Data Processing for Effective Modeling of Circuit Behavior

AZAM BEG and P. W. C. PRASAD
College of Information Technology
United Arab Emirates University,
UAE
{abeg, prasadc}@uaeu.ac.ae

Abstract: - Data transformation is an important step in developing practical and robust neural networks and can take a relatively large percentage of development efforts. In this paper, we present different techniques and their algorithms for data transformation as they apply to the neural network models for predicting Boolean function complexity. The data transformation techniques proposed in this paper yield a high level of model accuracy. The given techniques can also be applied to neural networks developed for other applications.

Key-Words: - Boolean function complexity, Neural network, Modeling, Data transformation

1 Introduction

Complexity of Boolean functions is an important topic in the computation theory. Researchers have in the past tried to classify Boolean functions on the basis of different complexity measures, for example, the minimum size to implement a computing entity [1]. 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 [2]. Mathematical and neural network (NN) models have been used in the past for addressing complexity-related problems [3][4][5].

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. *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 [7].

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 [8]. Data preparation can vary from simple scaling or range-compression to complex schemes such as polynomial expansion [9] and Fourier transformation.

The objective of this paper is to present three different methods of data transformation for use in Boolean function complexity (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 and their corresponding code snippets. Section 4 discusses the results and Section 5 presents the conclusions. Appendix A lists the code for three methods of data transformation.

2. Need for Transforming the Data

Yale [8] 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 [10].

We can see the need for data transformation in Fig. 1 that shows BFC curves for 2- to 14-variables (The plotted data was acquired from Boolean function simulations [5].). Number for *'minterms'* in a function is shown on the horizontal axis; on the vertical axis, *'nodes'* represents the complexity of a Boolean function. The curves for 2-6 variables, in their original form, are not only visually hard to see but also hard for a NN to *learn*. The minimum and maximum values on both axes of these curves vary widely and non-linearly, as listed in Table 1.

So the smaller variable curves could be ignored altogether during the NN-training process; data processing alleviates this issue by transforming the curves that have a similar set of minimum and maximum ranges.

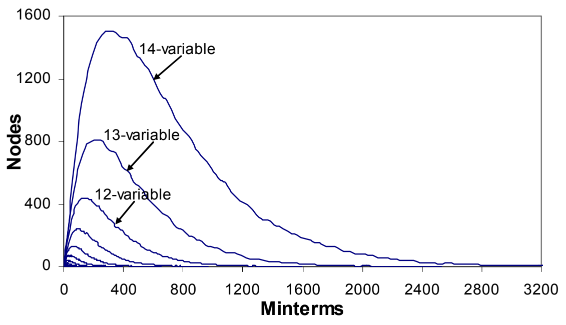

Fig. 1: Boolean function complexity data for 2- to 14-variables in original (raw) format. The smaller variable curves (lower left corner) are not as visible as the large ones and have the potential of not being correctly learnt by the NN.

Table 1: Minimum and maximum values for Boolean function complexity curves for 2- to 14-variables. The minimum and maximum values range widely making it difficult for a NN to model the BFC behavior accurately.

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

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.

3.1 Min-Max Transformation

In Fig. 1 and 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 minterms, 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 to 14.

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

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

The complete algorithm for min-max transformation and [0,1]-normalization of minterms, nodes and variable values is given by the code snippet in Appendix A.1.

3.2 Z-Score Transformation

Z-score normalization 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 [11]:

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

where $\bar{x}$ is the sample mean, and σ is the sample standard deviation defined as [12]:

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

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 tends to be 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.)

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

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

4. Results and Discussion

We used an NN software package called BrainMaker (version 3.75 for MS-Windows) to model the BFC behavior [13]. 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, the input neuron count is fixed at 2 (one for 'minterms' and the other for 'variables') and output neuron count at one (for 'node' (complexity) prediction); the NNs comprise of different number of hidden neurons.

We acquired 1186 data sets (also called facts) by running Boolean function simulations [5]. The simulation results have been 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 is presented to the NN, and not the validation set.

The application of minimum-maximum transformation and [0,1]-normalization on the original data yields the curves shown in Fig. 2. 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 Table 2. Training accuracy refers to the percentage of training data sets that were modeled by the NN with the desired accuracy. Similarly, validation accuracy refers to the percentage of validation data sets tested within the required accuracy limit.

Use of Z-scale transformation and normalization gives us the curves that we see in Fig. 3. The shapes of these curves are again somewhat similar to the originals while making their scales also the same. The accuracy of NN training using the processed data provides us with better accuracy (Table 2).

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. 4 shows the effect of logarithmic-transformation (with no [0,1] normalization). As compared to raw data, the NN-training results improve in this case also (Table 2).

In some cases, while post-processing the NNs' predicted values, the logarithmic processing method may result in lower accuracy than other two 'non-logarithmic' techniques. (Post-processing is done for restoration of actual ranges: anti-normalization followed by anti-logarithm (*10^x*) of the predicted values.)

We observed that the training and validation accuracy of NNs that made use of transformed data were higher than the NN that learnt from the raw (untransformed) data (94-99% vs. 90%). The comparison of the actual simulations and NN-predictions for the 6-variable case (utilizing log-transformation) is shown in Fig 5.

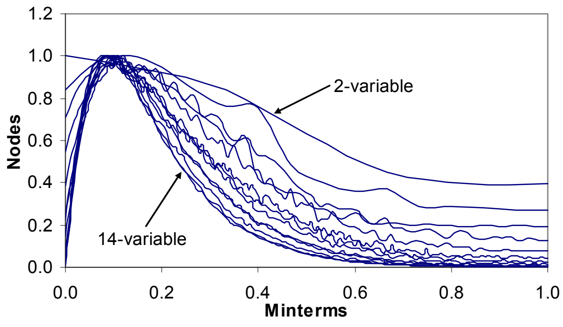

Fig. 2: Effect of min-max transformation and normalization on the original data. The general shape of the curves is retained. Notice the difference in positions of curves between this and Fig. 1.

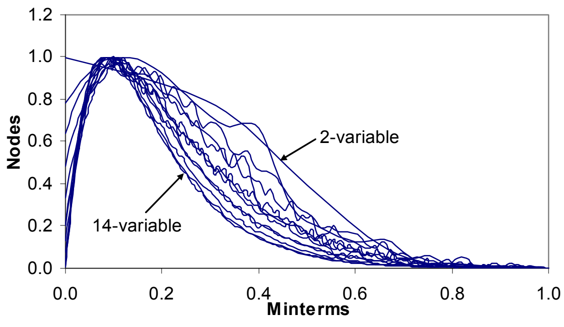

Fig. 3: Effect of Z-score transformation and normalization on the original data. The overall shape of the original curves stays somewhat closer to the original. Notice the difference in positions of curves between this and Fig. 1.

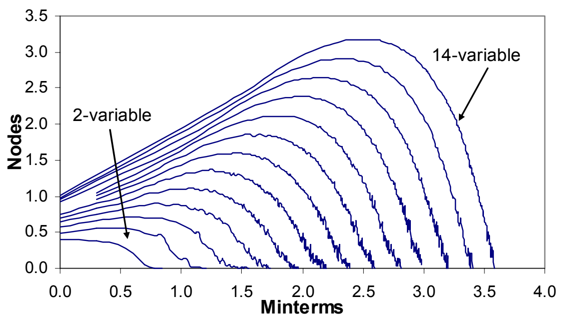

Fig. 4: Effect of logarithmic transformation on minterm and node curves (pre-[0,1]-normalization). Curves have changed shape while bringing them all closer and making them 'training-visible'. The order of the curves (i.e., 2-variable on bottom and 14-variable on top) is the same as the originals in Fig. 1.

Table 2: Training and validation statistics for a few NN models. NNs that used transformed data were found to perform generally better than the NNs using raw data.

Pre-Processing Technique →	**None**	**Linear**	**Z-score**	**Log10**
Input Layer Neurons	2	2	2	2
Hidden Layer Neurons	21	21	21	21
Output Neurons	1	1	1	1
Training Accuracy	90.8%	99.5%	94.1%	99.9%
Validation Accuracy	90.8%	100.0%	95.8%	100.0%

BrainMaker parameters: test/training tolerance = 0.1; stop training when average error <= 2.5%; activation function = sigmoid. The accuracy is dependent on initial neuron weights and the former can vary from one training session to the other.

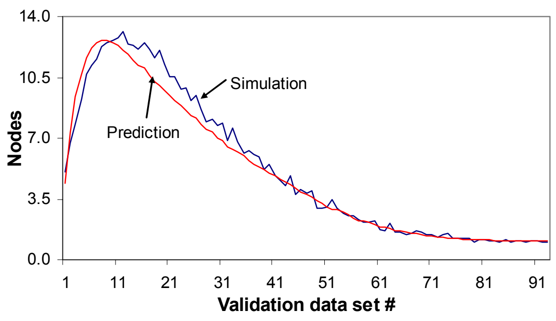

Fig. 5: Comparison of simulations and NN-predicted results for 6-variables (using log-transformation and [0,1]-normalization). There is a close match between the two curves showing the effectiveness of the NN model.

5. Conclusion

The data transformation techniques for NN-training presented in this paper exhibit their usefulness for the purpose of BFC modeling. Training accuracies of 94% to 99% were achieved with transformed data as compared to the raw data that gave only 90% accuracy. The proposed transformation methods or their variations can be helpful in developing robustly working NNs for other practical applications.

Appendix A

This section lists the code required for data transformation specific to NNs for BDD complexity 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 0. 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-scale 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 0

// Z-scale 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

[1] M. Nemani, and F.N. Najm, “High-level power estimation and the area complexity of Boolean functions,” *Proc. of IEEE Intl. Symp. on Low Power Electronics and Design,* 1996, pp. 329-334.

[2] A. Assi, P.W. C. Prasad , B. Mills, and A. El-Chouemi, “Empirical Analysis and Mathematical Representation of the Path Length Complexity in Binary Decision Diagrams”, in *Journal of Computer Science, Science Publications*, Vol. 2(3), 2005, pp. 236-244.

[3] L. Franco, M. Anthony, "On a generalization complexity measure for Boolean functions", IEEE Conference on Neural Networks, *Proceedings, v 2, 2004 IEEE International Joint Conference on Neural Networks – Proceedings,* 2004, pp. 973-978.

[4] L. Franco, "Role of function complexity and network size in the generalization ability of feedforward networks", Lecture Notes in Computer Science, v 3512, Computational Intelligence and Bioinspired Systems: *8th International Workshop on Artificial Neural Networks*, IWANN 2005, Proceedings, 2005. pp. 1-8.

[5] A. Assi, P. W. Chandana Prasad, and A. Beg, Modeling the Complexity of Digital Circuits Using Neural Networks, *WSEAS Transactions on Circuits and Systems*, June 2006.

[6] L. Franco, Generalization ability of Boolean functions implemented in feed forward neural networks. *Neurocomputing.* Vol. 70, pp. 351-361.

[7] M. Caudill, *AI Expert: Neural Network Primer*, Miller Freeman Publications, 1990.

[8] K. Yale, “Preparing the right data for training neural networks,” *IEEE Spectrum*, Vol. 34, Issue 3, Mar. 1997, pp. 64-66.

[9] D.L. Tuck, "Practical polynomial expansion of input data can improve neurocomputing results", *ANNES'93*, Los Alamitos, CA, 1993, pp. 42-45.

[10] T. Masters, *Signal and Image Processing with Neural Networks*” John Wiley & Sons, Inc., 1994.

[11] "What's a Z-Score and Why Use it in Usability Testing?" http://www.measuringusability.com/z.htm, 2007

[12] M. Triola, *Elementary Stastictics*, 6th ed, Addison-Wesley Publishing Co, 1994.

[13] *BrainMaker – User’s Guide and Reference Manual*, 7th ed., California Scientific Software Press, 1998.