Available online at www.sciencedirect.com
ScienceDirect
Expert Systems with Applications xxx (2007) xxx–xxx
Expert Systems with Applications
www.elsevier.com/locate/eswa

Investigating data preprocessing methods for circuit complexity models

P.W. Chandana Prasad [a], Azam Beg [b,*]

[a] *Faculty of Information Systems and Technology, Multimedia University, Malaysia*
[b] *College of Information Technology, United Arab Emirates University, United Arab Emirates*

Abstract

Preprocessing the data is an important step while creating neural network (NN) applications because this step usually has a significant effect on the prediction performance of the model. This paper compares different data processing strategies for NNs for prediction of Boolean function complexity (BFC). We compare NNs' predictive capabilities with (1) no preprocessing (2) scaling the values in different curves based on every curve's own peak and then normalizing to $[0,1]$ range (3) applying z-score to values in all curves and then normalizing to $[0,1]$ range, and (4) logarithmically scaling all curves and then normalizing to $[0,1]$ range. The efficiency of these methods was measured by comparing RMS errors in NN-made BFC predictions for numerous ISCAS benchmark circuits. Logarithmic preprocessing method resulted in the best prediction statistics as compared to other techniques.
© 2007 Published by Elsevier Ltd.

Keywords: Machine learning; Feed-forward neural network; Data preprocessing; Pattern recognition; Boolean function complexity; Computer-aided design

1. Introduction

Complexity of Boolean functions is an important topic in the computation theory. Researchers in the past have tried to classify Boolean functions on the basis of different complexity measures, for example, the minimum size to implement a computing entity (Nemani & Najm, 1996; Priyank, 1997; Wegener, 1987). The way a Boolean function is implemented directly affects the computation and memory resources. Being able to estimate the circuit complexity based on Boolean functions is useful for conducting design feasibility studies (Assi, Prasad, Mills, & El-Chouemi, 2005; Priyank, 1997). Mathematical and NN models have been used in the past for addressing complexity-related problems (Beg, Prasad, & Beg, in press; Dunne & van der Hoeke, 2004; Franco, 2005; Franco & Anthony, 2004; Nemani & Najm, 1996; Prasad, Assi, & Beg, 2006; Ramalingam & Bhanja, 2005; Raseen, Prasad, & Assi, 2005).

NNs are based on the principle of biological neurons. An NN may have one or more input and output neurons as well as one or more (*hidden*) layers of neurons interconnecting the input and output neurons. In the well-known *feed-forward* NNs, the outputs of one layer of neurons send data (only) to the next layer (Beiu, Peperstraete, Vandewalle, & Lauwereins, 1994; Caudill, 1990; Franco, 2006; Parberry, 1994; Shawe-Taylor, Anthony, & Kern, 1992). *Back-propagation* is a common scheme for creating (*training*) the NNs. During the process of NN-creation, internal *weights* of the neurons are iteratively adjusted so that the outputs are produced within desired accuracy (Parberry, 1994).

In order to train the NNs, known examples of input–output datasets are needed. The datasets have to be chosen prudently. Selection and preparation of suitable training data can take up to 80% of the NN development effort (Yale, 1997). Data preparation can vary from simple scaling or range-compression to complex schemes such as polynomial expansion (Tuck, 1993) and Fourier transformation.

Q1 * Corresponding author. Tel.: +971 50 583 6872; fax: +971 3 7626309.
E-mail addresses: m2160062@mmu.edu.my (P.W. Chandana Prasad), abeg@uaeu.ac.ae (A. Beg).

0957-4174/$ - see front matter © 2007 Published by Elsevier Ltd.
doi:10.1016/j.eswa.2007.09.052

The objective of this paper is to present three different methods of data transformation (preprocessing) for use in BFC models. The proposed techniques are generic enough to be used in other NN modeling applications as well. Section 2 of this paper explains the need for data transformation for BFC models. Section 3 describes the transformation techniques. Sections 4 and 5 discuss BFC-NN model, its predictions for ISCAS benchmark circuits, and the conclusions, respectively. Appendix A lists the code for three methods of data transformation.

2. Need for transforming the data

Yale (1997) identifies data transformation as a multi-step process for developing well-designed NNs. Processing of input data has to be done in such a manner that all input variables are given an equally distributed significance. Stated alternately, the inputs with larger absolute values should be given the same importance as the inputs that have smaller magnitudes (Masters, 1994).

We can see the need for data transformation in Table 1 that shows variation of BFC (in terms of nodes) for 2–14 variables. It can also be observed that for 2–6 variables, in their original form, could be hard for a NN to *learn*. The maximum values for the minterms and nodes vary widely and non-linearly. So the smaller variables could be ignored altogether during the NN-training process; data preprocessing alleviates this issue by transforming the curves that have somewhat similar set of minimum and maximum ranges.

3. Data transformation techniques

In this section, we analyze three arbitrarily chosen methods of data transformation that will be useful in creating efficient BFC-NN models: min–max, *z*-score, and logarithmic.

Table 1
Minimum and maximum values for Boolean function complexity curves for 2–14 variables

Variable	Minterm min	Minterm max	Node min	Node max
2	1	7	1	2.53
3	1	16	1	3.68
4	1	36	1	5.27
5	1	54	1	7.96
6	1	93	1	13.11
7	1	156	1	23.77
8	2	248	1	40.25
9	2	392	1	72.2
10	2	650	1	130.38
11	1	969	1.11	243.6
12	1	1597	1	439.73
13	1	2530	1	805.34
14	1	3806	1	1503.24

The minimum and maximum values range widely making it difficult for a NN to model the BFC behavior accurately.

3.1. Min–max transformation

In Table 1, we have seen how widely the minimum and maximum values of lower variable curves vary from the higher variable curves. Using min–max transformation, we first change all curves to one scale, in this case to the 14-variable curve's ranges. Then, we normalize the minterm, node and variable values to the $[0,1]$ range. No min–max transformation was applied to variable values due to their existing linearity and their limited range of 2–14.

$$x'_i = x_i - x_{\min}, \quad i = 0,\ldots,n-1 \tag{1}$$

$$x''_i = x'_i/x'_{\max}, \quad i = 0,\ldots,n-1 \tag{2}$$

Implementation details of min–max transformation and $[0,1]$ normalization of minterm, node and variable values is given by the code in Appendix A.1.

3.2. Z-score transformation

z-score transformation is a statistical technique of specifying the degree of deviation of a data value from the mean. In other words, *z*-score places different types of data on a common scale. *Z*-score is calculated by the following formula (Jeff, 2007):

$$Z = \frac{(x-\bar{x})}{\sigma} \tag{3}$$

where $\bar{x}$ is the sample mean, σ and is the sample standard deviation defined as (Triola, 1994):

$$\sigma = \sqrt{\frac{\sum(x-\bar{x})^2}{n-1}} \tag{4}$$

where n is the sample size.

For data transformation of minterms and nodes, we first apply the *z*-score transformation and then the $[0,1]$-normalization. (As explained in Section 3.1, the variable values were not *z*-score-transformed.) The code for the two data processing steps is given in Appendix A.2.

3.3. Logarithmic transformation

The logarithmic transformation is algorithmically simpler than the two techniques explained in Sections 3.1 and 3.2. Unlike previous procedures, we simply apply a base-10 logarithm to both the minterm and node values. (As discussed in Section 3.1, no log-transformation was applied to variable values.)

$$x'_i = \log_{10}(x_i), \quad i = 0,\ldots,n-1 \tag{5}$$

The $[0,1]$ normalization of minterms, nodes and variables is done in the same manner as before. The transformation-normalization can be performed by the code given in Appendix A.3.

4. NN modeling, results and discussion

We used an NN software package called BrainMaker (version 3.75 for MS-Windows) to model the BFC behavior (BrainMaker, 1998). The software uses fully-connected *feed-forward back-propagation* NNs, meaning all inputs are connected to all hidden neurons, and all hidden neurons are connected to the outputs.

In our NN models (NNMs), the input neuron count is fixed at 2 (one for *minterms* and the other for *variables*) and output neuron count at one (for *node* (BFC) prediction); the NNs comprise of different number of hidden neurons.

We acquired 1186 data sets (also called facts) by running Boolean function simulations (Assi, Prasad, & Beg, 2006). The simulation results were transformed and normalized before being utilized for NN-training. We use 10% of the data sets as the NN *training set* and the remaining 10% as the *validation set*. During NN-training, only the training set was *shown* to the NN, and not the validation set. Random initial weights were used at the beginning of each training session. Each NN-configuration was trained several times to find the best training and validation performance and to reduce the chances of ending up in the local minima.

Application of min–max transformation and [0, 1] normalization on the original data yields the curves shown in Fig. 1. The general shape of the curves stays close to the original. Due to shifted and scaled positions of 2–6 variable curves, we are able to attain better NN-training results. Comparative training and validation statistics for a few NN models are shown in Figs. 2–4 for log, min–max and *z*-score transformation, respectively. Training accuracy refers to the percentage of training data sets that were learnt by the NN with the desired accuracy (i.e., error of 15% or less). Similarly, validation accuracy refers to the percentage of validation data sets tested within the required accuracy limit (i.e., error of 15% or less).

The proximity of training and validation accuracies with log and *z*-score transformations shows that the training process was effective in avoiding over-training (Figs. 2 and 4). In contrast, min–max transformation seems to suffer from over-fitting for hidden neuron of count other than 4 (Fig. 3). Use of *z*-score transformation and normalization gives us the curves that we see in Fig. 5. The shapes of these curves are again somewhat similar to the originals while making their scales also the same. NN-training using the preprocessed data improves NN's predictive accuracy (Fig. 4).

Unlike the first two schemes, the logarithmic transformation changes the shapes of the original curves, while still achieving the goal of bringing their minimums and maximums to much smaller ranges. Fig. 6 shows the effect of

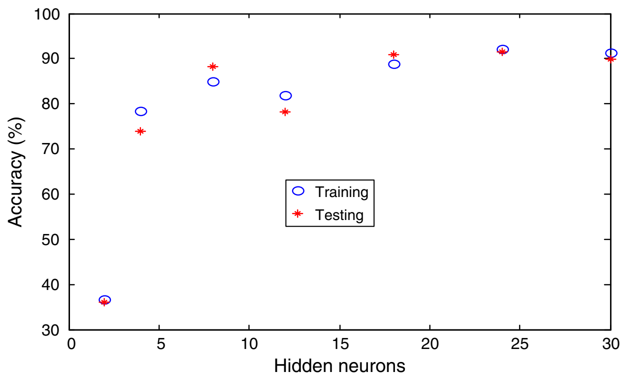

Fig. 2. Training and validation statistics for log transformation.

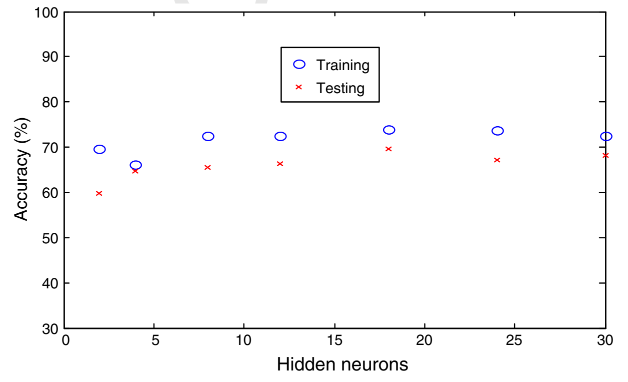

Fig. 3. Training and validation statistics for min–max transformation.

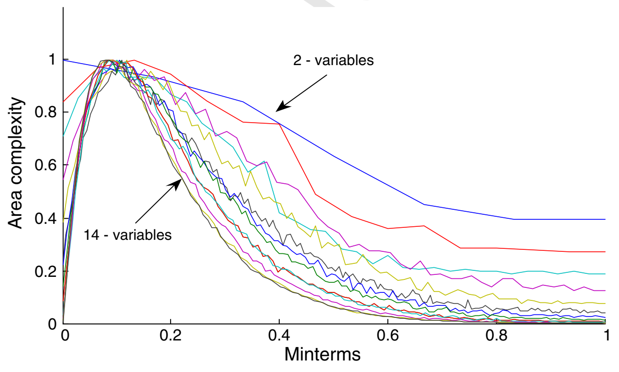

Fig. 1. Effect of min–max transformation and normalization on the original data. The general shape of the curves is retained.

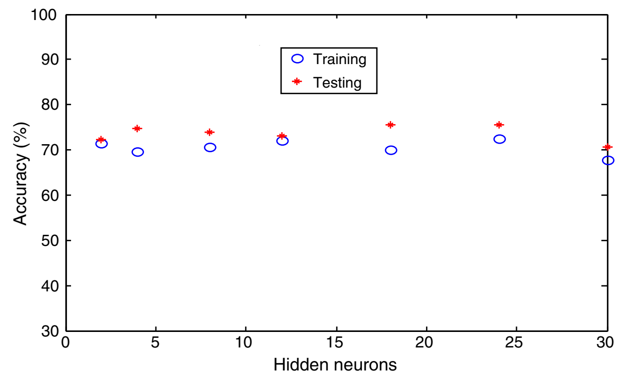

Fig. 4. Training and validation statistics for *z*-score transformation.

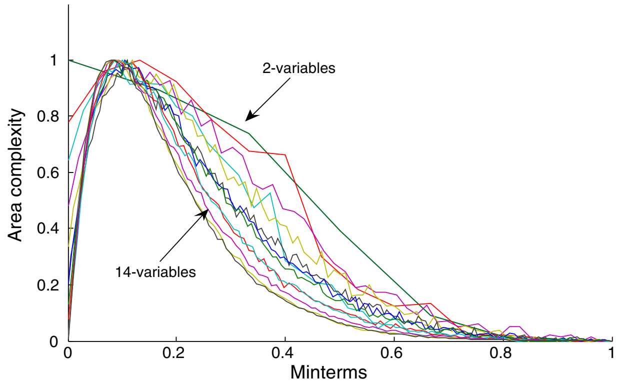

Fig. 5. Effect of *z*-score transformation and normalization on the original data. The overall shape of the original curves stays somewhat closer to the original. Notice the difference in positions of curves between this and Fig. 1.

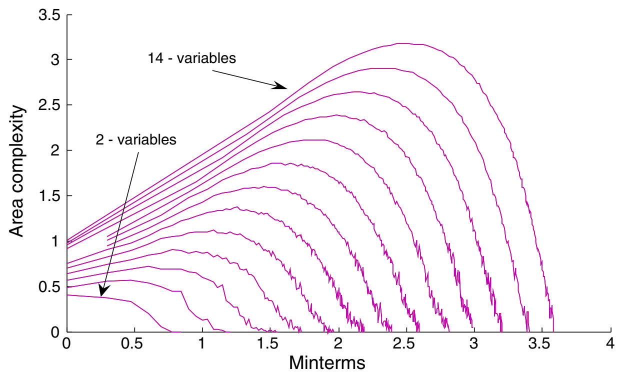

Fig. 6. Effect of logarithmic transformation on minterm and node curves (pre-[0, 1]-normalization). Curves have changed shape while bringing them all closer and making them 'training-visible'.

logarithmic transformation (with no [0, 1] normalization). As compared to training with the raw data, the NN-training results improve in this case as well (Fig. 2).

In some cases, while post-processing NNs' predicted values, the logarithmic processing method may result in lower

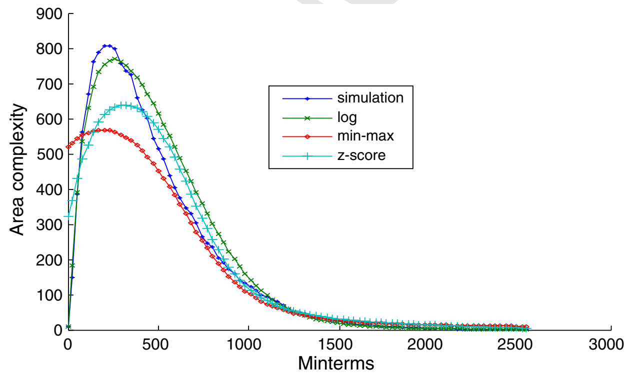

Fig. 7. Comparison of simulations and NN predictions with different data transformation techniques for 11 variables.

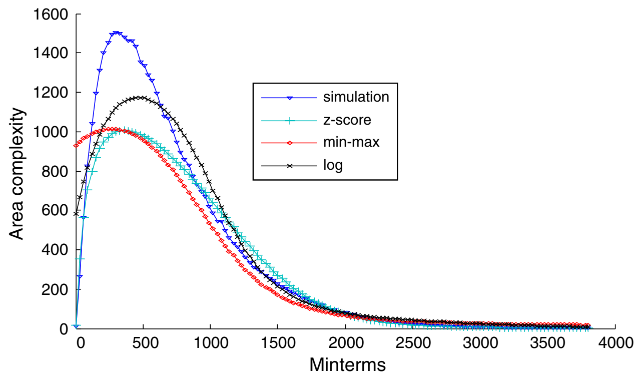

Fig. 8. Comparison of simulations and NN predictions with different data transformation techniques for 14 variables.

Table 2
NNM training and validation accuracies with different data transformation techniques

Preprocessing/ transformation technique	Maximum training accuracy (%)	Maximum validation accuracy (%)
None	42.1	35.3
Log	92.2	91.6
Min–max	74.0	69.7
Z-score	72.4	75.6

accuracy than other two 'non-logarithmic' techniques. (Post-processing is done for restoration of actual ranges; anti-normalization is followed by anti-logarithmization (10^x) of the predicted values.)

The comparison of the actual Colorado University Decision Diagram (CUDD) simulations and NN predictions with different transformation techniques for the 11 and 14 variables are shown in Figs. 7 and 8, respectively. The training and validation accuracies of NNs that made use of transformed data were higher than the NN that learnt from the raw (untransformed) data; a numerical comparison is presented in Table 2. Without data preprocessing, the NN-training and validation accuracies remain very low and the best accuracies are yielded with log transformation.

5. Circuit complexity (BFC) analysis using benchmark circuits

The validated results for data transformation techniques for selected ISCAS benchmark circuits (Brglez & Fujiwara, 1985; Hansen, Yalcin, & Hayes, 1999; Yang, 1991) are compiled in Table 3. The experimental results were obtained on a Pentium-IV machine with 512 MB RAM running on Linux environment. It is well known that running the models is generally faster than simulations, especially when larger circuits are involved (Hossain, Pease, Burns, & Parveen, 2002). Training of our NNMs with the circuit simulation data took a fair amount of time,

Table 3
NNM results for ISCAS benchmark circuits

ISCAS circuit name	Number of input variables	Number of circuits	Area complexity				Relative error		
			CUDD	Neural network models					
				Log	Min–max	*Z*-score	Log	Min–max	*Z*-score
5xp1	7	10	97	94	96	99	−0.03	−0.01	0.02
alu4	14	8	2505	2524	2432	2458	0.01	−0.03	−0.02
apex4	9	19	773	797	686	708	0.03	−0.11	−0.08
apex7	48	55	338	357	507	428	0.06	0.50	0.27
b1	3	4	4	4	3	3	0.19	−0.22	−0.06
b12	15	9	132	120	119	122	−0.09	−0.10	−0.08
b9	41	21	404	480	1186	1185	0.19	1.94	1.93
C17	4	2	10	9	8	9	−0.06	−0.17	−0.04
c8	28	17	165	152	672	486	−0.08	3.08	1.95
cc	21	18	85	78	99	97	−0.08	0.16	0.14
cht	47	36	200	182	166	190	−0.09	−0.17	−0.05
clip	9	5	332	342	279	283	0.03	−0.16	−0.15
cm138a	6	8	89	86	80	86	−0.04	−0.10	−0.04
cm152a	11	8	8	6	165	103	−0.26	19.01	11.44
cm162a	11	6	78	89	314	222	0.14	3.03	1.85
cm163a	9	5	59	53	85	73	−0.10	0.45	0.25
cm82a	5	3	18	14	15	18	−0.20	−0.14	−0.02
cmb	16	4	36	31	1169	724	−0.15	31.61	19.21
con1	6	2	17	16	16	17	−0.04	−0.04	0.01
cu	14	11	57	48	199	198	−0.15	2.47	2.45
decod	5	16	70	63	83	79	−0.09	0.20	0.13
i6	5	67	342	332	323	366	−0.03	−0.05	0.07
i7	6	67	480	445	531	567	−0.07	0.11	0.18
inc	15	57	114	124	150	156	0.08	0.31	0.37
majority	5	1	7	7	6	7	−0.06	−0.16	−0.05
misex1	8	7	94	100	144	143	0.07	0.54	0.53
Pcle	19	9	71	62	796	511	−0.13	10.15	6.16
rd53	5	3	17	15	18	20	−0.07	0.07	0.22
rd73	7	3	39	32	38	43	−0.18	−0.02	0.10
Sao2	10	4	291	285	379	312	−0.02	0.30	0.07
Sct	14	15	172	149	1576	1031	−0.13	8.17	5.00
sqrt8	8	4	19	15	15	17	−0.17	−0.20	−0.06
Squar5	5	8	57	52	50	56	−0.08	−0.12	0.00
ttt2	24	12	273	235	2883	1882	−0.14	9.56	5.89
X2	10	7	49	41	201	138	−0.16	3.13	1.82
x4	94	59	647	575	1965	2174	−0.11	2.04	2.36
z4ml	7	4	75	74	73	75	−0.01	−0.02	0.01
Total circuits		594		RMS error			0.11	6.71	4.11

but it was an up-front, once-only cost; subsequently the NNM would produce the results for various functions with different number of variables and minterms within a few milliseconds or less.

In Table 3, the 1st column indicates the ISCAS benchmark circuit name and the 2nd and 3rd columns are for the maximum number of input variables and number of output circuits for the respective benchmark circuit. In column 4, the actual circuit area complexity for the benchmark circuits have been calculated as nodes in binary decision diagram (Akers, 1978; Bryant, 1991; Drechsler & Sieling, 2001) using CUDD package (Somenzi, 2003). The area complexities (BFC) for all three data transformation techniques were calculated for the number of variables and number of minterms for each respective benchmark circuit. Columns 5–7 list the BFC predictions utilizing NNs that had made use of log, min–max and *z*-score techniques. The RMS relative errors for all three methods compared to the actual benchmark area complexity are given in columns 8–10. The relative errors were computed as the deviation of CUDD's measured (simulated) values from NNM's predicted values

$$\text{Relative error}_i = \frac{(\text{NNM predicted value}_i - \text{CUDD measured value}_i)}{\text{CUDD measured value}_i}, \quad i = 0, \ldots, n-1 \tag{6}$$

$$\text{RMS error} = \sqrt{\frac{\sum_{i=1}^{n} (\text{Relative error}_i)^2}{n}} \tag{7}$$

The deviations for BFC estimation for a complete set of 594 circuits were calculated and the RMS errors for log, min–max and *z*-score transformation techniques were 0.11, 6.71 and 4.11, respectively. It can be concluded from these results, that the log transformation technique produced a closer match with the actual CUDD results.

According to Table 3, the benchmarks tested were up to 94 input variables. The benchmarks circuits mostly consist of the minterms of 1–13 variables. It was observed that most of the benchmarks do not have minterms which can produce the maximum BFC (peak of the respective curve) for the variable of that minterm. Visually, for 11 and 14 variables, the simulated and NN-predicted data (with logarithmic transformation) show differences close to the peaks in Figs. 7 and 8. However, the benchmark circuit comparisons are quite acceptable. This is primarily because the evaluated benchmarks results were clustered well below the peak of each curve. It was also observed that out of 596 circuits tested, only 48 circuits were with 14 variables and 34 circuits with 11 variables, which meant that the data used from the graphs were less than 1%. Therefore, we can conclude that Q2 the rising edge of the graphs for any variables (Fig. 7 and 8) are only important for the validation of benchmarks for these methods. This can be the main reason behind the low RMS error for log transformation. It is obvious that importance of a full-scale match of the curves will be more difficult to justify because of the lack of sample minterms that can be extracted from the benchmarks.

6. Conclusions

The data preprocessing (transformation) techniques for NNMs presented in this paper exhibit varying degree of effectiveness for the purpose of BFC modeling. We compared the performance of various NNMs for large sets of Monte Carlo data for different number of variables and minterms up to 14 variables. Without making use of data transformation, the NNM prediction accuracy remained unacceptably low at approximately 42%; the min–max and *z*-score transformations improved these statistics to 74% and 72%, respectively. The logarithmic transformation yielded the best prediction capability with accuracy of more than 92%. The effectiveness of logarithmic transformation was also justified by the RMS error of 0.11 achieved for ISCAS benchmarks in comparison to 6.71 and 4.11 for min–max and *z*-score transformations, respectively. The proposed transformation methods or their variations can be helpful in developing robustly working NNs for other practical applications. We are currently exploring the extension of this work to identify the extrapolative behavior of the neural methods for circuits of more than 14 variables.

Appendix A.

This section lists the code required for data transformation specific to NNs for BFC modeling.

A.1. Min–max transformation

```
//-------------------------------------------
// Each data points is represented by 3
// values: number of variables, number of
// minterms (MT) and nodes (BFC).

// Original values for node complexity for
// each variable are stored in their
// individual array named Node_orig[2][n],
// Node_orig[3][m], … Node_orig[14][p]
// and the scaled values are in
// Node_scaled[][] array. Final normalized
// array is Node_norm[][]

// Original values for minterms for
// each variable are stored in their
// individual array named MT_orig[2][n],
// MT_orig[3][m], … MT_orig[14][p]
// and the scaled values are in
// MT_scaled[][] array. Final normalized
// array is MT_norm[][]

// PtCount[] represents points in each curve.
// For example, there are 7 points in 2-var-
// able curve so PtCount[2]=7; there are 16
// points in 2-variable curve so PtCount[3]=16
// and so on

// MTmax[v] represents the maximum MT value
// for variable v:

// MTmaxAll = max(MTmax[v]), v = 2 .. 14

// Nodemax[v] represents the maximum node
// value for variable v:
// NodemaxAll = max(Nodemax[v]), v = 2 .. 14

// Find minterm and node scaling factors
// (MT-SF[v], Node_SF[v]) for v = 2 .. 14
for (v=2; v<=14; v++) {
    MT_SF[v] = MTmaxAll/MTmax[v];
    Node_SF[v] = ModeMaxAll/ModeMax[v];
}
// Scale the curves for all variables
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {
    // Scale MT array
MT_scaled[v][p] =
   MT_orig[v][p] * MT_SF[v];

// Scale node array
Node_scaled[v][p] =
   Node_orig[v][p] * Node_SF[v]
}
}
// Normalize the MT, node and var arrays to
// [0,1] ranges
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {

// MT_scaled_min = min(MT_scaled[]) MT_norm_a[v][p] =
   MT_scaled[v][p] - MT_scaled_min;

// Node_scaled_min = min(Node_scaled[])
Node_norm_a[v][p] =
   Node_orig[v][p] - Node_scaled_min;
} // end of for-p

// var_min = min(var_orig[]) = 2
var_norm_a[v] = var_orig[v] - varmin
} // end of for-v
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {

// MT_scaled_max = max(MT_norm_a[])
MT_norm[v][p] =
   MT_norm_a[v][p]/MT_scaled_max;

// Node_scaled_max = max(Node_norm_a[])
Node_norm[v][p] =
   Node_norm_a[v][p]/Node_scaled_max;
// end of for-p

// var_max = max(var_norm_a[])
var_norm[v] = var_norm_a[v]/varmax
} // end of for-v
//-------------------------------------------
```

A.2. Z-Score Transformation

```
//-------------------------------------------
// Most variable definitions are the same as //
in Section A.1. Only the new variables are //
explained here.
// avgMT = average MT for all variables
// stdvMT = std-dev of MT for all variables
// avgNode = average node val for all
//          variables
// stdvNode = std-dev of for all variables

// Z-score minterm and node values for all
// variables
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {
    // Scale MT array

MT_scaled[v][p] =
(MT_orig[v][p] - avgMT)/stdvMT ;

// Scale node array
Node_scaled[v][p] = (Node_orig[v][p] - avgNode)/stdvNode
}
}

// Normalize the MT, node and var arrays to
// [0,1] ranges
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {

// MT_scaled_min = min(MT_scaled[])
MT_norm_a[v][p] =
   MT_scaled[v][p] - MT_scaled_min;

// Node_scaled_min=min(Node_scaled[])
Node_norm_a[v][p] =
   Node_orig[v][p] - Node_scaled_min;
} // end of for-p

// var_min = min(var_orig[]) = 2
var_norm_a[v] = var_orig[v] - varmin
} // end of for-v
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {

// MT_scaled_max = max(MT_norm_a[])
MT_norm[v][p] =
   MT_norm_a[v][p]/MT_scaled_max;

// Node_scaled_max = max(Node_norm_a[])
Node_norm[v][p] =
   Node_norm_a[v][p]/Node_scaled_max;
// end of for-p

// var_max = max(var_norm_a[])
var_norm[v] = var_norm_a[v]/varmax
} // end of for-v
//-------------------------------------------
```

A.3 Logarithmic Transformation

```
//-------------------------------------------
// Variable definitions are the same as in
// Section A.1

// Z-score minterm and node values for all
// variables
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {
    // Scale MT array
MT_scaled[v][p] =
   log10(MT_orig[v][p]);

// Scale node array
Node_scaled[v][p] =
   log10(Node_orig[v][p]);
}
}
// Normalize the MT, node and var arrays to
// [0,1] ranges
for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {
```

```
// MT_scaled_min = min(MT_scaled[])
MT_norm_a[v][p] =
   MT_scaled[v][p] - MT_scaled_min;

// Node_scaled_min=min(Node_scaled[])
Node_norm_a[v][p] =
   Node_orig[v][p] - Node_scaled_min;
} // end of for-p
// var_min = min(var_orig[]) = 2
var_norm_a[v] = var_orig[v] - varmin
} // end of for-v

for (v=2; v<=14; v++) {
    for (p=0; p<PtCount[v]; p++) {

// MT_scaled_max = max(MT_norm_a[])
MT_norm[v][p] =
   MT_norm_a[v][p]/MT_scaled_max;
// Node_scaled_max = max(Node_norm_a[])
Node_norm[v][p] =
   Node_norm_a[v][p]/Node_scaled_max;
// end of for-p

// var_max = max(var_norm_a[])
var_norm[v] = var_norm_a[v]/varmax
} // end of for-v
//-------------------------------------------
```

References

Akers, S. B. (1978). Binary decision diagram. *IEEE Transaction on Computers, 27*, 509–516.

Assi, A., Prasad, P. W. C., & Beg, A. (2006). Modeling the complexity of digital circuits using neural networks. *WSEAS Transacation on Circuits and Systems.* Q3

Assi, A., Prasad, P. W. C., Mills, B., & El-Chouemi, A. (2005). Empirical analysis and mathematical representation of the path length complexity in binary decision diagrams. *Journal of Computer Science, 2*(3), 236–244.

Beg, A., Prasad, P. W. C., & Beg, A. (in press). Applicability of feedforward and recurrent neural networks to boolean function complexity modeling. *Expert Systems with Applications.* Q4

Beiu, V., Peperstraete, J. A., Vandewalle, J., & Lauwereins, R. (1994). Placing feedforward neural networks among several circuit complexity classes. In *Proceedings of the WCNN'94* (pp. 584–589).

BrainMaker. (1998). *User's guide and reference manual* (7th ed.). California Scientific Software Press. Q5

Brglez, F., & Fujiwara, H. (1985). A Neutral netlist of 10 combinational circuits and a target translator in fortran. In *Proceedings of the international symposium on circuit and systems, special session on ATPG and fault simulation* (pp. 663–6985).

Bryant, R. E. (1991). On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers, 40*, 203–213.

Caudill, M. (1990). *AI expert: Neural network primer*. Miller Freeman Publications.

Drechsler, R., & Sieling, D. (2001). *Binary decision diagrams in theory and practice*. Springer-Verlag Transaction, pp. 112–136.

Dunne, P. E., & van der Hoeke, W. (2004). Representation and complexity in Boolean games. In *Proceedings of the nineth European conference on logics in artificial intelligence. LNCS 3229* (pp. 347–35).

Franco, L. (2005). Role of function complexity and network size in the generalization ability of feedforward networks. Lecture notes in computer science. computational intelligence and bioinspired systems. In *Proceedings of the eighth international workshop on artificial neural networks* (pp. 1–8).

Franco, L. (2006). Generalization ability of Boolean functions implemented in feedforward neural networks. *Neurocomputing, 70*, 351–361.

Franco, L., & Anthony, M. (2004). On a generalization complexity measure for Boolean functions. In *Proceedings of the IEEE international joint conference on neural networks. IJCNN 2004* (pp. 973–978).

Hansen, M., Yalcin, H., & Hayes, J. P. (1999). Unveiling the ISCAS-85 Benchmarks: A case study in reverse engineering. *IEEE Transaction on Design and Test, 16*, 72–80.

Hossain, A., Pease, D. J., Burns, J. S., & Parveen, N. (2002). Trace cache performance parameters. In *Proceedings of the IEEE international conference on computer design, VLSI in computers and processors* (pp. 348–355).

Jeff. (2007). What's a *Z*-score and why use it in usability testing? <http://www.measuringusability.com/z.htm>.

Masters, T. (1994). *Signal and image processing with neural networks*. John Wiley & Sons, Inc.

Nemani, M., & Najm, F. N. (1996). High-level power estimation and the area complexity of Boolean functions. In *Proceedings of the IEEE international symposium on low power electronics and design* (pp. 329–334). Q6

Parberry, I. (1994). *Circuit complexity and neural networks*. MIT Press.

Prasad, P. W. C., Assi, A., & Beg, A. (2006). Binary decision diagrams and neural networks. *Journal of Supercomputing, 2*(2), 141–147.

Priyank, K. (1997). VLSI logic test, validation and verification, properties & applications of binary decision diagrams. Lecture Notes-department of Electrical and Computer Engineering University of Utah, Salt Lake City. UT 84112.

Ramalingam, N., & Bhanja, S. (2005). Causal probabilistic input dependency learning for switching model in VLSI circuits. In *Proceedings of the ACM great lakes symposium on VLSI* (pp. 112–115).

Raseen, M., Prasad, P. W. C., & Assi, A. (2005). An efficient estimation of the ROBDD's complexity. *Integration – VLSI Journal, 39*(3), 211–228.

Shawe-Taylor, J. S., Anthony, M. H. G., & Kern, W. (1992). Classes of feedforward neural networks and their circuit complexity. *Neural Networks, 5*, 971–977.

Somenzi, F. (2003). CUDD: CU decision diagram package. <ftp://vlsi.colorado.edu/pub/>.

Triola, M. (1994). *Elementary Stastictics* (6th ed.). Addison-Wesley Publishing Co.

Tuck, D. L. (1993). Practical polynomial expansion of input data can improve neurocomputing results. In Proceedings of the artificial neural networks and expert systems (pp. 42–45).

Wegener, I. (1987). *The complexity of Boolean functions*. Wiley and Sons, Inc.

Yale, K. (1997). Preparing the right data for training neural networks. *IEEE Spectrum, 34*(3), 64–66.

Yang, S. (1991). *Logic synthesis and optimization benchmarks user guide version 3.0*. Technical Report. Microelectronic Center of North Carolina Research Triangle Park.