Neural Comput & Applic (2018) 29:1481–1492
https://doi.org/10.1007/s00521-016-2510-6

ORIGINAL ARTICLE

A genetic approach to automatic neural network architecture optimization

K. G. Kapanova[1] · I. Dimov[1] · J. M. Sellier[1]

Received: 29 October 2015 / Accepted: 20 July 2016 / Published online: 28 July 2016

Abstract The use of artificial neural networks for various problems has provided many benefits in various fields of research and engineering. Yet, depending on the problem, different architectures need to be developed and most of the time the design decision relies on a trial and error basis as well as on the experience of the developer. Many approaches have been investigated concerning the topology modelling, training algorithms, data processing. This paper proposes a novel automatic method for the search of a neural network architecture given a specific task. When selecting the best topology, our method allows the exploration of a multidimensional space of possible structures, including the choice of the number of neurons, the number of hidden layers, the types of synaptic connections, and the use of transfer functions. Whereas the backpropagation algorithm is being conventionally used in the field of neural networks, one of the known disadvantages of the technique represents the possibility of the method to reach saddle points or local minima, hence overfitting the output data. In this work, we introduce a novel strategy which is capable to generate a network topology with overfitting being avoided in the majority of the cases at affordable computational cost. In order to validate our method, we provide several numerical experiments and discuss the outcomes.

Keywords Neural networks · Architecture evolution · Genetic algorithms · Neural network architecture · Function approximation

✉ K. G. Kapanova
kapanova@parallel.bas.bg; kkapanova@gmail.com

[1] IICT, Bulgarian Academy of Sciences, Acad. G. Bonchev str. 25A, 1113 Sofia, Bulgaria

1 Introduction

Since their inception in the 1940s, artificial neural networks (ANN) have proved to be a successful approach to many problem solving tasks. While conventional software techniques need to be programmed to accomplish certain goals, neural networks utilize learning techniques to adapt their connections when new information is provided. Because of that, various paradigms and network architectures have been proposed and studied in the last 80 years which provide optimal performance for particular problems [1–3]. Many aspects of neural network structures and how the different parts of the network influence its performance have been investigated [4]. Important considerations have been given to the possible number of layers, the amount of neurons in the network and per layer, the type of connections, the type of activation functions, weight initialization, training algorithms, error functions and pre- and post-processing of data. Those components are crucial for the computational performance, efficiency and accuracy of every network [5, 6]. The architecture of ANNs actually influences the performance of the network, and particular example of this can be found in the field of computer vision, where researchers have found that convolutional networks provide the best performance in this particular case [7].

Nowadays, developers usually rely on a trial and error basis to reach the correct network topology[1] that can efficiently and effectively learn and generalize input patterns to output patterns. This method is prone to errors, often times resulting in loss of productivity and leading to high computational complexity. One may even say that we face

[1] The reader should note that in this paper we interchangeably use the words topology and architecture having in mind the same meaning.

an exponential number of parameters to choose from depending on the knowledge we have about the problem and the definition of these parameters. Moreover, the capacity of the system to learn is recognized as a distinct property of the network's structure, along with the values of the weights. To deal with those inherent difficulties, some types of self-organizing topologies have been proposed and implemented [7–10]. One can broadly separate those methods in two categories. The first one is parametric learning, where one searches for weight values. The second one is structural learning, where one searches for the best topology of the neurons and connections. Most of the time, the topology is defined before the initialization of the network, with one or more hidden layers and with full sequential connection of every neuron. The simplest implementation of the structural type of evolving topologies is through the addition or removal of neurons, which limits the architecture, without exploring the whole space of possible topologies [11, 12]. Another possible approach is the adoption of a predefined list of modifications such as adding only fully connected hidden neurons [13, 14]. If, for example, the network topology is deemed to be insufficient, a new one is developed and the previous is discarded. Unfortunately, in those cases, the provided possibilities often fall in a structural local optima, with a problem for the further identification of available parameters [5, 15]. In the last years, focus has been on the use of evolutionary algorithms (EAs) in two specific areas: in the optimization problem and in the automatic design of neural networks [5, 16–22]. Since EAs are heuristic methods, developed to mimic similar biological phenomena through the evolution of populations of individuals examining specific behavior, this method is efficient in exploring the possible space of solutions for the optimization problem [20]. Focusing on the automatic search for the various parts of a topology, one can implement genetic algorithms, as shown in [8, 23]. The latter model operates in a sequential layer search process, where the information for every layer is found in a specific genetic algorithm. This leads to prohibitive computational costs and high processing time. Additionally, most evolutionary approaches to network architectures implement the backpropagation training algorithm as the main training strategy for the networks [24]. In many cases, the developed architecture is often affected by the well known overfitting problem.

Therefore, a very crucial question is whether the implementing an automatic evolutionary algorithm for the network's topology along with parametric weight change can provide definite advantage over standard fixed-topology strategy. When one utilizes a strategy for the automatic evolution of a network architecture, one of the most viable considerations is about the computational cost and the expected benefits provided from such a method. In this work, we introduce a novel approach to the evolution of the neural network architecture in order to provide an automatic and computationally feasible self-organization of an artificial neural network layout to solve a given problem. The proposed technique introduces a new hybrid genetic algorithm, with several distinct advantages. On the one hand, the underlying method provides several degrees of freedom for the selection of a network structure. On the other hand, the hybrid genetic algorithm implements an additional stochastic layer, which provides a more sophisticated and computationally permissable strategy for the evolutionary algorithm. Through this new hybrid evolutionary strategy, we obtain a method which is easily parallelized and is capable of escaping overfitting of the output data when the minimal amount of training data is available.

The paper is organized as follows. In the next section, we introduce the evolutionary neural network architecture method, as well as a description of the possible structures available for the network to choose from. Then we show results from basic application of a network which task is to fit a known function. We restrict ourselves to this trivial problem for the sake of simplicity. Finally, we conclude with remarks and future directions.

2 Methodology and development

The current research on evolutionary neural network architecture has purposefully introduced various limitations to the number of possible parameters that could be randomized. The necessity for such limitations occurs due to the increase of complexity of the search space of possible architectures with the additions of more free structural parameters [5, 6].

In this paper, we first introduce an automatic strategy for the search of neural network architectures. Our approach allows to recombine the following components during the evolutionary process:

- number of possible hidden layers,
- number of possible neurons in every hidden layer,
- the connections between neurons,
- the type of activation function.

The recombination of all those components creates a multidimensional search space of possible structures, while capable of achieving sufficient performance at affordable computational resources.

For the validation of the approach and the specific task solved by the neural network, we have restricted the input layer to consist of one neuron with scalar activation function, and the output layer to represent only one neuron with scalar activation as well.

The following subsections provide more details about those options.

2.1 Layers, neurons and connections

The number of layers, the amount of neurons per hidden layer and the available connections are the first three parameters that could be randomly selected by the algorithm. There is no specific implementation in the code for the addition or removal of neurons during the process. Many evolutionary models described in the literature generally avoid arbitrary connectivity due to increase of computation cost, difficulties in simulation and analysis of the network. Our method implements dynamic connections, and we are not limiting the type of connections between neurons. The evolutionary algorithm can choose to connect the neurons sequentially or non-sequentially-connecting neurons irrespective of their position between layers.

We have implemented certain constraints to the minimum and maximum number of hidden layers, neurons and connection per neurons the algorithm can utilize. These restrictions are added to simplify the network in this particular work, but are in no way a limiting factor.

2.2 Activation functions

The fourth criterion provides several distinct activation functions which can be selected in the automatic evolutionary method:

- Identity Function—where $f(x) = x$. It is the simplest activation function where the activation is passed as the output of the neuron,
- Exponential Activation, where $f(x) = exp(x)$,
- Hyperbolic Tangent (tanh)—a trigonometric function with output in the range $[-1, +1]$. One of the most used activation functions,
- Function based on orthogonal polynomials—Laguerre, Legendre or Fourier.

There are at least two possibilities during the evolutionary search for the network topology: One may request all neurons in the hidden layer to use the same activation function, or to allow the algorithm to choose the best one for every neuron in the hidden layers.

In this particular work, for each hidden layer, we provide complete freedom to the network to choose the type of activation function.

2.2.1 The training process

A variety of evolutionary architecture algorithms have incorporated the backpropagation algorithm as a training strategy. It is a first-order method, which goal is to minimize the error function by updating the weights through a steepest descent method. A known drawback of the training technique is the presence of local minima [1]. The problem is even more pronounced in an evolutionary network topology task, where the high dimensionality of possible solutions increases the possibility of failure to escape from local minima, and therefore the presented output can be overfitted [2].

Therefore, for our method we use simulated annealing [25]. By analogy with physical systems, we initially increase an effective temperature to a maximum value and then we start to decrease it until the particles reach an equilibrium (which represents a solution to the task). One describes the probability of a point to move by

$$Pr[accept] = e^{\frac{-\Delta E}{T}}, \tag{1}$$

where ΔE is the difference between the actual energy and the energy before the move, and T is the effective temperature of the system. A move is accepted if the generated random number $[0, 1] \ni R < Pr[accept]$.

2.3 Evolutionary strategy

Genetic algorithms has been some of the most commonly utilized approaches in the studies of evolution of neural network architectures [5, 8, 9, 13, 23]. In this subsection, we introduce our hybrid genetic algorithm, the implementation of an entirely stochastic layer, as well as a combination of mutation and crossover approach, through which we evolve the possible architectural solutions where the offspring with the best fitness is selected.

2.3.1 Fitness and selection strategy

The evolutionary method assigns a fitness score to every individual belonging to the population, which characterizes the quality of the specific network topology. In this particular case, we use the $L_2 - norm$ fitness function. The algorithm selects the individual with the best fitness function for further evolution. The process is similar to the Tournament selection method [26].

2.3.2 Training process

Every individual from the population, at each time step, is trained by the simulated annealing method described above. The combination of the architecture selection and the training process can be seen as two nested optimization problem. It is possible for both strategies to influence each other, but we are incapable of defining the complexity of such interaction.

2.3.3 Initialization of the technique

The technique can select topologies by choosing from the four specified parameters described in the previous subsection: number of possible hidden layers, number of possible neurons in every hidden layer, number of connections per neurons and type of activation function.

The algorithm involves two synchronous steps. We initialize the genetic algorithm to create populations of individual architectures $C_1, C_2, C_3, \ldots C_n$ from the space of possible configurations. Simultaneously, the stochastic layer of the algorithm creates another batch of populations of individual architectures $S_1, S_2, S_3, \ldots S_n$. The next step evaluates the fitness of each individual from both batches and selects one individual from both the genetic and stochastic layer with the best fitness. The algorithm proceeds selecting the individual with the best fitness. The next iteration stochastically evolves the individual, using the controlled settings described above, creating a new set of populations of individual network topologies for both parts (genetic and stochastic) of the algorithm. The entire run of the algorithm utilizes both crossover, as well as mutation strategies at different steps.

The stochastic evolution of the best individual provides us with several conceptual advantages. Firstly, it allows the algorithm to escape from local optima, especially if the overall solution has various populations. This is accomplished through adaptation according to minimum design standards in the changing environment. Secondly, the stochastic evolution of a new individual allows the program to create a new architecture to solve a problem in an affordable computational time even in the presence of multidimensional search space.

2.3.4 Termination

The algorithm is terminated after one of two conditions is met. Either the maximum assigned number of generation is reached or the fitness error is decreased by more than a threshold value after r consecutive generations.

Fig. 1 The *plots* represent the output of the network (*left column*) and the respective architectures used in each case (*right column*). The first case is $3 \times 3 \times 2$ with the (*red*) *line* representing the function we try to fit, the (*blue*) *squares* describing the neural network output and the (*red*) *stars* representing the three data points provided for the training. The right side plots characterize the respective architecture for every case. The (*yellow*) node depicts the input node in the input layer, the (*red*) node shows the output node in the output layer, and the (*blue*) nodes represents the neurons in the hidden layers, respectively. The edges from the nodes account for the synaptic connections between nodes. The numbers next to every neuron display the type of activation function used by every neuron. The legend on the right lower side of each plot describes which function corresponds to which number. The *middle left* and *right* plots represent the output and network architecture in the case for $3 \times 4 \times 2$. The *lower left* and *right* plots represent the network's output and architecture in the case for $3 \times 6 \times 2$ (colour figure online)

3 Numerical experiments

To demonstrate the abilities of the proposed method, we perform a simple numerical experiment of fitting a known function $f(x) = x^2$. Although one might regard the function to be an oversimplified model, the purpose of this work is to introduce a novel technique for automated evolutionary architecture of neural networks. The reader should nonetheless note that the presented results in the section are averaged over 4×5 number of runs of the algorithm, involving the computation of 32 points. Several examples are provided, presenting the method's outcomes with restrictions in the number of hidden layers and neurons in them. We present architectures from 4 scenarios—implementing architectures with 3, 4, 5 and 6 hidden layers and different number of neurons allowed for each hidden layer 2, 3, 4, 5, 6. In this numerical analysis, we have limited the amount of input connections per neuron to be maximum 2. The reader should note that those restrictions are applied only for these particular experiments and are not intrinsic to the efficient work of the proposed evolutionary method.

The training data used in this numerical experiment are limited to three points—(0.1, 0.01), (0.5, 0.25) and (0.9, 0.81). To provide a good function approximation NNs

Table 1 Scenario 1: network architecture is limited to only one hidden layer process

Case	Num. layers	# Neurons in hidden layer	# Neurons in the network	Error
$3 \times 2 \times 2$	3	1	3	0.017309
$3 \times 3 \times 2$	3	2	4	0.018815
$3 \times 4 \times 2$	3	1	3	0.017909
$3 \times 5 \times 2$	3	1	3	0.0185091
$3 \times 6 \times 2$	3	3	5	0.0477594

There are 5 cases in this scenario—with the maximum of 2, 3, 4, 5 and 6 neurons per hidden layer. The fourth column represents the total amount of neurons in the network (including the neurons in the input and output layer), while the last column represents the final error from the evolutionary process

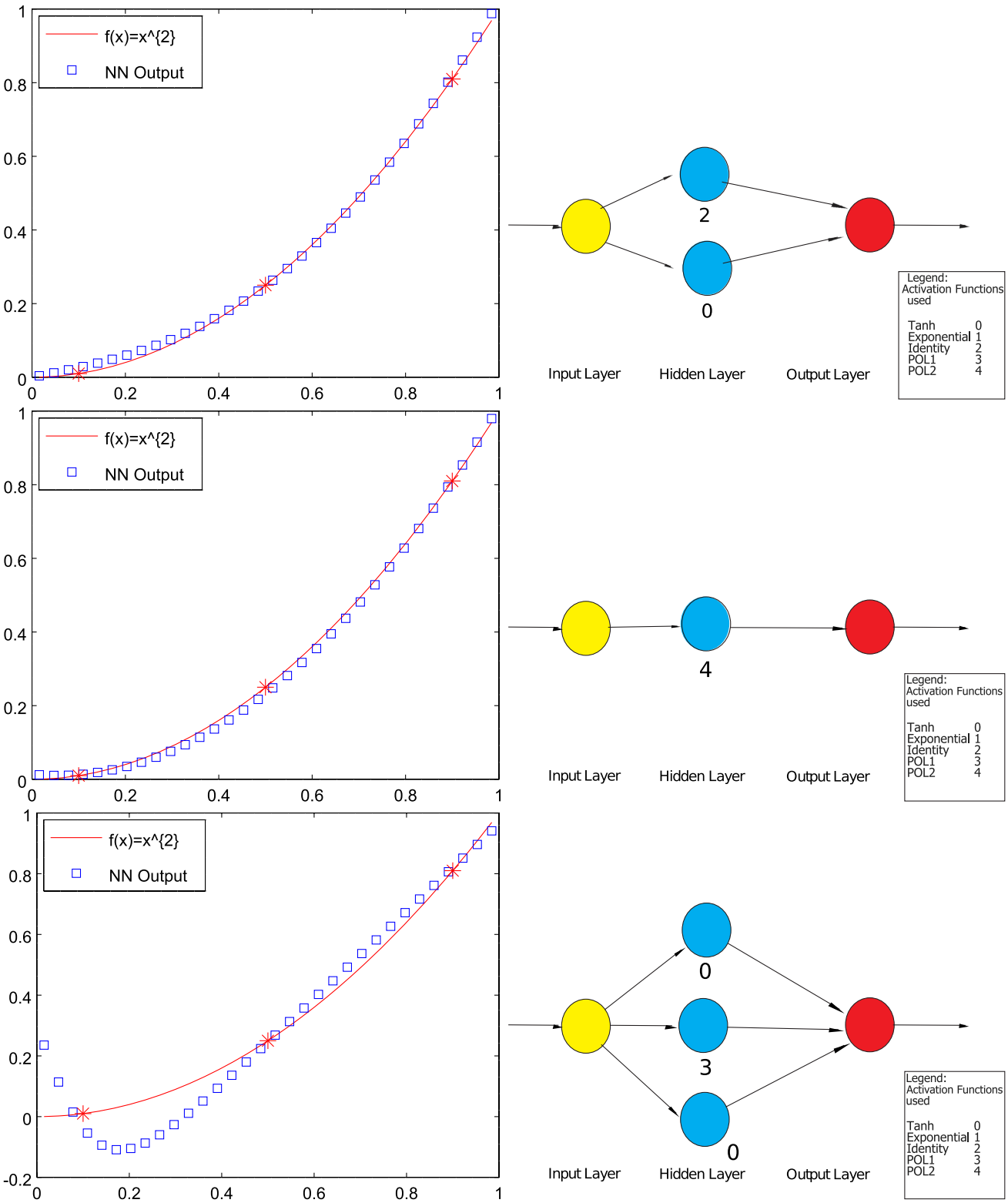

f(x)=x^{2}
NN Output
2
0
Input Layer
Hidden Layer
Output Layer
Legend:
Activation Functions used
Tanh 0
Exponential 1
Identity 2
POL1 3
POL2 4
f(x)=x^{2}
NN Output
4
Input Layer
Hidden Layer
Output Layer
Legend:
Activation Functions used
Tanh 0
Exponential 1
Identity 2
POL1 3
POL2 4
f(x)=x^{2}
NN Output
0
3
0
Input Layer
Hidden Layer
Output Layer
Legend:
Activation Functions used
Tanh 0
Exponential 1
Identity 2
POL1 3
POL2 4

Table 2 Scenario 2: network architecture is limited to only two hidden layers process

Case	Num. layers	# Neurons in hidden layer	# Neurons in the network	Error
$4 \times 2 \times 2$	4	Hidden #1—1 neuron	4	0.006946
		Hidden #2—1 neuron		
$4 \times 3 \times 2$	3	Hidden #1—2 neurons	4	0.017937
$4 \times 4 \times 2$	4	Hidden #1—3 neurons	8	0.019676
		Hidden #2—3 neurons		
$4 \times 5 \times 2$	4	Hidden #1—2 neurons	6	0.017426
		Hidden #2—2 neurons		
$4 \times 6 \times 2$	4	1 hidden—1 neuron	4	0.034233

Five cases in this scenario were developed—with implementation of maximum 2, 3, 4, 5 and 6 neurons per hidden layer. The fourth column represents the total amount of neurons in the network (including the neurons in the input and output layer), while the last column represents the final error from the evolutionary process

require large enough training samples [2]. Often when limited training data are supplied, the network is unable to learn complex relationship and overfitting occurs [27], especially with the implementation of the backpropagation training algorithm. Therefore, one of the aims is to show that when limited training data are supplemented, our evolutionary architecture could overcome those limitations.

3.1 Three layers from 2 to 6 neurons

The first experiment limits the possible layers to 3—one input layer, one hidden layer and one output layer. For all the numerical experiments, we have implemented limitations of the maximum amount of neurons in every hidden layer to 3, 4, 5 or 6. The results in Table 1 indicate that the cases[2] ($3 \times 2 \times 2$ and $3 \times 4 \times 2$) with the smallest error are the ones when only one neuron in the hidden layer is generated. One should note that in the last case (where 6 neurons can be used in the hidden layer) the model rarely utilizes all the available neurons. The inherent limitations of the number of hidden layers enforce our model to generate feedforward networks. In the cases of $3 \times 3 \times 2$ and $3 \times 4 \times 2$ the method provides an output with nearly perfect function approximation (see Fig. 1, upper left and middle left plots, respectively).

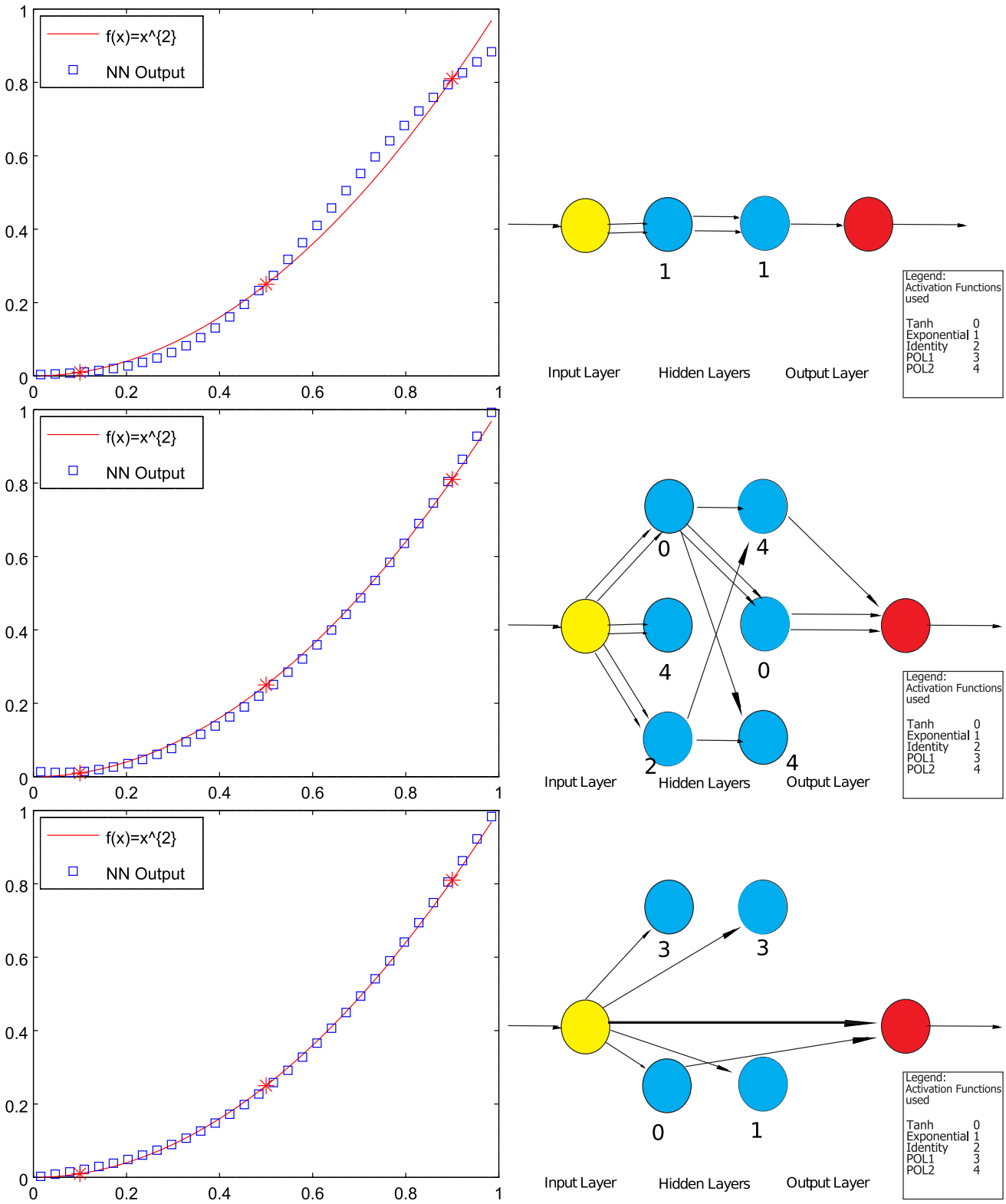
All the cases in this scenario provide for a model of a feedforward neural network, with the $3 \times 3 \times 2$ and $3 \times 4 \times 2$ networks and their outputs in the range of desired accuracy (see Fig. 1, left column, top and middle rows, respectively). In the last case, $3 \times 6 \times 2$, one might draw the conclusion that the type of activation function selected may impact the desired network performance. On the other hand, the number of neurons selected in the hidden layer appears to expose the network to an overfitting of the output. Such a conclusion is supported by [27].

Fig. 2 The plots represent the output of the network (*left column*) and the respective architectures used in each case (*right side plots*). The first case is $4 \times 2 \times 2$ with the (*red*) *line* representing the function we try to fit, the (*blue*) *squares* describing the neural network output and the (*red*) stars representing the three data points provided for the training. The right side plots characterize the respective architecture for every case. The (*yellow*) node depicts the input node in the input layer, the (*red*) node shows the output node in the output layer, and the (*blue*) nodes represents the neurons in the hidden layers, respectively. The edges from the nodes account for the synaptic connections between nodes. The *numbers* next to every neuron display the type of activation function used by every neuron. The legend on the *right lower* side of each plot describes which function corresponds to which number. The *middle left* and *right plots* represent the output and network architecture in the case for $4 \times 4 \times 2$. The *lower left* and *right* plots represent the network's output and architecture in the case for $4 \times 5 \times 2$ (colour figure online)

3.2 Four layers from 2 to 6 neurons

In the second scenario, the space of possible network topologies is increased by adjusting the available hidden layers to 2. There are 5 possible occurrences where the maximum amount of neurons per hidden layer are 2, 3, 4, 5 and 6. One may observe that the expansion of hidden layers and available neurons in them, facilitates the automatic evolutionary method to select network architectures with the same number of neurons in every hidden layer (see Table 2). In $4 \times 2 \times 2$, the best possible architecture the method finds, consists of 2 hidden layers, each with 1 neuron. The architecture represents a type of feedforward network, which differentiates from the classical one in the amounts of synaptic connections between neurons—where each neuron has two connections to the next one. Moreover, the neurons employ the same exponential activation function. The selected topology produces the smallest error (0.006946, in the $L_2 - norm$), even when the network's output is overfitted (see Fig. 2, first row, left and

[2] The reader should note that when the notation $n_l \times n_{max} \times c_{max}$ is used it signifies information about the number of layers in the architecture (in this case n_l), the maximum number of neurons in every hidden layer (in this case n_{max}) and finally the maximum amount of connections from neuron to neuron (in this case c_{max}).


f(x)=x^{2}
NN Output
1
1
Input Layer
Hidden Layers
Output Layer
Legend:
Activation Functions used
Tanh 0
Exponential 1
Identity 2
POL1 3
POL2 4
f(x)=x^{2}
NN Output
0
4
4
0
2
4
Input Layer
Hidden Layers
Output Layer
Legend:
Activation Functions used
Tanh 0
Exponential 1
Identity 2
POL1 3
POL2 4
f(x)=x^{2}
NN Output
3
3
0
1
Input Layer
Hidden Layers
Output Layer
Legend:
Activation Functions used
Tanh 0
Exponential 1
Identity 2
POL1 3
POL2 4

right column) from which one may infer that the decrease of the cost function might lead to irregular output pattern.

When we increase the number of hidden layers to 2 and the maximum number of neurons per hidden layer to 4, our evolutionary architecture provides a network topology where we have the same amount of neurons (3) for every hidden layer. The method selected the logistic activation function used in 2 neurons, the identity and second-order polynomial as activation functions for the neurons. One may observe that in this case the automatic evolution method has removed certain neurons from connecting any further (see Fig. 2, right column, second right row). The final error is actually bigger, compared to the other cases as shown in Table 2, but the output of the network (see Fig. 2, left column, middle row) is in the desired range accuracy.

In Fig. 2 lower right plot (case $4 \times 5 \times 2$), we show the capability of our evolutionary method to utilize the neuron connection freedom. The topology is constituted of 4 layers—1 input, two hidden and 1 output layer. The input layer connects to every other available neuron in this network, including the neuron in the output layer. The only other connection to the output neuron is established from the second neuron in the first hidden layer. Our method essentially removes the implementation of the second hidden layer by the elimination of output connections from those neurons. According to Table 2, the error reaches the

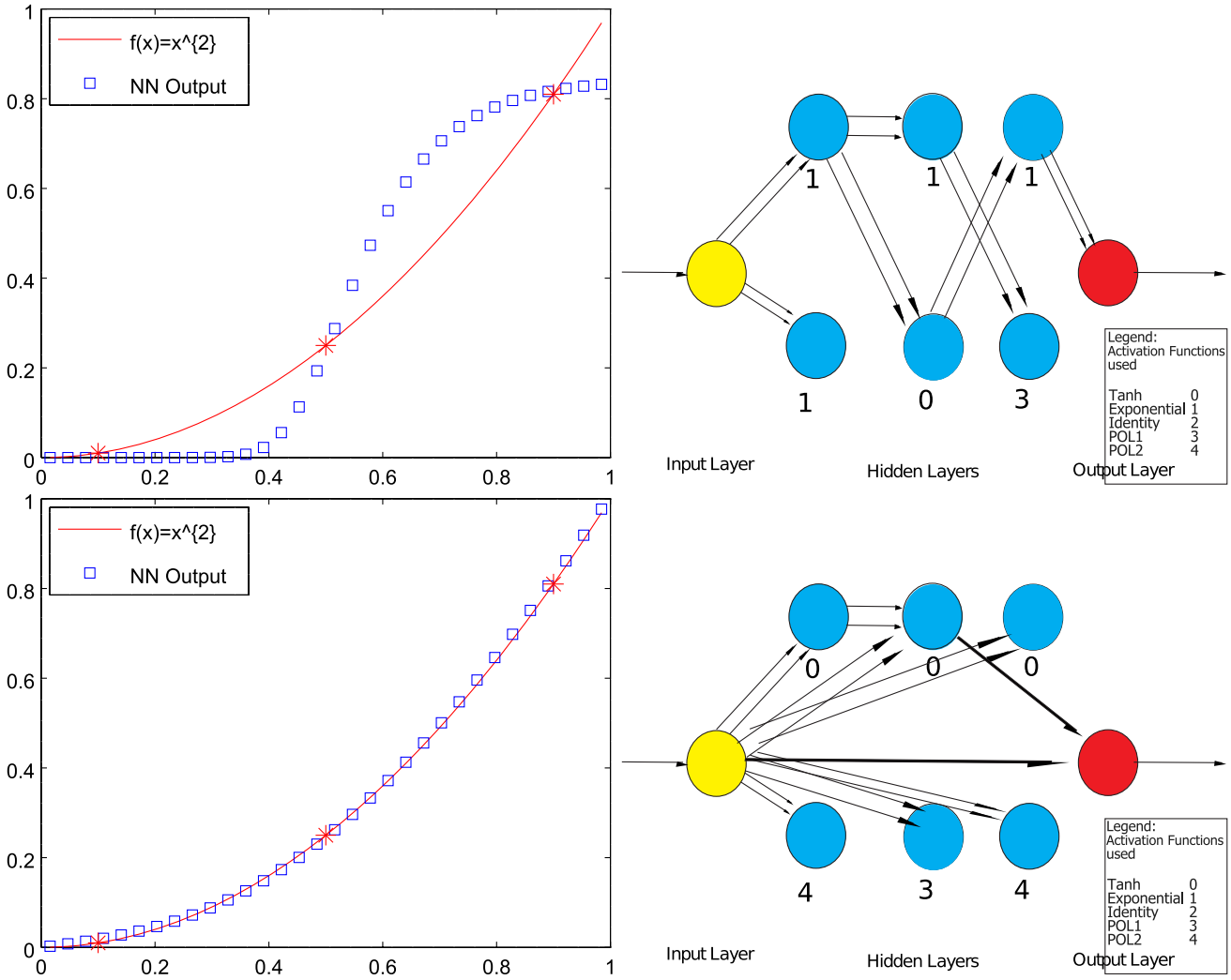


Fig. 3 The plots represent the output of the network (*left column*) and the respective architectures used in each case (*right column*). The first case is $5 \times 3 \times 2$ with the (*red*) *line* representing the function we try to fit, the (*blue*) *squares* describing the neural network output and the (*red*) *stars* representing the three data points provided for the training. The right side plots characterize the respective architecture for every case. The (*yellow*) node depicts the input node in the input layer, the (*red*) node shows the output node in the output layer, and the (*blue*) nodes represents the neurons in the hidden layers, respectively. The edges from the nodes account for the synaptic connections between nodes. The numbers next to every neuron display the type of activation function used by every neuron. The legend on the right lower side of each plot describes which function corresponds to which number. The *lower left* and *right* plots represent the network's output and architecture in the case for $5 \times 5 \times 2$ (colour figure online)

accepted minimum and the output of the network represents the intended function approximation. One may attempt to explain the small error and the desired output by means of the results presented in [28] (important observation in this case is the removal of neurons through their connection). Therefore, it seems likely that during the evolutionary process our method builds a representation of a model, where the option of discarding input connections is viable and seemingly beneficial to the particular method.

3.3 Five layers from 2 to 6 neurons

To evaluate the performance of our method when the multidimensional space of possible structures is expanded, we examined a situations with 3 hidden layers, and the maximum of 2, 3, 4, 5 and 6 neurons can be included in every hidden layer. One might expect higher computational costs, as well as decrease in performance as well as an output diverging from the desired range of solution. The latter is a probable outcome from the inability of the network to learn complex patters when we provide limited training data and purposefully increase the number of neurons in the network [27].

In the first architecture from this scenario—$5 \times 3 \times 2$, the 3 hidden layers utilize 2 neurons each. The method has further adjusted the network's architecture by reducing the number of connected neurons (see Fig. 3, second column first row). The removal of neurons as a strategy has been used by the evolutionary method in previous cases, although in this particular implementation one observes rather high final error (see Table 3), and a sharp overfitting of data. Reasonable assumption in this setting may be that the resemblance of a regular feedforward topology with limited data for learning could limit the capability of the method to provide a sufficient function generalization.

The same architecture of 3 hidden layers, each consisting of 2 neurons is chosen in the $5 \times 5 \times 2$ experiment as well. Similarly to the $4 \times 5 \times 2$ network topology, our method builds connections from the neuron in the input layer to every other neuron in the structure, including the one in the output layer (see Fig. 3, second column, second row). Once again, the obtained topology has removed almost all connections from the rest of the neurons, leaving neuron 1 from hidden layer 2 to connect to the output layer, along with the connection between the input and the output neuron. The provided output achieves near optimal approximation (see Fig. 3, first column, second row).

3.4 Six layers from 2 to 6 neurons

The concluding experiment introduces in the space of possible architectures the ability to choose up to 4 hidden layers with variations of 2, 3, 4, 5 and 6 per hidden layer.

According to Fig. 4 upper right plot ($6 \times 3 \times 2$ case), our evolutionary algorithm provided an architecture with 4 hidden layers, each with 1 neuron, sequentially connected by 2 synaptic connections, with 3 different activation functions assigned to every neuron. Analogous to the $4 \times 2 \times 2$ scenario, the feedforward topology produces bigger error (see Table 4) and overfitting of the output data. Nevertheless, when one considers, in this particular case, the multidimensional space of possible architectures, with the various levels of freedom for the number of layers, neurons and the types of activation functions, the method has still provided a sufficiently good data fitting at affordable computational cost.

The composition of the topology in the last case—$6 \times 6 \times 2$ employs 4 hidden layers, with 2 neurons each. The variation in this architecture is that the network is not sequentially connected, as compared to the $6 \times 3 \times 2$ case.

Table 3 Scenario 3: network architecture is limited to only three hidden layers process

Case	Num. layers	# Neurons in hidden layer	# Neurons in the network	Error
$5 \times 2 \times 2$	3	Hidden #1—1 neuron	3	0.019079
$5 \times 3 \times 2$	5	Hidden #1—2 neurons	8	0.016791
		Hidden #2—2 neurons		
		Hidden #3—2 neurons		
$5 \times 4 \times 2$	5	Hidden #1—1 neuron	5	0.006132
		Hidden #2—1 neuron		
		Hidden #3—1 neuron		
$5 \times 5 \times 2$	5	Hidden #1—2 neurons	8	0.015428
		Hidden #2—2 neurons		
		Hidden #3—2 neurons		
$5 \times 6 \times 2$	4	1 hidden—1 neuron	10	0.008473

Five cases in this scenario were developed—with implementation of maximum 2, 3, 4, 5 and 6 neurons per hidden layer. The fourth column represents the total amount of neurons in the network (including the neurons in the input and output layer), while the last column represents the final error from the evolutionary process

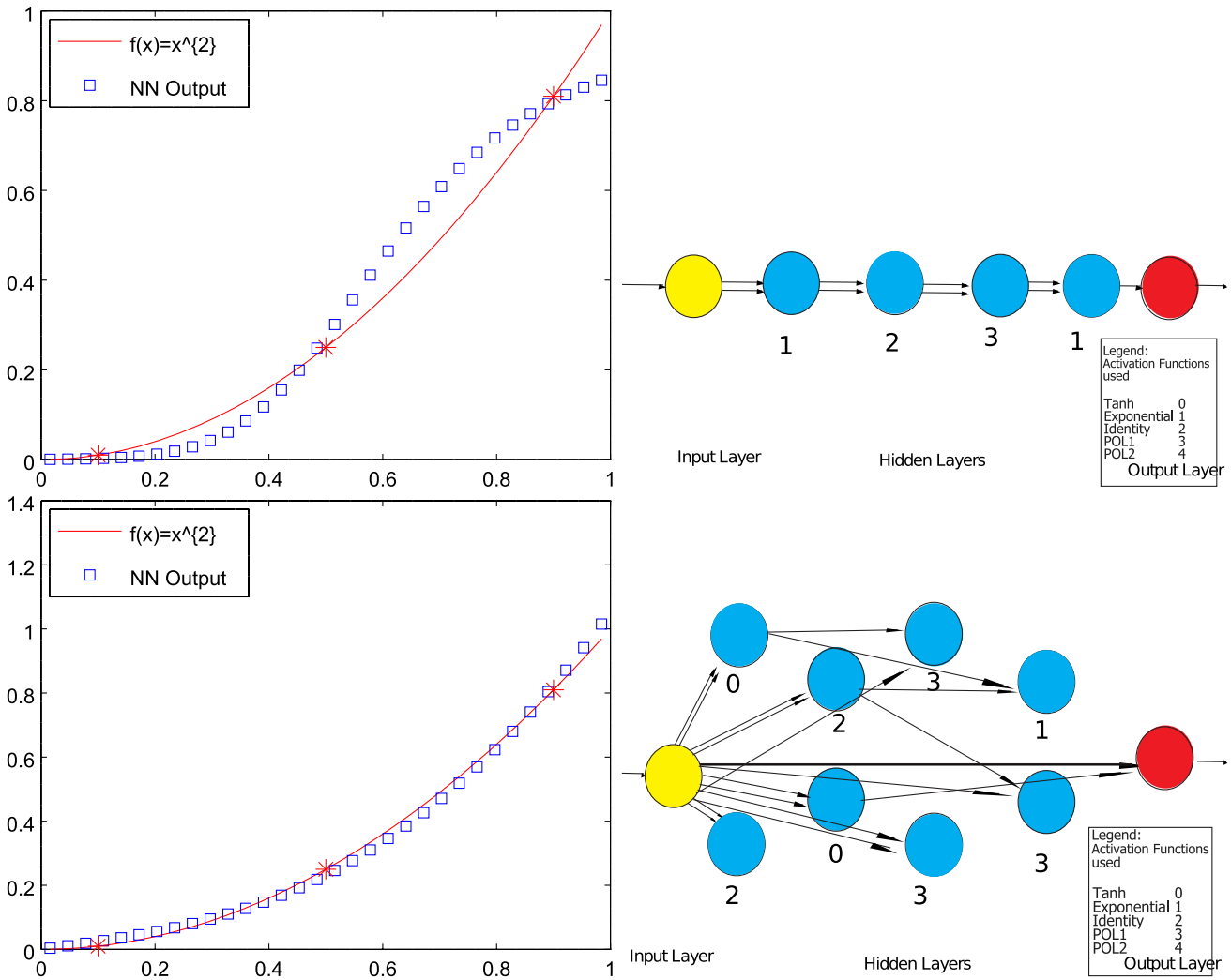


Fig. 4 The plots represent the output of the network (*left column*) and the respective architectures used in each case (*right* side plots). The first case is $6 \times 3 \times 2$ with the (*red*) *line* representing the function we try to fit, the (*blue*) *squares* describing the neural network output and the (*red*) *stars* representing the three data points provided for the training. The right side plots characterize the respective architecture for every case. The (*yellow*) node depicts the input node in the input layer, the (*red*) node shows the output node in the output layer, and the (*blue*) nodes represents the neurons in the hidden layers, respectively. The edges from the nodes account for the synaptic connections between nodes. The numbers next to every neuron display the type of activation function used by every neuron. The legend on the right lower side of each plot describes which function corresponds to which number. The *lower left* and *right* plots represent the network's output and architecture in the case for $6 \times 6 \times 2$ (colour figure online)

Moreover, our method exploits once again the randomized synaptic connections. The input neuron connects to every other available neuron with one or two input connections. Through this strategy, the algorithm removes nearly 90% of the neurons in the network. Apart from the input neuron, only the second neuron in hidden layer 2 provides a direct connection to the neuron in the output layer. In a similar manner to the $4 \times 5 \times 2$ and $5 \times 5 \times 2$ cases, after the randomized connections are utilized, there is negligible overfitting of the output in the upper $3-4$ data points. Nevertheless, the network seems to be capable of building a inner model and fitting the data regardless the training limitations imposed and the larger error (see Table 4).

According to the numerical experiments performed and discussed in this section, one could infer that not every time sequentially connected network provides the best solution to a certain task. At the same time, when our proposed evolutionary method uses more than one hidden layer with more neurons, it often reaches a model of a neuron removal, essentially discontinuing redundant calculations. In those situations, the network provides sufficiently good results according to the user specified cost function. One important observation is that most of the time, the neurons that are connected to the output layer use the logistic activation function. One might therefore conclude that in the bigger architectures, which provide

Table 4 Scenario 4: network architecture is limited to only four hidden layers process

Case	Num. layers	# Neurons in hidden layer	# Neurons in the network	Error
6 × 2 × 2	6	Hidden #1—1 neuron	6	0.010450
		Hidden #2—1 neuron		
		Hidden #3—1 neuron		
		Hidden #4—1 neuron		
6 × 3 × 2	6	Hidden #1—1 neuron	6	0.027712
		Hidden #2—1 neuron		
		Hidden #3—1 neuron		
		Hidden #4—1 neuron		
6 × 4 × 2	6	Hidden #1—2 neurons	10	0.039770
		Hidden #2—2 neurons		
		Hidden #3—2 neurons		
		Hidden #4—2 neurons		
6 × 5 × 2	3	Hidden #1—1 neuron	3	0.018634
6 × 6 × 2	6	Hidden #1—2 neurons	10	0.0271704
		Hidden #2—2 neurons		
		Hidden #3—2 neurons		
		Hidden #4—2 neurons		

Five cases in this scenario were developed—with implementation of maximum 2, 3, 4, 5 and 6 neurons per hidden layer. The fourth column represents the total amount of neurons in the network (including the neurons in the input and output layer), while the last column represents the final error from the evolutionary process

expanded search space of possible solutions, the evolutionary algorithm comes with several layers, but only one neuron from them is connected to the output layer.

4 Conclusions

In this paper, we have introduced a novel method for the automatic search of an optimal neural network architecture, given a specific problem (in our case, the fitting of a function). The underlying approach in the design of the method provides several degrees of freedom for the evolutionary algorithm to work with when searching for a network structure, which can eventually become quite complex. Even in the presence of this multidimensional space of possible topologies, our proposed strategy achieves sufficient performance at affordable computational cost. Our approach is capable to generate a network architecture while limiting the negative effects of overfitting. The numeric experiments and the obtained results validate our approach. Analogous to nature, many times simple tasks can be effectively solved with simplified procedures. According to the specific task, we believe the evolutionary strategy in our method reaches the optimal network topologies—whether through the automatic removal of unnecessary connections, or through the frequent usage of better activation function. Therefore, we obtain a reliable network topology not based on the experience of the researcher, but on well-defined automatic evolutionary strategy.

Acknowledgments This work has been supported by the project EC AComIn (FP7-REGPOT-20122013-1), by the Bulgarian Science Fund under Grant DFNI I02/20, and by the Grant DFNP-176-A1.

References

1. Haykin S (2009) Neural networks and learning machines, 3rd edn. Pearson Education, Upper Saddle River
2. Bishop CM (1993) Neural networks for pattern recognition. Clarendon Press, Cambridge
3. Mucherino A, Papajorgji PJ, Pardalos PM (2009) Data Mining in Agriculture, vol 34. Springer Science & Business Media
4. Hagan MT, Demuth HB, Beale MH, De Jesus O (2014) Neural network design, 2nd edn. Martin Hagan, New York
5. Kordik P, Koutnik J, Drchal J, Kovarik O, Cepek M, Snorek M (2010) Meta-learning approach to neural network optimization. Neural Netw 23(4):568–582
6. Almeida LM, Ludermir TB (2010) A multi-objective memetic and hybrid methodology for optimizing the parameters and performance of artificial neural networks. Neurocomputing 73:1438–1450
7. LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86:2278–2324
8. Yao X, Yong L (1997) A new evolutionary system for evolving artificial neural networks. IEEE Trans Neural Netw 8(3):694–713
9. Branke J (1995) Evolutionary algorithms for neural network design and training. In: Proceedings of the First Nordic Workshop on Genetic Algorithms and its Applications

10. Carvalho R, Ramos FM, Chaves AA (2011) Metaheuristics for the feedforward artificial neural network (ANN) architecture optimization problem. Neural Comput Appl 20(8):1273–1284
11. Balkin SD, Ord JK (2000) Automatic neural network modeling for univariate time series. Int J Forecast 16:509515
12. Ma L, Khorasani K (2003) A new strategy for adaptively constructing multilayer feedforward neural networks. Neurocomputing 51:361385
13. Stanley KO, Miikkulainen R (2002) Efficient evolution of neural network topologies. In: IEEE Proceedings of the 2002 Congress on Evolutionary Computation, vol 2
14. Stanley KO, Bryant BD, Miikkulainen R (2003) Evolving adaptive neural networks with and without adaptive synapses.In: IEEE The 2003 Congress on Evolutionary Computation, vol 4
15. Fahlman SE, Lebiere C (1991) The Cascade-Correlation Learning Architecture Technical report
16. Moriarty DE, Mikkulainen R (1996) Efficient reinforcement learning through symbiotic evolution. Mach Learn 22:11–32
17. Moriarty DE, Miikkulainen R (1997) Forming neural networks through efficient and adaptive coevolution. Evolut Comput 5(4):373–399
18. Angeline PJ, Saunders GM, Pollack JB (1994) An evolutionary algorithm that constructs recurrent neural networks. Trans Neural Netw 5(1):54–65
19. Gruau F, Whitley D, Pyeatt L (1996) A comparison between cellular encoding and direct encoding for genetic neural networks. In: Koza JR et al (eds) Genetic programming: proceedings of the first annual conference. MIT Press, Cambridge, pp 81–89
20. Coello CA, Van Veldhuizen DA, Lamont GB (2002) Evolutionary algorithms for solving multi-objective problems, vol 242. Kluwer Academic, New York
21. Yu J, Wang S, Xi L (2008) Evolving artificial neural networks using an improved PSO and DPSO. Neurocomputing 71(4):1054–1060
22. Liu LB, Wang L, Jin Y, Huang D (2007) Designing neural networks using PSO-based memetic algorithm. In: International Symposium on Neural Networks. Springer, Berlin, pp. 219–224
23. Maniezzo V (1994) Genetic evolution of the topology and weight distribution of neural networks. IEEE Trans Neural Netw 5(1):39–53
24. Yao X (1999) Evolving artificial neural networks. Proc IEEE 87(9):1423–1447
25. Kirkpatrick S, Gelatt CD Jr, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680
26. Thierens D, Goldberg D (1994) Convergence models of genetic algorithm selection schemes, parallel problem solving from nature PPSN III. Springer, Berlin Heidelberg
27. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to rpevent neural networks from overfitting. J Mach Learn Res 15:1929–2958
28. Cybenko G (1989) Approximation by superpositions of a sigmoidal function. Math Control Signals Syst 2(4):303–314