Genetic and Eugenetic Learning of RBF Networks

Kudová, Petra
2002

Genetic and Eugenic Learning of RBF Networks

doktorand:
PETRA KUDOVÁ
Ústav informatiky AV ČR
Pod vodárenskou věží 2,
18 207 Praha 8
petra@cs.cas.cz

školitel:
ROMAN NERUDA
Ústav informatiky AV ČR
Pod vodárenskou věží 2,
18 207 Praha 8
roman@cs.cas.cz

obor studia:
Teoretická informatika

Abstrakt

Learning methods for RBF networks are briefly reviewed. The focus is on evolutionary methods. As an alternative to the genetic algorithms we introduce the Eugenic learning, evolving separately a pool of hidden units and complete networks. Described algorithms are demonstrated on some experiments.

1. RBF network

An *RBF network* is a feed-forward neural network with one hidden layer of RBF units and a linear output layer. By an *RBF unit* we mean a neuron with multiple real inputs $\vec{x} = (x_1, \ldots, x_n)$ and one output y computed as:

$$y = \varphi(\xi); \quad \xi = \frac{\| \vec{x} - \vec{c} \|_C}{b} \tag{1}$$

where $\varphi : \mathbb{R} \to \mathbb{R}$ is a suitable activation function, let us consider Gaussian $\varphi(z) = e^{-z^2}$. The center $\vec{c} \in \mathbb{R}^n$, the width $b \in \mathbb{R}$ and an $n \times n$ real matrix C are a unit's parameters, $||\cdot||_C$ denotes a weighted norm defined as $\| \vec{x} \|_C^2 = (\mathbf{C}\vec{x})^T(\mathbf{C}\vec{x}) = \vec{x}^T\mathbf{C}^T\mathbf{C}\vec{x}$.

Thus, the network represents the following real function $\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$:

$$f_s(\vec{x}) = \sum_{j=1}^{h} w_{js} e^{-\left(\frac{\|\vec{x}-\vec{c}_j\|_{C_j}}{b_j}\right)^2}, \qquad s = 1, \ldots, m, \tag{2}$$

where $w_{js} \in \mathbb{R}$ are weights of s-th output unit and f_s is the s-th network output.

The goal of an RBF network learning is to find suitable values of RBF units' parameters and the output layer's weights, so that the RBF network function approximates a function given by a set of examples of

inputs and desired outputs $T = \{\vec{x}(t), \vec{d}(t); t = 1, \ldots, k\}$, called a *training set*. The quality of the learned RBF network is measured by the *error function*:

$$E = \frac{1}{2} \sum_{t=1}^{k} \sum_{j=1}^{m} e_j^2(t), \qquad e_j(t) = d_j(t) - f_j(t). \tag{3}$$

2. RBF network learning

There is quite a wide range of methods used for RBF networks learning, three main approaches were described in [**?**].

From one point of view an RBF network is a special type of a feed-forward network, like a multilayer perceptron (MLP). Therefore we can use a modification of the Back Propagation algorithm, designed for MLPs. It is an iterative algorithm based on a straightforward gradient minimization of the error function E from (3). Starting with random initialization, we shift the vector of all network parameters in the direction given by the gradient of the error function, and iterate until a local minimum or the desired value of the error function is reached. We call it the *Gradient learning*, since there is only one hidden layer and therefore no "back propagation" during the evaluation of the gradient.

The Gradient learning unifies all parameters by treating them in the same way, even though the meaning of the RBF network parameters is well defined. Another method, *Tree-step method* takes advantage of this knowledge, divides the parameters into three categories and learns each category separately using a customized method.

First we need to set the centers of RBF units. They should reflect the density of training data points and therefore some vector quantization algorithm is usually used. Then we find suitable values for additional units' parameters (widths, norm matrices), which determine the size and the shape of the area controlled by a unit. Some heuristics are used to cover the whole input space with these areas and at the same time to keep their local character. Remaining weights of a linear combination are determined by a linear least squares method.

A completely different approach to RBF learning is represented by evolutionary methods, such as *genetic algorithms*. It is a stochastic optimization method, working with a population of individuals and operators of crossover, mutation and selection constructing new generations. Each individual represents one feasible setting of network parameters, the better the represented network, the higher the individual's *fitness* (a probability of being selected into a new generation).

In [**?**] the canonical version of the genetic algorithm is introduced. It reduces the searched parameter space using the fact that RBF networks with the same but permuted RBF units represent the same function and thus are equivalent.

Another evolutionary method, based on Eugenic evolution [**?**], is described in the next section.

All mentioned methods are candidates for RBF network learning. The real choice depends on actual computational resources or a character of a given data set. The Three-step method is the best choice from the computational time point of view, the Gradient method usually finds the most accurate solution. Evolutionary methods are quite time consuming but can be used also for determination of a number of hidden units or for learning of networks with different types of units in the hidden layer.

3. RBF network learning based on Eugenic evolution

Since an RBF network represents a linear combination of RBF units, we can understand it as a *team* of cooperating units. Thanks to the local character of RBF units, we can assume that if some unit works well on its local area it would probably be a good member of a team. Thus, if we had a good supply of those units, we would probably be able to select teams forming good networks.

Now we introduce a learning technique based on the eugenic evolution described in [**?**].

The evolution takes place on two levels. On the first level we evolve a pool of RBF units that serves as a store of building blocks for RBF networks. Good units are those that are able to cooperate with other units well enough to solve a given task.

A network, understood as a team of units, is represented by a *blueprint* specifying which units form its hidden layer. A population of blueprints is evolved on the second level.

RBF units evolution

First we describe one step of the first level evolution, i.e. evolution of units:

- According to a chosen blueprint we create a network that is used to solve a given task (i.e. is applied on a given training set) in a trial, and is rewarded by a fitness based on the error function. All units of the current team increase their trial count and the network fitness is added to their fitness count.
- After enough trials have been performed, each neuron average fitness is evaluated dividing its fitness count by its trial count.
- New units are constructed using the operators of selection, crossover and mutation and they replace the lower ranking part of the population.

Starting with randomly initialized neurons and blueprints, we iterate until a suitable solution is found.

To form a network we could choose a team in some random way, but we would prefer to build teams of units that are likely to work well together. Therefore the evolution on the second level is done by means of a eugenic algorithm.

Here, an individual represents one blueprint and it is coded by a bitmap. Each bit represents one gene and corresponds to one RBF unit from a pool. It is set to one, if this unit is included in the hidden layer of a particular network.

The introduction to the general Eugenic algorithm follows.

Eugenic algorithm

Unlike a standard genetic algorithm, where crossover, mutation and selection are used to form a new individual, in Eugenic algorithm a new individual is constructed one gene after another according to the statistics over the entire population.

Starting with a random initialization, we iterate until a suitable solution is found.

In each generation only one new individual $\mathcal{N}$ is generated:

1. Start with the set of all genes $U = \{x_1, x_2, \ldots, x_n\}$
2. Select the most significant gene x_{sig}. This should be the gene that is most strongly correlated with the fitness of the individuals. Assume that the larger the $|f(x_{g,0}) - f(x_{g,1})|$, the stronger the influence of the gene x_g is. ($f(x_{g,0})$ is the average fitness of those individuals having 0 at gene x_g, similarly for $f(x_{g,1})$).
3. Set the value of the gene x_{sig} according to the probability $f(x_{sig,a})/(f(x_{sig,1}) + f(x_{sig,0}))$, where a is either 1 or 0. Higher probability have those values that are more likely (according to a current population) to form good individuals.
4. The gene x_{sig} is removed from U.
5. A rough estimate of *epistasis* is evaluated for the population.

   It is a measure of how strongly the fitness $f(x_{g,a})$ depends on the values of the other genes. If for all genes g the difference between $f(x_{g,0})$ and $f(x_{g,1})$ is low, i.e. both choices seems to be equally

good, we speak about high epistasis. High epistasis indicates that there is no really significant gene and therefore the values for remaining genes should not be chosen independently.

6. If epistasis is high, the population is restricted to those individuals having the same value of gene x_{sig} as the individual $\mathcal{N}$.
7. Use the restricted population and continue with step 2 until U is empty.
8. Compute the fitness of the individual $\mathcal{N}$ (including error function evaluation of the corresponding network) and replace the individual with worst fitness.

The described algorithm doesn't control the number of units in the hidden layer. To avoid large or empty networks, we explicitly define the minimal and maximal number of units. Creation of a new individual is stopped after the maximal number of units is reached (i.e. the same number of bits is set to 1). If the created individual has less than minimal number of units, randomly chosen bits are set to 1. Then the algorithm determines not only the values of networks parameters, but also the number of hidden units.

4. Experiments

In this section we present some results of our experiments.

All experiments were run on a Linux cluster, each computation on an individual node with a Celeron 1.1GHz processor and 0.5 GB of memory.

The well-known data set *Iris plants* was used. It contains 3 classes of 50 instances each, where each class refers to a type of an iris plant. We used a network with three output neurons, one neuron for each class. The class number is then coded by three binary values, value 1 on the position corresponding to the number of class and zeros on the others. So each training sample consists of 4 input values describing some features of a plant and 3 output values coding its class.

We tried to learn an RBF network with 3, 6, 9 and 12 RBF units, using genetic algorithms and eugenic learning. Each computation was run 5 times, an average computation was picked up. The average time of 100 iterations was 72.0 s for genetic learning, 8.6 s for eugenic learning.

Figures 6 and 7 shows the typical fall of the error function comparing the canonical and the classical version of genetic algorithms.

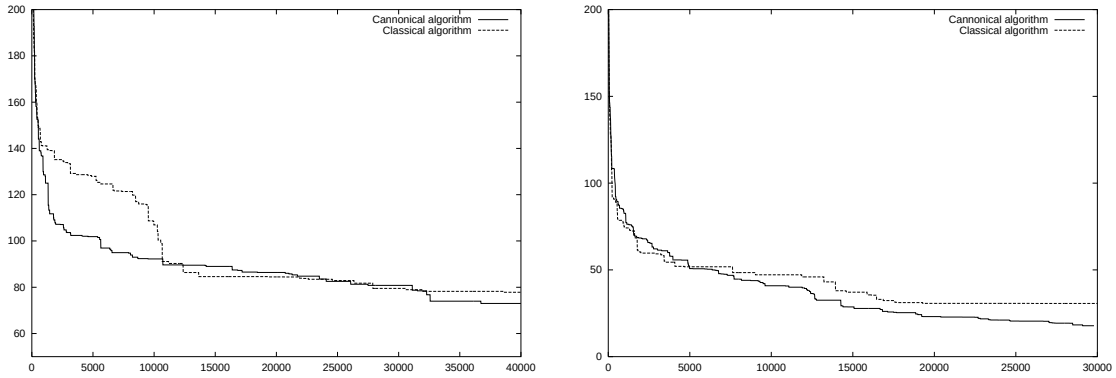
On figure 8a) you can see the typical fall of the error function using eugenic learning for networks with 3, 6, 9 and 12 units.

Eugenic learning was run also with a varying number of RBF units, the minimal number of units was 3, maximal 12. Figure 8b) shows the fall of the error function.

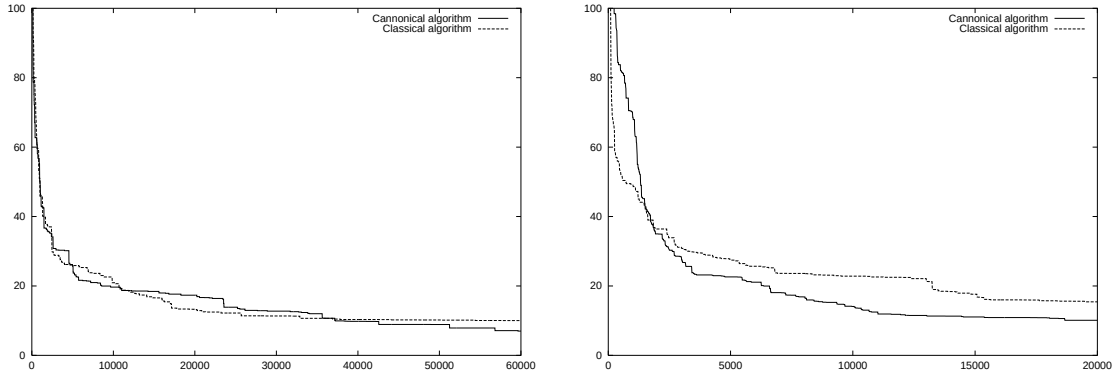
Genetic and Eugenic learning of an RBF network with 12 hidden units are compared in the table 1, that shows the number of iterations and time needed to achieve given ε.

Note that, while Genetic learning creates a whole population of new individuals and evaluates the value of the error function for all of them each iteration, the Eugenic learning evaluates the error function only for the new individual and those individuals that mutated during last iteration. Therefore there is no use of comparing the numbers of iterations and we have to measure time.

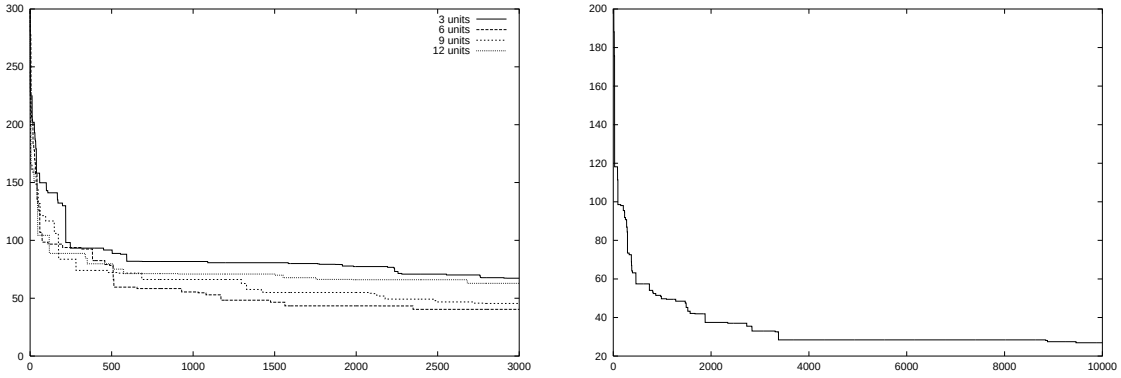
On the *Iris plant* problem we demonstrated, that Eugenic algorithm can achieve results comparable to genetic algorithms in shorter time.



Obrázek 6: The fall of error function in canonical genetic learning and classical genetic learning, for the RBF network with a) 3 units b) 6 units.



Obrázek 7: The fall of error function in canonical genetic learning and classical genetic learning, for the RBF network with a) 9 units b) 12 units.



Obrázek 8: The fall of the error function in eugenic learning for the RBF network with a) 3, 6, 9 and 12 units. b) the varying number of units, from 3 to 12.

ε	**EuGA**		CanGA		GA	
100	**95**	**8 s**	30	22 s	20	14 s
80	**291**	**25** s	240	3 min 18 s	100	1 min 22 s
60	**463**	**40 s**	1320	18 min 12 s	720	9 min 55 s
40	**1879**	**2 min 42 s**	3420	47 min 11 s	6810	1 hours 33 min 11 s
20	**151717**	**3 hours 37 min 27 s**	22390	5 hours 8 min 58 s	28590	6 hours 35 min 54 s

Tabulka 1: Average number of iterations and time needed for the fall of the error function under the given ε.

5. Conclusion

We reviewed common learning methods for RBF networks, including evolutionary methods. The Eugenic learning based on two-level evolution was introduced. It takes advantage of local character of hidden units and more sophisticated way for creating new individuals is used. It seems to be a promising alternative to genetic algorithms.

Our future focus should be moved to hybrid methods combining known algorithms.

In [**?**] it was already shown that the Three step method combined with Gradient learning could bring better results than these methods alone. Also genetic algorithms could be easily combined with other methods, for instance they are very good at setting centers in the Three step method.

And finally, the Eugenic learning described in this paper takes advantage of local character of units while evolving a pool of them. But we have to learn also the weights of the output layer, which are highly dependent on other weights and units in a current network. Therefore we want to let the Eugenic learning learn only the hidden layer and set the output weights using some linear least squares method.