible to further exploit the existing structures. By exploitation we mean optimization of the weights of the neural networks. In order to initiate the structural mutation, one has to detect the condition that it is not possible to further exploit the existing structures. For this purpose, a buffer length of n is used to store the best fitness values of the last n generations. Assuming that we are at the ith generation, and if $F_i - F_{i-n}$, where F_i and F_{i-n} are the best fitness values of the ith and $(i-n)$th generations, is less than some threshold, then it means that the system can not further exploit the existing structures and structural mutation is automatically initiated to search for new structures. In other words, if the rate of increase of the fitness value of the best individual, where the rate is defined over n generations, is less than some threshold, then structural mutation is initiated.

The probability of executing a structural mutation is adjusted so that whenever the structural mutation is initiated, the number of neurons whose structural property will be changed remains constant. The mutation probability is adjusted using

$$p_{current} = p_{start} \left(\frac{N_{start}}{N_{current}} \right), \tag{2}$$

where N_{start} and $N_{current}$ are the starting and the current number of neurons in the population respectively. Likewise, p_{start} and $p_{current}$ are the starting and the current structural mutation probabilities respectively. This kind of adjustment results in a strong structural mutation at the beginning of the evolution. As the evolution goes on, the networks become more complex and the effect of structural mutation decreases. That is, the effect of structural mutation decreases as the performance of the neural networks increases or as the neural networks approach the optimal minimal structure for a given learning task.

Whenever new structures are discovered, they are carried on along the evolution for a certain number of generations 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 exploitation of existing structures begins by clustering structures according to their structural similarities. The members of a cluster are made identical with respect to the weights of the nodes after each generation. Only a representative of a cluster is used in optimizing the weights of a structure. The number of evaluations of a representative of a cluster per generation is determined by

$$n_e = (F_k/F_t)PS, \tag{3}$$

where n_e is the number of evaluations used in optimizing the weights of a given structure, F_k is the average fitness value of the individuals in the kth cluster and F_t is the sum of the average fitness values of all clusters in the population and PS is the population size.

CMA-ES [8] is used in optimizing the weights of the neural networks. Every cluster has its own covariance matrix C and a global step-size σ. As long as the cluster exists in the population, the covariance matrix and the global step-size continue to develop using the covariance matrix adaptation (CMA). The number of evaluations per generation used in optimizing the weights of a representative of a cluster is given by equation (3). A newly formed cluster first initializes its covariance matrix to identity matrix and then copies the entries of the covariance matrix of the parent cluster so that the already developed correlations between the old nodes is maintained. The global step-size is however initialized to some initial value σ_o.

3 Experiments

We have run two experiments to measure the performance of our algorithm. The first experiment tries to answer if EANT is able to evolve the minimum necessary neural structure for a given learning task and the second experiment

compares the performance of our algorithm on double pole balancing tasks with other algorithms tested on the same tasks [1, 3, 4].

Evolving the Necessary Minimal Neural Structure

In this experiment, EANT is made to evolve an XOR network. A neural network must have at least one hidden node in order to solve the XOR problem. The first generation contains networks with no hidden nodes. On 100 runs, EANT is able to find networks having on the average 1.52 hidden nodes and it takes EANT on the average 1234 network evaluations to get a solution. EANT is consistent in finding the minimal neural structure to the XOR problem.

Double Pole Balancing

Both double pole balancing tasks with and without velocity information are considered in our experiments. All the specifications of the experiments including the fitness functions are made to be the same as that described in [1, 3, 4]. Here we give only results of our experiments with the results of other algorithms on the double pole balancing tasks. The experiments in EANT are run for 120 times and the values for EANT in table 1 show the average value of network evaluations needed to solve a given task.

Method	Double pole balancing with velocity	Double pole balancing without velocity	
	Evaluations	Evaluations	Generalization
CE [9]	34000	840000	300
ESP [1]	3800	169466	289
NEAT [4]	3600	33184	286
CMA-ES [3]	895	6061	250
EANT	1580	15762	262

Table 1: The average network evaluations (trials) needed by various methods in solving the double pole balancing tasks. For CMA-ES, results for a neural network having 3 hidden nodes without a bias is shown. Generalization measures the ability of an evolved neural network to balance the poles from different starting states. The numbers under the column "Generalization" show the number of successful balances starting from 625 different starting states.

From table 1 one can see that the EANT algorithm is better than other algorithms that evolve both the structure and weights of a neural network (CE, ESP, NEAT). The CMA-ES has outperformed EANT on both double pole balancing tasks for an already manually chosen network topology.

4 Conclusion and Outlook

An efficient and compact genetic encoding of neural networks onto a linear genome, which enables one to evaluate the network without decoding it, is presented. The topology of the network is implicitly encoded in the ordering of the elements of the linear genome. The concept of exploiting the existing structures and exploring for new ones whenever it is not possible to further exploit the existing ones is also presented.

In the future, we are planning to extend the system to handle the evolution of hierarchical structures and modular networks. We are also planning to develop ways of describing the search space as well as the final resultant networks.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.