Reprinted from (M. Egmont-Petersen. “Homomorphic transformation from neural networks to rule bases,” In: E. Mosekilde (Eds.), Proceedings of the European Simulation Multiconference 1991, pp. 260-265, 1991).

For more information, see the Homepage of Michael Egmont-Petersen:
http://www.cs.uu.nl/people/michael

Comments and questions can be sent to: michael@cs.uu.nl

Homomorphic transformation from Neural Networks to rule bases

Michael Egmont-Petersen, Industrial Ph.D.-student
Institute of Computer and Systems Sciences
Copenhagen Business School
Jul. Th. Plads 10,
DK-1925 Frb. C.
Denmark
E-mail: michael@dasy.cbs.dk

Abstract: In this article a method to extract the knowledge induced in a neural network is presented. The method explicates the relation between a network's inputs and its outputs. This relation is stored as logic rules. The feasibility of the method is studied by means of three test examples. The result is that the method can be used, though some drawbacks are detected. One is that the method sometimes generates a lot of rules. For fast retrieval, these rules can well be stored in a B-tree.

1. INTRODUCTION

Neural networks is a simulation tool which have been used in a variety of tasks during the recent years. They have been trained to classify patterns from many different domains (Confer e.g. Eberhart & Tzanakou 1990). Their ability to recognize or associate patterns has been an example of how to "reason" within fuzzy domains. In spite of these abilities, Neural Networks have not been utilized practically as much as they could because they have some drawbacks. One is the way in which knowledge is represented in a network. To overcome this difficulty (as well as others) researchers have tried to link neural networks and symbolic systems together to utilize the benefits of each. In one article (Myllymäki et.al. 1990) it is described how neural networks can be used to link different parts (slots) of a frame-based system. This hybrid system can associate mutually relevant information. Neural networks have also been used to add procedural knowledge to a rule-based system (Gutknecht & Pfeifer 1990). But the work of (Bochereau & Bourgine 1990) is mostly related to the experiments outlined in this article. In their paper a method for explicating the knowledge induced in neural networks is discussed. The strategy pursued was to extract rules from a trained network manually. These rules took form as inequalities.

Here, some experiments done to extract rules from neural networks are presented. First, a training set had been established. This set was generated on the basis of a set of known logic rules stored in a prolog program. This program generated the complete truth table related to each rule. Each truth table was used as the training set for a series of networks. Networks with a different number of hidden neurons were trained with the same truth table in order to discover whether increasing the number of hidden neurons degenerates the networks' abilities to generalize. Moreover, each of these different network configurations were trained with different fractions of the training sets.

After the networks have been trained, a rule extraction algorithm was applied. Its output comprised a set of rules which, if fulfilled, result in the output *true*. Each set of rules was merged into a decision tree. The quality of these trees is then analyzed. Thus, the purpose of the work presented here is to discover whether the rule extracting approach is feasible or not.

2. SYNTHESIZING RULE BASES PARSIMONIOUSLY

The rules used in the experiments were based on a

three-valued logic, that is, it contains: "True", "False" and "Don't know". When a rule was evaluated, its premises were bound to a configuration within the "truth-space". A neural network was then trained with this three valued truth table for a number of cycles by means of the traditional Backpropagation algorithm (this was based on the hyperbolic tangent function). The output of the network fell into the space]-1,1[, but actually only 0 and 1 were used as training outputs. In this way, *false* was designated as a 0 and *true* was designated as a 1. When the premises of a rule were fulfilled, its output was, thence, *true*.

During training, the weights were arranged in a configuration which minimized the squared error. Afterwards, these weights were loaded into the algorithm which performed a synthesis of the network's response to a large number of questions.

The rule-extraction algorithm applied afterwards is rather straight forward. It combines the *entropy* measure (known from information theory and used in the ID3 algorithm) with a recursive algorithm capable of proposing "questions" to the trained network. The basic idea is as follows: First, the entropy related to each of the premises with respect to classifying the output correctly, is calculated. The premises are, then, sorted in ascending order with respect to their entropy. Afterwards, the recursive algorithm is applied. This algorithm binds combinations of premises which are clamped onto the neural network's input side. If the result of a certain combination of premises is e.g. *true* then the least significant premise is changed and this, slightly changed, combination of premises is then clamped onto the network. (The least significan premise is the one with the highest entropy). If the network's output is approximately the same (in fact above a certain threshold, here ½), then the premise is changed again. This way, it takes three trials to evaluate the influence of each premise i.e. *True*, *False*, and *Don't know*. If all three combinations results in the same classification the premise is taken to be a *Don't care* premise combined with the values held by the rest of the premises. Hence, the complexity of the rule tree is reduced by one level. The following can well serve as an example (In the notion used the comma ',' serve as a logical *and*). If the algorithm has been called with the combination of premises:

D=true,	*A=true,*	*B=false,*	*C=true,*	*(E=?)*
0.51	0.56	0.89	0.91	0.94

and the network gives an output above e.g. 0.5 for *E* being both *true*, *dont_know*, and *false*, then the last part of the hypothesized rule is left out due to its status as a *don't-care* premise (The number below each premise is the entropy of each premise). Hence, the hypothesized rule is now:

D=true, A=true, B=false, C=true

The same procedure is continued until a change of one of the premises makes a significant change in the output of the network. Thus, if the change of e.g. *B* from *false* to *dont_know* draws the output beneath the threshold, a boundary has been found. Hereafter, the hypothesis:

D=true, A=true, B=false

is stored as a rule resulting in a decision which belongs to the class *true*.

The entropy measure is used to conduct the sequence in which the premises are changed. In the example shown, *D* had the lowest entropy and was, therefore, the last premise to be changed e.g. from *D=true* to *D=dont_know*.

3. DESCRIPTION OF THE EXPERIMENTS

The experiments done cover three different training sets i.e. three rules. These are:

1

A=true, B=true *or*
C=true, A=false *or*
C=true, A=dont_know

2

C=dont_know, A=true , B=true *or*
C=false , A=true , B=dont_know *or*
C=false , A=dont_know, B=false *or*
C=dont_know, D=true

3

A=true *or*
A=dont_know, B=true *or*
A=dont_know, B=dont_know, C=true *or*
A=dont_know, B=dont_know, C=dont_know, D=true

or

A=dont_know, B=dont_know, C=dont_know, D=dont_know, E=true.

The truth space spanned out by each rule was $3^5=243$ possible combinations of *true*, *dont_know*, and *false*. The prolog program generated the four complete truth tables by evaluating the expressions above. 5 premises were included in each network even though some premises were redundant in relation to a rule. Networks were then trained with these truth tables. In the truth tables e.g. premise *A* was represented by three values ($P_{A,1}$, $P_{A,2}$, $P_{A,3}$) i.e. one of these were clamped to 1 and the two others to 0 when the premise was either *true*, *false*, or *dont_know*. For each truth table, three different network sizes were applied: 15-7-1, 15-9-1, and 15-11-1, where the first number stands for the amount of input neurons, the second the amount of hidden etc. Each network configuration was trained three times, each with a different fraction of the training set picked out randomly: 100%, 80%, and 50%. This gives 3*3=9 simulations carried out on each truth table.

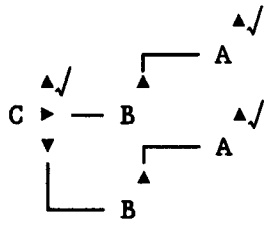
The training itself were proceeded either until the squared derivations (TSS) were less than 5 or until a network has been trained for 300 epochs. Hence, the networks were not trained for thousands and thousands of epochs because just a few well converged networks could suffice.

After the networks have been trained, the weights and biases together with the training set, are fed into the rule-extraction program. This program first calculates the entropy of each premise in the fraction of the truth table used to train each network. Thus, if 50% of the training cases are rejected from a simulation, the same cases are excluded from the entropy calculation.

The results have not been analyzed statistically. This is due both to the complexity of the rule base generated by the extraction algorithm and to the sometimes small changes in the rules bases when e.g. the network size was increased.

3.1. First Experiment

In this experiment rule (1) has been used to generate the training set. Three different network topologies have been trained with respectively 100%, 80%, and 50% of the training set. The results are promising, though some enhancements to the algorithm are warranted. The correct tree is shown in figure 1 (The notation used in the trees is as follows: '▲' means *true*, '►' *dont_know*, '▼' *false*, and '✓' symbolizes a leave which leads to a *true* conclusion). In figure 2 a tree generated by the extraction algorithm is shown. The network which generated the rules was trained with only 80% of the cases. At first view, the figure 2 tree seems more complicated than the reference tree shown in figure 1. Though, the rules displayed in the bottom of figure 2 can be reduced to the rules presented in figure 1. The difference is due to the dissimilar samples used to calculate the entropy of the premises. Hence, premise D has a smaller entropy than A in the simulation when only 80% of the cases were presented. This causes the algorithm to call D before A while generating the rules. The point of interest is that the network in this situation is more noise resistant than the entropy measure given the way it is applied here.



Rules:

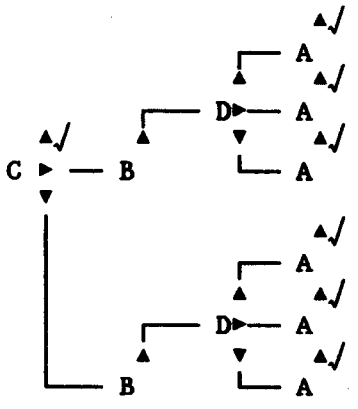
C=false, B=true, A=true *or*
C=dont_know, B=true, A=true *or*
C=true

Figure 1. This decision tree is the reference tree used in first experiment.

When only 50% of the training set is ued to train the 15-7-1 network the tree shown in figure 3 is generated. This tree is a specialization of the tree shown in figure 2 because the lower part of the figure 3 tree (containing C=*dont_know* and C=*true*) is identical to the tree in figure 2. Only the top branch differs. The

interesting point is that when a larger network is used to learn the patterns (with nine instead of seven hidden neurons), the correct tree is generated. Thus, the tree shown in figure 2 is identical to the one extracted from the network with 9 neurons, even though this had only seen 50% of the patterns (this tree is not shown). The only difference of this tree is the order of the premises. This different order, in fact, gives a more complicated tree. This indicates that the neural network is more resistant to noise than the entropy measure. Others have shown that nonlinear neural networks are more noise resistant than the ID3 algorithm (Shavlik et.al. 1990).

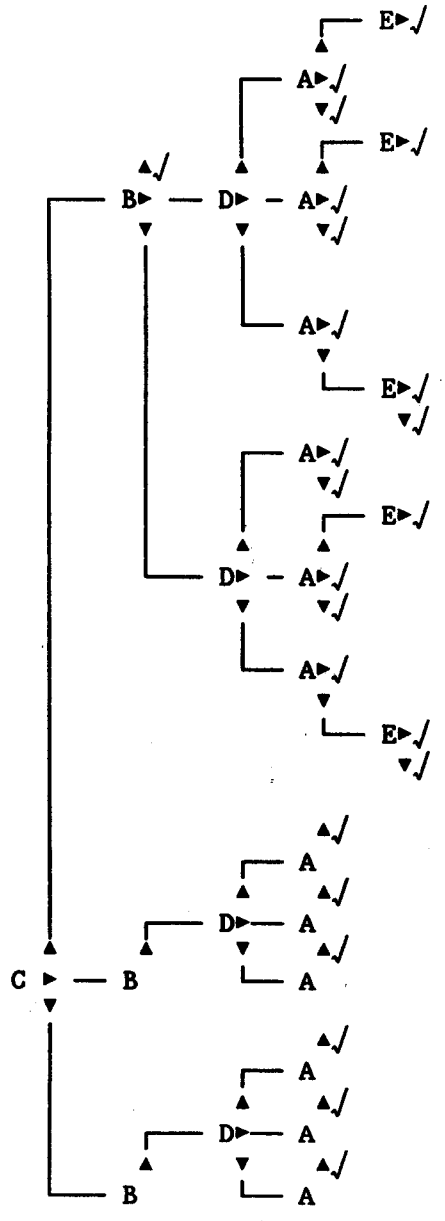
The conclusion of the first experiment is that the rule extraction algorithm seems to be applicable but the entropy measure is not as robust as warranted. In this way, the trees generated may have redundant branches.



Rules:

C=false, B=true, D=true, A=true *or*
C=false, B=true, D=dont_know, A=true *or*
C=false, B=true, D=false, A=true *or*
C=dont_know, B=true, D=true, A=true *or*
C=dont_know, B=true, D=dont_know, A=true *or*
C=dont_know, B=true, D=false, A=true *or*
C=true

Figure 2. This tree is a variant of the tree pictured in figure 1. At first aquaintance it looks more complicated but it is actually identical to the first tree.



Rules:
(not showed)

Figure 3. This tree is extracted from a network which is trained with 50% of the cases from rule (1).

3.2. Second Experiment

In this experiment rule (2) was used to generate the training set. The networks trained did not learn these patterns as easily as the patterns from the first experiment. Thus, the number of training cycles used to make

the network converge was high (agv. over 300).

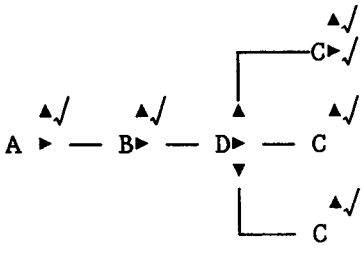
The number of rules extracted from the networks trained with the second training set was immense compared to their source (Therefore, none of the trees are shown here). This is due to the entropy measure. This measure positions premise *C* in the last position of the sequence because this premise has the highest entropy. Hence, a premise which could reduce the size of the decision tree, if it had been positioned in the root, is positioned as a leave. The fact that premise *C* in many cases cannot be ignored expands the number of rules produced by the algorithm.

The conclusion drawn upon experiment two is that the entropy measure is not always the optimal criterion to use for sorting the premises. In this experiment, the entropy measure produced a very deep tree!

3.3. Third Experiment

The third experiment is produced specially to reveal some potential deficiencies of the rule extracting approach. Neural nets are good to generalize (Movellan 1990) but some times this can be a disadvantage. If e.g. some important specializations of a rule are ignored the consequences can be fatal. The training set used in this experiment originates from rule 3 which has a complicated exception structure. If *A* is *true*, then the rule is *true*. But, if we don't know the value of *A*, then the value of *B* suddenly is important etc. Rule number (3) produces an unbalanced tree.

Figure 4 shows the tree based on the rules extracted from a 15-9-1 network trained with 80% of the cases. As can be seen, the network almost learned the patterns completely correct. Only the exception with premise *E* was not learned. The one pattern (out of 243) which included *E=true*, while the others were *dont_know*, was picked out by random. Hence, this is the reason for the missing exception in the tree:
A=dont_know, B=dont-know, C=dont_know, D=dont_know, E=True
Only the entropy measure fails to position the premises in the correct sequence because placing premise *D* before premise *C* introduces redundancy into the tree. This can be avoided by reducing the expressions afterwards by means of simple boolean rules. However, this issue is not examined in further detail here.



Rules:
A=dont_know, B=dont_know, D=true, C=true
or
A=dont_know, B=dont_know, D=true, C=dont_know
or
A=dont_know, B=dont_know, D=dont_know, C=true
or
A=dont_know, B=dont_know, D=false, C=true
or
A=dont_know, B=true
or
A=true

Figure 4. This tree is extracted form a 15-9-1 network which was confronted with 80% of the training cases from the third experiment. The tree can be simplified by switching the premises *C* and *D*.

The conclusion of the third experiment is that the algorithm is feasible and that the networks have captured the special structure of the rule well even though 50% of the training set has been left out.

4. PRACTICAL APPLICABILITY OF THE ALGORITHM

The algorithm discussed above has been analyzed only on small samples. Furthermore, these samples originated from existing rules. Employing the algorithm in a domain in which the real rules are not known, would be a better way to assess the feasibility of this. But such investigations have at least two major pitfalls: First, each time another premise is added to the network the number of possibilities grow by a factor three. Thence, one encounters the problem of combinatorial

explosion. Second, if most of the premises are dependent of the others a lot of rules will be produced because the tree generated will have no exits before a bunch of premises have been included. Hence, also the underlying domain can cause a combinatorial explosion. It is important to take to two pitfalls mentioned above under consideration. More specifically, the user must know which kind of dependencies maybe exist between the premises fed into the network. If the user trains networks to model causal connections between the inputs and the outputs (s)he must know which factors serve as causes and which factors are effects. This issue is shared with other quantitative models of cause-effect relations.

Another issue regarding the applicability of the algorithm is the massive number of rules which can be produced by the algorithm. These rules must be stored in memory if they should be beneficial to the user. It is with this concern in mind that the B-tree is proposed as a storing medium for the rules generated. Hence, the B-tree can handle more than two branches in each node (in this case three). Furthermore, some research has been done regarding how to speed the access time, amount of storage consumed etc. (Confer e.g.). If the algorithm is used to generate rules for an expert system these can be stored in a "compiled" form as a decision tree. This would facilitate a quick inference process.

If the algorithm is applied to domains in which it is not known which factors are the "true" causes and which just covariate with the outputs. Therefore, the generated rules must be examined by means of some heuristic. This means that the user can search the rules generated for e.g. the rule with the fewest premises included (This is similar to searching for the minimal necessary sufficient cause for some phenomenon). Or the user can e.g. search the rule base for the rules which include premise *C*. In this way, the neural network is applied as a kind of hypothesis generator for the user. The hypotheses generated can, afterwards, be tested by means of traditional statistical methods.

5. CONCLUSION

An algorithm to extract rules from trained neural networks has been presented. Furthermore, three different experiments with the algorithm were carried out and the results analyzed. The conclusion is that the rule extraction algorithm is feasible but that the entropy measure is not robust enough when applied on only a fraction of the cases belonging to a domain. Thus, the neural network seemed to be more noise resistant. It could not be shown whether increasing the number of hidden neurons in a network enhances the complexity of the rules generated, or not. Hence, this issue has to be explored further.

Additionally, a few drawbacks inherent to the algorithm have been pointed out. One is that adding a premise to the network increases the number of queries with a factor three. Another issue is that the number of rules generated depends of the character of the underlying domain. A caveat is to ignore which factors are causes and which are effects.

ACKNOWLEDGEMENTS

Thanks to Herco Fonteijn for comments on the paper.

REFERENCES

Bochereau, Laurent & Bourgine, Paul 1990. "Extraction of semantic features and logical rules from a multilayer neural network." *International Joint Conference on Neural Networks (IJCNN). P. II-579 to II-582.*

Gutknecht, Matthias & Pfeifer, Rolf 1990. "An approach to integrating expert systems with connectionist networks." *AICOM Vol. 3 NO. 3. P. 116-127.*

Eberhart, Russel & Tzanakou, Evangelia (ED) 1990. "Special issue on the application of neural networks in biology and medicine." *IEEE Engineering in Medicine and Biology.*

Myllymäki, P. et. al. 1990. "Compiling High-level specifications into neural networks." *International Joint Conference on Neural Networks (IJCNN). P. II-475 to II-478.*

Shavlik, Jude et. al. 1990. "Symbolic and neural learning algorithms: An experimental comparison." *Computer Sciences Technical Report #955, University of Wisconsin-Madison.*