Construction and adjustment of differential polynomial neural network

Ladislav Zjavka

Faculty of Management Science and Informatics, University of Žilina, Univerzitná 8215/1, 010 01 Žilina, Slovakia.
E-mail: lzjavka@gmail.com.
Accepted 26 January, 2011

Artificial neural networks in general are used to identify patterns according to their entire relationship, responding to related patterns with a similar output of applying absolute values of variables. However, a lot of real data contain some unknown relations of variables. Learning of these dependencies could be a new way of modelling complex systems instead of usual time series prediction based on pattern similarity. Differential polynomial neural network, which constructs a differential equation of fractional terms using multi-parametric polynomial functions, is a new type of neural network developed by the author. Its functionality is based on principles, which are applied in human brain learning. The brain does not utilize absolute values of variables, but relative ones, which are created by time-delayed dynamic periodic activation functions of biological neurons. They take part in differential equation composition as partial derivative terms, describing a relative change of particular dependent variables.

Key words: Polynomial neural network, dependence of variables identification, differential equation approximation, rational integral function, modelling of complex system.

INTRODUCTION

Artificial neural networks (ANN) are trained to classify certain patterns into groups, and then they are used to identify the new ones, which were never presented before. If ANN is trained, for example, to identify a shape, it can correctly classify only the incomplete or similar patterns as compared to the trained ones (Kvasnička et al., 1997), but in a case the shape is moved or its size is changed in the input matrix of variables the neural network identification will fail. The principal lack of the ANN identification in general is its disability of input pattern generalization. It utilizes only the absolute values of variables, but these can differ enormously, while their relations may be the same. ANN is in principle a simplified form of polynomial neural network (PNN), whose combinations of variables are missing. These partly describe the data relations in polynomials through exponential functions, created by multiplications.

A trial should be made in looking at the vector of the input variables as a no “pattern”, but a bound dependent point set of N-dimensional space. Likewise, the ANN classified the patterns, while the study tried to identify any unknown relations of the data variables. The response of this neural network should be the same for all patterns (dependent sets), whose variables are performed with the trained dependence. It does not matter what values they become, a multi-parametric non-linear function can describe their relations to each other. So, if neural network creates such a compound function, its neurons must apply some $n$-parametric polynomial functions to catch the partial dependence of its $n$-inputs. The biological neural cell (Figure 1) applies a similar principle and its dendrites collect signals that come from other neurons, but unlike the artificial neuron, the signals already interact in single branches (dendrites). This could be modelled with multiplications of some inputs in polynomials (PNN). Subsequently, these weighted combinations are summed in the cell of the body and are transformed using time-delayed dynamic periodic activation function (the activated neural cell generates a series of time-delayed output pulses, in response to its input signals) (Beňušková, 2002). The period of this function depends on some combinations of input variables and seems to represent the derivative part of a partial derivation of an entire polynomial (as a term of differential equation).

Polynomial neural network for dependence of variables identification (or differential polynomial neural network – D-PNN, because it constructs a differential equation)
describes a functional dependence of input variables that are not entirely patterns as ANN are. This could be regarded as a pattern abstraction, similar to the ones utilized by the brain, in which identification is not based on values of variables, but only relations of these. D-PNN forms its functional output as a generalization of input patterns.

GMDH polynomial neural network

General connection between input and output variables is expressed by the Volterra functional series, a discrete analogue of which Kolmogorov-Gabor polynomial (1) is:

\[
y = a_0 + \sum_{i=1}^{m} a_i x_i + \sum_{i=1}^{m} \sum_{j=1}^{m} a_{ij} x_i x_j + \sum_{i=1}^{m} \sum_{j=1}^{m} \sum_{k=1}^{m} a_{ijk} x_i x_j x_k + \ldots \tag{1}
\]

where:

- \( m \) – number of variables
- \( X(x_1, x_2, \ldots, x_m) \) - vector of input variables
- \( A(a_1, a_2, \ldots, a_m) \) - vectors of parameters

This polynomial can approximate any stationary random sequence of observations and can be computed by either adaptive methods or the system of Gaussian normal equations (Ivakhnenko, 1971).

The starting point of the new neural network type D-PNN development was the GMDH polynomial neural network, created by a Ukrainian scientist Aleksey Ivakhnenko in 1968. When the back-propagation technique was not known yet, a technique called Group Method of Data Handling (GMDH) was developed for neural network structure design and parameters of polynomials adjustment. An attempt was made for it to have a resemblance with the Kolmogorov-Gabor polynomial (1) by using low order polynomials and (2) every pair of the input values (Galkin, 2000):

\[
y' = a_0 + a_1 x_i + a_2 x_j + a_3 x_i x_j + a_4 x_i^2 + a_5 x_j^2 \tag{2}
\]

The GMDH neuron has two inputs and its output is a quadratic combination of 2 inputs, totalling 6 weights (Figure 2). Thus, GMDH network builds up a polynomial...
(actually a multinomial) combination of the input components. The typical GMDH network (Figure 3) maps a vector input \( x \) to a scalar output \( y' \), which is an estimate of the true function of \( f(x) = y \). Each neuron of the polynomial network fits its output to the desired value \( y \) for each input vector \( x \) from the training set. The manner in which this approximation is accomplished is through the use of linear regression (Galkin, 2000).

In the hope to capture the complexity of a process, this neural network attempts to decompose it into many simpler relationships, each described by a processing function of a single neuron (2). It defines an optimal structure of the complex system model, identifying non-linear relations between input and output variables. Polynomial neural network (PNN) is a flexible architecture, whose structure is developed through learning. The number of layers of the PNN is not fixed in advance, but it becomes dynamically meaningful that this self-organising network grows over the trained period (Oh et al., 2003).

**Differential equation approximation**

The basic idea of the author’s D-PNN is to approximate a differential equation (3), which can define relations of variables (Hronec, 1958), with a special type of root fractional multi-parametric polynomials; for example Equations (4) and (5):

\[
Y = a + \sum_{i=1}^{a} b_{i} \frac{\partial u}{\partial x_{i}} + \sum_{i=1}^{a} \sum_{j=1}^{a} c_{ij} \frac{\partial^{2} u}{\partial x_{i} \partial x_{j}} + \ldots = \text{const.} \quad (3)
\]

\[
u = f(x_{1}, x_{2}, \ldots, x_{n}) \quad \text{function of input variables}
\]

\[
a, B(b_{1}, b_{2}, \ldots, b_{n}), C(c_{11}, c_{12}, \ldots) \quad \text{parameters.}
\]

Elementary methods of the differential equation (DE) solution express the solution in special elementary functions – polynomials (such as Bessel’s functions or power series). Numerical integration of differential equations is based on their approximation using:

(a) Rational integral functions.

(b) Trigonometric series.

The most simplest way for this integration have been selected using the method of integral analogues, by replacing mathematical operators in equations with a ratio of pertinent values (Kuneš et al., 1989).

\[
y_{i} = \frac{(a_{i} + a_{i}x_{1} + a_{i}x_{2} + a_{i}x_{3} + a_{i}x_{4} + a_{i}x_{5} + a_{i}x_{6} + a_{i}x_{7})^{\gamma_{i}}}{(b_{i} + b_{i}x_{1} + b_{i}x_{2} + b_{i}x_{3} + b_{i}x_{4} + b_{i}x_{5} + b_{i}x_{6} + b_{i}x_{7})^{\gamma_{i}}} \quad (4)
\]

\[
y_{i} = \frac{(a_{i} + a_{i}x_{1} + a_{i}x_{2} + \ldots + a_{i}x_{n} + a_{i}x_{1}x_{2} + \ldots)^{\gamma_{i}}}{(b_{i} + b_{i}x_{1} + \ldots)^{\gamma_{i}}} \quad (5)
\]
$n$ – combination degree of $n$-input variables of the numerator.  
$m$ – combination degree of denominator ($m<n$).

The fractional polynomials (4 and 5), which can describe a partial dependence of $n$-input variables of each neuron, are applied as terms of the DE (3) composition. They partly create an unknown multi-parametric non-linear function, which codes relations of input variables. The numerator of Equations (4) and (5) is a polynomial of complete $n$-input combination degree of a single neuron and it realizes a new function $z$ of formula (6). The denominator of Equations (4) and (5) is a derivative part, which gives a partial mutual change of some neuron input variables and its polynomial combination degree $m$ is less than $n$. It arose from the partial derivation of the complete $n$-variable polynomial by competent variable(s). In general, it is possible to express this approximation in formula (6) (Hronec, 1958):

$$Y = w_0 + w_1 \frac{\partial z}{\partial x_1} + w_2 \frac{\partial z}{\partial x_2} + \ldots + w_m \frac{\partial z}{\partial x_m} + \cdots + \text{const}$$  \hspace{1cm} (6)

$z$ – function of $n$-input variables

$w_i$ – weights of terms

Each layer of the D-PNN consists of blocks of neurons. Block contains derivative neurons, one for the variables combination of each fractional polynomial (4) and (5), thereby defining a partial derivative dependent change of some input variables (Figure 4). The root functions of denominators are lower than $n$, according to the combination degree which take the polynomials of neurons (4) and (5) into competent power degree. Neurons do not affect the block output but are applied only for the total output calculation (DE composition). Each block also contains a single polynomial (without the derivative part), which forms its output entrance into the next hidden layer. Each neuron has 2 vectors of adjustable parameters ($a$ and $b$) and each block contains 1 vector of adjustable parameters of the output polynomial. Inputs of constant combination degree ($n=2,3,\ldots$) forming particular combination of variables, enter each block, where they are substituted into polynomials. Therefore, it is necessary to adjust not only the polynomial parameters, but the D-PNN’s structure too. This means some neurons, in terms of role of the DE, have to be left out.
Following the examples, simplicity is assumed only for linear dependencies of variables. If there is an occurrence of a non linear dependence in the input data, square power exponent variables would likely extend the combination polynomials and could be applied also as competent derivative terms.

### Identification of simple dependencies of variables

When considering a very simple dependence of the 2-input variables, multiplicity is constant (for example = 2). D-PNN will contain only 1 block of 2 polynomial neurons (7) and (8) as terms of DE (Figure 5). As the input variables do not change constantly, it is necessary to add both terms (fractional polynomial of derivative variable \( x_1 \) and \( x_2 \)) in the DE (block). D-PNN will learn this relation according to samples of the training data set by means of genetic algorithm (GA).

\[
y_1 = w_1 \frac{\left( a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 \right)^{\frac{1}{2}}}{b_0 + b_1 x_1} \quad (7)
\]

\[
y_2 = w_2 \frac{\left( a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 \right)^{\frac{1}{2}}}{b_0 + b_1 x_2} \quad (8)
\]

Consider a more complicated dependence, where 2 variables depend on the 3rd variable. For example, the sum of the first 2 variables equals the 3rd one \( (x_1 + x_2 = x_3) \). This task seems to be very simple, but it is rather difficult for D-PNN to learn it. The complete DE (for derivatives 1 and 2-combinations of block) consists of 6 terms (neurons), but only 3 of them will be enough for the derivative terms \( x_3 \) (9), \( x_1 x_3 \) (10) and \( x_2 x_3 \) (11). If other terms (neurons) are added, the D-PNN will work amiss (Figure 6).

Figure 5. Identification of a constant quotient of 2 variables \( (x_1 = 2x_2) \).

Figure 6. Identification of the sum dependence.
This 3-variable dependence is described by more complicated exponential functions. The D-PNN as well is charged by the possible 2-sided change of input variables. For example \(1 + 9 = 10\) is the same sum as \(9 + 1 = 10\). The 2-combination polynomials of numerators, which could improve the D-PNN functionality, can also be applied. The principal phase of its adjustment resides in the elimination of some neurons (in terms of DE).

**Multi-layered D-PNN**

It is possible to solve this problem of multi-layered D-PNN with 2-combination blocks of variables. Even though, in this way, more mistakes can occur by identification, more complex DE would be created (that is, much more terms arise). However, it can be clearly seen that the D-PNN operates. If the sum of the first two variables is less than it should be, the output of the D-PNN is less than the desired round and the other hand round. So, it is shown that a separating plane is detached from the relative “classes”, which have the same characteristic (dependence); likewise, it is current by ANN’s pattern identification. The problem of the multi-layered D-PNN construction reside creates every partial combination term for a complete DE in utilizing some fixed low combination degrees (2, 3), while the amount of variables is higher.

Each block of the D-PNN takes part in the total network output calculation (forms the partial DE terms) and utilizes its basic and extended neurons with back connected blocks of previous layers. The single adjustable polynomial (Figure 7) without a derivative part creates the block output (applied in the next hidden layer), but the neurons are applied only for the total DE composition. The blocks of the 2nd and the following hidden layers extend the basic DE terms (neurons) using their additional neurons, outputs and inputs of back connected blocks of the previous layers to create compound terms (Figure 7) of the DE. Consider for instance the 1st block of the last hidden layer, which takes its own neurons as 2 basic terms (12) of the DE (6).
Subsequently, it creates 4 extended terms of the 2nd (previous) hidden layer, using reverse output polynomials and inputs of bound blocks. It joins these 2 blocks and creates 4 fractional terms of the DE for 4 derivative input variables of 2 previous blocks, for example (13).

\[
y'_1 = w_i \sqrt{\frac{x_i}{2(b_0 + b_1 x_i)}} = w_i \frac{(a_0 + a_1 x_i + a_2 x_i^2 + a_4 x_i^4)^{1/2}}{2(b_0 + b_1 x_i)}
\]

(12)

\[
y'_2 = w_i \sqrt{\frac{1}{2x_i}} 
\]

(13)

\[
y'_3 = w_i \sqrt{\frac{x_i}{2(c_0 + c_1 x_i)}}
\]

The back connection of the previous layer(s) is realized through the output polynomials of the linked 2nd (or 1st) layer opposite the blocks, forming fractional parts in numerators of formulas (13) and (14). Likewise, terms can be created for the 1st hidden layer (14). The 3 linked blocks, forming 8 terms of the DE were attached to the layer and were performed well by a recursive algorithm. The multiplication of “2” (or higher) in the denominators of formulas (12), (13) and (14) is applied in decreasing the D-PNN’s total output value. It was not every term that was used in the complete DE; some of them were necessarily left out. This indicates “0” or “1” in the neurons of blocks and is easy to use them as genes of GA.

**Construction of “wedge” networks of the D-PNN**

It can be seen that the D-PNN (Figure 7) substantially consists of 3 overlaying “wedge” networks (WN), each going back out from the blocks of the last hidden layer and gradually attaching to the derivative variables of previous layers.

The D-PNN of the 4 dependent input variables using 2-combination blocks will have totally 6 blocks of all input combination couples in the 1st hidden layer. It could have at first 6 hidden layers at the back of the 1st layer derivative input variables (from the last hidden layer blocks) and several overlaying WNs partly in the layer (Figure 8). The number of layers can be brought down to 4 and this decreases the number of combination blocks of WNs too. The number of combinations for each variable increases enormously in the next hidden layer. This had caused problems, which were solved by applying WNs, as only some of the blocks were created and used. Each block of the 1st hidden layer was taken as a dominant one and the 2nd layer blocks are created from its WN as a combination with the rest of its WN blocks. This way, the number of all WN blocks decreases in the next hidden layer until it reached just 1 block. Some WN
layers overlay each other and so the blocks can be used several times by different WNs (Figure 7). Block inputs of the 2nd and the following layers can be subsequently reconnected and this could compensate the missing combination blocks and WNs. However, it is better to preserve the rough structure of each WN, which warranted the best results. The connections of the complete 1st hidden layer blocks are fixed.

**ADJUSTMENT OF THE D-PNN**

**Genetic algorithm**

GA can perform the search of the fittest input combinations of higher layer blocks using permutation encoding of chromosomes, for example:

2nd layer: 12 15 23 24 26 34 35 4 5 ...  
3rd layer: 13 14 18 19 23 25 36 3 10 ...

Only some of the entire possible DE terms can be applied. GA can again find their fittest combination using binary encoding of them and crossover operation for example:

0 1 0 0 1 0 1 0 1 ...

If too many DE terms (or WNs) are employed, an effect of “overlearning”, could be shown when the dependence generalization “slides off” (turns) into pattern identification (which is undesirable). The total output error of training data set decreases, but the dependence generalization is not reached. Terms of higher layer blocks seem to influence strongly the right dependence identification.

**Exact parameter calculation**

The adjustment is another problem that will be solved besides the D-PNN’s construction. A trial could be made to calculate the precise polynomial parameter change according to total and partial output errors. This method requires exactly the rules of the DE term’s behaviour to know. The Hebb’s delta rule (Kvasnička et al., 1997) has been applied by the DE term parameter adjustment. So, the bigger the values of the input variables of a term, the bigger its polynomial parameter change (+ or – according to the output error) will be. The parameter change must be given relatively to the absolute wide range of the input variables’ values and with regard to nominator or denominator setting in fractions, for example, (15), (16) and (17) - for basic neurons of the 1st block of the D-PNN’s last hidden layer (Figure 7). This way, the total output error can be brought down to certain limit, but the effect of dependence generalization is not shown finally (or slightly). There might be some other unknown principles of the DE term’s functionality applied.

\[
\begin{align*}
\delta_0 &= \frac{a_0}{\sqrt{a_3 x_1 x_2}} \\
\quad &\quad \frac{b_0 + b_1 x_1}{\\}
\delta_2 &= \frac{x_2}{a_3 x_1 x_2} \\
\quad &\quad \frac{b_0 + b_1 x_1}{\\}
\delta_3 &= \frac{\sqrt{x_1 x_2}}{b_0 + b_1 x_1}
\end{align*}
\] (15) (16) (17)

**Evolution strategy**

Parameters of polynomials are represented by real numbers, in which initial values are randomly generated from the interval of 0.5 and 1.5. A chromosome is a sequence of their values, which can be easily mutated. Polynomial outputs can not be a negative value, but parameters can. Evolution strategy (ES) creates descendants usually by modification (mutation) of only 1 individual; because it need not use the operation of crossover. Problem solving can be expressed as looking for the minimum of error (fitness) function of a search space (Figure 9) (Obitko, 1998). There can arise many local solutions, caused by plenty of possible
combinations of block inputs that are composed of DE terms (only some of them may be employed). The study's ES is based on evolving 1 individual, whose parameters are in the 1st generation (cycle), mutated only randomly with decreasing learning coefficients. As a result, each cycle creates a solution that is finished with high probability in a local error minimum. These local solutions, form “learning schemes” (LS), give directions of parameter changes in the next generations. Fitter solutions (with lower error) can arise for each LS in the next generation and can replace the existing older LS. LS define the particular parameter changes that are applied simultaneously on the actually modified 1 individual (18), besides its mutation. So each actual individual utilises the existing partial local solutions of LS to find a better (global) error minimum.

\[ X_{\text{act}} = X_{\text{act}} + E \ast (X_{lsi} - X_{\text{act}}) \]  \hspace{1cm} (18)

\[ X(a_1, a_2, ..., a_n) \text{- vectors of parameters; } \text{act=actual, } lsi= i-th \text{ LS; } E = <0, 1> \text{- learning rate} \]

**Evolution algorithm**

Difference evolution algorithm (DEA) was developed for optimizing real parameter functions. The main difference between DEA and GA is that GA gives more importance to crossover operation, while DEA’s main operation is based on the differences of randomly sampled pairs of solutions in the population. DEA generates new parameter vectors by adding the weighted difference vector between 2 population members and a 3rd member. If the resulting vector yields a lower objective function value, the newly generated vector replaces the vector that it was compared with. Like other evolutionary algorithms, the performance of DEA deteriorates with the increase in dimensionality of the objective function. \( D = 10 \text{–} 20 \) can be applied to individuals of each population \( P(x_1, x_2, ..., x_d) \). Each individual from the current population is presented as \( N \)-dimensional vector, where \( N \) corresponds to the number of required variables. The core of DEA is built in the strategy of creating mutation vector \( u(u_1, u_2, ..., u_n) \). The most frequent strategy applies weighted difference from randomly chosen solutions (19), to each random individual from target population \( P \). Another strategy can generate mutation vector \( u \) from the best solution \( x_{\text{min}} \) in the previous and present population (20). Control factor \( F \) is entered from interval \([0, 1] \) (Hájek, 2010).

\[ u = x_{i1} + F(x_{i2} - x_{i3}) \quad r_1 \neq r_2 \neq r_3 \neq i \]  \hspace{1cm} (19)

\[ u = x_{\text{min}} + F(x_{i1} - x_{i2}) \quad r_1 \neq r_2 \neq \text{min} \neq i \]  \hspace{1cm} (20)

After creating a mutation \( u \) (called perturbed or “noise” vector), it is possible to make \( y_i \) (trial vector) a descendant from parents \( u \) and \( x_i \) using crossover operation according to following Equation (21).

\[ y_{j,i} = \begin{cases} u_j & \text{if rand}_j \leq \text{CR} \lor j = k \\ x_{j,i} & \text{otherwise} \end{cases} \]  \hspace{1cm} (21)

Variable \( j = 1, 2, ..., n \), \( k \in \{1, ..., n\} \) is a random parameter's index chosen for each \( i \), whereas CR (frequency of crossing) is from interval \([0, 1] \). This last parameter represents a probability of crossover that influences the generation of the perturbed individual \( u \), by controlling the amount of genes, which will be changed from the target individual \( x_i \) to the trial one \( y_i \) (Ali et al., 2009).

\[ X_{i,G+1} = \begin{cases} Y_{i,G} & \text{if } f(Y_i) \leq f(X_i) \\ X_{i,G} & \text{otherwise} \end{cases} \]  \hspace{1cm} (22)

The population for the next generation is selected from the target individual \( x_i \) in the current generation and its corresponding trial vector \( y_i \). Each individual of the trial population is compared with its counterpart in the current population (22). The one with the lower objective function value will survive from the tournament selection to the population of the next generation. As a result, all individuals of the next generation are as good as better than their counterparts in the current generation. In DEA, the trial vector was not compared to the individuals in the current generation, but was compared only against 1 individual, which was its counterpart (Ali et al., 2009).

DEA can be simply described as follows:

1. Create 'initial' random population \( P(x_1, x_2, ..., x_d) \)
2. Create 'mutation' vector \( u(u_1, u_2, ..., u_n) \)
3. Create 'new solution' \( y_i \) by combining parents \( u \) and \( x_i \)
4. If \( f(y_i) \leq f(x_i) \), then replace \( x_i \) by \( y_i \) (\( f = \text{target function} \))
5. Repeat until the stop condition

There is no need considering the output of the D-PNN, so it is adjusted to an average random value according to the initial parameters. Same applies to all samples of the training data set. Then the response to all input vectors, performed with the trained dependence of variables, should follow the same trend too. The output error of all training data samples (22) must be computed relatively to the present average output value (\( y_{\text{avg}} \), as omission of neurons (DE terms) and reconnecting inputs of blocks influence its computation. The omitted neuron (term) would always decrease it without employing the denominator of (22).
Table 1. Responses to random input vectors with dependent variables.

<table>
<thead>
<tr>
<th>Input vector</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 + 42 = 6 + 46</td>
<td>3.901</td>
</tr>
<tr>
<td>20 + 2 = 21 + 1</td>
<td>4.142</td>
</tr>
<tr>
<td>45 + 20 = 38 + 27</td>
<td>3.994</td>
</tr>
<tr>
<td>2 + 40 = 38 + 4</td>
<td>3.896</td>
</tr>
<tr>
<td>25 + 13 = 8 + 30</td>
<td>3.926</td>
</tr>
<tr>
<td>17 + 3 = 4 + 16</td>
<td>3.898</td>
</tr>
</tbody>
</table>

Table 2. Responses to random independent input vector variables.

<table>
<thead>
<tr>
<th>Input vector</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>58 + 10 &lt; 24 + 54</td>
<td>3.845</td>
</tr>
<tr>
<td>22 + 40 &lt; 37 + 35</td>
<td>3.691</td>
</tr>
<tr>
<td>29 + 50 &lt; 51 + 38</td>
<td>3.740</td>
</tr>
<tr>
<td>22 + 53 &lt; 51 + 34</td>
<td>3.704</td>
</tr>
<tr>
<td>59 + 60 &lt; 36 + 93</td>
<td>3.874</td>
</tr>
<tr>
<td>24 + 39 &lt; 44 + 29</td>
<td>3.745</td>
</tr>
<tr>
<td>36 + 56 &lt; 54 + 48</td>
<td>3.747</td>
</tr>
<tr>
<td>39 + 26 &lt; 56 + 19</td>
<td>3.788</td>
</tr>
<tr>
<td>34 + 10 &lt; 6 + 48</td>
<td>3.756</td>
</tr>
<tr>
<td>21 + 54 &lt; 28 + 57</td>
<td>3.654</td>
</tr>
</tbody>
</table>

The advantage of the D-PNN is its ability to be trained only with small input-output data samples (likewise, the GMDH polynomial neural network does) to learn any dependence. The training data set can consist of the following 6 input vectors, for example:

\[
\{1 + 22 = 20 + 3\}, \{16 + 1 = 2 + 15\}, \{34 + 3 = 30 + 7\}, \{60 + 30 = 42 + 48\}, \{5 + 25 = 2 + 28\}, \{2 + 5 = 4 + 3\}
\]

A separating plane detaching relative “classes” (Tables 1 and 2), which are of the same characteristic can be noticed (if the sum of the 1st couple is less than the 2nd one, the output is less and opposite). The searching space contains a large amount of local error solutions, which DEA can finish easily. This problem is caused by a lot of possible combinations of block inputs and is composed of DE terms (only some of them are employed), whose selection is a critical phase of the D-PNN’s construction, besides the simultaneous parameter adjustment of DEA. The three previously described algorithms will be alternated by D-PNN’s formation as follows:

1. Construction: GA searching of input combinations of blocks (repeat until an improvement is seen).
2. Selection: GA searching of the fittest DE term combination (repeat until improved performance).
3. DEA parameter and weight adjustment (repeat until it reached the desired error value, <).

In the beginning, the study was predominated by the 1st GA of the D-PNN’s construction, while it was subsequently followed by the 2nd GA of the DE term selection and finally, by the adjustment of the DEA. Weights of terms are initialised with value 1.0 and adjusted likewise by the parameters of polynomials in separate cycles of DEA.

CONCLUSION

D-PNN is a new neural network type designed by the author, and can be used to identify any unknown dependencies of data set variables (but not the entire patterns as the ANNs do). Like the brain, it does not utilize absolute values of variables, but relative ones. This identification could be regarded as a pattern of abstraction (or generalization), similar to that utilized by the human brain according to data relations. However, it applies the approximation with time-delayed periodic activation functions of biological neurons in high dynamic system of behaviour (Beňušková, 2002). D-PNN constructs a differential equation, which describes a system of dependent variables, with rational integral polynomial functions. The problem of the partial differential terms’ composition of the multi-layered D-PNN is inherent in the method of creation of all possible combinations (compound fractions) and the realization of a partial derivation (describing a dependence) of some input variables (through fractional or periodic functions).

Relations of some data variables describe a lot of complex systems, in that D-PNN could model their behaviour and utilise the unknown generalized relations of data. DE approximation would better describe the states of a complex system with dependent variables than a common time-series prediction, which is based on pattern similarity. Only linear dependencies of variables have been assumed for simplicity in the examples presented. If there is an occurrence of a non-linear dependence of the input data, it is likely that the square power exponent variables would be attached to combinations of the extended polynomials and applied also as competent derivative terms. A real data example might solve the weather forecast based on some trained data relations, which are used for calculating the next state of a system.

REFERENCES


