ELeaRNT: Evolutionary Learning of Rich Neural Network Topologies

Matteo Matteucci
Center for Automated Learning and Discovery
Technical Report N. CMU-CALD-02-103
matteo@cs.cmu.edu

Advisor:

Manuela Veloso
veloso@cs.cmu.edu

In this paper we present ELeaRNT an evolutionary strategy which evolves rich neural network topologies in order to find an optimal domain–specific non–linear function approximator with a good generalization performance. The neural networks evolved by the algorithm have a feed–forward topology with shortcut connections and arbitrary activation functions at each layer. This kind of topologies has not been thoroughly investigated in literature, but is particularly well suited for non–linear regression tasks.

The experimental results prove that, in such tasks, our algorithm can build, in a completely automated way, neural network topologies able to outperform classic neural network models designed by hand. Also when applied to classification problems, the performance of the obtained neural networks is fully comparable to that of classic neural networks and in some cases noticeably better.

Center for Automated Learning and Discovery
School of Computer Science
Carnegie Mellon University

ELeaRNT: Evolutionary Learning of Rich Neural Network Topologies

Matteo Matteucci
Center for Automated Learning and Discovery
School of Computer Science
Carnegie Mellon University
matteo@cs.cmu.edu

***Abstract*— In this paper we present ELeaRNT, an evolutionary strategy which evolves rich neural network topologies in order to find an optimal domain–specific non–linear function approximator with a good generalization performance. The neural networks evolved by the algorithm have a feed–forward topology with shortcut connections and arbitrary activation functions at each layer. This kind of topologies has not been thoroughly investigated in literature, but is particularly well suited for non–linear regression tasks. The experimental results prove that, in such tasks, our algorithm can build, in a completely automated way, neural network topologies able to outperform classic neural network models designed by hand. Also when applied to classification problems, the performance of the obtained neural networks is fully comparable to that of classic neural networks and in some cases noticeably better.**

I. Introduction

Artificial neural networks are generic non–linear function approximators. In the literature, they have been used extensively for various purposes like regression, classification, and feature reduction. A neural network is a collection of basic units, neurons, computing a non–linear function of their input. Every input has an assigned weight that determines the impact this input has on the output of the node. By interconnecting the correct number of nodes in a suitable way and setting the weights to appropriate values, a neural network can approximate any function, linear or non–linear. This structure of nodes and connections, known as the network's topology, together with the weights of the connections, determines the network's final behavior.

Given a neural network topology and a training set, it is possible to optimize the values of the weights in order to minimize an error function by means of any backpropagation–based algorithm or standard optimization techniques. During this learning phase, the topology of the neural network plays a critical role in whether or not the network can be trained to learn a particular dataset [13], but no algorithm exists for finding an optimal solution for the design of the topology.

We cannot easily answer the question of how many nodes and connections a neural network should have. Clearly, the simpler the topology, the simpler the function the neural network is computing. A simple topology will result in a network that cannot learn to approximate a complex function, while a large topology is likely to result in a network losing its generalization capability. This loss of generalization is the result of *overfitting* the training data: instead of approximating a function present in the data, a neural network that has an overly complex structure may have the ability to memorize the dataset, allowing noise within the data to be learned, resulting in inaccurate predictions on future samples.

Producing or deciding what will be a suitable network topology is a task that is usually solved with heuristic algorithms or left to human experts. The process of "manually" design such a topology is iterative, often requires a certain amount of expertise, and it is definitively tedious. There is no known automatic way to systematically create an optimal or even near-optimal topology for a specific task. Moreover, the correct topology is application–dependent, so any method that aims to create the correct topology should be mostly data driven.

In this paper, we present ELeaRNT an evolutionary strategy capable of finding an effective application–specific network topology by searching the space of possible rich topologies. The search space is composed of feed–forward topologies with shortcut connections and abitrary non–linear activation functions for each layer. Our claim is that, by means of topologies enriched by the use of different non–linear bases, it is possible to improve the performance of the learned model with respect to classical neural network architectures that use only sigmoidal functions for hidden layers and linear functions for the output. Our research starts from the work presented in [5] and aims to provide an automatic tool for rich topology design.

In the next section, we briefly summarize the state of the art in neural network topology search. Section III gives a general description of the ELeaRNT algorithm and in section IV, V, and VI we presents the details of the algorithm implementation. Section VII shows the empirical validation of the algorithm on synthetic and real datasets; the conclusion and future work are presented in section VIII.

II. Related work

The problem of finding an optimal topology can be thought of as a search problem, where the search space is the space of all possible network topologies, and where the goal is to minimize an error function while preserving generalization capabilities.

Matteo Matteucci is also at Dipartimento di Elettronica e Informazione – Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy. http://www.elet.polimi.it/~matteucc

Usually, the neural network connectivity and the activation functions are fixed by the human designer and the only parameters for model selection are the number of neurons for each layer and, in some occasions, the number of layers. Those parameters are then optimized with respect to the dataset with a *trial & error* procedure or using *cross–validation*.

More often a *destructive* search is used: first the algorithm trains a big neural network on the data, and then prunes it to increase its generalization capability while preserving its accuracy. The main issue with this kind of approach is the choice of the initial network that has to be big "enough" in order to be effectively pruned. In this case, techniques like Optimal Brain Damage [14] or Optimal Brain Surgeon [10] are used to selectively remove connections while maintaining network accuracy.

To overcome the issue of choosing an initial big network to be trained and to reduce the computational complexity of training over–sized networks, *constructive* algorithms start with a small neural network and then add nodes and connections during training [3]. In regression tasks the cascade correlation algorithm [4] is usually used. I uses the correlation between the actual network error and the input to train new nodes to be added to the network. This algorithm is particularly known for the use of fast training algorithms (e.g., quickprop) and for training a single neuron at a time in order to speed up the architecture development.

Finally, there are many works in literature on using *genetic algorithms* to optimize neural networks, as can be seen from the numerous papers on the topic [25] [2]. Genetic algorithms have been used to optimize almost all the parameters that characterize a neural network (e.g. weights, learning algorithm, topology). The algorithm we present in this paper belongs to the general framework for neural network evolution presented in [26] and focuses on the development of rich neural network topologies using an evolutionary strategy. The main interest in this kind of topology is to state the effectiveness of using various activation functions for the network layers [17] [19] [16] using standard benchmarks.

III. The ELeaRNT Algorithm

Genetic algorithms have proved to be a powerful search tool when the search space is large and multimodal, and when it is not possible to write an analytical form for the error function in such a space. In these applications, genetic algorithms excel because they can simultaneously and thoroughly explore many different parts of a large solution space seeking a suitable solution. At first, completely random solutions are tried and evaluated according to a fitness function, and then the best ones are combined using specific operators. This gives the ability to adequately explore possible solutions while, at the same time, preserving from each solution the parts which work properly.

In Figure 1 we show the general scheme for a genetic algorithm. The initial population is randomly initialized choosing for each individual the number of layers according to a uniform distribution. Once the number of layers is defined, the number of neurons and the activation function for each of them is randomly chosen using again a uniform distribution. At this point the connectivity of the network in generated checking the fan–in and fan–out of each layer in order to prune the unused ones (see section IV-B).

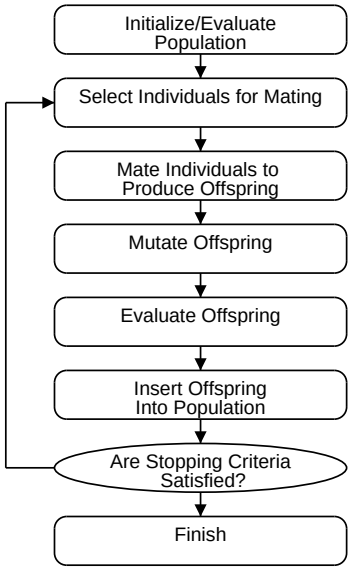


Fig. 1. General schema for the genetic algorithm

After initializing the first population of random solutions, each individual is evaluated, its fitness is computed[1], and their topologies are ranked according to it. We evolve the population by selecting the individuals according to their fitness and stochastically applying to them the genetic operators *crossover* and *mutation*. Once a new offspring has been generated, the new individuals are trained and their fitness is evaluated. This process continues until a stopping criterium is met.

The basic genetic algorithm used in our implementation is the Simple Genetic Algorithm Goldberg describes in his book [7]. It uses non-overlapping populations and at each generation creates an entirely new population of individuals by selecting from the previous population, and then mating them to produce the offspring for the new population. In all our experiments we use *elitism*, meaning that the best individual from each generation is carried over to the next generation[2]. Once the evolution is terminated, the best individual defines the optimized network topology for the specific application. During the evolution we do not keep track of the weights learned during the fitness evaluations so, once the best individual has been selected, the rich neural network is trained from a different starting point and the best set of weights in cross-validation is selected.

Figure 2 presents a detailed description of the ELeaRNT algorithm. In our setting we designed two crossover operators (single point crossover and two point crossover), and we implemented six different types of mutation in order to explore the whole solution space in an effective way (see section V). The genetic operators of our algorithm are designed taking into ac-

[1]Genetic algorithms can be used also for network weights optimization, but they are best suitable for global search instead of local optimization. We prefer not to use them for weight optimization; in common applications it is possible to evaluate the error function and its derivative and effectively apply classic optimization techniques. In our implementation we use Polak-Ribiere deterministic conjugate gradient with golden line search [6].

[2]We will show in the experimental section that, due to the convergence of the population fitness, elitism is not mandatory to the effectiveness of the evolutionary search process.

Initialize Population with N Random [valid] Individuals
Evaluate Fitness for all Individuals in the Population
do
  do
    Select Individuals I_1 and I_2 According to Fitness
    with Probability p_{cross}
      Select Crossover Operator $\mathbf{Cross}$ to Apply
      $I'_1, I'_2 \leftarrow \mathbf{Cross}(I_1, I_2)$
    with Probability $1 - p_{cross}$
      $I'_1 \leftarrow I_1$
      $I'_2 \leftarrow I_2$
    with Probability p_{mut}
      Select Mutation Operator $\mathbf{Mut}$ to Apply
      $I''_1 \leftarrow \mathbf{Mut}(I'_1)$
    with Probability $1 - p_{mut}$
      $I''_1 \leftarrow I'_1$
    with Probability p_{mut}
      Select Mutation Operator $\mathbf{Mut}$ to Apply
      $I''_2 \leftarrow \mathbf{Mut}(I'_2)$
    with Probability $1 - p_{mut}$
      $I''_2 \leftarrow I'_2$
    Add Individuals I''_1 and I''_2 to New Population
  until(Created New Population)
  Evaluate Fitness for New Individuals in Population
until(Meet Stopping Criterium)

Fig. 2. The ELeaRNT algorithm

count their result on the evolved network implementing in such a way an *evolutionary strategy* [12] to learn rich neural topologies.

As pointed out in [20], any algorithm attempting to design a genetic algorithm capable of generating an application–specific neural network topology has to address the following open–ended questions (our answers will be the topic of the next sections):

- How should a neural network's topology be *represented* in a genotype[3]?
- How are the *genetic operators* defined for the particular representation?
- How should a *topology's fitness* be calculated?

IV. Network Representation

The first step in the design of a genetic algorithm is deciding on the genotype, that is how to represent a neural network topology as a chromosome. There are two mainstream approaches in the research community: direct and indirected encoding. In direct encoding, every detail of the architecture (i.e., number of neurons, activation functions, connections, etc.) is specified in the genotype. In indirect encoding, the chromosome consists only of information about how the network should be constructed, such as a set of rules for building its architecture.

The main argument for this second, less explicit, encoding schema its the biological plausibility. DNA, the encoding method used as a blueprint for the human body, is clearly not large enough to contain a direct encoding of the structure of the human brain. Apparently, DNA contains some sort of growth rules to instruct the creation of the brain [11]. From our point of view the performance with respect to the application and the generalization ability is what matters and not biological plausibility, hence we choose the direct coding schema. A direct coding allows us a more focused design of the genetic operators that result to be closed with respect to the chosen phenotype[4].

[3]The term *genotype* refers to the symbolic representation of each possible solution, and the term *phenotype* refers to its realization.

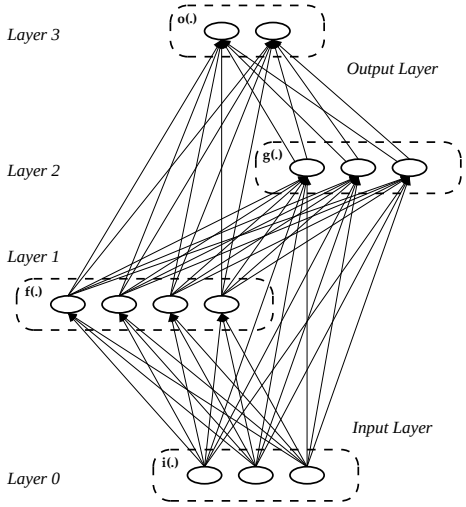


Fig. 3. An example of a phenotype evolved by our evolutionary strategy

Direct encoding has been proved to be less effective with larger genotypes, because the effects of crossover and mutation are often unfavorable for retaining any kind of high level network structure that may have been evolved [15]. For this reason, the coding we propose in section IV-B is suitable for keeping the networks representation compact, trying to avoid the "competing convention" issue that arises from the fact that there is no significance to the order of the nodes in the hidden layers of neural networks [8].

A. The Network Model: Phenotype

The term phenotype defines the implemented non–linear model once the chromosome is decoded in a neural network. The genetic algorithm we implemented evolves feed-forward topology neural networks with shortcut connections and several activation functions (i.e., logistic, tanh, linear, exp, gauss, sin, cos), eventually applying a different function for each neuron. An example of a topology we are designing is represented in Figure 3. The neural network is composed of N layers; each layer is fully connected with some of the other layers in a feed-forward manner implementing a generalized feed-forward topology.

Each layer has at least one neuron, and a different activation function. The number of neurons in the first and last layers is fixed, since that is the number of input and output variables of the specific problem. The transfer function for the input layer is usually the *identity function* and for the other layers it can be any of the following: *identity*, *logistic*, *tanh*, *linear*, *exp*,

[4]i.e. applying a genetic operator to a valid genotype coding a rich neural network topology produces another valid genotype coding another rich neural network topology.

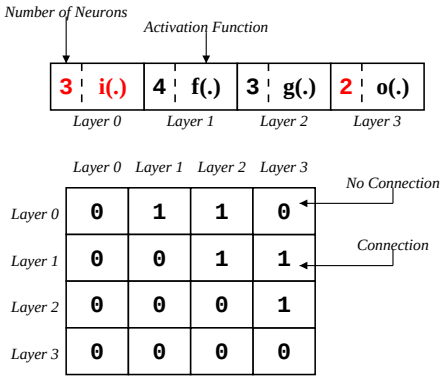


Fig. 4. The genotype for the network in Figure 3

gaussian, *sin*, *cos*. This phenotype subsumes the classical fully connected feed-forward architecture and allows more flexibility for the use of various activation functions and for the capability of describing non-fully connected topologies with shortcut connections.

B. The Genetic Coding: Genotype

Each phenotype is coded by a two part genotype. The first part encodes the layer information (i.e. number of neurons and activation function), and the second part encodes the connectivity of the network (see Figure 4). The number of neurons in the first and in the last layer is fixed since they are fixed by the specific application. To specify a proper feed–forward neural network only the elements above the diagonal in the adjacency matrix can differ from 0. Since we chose the identity function for the first layer, the activation function for that part of the genotype cannot be changed during the evolution.

It is possible that during the genetic evolution a genotype codes an "invalid" phenotype. That happens when either a column (i.e., the fan–in a neuron layer) or a row (i.e., the fan–out of a neuron layer) is filled with 0s implying that a group of neurons is not reachable from the input or does not participate in the final output. To avoid this issue, the genetic operators have been designed as closed to the phenotype family and, in case they detect any irregularity, they correct the genotype.

V. Genetic Operators

In our implementation we define two crossover operators, and six different mutation operators. Crossover and mutation occurrences have different probabilities, and each crossover or mutation operator has uniform probability once the application of a specific genetic operation has been chosen.

The two crossover operators combine two different topologies in two different manners, both trying to recombine the genotype while preserving functional blocks. The six mutation operators have a different effect on the topology of the net. Two modify the number of neurons in a layer and its activation function so they work only on the first part of the genotype. Two operators modify the structure by removing or adding a new layer, and the last two operators modify the connectivity of the net (generally without changing the layers in it).

A. Single Point Crossover

This method combines two networks by cutting their topologies in two pieces using a cutting surface that entirely separates the input and the output of the network.

In order to guarantee this operator is closed with respect to the valid genotype family, we have to restore all the connections between the pieces of the networks. Any connection coming out of the input half of the first network has to be joined to any connection going into the output part of the second network, and viceversa. In this manner the final number of connections between the newly generated individuals is greater than the original one, but the validity of the genotype is preserved.

Figure 5 describes the effect of the single point crossover operator on two different topologies. Two random points in the first part of the two genotypes are chosen. The input part of the first genotype is combined with the second part of the output genotype and viceversa. To obtain the final connection matrix of each new individual, the top left sub–matrix of one of the parents is used as top left sub–matrix of the child and the bottom right sub–matrix of the other parent is used as bottom right sub–matrix of the child. This operation preserves the internal connectivity of the two parts joined in the new individual, and these two parts are joined by filling the top right sub–matrix of the genotype.

Cell (i, j) in the top right sub–matrix of the genotype has a connection *iff* any of the cells in i^{th} row of the parent providing that input part have a connection and any of the cells in j^{th} column of the parent providing the output part have a connection.

B. Two Point Crossover

Figure 6 describes the effect of the two point crossover operator. This method combines two networks by extracting a connected subgraph from each of them, and exchanging the two subgraphs between the networks.

In order to guarantee this operator is closed with respect to the valid genotype family, we have to restore all the connections between the remaining network and the new block. Any connections coming out of or going into the new block have to be joined to any connections going into or coming out of the old block. The final number of connections between the newly generated individuals is again greater than the original one, but the validity of the genotype is preserved.

Figure 6 illustrates the application of the operator. Two random points for each first part of the two genotypes are chosen and the blocks between these two cutting points are exchanged. To obtain the final connection matrix of each new individual, the top left, top right, and bottom right sub–matrixes of one of the parents are used as top left, top right, and bottom right sub–matrixes of the child and the central sub–matrix of the other parent is used as central sub–matrix of the child. This operation preserves the internal connectivity of the two parts joined in the new individual.

Note that, to join the new block into the "hosting" network, the top middle and right middle sub–matrixes have to be filled in a specific way. A cell (i, j) in the top middle sub–matrix has a connection *iff* any of the cells in the i^{th} row of the original top middle sub–matrix of the parent network hosting the new

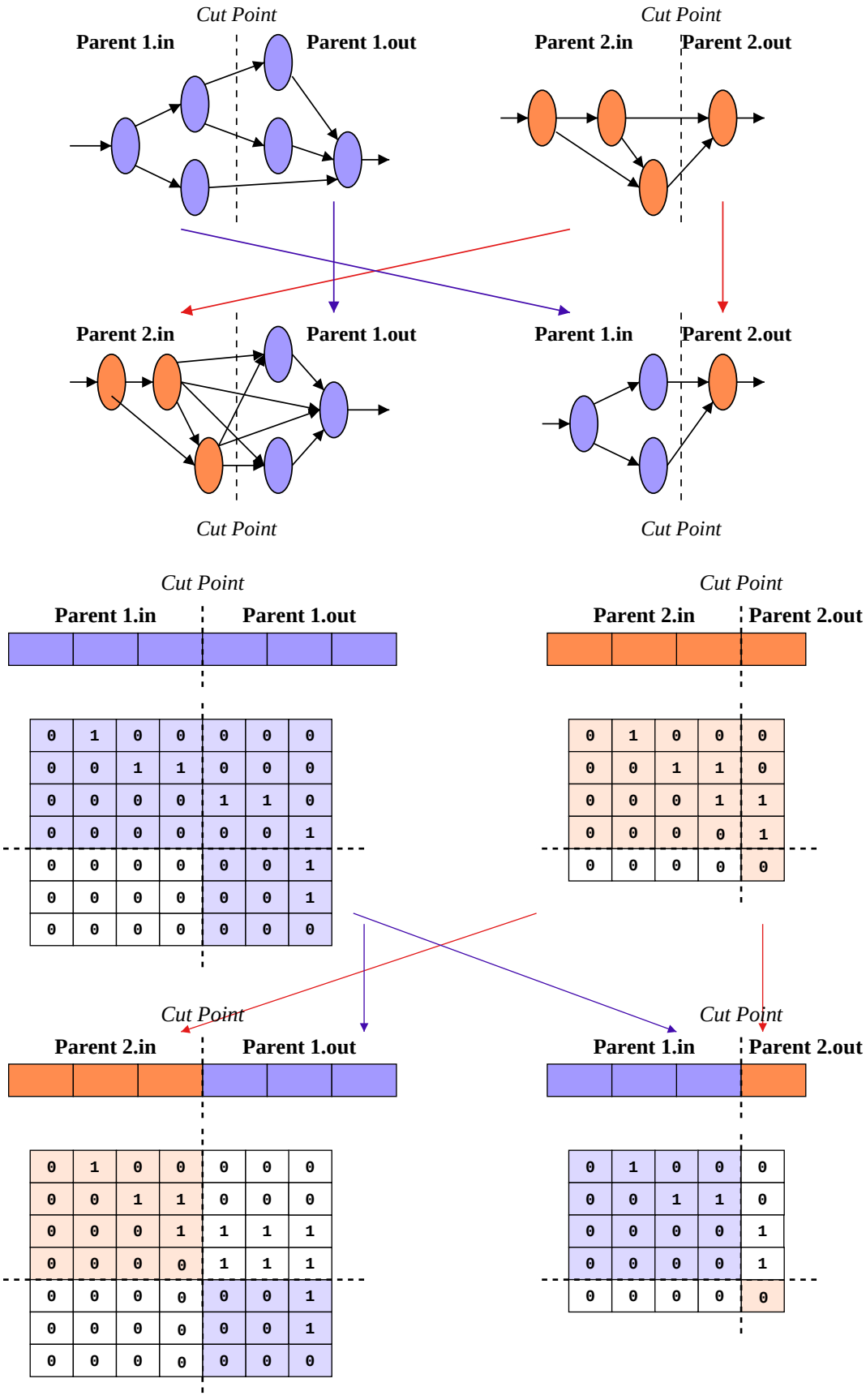


Fig. 5. The single point crossover genetic operator

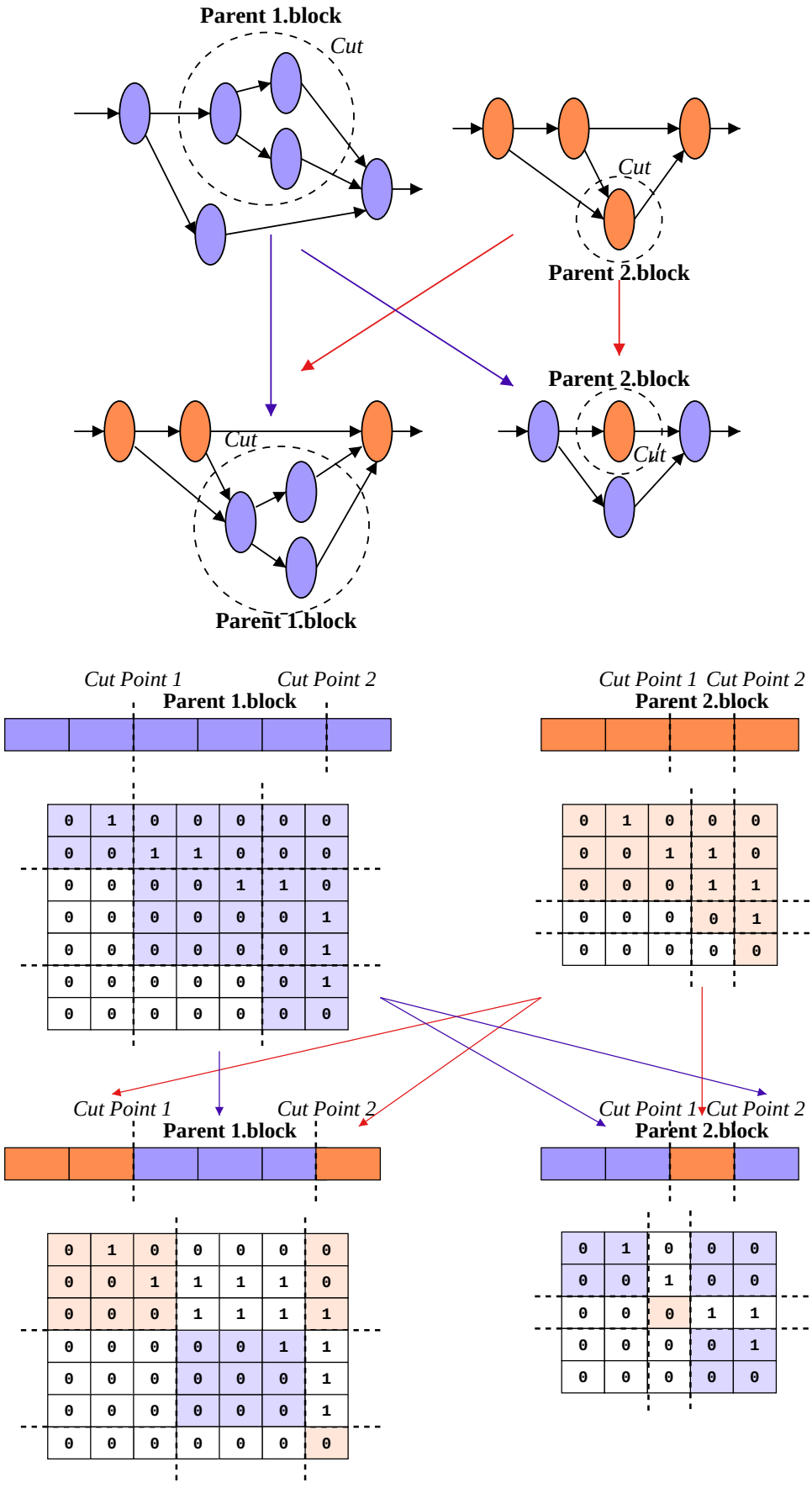


Fig. 6. The TwoPointCrossover genetic operator

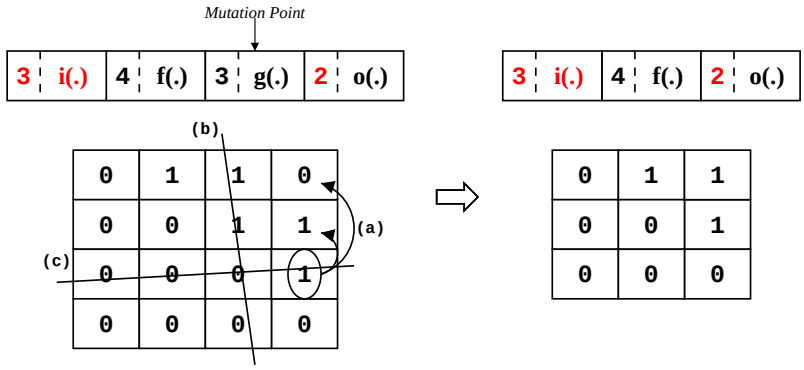


Fig. 7. The `MutatorDropNode` genetic operator

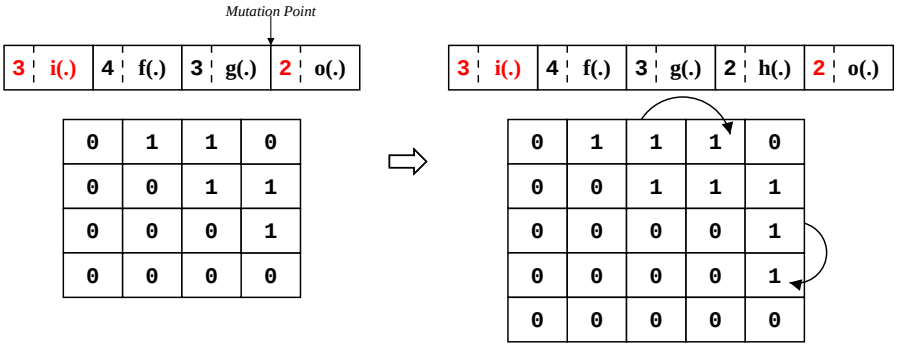


Fig. 8. The `MutatorAddNode` genetic operator

block have a connection and any of the cells in j^{th} column of the parent providing the block have a connection. A cell (i, j) in the right middle sub–matrix has a connection *iff* any of the cells in the j^{th} column of the original right middle sub–matrix of the parent network hosting the new block have a connection and any of the cells in i^{th} row of the parent providing the block have a connection.

C. Mutator: Drop Node

This mutation operator randomly selects a layer and removes it from the network structure. Figure 7 illustrates the application of the operator. Before removing the layer from network structure, each input connection of it is connected to all the destination of its output connections[5]. This operator is guaranteed to be closed with respect to the valid genotype family, thus a genotype modified by it does not require to be checked.

D. Mutator: Add Node

This mutation operator adds a layer to the network topology. Figure 8 illustrates the application of the operator. An existing layer is randomly selected and its connectivity is duplicated. After that a random activation function and a different number of neurons are initialized. Since a valid copy of an existing neuron connectivity sub–matrix is used, this operator is guaranteed to be closed with respect to the valid genotype family, so a genotype modified by it does not need to be checked.

[5]This is equivalent to setting the activation function of the layer to the identity, but reduces the number of weights for the network and also the number of free parameters.

E. Mutator: Number of Neurons

This mutation operator, changes the number of neurons in a specific layer of the network. Figure 9 illustrates the application of the operator. A random mutation point is chosen and the number of neurons in the specific layer is changed according to a uniform distribution. This operator is guaranteed to be closed with respect to the valid genotype family, thus a genotype modified by it does not require to be checked.

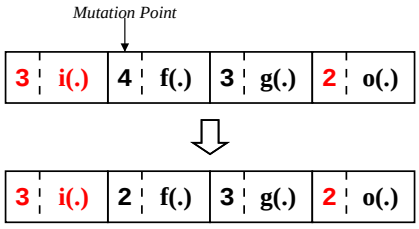


Fig. 9. The `MutatorNodeNumNeur` genetic operator

F. Mutator: Drop Connection

This mutation operator removes a connection from the connectivity matrix of the network. Figure 10 illustrates the application of the operator. Once the connection is removed, the operator checks for layers that are not reachable from the input or that do not participate in the output of the network. These layers have to be removed and this can lead to the complete destruction of the network. This operator is not guaranteed to be closed with respect to the valid genotype family, thus a genotype modified by it has to be checked. In the case a non–valid

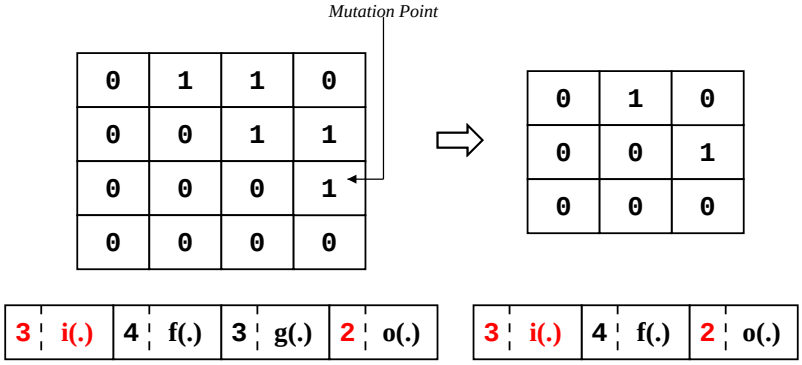


Fig. 10. The `MutatorDropConnection` genetic operator

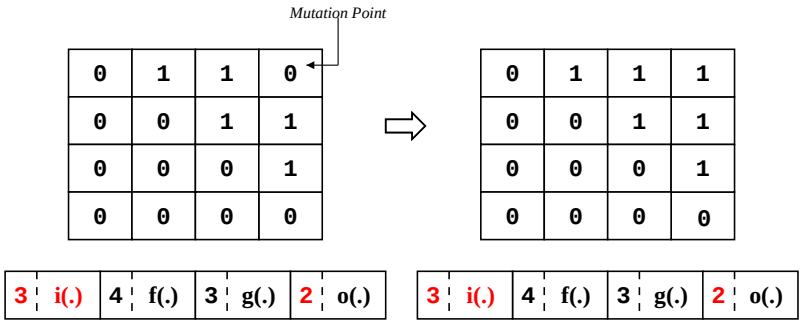


Fig. 11. The `MutatorAddConnection` genetic operator

genotype is generated, a new one is randomly initialized and substituted for the original one.

G. Mutator: Add Connection

This mutation operator adds a new connection in the connectivity matrix of the network. Figure 11 illustrates the application of the operator. If the network is completely connected, the genotype is left unchanged. This operator is guaranteed to be closed with respect to the valid genotype family, thus a genotype modified by it does not require to be checked.

H. Mutator: Activation Function

This mutation operator changes the activation function in a network layer. Figure 12 illustrates the application of the operator. A random mutation point is chosen and the activation function in that specific layer is changed according to a uniform distribution over the available activation functions. This operator is guaranteed to be closed with respect to the valid genotype family, thus a genotype modified by it does not require to be checked.

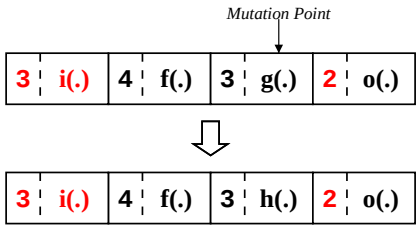


Fig. 12. MutatorNodeActFunc genetic operator

VI. Fitness Function

The average of the squared differences between the desired output t and the actual output y of a neural network (Mean Squared Error) is commonly used as a fitness measure in this kind of applications, and it is called *standard fitness*:

$$\frac{1}{R} \cdot \sum_{r}^{R} (y_r - t_r)^2$$

This is the measure which most implementations of genetic algorithms operating on neural networks try to minimize, but when calculating a network's overall performance, it is important to take into account also how the network is going to be used and what its generalization ability will be (i.e., how well the model will do when it is asked to make new predictions for data it has not already seen).

In our case, a network is tested using *k–fold cross-validation* [21]. The data set is divided into k subsets. Each time, one of the k subsets is used as the test set and the other $k-1$ subsets are put together to form a training set. Then the average of the standard fitness across all k trials is computed. The variance of the resulting estimate is reduced as k is increased. But, since the training algorithm has to be rerun from scratch k times, it takes k times as much computation to make a fitness evaluation.

VII. Algorithm Empirical Validation

We conducted a set of experiments on several datasets to evaluate the performance of our genetic search for an effective

rich neural network topology[6]. The main risk in this kind of applications is overfitting; this is particularly relevant in our case since we allow the network to get large and highly non–linear. To check if our algorithm suffers from this problem, in the first set of experiments we use a synthetic dataset with a sinusoidal wave and gaussian noise.

The second set of experiments is based on real data from the suite of benchmarking problems PROBEN1 [18] and explores two problems for which neural networks are commonly used: regression and classification. The main reason for these experiments is to prove that in real applications the performance of the networks obtained by the genetic algorithm without any a–priori knowledge or design effort is reasonably close to the performance of the networks obtained by "educated" manual design.

A. Synthetic data

In the first set of experiments, the algorithm tries to find the correct neural network to fit a sinusoidal dataset with gaussian noise:

$$y = sin(2\pi x) + \epsilon \quad \epsilon \sim N(0, \sigma^2).$$

Note that, in this synthetic example, the process generating the data belongs to the family of models that can be represented by our algorithm: a single neuron network with a sinusoidal activation function.

Figure 13 shows one of the models learned by the algorithm. It presents one of the issues that typically arise with the use of a genetic search: the *"bloat" phenomenon*. Especially with genetic programming, the size of the solution tends to grow rapidly as the population evolves; Angeline [1], for example, observes that many of the evolved solutions contained code that, when removed, did not alter the result produced. In our case would be possible to intervene on the models and remove the nodes that are not useful using an a-posteriori analysis of the weights in the network.

Since this is a unidimensional problem, we can plot the predictions of the networks during one of the evolutions (see Figure 14). From the plots, it is possible to understand how, during the evolution, various functions are chosen in order to fit the data, and how the genetic algorithm finally stabilizes on the sinusoid model.

[6]In all the experiments we use a population with 20 individuals, the evolution is stopped after 50 generations, $p_{cross} = 0.75, p_{mut} = 0.25$, and individuals are selected with probability proportional to their fitness.

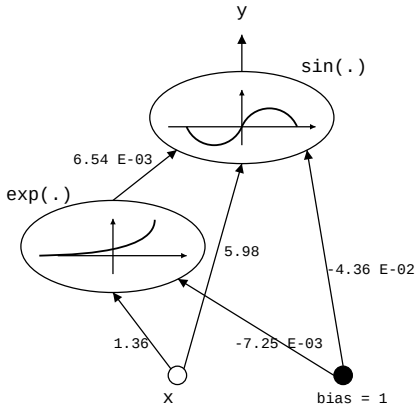


Fig. 13. A model learned by the algorithm for the sinusoidal dataset

B. PROBEN1 Benchmarks

The second set of experiments uses four real datasets from the PROBEN1 test suite. The benchmarking in this case is done strictly following the rules presented in [18] in order to obtain results comparable with the ones presented in the paper. Two datasets are chosen for regression tasks and two datsets for classification tasks. For each of them we use the three different train/validation/test splits proposed by the PROBEN1 suite.

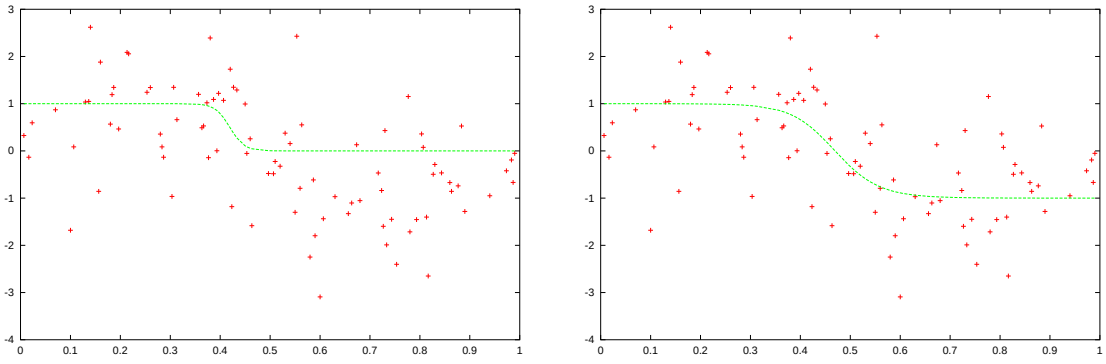
In the regression case, the error measure presented is the **squared error percentage**, a normalized version of the mean squared error that takes into account the number of output coefficients in the problem representation and the range of the output values used:

$$SEP = 100 \cdot \frac{y_{max} - y_{min}}{N \cdot P} \sum_{p=1}^{P} \sum_{i=1}^{N} (y_{pi} - t_{pi})^2$$

where y_{min} and y_{max} are the minimum and maximum values of the output coefficients in the problem representation (assuming these are the same for all output nodes), N is the number of output nodes of the network, and P is the number of patterns in the dataset considered. The two datasets used in this kind of task are:

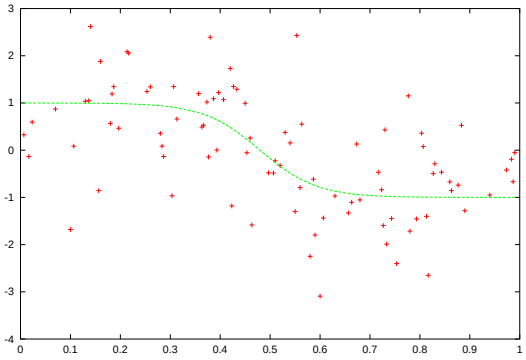
1) The `building` datset: for this dataset, the goal of the non-linear function approximator is to predict the energy consumption in a building. More specifically, the task is to predict the hourly consumption of electrical energy, hot water, and cold water, based on the date, time of the day, outside temperature, outside air humidity, solar radiation, and wind speed. It has 14 input & 3 output real variables, and 4208 examples. This problem in its original formulation was a prediction task; complete hourly data for four consecutive months was given for training, and output data for the following two months is to be predicted. The dataset `building1` reflects this formulation of the task – its examples are in chronological order. The other two versions, `building2` and `building3` are random permutations of the examples, simplifying the task to interpolation[7].
2) The `flare` dataset: in this dataset, the function approximator is trained to predict solar flares. The task is to forecast the number of solar flares of small, medium, and large size that will occur during the next 24 hours period in a fixed active region of the sun surface. Input values describe previous flare activity and the type and history of the active region. The problem has 24 input & 3 output real variables, and 1066 examples, but 81% of the examples are zero in all three output values resulting in

[7]These datasets were created for the PROBEN1 test suite based on problem A of "The Great Energy Predictor Shootout - The First Building Data Analysis And Prediction Competition" organized in 1993 for ASHRAE Meeting in Denver, Colorado.

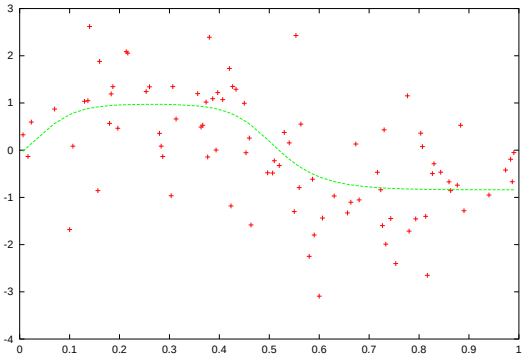


Best Individual in Pop_0

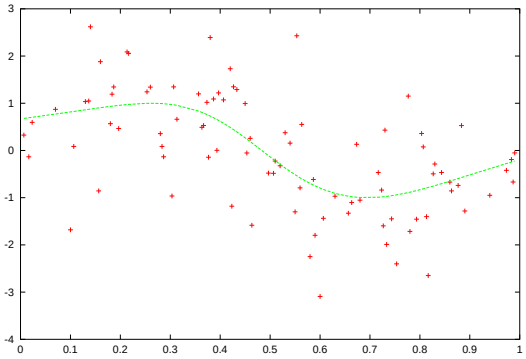
Best Individual in Pop_1



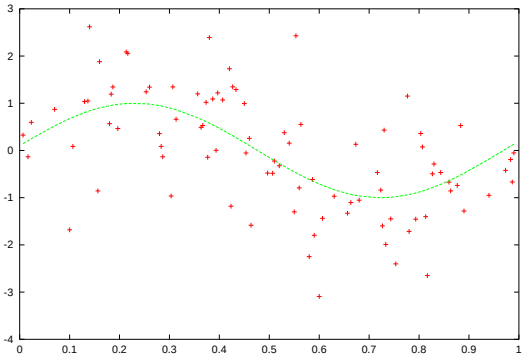
Best Individual in Pop_2



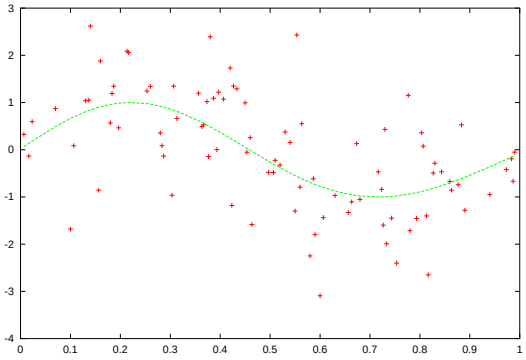
Best Individual in Pop_3



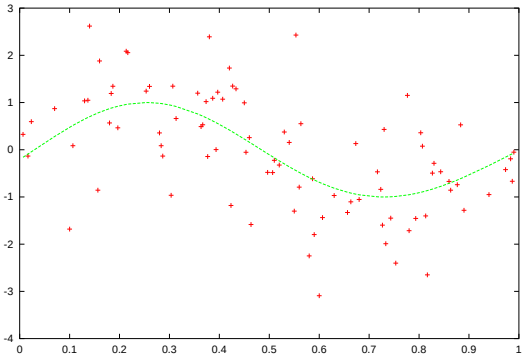
Best Individual in Pop_4



Best Individual in Pop_5

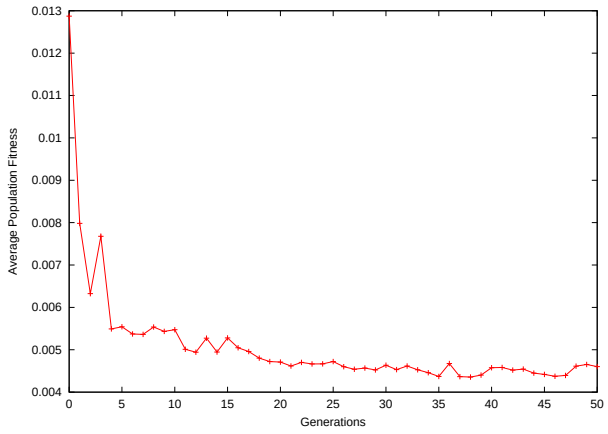


Best Individual in Pop_6

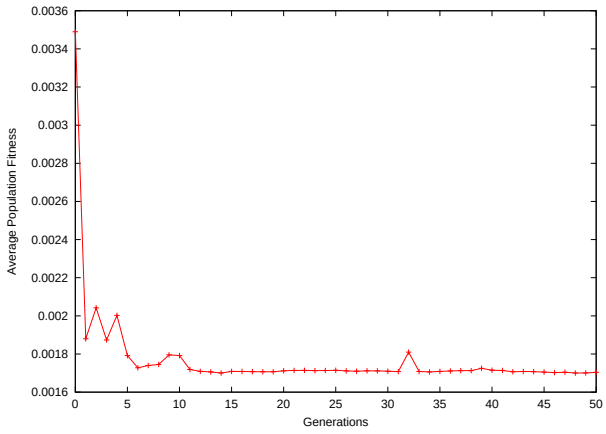


Best Individual in Pop_7

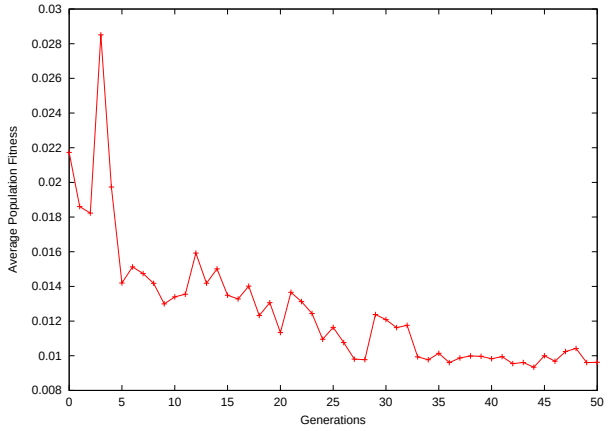
Fig. 14. Evolution for the sinusoidal model with Gaussian noise over time



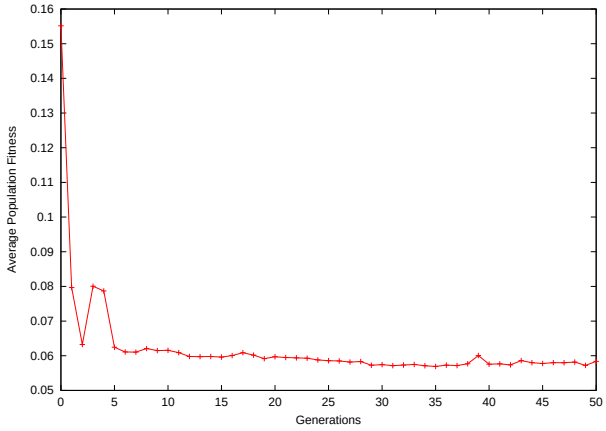
Population average fitness on the `building1` dataset



Population average fitness on the `flare1` dataset



Population average fitness on the `cancer2` dataset



Population average fitness on the `glass2` dataset

Fig. 15. Convergence in the average fitness of the population for the 4 datasets

a "bursty" phenomenon difficult to predict using smoothing methods[8].

For the classification tasks the error measure presented is the classification error. We use the $1-of-m$ encoding for m classes using output values 0 and 1; the classification method used is the *winner takes all* (i.e., the output with the highest activation determinates the class). The two datasets used in this kind of task are:

1) The `cancer` dataset: the task is to diagnose breast cancer. The neural network has to classify a tumor as either benign or malignant, based on cell descriptions gathered by a microscopic examination. Input attributes are, for instance, the clump thickness, the uniformity of cell size and cell shape, the amount of marginal adhesion and the frequency of bare nuclei. The dataset has 9 input real variables, 2 output classes, and 699 examples[9].
2) The `glass` dataset: the results of a chemical analysis of glass splinters (percent content of 8 different elements) plus the refractive index are used to classify a sample as either float processed or non float processed building windows, vehicle windows, containers, tableware, or heat lamps glass. This task is motivated by forensic needs in criminal investigations, and in this formulation it has 9 input real variables, 6 output classes, and 214 examples. From previous analysis two of the variables are known to have hardly any correlation with the result. As the number of examples is quite small, the problem is sensitive to algorithms that waste information[10].

The graphs in Figure 15 show the average population fitness during the evolution of the models for one splitting of each of the datasets. In all the four cases the algorithm has converged to a stable population of models. As can be seen from Tables I and II, the neural networks evolved by our genetic algorithm mostly perform similarly to the hand–designed ones presented in the PROBEN1 paper for the `flare` and `cancer` datasets. The results in these cases are fully comparable and the model learned by the genetic algorithm can obtain better performance depending on the split. On the `building` and `glass` dataset our networks clearly outperform the best models presented in the PROBEN1 suite independent of the specific spitting of the

[8]This dataset was created based on the "solar flare" problem dataset from the UCI repository of machine learning database.

[9]This dataset was created based on the "breast cancer Wisconsin" problem dataset obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg and is available at the UCI repository of machine learning databases [24].

[10]The dataset was created based on the "glass" problem dataset from the UCI repository of machine learning databases.

data.

Also in these cases, as we did for the synthetic dataset, it is possible to look at the model found by the algorithm and try to use them in an explanatory way, but they are usually complex and generally they do not convey any intuitive insights about the phenomenon modelled. Sometimes, like with the `building1` dataset, the resulting model can lead to an interpretation of the non–linear function implementing the regression. The model developed for the `building1` dataset presents a neural network with three layers implementing each a specific function:

- 9 neurons with *tanh* activation function which normalize the data between -1 and 1
- 9 neurons with *sin* activation function to capture the periodicity of the phenomenon
- 3 neurons with *logistic* activation function to transform the data in the range between 0 and 1

The solutions automatically generated for classification tasks usually have a logistic activation function on the output layer (and that is also the usual design rule in this kind of tasks), but the rest of the functions implemented are hard to interpret.

VIII. Conclusion and Future Work

In this paper we presented ELeaRNT, a new evolutionary strategy that evolves rich neural network topologies and finds domain–specific neural networks for classification and regression tasks. We presented the details of our genetic operators and the results showing how they are able to find rich topologies in a completely automated way. The generalization performance of the obtained networks is fully comparable to that of hand designed ones, outperforming those for some of the datasets used for benchmarking.

In this paper we optimize the neural network topology with respect to the generalization capability of the trained network and to do that we use the standard fitness to evaluate the individuals in the current population. This fitness function is designed for regression problems, and we suggest that another fitness function, better suited for classification, would achieve better results in classification tasks [22]. Moreover, if the network is in a situation where learning will be taking place frequently, even at real-time speeds, then a network's learning speed would be an important part of the topology design, and the algorithm should use an appropriate term in the fitness computation to optimize correctly the topologies for this kind of task.

A further improvement to the algorithm should include some internal heuristics to guide the genetic operators in a *credit/blame* fashion [23] during the search of the solution. Using information about parts of the actual solution it would be possible to direct the operators on those parts that do not partecipate to the final result or to those part that might need more neurons. This kind of approach would lead to a more informative evolution of the population with faster convergence and more accurate results.

Finally, since the average population fitness stabilizes around the best individuals at the end of the evolution, it would be possible to use the whole population of networks to implement a committee of network instead of using only the best individual [9].

	PROBEN1 Best		Evolved Model	
	Validation	Testing	Validation	Testing
building1	0.7583	0.6450	**0.4538**	**0.3983**
building2	0.2629	0.2509	**0.2049**	**0.1887**
building3	0.2460	0.2475	**0.1866**	**0.1856**
flare1	0.3349	**0.5283**	**0.3196**	0.5506
flare2	0.4587	0.3214	**0.4310**	**0.2861**
flare3	0.4541	**0.3568**	**0.4517**	0.3576

TABLE I

Squared Error Percentage for the regression tasks

	PROBEN1 Best		Evolved Model	
	Validation	Testing	Validation	Testing
cancer1	1.714	**1.149**	**1.142**	2.873
cancer2	**1.143**	5.747	1.714	**4.023**
cancer3	**2.857**	**2.299**	**2.857**	3.448
glass1	31.48	32.08	**25.93**	**28.30**
glass2	38.89	52.83	**33.33**	**39.62**
glass3	33.33	33.96	**31.48**	**30.19**

TABLE II

Classification error for the classification tasks

Acknowledgements

I would like to thank my advisor Manuela Veloso for her invaluable advices during the duration of the project and Elena Eneva for her help and suggestions on the previous versions of this paper. This research was sponsored by an Ambassadorial Scholarship from the Rotary International, a fellowship from the Center for Automated Discovery, and a DARPA Grant No. F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the funding institutions.

References

[1] P. J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 75–98. MIT Press, 1994.

[2] K. Balakrishnan and V. Honavar. Evolutionary Design of Neural Architectures: A Preliminary Taxonomy and Guide to Literature. Technical Report CS TR 95-01, Department of Computer Science, Iowa State University, Ames, Iowa, 1995.

[3] C. Campbell. Constructive learning techniques for designing neural network systems. In CT Leondes, editor, *Neural Network Systems Technologies and Applications*. Academic Press, 1997.

[4] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Denver 1989, 1990. Morgan Kaufmann, San Mateo.

[5] G.W. Flake. *Nonmonotonic activation functions in multilayer perceptrons*. PhD thesis, Department of Computer Science University of Maryland, December 1993.

[6] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 1987.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[8] P. J. B. Hancock. Genetic algorithms and permutation problems: a comparison of recombination operators for neural net structure specification. In D Whitley, editor, *Proceedings of COGANN workshop, IJCNN, Baltimore*. IEEE, 1992.

[9] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

[10] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems*, volume 5, pages 164–171. Morgan Kaufmann, CA, 1992.

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

[12] F. Kursawe. Evolution strategies for vector optimization. In *Procedings of the Tenth International Conference on Multiple Criteria Decision Making*, pages 187–193, Taipei, China, 1992.

[13] S. Lawrence, C. L. Giles, and A. C. Tsoi. What size neural network gives optimal generalization? Convergence properties of backpropagation. Technical Report UMIACS-TR-96-22 and CS-TR-3617, University of Maryland, April 1996.

[14] Y. LeCun, J. Denker, S. Solla, R. E. Howard, and L. D. Jackel. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, San Mateo, CA, 1990. Morgan Kauffman.

[15] Y. Liu and X. Yao. A population-based learning algorithm which learns both architectures and weights of neural networks. *Chinese Journal of Advanced Software Research*, 3(1):54–65, 1996.

[16] D. Lovell and A. Tsoi. The performance of the neocognitron with various s-cell and c-cell transfer functions. Technical report, Intelligent Machines Lab., Dept. of Elec. Eng., Univ. of Queensland, 1992.

[17] G. Mani. Learning by gradient descent in function space. Technical Report WI 52703, Computer Sciences Department, University of Winsconsin, Madison, 1990.

[18] L. Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, University of Karlsruhe, 1994.

[19] E. Ronald and M. Schoenauer. Genetic lander: An experiment in accurate neuro-genetic control. In *The Third Conference on Parallel Problem Solving from Nature*, pages 452–461. Springer-Verlag, 1994.

[20] J. Schaffer, D. Whitley, and L. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37, 1992.

[21] M. Stone. Cross-validation choice and assessment of statistical procedures. *Journal Royal of Statistical Society*, B(36):111–147, 1974.

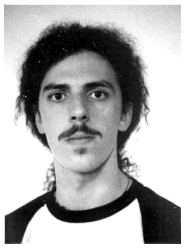
[22] B. A. Telfer and H. H. Szu. Energy functions for minimizing misclassification error with minimum-complexity networks. *Neural Networks*, 7(5):809–818, 1994.

[23] A. Teller and M. Veloso. Neural programming and an internal reinforcement policy. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 186–192, Stanford University, CA, USA, 28–31 1996. Stanford Bookstore.

[24] W. H. Wolberg and O. L. Mangarasian. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proceedings of the National Academy of Sciences of the USA*, 87:9193–9196, 1990.

[25] X. Yao. *A Review of Evolutionary Artificial Neural Networks*. Commonwealth Scientific and Industrial Research Organization, Victoria, Australia, 1992.

[26] X. Yao. Evolving artificial neural networks. *PIEEE: Proceedings of the IEEE*, 87, 1999.



Matteo Matteucci got a Laurea degree in Computer Engineering from Politecnico di Milano in 1999 and a Masters degree in Knowledge Discovery & Data Mining from Carnegie Mellon University in 2002. He is a PhD student in Computer Engineering and Automatics at Politecnico di Milano. His main research interests include: soft computing (fuzzy systems, neural networks, genetic algorithms), reinforcement learning, statistical approaches to learning and discovery, autonomous robots, behavior engineering, and computer vision.