The Automated Design of Artificial Neural Networks Using Evolutionary Computation

Jae-Yoon Jung and James A. Reggia

Department of Computer Science and UMIACS, University of Maryland, College Park, MD 20742, USA, jung@cs.umd.edu, reggia@cs.umd.edu

Summary. Neuroevolution refers to the design of artificial neural networks using evolutionary algorithms. It has been one of the promising application areas for evolutionary computation, as neural network design is still being done by human experts and automating the design process by evolutionary approaches will benefit developing intelligent systems and understanding "real" neural networks. The core issue in neuroevolution is to build an efficient, problem-independent encoding scheme to represent repetitive and recurrent modules in networks. In this chapter, we have presented our descriptive encoding language based on genetic programming and showed experimental results supporting our argument that high-level descriptive languages are a viable and efficient method for development of effective neural network applications.

1 Introduction

Most development of neural networks today is based upon manual design. A person knowledgeable about the specific application area specifies a network architecture and activation dynamics, and then selects a learning method to train the network via connection weight changes. While there do exist non-evolutionary methods for automatic incremental network construction [32], these generally presume a specific architecture, do not automatically discover network modules appropriate for specific training data, and have not enjoyed widespread use. This state of affairs is perhaps not surprising, given that the general space of possible neural networks is so large and complex that automatically searching it for an optimal network architecture may in general be computationally intractable or at least impractical for complex applications [6, 33].

Neuroevolution refers to the design of artificial neural networks using evolutionary algorithms, and it has attracted much research because (1) successful approaches will facilitate wide-spread use of intelligent systems based on artificial neural networks; and (2) it will shed lights on our understanding of how "real" neural networks have been evolved [21, 45]. Recent successes using

J.-Y. Jung and J.A. Reggia: *The Automated Design of Artificial Neural Networks Using Evolutionary Computation*, Studies in Computational Intelligence (SCI) **92**, 19–41 (2008)
www.springerlink.com

evolutionary computation methods as design/creativity tools in electronics, architecture, music, robotics, and other fields [3, 5, 30] suggest that creative evolutionary systems could have a major impact on the effectiveness and efficiency of designing neural networks. This hypothesis is supported by an increasing number of neuroevolutionary methods that search through a space of weights and/or architectures without substantial human intervention, trying to obtain an optimal network for a given task (reviewed in [2, 42, 43, 51]). Specifically, evolutionary algorithms have been successfully applied to many aspects of neural network design, including connection weights, learning rules, activation dynamics, and network architectures.

Evolution of connection weights in a fixed network architecture has been done successfully since the earliest stages of neuroevolution (e.g., [13, 14, 35, 46]). With either binary or real representations, all weights are encoded into a chromosome, then appropriate weights are searched for using genetic operations and selection. A problem in encoding all weights is that it may not be efficient in a large scaled network, with dense connectivity. A hybrid training approach that combines an evolutionary algorithm for global search with other local learning algorithms (e.g., backpropagation) for the fine tuning of weights has also been successful in many cases (e.g., [1,4,51]).

In many experiments, learning rules and activation dynamics have been predefined by the neural network designer and remain fixed throughout the evolution process. But if there is little knowledge about the problem domain, it is reasonable to include them as a part of evolution, since a small change in parameters or learning/activation rules can lead to quite different performance by the network. Encoding learning parameters into chromosomes can be considered as a first step towards the evolution of learning rules. In [22], the learning rate as well as the network architecture were encoded and evolved at the same time. Recently, Abraham encoded learning parameters and activation functions (e.g., sigmoid, tanh, logistic, and so on) and tried to evolve an optimal network for three time series [1]. Another interesting approach related to the learning rule is to use evolutionary algorithms to discover new learning rules. Chalmers assumed that a learning rule must be some linear combination of local information, and tried to find the needed coefficients using a genetic algorithm [10]. In [38], genetic programming was used instead of a standard genetic algorithm, since symbolic learning rules as well as their coefficients can be evolved within genetic programming.

Finally, there has been substantial interest in evolving neural network architectures (structures). There are roughly two different schemes that have been used to encode an architecture: direct and indirect encoding schemes. With the direct encoding scheme, connection information is explicitly specified in a connectivity matrix [33,34]. It is simple and all possible network architectures within a fixed matrix size can be represented. However, for a large neural network, searching for the optimal architecture with a direct encoding scheme can be impractical since the size of the space increases exponentially with the network size. On the other hand, only some essential features of the network

architecture are specified with an indirect encoding scheme [9, 20, 29, 47]. For example, a network is represented by the size of each neuron block and their connections to the other blocks [22]. But this approach assumes that other features are predefined (e.g., all linked blocks are assumed to be fully connected in this example), so it searches a limited space.

Many previous studies involving evolution of neural networks start with randomly generated network architectures, undertake evolution without any specific bounds on the range of structures permitted, and/or are limited to an evolutionary process that occurs on a single level (e.g., only at the level of numbers of individual nodes and their connections, versus only at the higher level of whole network layers and their interconnecting pathways). In this chapter we describe our recent work developing a non-procedural language to specify the initial networks and the search space, and to permute virtually any aspects of neural networks to be evolved [25, 26]. This is done by using a high-level descriptive language that represents modules/layers and inter-layer connections (rather than individual neurons and their connections) in a hierarchical fashion that can be processed by genetic programming methods [3, 30], and in this way our approach permits the designer to incorporate domain knowledge efficiently. This domain knowledge can greatly restrict the size of the space that the evolutionary process must search, thereby making the evolutionary process much more computationally tractable in specific applications without significantly restricting its applicability in general. Our system does not a priori restrict whether architecture, layer sizes, learning method, etc. form the focus of evolution as many previous approaches have done, but instead allows the user to specify explicitly which aspects of networks are to evolve, and permits the simultaneous evolution of both intra-modular and inter-modular connections.

2 Descriptive Encoding Methodology

We refer to our encoding scheme as a *descriptive encoding* since it enables users to describe the target space of neural networks that are to be considered in a natural, non-procedural and human-readable format. A user writes a text file like the ones shown later in this paper to specify sets of modules (layers) with appropriate properties, their range of legal evolvable property values, and allowable inter-module connectivity ("pathways"). This input description does *not* specify individual neurons, connections, nor their weights.[1] The specification of legal property values affects the range of valid genetic operations. In other words, a description file specifies the initial population and environment variables, and restricts the space to be searched by genetic operators during evolution.

[1] Individual neurons, connections and weights *can* be specified by creating layers/modules containing single neurons, but this does not take advantage of the language's ability to compactly describe large scale networks.

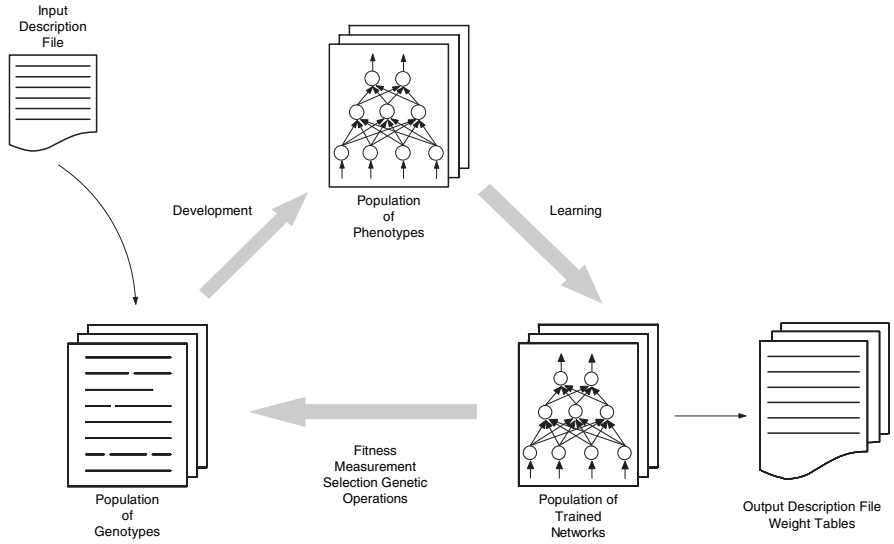


Fig. 1. The iterative three-step development, learning, and evolution procedure used in our system. The input description file (*upper left*) is a human-written specification of the class of neural networks to be evolved (the space to be searched) by the evolutionary process; the output description file (*lower right*) is a human-readable specification of the best specific networks obtained

The evolutionary process in our system involves an initialization step plus a repeated cycle of three stages, as shown in Fig. 1. First, the text description file prepared by the user is parsed and an initial random population of chromosomes (genotypes) is created within the search space represented by the description (leftmost part of Fig. 1). During the development stage, a new population of realized networks (phenotypes) is created or "grown" from the genotype population. Each phenotype network has actual and specific individual nodes, connection weights, and biases. The learning stage involves training each phenotype network, assuming that the user specifies one or more learning rules in the description file, making use of an input/output pattern file that contains training data. As evolutionary computation is often considered to be less effective for local, fine tuning tasks [17, 51], we adopt neural network training methods to adjust connection weights. After the training stage, each individual network is evaluated according to user-defined fitness criteria and genetic operators are applied to the genotypes. Fitness criteria may reflect both network performance (e.g., mean squared error) and a penalty for a large network (e.g., total number of nodes), or other measures. The end result of an evolutionary run consists of two things. First, an output description is produced (see Fig. 1, bottom right). This file uses the same syntax as the input description file to specify the most fit specific network architectures discovered by the evolutionary process. Second, another file gives a table of

all the weights found by the final learning process using these specific network architectures.

2.1 Language Description

The basic unit of our encoding is a module that we call a *layer*, which is defined as an array of network nodes that share common properties. In other words, individual neurons are not the atomic unit of evolution, but sets of neurons are. The description of a layer/module starts with an identifier LAYER, which is followed by an optional layer name and a list of properties. Properties of a layer can be categorized into three groups: structure (e.g., BIAS, SIZE, NUM_LAYER), dynamics (e.g., ACT_RULE, ACT_INIT, ACT_MIN, ACT_MAX), and connectivity (e.g., CONNECT, CONNECT_RADIUS, CONNECT_INIT, LEARN_RULE). The order of declared properties in a layer description does not matter in general. Individual properties can be designated to be evolvable within some range, or to be fixed. Each property has its own default value for simplicity: if some properties are missing in the description file, they will be replaced with the default values during the initialization stage and considered as being constant throughout the evolutionary process (i.e., the chromosome is in fact more strict than the description; it requires all default, fixed, and evolvable properties to be present in some form). Layer properties used in this article are illustrated in Table 1. The meaning of each property is fairly straightforward, but the [CONNECT_RADIUS r] property with $0.0 \leq r \leq 1.0$ needs more explanation. It defines the range of the connectivity from each node in a source layer to the nodes in a target layer. For example, if $r = 0.0$, each node in the source layer is connected to just

Table 1. Example module/layer properties

Property	What it specifies about a module/layer
BIAS	Whether to use bias units and their initial value ranges if so
SIZE	Number of nodes in the current layer
NUM_LAYER	Number of layers of this type
ACT_RULE	Activation rule for nodes in the layer
ACT_INIT	Initial activation value for nodes
ACT_MIN	Minimum activation value
ACT_MAX	Maximum activation value
CONNECT	Direction of connections starting from this layer
CONNECT_RADIUS	Range of connectivity from 0.0 to 1.0 (see text)
CONNECT_INIT	Initial (range of) weights in the current connections
LEARN_RULE	Learning rule for the current connections

a single node in the matching position of the target layer. If r is a positive fraction less than one, each source node connects to the matching destination layer node and its neighbor nodes out to a fraction r of the radius of the target layer; thus, if $r = 1.0$, the source and target layers are fully connected. While these connectivity properties are basically intended to specify connections between two layers (i.e., an inter-modular connection), intra-modular connections such as self-connectivity can also be designated using the same properties. For example, one can specify that each node in a layer named *layer*1 connects to itself by using a combination of [CONNECT *layer*1] and [CONNECT_RADIUS 0.0]. The user can also change the default value of each property for a specific problem domain by declaring and modifying property values in the evolutionary part of a description file, which will be explained later.

Figure 2a illustrates part of an input description file written in our language for evolving a recurrent network. Each semantic block, enclosed in brackets [ ], starts with a type identifier followed by an optional name and a list of properties about which the user is concerned in the given problem. A network may contain other (sub)networks and/or layers recursively, and a network type identifier (SEQUENCE, PARALLEL, or COLLECTION) indicates the conceptual arrangement of the sub-networks contained in this network. If a network module starts with the SEQUENCE identifier, the sub-networks contained in this module are considered to be arranged in a sequential manner (e.g., like a typical feed-forward neural network). Using the PARALLEL identifier declares that the sub-networks are to be arranged in parallel, and the COLLECTION identifier indicates that an arbitrary mixture of sequential and parallel layers may be used and evolved. The COLLECTION identifier is especially useful when there is little knowledge about the appropriate relationships

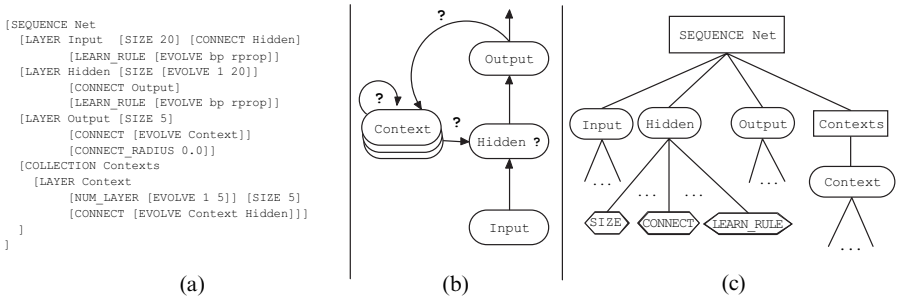


Fig. 2. (**a**) The first part of a description file specifies the network structure in a top-down hierarchical fashion (other information in the description file, such as details of the evolutionary process, is not shown). (**b**) A schematic illustration of the corresponding class of recurrent networks that are described in (**a**). *Question marks* indicate architecture aspects that are considered evolvable. (**c**) Part of the tree-like structure corresponding to the genotype depicted in (**a**). Each *rectangle*, *oval*, and *hexagon* designates a network, layer, and property, respectively

between the layers being evolved. As described earlier, we define a layer as a set (sometimes one or two dimensional, depending on the problem) of nodes that share similar properties, and it is the basic module of our network representation scheme. For example, the description in Fig. 2a indicates that a sequence of four types of layers are to be used: input, hidden, output, and context layers, as pictured in Fig. 2b. Properties fill in the details of the network architecture (e.g., layer size and connectivity) and in general specify other network features including learning rules and activation dynamics.

Most previous neuroevolution research has focused a priori on some limited number of network features (e.g., network weights, number of nodes in the hidden layer) assuming that the other features are fixed, and this situation has impeded past neuroevolutionary models from being used more widely in different environments. To overcome this limitation, we let users decide which fixed properties are necessary to solve their problems, and which other factors should be evolved, from a set of supported properties that span many aspects of neural networks. Unspecified properties are replaced with default values and are treated as being fixed after initialization. So, for example, in the description of Fig. 2a, the input layer has a fixed, user-assigned number of 20 nodes and is connected to the hidden layer, while the single hidden layer has an evolvable SIZE within the range 1–20 nodes. The EVOLVE attribute indicates that the hidden layer's size will be randomly selected initially and is to be modified within the specified range during the evolution process. Note that the learning rules to be used for connections originating from both input and hidden layers are also declared as an evolvable property (in this case, a choice between two variants of backpropagation). The description in Fig. 2a also indicates that one to five context layers are to be included in the network; this is the main architectural aspect that is to be evolved in this example. These context layers are to be ordered arbitrarily, all contain five neurons, and they evolve to have zero or more inter-layer output connections to either other context layers or to the hidden layer (Fig. 2b). Finally, the output layer evolves to propagate its output to one or more of the context layers, where the CONNECT_RADIUS property defines one-to-one connectivity in this case. Since the number of layers in the context network may vary from 1 to 5 (i.e., LAYER context has an evolvable NUM_LAYER property), this output connectivity can be linked to any of these layers that were selected in a random manner during the evolution process. Figure 2b depicts the corresponding search space schematically for the description file of Fig. 2a, and the details of each individual genotype (shown as question marks in the picture) will be assigned within this space at the initialization step and forced to remain within this space during the evolution process. Note that since the genotype structure is a tree, as shown in Fig. 2c, fairly standard tree-manipulation genetic operators as used in GP [3, 30] can be easily applied to them with our approach.

The remainder of the description file consists of information about training and evolution processes (not shown in Fig. 2a). A *training block* specifies

the file name where training data is located and the maximum number of training epochs. Default property values may also be designated here, like the LEARN_RULE to be used. In other words, when a property value is specified in this block, the default value of that property is changed accordingly and affects all layers which have that property. An *evolution block* can also be present and specifies parameters affecting fitness criteria, selection method, type and rate of genetic operations, and other population information (illustrated later).

3 Evaluation Results

While the approach to evolutionary design of neural networks described above may seem plausible, it remains to actually establish that the combination of search space restriction via a high-level language, developmental encoding, modular network organization, and integrated use of evolution and network learning can be used effectively in practice. Although our system is intended ultimately to evolve any aspect of neural networks, here we focus on evolving network architectures, evaluating our system on two different problems in which the task is to discover an architecture that both solves the problem and also sheds light on the modular nature of the solution, as a first step in establishing such effectiveness. Our goal is simply to show that when one uses a high-level descriptive language (with all of its benefits as outlined in the Introduction), it is still possible to discover interesting neural network solutions to problems using evolutionary computation methods. While we do not claim that these results are specifically due to our language, we believe our language facilitates the specification of search spaces, supports analysis of initial conditions and final results, makes the tasks much more efficient for the human investigator, and encourages the emergence of modularity in network solutions (via the use of built-in cost functions).

3.1 Module Formation in a Feed-Forward Network

Many animal nervous systems have parallel, almost structurally independent sensory, reflex, and even cognitive systems, a fact that is sometimes cited as contributing to their information processing abilities. For example, biological auditory and visual systems run in parallel and are largely independent during their early stages [27], the same is true for segmental monosynaptic reflexes in different regions of the spinal cord [27], and the cerebral cortex is composed of regional modules with interconnecting pathways [36]. Presumably evolution has discovered that such *partitioning* of neural networks into parallel multi-modular pathways is both an effective and an efficient way to support parallel processing when interactions between modules are not necessary. However, the factors driving the evolution of modular brain architectures

Table 2. Training data for a 2-partition problem

Input	Output	Input	Output	Input	Output	Input	Output
0 0 0 0	0 0	0 1 0 0	1 0	1 0 0 0	1 0	1 1 0 0	0 0
0 0 0 1	0 1	0 1 0 1	1 1	1 0 0 1	1 1	1 1 0 1	0 1
0 0 1 0	0 1	0 1 1 0	1 1	1 0 1 0	1 1	1 1 1 0	0 1
0 0 1 1	0 0	0 1 1 1	1 0	1 0 1 1	1 0	1 1 1 1	0 0

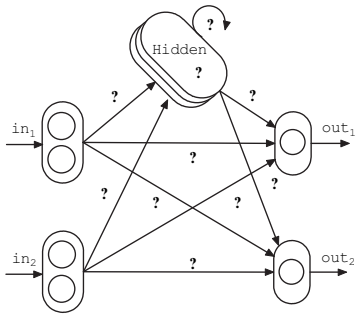
having components interconnected by distinct pathways have long been uncertain and currently remain a very active area of discussion and investigation in the neurosciences [8, 15, 28, 48].

Inspired by such modular neurobiological organization, and as a first test problem for our descriptive encoding system, we examined whether an evolutionary process could discover the existence and details of n independent neural pathways between subsets of input and output units that are implied by training data. We call such problems *n-partition problems*. Table 2 gives an example when $n = 2$ of a 2-partition problem. The goal is to evolve a minimal neural network architecture that can learn the given mapping from four input units to two output units, all of which are assumed to be standard logistic neurons in this case. The data in Table 2 implicitly represent a 2-partition problem in that the correct value of the first output depends only on the values of the first two input units (leftmost columns in the table), while the correct value of the second output depends only on the values of the remaining two input units. Thus, two parallel independent pathways from inputs to outputs are implied, and in this specific example each output is arranged to be the exclusive-or function of its corresponding inputs. A human designer would recognize from this information both that hidden units are necessary to solve the problem (since exclusive-or is not linearly separable), and that two separate parallel hidden layers are a natural minimal architecture. Of course, given just the training data in Table 2 and not knowing a priori the input/output relationships, such a design would not be evident in advance to an evolutionary algorithm, nor most likely to a person, and the question being asked here is whether an evolutionary process would discover it when given a suitable descriptive encoding.

The description file that we used to evolve an architecture to solve the 2-partition problem of Table 2 is shown in Fig. 3a. It specifies the initial architecture for the 2-partition problem, in which only the number of input and output nodes are fixed while other aspects, such as inter-layer connectivity and hidden layers' structure as shown in Fig. 3b, are randomly created during initialization and evolve. In this example we separate the input neurons into groups that form the basis for the distinct pathways, but note that the learning algorithm makes no use of this fact. The descriptive encoding also specifies the boundaries of the search space, indicating that there can be 0–10 hidden layers organized in any possible interconnecting architecture and the number

```
[SEQUENCE Dual_xor
  [PARALLEL Input
    [LAYER in  [NUM_LAYER 2][SIZE 2]
               [CONNECT     [EVOLVE Hidden Output]]]]
  [COLLECTION Hidden
    [LAYER hid [NUM_LAYER   [EVOLVE 0 10]]
               [SIZE        [EVOLVE 1  5]]
               [CONNECT     [EVOLVE Hidden Output]]]]
  [PARALLEL Output
    [LAYER out [NUM_LAYER 2][SIZE 1]]]]
[TRAINING
  [TRAIN_DATA "./inout_pattern.txt"] [MAX_TRAIN 100]
  [LEARN_RULE rprop]]
[EVOLUTION
  [FITNESS WEIGHTED_SUM
    [MSE:0.5 TOT_NODE:0.2 TOT_CONN:0.2]]
  [SELECTION TOURNAMENT] [TOURNAMENT_POOL 3] [ELITISM 0]
  [MUTATION_PROB 0.7]    [CROSSOVER_PROB 0.0]
  [MAX_GENERATION 50]    [MAX_POPULATION 50]]
```

(a)



(b)

Fig. 3. (**a**) The initial network description and (**b**) a sketch of the space of networks to be searched for the 2-partition problem of Table 2

of nodes (SIZE property) in each hidden layer may vary from 1 to 5. The NUM_LAYER and SIZE properties in the hidden structure and all CONNECT properties are declared as evolvable (indicated by question marks in Fig. 3b), so these properties and only these properties can be modified by genetic operators during evolution.

As explained earlier, a descriptive encoding generally provides additional information about the training and evolutionary processes to be used, and we illustrate that here. This user-defined information follows the network part of the description (Fig. 3a, top), setting various parameter values to control the training and evolutionary procedure. In this case, each phenotype network is to be trained for 100 epochs with the designated input/output pattern file that encodes the information from Table 2. The default learning rule is defined as a variant of backpropagation (RPROP [41]). Note that there is no issue of generalization in learning the boolean function here since all inputs and the correct outputs for them are given a priori. The EVOLUTION part of the description (Fig. 3a, bottom) indicates that a weighted sum method of three criteria to be used: mean squared error (MSE, e), total number of network nodes (n), and total number of layer-to-layer connections (c). MSE reflects the output performance of the network, and the other two criteria are adopted as penalties for larger networks. These three criteria are reciprocally normalized and then weighted with coefficients assigned in the description file. More specifically, the fitness value of the ith network, $Fitness_i$ that is described here is

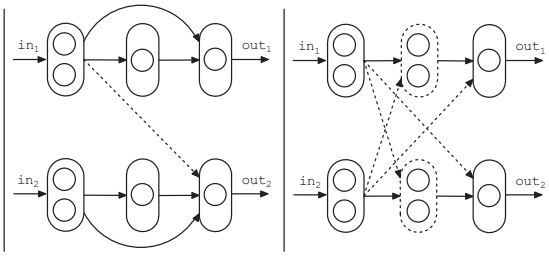
$$Fitness_i = w_1 \left(\frac{e_{max} - e_i}{e_{max} - e_{min}} \right) + w_2 \left(\frac{n_{max} - n_i}{n_{max} - n_{min}} \right) + w_3 \left(\frac{c_{max} - c_i}{c_{max} - c_{min}} \right), \tag{1}$$

where $x_{min}(x_{max})$ denotes the minimum (maximum) value of criterion x among the population, and the coefficients w_1, w_2, and w_3 are empirically defined as 0.5, 0.2, and 0.2, respectively. In words, the fitness of an individual neural network is increased by lower error (a behavioral criterion), or by fewer nodes and/or connections (structural criteria). An implicit hypothesis represented in the fitness function is that minimizing the latter two structural costs may lead to fewer modules and independent pathways between them in evolved networks. Note that the EVOLUTION part of the description (Fig. 3a) specifies the coefficients in the fitness function above, and it also specifies tournament selection with a pool size of 3 as the selection method, a mutation rate of 0.7, and that no crossover and no elitism are to be used. Operators in this case can mutate layer size and direction of an existing inter-layer connection, and can add or delete a new layer or connection.

We ran a total of 50 simulations with the fixed population size (50) and the fixed number of generations of 50. Between simulations, the only changes are the initial architectures plus the initial value of connection weights that are assigned randomly in the range from -1.0 to 1.0, as used in [40]. For all runs, each final generation contained near-optimal networks that both solved the 2-partition problem (i.e., MSE $\sim$ 0.0) and had a small number of nodes and connections. Converged networks can be categorized into two groups identified by their connectivity pattern as depicted in Fig. 4. The first group of networks, found during 44% of the runs, showed a dual independent pathway where each input pair has their own hidden layer and a direct connection to the corresponding output node (Fig. 4a and 4b). Ignoring the dotted line connections which have near zero weight values shown in Fig. 4a, this is an optimal network for the 2-partition problem in terms of the total number of network nodes and connections. In the second group of networks, found during 52% of the runs, input layers share a single hidden layer, without having direct connections to the corresponding output nodes. Such solutions require

```
[SEQUENCE Dual_xor
   [PARALLEL Input
      [LAYER in1 [SIZE 2]
                 [CONNECT h1 out1 out2]]
      [LAYER in2 [SIZE 2][CONNECT h2 out2]
   ]
   [PARALLEL Hidden
      [LAYER h1 [SIZE 1][CONNECT out1]]
      [LAYER h2 [SIZE 1][CONNECT out2]]
   ]
   [PARALLEL Output
      [LAYER out1 [SIZE 1]]
      [LAYER out2 [SIZE 2]]
   ]
]
```

(a)



(b) (c)

Fig. 4. Typical network architectures found during evolution for the 2-partition problem are depicted. *Dotted lines* show connectivity with near-zero weights. (**a**) Final output description file having two independent pathways. (**b**) Conceptual network architecture described by (**a**). (**c**) Dual pathway network without direct input-to-output connections. Implicit hidden sub-layers are indicated by *dotted ovals*

four hidden nodes, rather than two, to be an optimal network. Inspection of the connection weights shows that this model implicitly captures/discovers two distinct pathways embedded in the explicit hidden layer as illustrated in Fig. 4c (near zero weight connections that do not affect the performance are pruned). This type of network is also an acceptable near-optimal solution in terms of the number of connections needed for XOR problems and is sometimes used to illustrate layered solutions to single XOR problems in textbooks (e.g., [23]). The remaining 4% of the runs did not converge on just one type of network as described above, but both types are found in the final population. Thus, the evolutionary process generally discovered that "minimal cost" solutions to this problem involve independent pathways. While the networks considered here are very simple relative to real neurobiological systems, the frequent emergence of distinct and largely independent pathways rather than more amorphous connectivity during simulated evolution raises the issue of whether parsimony pressures may be an underrecognized factor in evolutionary morphogenesis, as outlined at the beginning of this section (see [45] for further discussion).

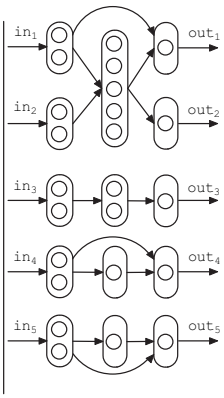
Without changing the evolutionary part of the description file, we tested n-partition problems for $n = 2, 3, 4$, or 5 (the latter requires $2^{2n} = 1{,}024$ patterns for training). A typical input/output description file and network structure for $n = 5$ are illustrated in Fig. 5. Table 3 summarizes the experimental results. Minimum hidden nodes is the smallest number of hidden nodes found during an experiment, and numbers in the parentheses show the theoretically minimum number of nodes possible In an n-partition problem

```
[SEQUENCE 5_xor
  [PARALLEL Input
    [LAYER in
      [NUM_LAYER 5]
      [SIZE      2]
      [CONNECT
        [EVOLVE Hidden Output]]]
  [COLLECTION Hidden
    [LAYER h
      [NUM_LAYER [EVOLVE 0 10]]
      [SIZE      [EVOLVE 1  5]]
      [CONNECT
        [EVOLVE Hidden Output]]]
  [PARALLEL Output
    [LAYER out
      [NUM_LAYER 5]
      [SIZE      1]]]
]
```

(a)

```
[SEQUENCE 5_xor
  [PARALLEL Input
    [LAYER in1 [SIZE 2][CONNECT h1 out1]]
    [LAYER in2 [SIZE 2][CONNECT h1]]
    [LAYER in3 [SIZE 2][CONNECT h2]]
    [LAYER in4 [SIZE 2][CONNECT h3 out4]]
    [LAYER in5 [SIZE 2][CONNECT h4 out5]]]
  [PARALLEL Hidden1
    [LAYER h1  [SIZE 5][CONNECT out1 out2]]
    [PARALLEL Hidden2
      [LAYER h2  [SIZE 2][CONNECT out3]]
      [LAYER h3  [SIZE 1][CONNECT out4]]
      [LAYER h4  [SIZE 1][CONNECT out5]]]]
  [PARALLEL Output
    [LAYER out1 [SIZE 1]]
    [LAYER out2 [SIZE 1]]
    [LAYER out3 [SIZE 1]]
    [LAYER out4 [SIZE 1]]
    [LAYER out5 [SIZE 1]]]]
```

(b)



(c)

Fig. 5. A typical example of a final evolved network for the 5 XOR partition problem. (**a**) Initial network description. Properties to be evolved are in *bold font*. (**b**) Final description file produced as output by the system. All EVOLVE properties have been replaced by the specific choices in *bold font*. Only SIZE and CONNECT properties are shown. (**c**) Depicted network architecture. Connections that have near-zero weights are pruned

Table 3. Parallel n-partition problem results

N	# of patterns	Minimum hidden nodes	Minimum connections	Minimum MSE	Fully connected MSE	Average time
2	16	2 (2)	7 (6)	0.00000	0.20823	2,250
3	64	4 (3)	9 (9)	0.00021	0.20710	2,848
4	256	5 (4)	14 (12)	0.00073	0.22224	3,471
5	1,024	9 (5)	19 (15)	0.00236	0.20821	6,273

involving exclusive-OR relations, at least n hidden nodes are necessary even if direct connections from input to output are allowed. The Minimum connections column shows the minimum number of layer-to-layer connections found in the best individual. Again assuming direct connectivity from input to output and without increasing the number of hidden nodes, the best possible number of connections in an n-partition problem is $3n$ (e.g., input to output, input to the corresponding hidden layer, hidden to output). For each n partition problem, we also compared the best MSE results gathered from each evolutionary simulation with that of standard fully connected, single hidden layer backpropagation networks.

These latter networks have a single fixed hidden layer size of n, which is the theoretically minimal (optimal) number for each partition problem, initial weights randomly chosen from -1.0 to 1.0 (same as in the evolutionary simulations), and the MSE results averaged over 50 runs. With all other conditions set to be the same, post-training errors with the evolved networks are significantly less for each problem (p values on t-test were less than 10^{-5} for each of the four comparisons). More importantly, the fully connected networks sometimes produced totally wrong answers (i.e., absolute errors in output node values were more than 0.5), while this problem did not occur with the evolved networks. This shows the value of searching the architectural space even if it is believed that a fully connected network can theoretically approximate any function [12] ([49, 50] for general discussion). The Average time column in Table 3 shows the mean time (seconds) needed for a single evolutionary run. This result shows that our system can identify the partial relationships between input and output patterns and represent them within an appropriate modular architecture.

3.2 Learning Word Pronunciations Using Recurrent Networks

Recurrent neural networks have long been of interest for many reasons. For example, they can learn temporal patterns and produce a sequence of outputs, and are widely found in biological nervous systems [27]. They have been applied in many different areas (e.g., word pronunciation [39], and learning formal grammars [18]) and several models of recurrent neural networks have been proposed (see [23]). Here we establish that our high-level descriptive

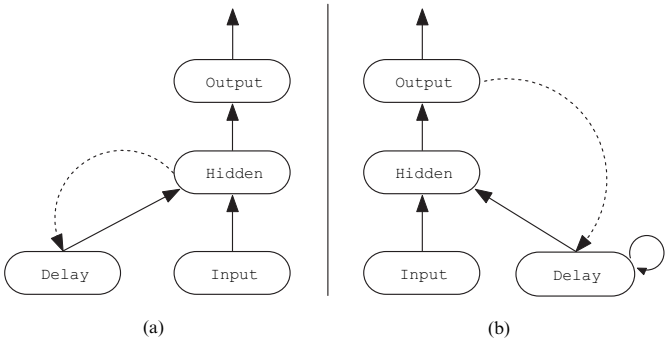


Fig. 6. The Elman (**a**) and Jordan (**b**) network architectures. *Dotted lines* show the backward/recurrent one-to-one connections that essentially represent a copying of the output at one time step to a delay layer that serves as input at the next time step

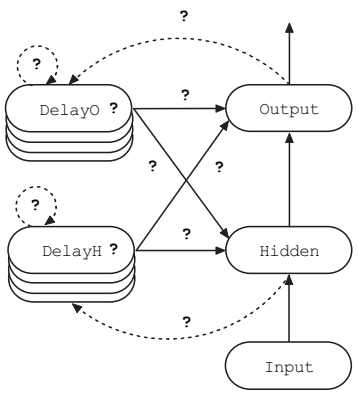
language is sufficiently powerful to support the evolution of recurrent networks in situations involving multi-objective optimization.

Two well-known, partially recurrent architectures that let one use basic error backpropagation from feed-forward nets essentially unchanged (because feedback connection weights are fixed and unlearnable) are often referred to as Elman networks and Jordan networks. Elman [16] suggested a recurrent network architecture in which a copy of the contents of the hidden layer (saved in the delay layer) acts as a part of the input data in the next time step, as shown in Fig. 6a. Jordan [24] proposed a similar architecture except that the content of the output layer is fed back to the delay layer where nodes possibly also have a "decaying" self-connection, as shown in Fig. 6b. These networks were originally proposed for different purposes: the Elman architecture for predicting the next element in a temporal sequence of inputs, and the Jordan architecture for generating a temporal sequence of outputs when given a single fixed input pattern. However, little is known about how to select the best recurrent network architecture for a given sequence processing task and, to our knowledge, no systematic experimental comparison between these different recurrent neural network architectures has ever been undertaken, except for some specific application comparisons (e.g., [37]).

In this context, our second problem is to find appropriate recurrent networks to produce a sequence of phoneme outputs, given a *fixed* input of the corresponding input word pattern. For example, for the word *apple*, a fixed pattern of the five letters A P P L E is the input, and the correct output temporal sequence of phonemes would be /ae/, /p/, and /l/, followed by an end of word signal. This challenging task was originally tackled in [39] using Jordan networks, and here we expand the size of input data (total of 230, two to six phoneme words selected randomly from the NetTalk corpus [44]), and focus on finding the optimal architecture of delay layers and their connectivity.

```
[SEQUENCE Psg_problem
   [LAYER Input   [SIZE 156] [CONNECT Hidden]]
   [LAYER Hidden  [SIZE  52]
      [[CONNECT DelayH:1] [CONNECT_RADIUS 0.0]
       [LEARN_RULE NONE]]
      [[CONNECT Output]]]
   [SEQUENCE DelayH
      [LAYER [NUM_LAYER [EVOLVE 0 4]] [SIZE 52]
         [[CONNECT FWD] [CONNECT_RADIUS 0.0]
          [LEARN_RULE NONE]]
         [[CONNECT [EVOLVE Hidden Output]]]]]
   [LAYER Output  [SIZE  52]
      [[CONNECT DelayO:1] [CONNECT_RADIUS 0.0]
       [LEARN_RULE NONE]]]
   [SEQUENCE DelayO
      [LAYER [NUM_LAYER [EVOLVE 0 4]] [SIZE 52]
         [[CONNECT FWD] [CONNECT_RADIUS 0.0]
          [LEARN_RULE NONE]]
         [[CONNECT [EVOLVE Hidden Output]]]]]
]
```

(a)



(b)

Fig. 7. (**a**) The network description file for the phoneme sequence generation (psg) task. FWD, delayH:1, and delayO:1 mean to make a connection to the next layer in the same network block, to the first layer in the delayH network block, and to the first layer in the delayO network block, respectively. If such a block does not exist, the corresponding connectivity properties are ignored. The evolvable properties are in *bold font*. (**b**) A schematic illustration of the space of neural network architectures corresponding to the description file in (**a**) that are to be searched for the phoneme sequence generation problem. *Dotted lines* designate non-trainable, one-to-one feedback connections; *solid lines* indicate weighted, fully connected pathways trained by error backpropagation. Note that the Elman and Jordan networks of Fig. 6 are included within this space as special cases

The question being asked is whether the high-level, modular developmental approach supported by our language can identify the "best" recurrent architecture to use, or at least clarify the tradeoffs.

Figure 7 gives the descriptive encoding and a corresponding schematic representation of the space of networks that we used for this problem. The fixed part of the structure is a feed-forward, three layer network consisting of input, hidden, and output layers (depicted on the right side of Fig. 7b). The size of the input layer is decided by the maximum length of a word in our training data and the encoding representation, and the output layer size is 52 since we use a set of 52 output phonemes, including the end of a word signal. The number of hidden nodes is arbitrarily set to be the same as the output layer size. Note that hidden and delay layers may have connections to different destination layers with different configurations (e.g., CONNECT_RADIUS and LEARN_RULE), such that each set of properties for connections has been separated from the other by using double brackets in the description of Fig. 7a. As shown here, the space of architectures to be searched by the evolutionary process consists of varying numbers of delay layers that receive recurrent one-to-one feedback connections (dotted arrows) from either the hidden layer

(delayH) or the output layers (delayO). In either case, 0–4 delay layers may be evolved, but however many are evolved in each feedback pathway, they must be organized in a serial fashion, thus representing feedback delays of 0–4 time steps. Both feedback pathways from output and hidden layers may have zero layers, which means there are four possible architectures being considered during evolution: (1) feed-forward network only without delays; (2) hidden layer feedback only; (3) output layer feedback only; and (4) both types of feedback. In addition, for each class where a feedback pathway exists, it may have a varying amount of delay (1–4 time steps) and may provide feedback to the output layer, the hidden layer, or both. Each delay layer is sequentially connected to the adjacent delay layer by a one-to-one, fixed connection of weight 0.5, which acts as a decaying self-connection. Thus a total of 169 architectures are considered by the evolutionary process.[2]

We based fitness criteria on two cost measures or objectives: root mean squared error (RMSE) for performance, which was adjusted to be comparable with previous results [39], and the total sum of absolute weight values to penalize larger networks. The latter unbounded measure simply adds together the absolute values of all weights in the network after training. We used it rather than the number of nodes and connections as in n-partition problems, since the latter vary stepwise in this experiment while the summed weights are a continuous measure. This summed absolute weights measure is especially useful here as it can potentially discriminate between two different architectures having the same numbers of node and connections (e.g., Jordan vs Elman networks). Similar weight minimization fitness criteria have been used previously evolving neural networks and can be viewed as a "regularization term" that acts indirectly on an evolutionary time scale rather than directly during the learning process (see [45] for discussion). We adopted a multi-objective evolutionary algorithm (SPEA [52]) based on these two fitness criteria, which enables one to get a sense of the tradeoffs in performance and parsimony among the different good architectures found during evolution. This also illustrates that our system consists of components that can be expanded or plugged in depending on the specific problem. The population size was decreased to 25 because of the large computational expense of learning and evolution, and the archive in which non-dominated individuals are stored externally in SPEA was set to be the same size as the population. The maximum number of generations was fixed at 50, and all networks trained for 200 epochs with RPROP, a variant of error backpropagation which has been shown to be very effective in previous research [39]. For genetic operations, only mutating the number of layers and their connectivity are allowed, specified by default and applied within the range of property values designated in the network description file.

[2] No delay: 1, delayH only: 4 delays $\times$ 3 directions, delayO only: 4×3, both delays: $(4 \times 3) \times (4 \times 3) = 169$.

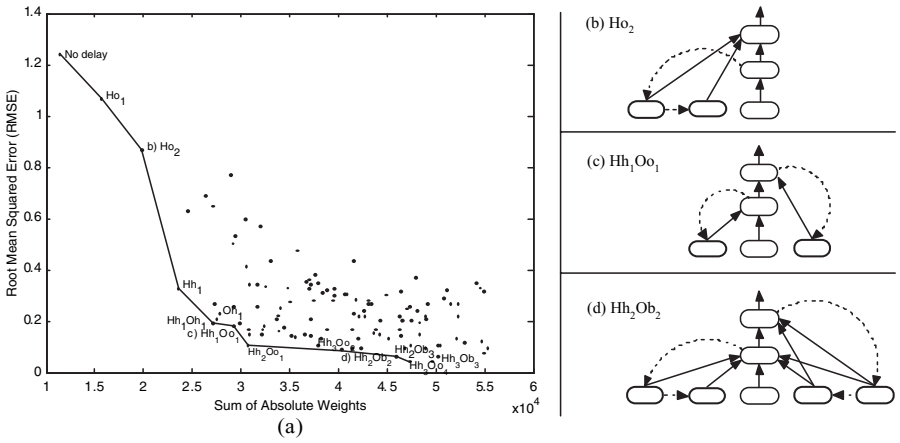


Fig. 8. (**a**) The performance/weights result of networks from all final generations are depicted. Each point represents one network architecture's values averaged over all evolutionary runs (most points are not labeled). The points on the *solid line* represent the Pareto optimal set, and the labels on some of these latter points designate the type of network that they represent. For example, label Hh_3Oh_1 means that the network represented by that node has both hidden (H) and output (O) delay layers, while there are three hidden and one output delay layers, in both cases connected to the hidden (h) layer (see text). (**b**)–(**d**) Example of evolved network architectures. Evolved layers are shown in *bold ovals*

We ran a total of 100 runs, randomly changing the initial network architectures and their weights in each run. The results are shown in Fig. 8, averaged over the same architectures (i.e., each point in Fig. 8 represents a network having a specific number of hidden and output delays, and a specific layer-to-layer connectivity, with the RMSE and weight sum averaged over all runs). In a label "H$c_{\#}$O$c_{\#}$" in Fig. 8, # indicates the number of hidden (H) or output (O) delays, and c designates the destination of delay outputs, either to the hidden (h), output (o), or both (b) layers. For example, "Hh_1Ob_2" means that there is one hidden delay layer (connected back to the hidden layer) and two output delay layers connected back to both hidden and output layer. Figure 8b–d shows some other examples of evolved network architectures. An example of the final network descriptions for Elman and Jordan networks is illustrated in Fig. 9.

Several observations can be made from Fig. 8. First, and surprising to us, feed-forward only networks without delays still remain in the final Pareto optimal set (upper left). The Pareto optimal set in this context consists of "non-dominated" neural networks for which no other neural network has been found during evolution that is better on all of the objective fitness criteria. Thus, feed-forward networks are included in the Pareto optimal set because of their quite small weight values, even though their performance is poor relative

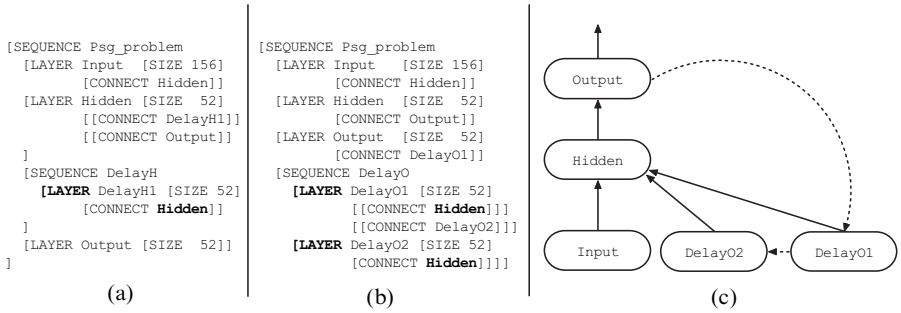


Fig. 9. The final network description of (**a**) an Elman network with single delay (labeled "Hh_1" in the text) and (**b**) a Jordan-like network with double delays (labeled "Oh_2"). Only SIZE and CONNECT properties are shown. The evolved properties (including the number of layers) are in *bold font*. (**c**) An illustration of the Jordan network specified in (**b**). *Dotted lines* designate fixed, one-to-one connections

to the other types. Following the Pareto optimal front downward, we see that networks with one or two hidden delay layers connected to the output layer (labeled "Ho_1" and "Ho_2") are the next Pareto-front points (upper left of Fig. 8). This type of network in which delays are connected to the output layer does not provide good performance in general. A big increase in performance occurs however with networks having only hidden delay layers connected to the hidden layer (bottom left): an Elman network (labeled "Hh_1" in Fig. 8 and depicted in Fig. 9a) performs much better and is on the Pareto front. A Jordan network (labeled "Oh_1", lower right of "Hh_1" in Fig. 8) performs even better at the cost of increased weights. Finally, networks with increasing numbers of delay layers that combine hidden and output delays generally performed progressively better, although at the cost of increasing numbers of weights and connections (bottom right in Fig. 8). Surprisingly, "Hh_1Oh_1" on the Pareto front of Fig. 8 performs better than the original Elman (Hh_1) and Jordan (Oh_1) networks with smaller total weight values than the latter, and would be a very good choice for an architecture for this problem (and one that was not evident prior to the evolutionary process). Summarizing, the Pareto-optimal front in Fig. 8 and, more generally, the correlations between architectures and performance given in Table 4, explicitly lay out the tradeoffs for the human designer selecting an architecture. From a practical point of view, which Pareto optimal architecture one would adopt depends on the relative importance one assigns to error minimization vs network size in a specific application. A very reasonable choice would be networks such as Hh_2Oo_1 or Hh_2Ob_2 (Fig. 8d) that produce low error by combining features from both Jordan and Elman networks while still being constrained in size. These results show that our evolutionary approach using a high-level descriptive language can be applied effectively in generating and evaluating alternative neural networks even for complex temporal tasks requiring recurrent networks.

Table 4. Representative results for the phoneme sequence generation problem

Architecture	RMSE	Absolute weight sum	PCT[a]
No delay	1.239	11,410.4	23.3
Ho_1	1.066	15,711.7	43.2
Ho_2	0.869	19,892.6	62.3
Oo_1	0.768	28,909.3	70.5
Ob_1	0.599	30,502.8	82.1
Hb_2	0.531	29,421.4	85.9
Ho_1Ob_1	0.501	29,243.5	87.5
Hh_1	0.328	23,604.5	94.6
Hh_2	0.270	27,232.6	96.4
Hh_3	0.255	29,250.4	96.7
Oh_1	0.232	27,893.6	97.3
Hb_3Oo_4	0.219	48,989.6	97.6
Hh_2Oh_1	0.210	27,540.5	97.8
Hh_1Oh_1	0.191	27,090.4	98.2
Hh_1Oo_1	0.180	29,290.6	98.4
Hh_3Oo_1	0.152	32,109.6	98.9
Hh_2Oo_1	0.107	30,718.3	99.4
Hh_3Oo_2	0.107	37,946.3	99.4
Hh_2Ob_2	0.088	40,383.5	99.6
Hh_2Ob_3	0.062	45,920.2	99.8
Hh_3Ob_3	0.044	49,589.8	99.9

[a] PCT = Percentage of phonemes generated completely correctly [39]

4 Discussion

Recent advances in neuroevolutionary methods have repeatedly been successful in creating innovative neural network designs [2, 7, 11, 19, 20, 31, 42, 43, 51], but these successes have had little practical influence on the field of neural computation. We believe this is partially because of a dilemma: the general space of neural network architectures and methods is so large that it is impractical to search efficiently, yet attempting to avoid this problem by hand-crafting the evolution of neural networks on a case-by-case basis is very labor intensive and thus also impractical.

In this context, we explored the hypothesis that a high-level descriptive language can be used effectively to support the evolutionary design of task-specific neural networks. This approach addresses the impracticality of searching the enormous general space of neural networks by allowing a designer to easily restrict the search space to architectures and methods that appear a priori to be relevant to a specific application, greatly reducing the size of the space that an evolutionary process must search. It also greatly reduces the time needed to create a network's design by allowing one to describe the class of neural networks of interest at a very high level in terms of possible modules and inter-module pathways, rather than in terms of individual

neurons and their connections. Filling in the "low level" details of individual networks in an evolving population is left to an automated developmental process (the neural networks are "grown" from their genetic encoding) and to well-established neural learning methods that create connection weights prior to fitness assessment.

In this chapter, we have presented experimental results suggesting that evolutionary algorithms can be successfully applied to the automated design of neural networks, addressing the dilemma stated above using a novel encoding scheme. We showed that human-readable description files could guide an evolutionary process to produce near-optimal modular solutions to n-partition problems. We also showed that this approach could not only create effective recurrent architectures for temporal sequence generation, but that it could simultaneously indicate the tradeoffs in the costs of architectural features versus network performance (via multi-objective evolution). All that was needed to guide the evolutionary processes in both of these applications was short description files like those illustrated in Figs. 3a and 7a.

References

1. Abraham A (2002) Optimization of evolutionary neural networks using hybrid learning algorithms. In: Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN '02), IEEE, vol. 3, pp. 2797–2802
2. Balakrishnan K, Honavar V (2001) Evolving neuro-controllers and sensors for artificial agents. In: Advances in the Evolutionary Synthesis of Intelligent Agents, MIT, pp. 109–152
3. Banzhf W, Nordin P, Keller RE, Francone FD (1997) Genetic Programming: An Introduction. Morgan Kaufmann, San Francisco, CA
4. Belew RK, McInerney J, Schraudolph NN (1991) Evolving networks: Using the genetic algorithm with connectionist learning. CSE Technical Report #CS90-174, Computer Science and Engineering Department, UCSD
5. Bentely PJ, Corne DW (2001) Creative Evolutionary Systems. Morgan Kaufmann, San Francisco, CA
6. Blum AL, Rivest RL (1992) Training a 3-node neural network is np-complete. Neural Networks 5(1):117–127
7. Bonissone PP, Subbu R, Eklund N, Kiehhl TR (2006) Evolutionary algorithms + domain knowledge = real-world evolutionary computation. IEEE Transactions on Evolutionary Computation 10(3):256–280
8. Brown M, Keynes R, Lumsden A (2001) The Developing Brain. Oxford University Press, Oxford
9. Cangelosi A, Parisi D, Nolfi S (1994) Cell division and migration in a genotype for neural networks. Network: Computation in Neural Systems 5(4):497–515
10. Chalmers DJ (1990) The evolution of learning: An experiment in genetic connectionism. In: Touretsky DS, Elman JL, Sejnowski TJ, Hinton GE (eds.) Proceedings of the 1990 Connectionist Summer School, Morgan Kaufmann, pp. 81–90, URL citeseer.ist.psu.edu/chalmers90evolution.html

11. Chong SY, Tan MK, White JD (2005) Observing the evolution of neural networks learning to play the game of othello. IEEE Transactions on Evolutionary Computation 9(3):240–251
12. Cybenko G (1989) Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems 2(4):303–314
13. De Garis H (1991) GenNETS: Genetically programmed neural nets using the genetic algorithm to train neural nets whose inputs and/or output vary in time. In: Proceedings of the International Joint Conference on Neural Networks (5th IJCNN '91), IEEE, Singapore, vol. 2, pp. 1391–1396
14. Dill FA, Deer BC (1991) An exploration of genetic algorithms for the selection of connection weights in dynamical neural networks. In: Proceedings of the IEEE 1991 National Aerospace and Electronics Conference (NAECON '91), IEEE, New York, NY, Dayton, OH, vol. 3, pp. 1111–1115
15. Dimond S, Blizard D (1977) Evolution and Lateralization of the Brain. New York Academy of Sciences, New York
16. Elman JE (1990) Finding structure in time. Cognitive Science 14(2):179–211
17. Ferdinando AD, Calabretta R, Parisi D (2001) Evolving modular architectures for neural networks. In: French R, Sougné J (eds.) Proceedings Sixth Neural Computation and Psychology Workshop Evolution, Learning, and Development
18. Giles CL, Miller CB, Chen D, Sun GZ, Chen HH, Lee YC (1992) Extracting and learning an *unknown*, grammar with recurrent neural networks. In: Moody JE, Hanson SJ, Lippmann RP (eds.) Advances in Neural Information Processing Systems 4, Morgan Kaufmann, Denver, CO, pp. 317–324
19. Gruau F (1995) Automatic definition of modular neural networks. Adaptive Behavior 3:151–183
20. Gruau F, Whitley D, Pyeatt L (1996) A comparison between cellular encoding and direct encoding for genetic neural networks. In: Proceedings of the Sixth International Conference on Genetic Programming, Stanford University Press
21. Grushin A, Reggia JA (2005) Evolving processing speed asymmetries and hemispheric interactions in a neural network model. Neurocomputing 65:47–53
22. Harp S, Samad T, Guha A (1989) Towards the genetic synthesis of neural networks. In: Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, pp. 360–369
23. Haykin S (1999) Neural Networks: A Comprehensive Foundation. Prentice Hall, Upper Saddle River, NJ
24. Jordan MI (1986) Attractor dynamics and parallelism in a connectionist sequential machine. In: Proceedings of the Eighth Conference of the Cognitive Science Society, Erlbaum, pp. 531–546
25. Jung JY, Reggia JA (2004) A descriptive encoding language for evolving modular neural networks. In: Genetic and Evolutionary Computation – GECCO-2004, Part II, Springer, Lecture Notes in Computer Science, vol. 3103, pp. 519–530
26. Jung JY, Reggia JA (2006) Evolutionary design of neural network architectures using a descriptive encoding language. IEEE Transactions on Evolutionary Computation 10:676–688
27. Kandel E, Schwartz J, Jessel T (1991) Principles of Neural Science. Appleton and Lange, Norwalk, CT
28. Killackey H (1996) Evolution of the human brain: A neuroanatomical perspective. In: Gazzaniga M (ed.) The Cognitive Neurosciences, MIT, pp. 1243–1253

29. Kitano H (1994) Neurogenetic learning: An integrated method of designing and training neural networks using genetic algorithms. Physica D 75:225–238
30. Koza J, Bennett F, Andre D, Keane M (1999) Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, San Francisco, CA
31. Lehmann KA, Kaufmann M (2005) Evolutionary algorithms for the self-organized evolution of networks. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, New York, NY, USA, pp. 563–570, DOI http://doi.acm.org/10.1145/1068009.1068105
32. Mehrotra K, Mohan CK, Ranka S (1997) Elements of Artificial Neural Networks. MIT, Cambridge, MA
33. Miller GF, Todd PM, Hegde SU (1989) Designing neural networks using genetic algorithms. In: Proceedings of third International Conference on Genetic algorithms (ICGA89), pp. 379–384
34. Mitchell M (1996) An Introduction to Genetic Algorithms. MIT, Cambridge, MA
35. Montana D, Davis L (1990) Training feedforward neural networks using genetic algorithms. In: Proceedings of eleventh International Joint Conference on Artificial Intelligence, Morgan Kaufmann, pp. 370–374
36. Mountcastle V (1998) The Cerebral Cortex. Harvard University Press, Cambridge, MA
37. Pérez-Ortiz J, Calera-Rubio J, Forcada M (2001) A comparison between recurrent neural architectures for real-time nonlinear prediction of speech signals. In: Miller D, Adali T, Larsen J, Hulle MV, Douglas S (eds.) Neural Networks for Signal Processing XI, Proceedings of the 2001 IEEE Neural Networks for Signal Processing Workshop (NNSP '01), IEEE Signal Processing Society, pp. 73–81
38. Radi A, Poli R (1998) Genetic programming can discover fast and general learning rules for neural networks. In: Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, Riolo R (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, pp. 314–323
39. Radio MJ, Reggia JA, Berndt RS (2001) Learning word pronunciations using a recurrent neural network. In: Proceedings of International Joint Conference on Neural Networks (IJCNN '01), vol. 1, pp. 11–15
40. Riedmiller M (1994) Advanced supervised learning in multi-layer perceptrons: from backpropagation to adaptative learning algorithms. Computer Standards & Interfaces 16(3):265–278
41. Riedmiller M, Braun H (1993) A direct adaptive method for faster backpropagation learning: The rprop algorithm. In: Proceedings of 1993 IEEE International Conference on Neural Networks, vol. 1, pp. 586–591
42. Ruppin E (2002) Evolutionary autonomous agents: A neuroscience perspective. Nature Reviews Neuroscience 3(2):132–141
43. Saravanan N, Fogel D (1995) Evolving neural control systems. IEEE Expert 10:23–27
44. Sejnowski T, Rosenberg C (1987) Parallel networks that learn to pronounce english text. Complex Systems 1:145–168
45. Shkuro Y, Reggia JA (2003) Cost minimization during simulated evolution of paired neural networks leads to asymmetries and specialization. Cognitive Systems Research 4(4):365–383

46. Srinivas M, Patnaik LM (1991) Learning neural network weights using genetic algorithms- improving performance by search-space reduction. In: 1991 IEEE International Joint Conference on Neural Networks, IEEE, Singapore, vol. 3, pp. 2331–2336
47. Stanley KO, Miikkulainen R (2002) Evolving neural network through augmenting topologies. Evolutionary Computation 10(2):99–127
48. Tooby J, Cosmides L (2000) Toward mapping the evolved functional organization of mind and brain. In: Gazzinga M (ed.) The New Cognitive Neurosciences, MIT, pp. 1167–1178
49. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1):67–82
50. Wolpert DH, Macready WG (2005) Coevolutionary free lunches. IEEE Transactions on Evolutionary Computation 9(6):721–735
51. Yao X (1999) Evolving artificial neural networks. Proceedings of the IEEE 87(9):1423–1447
52. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. IEEE Transactions on Evolutionary Computation 3(4):257–271

http://www.springer.com/978-3-540-76285-0

Success in Evolutionary Computation
Yang, A.; Shan, Y.; Bui, L.T. (Eds.)
2008, VIII, 372 p., Hardcover
ISBN: 978-3-540-76285-0