Erschienen in: Fuzzy neuro-systems '98 - computational intelligence : proceedings of the 5. International Workshop "Fuzzy Neuro-Systems '98" (FNS '98) ; March 19 - 20, 1998, Munich, Germany / Wilfried Brauer (ed.). - Sankt Augustin : Infix, 1998. - S. 218-225. - (Proceedings in artificial intelligence ; 7). - ISBN 3-89601-010-7

Formalizing Neural Networks

Ingrid Fischer[1], Manuel Koch[2], and Michael R. Berthold[3]*

[1] IMMDII, University of Erlangen, Martenstr. 3, 91058 Erlangen, Germany,
email: idfische@informatik.Uni-Erlangen.DE
[2] FB 13, Computer Science, Technical University of Berlin, Franklinstr. 28/29, 10587 Berlin, Germany,
email: carr@cs.TU-Berlin.DE
[3] Department of EECS, Computer Science Division, University of California, Berkley, CA 94720, USA,
email: berthold@cs.berkeley.edu

Abstract. Graph transformations offer a unifying framework to formalize neural networks together with their corresponding training algorithms. It is straightforward to describe different kinds of training algorithms within this theoretical methodology, ranging from simple parameter adapting algorithms, such as Error Backpropagation, to more complex algorithms that also change the network's topology. Among the several benefits of using a formal approach is the support for proving properties of the training algorithms. Additionally the well-founded operational semantics of the graph transformation systems offers a possibility for the direct execution of the specification. This way graph transformations can be used to visualize and to help designing new training algorithms.

1 Introduction

In the past decade many new types of neural networks along with new and variations of old algorithms for their training have been introduced. Formal ways to describe these networks and their training have been proposed by numerous researchers, but most of them are limited to a specific type of network or algorithm. There still remains a need for a unifying framework which allows to formalize the network's structure together with all required algorithms (i.e. the computation of the network's activations and the training algorithm) in order to offer ways to formalize proofs about the behaviour of the network and also allow for an easy creation of new types of neural nets together with their corresponding algorithms.

Different types of graph transformations have already been used to generate families of neural networks; in [6] this approach together with a genetic algorithm was used

* M. Berthold was supported by a stipend of the "Gemeinsame Hochschulsonderprogramm III von Bund und Ländern" through the DAAD.

to generate the topology of a neural network from an evolving encoding. In [7] graph transformations are used to formalize training and pruning algorithms for Multi Layer Perceptrons, whereas in [9] another formalism to describe neural networks is presented, using graph grammars to grow the networks from a description. Another interesting aspect is presented in [13], here a single-flow graph representation is used to show the equivalence of different learning algorithms.

In all these approaches graph transformations are used to model only some aspects of interest. In contrast, we discuss here a unifying framework for neural nets which is independent of the type of net or the used training algorithm to provide a basis for the comparison of different algorithms and nets, where we can use the visual and theoretical advantages of the underlying system. The approach is based on graph transformations, which offer a natural representation of the neural network's architecture and the computation and training algorithm which can intuitively be represented through a set of transformation rules. Additionally the well-founded operational semantics of the graph transformation systems offers a possibility for the direct execution of the specification. Furthermore ways to formalize proofs about the network's behaviour and properties of the training algorithm (e.g. termination, convergence, etc.) are provided. This idea was proposed in [2]. In the following this approach is discussed and a topology changing algorithm for Probabilistic Neural Networks [1,12] is used as an example to demonstrate the methodology. In the next section we give a short introduction to graph transformations, additional information can be found in [10].

2 Transforming Graphs

A labeled graph G consists of a set of edges G_E, a set of nodes (vertices) G_V and two mappings $s_G, t_G : G_E \to G_V$ that specify source and target node for each edge. Additionally nodes and edges can be labeled several times with elements of different label sets[1] $L_{E_i}, L_{V_j} (0 \leq i \leq n, 0 \leq j \leq m)$. This is done with the help of mappings $l_{G_{E_i}} : G_E \to L_{E_i}, l_{G_{V_j}} : G_V \to L_{V_j}$. A morphism $(f_E, f_V, f_{L_{E_i}}, f_{L_{V_j}})$ between two graphs G and H labeled with the same alphabets consists of a mapping f_E between the edges, a mapping f_V between the nodes and f_{L_k} $(k \in \{E_i, V_j\})$ between the label sets of the graphs. A morphism must ensure commutativity with the mappings used for the graph definition, e.g. $s_H \cdot f_E = f_V \cdot s_G, f_{L_{E_i}} \cdot l_{G_{E_i}} = l_{H_{E_i}} \cdot f_E$. The same must hold for the labeling of nodes and the target mappings of the graph. A graph can be modified by graph rewriting rules. A graph rewriting rule consists of two morphisms $l : K \to L, r : K \to R$,

[1] n is the number of labels sets for edges, m the number of label sets for nodes.

a set of morphisms $A_G = \{a_i : L \to A_i | 1 \leq i \leq n\}$, called graphical conditions, and a set A_L of boolean expressions, called label conditions. In the upper half of Figure 1 a graph rewriting rule $(l : K \to L, r : K \to R)$ with one graphical $(a : L \to A)$ and one label condition $(x < y)$ is shown. Here nodes are labeled with elements from the term algebra for integers, whereas edges are not labeled. The terms x, y, z are variables. In K and C we use λ as an extra element of our label set to indicate that labels have to be changed. When applying a rewriting rule to a graph G first an embedding g of the left hand side

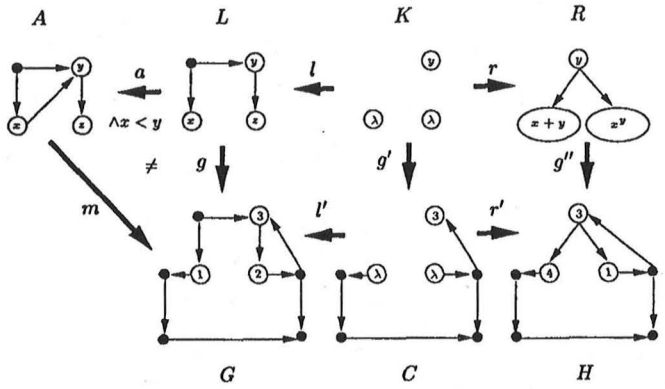


Figure 1: *An example of a rewriting rule .*

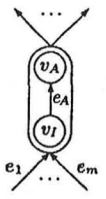
L into G must be sought. By the embedding the values of the variables are substituted by integers. In our example the variables x, y, z are replaced with the values $1, 3, 2$. Next the label conditions have to be checked. The embedding satisfies a label condition if its evaluation is true under the variable assignment of g. In Figure 1 $x < y$ denotes that the value substituted for x must be smaller than the value substituted for y, which is true in our example. Then the graphical conditions a_i have to be checked. The embedding g satisfies a_i if there is *no* embedding $m : A_i \to G$ such that $m \cdot a_i = g$, i.e. a_i specifies forbidden graphical structures. In Figure 1 there is a graphical condition forbidding an edge between the node labeled x and the node labeled y. In our example the embedding satisfies the graphical condition. In the following these negative graphical conditions will be written with $\neg\exists$ if the morphisms a_i are obvious from the graphs. If all labels as well as all graphical conditions are satisfied by g, the rewriting rule can be applied. Otherwise a new embedding must be sought if possible. We apply the rule on G by deleting the left hand side L from G except K which contains nodes and edges that are necessary to insert R. The result is the context graph C. Theoretically C is constructed as $G - g(L - l(K))$. Then R can be inserted in C according to the gluing graph K. The result H is calculated

as $C+(R-r(K))$. The new labels of graph H are obtained by evaluating the expressions given in R under the variable assignment given by the embedding g. In Figure 1 the nodes in the result graph H are now labeled with the results of addition and power resp. under the assignment $x=1, y=3, z=2$.

3 Neural Networks seen as Graphs

Before formalizing the computation and training algorithms using graph rewriting systems, the structure of the neural net has to be represented through a graph. For the labeling of edges we use pairs of real numbers of $\mathbf{R}$ and functions $f : \mathbf{R} \longrightarrow \mathbf{R}$ over real numbers. Nodes are labeled with elements of $\mathbf{R}$ together with a control symbol, which is used during training to indicate the successful update of the node's value.

As an example of how this definition can be used to describe neural networks, one neuron of a neural network is shown in Figure 2. This representation is general enough to describe a whole class of neurons of different networks, where Figure 2 shows two examples, Probabilistic Neural Networks [12] and the well known Multi Layer Perceptron [11].



	PNN	MLP
$l_E(e_i)$	$\| x-y \|, y$	$x \cdot y, y$
$l_E(e_A)$	$e^{(-x^2/y^2)}, y$	$1/(1+e^{y \cdot x}), y$
$l_V(v_I)$	eucl. distance, control symbol	weighted sum, control symbol
$l_V(v_A)$	Gaussian act., control symbol	Sigmoidal act,. control symbol

Figure 2: *A single neuron modeled through a graph. Using different ways to label the edges of the graph one can model neurons of either a PNN (with the typical Gaussian activation function) or an MLP (Sigmoidal activation function).*

The neuron is split into two parts, v_I holds the input value (in the case of the PNN an Euclidean distance), v_A the activation of the neuron. Both nodes are additionally labeled with a control symbol which reflects the current status of this node (e.g. computation done/not done). Using two nodes per neuron makes it possible to independently model the propagation- and activation-function using different labels for the edges e_i $(1 \leq i \leq m)$ and e_A. The propagation of activations between neurons is realized through edges e_i. The propagation function together with the corresponding weight of each connection can be retrieved using the labeling function l_E of the edge. The propagation function of the PNN

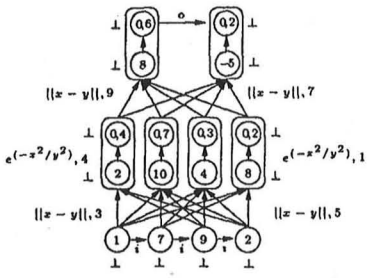


Figure 3: *A Probabilistic Neural Network seen as a graph.*

computes an Euclidean distance ($|| x - y ||$), whereas for the MLP a multiplication with the weight is computed ($x \cdot y$). The two nodes representing one neuron are connected through an edge e_A which is labeled with the neuron's activation function. Additionally, as edges are marked with pairs consisting of a function and a real number, a scalar $y \in \mathbf{R}$ is added. In the case of the PNN the activation function is typically the Gaussian for the hidden nodes ($l_V(e_A) = \{\mathcal{N}(x, y), y\}$), the scalar y represents the standard deviation of the Gaussian. The well known sigmoidal squashing function would be used for the MLP, where the scalar y represents the slope of the sigmoid. Using this way of distinguishing between propagation and activation functions makes it possible to model different neural network models by simply changing the labeling of the edges. Also hybrid models can easily be represented using different labels throughout the graph.

In Figure 3 an example of a Probabilistic Neural Network (PNN, [12]) with four inputs and two outputs having 4 neurons in the hidden layer is shown[2]. The nodes are labeled with real numbers modeling the input resp. output activation of a neuron and the control symbol $\perp$ indicates that no calculation took place on this node. The edges are labeled with a function and a real number as described in Figure 2. The modeling of a Multi Layer Perceptron is done by simply changing these edge labels. Input and output neurons are connected through horizontal edges to ensure the order of the nodes. Input nodes are just modeled through one node as they simply hold the values of the input vector without any further functionality.

4 Computation and training on the Network

Doing a forward computation, that is, computing the activations of all neurons, requires to change the labels representing the activation of the net and the control symbols of

[2] For sake of clarity not all labels are shown, however the missing labels should be clear from the context.

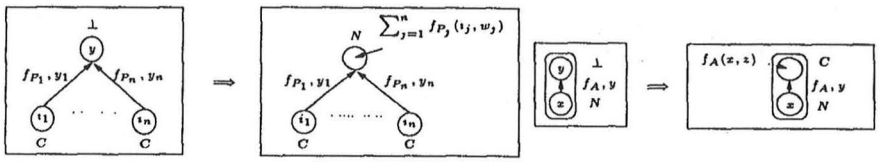


Figure 4: *These rules compute the activations of a neuron. For the PNN the typical Gaussian will be used for f_A where y denotes the standard deviation.*

these nodes to indicate that an update of the activation has taken place. For this only two relatively simple rules are necessary. Figure 4 shows one rule for the computation between layers and one rule computing a neuron's activation. Please note that in these rules the graphs are actually labeled with variables instead of functions and numbers. When applying such a rule these variables are substituted with the functions and numbers contained in the graph the rule is applied to. The control symbol ($\perp$) is changed to N (neuron) and C (connection) to indicate that a computation already took place[3]. This set of rules is independent of the actual type of network considered as all necessary information is provided in the net and these rules operate solely on variables.

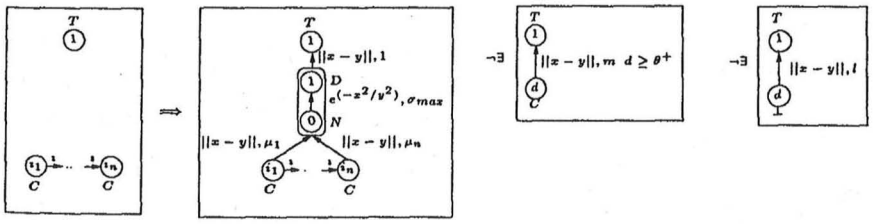


Figure 5: *If no neuron has a sufficient activation $d \geq \theta^+$ this rule inserts a new neuron in the network (commit). The application conditions ensure that no other neuron exists having a $d \geq \theta^+$ and that all necessary computations have been done.*

Similar to the computation phase training algorithms that simply adapt the network parameters can be described (for example the classic Error Backpropagation algorithm [14]). It is of much more interest, however, to also specify algorithms that change the network's topology; that is, introduce new neurons or erase existing ones. Examples for these algorithms are Cascade Correlation [3] which builds Multi Layer Perceptrons and the Dynamic Decay Adjustment (DDA, [1]) used to construct Probabilistic Neural Networks. Parts of the later algorithm are used here to demonstrate the ability of graph rewriting systems to

[3] The rule to insert a training pattern is omitted here. Within this rule the control symbol of the input neurons would be set to C.

formalize also topology changing training algorithms. A PNN contains one hidden layer where each neuron in the hidden layer has a local, usually Gaussian activation function (see Fig. 2). Thus PNNs are used to model probability density functions for classification problems. The DDA constructs PNNs from scratch by introducing new neurons (called *commit*) whenever a pattern is not sufficiently classified to the correct class (using a threshold θ^+, the so called *correct classification probability* (see [1] for more details)).

Fig. 5 now shows the rule which models this commit-step of the algorithm. Whenever no existing neuron has a sufficient activation ($\geq \theta^+$, modeled through the application condition on the right) a new neuron will be introduced. The second application condition ensures that the computation phase (see Fig. 4) is already finished for all neurons in the this layer, that is, no neuron is marked with the $\perp$-label.

5 Proving Properties

One of the main advantages of using graph transformation systems is the possibility to use theoretical results from graph rewriting or general rewriting theory to prove properties of neural networks and their training algorithms. Issues to prove can range from properties of the network or the used training algorithm to showing the relationship between different training algorithms [4,5].

In [4] it was shown that a constructive training algorithm for Probabilistic Neural Networks [1] terminates. It was possible to prove that each epoch terminates and that an upper bound on the number of required epochs exists. Defining a well-founded order on the set of rules offered an intriguing proof for the algorithm's termination.

In [5] the equivalence of two algorithms for recurrent neural networks is shown. A variant of Real Time Backpropagation [8,15] and Backpropagation Through Time [14] were modeled using one set of rules. Equivalence then means that no matter which rules are applied, the resulting network will always be the same. For this proof results from graph and term rewriting were used, especially concerning confluence of the used set of rules; that is, the indeterminism in the set of rules does not affect the final result. An additional, interesting result was the emergence of an entire class of algorithms, all producing the same result. The two algorithms being investigated are merely special cases of the entire class.

6 Conclusions

In this paper we discussed a new paradigm for describing neural networks. The used graph transformation systems offer a formal, unifying framework to specify neural networks and their training algorithms. With the theory of graph rewriting it is possible to use a large variety of techniques and theorems that have already been developed in this area. Also findings of general rewriting theory or even term rewriting can be helpful to show e.g. the convergence and termination of e.g. training algorithms. Additionally the intuitive specification of networks and algorithms enables the creation of complex, topology-adapting algorithms, which is usually rather complicated using classical approaches. Finally the presented methodology can also be used for tools that implement visual design of networks and algorithms, resulting in a tool box to build neural networks.

Literature

[1] M. R. Berthold and J. Diamond: Constructive Training of Probabilistic Neural Networks, in *Neurocomputing*, Elsevier Publisher, 1998, to appear.

[2] M. R. Berthold and I. Fischer: Modelling Neural Networks with Graph Transformation Systems, Proceedings of the *International Joint Conference on Neural Networks*, **1**, pp. 275–280, 1997.

[3] S. E. Fahlman and C. Lebiere: The cascade-correlation learning architecture, in D. S. Touretzky (ed.) *Advances in Neural Information Processing Systems*, **2**, Morgan Kaufmann, California, pp. 524–532, 1990.

[4] I. Fischer, M. Koch, and M. R. Berthold: Proving Properties of Neural Networks with Graph Transformation, Proceedings of the *International Joint Conference on Neural Networks*, Anchorage, Alaska, May 4-9, 1998.

[5] M. Koch, I. Fischer, and M. R. Berthold: Showing the Equivalence of two Training Algorithms, Proceedings of the *International Joint Conference on Neural Networks*, Anchorage, Alaska, May 4-9, 1998.

[6] F. Gruau: Genetic Micro Programming of Neural Networks, in K. E. Kinnear, Jr.: *Advances in Genetic Programming*, MIT Press, Cambridge, MA, 1994.

[7] B. Haberstroh and R. Freund: Tools for Dynamic Network Structures: GRAPE Grammars, *International Joint Conference on Neural Networks*, Baltimore, 1992.

[8] J. A. Hertz, A. Krogh, and R. G. Palmer: *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading.

[9] S. M. Lucas: Growing Adaptive Neural Networks with Graph Grammars, *European Symposium on Artificial Neural Networks*, Bruxelles, 1995.

[10] G. Rozenberg (ed): *Handbook on Graph Grammars: Foundations*, **1**, World Scientific, 1997.

[11] D. E. Rumelhart and J. L. McClelland: *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, MIT Press, Cambridge, MA, 1987.

[12] D. F. Specht: Probabilistic Neural Networks, in *Neural Networks*, **3**, pp. 109–118, 1990.

[13] E. A. Wan and F. Beaufays: Diagrammatic Derivation of Gradient Algorithms for Neural Networks, in *Neural Computation*, 8:1, pp. 182–201, 1996.

[14] P. Werbos: Backpropagation Through Time: What it does and how it does it, *Proceedings IEEE, special issue on Neural Networks 2*, 1550-1560, 1990.

[15] R. J. Williams and D. Zipser: A learning algorithm for continually running fully recurrent Neural Networks, *Neural Computation*, 1:2, pp. 270-280, 1989.