Chapter 6
GAME - Hybrid Self-Organizing Modeling System based on GMDH

6.1 Introduction

In this chapter, you will find a description of the recently introduced Adaptive Models Evolution (GAME) algorithm [24] with respect to its self-organizing properties and the hybrid nature of its building blocks. The GAME algorithm uses a data driven approach. Resulting models fully reflect the character of a data set used for training. For simple problems, it evolves simple models (in terms of topology and transfer functions) and for a complex relationship of variables, a complex model is evolved. The GAME algorithm proceeds from the Multilayered Iterative Algorithm (MIA *GMDH*) [18].

GAME models are *self-organized* layer by layer by means of a special genetic algorithm preserving diverse solutions.

The optimization of transfer functions in neurons (or units or partial descriptions in the GMDH terminology) is solved independently. Several optimization methods compete to adjust the coefficients of transfer functions.

Neurons can be of several types – the polynomial transfer function can be good at fitting certain relationships, but often a different transfer function is needed (e.g. sigmoid for classification purposes). GAME models mostly consist of several different types of neurons that are optimized by different methods – they have a *hybrid* character. Also the ensemble of GAME models is often produced to get even better bias-variance trade-off and to be able to estimate the credibility of the output for any configuration of input variables.

The hybrid character of GAME models and their self-organizing ability give them an advantage over standard data mining models. Our experiments show that the performance of hybrid models is superior on a large scale of different data sets.

Below, you will find a detailed description of the GAME algorithm and the ideas behind it.

6.1.1 Self-organizing modelling

The Group Method of Data Handling (GMDH) was invented by A.G. Ivakhnenko in the late 1960s [18]. He was looking for computational instruments allowing him to model real world systems characterized by data with many inputs (dimensions) and few records. Such ill-posed problems could not be solved traditionally (ill-conditioned matrixes) and therefore a different approach was needed. Prof. Ivakhnenko proposed the GMDH method, which avoided the solution of ill-conditioned matrixes by decomposing them into submatrices of lower dimensionality that could be solved easily. The main idea behind the GMDH is the adaptive process of combining these submatrices back to the final solution, together with external data set validation preventing data overfitting. The original GMDH method is called Multilayered Iterative Algorithm (MIA GMDH). Many similar GMDH methods based on the principle of induction (problem decomposition and combination of partial results) have been developed since then.

The only possibility of modelling real world systems before the GMDH was to manually create a set of mathematical equations mimicking the behavior of a real world system. This involved a lot of time, domain expert knowledge and also experience with the synthesis of mathematical equations.

The GMDH allowed for the automatic generation of a set of these equations. A model of the real world system can also be created by Data Mining (DM) algorithms, particularly by artificial Neural Networks (NNs).

Some DM algorithms such as decision trees are simple to understand, whereas NNs often have so complex structure that they are necessarily treated as a black-box model. The MIA GMDH is something in between - it generates polynomial equations which are less comprehensible than a decision tree, but better interpretable than any NN model.

The main advantage of the GMDH over NNs is that the optimal topology of the network (number of layers and neurons, transfer functions) is determined automatically. Traditional neural networks such as MLP [32] require the user to experiment with the size of the network. Recently, some NNs also adopted the self-organizing principle and induct topology of models from data.

The MIA GMDH builds models layer by layer, while the accuracy of the model on the validation data set increases (as described in the Chapter 1. of this book). However, the accuracy of resulting models is not very high when applied to different benchmarking problems [27]. The reason is that it selects the optimal model from a very limited state space of possible topologies.

6.1.1.1 Limitations of MIA GMDH

The MIA GMDH (see Fig. 6.1) was invented 40 years ago and therefore incorporates several limitations that were necessary to make the computation feasible.

The limitations of the MIA GMDH are as follows:

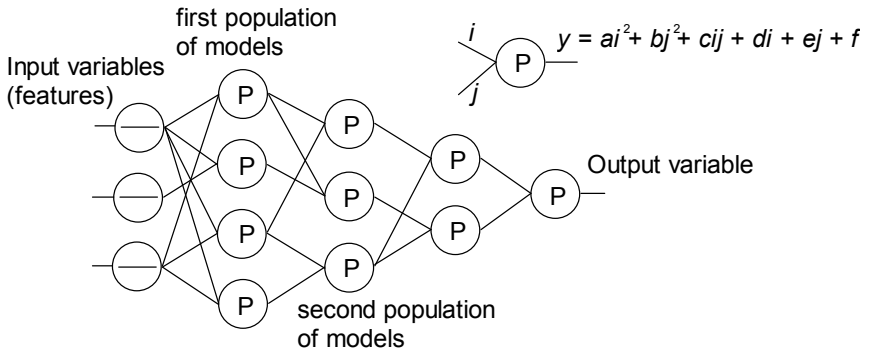

Fig. 6.1 An example of the inductive model produced by the GMDH MIA algorithm.

- All neurons have the same transfer function
- Transfer function is simple polynomial
- Each polynomial neuron has exactly two inputs
- Inputs are chosen from the previous layer only

These structural limitations allow the MIA algorithm to check all possible combinations of a neuron's interconnections and choose the best. In other words, the algorithm searches the whole state space of possible MIA GMDH topologies, and returns the optimal topology for this state space.

The problem is that all other possible topologies of models (e.g. model containing a neuron with 4 inputs and sigmoid transfer function) are not examined, although they can provide us with a better solution.

If we decide to drop GMDH MIA limitations, our search space expands in both size and dimensionality. With new degrees of freedom (number of layers and neurons, interconnections of neurons, their type of transfer functions, values of coefficients, type of optimization method used etc.), the search space of all possible topologies becomes mind-bogglingly huge.

Advances in computer technology and the appearance of modern heuristic optimization methods allow an algorithm to navigate through the search space efficiently, obtaining almost the optimal solution in a reasonable time.

The experimental results show that the GAME algorithm, we have proposed for this purpose, outperforms GMDH MIA considerably.

6.1.1.2 Self-organizing neural networks

Self Organizing Map (SOM) [22] is a typical example of how *self-organization* is understood in the area of neural networks. This network has fixed topology and neurons self-organize during training by weights updates to reflect the density of data in hyperspace. In the Growing SOM [40] variant, the matrix of neurons increases in size (and dimension) from the minimal form – the topology is self-organized as well.

These networks are unsupervised, whereas this book focuses mainly on supervised methods.

Supervised neural networks such as MLP have fixed topology and only weights and neuron biases are the subject of training. A suitable topology for given problem has to be determined by user, usually by an exhaustive trial and error strategy.

Some more recently introduced neural networks demonstrate self-organizing properties in terms of topology adaptation.

The Cascade Correlation algorithm [10] generates a feedforward neural network by adding neurons one by one from a minimal form. Once a neuron has been added to the network, its weights are frozen. This neuron then becomes a feature-detector in the network, producing outputs or creating other feature detectors. This is a very similar approach to the MIA GMDH as described in Chapter 1.

It has been shown [10] that cascade networks perform very well on the ”two intertwined spirals” benchmarking problem (a network consisting of less than 20 hidden neurons was able to solve it) and the speed of training outperformed Backpropagation.

According to experiments on real-world data performed in [52], the algorithm has difficulties with avoiding premature convergence to complex topological structures. The main advantage of the Cascade Correlation algorithm is also its main disadvantage. It easily solves extremely difficult problems therefore it is likely to overfit the data.

In the next section, we introduce a robust algorithm that can generate feedforward neural networks with adaptive topology – learning, structure of network and transfer functions of neurons are tailored to fit the data set.

6.2 Group of Adaptive Models Evolution (GAME)

6.2.1 The concept of the algorithm

The Multilayered perceptron neural networks trained by the Backpropagation algorithm [41] are very popular even today, when many better methods exist. The success of this paradigm is mostly given by its robustness. It works reasonably well for a large scale of problems of different complexity, despite the fixed topology of the network and uniform transfer functions in neurons. The Group of Adaptive Models Evolution (GAME) algorithm, proposed in this chapter, has the ambition to be even more robust and more adaptive to individual problems. The topology of GAME models adapts to the nature of a data set supplied.

6.2.1.1 The pseudocode of the GAME algorithm

The GAME algorithm is a supervised method. For training, it requires a dataset with input variables (features) and the output variable (target).

The GAME algorithm, in summary, is described below:

1. Separate the validation set from the training data set (50% random subset)
2. Initialize first population of neurons – input connections, transfer functions and optimization methods chosen randomly
3. Optimize coefficients of neurons' transfer functions by assigned optimization method – error and gradients computed on the training data set
4. Compute fitness of neurons by summarizing their errors on the validation set
5. Apply Deterministic Crowding to generate new population (randomly select parents, competition based on fitness and distance to be copied into the next generation)
6. Go to 3), until diversity level is to low or the maximum number of epochs (generations) is reached
7. Select the best neuron from each niche – based on fitness and distance, freeze them to make up the layer and delete the remaining neurons
8. 8) Until the validation error of the best neuron is significantly lower than the best neuron from the previous layer, proceed with the next layer and go to 2)
9. Mark the neuron with the lowest validation error as the output of the model and delete all neurons not connected to the output

6.2.1.2 An example of the GAME algorithm on the Housing dataset

We will demonstrate our algorithm on the Housing dataset that can be obtained from the UCI repository [2]. The dataset has 12 continuous input variables and one continuous output variable. In Fig. 6.2 you can find the description of the most important variables.

Firstly, we split the data set into a subset used for training (A+B) and a test set to get an unbiased estimate of the model's error (see Fig. 6.3). Alternatively, we can perform k-fold crossvalidation [21].

Then we run the GAME algorithm, which separates out the validation set (B) for the fitness computation and the training set (A) for the optimization of coefficients.

In the second step, the GAME algorithm initializes the population of neurons (the default number is 15) in the first layer. For instant GAME models, the preferable option is *growing complexity* (number of input connections is limited to index of layer). Under this scheme, neurons in the first layer cannot have more than one input connection, as shown in Fig. 6.2.1.2. The type of the transfer function is assigned randomly to neurons, together with the type of method to be used to optimize coefficients of transfer function.

The type of transfer function can be sigmoid, Gaussian, linear, exponential, sine and many others (a complete and up-to-date list of implemented transfer functions is

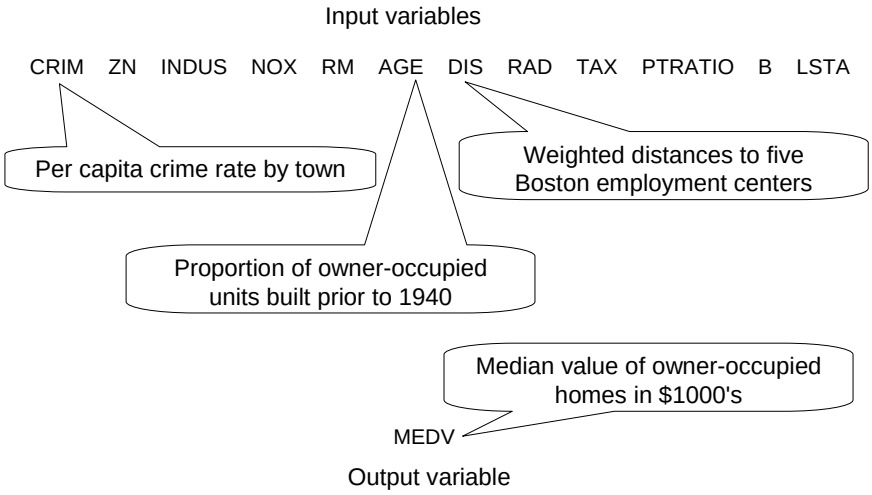

Fig. 6.2 Housing data set: the description of the most important variables.

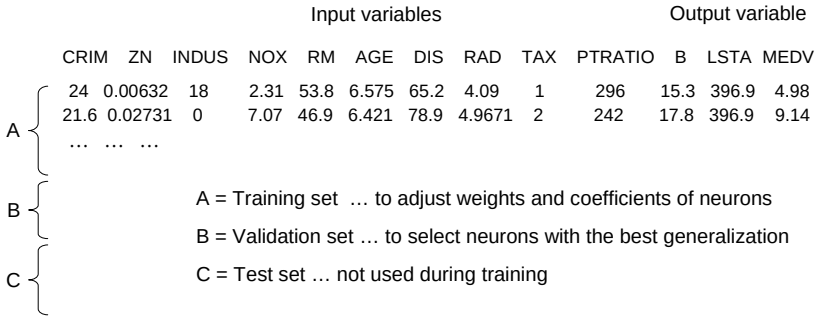

Fig. 6.3 Splitting the data set into the training and the test set; the validation set is separated from the training set automatically during GAME training.

available in the FAKEGAME application [5]), see Fig. 6.5. If the sigmoid transfer function is assigned to a neuron, the output of this neuron can be computed, for example as MEDV=1/(1-exp(-$\mathbf{a}_1$*CRIM+ $\mathbf{a}_0$)), where coefficients $\mathbf{a}_0$ and $\mathbf{a}_1$ are to be determined.

To determine these coefficients, an external optimization method is used. The optimization method is chosen from a list of available methods (Quasi-Newton, Differential Evolution, PSO, ACO, etc.).

..

The fitness of each individual (neuron) is computed as the inverse of its validation error. The genetic algorithm performs selection, recombination and mutation and the next population is initialized. After several epochs, the genetic algorithm is stopped and the best neurons from individual niches are frozen in the first layer (Fig. 6.6).

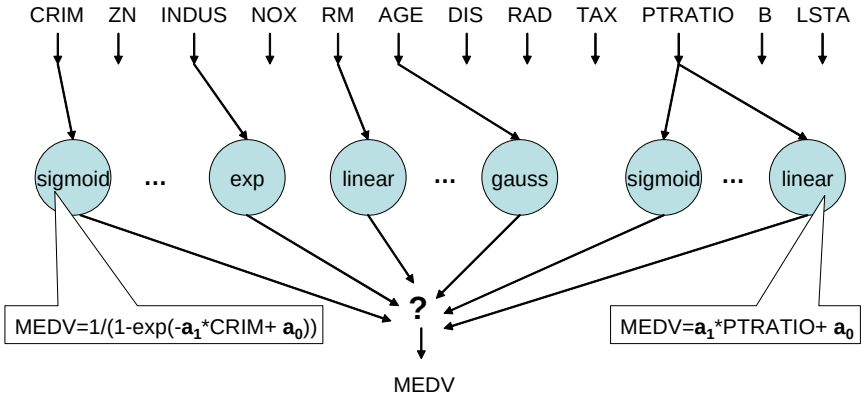

Fig. 6.4 Initial population of neurons in the first GAME layer.

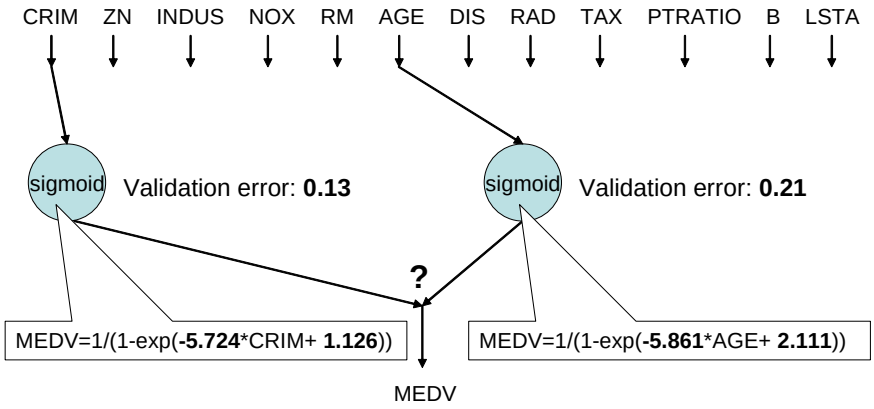

Fig. 6.5 Two individuals from different niches with coefficients optimized on set A and validated on set B. The neuron connected to the AGE feature has a much higher validation error than neurons connected to CRIM and survives thanks to *niching*.

Then the GAME algorithm proceeds with the second layer. Again, an initial population is generated with random chromosomes, evolved by means of the niching genetic algorithm, and then the *best and diverse* neurons are selected to be frozen in the second layer.

The algorithm creates layer by layer, until the validation error of the best individual decreases significantly. Fig. 6.2.1.2 shows the final model of the MEDV variable.

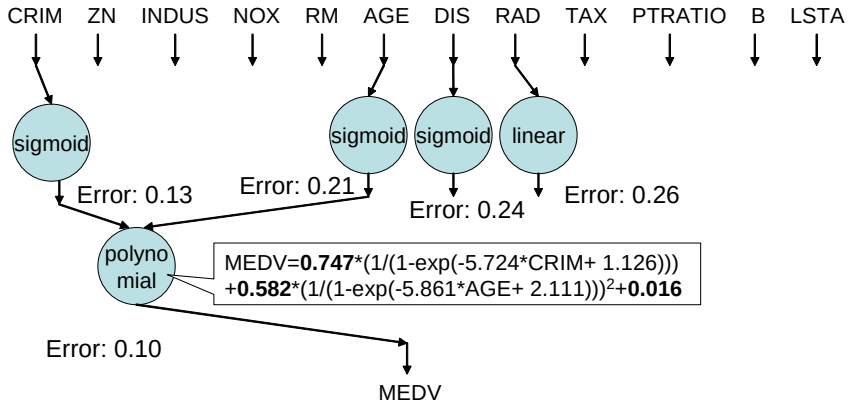

Fig. 6.6 In our example, the best individual evolved in the second layer combines the outputs of neurons frozen in the first layer (feature detectors).

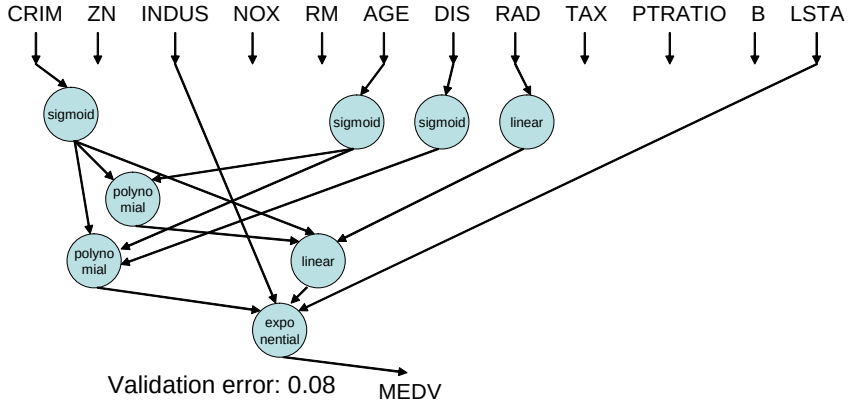

Fig. 6.7 The GAME model of the MEDV variable is finished when new layers do not decrease the validation error significantly.

6.2.2 Contributions of the GAME algorithm

The GAME algorithm proceeds from the MIA GMDH algorithm. In this section, we summarize improvements to the Multilayered GMDH as described in Chapter 1.

The Fig. 6.8 illustrates the difference between models produced by the two algorithms.

The GAME model (see Fig. 6.8, right) has more degrees of freedom (neurons with more inputs, interlayer connections, transfer functions etc.) than MIA GMDH models. To search the huge state space of model's possible topologies, the GAME algorithm incorporates the niching genetic algorithm in each layer.

Improvements to the MIA GMDH are discussed below in more detailed form.

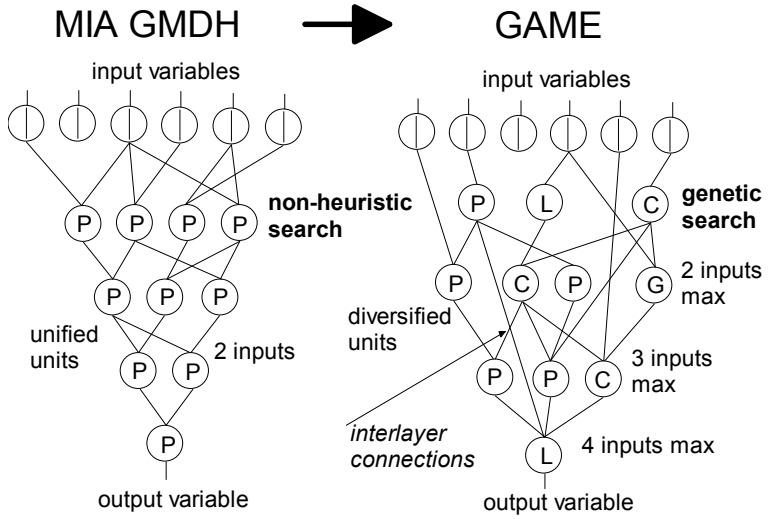

Fig. 6.8 Comparison: original MIA GMDH network and the GAME network

- *Heterogeneous neurons* - several types of neurons compete to survive in GAME models.
- *Optimization of neurons* - Efficient gradient based training algorithm developed for hybrid networks.
- *Heterogeneous learning methods* - Several optimization methods compete to build the most successful neurons.
- *Structural innovations* - Growth from a minimal form, interlayer connections etc.
- *Regularization* - Regularization criteria are employed to reduce the complexity of transfer functions.
- *Genetic algorithm* - A heuristic construction of GAME models. Inputs of neurons are evolved.
- *Niching methods* - Diversity promoted to maintain less fit but more useful neurons.
- *Evolving neurons (active neurons)* - Neurons such as the CombiNeuron evolve their transfer functions.
- *Ensemble of models generated* - Ensemble improves accuracy; the credibility of models can be estimated.

6.2.2.1 Heterogeneous neurons

In MIA GMDH models, all neurons have the same polynomial transfer function. The Polynomial Neural Networks (PNN) [36] models supports multiple types of polynomials used within a single model.

Our previous research showed that employing heterogeneous neurons within a model gives better results when using neurons of a single type only [27]. Hybrid

models are often more accurate than homogeneous ones, even if the homogeneous model has a suitable transfer function appropriate for modelled system.

In GAME models, neurons within a single model can have several types of transfer functions (Hybrid Inductive Model). Transfer functions can be linear, polynomial, logistic, exponential, Gaussian, rational, perceptron network etc. (see Table 6.1 and Fig. 6.9).

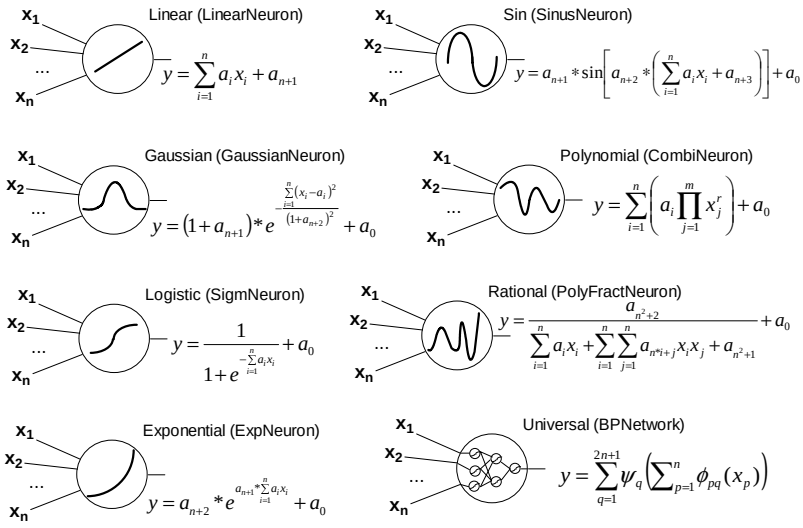

Fig. 6.9 Neurons are building blocks of GAME models. Transfer functions of neurons can be combined in a single model (then we call it a hybrid model with heterogeneous neurons). The list of neurons includes some neurons implemented in the FAKE GAME environment.

The motivation, for implementing so many different neurons was as follows. Each problem or data set is unique. Our previous experiments showed [27] that for simple problems, models with simple neurons were superior, whereas for complex problems, the winning models were those with neurons having a complex transfer function. The best performance on all tested problems was achieved by models where the neurons were mixed.

6.2.2.2 Experiments with heterogeneous neurons

To prove our assumptions and to support our preliminary results [27], we designed and conducted the following experiments. We used several real world data sets of various complexity and noise levels. For each data set, we built simple ensembles [13] of 10 models. Each ensemble had a different configuration. In ensembles of homogeneous models, there was just a single type of neuron allowed (e.g. **Exp** stands for an ensemble of 10 models consisting of ExpNeuron neurons only). Ensembles

Table 6.1 Summary of neuron types appearing in GAME networks

Name of neuron	Transfer function.	Learning method
LinearNeuron	Linear	- any method -
LinearGJNeuron	Linear	Gauss-Jordan method
CombiNeuron	Polynomial	- any method -
PolySimpleNeuron	Polynomial	- any method -
PolySimpleGJNeuron	Polynomial	Gauss-Jordan method
PolyHornerNeuron	Polynomial	- any method -
PolySimpleNRNeuron	Polynomial	- any method + GL5 -
SigmNeuron	Sigmoid	- any method -
ExpNeuron	Exponential	- any method -
PolyFractNeuron	Rational	- any method -
SinusNeuron	Sinus	- any method -
GaussNeuron	Gaussian	- any method -
MultiGaussNeuron	Gaussian	- any method -
GaussianNeuron	Gaussian	- any method -
BPNetwork	Universal	BackPropagation algorithm
NRBPNetwork	Universal	BP alg. + GL5 stop.crit.

of heterogeneous inductive models, where all types of neurons are allowed to participate in the evolution, are labelled **all**, **all-simple** and **all-fast** respectively. In the all-simple configuration, Linear, Sigmoid and Exponential functions were enabled, in the all-fast configurations, Linear, Sigmoid, Polynomial, Exponential, Sine, Rational and Gaussian transfer functions were employed.

For all experiments in this section, we used only one optimization method (Quasi-Newton) to avoid biased results.

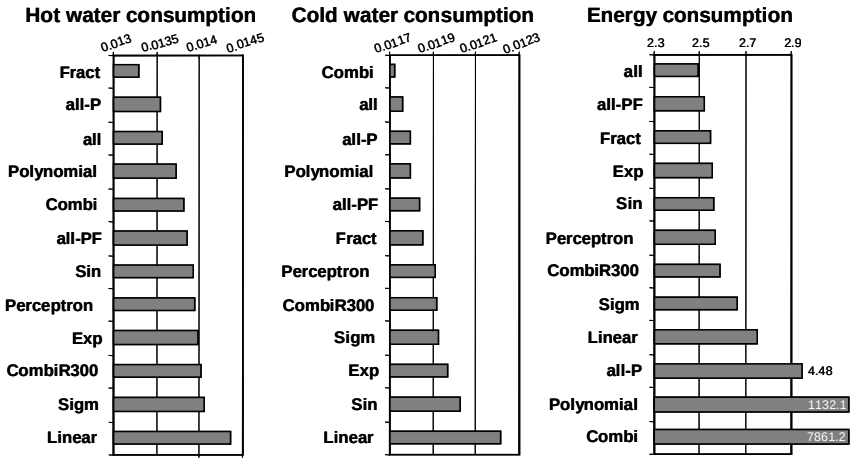

Fig. 6.10 Performance comparison of GAME neurons on the Building data set. In the all-PF configuration all neurons except the Perceptron and Fract neurons were enabled; similarly in all-P only the Perceptron neuron was excluded.

The first experiment was performed on the Building data set. This data set has three output variables. One of these variables is considerably noisy (Energy consumption) and the other two output variables have low noise levels. The results are consistent with this observation. The **Combi** and the **Polynomial** ensembles perform very well on variables with low noise levels, but for the third, "noisy" variable, they both overfitted the training data (having a huge error on the testing data set).

Notice that the configuration **all** has an excellent performance for all three variables, no matter what the level of noise (Fig. 6.10).

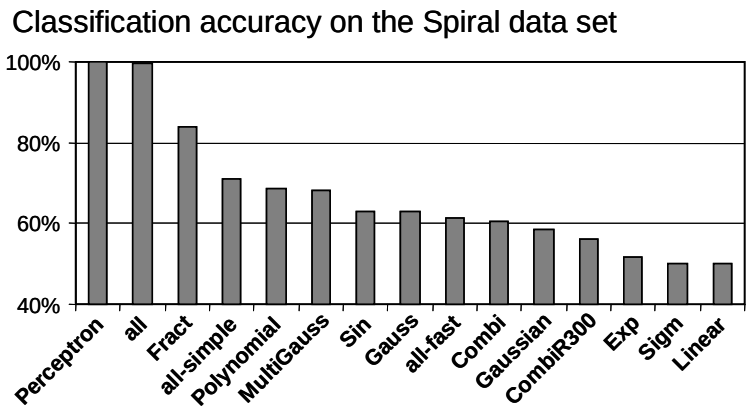

Fig. 6.11 Performance comparison of GAME neurons on the Spiral data set.

In the Fig. 6.11, we present the results of the experiment on the Spiral data set [20]. As you can see, the **Perceptron** ensemble learned to tell two spirals apart without any mistake. The second best performing configuration was **all** with almost one hundred percent accuracy[1]. The worst performing ensembles were **Linear** and **Sigm** (neurons with linear and logistic transfer functions). Their 50% classification accuracy signifies that these neurons absolutely failed to learn the Spiral data set. The failure of neurons with logistic transfer function signifies that the GAME algorithm is not as efficient on this problem as the Cascade Correlation algorithm using 16 sigmoid neurons on average to solve this problem.

We performed a number of similar experiments with other real world data sets. We can conclude that the **all** ensemble performed extremely well for almost all data sets under investigation.

The conclusion of our experiments is that for best results we recommend enabling all neurons which have been so far implemented in the GAME engine. The more types of transfer function we have, the more diverse relationships we can model. The type of selected neurons depends only on the nature of the data modelled. The main advantage of using neurons of various types in a single model is

[1] We have to mention that building the ensemble of **all** models took just a fraction of time needed to build the ensemble of **Perceptron** models (consisting of BPnetwork neurons).

that models are adapted to the character of the modelled system. Only neurons with appropriate transfer function survive. Hybrid models also better approximate relationships that can be expressed by the superposition of different functions (e.g. polynomial * sigmoid * linear).

These results are a significant step towards automated data mining, where exhaustive experiments with optimal configuration of data mining methods are no longer necessary.

In the same sense, we also use several types of optimization methods.

6.2.3 Optimization of GAME neurons

The process of learning aims to minimize the error of each neuron (distance of output from target variable for every training instance).

$$E = \sum_{j=0}^{m} (y_j - d_j)^2, \tag{1}$$

where $y_j$is the output of the model for the jth training vector and $d_j$is the corresponding target output value.

The optimal values of parameters are values minimizing the difference in behavior between a real system and its model. This difference is typically measured by a root mean squared error.

The aim of the learning process is to find values of transfer function coefficients $a_1, a_2, \cdots, a_n$ in order to minimize the error of the neuron.

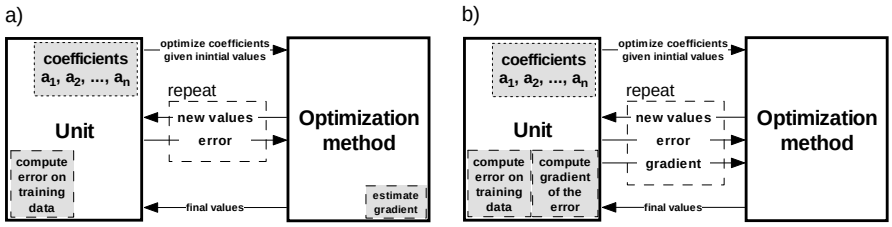

Fig. 6.12 Optimization of the coefficients can be performed without the analytic gradient a) or with the gradient supplied b). Utilization of the analytic gradient significantly reduces the number of iterations needed for the optimization of coefficients.

Most of the coefficients are continuous without constraints. When the transfer function of the neuron is differentiable, we can derive the gradient of the error. The analytic gradient helps the optimization method to adjust coefficients efficiently, providing faster convergence.

Optimal values of coefficients cannot be determined in one step. After random initialization of their values, the error of the neuron is computed (e.g. 1) and the

optimization method proposes new values of coefficients, after that, error is computed again (see Fig. 6.12a). This single optimization step is called iteration. If the analytical gradient of the error can be computed, the number of iterations would be significantly reduced, because we know in which direction coefficients should be adjusted (see Fig. 6.12b). The gradient of the error $\nabla\mathbf{E}$ in the error surface of a GAME neuron can be written as

$$\nabla\mathbf{E} = \left(\frac{\partial\mathbf{E}}{\partial a_1}, \frac{\partial\mathbf{E}}{\partial a_2}, \cdots, \frac{\partial\mathbf{E}}{\partial a_n}\right), \tag{2}$$

where $\frac{\partial\mathbf{E}}{\partial a_i}$ is a partial derivative of the error in the direction of the coefficient a_i. It tell us how to adjust the coefficient to get a smaller error E on the training data. This partial derivative can be computed as

$$\frac{\partial\mathbf{E}}{\partial a_i} = \sum_{j=0}^{m} \frac{\partial\mathbf{E}}{\partial y_j} * \frac{\partial y_j}{\partial a_i}, \tag{3}$$

where m is the number of training vectors. The first part of the summand can be is easily derived from the Equation 1 as

$$\frac{\partial\mathbf{E}}{\partial y_j} = 2\sum_{j=0}^{m} (y_j - d_j). \tag{4}$$

The second part of the summand from the Equation 3 is unique for each neuron, because it depends on its transfer function. We demonstrate the computation of the analytic gradient for the Gaussian neuron. For other neurons the gradient is computed in a similar manner.

6.2.3.1 The analytic gradient of the Gaussian neuron

Gaussian functions are very important and can be found almost everywhere. The most common distribution in nature follows the Gaussian probability density function $f(x) = \frac{1}{2\pi\sigma} * e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. Neurons with Gaussian transfer function are typically used in Radial Basis Function Networks. We have modified the function for our purposes. We added coefficients to be able to scale and shift the function. The first version of the transfer function as implemented in *GaussianNeuron* is the following:

$$y_j = (1 + a_{n+1}) * e^{-\frac{\sum_{i=1}^{n}\left(x_{ij} - a_i\right)^2}{\left(1+a_{n+2}\right)^2}} + a_0 \tag{5}$$

The second version (*GaussNeuron*) proved to perform better on several low dimensional real world data sets:

$$y_j = (1+a_{n+1}) * e^{-\frac{\sum_{i=1}^{n}\left(a_i * x_{ij} - a_{n+3}\right)^2}{\left(1+a_{n+2}\right)^2}} + a_0 \tag{6}$$

Finally, the third version (*MultiGaussNeuron*), as the combination of the transfer functions above showed the best performance, but sometimes exhibited almost fractal behavior.

$$y_j = (1+a_{2n+1}) * e^{\underbrace{-\frac{\sum_{i=1}^{n}\left(a_i * x_{ij} - a_{n+i}\right)^2}{\left(1+a_{2n+2}\right)^2}}_{\rho_j}} + a_0 \tag{7}$$

We computed gradients for all these transfer functions. Below, we derive the gradient of the error (see Equation 2) for the third version of the Gaussian transfer function (Equation 7). We need to derive partial derivatives of the error function according to Equation 3. The easiest partial derivative to compute is the one in the direction of the a_0 coefficient. The second term $\frac{\partial y_j}{\partial \rho_j}$ is equal to 1. Therefore we can write $\frac{\partial \mathbf{E}}{\partial a_0} = 2\sum_{j=0}^{m}(y_j - d_j)$. In the case of the coefficient a_{2n+1}, the equation becomes more complicated

$$\frac{\partial \mathbf{E}}{\partial a_{2n+1}} = 2\sum_{j=0}^{m}\left[(y_j - d_j) * e^{-\frac{\sum_{i=1}^{n}\left(a_i * x_{ij} - a_{n+i}\right)^2}{\left(1+a_{2n+2}\right)^2}}\right]. \tag{8}$$

Remaining coefficients are in the exponential part of the transfer function. Therefore the second summand in the Equation 3 cannot be formulated directly. We have to rewrite the Equation 3 as

$$\frac{\partial \mathbf{E}}{\partial a_i} = \sum_{j=0}^{m}\left[\frac{\partial \mathbf{E}}{\partial y_j} * \frac{\partial y_j}{\partial \rho_j} * \frac{\partial \rho_j}{\partial a_i}\right], \tag{9}$$

where ρ_j is the exponent of the transfer function 7. Now we can formulate partial derivatives of remaining coefficients as

$$\frac{\partial \mathbf{E}}{\partial a_{2n+2}} = 2\sum_{j=0}^{m}\left[(y_j - d_j) * (1+a_{2n+1}) e^{\rho_j} * 2\frac{\sum_{i=1}^{n}\left(a_i * x_{ij} - a_{n+i}\right)^2}{\left(1+a_{2n+2}\right)^3}\right] \tag{10}$$

$$\frac{\partial \mathbf{E}}{\partial a_i} = 2\sum_{j=0}^{m}\left[(y_j - d_j) * (1+a_{2n+1}) e^{\rho_j} * -2\frac{a_i^2 * x_{ij}^2 - a_{n+i} * x_{ij}}{\left(1+a_{2n+2}\right)^2}\right] \tag{11}$$

$$\frac{\partial \mathbf{E}}{\partial a_{n+i}} = 2\sum_{j=0}^{m}\left[(y_j - d_j) * (1+a_{2n+1}) e^{\rho_j} * -2\frac{a_{n+i} - a_i * x_{ij}}{\left(1+a_{2n+2}\right)^2}\right]. \tag{12}$$

We derived the gradient of error on the training data for the Gaussian transfer function neuron. An optimization method often requires these partial derivatives' every

iteration to adjust parameters in the proper direction. This mechanism (as described in Fig. 6.12b) can significantly save the number of error evaluations needed (see Fig. 6.13).

6.2.3.2 The experiment: analytic gradient saves error function evaluations

We performed an experiment to evaluate the effect of analytic gradient computation.

The Quasi-Newton optimization method was used to optimize the SigmNeuron neuron (a logistic transfer function). In the first run the analytic gradient was provided and in the second run, the gradient was not provided so the QN method was forced estimate the gradient itself. We measured the number of function evaluation calls and for the first run we recorded also the number of gradient computation requests.

Table 6.2 Number of evaluations saved by supplying gradient depending on the complexity of the energy function.

Complexity energy fnc.	Avg. evaluations without grad.	Avg. evals. with grad.	Avg. gradient calls	Evaluations saved	Computation time saved
1	45.825	20.075	13.15	56.19%	13.15%
2	92.4	29.55	21.5	68.02%	33.12%
3	155.225	44.85	34.875	71.11%	37.41%
4	273.225	62.75	51.525	77.03%	48.75%
5	493.15	79.775	68.9	83.82%	62.87%

The results are displayed in the Fig. 6.13 and in the Table 6.2. In the second run, without the analytic gradient provided, the number of error function evaluation calls increased exponentially with rising complexity of the error function. For the first run, when the analytic gradient is provided, number of error function evaluation calls increases just linearly and the number of gradient computations grows also linearly. The computation of gradient is almost equally time-consuming as the error function evaluation. When we sum up these two numbers for the first run, we still get growth increasing linearly with the number of layer (increasing complexity of the error surface). This is perfect result, because some models of complex problems can have 20 layers, the computational time saved by providing the analytic gradient is huge. Unfortunately some optimization methods such as genetic algorithms and swarm methods are not designed to use the analytic gradient of the error surface. On the other hand, for some data sets, the usage of analytic gradient can worsen convergence characteristic of optimization methods (getting stuck in local minima).

The training algorithm described in this Section enables possibility of efficient training of hybrid neural networks. The only problem that remains is to select appropriate optimization method.

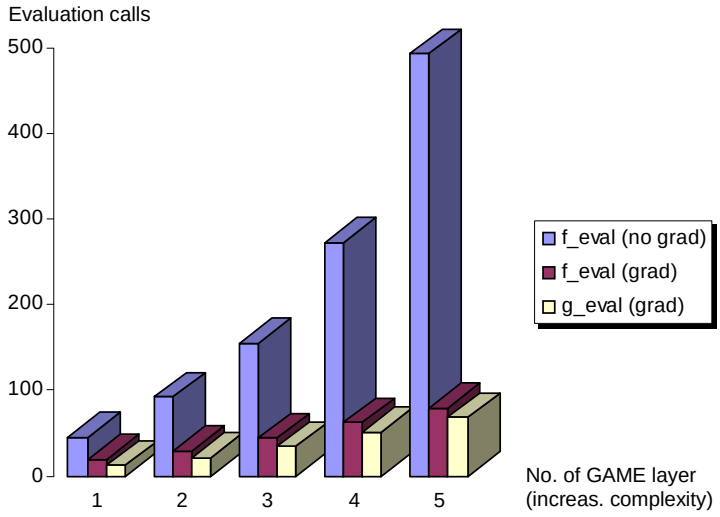

Fig. 6.13 When the gradient have to be estimated by the optimization method, number of function evaluation calls grows exponentially with an increasing complexity of the problem. When the analytic gradient is computed, the growth is almost linear.

6.2.4 Optimization methods (setting up coefficients)

The question ”Which optimization method is the best for our problem?” has not a simple answer. There is no method superior to others for all possible optimization problems. However there are popular methods performing well on whole range of problems.

Among these popular methods, we can include gradient methods - the Quasi Newton method, the Conjugate Gradient method and the Levenberg-Marquardt method. They use an analytical gradient (or its estimation) of the problem error surface. The gradient brings them faster convergence, but in cases when the error surface is jaggy, they are likely to get stuck in local optima.

Other popular optimization methods are genetic algorithms. They search the error surface by jumping on it with several individuals. Such search is usually slower, but more prone to get stuck in local minima. The Differential Evolution (DE) perform genetic search with an improved crossover scheme.

The search performed by swarm methods can be imagined as a swarm of birds flying over the error surface, looking for food in deep valleys. You can also imagine that for certain types of terrain, they might miss the deepest valley. Typical examples of swarm methods are Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) that mimics the behavior of real ants and their communication using pheromone.

Optimization methods with different behavior are often combined in one algorithm such as Hybrid of the Genetic Algorithm and the Particle Swarm Optimization (HGAPSO).

In our case, we use optimization methods to adjust coefficients of neurons – building blocks of inductive model. The inductive model is created from particular data set. The character of the data set influences which transfer functions will be used and also the complexity of error surface. The surface of a model's RMS error depends heavily on the data set, transfer functions of optimized neuron and also on preceding neurons in the model. The problem is to decide which optimization method should be used to minimize the error. Each data set has different complexity. Therefore we might expect there is no universal optimization method performing optimally on all data sets. We decided to implement several different methods and test their performance on data sets of various complexities.

Below, you can find short description of optimization methods used in GAME algorithm.

6.2.4.1 Optimization methods used in GAME

Optimization methods we have so far implemented to the GAME engine are summarized in the Table6.3.

There are three different classes of optimization methods named after type of search, they utilize – gradient, genetic and swarm. We will shortly describe particular algorithms we experiment with.

Table 6.3 Optimization methods summary

Abbrevation	Search	Optimization method
QN	Gradient	Quasi-Newton method
CG	Gradient	Conjugate Gradient method
PCG	Gradient	Powell CG method
PalDE	Genetic	Differtial Evolution ver. 1
DE	Genetic	Differtial Evolution ver. 2
SADE	Genetic	SADE genetic method
PSO	Swarm	Particle Swarm Optimization
CACO	Swarm	Cont. Ant Colony Opt.
ACO*	Swarm	Ext. Ant Colony Opt.
DACO	Swarm	Direct ACO
AACA	Swarm	Adaptive Ant Colony Opt.
API	Swarm	ACO with API heur.
HGAPSO	Hybrid	Hybrid of GA and PSO
SOS	Other	Stoch. Orthogonal Search
OS	Other	Orthogonal Search

Gradient based methods

The most popular optimization method of nonlinear programming is the Quasi-Newton method (QN) [39]. It computes search directions using gradients of an energy surface. To reduce their computational complexity, second derivatives (Hessian matrix) are not computed directly, but estimated iteratively using so called updates [38].

The Conjugate gradient method (CG) [51], a non-linear iterative method, is based on the idea that the convergence can be improved by considering also all previous search directions, not only the actual one. Several variants of the direction update are available (Fletcher-Reeves, Polak-Ribiere, Beale-Sorenson, Hestenes-Stiefel) and bounds are respected. Restarting (previous search direction are forgotten) often improves properties of CG method [42].

Genetic search

Genetic Algorithms (GA) [15] are inspired by Darwin's theory of evolution. Population of individuals are evolved according simple rules of evolution. Each individual has a *fitness* that is computed from its genetic information. Individuals are crossed and mutated by genetic operators and the most fit individuals are selected to survive. After several generations the mean fitness of individuals is maximized.

Niching methods [31] extend genetic algorithms to domains that require location of multiple solutions. They promote the formation and maintenance of stable subpopulations in genetic algorithms (GAs). The GAME engine uses the *Deterministic Crowding* (DC) [30] niching method to evolve structure of models. There exist several other niching strategies such as fitness sharing, islands, restrictive competition, semantic niching, etc.

The Differential Evolution (DE) [47] is a genetic algorithm with special crossover scheme. It adds the weighted difference between two individuals to a third individual. For each individual in the population, an offspring is created using the weighted difference of parent solutions. The offspring replaces the parent in case it is fitter. Otherwise, the parent survives and is copied to the next generation. The pseudocode, how offsprings are created, can be found e.g. in [50].

The Simplified Atavistic Differential Evolution (SADE) algorithm [16] is a genetic algorithm improved by one crossover operator taken from differential evolution. It also prevents premature convergence by using so called radiation fields. These fields have increased probability of mutation and they are placed to local minima of the energy function. When individuals reach a radiation field, they are very likely to be strongly mutated. At the same time, the diameter of the radiation field is decreased. The global minimum of the energy is found when the diameter of some radiation field descend to zero.

Swarm methods

The Particle Swarm Optimization method (PSO) use a swarm of particles to locate the optimum. According to [19] particles ”communicate” information they find about each other by updating their velocities in terms of local and global bests; when a new best is found, the particles will change their positions accordingly so that the new information is ”broadcast” to the swarm. The particles are always drawn back both to their own personal best positions and also to the best position of the entire swarm. They also have stochastic exploration capability via the use of the random constants.

The Ant colony optimization (ACO) algorithm is primary used for discrete problems (e.g. Traveling Salesman Problem, packet routing). However many modifications of the original algorithm for continuous problems have been introduced recently [48]. These algorithms mimic the behavior of real ants and their communication using pheromone. We have so far implemented the following ACO based algorithms:

The Continuous Ant colony optimization (CACO) was proposed in [8] and it works as follows. There is an ant nest in a center of a search space. Ants exits the nest in a direction given by quantity of pheromone. When an ant reaches the position of the best ant in the direction, it moves randomly (the step is limited by decreasing diameter of search. If the ant find better solution, it increases the quantity of pheromone in the direction of search [28].

The Ant Colony Optimization for Continuous Spaces (ACO*) [7] was designed for the training of feed forward neural networks. Each ant represents a point in the search space. The position of new ants is computed from the distribution of existing ants in the state space.

Direct Ant Colony Optimization (DACO) [23] uses two types of pheromone - one for mean values and one for standard deviation. These values are used by ants to create new solutions and are updated in the ACO way.

The Adaptive Ant Colony Algorithm (AACA) [29] encodes solutions into binary strings. Ants travel from least significant bit to the most significant bit and back. After finishing the trip, the binary string is converted to the solution candidate. The probability of change decreases with significance of bit position by boosting pheromone deposits.

The API algorithm [33] is named after Pachycondyla apicalis and it simulates the foraging behavior of these ants. Ants move from nest to its neighborhood and randomly explore the terrain close to their hunting sites. If an improvement occurs, next search leads to the same hunting site. If the hunt is unsuccessful for more than p times for one hunting site, the hunting site is forgotten and ant randomly generates a new one.

Hybrid search

The Hybrid of the GA and the PSO (HGAPSO) algorithm was proposed in [19]. PSO works based on social adaptation of knowledge, and all individuals are considered to be of the same generation. On the contrary, GA works based on evolution from generation to generation, so the changes of individuals in a single generation are not considered. In nature, individuals will grow up and become more suitable to the environment before producing offspring. To incorporate this phenomenon into GA, PSO is adopted to enhance the top-ranking individuals on each generation.

Other methods

The Orthogonal Search (OS) optimizes multivariate problem by selecting one dimension at a time, minimizing the error at each step. The OS can be used [6] to train single layered neural networks.

We use minimization of a real-valued function of several variables without using gradient, optimizing variables one by one. The Stochastic Orthogonal Search (SOS) differs from OS just by random selection of variables.

6.2.4.2 Benchmark of optimization methods

Following experiments are to demonstrate performance of individual methods applied to optimize models of several real world data sets.

For each data set, we generated models, where neurons with simple transfer functions (Linear, Sigm, Combi, Polynomial, Exp) were enabled. Coefficients of these neurons were optimized by a single method from the Table 6.3. The configuration **all**, will be explained later. Because these experiments were computationally expensive (optimization methods not utilizing the analytic gradient need many more iterations to converge), we built the ensemble of 5 models for each configuration and data set.

Results on Boston data set from UCI repository are summarized in the Fig.6.14. For all optimization methods the difference between their error on training and testing data set was almost the same. It signifies that this data set is not very noisy so the overfitting did not occur. The best performance showed the Conjugate Gradient method, but all methods except the worst performing one (Orthogonal Search) achieved similar results. Both training and testing errors of models optimized by OS were significantly higher.

The results on the Building data set for it's three output variables are shown in the Fig.6.15. There is no significant difference between results for the noisy variable (Energy consumption) and the other two. We can divide optimization methods into the good and bad performing classes. Good performers are Conjugate Gradient, Quasi Newton, SADE genetic algorithm, Differential Evolution, and the all configuration standing for all methods participation in models evolution. On the other hand

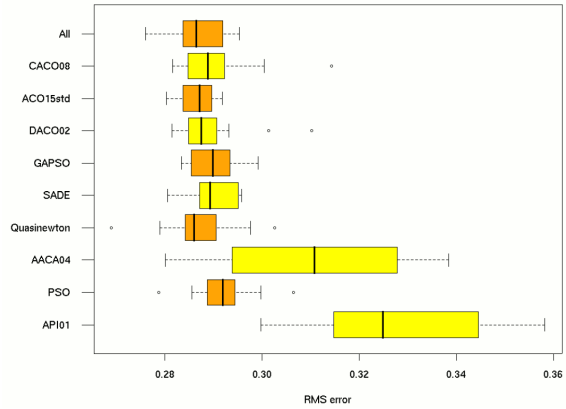

Fig. 6.14 The RMS error of models on the Boston data set. Neurons of models were optimized by individual methods.

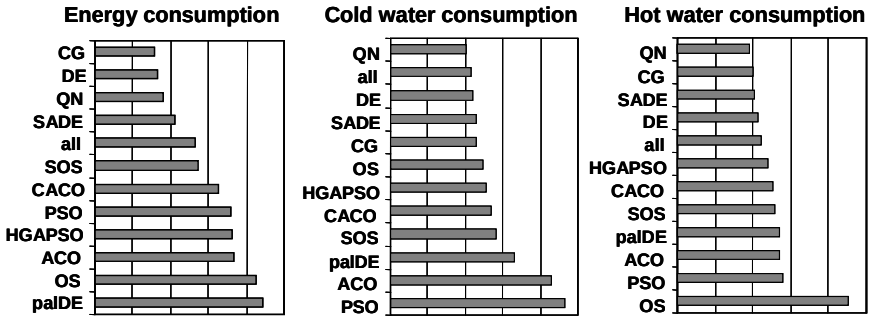

Fig. 6.15 The performance comparison of optimization methods on the Building data set. The size of bars for individual methods is proportional to the average testing RMS error of models generated using these methods on the Building data set. Models were generated individually for each output variable.

badly performing optimization methods for the Building data set are Particle Swarm Optimization, PAL- Differential Evolution[2] and the Ant Colony Optimization.

In accordance with results published in [50], our version of differential evolution outperformed swarm optimization methods. On the other hands, experiment with the Spiral data (telling apart two intertwined spirals) showed different results. Fig. 6.16 shows that Ant Colony based methods trained better models than methods based on Differential Evolution or gradient based methods. Spiral data set is very hard classification problem and it is difficult to solve it. Error surface has plenty local

[2] The palDE is the second version of the Differential Evolution algorithm implemented in the GAME engine. The result when the first version of DE performed well and the second version badly is pellicular. It signifies that the implementation and the proper configuration of a method is of crucial importance.

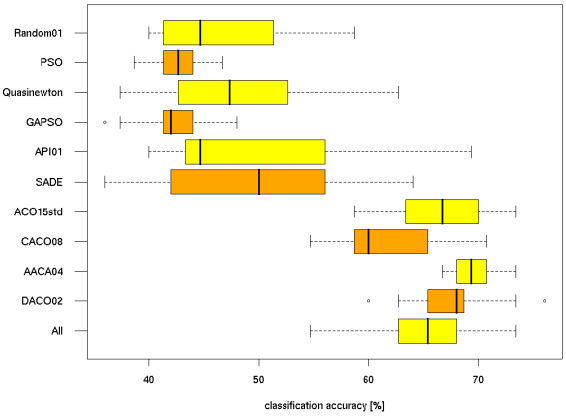

Fig. 6.16 The classification accuracy of models optimized by individual methods on two intertwined spirals problem.

optima and ant colony methods were able to locate diverse solutions of high quality. Combining them provided increased accuracy.

Conclusion of our experiment with several different data sets is in accordance with our expectations. There is no universal optimization method applicable to arbitrary problem (data set). This statement is also one of the consequences of so called ”No Free Lunch Theorem” [14].

6.2.5 Combining optimization methods

We assumed that for each data set, some optimization methods are more efficient than others. If we select appropriate method to optimize coefficients of each neuron within single GAME network, the accuracy will increase. The problem is to find out which method is appropriate (and most effective).

In the ”**all**” configuration, we used simple strategy. When new neuron was initialised, random method was assigned to optimize the coefficients of neurons. In case the optimization method was inappropriate, coefficients were not set optimally and neuron did not survived in the genetic algorithm evolving neurons in the layer of the GAME model. Only appropriate optimization methods were able to generate fittest neurons.

6.2.5.1 Evolution of optimization methods

The question is if it is better to assign optimization method randomly or inherit it from parent neurons.

The type of optimization method can be easily inherited from parents, because neurons are evolved by means of niching genetic algorithm. This genetic algorithm can also assign appropriate optimization methods to neurons being evolved. We added the type of the optimization into the chromosome (see Fig. 6.18). When new neurons are generated by crossover to the next generation, they also inherit type of optimization from their parent neurons. The result should be that methods, training successful neurons, ale selected more often than methods, training poor performers on a particular data set.

Again, an experiment was designed to prove this assumption. We prepared configurations of the GAME engine with several different inheritance settings. In the configuration $p0\%$ new neurons inherit their optimization method from their paren neurons. In the configuration $p50\%$ offsprings have 50% chance to get random method assigned. In the configuration $p100\%$ nothing is inherited, all optimization methods are set randomly.

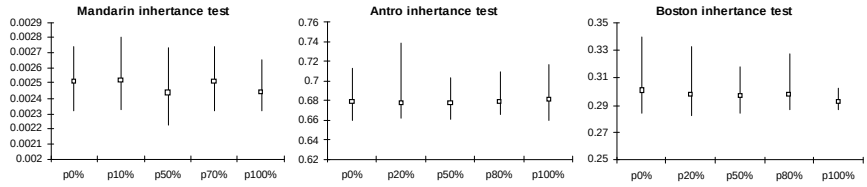

Fig. 6.17 The experiments with the inheritance of transfer function and learning method. For all three data sets, the fifty percent inheritance level is a reasonable choice.

We have been experimenting with the Mandarin, Antro and Boston data sets. The description of these data sets can be found in [25].

For each configuration 30 models were evolved. The maximum, minimum and mean of their RMS errors for each configuration are displayed in the Fig. 6.17. Results are very similar for all configurations and data sets. There is no configuration significantly better than others. For all data sets we can observe that the $p50\%$ and the $p100\%$ configuration have slightly better mean error values and lower dispersion of errors. We chose the $p50\%$ configuration to be default in the GAME engine. It means offspring neurons have 50% chance to get random optimization method assigned otherwise their methods are inherited from parent neurons.

The result of this approach is that methods, training successful neurons, ale used more often than methods, training poor performers on a particular data set. Also, advantages of several different optimization approaches are combined within single model making possible to obtain diverse solutions of high quality.

Again, we demonstrate the advantage of our approach by experimental results.

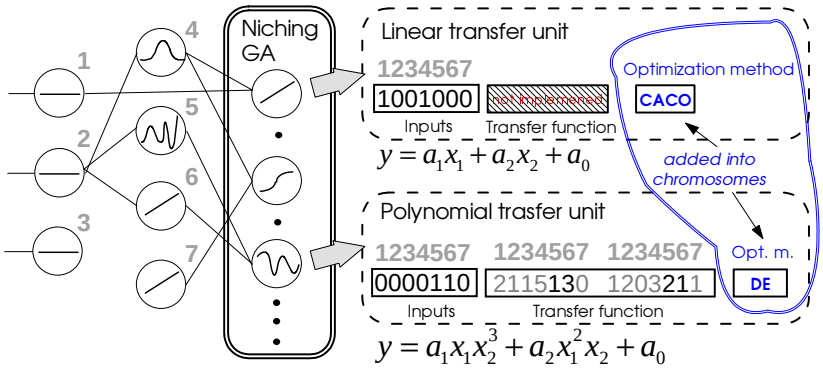

Fig. 6.18 The example of chromosomes for GAME neurons with linear and polynomial transfer function. Chromosomes contain encoded input connections and for some neurons, the structure of the transfer function is also encoded to be able to evolve it. The type of the optimization method was appended to the chromosome.

6.2.5.2 Combined optimization outperforms individual methods

We measured the performance of individual methods and the combined optimization (all) on several data sets.

We used the same methodology as for previous experiments on Building data set, Boston and Spiral data sets. Along with these data sets, we used diverse real world data sets described in [25].

Optimization methods are ranked according to the accuracy of their models on several data sets. Fig. 6.19 displays the results.

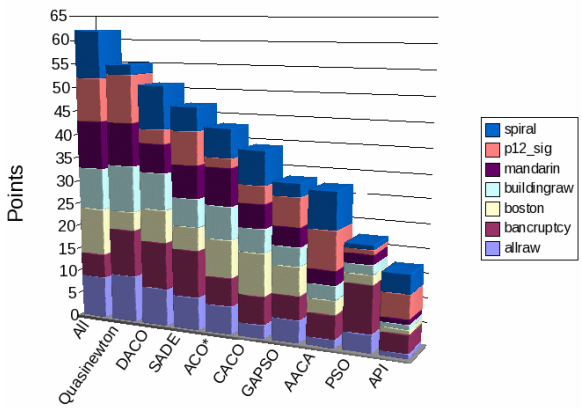

Fig. 6.19 Final comparison of all tested methods. Points are derived from the ranking for each data test - better position means more points.

Final ranking shows, that the Quasi-Newton optimization method was the most successful from individual methods. It was also the fastest. Combined optimization (**All**) clearly outperformed all individual methods, but it was much slower than Quasi-Newton method. The reason is that computing time was wasted by inefficient methods that do not use analytic gradient of the error surface (such as PSO). Possible solution is to exclude the least efficient methods (accuracy will decrease just marginally), or to enhance these methods by hybridizing them with gradient based methods.

The experiments in this section showed that gradient methods like Quasi Newton and Conjugate Gradients performed very well for all data sets we have been experimenting with. When **All** methods are used, superb performance is guarantied, but the computation is significantly slower (some methods need many iterations to converge). At this stage of the research and implementation, we recommend using the Quasi Newton (QN) optimization method only, because it is the fastest and very reliable. If the computing time is not important for you, the evolution of optimization methods is the best choice.

Another modification of the MIA GMDH algorithm is the topology of models produced.

6.2.6 Structural innovations

As stated in the introductional part of this chapter, the topology of the original MIA GMDH was adapted to computational capabilities of early seventies. Experiments that can be nowadays run on every personal computer were intractable even on the most advanced supercomputer.

To make the computation of an inductive model possible, several restrictions on the structure of the model had to be imposed. Because of growing computational power and the development of heuristic methods capable of approximative solutions np-hard problems, we can leave out some of these restrictions.

The most restrictive rule of the original MIA GMDH is the fixed number of neurons' inputs (two) and a polynomial transfer function that is constant (except coefficients) for all neurons in a model.

The next restriction is the absence of layer breakthrough connections. In the original version inputs to a neuron can be from previous layer only.

6.2.6.1 Growth from a minimal form

The GAME models grow from a minimal form. There is a strong parallel with state of the art neural networks as the NEAT [44]. In the default configuration of the GAME engine, neurons are restricted to have at least one input and the maximum number of inputs must not exceed the number of the hidden layer, the neuron belongs to.

The number of inputs to the neuron increases together with the depth of the neuron in the model. Transfer functions of neurons reflect growing number of inputs. We showed [27] that increasing number of neuron's inputs and allowing interlayer connections plays significant role in improving accuracy of inductive models. The growing limit of inputs a neuron is allowed to have is crucial for inductive networks. It helps to overcome the curse of dimensionality. According to the induction principle it is easier to decompose problem to one-dimensional interpolation problems and then combine solutions in two and more dimensions, than to start with multi-dimensional problems (for full connected networks - dimensionality is proportional to the number of input features).

To improve the modeling accuracy of neural networks, artificial input features are often added to the training data set. These features are synthesized from original features by math operations and can possibly reveal more about the modelled output. This is exactly what is GAME doing automatically (neurons of the first hidden layer serve as additional synthesized features for the neurons deeper in the network).

Our additional experiments showed that the restriction on the maximum number of inputs to neurons has moderately negative effect on the accuracy of models. However when the restriction is enabled[3], the process of model generation is much faster. The accuracy of produced models is also more stable than without the restriction. Without the restriction we would need many more epochs of the genetic algorithm evolving neurons in a layer (models accuracy would be stable and the feature ranking algorithm deriving significance from proportional numbers of neurons connected to a particular feature would work properly (feature ranking algorithm will be described later in this chapter).

6.2.6.2 Interlayer connections

Neurons have no longer inputs just from previous layer. Inputs can be connected to the output of any neuron from previous layers as well as to any input feature. This modification greatly increases the state space of possible model topologies, but the improvement in accuracy of models is rather high [27].

The GMDH algorithm implemented in the KnowledgeMiner software [34] can also generate models with layer breakthrough connections.

Expanded search space of possible model topologies requires methods of efficient heuristic search.

6.2.7 Genetic algorithm

The genetic algorithm is frequently used to optimize a topology of neural networks [32, 41, 45]. Also in GMDH related research, recent papers [36, 35] report improv-

[3] No restriction on the maximal number of inputs does not mean a fully connected network!

ing the accuracy of models by employing genetic search to identify their optimal structure.

In the GAME engine, we also use genetic search to optimize the topology of models and also the configuration and shapes of transfer functions within their neurons. The model is constructed layer by layer as in the MIA GMDH. In each layer, genetic is executed and after several generations, most fit and diverse neurons are selected to form the layer. After that, the construction process continues with next layers. Neurons from these layers can be connected to input variables and also to neurons from preceding layers.

If the optimal structure of model is to be identified, we need to find optimal interconnection of neurons, types and structure of their transfer functions and size of the model (number of neurons in each layer and number of layers). Connections of neurons and structure of their transfer functions can be optimized by genetic algorithm. The example of encoding into chromosomes are depicted in the Fig. 6.20.

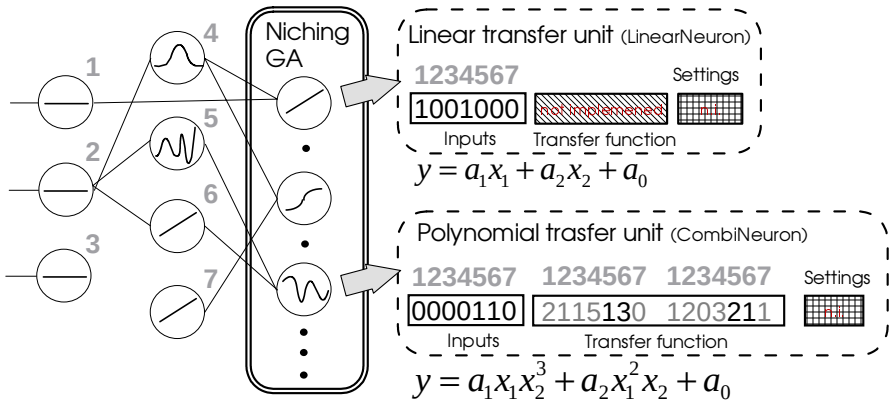

Fig. 6.20 Neurons are encoded into chromosomes and evolution identifies optimal inputs of neurons and structure of their transfer functions.

The individual in the genetic algorithm represents one particular neuron of the GAME network. Inputs of a neuron are encoded into a binary string chromosome. The structure of transfer function can be also added into the chromosome. The chromosome can also include type of the optimization method and configuration options such as stopping criteria, strategies utilized during optimization of parameters, etc.

The length of the ”Inputs” part of the chromosome equals to the number of input variables plus number of neurons from previous layers, the neuron can be connected to. The existing connection is represented by ”1” in the corresponding gene. The number of ones is restricted to maximal number of neuron’s inputs. The example how the transfer function can be encoded is in the Fig. 6.20.

If two neurons of different types are crossed, just the ”Inputs” part of the chromosome comes into play. If two Polynomial neurons cross over, also the second part encoding transfer function is involved.

Note that coefficients of the transfer functions (a_0,a_1,...,a_n) are not encoded in the chromosome Fig. 6.20. These coefficients are adjusted separately by optimization methods as described in previous section. This is crucial difference from the Topology and Weight Evolving Artificial Neural Network (TWEANN) approach [45].

The fitness of the individual is inversely proportional to the error of the individual computed on the validation data set.

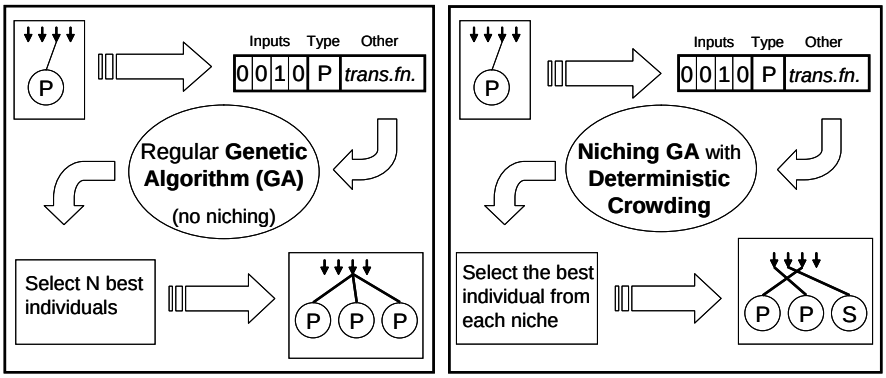

Fig. 6.21 GAME neurons in the first layer are encoded into chromosomes, then GA is applied to evolve the best performing neurons. After few epochs all neurons will be connected to the most significant input and therefore correlated. When the Niching GA is used instead of the basic variant of GA, neurons connected to different inputs survive.

The application of the genetic algorithm in the GAME engine is depicted in the Fig. 6.21. The left schema describes the process of single GAME layer evolution when the standard genetic algorithm [15] is applied.

Neurons randomly initialized and encoded into chromosomes. Then the genetic algorithm is executed. After several epochs of the evolution, individuals with the highest fitness (neurons connected to the most significant input) dominate the population. The best solution represented by the best individual is found.

Whole population have very similar (or the same) chromosomes as the winning individual has. This is also the reason why all neurons surviving in the population (after several epochs of evolution by the regular genetic algorithm) are highly correlated.

The regular genetic algorithm found one best solution. We want to find also multiple suboptimal solutions (e.g. neurons connected to the second and the third most important input). By using less significant features we get more additional information than by using several best individuals connected to the most significant feature, which are in fact highly correlated (as shown on Fig. 6.22.). Therefore we employ a

niching method described below. It maintains diversity in the population and therefore neurons connected to less significant inputs are allowed to survive, too (see Fig. 6.21 right).

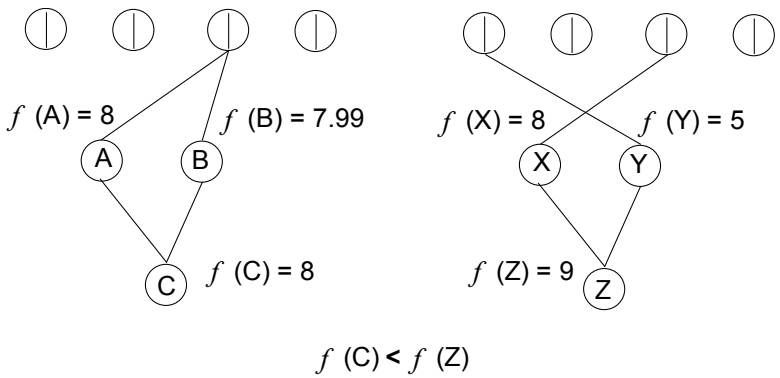

Fig. 6.22 Fitness of neuron Z is higher than that of neuron C, although Z has less fit inputs.

6.2.7.1 Experiments with Deterministic Crowding

The major difference between the regular genetic algorithm and a niching genetic algorithm is that in the niching GA the distance among individuals is defined.

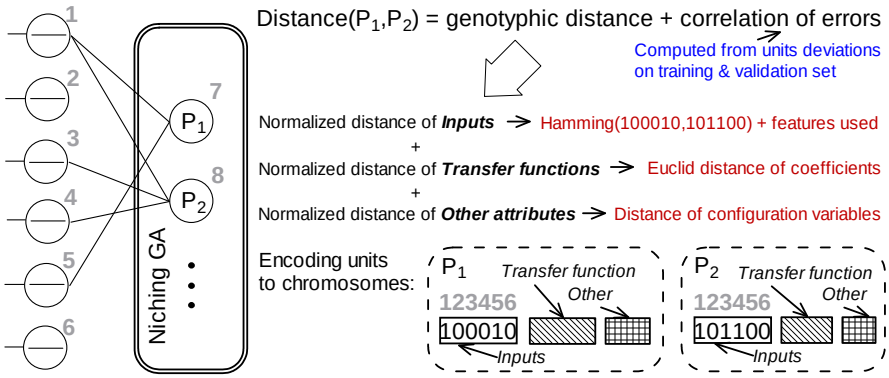

Fig. 6.23 The distance of two neurons in the GAME network.

The distance of two individuals from the pseudocode of Deterministic Crowding [30] can be based on the phenotypic or genotypic difference of neurons. In the GAME engine, the distance of neurons is computed from both differences. Fig. 6.23

shows that the distance of neurons is partly computed from the correlation of their errors and partly from their genotypic difference. The genotypic difference consists the obligatory part ”difference in inputs”, then some neurons add ”difference in transfer functions” and also ”difference in configurations” can be defined.

Neurons that survive in layers of GAME networks are chosen according to the following algorithm. After the niching genetic algorithm finished the evolution of neurons, a multi-objective algorithm sorts neurons according to their RMS error, genotypic distance and the correlation of errors. Surviving neurons have low RMS errors, high mutual distances and low correlations of errors.

Niches in GAME are formed by neurons with similar inputs, similar transfer functions, similar configurations and high correlation of errors.

The next idea is that neurons should inherit their type and the optimization method used to estimate their coefficients. This improvement allows reducing time wasted with optimizing neurons with an improper transfer function by optimization methods not suitable to processed data.

Evaluation of the distance computation

The GAME engine enables the visual inspection of complex processes that are normally impossible to control. One of these processes is displayed in the Fig. 6.2.7.1. From left we can see the matrix of genotypic distances computed from chromosomes of individual neurons during the evolution of the GAME layer. Note that this distance is computed as a sum of three components: distance of inputs, transfer functions and configuration variables, where last two components are optional.

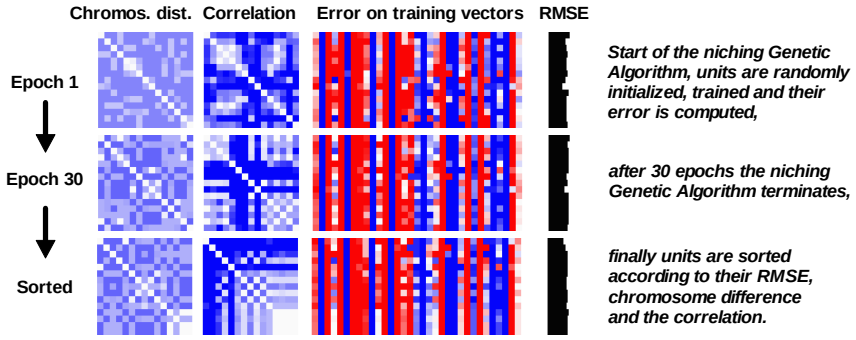

Fig. 6.24 During the GAME layer evolution, distances of neurons can be visually inspected. The first graph shows their distance based on the genotypic difference. The second graph derives distance from their correlation. Third graph shows deviations of neurons on individual training vectors and the most right graph displays their RMS error on the training data.

The darker color of background signifies the higher distance of corresponding individuals and vice versa. The next matrix visualizes distances of neurons based

on the correlation of their errors. Darker background signifies less correlated errors. The next graph shows deviations of neurons output from the target value of individual training vectors. From these distances the correlation is computed. The most right graph of the Fig. 6.2.7.1 shows a normalized RMS error of neurons on the training data.

All these graphs are updated as the evolution proceeds from epoch to epoch. When the niching genetic algorithm finishes, you can observe how neurons are sorted (multi objective sorting algorithm based on the Bubble sort algorithm) and which neurons are finally selected to survive in the layer. Using this visual inspection tool, we have evaluated and tuned the distance computation in the niching genetic algorithm.

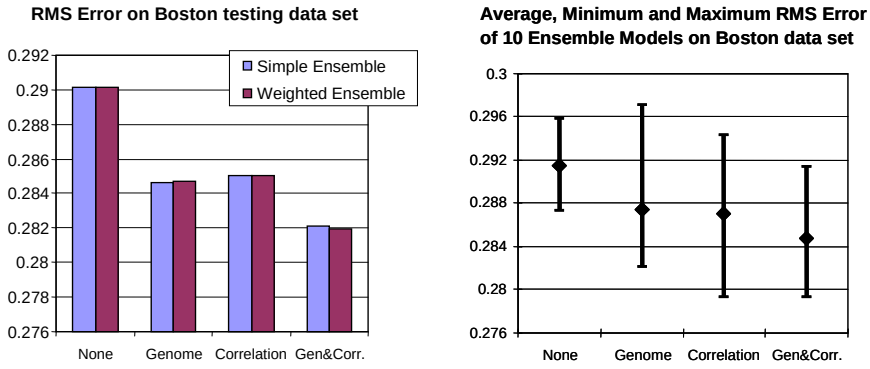

Fig. 6.25 The best results were obtained when the distance of neurons is computed as a combination of their genotypic distance and the correlation of their errors on training vectors.

The next goal was to evaluate if the distance computation is well defined. The results in the Fig. 6.2.7.1 show that the best performing models can be evolved with the proposed combination of genotypic difference and correlation as the distance measure. The worst results are achieved when the distance is set to zero for all neurons. Medium accuracy models are generated by either the genotypic difference based distance or the correlation of errors based distance.

In the Fig. 6.26, there is a comparison of the regular genetic algorithm and the niching GA with the Deterministic Crowding scheme. The data set used to model the output variable (Mandarin tree water consumption) has eleven input features. Neurons in the first hidden layer of the GAME network have a single input, so they are connected to a single feature.

The population of 200 neurons in the first layer was initialized randomly (genes are uniformly distributed - approx. the same number of neurons connected to each feature). After 250 epochs of the regular genetic algorithm the fittest individuals (neurons connected to the most significant feature) dominated the population. On

the other hand the niching GA with DC maintained diversity in the population. Individuals of three niches survived. As Fig. 6.26 shows, the functionality of niching genetic algorithm in the GAME engine is evident.

When you look at the Fig. 6.26 you can also observe that the number of individuals (neurons) in each niche is proportional to the significance of the feature, neurons are connected to. From each niche the fittest individual is selected and the construction goes on with the next layer. The fittest individuals in next layers of the GAME network are these connected to features which brings the maximum of additional information. Individuals connected to features that are significant, but highly correlated with features already used, will not survive. By monitoring which individuals endured in the population we can estimate the significance of each feature for the output variable modelling. This information can be subsequently used for the feature ranking.

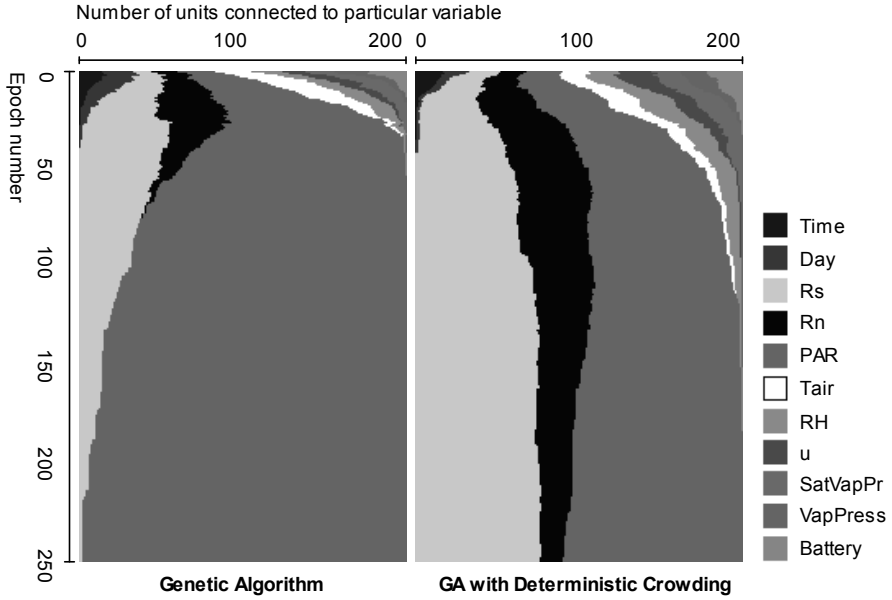

Fig. 6.26 The experiment demonstrated that the regular Genetic Algorithm approaches an optimum relatively quickly. Niching preserves different neurons for many more iterations so we can chose the best neuron from each niche at the end. Niching also increases a probability of the global minimum not being missed.

We also compared the performance (the inverse of RMS error on a testing data) of GAME models evolved by means of the regular GA and the niching GA with Deterministic Crowding respectively. Extensive experiments were executed on the complex data (Building dataset) and on the small simple data (On-ground nuclear tests dataset).

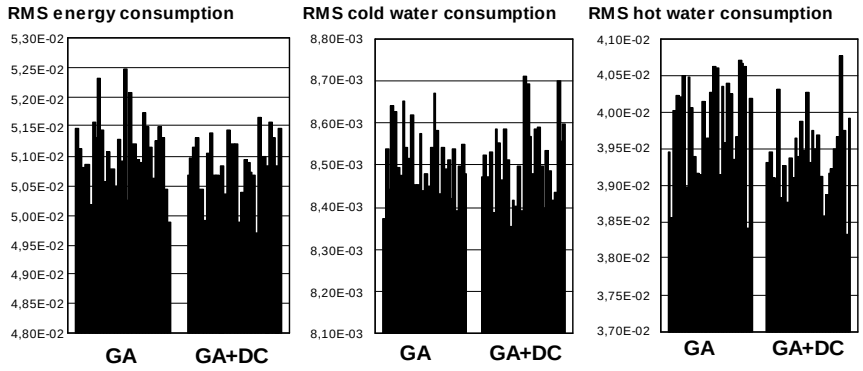

Fig. 6.27 RMS error of GAME models evolved by means of the regular GA and the GA with the Deterministic Crowding respectively (on the complex data). For the the hot water and the energy consumption, the GA with DC is significantly better than the regular GA

The statistical test proved that on the level of significance 95%, the GA with DC performs better than simple GA for the energy and hot water consumption. The Fig. 6.27 shows RMS errors of several models evolved by means of the regular GA and the GA with Deterministic Crowding respectively.

The results are more significant for the On-ground nuclear dataset. The Fig. 6.28 shows the average RMS error of 20 models evolved for each output attribute. Leaving out models of the fire radius attribute, the performance of all other models is significantly better with Deterministic Crowding enabled.

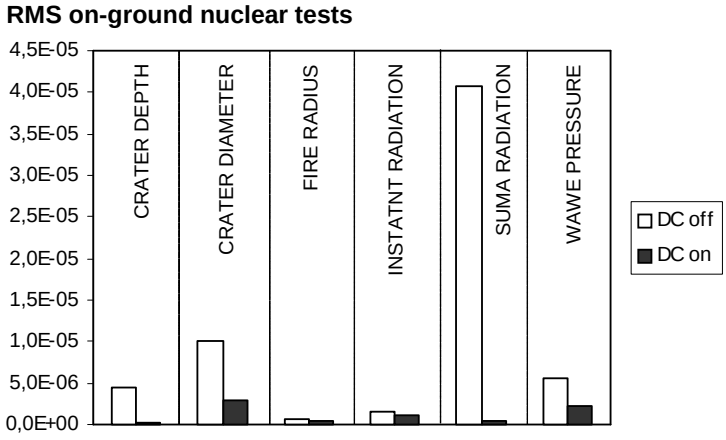

Fig. 6.28 Average RMS error of GAME models evolved by means of simple GA (DC off) and GA with Deterministic Crowding (DC on) respectively (on the simple data). Here for all variables, the Deterministic Crowding attained the superior performance.

We can conclude, than niching strategies significantly improved the evolution of GAME models. Generated models are more accurate than models evolved by the regular GA as showed our experiments with real world data.

6.2.8 Ensemble techniques in GAME

The GAME method generates on the training data set models of similar accuracy. They are built and validated on random subsets of the training set (this technique is known as bagging [17]). Models have also similar types of neurons and similar complexity. It is difficult to choose the best model - several models have the same (or very similar) performance on the testing data set. We do not choose one best model, but several optimal models - ensemble models [9].

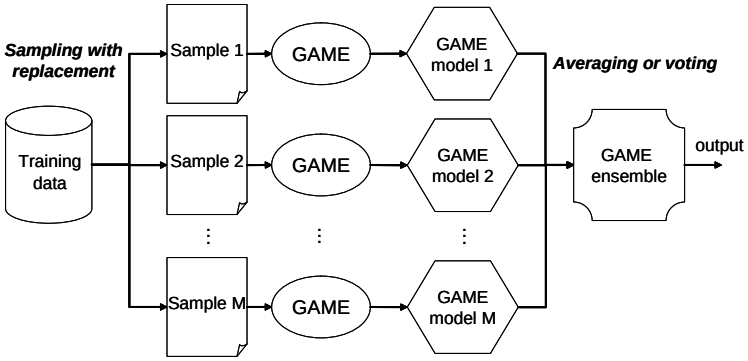

Fig. 6.29 The Bagging approach is used to build an ensemble of GAME models, models are then combined by the Simple or Weighted averaging.

The Fig. 6.29 illustrates the principle how GAME ensemble models are generated using bootstrap samples of training data and later combined into a simple ensemble or a weighted ensemble. This technique is called Bagging and it helps that member models demonstrate diverse errors on a testing data.

Other techniques that promote diversity in the ensemble of models play significant role in increasing the accuracy of the ensemble output. The diversity in the ensemble of GAME models is supported by following techniques:

- Input data varies (Bagging)
- Input features vary (using subset of features)
- Initial parameters vary (random initialization of weights)
- Model architecture varies (heterogeneous neurons used)
- Training algorithm varies (several training methods used)
- Stochastic method used (niching GA used to evolve models)

We assumed that the ensemble of GAME models will be more accurate than any of individual models. This assumption appeared to be true just for GAME models whose construction was stopped before they reached the optimal complexity (Fig. 6.2.8 left).

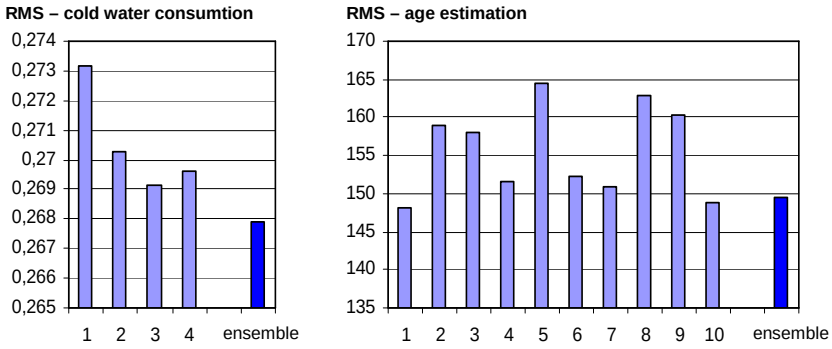

Fig. 6.30 The Root Mean Square error of the simple ensemble is significantly lower than RMS of individual suboptimal models on testing data (left graph). For optimal GAME models it is not the case (right).

We performed several experiments on both synthesized and real world data sets. These experiments demonstrated that ensemble of optimal GAME models is seldom significantly better then single the best performing model from the ensemble Fig. 6.2.8 right).

The problem is, we cannot say in advance which single model will perform the best on testing data. The best performing model on training data can be the worst performing one on testing data and vice versa.

Usually, models badly performing on training data perform badly also on testing data. Such models can impair the accuracy of ensemble model. To limit the influence of bad models on the output of ensemble, models can be weighted according to their performance on training data set. Such ensemble is called the weighted ensemble and we discuss its performance below.

Contrary to the approach introduced in [12], we do not use the whole data set to determine performances (Root Mean Square Errors) of individual models in the weighted ensemble.

In Fig. 6.31 you can see that weighted ensemble has tendency to overfit the data - stronger than simple ensemble. While its performance is superior on the training and validation data, on the testing data there are several individual models performing better.

The theoretical explanation for such behavior might be the following. Fig. 6.2.8 a shows ensemble of two models that are not complex enough to reflect the variance of data (weak learner). The error of the ensemble is lower than that of individual models, similarly like in the first experiment mentioned above.

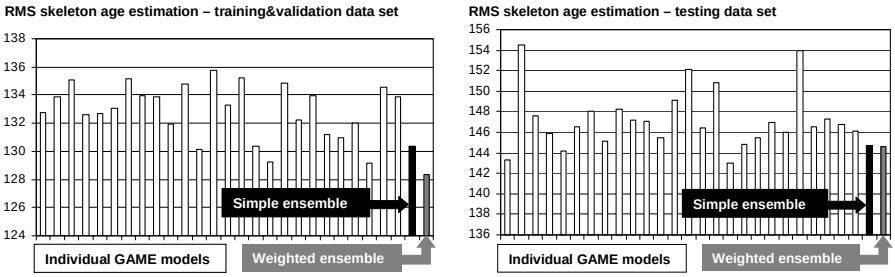

Fig. 6.31 Performance of the simple ensemble and weighted ensemble on very noisy data set (Skeleton age estimation based on senescence indicators).

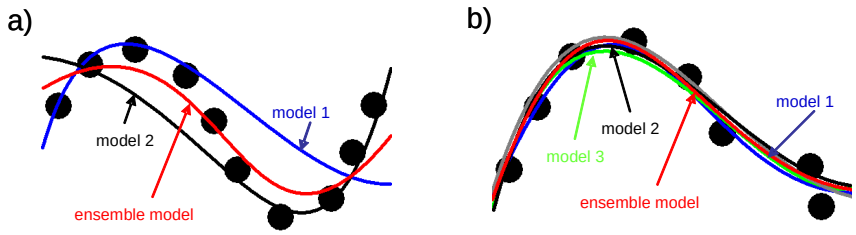

Fig. 6.32 Ensemble of two models exhibiting diverse errors can provide significantly better result.

In Fig. 6.2.8 b, there is an ensemble of three models having the optimal complexity. It is apparent that the accuracy of the ensemble cannot be significantly better, than those of individual models. The negative result of second experiment is therefore caused by the fact, that the bias of optimal models cannot be further reduced.

We can conclude that by using the simple ensemble, instead of single GAME model, we can in some cases improve the accuracy of modeling. The accuracy improvement is not only advantage of using ensembles. There is highly interesting information encoded in the ensemble behavior. It is the information about the credibility of member models.

These models approximate data similarly and their behavior differ outside of areas where system can be successfully modelled (insufficient data vectors present, etc.). In well defined areas all models have compromise response. We use this fact for models' quality evaluation purposes and for estimation of modeling plausibility in particular areas of the input space.

6.3 Benchmarking the GAME method

In this section we benchmark the regression and classification performance of the GAME method against the performance of methods implemented in the Weka machine learning environment.

We performed experiments on the A-EGM data set, described in [26]. At first, we studied the regression performance of GAME models produced by different configurations of the GAME algorithm. The target variable was the average A-EGM signal ranking by three experts (the A-EGM-regression data set). We found out, and it is also apparent in the boxplot charts, that comparison of the 10-fold cross validation error is not stable enough to decide, which configuration is better. Therefore we repeated the 10-fold cross validation ten times, each time with different fold splitting. For each box plot it was necessary to generate and validate one hundred models.

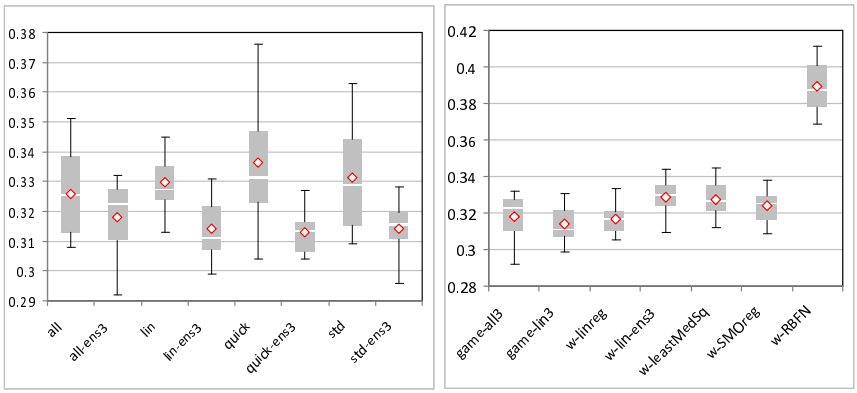

Fig. 6.33 The comparison of RMS cross validation errors for several configuration configuration of the GAME engine(left). Selected GAME models compared with models generated in Weka environment(right).

For all experiments we used three default configurations of the GAME algorithm available in FAKE GAME environment [5]. The *std* configuration uses just subset of neurons (those with implemented analytic gradient for faster optimization). It evolves 15 neurons for 30 epochs in each layer. The *quick* configuration is the same as *std* except that it do not use the niching genetic algorithm (just 15 neurons in the initial population). The *linear* restricts type of neurons that can be used to linear transfer function neurons. The *all* configuration is the same as *std*, in addition it uses all neurons available in the FAKE GAME environment. This configuration is more computationally expensive, because it also optimizes complex neurons such as BPNetwork containing standard MLP neural network with the back-propagation of error [32].

The GAME algorithm also allows to generate ensemble of models [13, 9]. Ensemble configurations contain digit (number of models) in their name.

Fig. 6.33 shows that the regression of the AER output is not too difficult task. All basic GAME configurations performed similarly (left chart) and ensembling of three models further improved their accuracy. The ensemble of three linear models performed best in average, but the difference from $all - ens3$ configuration is not significant.

In Weka data mining environment, *LinearRegression* with embedded feature selection algorithm was the best performing algorithm. Ensembling (bagging) did not improved results of generated model, quite the contrary. The Radial Basis Function Network (RBFN) failed to deliver satisfactory results in spite of experiments with its optimal setting (number of clusters).

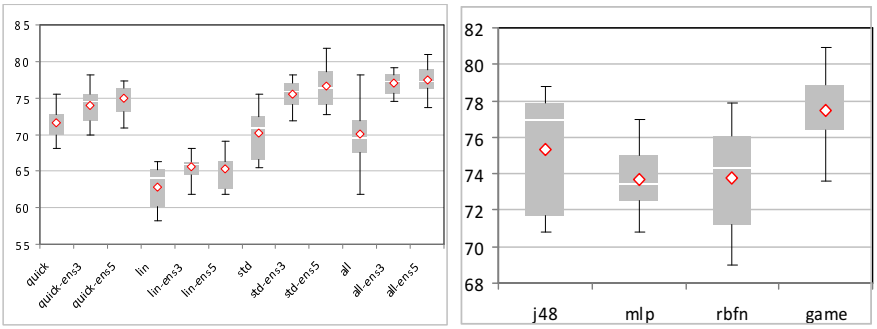

Fig. 6.34 Classification accuracy in percent for several GAME configurations (left) and comparison with Weka classifiers (right).

Secondly, our experiments were performed on the A-EGM-classification data set. The methodology remained the same as for regression data. Additionally we tested classification performance of 5 models ensembles. Fig. 6.34 left shows that the classes are not linearly separable - *linear* configuration generates poor classifiers and ensembling does not help. Combining models in case of all other configurations improve the accuracy. For *all* configuration the dispersion of cross validation errors is quite high. The problem is in the configuration of the genetic algorithm - with 15 individuals in the population some ”potentially useful” types of neurons do not have chance to be instantiated. Ensembling models generated by this configuration improves their accuracy significantly.

Comparison with Weka classifiers (Fig. 6.34 right) shows that GAME ensemble significantly outperforms Decision Trees (j48), MultiLayered Perceptron (mlp) and Radial Basis Function network (rbfn) implemented in Weka data mining environment.

The last experiment (Fig. 6.35) showed that the best split of the training and validation data set is 40%/60% (training data are used by optimization method to

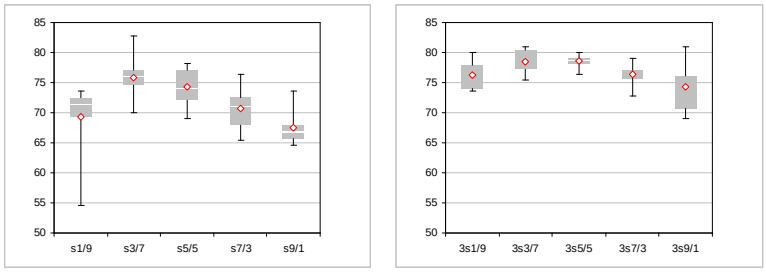

Fig. 6.35 Classification performance for different ratios of training/validation data split. Left - results for single game models generated by *std* configuration. Right - results for GAME ensemble ($std - ens3$).

adjust parameters of GAME neurons transfer functions, whereas from validation part, the fitness of neurons is computed).

Implicitly, and in all previous experiments, training and validation data set was divided 70%/30% in the GAME algorithm. Changing the implicit setting to 40%/60% however involves additional experiments on different data sets.

6.3.1 Summary of results

For this data set, the GAME algorithm outperforms well established methods in both classification and regression accuracy. What is even more important, both winning configurations were identical $all - ens$. Natural selection evolved optimal models for very different tasks - that is in accordance with our previous experiments and with our aim to develop automated data mining engine.

6.4 Case studies – data mining using GAME

6.4.1 Fetal Weight Prediction Formulae Extracted from GAME

An accurate model of ultrasound estimation of fetal weight (EFW) can help in decision if the cesarean childbirth is necessary. We collected models from various sources and compared their accuracy. These models were mostly obtained by standard techniques such as linear and nonlinear regression. The best performing model, from 14 we have been experimenting with, was the equation published by Hadlock et al:

$$\log_{10} EFW = 1.326 - 0.00326 \times AC \times FL + 0.0107 \times HC$$

$$+0.0438 \times AC + 0.158 \times FL \tag{13}$$

Alternatively, we generated several linear and non-linear models by using the GAME algorithm. GAME models can be serialized into simple equations that are understandable by domain experts.

We generated several models (see Eq. 14, 15, 16) by the GAME algorithm and compare them with well known EFW models, which has been found by linear and nonlinear regression methods by various authors in the past.

We loaded the data into the FAKE GAME open source application [5] and generated models by using standard configuration (if not indicated differently) of the GAME engine.

All generated models are simple and we also checked regression graphs of each model in GAME toolkit and see that every model has smooth progression (see Fig. 6.4.1) and approximate the output data set by hyperplane. Because the error is measured on testing data and the regression hyperplane is smooth, we can see that models are not overtrained and have good generalization ability.

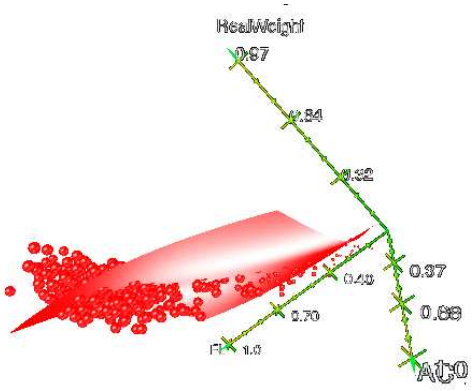

Fig. 6.36 An example of GAME model evolved on FL data. Regression hyperplane is smooth as expected.

The first model was serialized into polynomial formula (just polynomial neurons were enabled and the penalization for complexity of the transfer function was applied to get simple formulae). The error of the model is therefore higher (Tab. 6.5) than that of models with more complex formulas obtained with the standard configuration of the GAME engine:

$$EFW = 0.0504 \times AC^2 - 16.427 \times AC + 38.867 \times FL + 284.074 \tag{14}$$

$$EFW = -7637.17 + 7870.09 \times e^{3.728 \times 10^{-6} \times (AC-163)^2 + 0.0002 \times HC}$$

$$\times e^{\frac{1}{10.676+40011\times e^{-0.096\times BPD}}+\frac{1}{7.557+6113.68\times e^{-0.102\times FL}}} \quad (15)$$

Note that exponential and sigmoid neurons are very successful on this data set. Observed relationship of variables (Fig. 6.4.1) is apparently nonlinear. To simplify generated equations, we transformed the output into logarithmic scale for the last model. Model produced by GAME does not contain exponential terms any more, but neurons with sine transfer function were more successful than polynomial neurons:

$$\begin{aligned}\log_{10} EFW = {} & 2.18 + 0.0302 \times BPD + 0.0293 \times FL \\ & -0.603 \sin(0.524 - 0.0526 \times AC) \\ & -0.344 \sin(-0.029 \times AC - 0.117 \times FL + 0.946)\end{aligned} \quad (16)$$

In case that experts prefer polynomial equation, Sine neurons can be easily disabled in the configuration of the GAME engine.

6.4.1.1 Statistical evaluation of models

Table 6.4 Basic statistic characteristics of models, Major Percentiles [g]

Method	5%	10%	50%	90%	95%
Hadlock (13)	894	1401	3168	3678	3836
GAME (14)	950	1462	3149	3623	3779
GAME (15)	937	1424	3145	3633	3741
GAME (16)	886	1394	3173	3625	3720

Table 6.5 Model Correlation with Actual Birth Weight R^2, Mean absolute Error $\pm$ Standard deviation, RMS Error

Method	R^2	Mean Abs. Error [g] $\pm$ SD	RMS Error [g]
Hadlock (13)	0.91	199 ± 171	261
GAME (15)	0.91	199 ± 168	261
GAME (16)	0.91	203 ± 173	266
GAME (14)	0.91	209 ± 174	272

All Fake Game models are at least good as best models found by statistical approach. We succeeded to find models with the same R^2, lower mean absolute error, lower RMS error and lower standard deviation than models found by traditional techniques. We also decreased mean absolute error, standard deviation and RMS error by using ensemble of three models which increases accuracy of estimation of fetal weight (see [43]).

6.5 The FAKE GAME project

Knowledge discovery and data mining are popular research topics in recent times. It is mainly due to the fact that the amount of collected data significantly increases. Manual analysis of all data is no longer possible. This is where the data mining and the knowledge discovery (or extraction) can help.

The process of knowledge discovery [11] is defined as the non-trivial process of finding valid, potentially useful, and ultimately understandable patterns. The problem is that this process still needs a lot of human involvement in all its phases in order to extract some useful knowledge. Our research focuses on methods aimed at significant reduction of expert decisions needed during the process of knowledge extraction. Within the FAKE GAME environment we develop methods for automatic data preprocessing, adaptive data mining and for the knowledge extraction (see Fig. 6.37). The data preprocessing is very important and time consuming

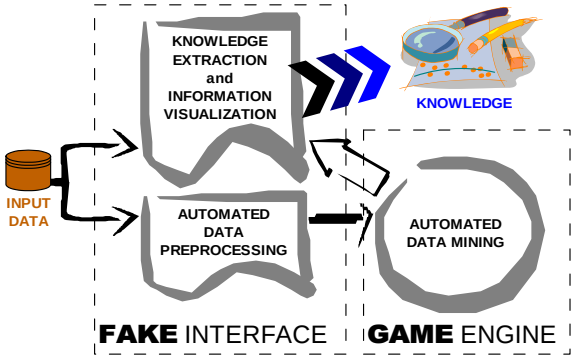

Fig. 6.37 FAKE GAME environment for the automated knowledge extraction.

phase of the knowledge extraction process. According to [37] it accounts for almost 60% of total time of the process. The data preprocessing involves dealing with non-numeric variables (alpha values coding), missing values replacement (imputing), outlier detection, noise reduction, variables redistribution, etc. The data preprocessing phase cannot be fully automated for every possible data set. Each data have unique character and each data mining method requires different preprocessing.

Existing data mining software packages support just very simple methods of data preprocessing [3]. There are new data mining environments [4, 1] trying to focus more on data preprocessing, but their methods are still very limited and give no hint which preprocessing would be the best for your data. It is mainly due to the fact that the theory of data preprocessing is not very developed. Although some preprocessing methods seem to be simple, to decide which method would be the most appropriate for some data might be very complicated. Within the FAKE interface we develop more sophisticated methods for data preprocessing and we study which

methods are most appropriate for particular data. The final goal is to automate the data preprocessing phase as much as possible.

In the knowledge extraction process, the data preprocessing phase is followed by the phase of data mining. In the data mining phase, it is necessary to choose appropriate data mining method for your data and problem. The data mining method usually generates a predictive, regressive model or a classifier on your data. Each method is suitable for different task and different data. To select the best method for the task and the data, the user has to experiment with several methods, adjust parameters of these methods and often also estimate suitable topology (e.g. number of neurons in a neural network). This process is very time consuming and presumes strong expert knowledge of data mining methods by the user.

In the new version of one commercial data mining software [46], an evolutionary algorithm is used to select the best data mining method with optimal parameters for actual data set and a problem specified. This is really significant step towards the automation of the data mining phase. We propose a different approach. The ensemble of predictive, regressive models or classifiers is generated automatically using the GAME engine. Models adapt to the character of a data set so that they have an optimal topology. We develop methods eliminating the need of parameters adjustment so that the GAME engine performs independently and optimally on bigger range of different data.

The results of data mining methods can be more or less easily transformed into the knowledge, finalizing the knowledge extraction process. Results of methods such as simple decision tree are easy to interpret. Unfortunately majority of data mining methods (neural networks, etc.) are almost black boxes - the knowledge is hidden inside the model and it is difficult to extract it.

Almost all data mining tools bound the knowledge extraction from complex data mining methods to statistical analysis of their performance. More knowledge can be extracted using the techniques of information visualization. Recently, some papers [49] on this topic had been published. We propose techniques based on methods such as scatterplot matrix, regression plots, multivariate data projection, etc. to extract additional useful knowledge from the ensemble of GAME models. We also develop evolutionary search methods to deal with the state space dimensionality and to find interesting projections automatically.

6.5.1 The goal of the FAKE GAME environment

The ultimate goal of our research is to automate the process of knowledge extraction from data. It is clear that some parts of the process still need the involvement of expert user. We build the FAKE GAME environment to limit the user involvement during the process of knowledge extraction. To automate the knowledge extraction process, we research in the following areas: data preprocessing, data mining, knowledge extraction and information visualization (see Fig. 6.38).

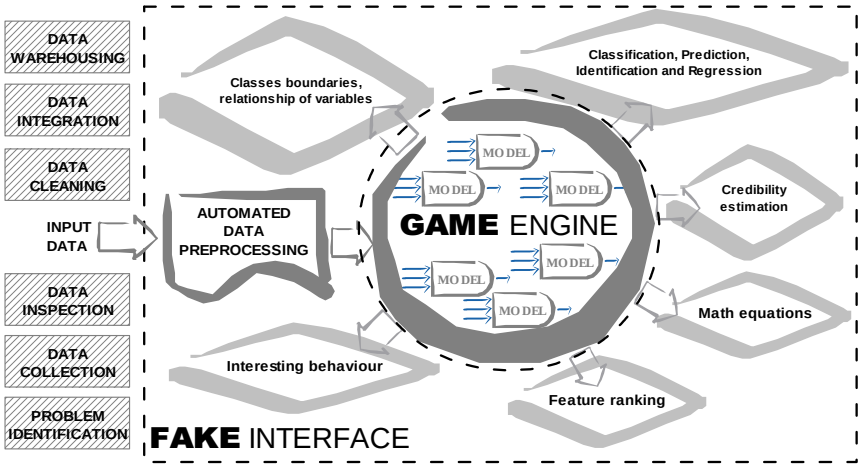

Fig. 6.38 Fully Automated Knowledge Extraction (FAKE) using Group of Adaptive Models Evolution (GAME)

6.5.1.1 Research of methods in the area of data preprocessing

In order to automate the data preprocessing phase, we develop more sophisticated methods for data preprocessing.We focus on data imputing (missing values replacement), that is in existing data mining environments [4, 1] realized by zero or mean value replacement although more sophisticated methods already exist [37]. We also developed a method for automate nonlinear redistribution of variables. FAKE

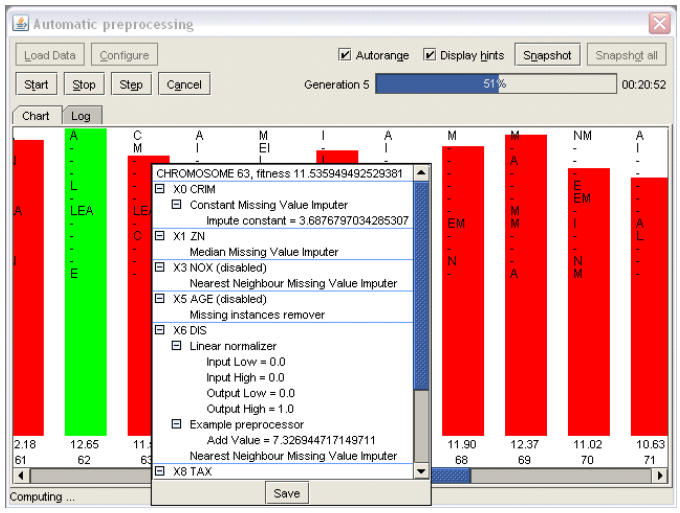

Fig. 6.39 Automated preprocessing module implemented in the FAKE GAME.

GAME is not focusing on data warehousing, because this process is very difficult to automate in general. It is very dependent on particular conditions (structure of databases, information system, etc.) We assume that source data are already collected cleansed and integrated (Fig. 6.38).

6.5.1.2 Automated data mining

To automate the data mining phase, we develop an engine that is able to adapt itself to the character of data. This is necessary to eliminate the need of parameter tuning. The GAME engine autonomously generates the ensemble of predictive, regressive models or classifiers. Models adapt to the character of data set so that they have optimal topology. Unfortunately, the class of problems where the GAME engine performs optimally is still limited. To make the engine more versatile, we need to add more types of building blocks, more learning algorithms, improve the regularization criteria, etc.

6.5.1.3 Knowledge extraction and information visualization

To extract the knowledge from complex data mining models is very difficult task. Visualization techniques are promising way how to achieve it. Recently, some papers [49] on this topic had been published. In our case, we need to extract information from an ensemble of GAME inductive models. To do that we enriched methods such as scatterplot matrix, regression plots by the information about the behavior of models. For data with many features (input variables) we have to deal with curse of dimensionality. The state space is so big, that it is very difficult to find some interesting behavior (relationship of system variables) manually. For this purpose, we developed evolutionary search methods to find interesting projections automatically.

Along with the basic research, we implement proposed methods in Java programming language and integrate it into the FAKE GAME environment [5] so we can directly test the performance of proposed methods, adjust their parameters, etc. Based on the research and experiments performed within this dissertation, we are developing the open source software FAKE GAME. This software should be able to automatically preprocess various data, to generate regressive, predictive models and classifiers (by means of GAME engine), to automatically identify interesting relationships in data (even in high-dimensional ones) and to present discovered knowledge in a comprehensible form. The software should fill gaps which are not covered by existing open source data mining environments [3, 4]. You can download the application to experiment with your data or join our community at Sourceforge [5].

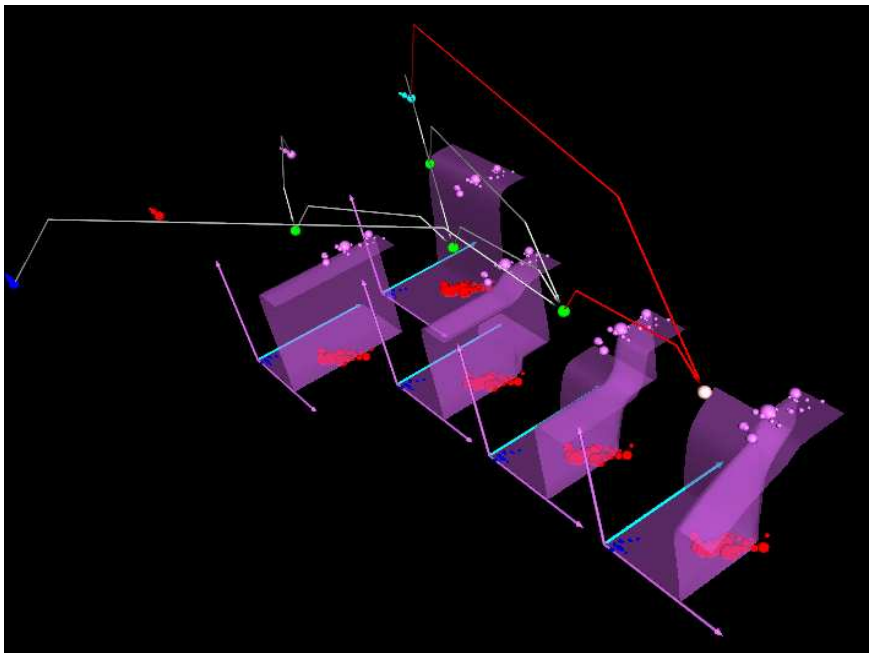

Fig. 6.40 3D inspection of GAME model topology and behavior (Iris Versicolor class).

6.6 Acknowledgement

I would like to thank to my collaborators Miroslav Cepek, Jan Drchal, Ales Pilny, Oleg Kovarik, Jan Koutnik, Tomas Siegl, members of the Computational Intelligence Research Group and all students participating in the FAKE GAME project. Thanks to head and former head of our research group Miroslav Skrbek and Miroslav Snorek.

This research is partially supported by the grant Automated Knowledge Extraction (KJB201210701) of the Grant Agency of the Academy of Science of the Czech Republic and the research program ”Transdisciplinary Research in the Area of Biomedical Engineering II” (*MSM*6840770012) sponsored by the Ministry of Education, Youth and Sports of the Czech Republic.

References

1. The sumatra tt data preprocessing tool. available online at http://krizik.felk.cvut.cz/sumatra/, September 2006.
2. Uci machine learning repository. available at http://www.ics.uci.edu/ mlearn/MLSummary.html, September 2006.

3. Weka open source data mining software. available online at http://www.cs.waikato.ac.nz/ml/weka/, September 2006.
4. The yale open source learning environment. available online at http://www-ai.cs.uni-dortmund.de/SOFTWARE/YALE/intro.html, September 2006.
5. The fake game environment for the automatic knowledge extraction. available online at: http://www.sourceforge.net/projects/fakegame, November 2008.
6. K. Adeney and M. Korenberg. An easily calculated bound on condition for orthogonal algorithms. In *IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN'00)*, volume 3, page 3620, 2000.
7. G. Bilchev and I. C. Parmee. The ant colony metaphor for searching continuous design spaces. In *Selected Papers from AISB Workshop on Evolutionary Computing*, pages 25–39, London, UK, 1995. Springer-Verlag.
8. C. Blum and K. Socha. Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In *Proceedings of Hybrid Intelligent Systems Conference, HIS-2005*, pages 233–238, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
9. G. Brown. *Diversity in Neural Network Ensembles*. PhD thesis, The University of Birmingham, School of Computer Science, Birmingham B15 2TT, United Kingdom, January 2004.
10. S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, Carnegie Mellon University Pittsburgh, USA, 1991.
11. U. Fayyad, G. Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54, 1996.
12. P. Granitto, P. Verdes, and H. Ceccatto. Neural network ensembles: evaluation of aggregation algorithms. *Artificial Intelligence*, 163:139–162, 2005.
13. L. Hansen and P. Salamon. Neural network ensembles. *IEEE Trans. Pattern Anal. Machine Intelligence*, 12(10):993–1001, 1990.
14. Y.-C. Ho and D. Pepyne. Simple explanation of the no free lunch theorem of optimization. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 5, pages 4409–4414vol.5, 4-7 Dec. 2001.
15. J. Holland. *Adaptation in Neural and Artificial Systems*. University of Michigan Press, 1975.
16. O. Hrstka and A. Kučerová. Improvements of real coded genetic algorithms based on differential operators preventing premature convergence. *Advances in Engineering Software*, 35(3-4):237–246, March-April 2004.
17. M. Islam, X. Yao, and K. Murase. A constructive algorithm for training cooperative neural network ensembles. *IEEE Transitions on Neural Networks*, 14(4), July 2003.
18. A. G. Ivakhnenko. Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(1):364–378, 1971.
19. C.-F. Juang and Y.-C. Liou. On the hybrid of genetic algorithm and particle swarm optimization for evolving recurrent neural network. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, volume 3, pages 2285–2289, Dept. of Electr. Eng., Nat. Chung-Hsing Univ., Taichung, Taiwan, 25-29 July 2004.
20. H. Juille and J. B. Pollack. Co-evolving intertwined spirals. In P. J. A. Lawrence J. Fogel and T. Baeck, editors, *Proceedings of the Fifth Annual Conference on Evolutionary Programming*, Evolutionary Programming V, pages 461–467. MIT Press, 1996.
21. R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1995.
22. T. Kohonen. *Self-Organizing Maps*. Springer, Berlin, Heidelberg, New York,, 2001.
23. M. Kong and P. Tian. A direct application of ant colony optimization to function optimization problem in continuous domain. In *Ant Colony Optimization and Swarm Intelligence, 5th International Workshop, ANTS 2006, Brussels, Belgium, September 4-7, 2006. Proceedings*, Lecture Notes in Computer Science, pages 324–331. Springer, 2006.
24. P. Kordík. Game - group of adaptive models evolution. Technical Report DCSE-DTP-2005-07, Czech Technical University in Prague, FEE, CTU Prague, Czech Republic, 2005.
25. P. Kordík. *Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution*. PhD thesis, Czech Technical University in Prague, FEE, Dep. of Comp. Sci. and Computers, FEE, CTU Prague, Czech Republic, September 2006.

26. P. Kordík, V. Křemen, and L. Lhotská. The game algorithm applied to complex fractionated atrial electrograms data set. In *Artificial Neural Networks - ICANN 2008, 18th International Conference Proceedings*, volume 2, pages 859–868, Heidelberg, 2008. Springer.
27. P. Kordík, P. Náplava, M. Šnorek, and M. Genyk-Berezovskyj. The Modified GMDH Method Applied to Model Complex Systems. In *International Conference on Inductive Modeling - ICIM 2002*, pages 150–155, Lviv, 2002. State Scientific and Research Institute of Information Infrastructure.
28. L. Kuhn. *Ant Colony Optimization for Continuous Spaces*. PhD thesis, The Department of Information Technology and Electrical Engineering The University of Queensland, October 2002.
29. Y.-j. Li and T.-j. Wu. An adaptive ant colony system algorithm for continuous-space optimization problems. *J Zhejiang Univ Sci*, 4(1):40–6, 2003.
30. S. W. Mahfoud. A comparison of parallel and sequential niching methods. In *Sixth International Conference on Genetic Algorithms*, pages 136–143, 1995.
31. S. W. Mahfoud. Niching methods for genetic algorithms. Technical Report 95001, Illinois Genetic Algorithms Laboratory (IlliGaL), University of Ilinios at Urbana-Champaign, May 1995.
32. M. Mandischer. A comparison of evolution strategies and backpropagation for neural network training. *Neurocomputing*, (42):87–117, 2002.
33. N. Monmarché, G. Venturini, and M. Slimane. On how pachycondyla apicalis ants suggest a new search algorithm. *Future Gener. Comput. Syst.*, 16(9):937–946, 2000.
34. J. A. Muller and F. Lemke. *Self-Organising Data Mining*. Berlin, 2000. ISBN 3-89811-861-4.
35. N. Nariman-Zadeh, A. Darvizeh, A. Jamali, and A. Moeini. Evolutionary design of generalized polynomial neural networks for modelling and prediction of explosive forming process. *Journal of Materials Processing Technology*, (165):1561–1571, 2005.
36. S.-K. Oh, W. Pedrycz, and B.-J. Park. Polynomial neural networks architecture: analysis and design. *Computers and Electrical Engineering*, 29(29):703–725, 2003.
37. D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufman, 1999. Fondi di Ricerca Salvatore Ruggieri - Numero 421 d'inventario.
38. Salane and Tewarson. A unified derivation of symmetric quasi-newton update formulas. *Applied Math*, 25:29–36, 1980.
39. R. Schnabel, J. Koontz, and B. Weiss. A modular system of algorithms for unconstrained minimization. Technical Report CU-CS-240-82, Comp. Sci. Dept., University of Colorado at Boulder, 1982.
40. U. Seiffert and B. Michaelis. Adaptive three-dimensional self-organizing map with a two-dimensional input layer. In *Intelligent Information Systems, 1996., Australian and New Zealand Conference on*, pages 258–263, 18-20 Nov. 1996.
41. R. S. Sexton and J. Gupta. Comparative evaluation of genetic algorithm and backpropagation for training neural networks. *Information Sciences*, (129):45–59, 2000.
42. J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, August 1994.
43. T. Siegl, P. Kordík, M. Šnorek, and P. Calda. Fetal weight prediction models: Standard techniques or computational intelligence methods? In *Artificial Neural Networks - ICANN 2008, 18th International Conference Proceedings*, volume 1, pages 462–471, Heidelberg, 2008. Springer.
44. K. Stanley, B. Bryant, and R. Miikkulainen. Real-time neuroevolution in the nero video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653–668, Dec. 2005.
45. K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. Massachusetts Institute of Technology.
46. Statsoft. Statistica neural networks software. More information at http://www.statsoft.com/products/stat_nn.html, September 2006.
47. R. Storn and K. Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

48. S. Tsutsui, M. Pelikan, and A. Ghosh. Performance of aggregation pheromone system on unimodal and multimodal problems. In *The IEEE Congress on Evolutionary Computation, 2005 (CEC2005)*, volume 1, pages 880–887. IEEE, 2-5 September 2005.
49. F.-Y. Tzeng and K.-L. Ma. Opening the black box - data driven visualization of neural networks. In *Proceedings of IEEE Visualization '05 Conference*, pages 23–28, Minneapolis, USA, October 2005.
50. J. Vesterstrom and R. Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Proceedings of the 2004 Congress on Evolutionary Computation*, volume 2, pages 1980–1987, 2004.
51. J. G. Wade. Convergence properties of the conjugate gradient method. available at www-math.bgsu.edu/ gwade/tex_examples/example2.txt, September 2006.
52. D. Wickera, M. M. Rizkib, and L. A. Tamburinoa. E-net:evolutionary neural network synthesis. *Neurocomputing*, 42:171–196, 2002.