Guiding Backprop by Inserting Rules

Sebastian Bader[1] and **Steffen Hölldobler**[2] and **Nuno C. Marques**[3]

Abstract. We report on an experiment where we inserted symbolic rules into a neural network during the training process. This was done to guide the learning and to help escape local minima. The rules are constructed by analysing the errors made by the network after training. This process can be repeated, which allows to improve the network performance again and again. We propose a general framework and provide a proof of concept of the usefullness of our approach.

1 Introduction and Motivation

Rule insertion prior to training can lead to faster convergence and to better results (see e.g. [9, 4]). Here, we will investigate whether we can do this repeatedly during the training process.

Artificial neural networks are a very powerful tool to learn from examples in high dimensional and noisy data. Standard feed-forward networks together with back-propagation of errors have been successfully applied in many domains. However, in most domains, at least some background knowledge in form of symbolic rules is available. And this knowledge can not be used easily. As mentioned above, there have been studies in which this knowledge is embedded prior to the training. But there is no way to guide the learning process using this rules.

The following observations have triggered this research:

- Very general rules, i.e., those that cover many training samples, are quickly acquired by back-propagation. And hence, embedding those rules does not help too much.
- Very specific rules that are embedded prior to the training will very likely be overwritten by newly learned rules.
- We can analyse the errors made by the network to obtain correcting rules.

In the area of natural language processing, the combination of rule based and machine learning based methods seems to be very promising. Recently, we showed that we can accuracy by inserting symbolic rules into an artificial neural network [7]. We believe that we can even improve even further if we embed rules repeatedly during the training. In this paper, we will describe how to do this and why we believe that this seems to be a good idea.

We propose a general method to guide the training of artificial neural networks. Our approach is based on the repeated embedding of rules during the training. Here, we will focus on the description of an experiment to verify our claims. This paper is only meant to be a case study and to serve as a proof of concept. And shall be the starting point for a bigger investigation. We are currently preparing larger experiments in the area of natural language processing.

[1] International Center for Computational Logic (ICCL), Technische Universität Dresden, Germany, email: sebastian.bader@inf.tu-dresden.de
[2] ICCL, email: sh@iccl.tu-dresden.de
[3] Centria, DI-FCT-UNL, Lisbon, Portugal, nmm@di.fct.unl.pt. Work developed while the author was a visiting the ICCL.

2 Preliminaries

We assume the reader to be familiar with basic concepts of neural networks and the training by back-propagation as e.g., described in [8] and implemented in [10]. Here, we will only use simple three layer fully connected feed-forward neural networks applying the $\tanh$-function in hidden and output layer. We will train them using back-propagation without enhancements like e.g., momentum and fixed learning parameters.

The rules we will use here, are simple propositional if-then rules. Each rule consists of multiple preconditions which have to be satisfied simultaneously, and a single atomic consequence. Below we will study a classification task for tic-tac-toe boards, hence the consequence will be the class and the preconditions will be (partial) board descriptions.

3 A General Method for Guiding Backprop

Our approach, is based on the repeated modification of the network during the training. After a number of training cycles, the errors made by the network are analysed. This analysis should yield a rule that can be used to correct some of the errors. We can now embed the rule into the network and continue with the next epoch. This process is depicted in Figure 1. As we will see below, this will help the network to escape local minima during the training. To summarise, our approach works as follows:

1. Initialise the network.
2. Repeat until some stopping condition is satisfied:
   (a) Train the network for a given number of cycles.
   (b) Analyse the errors of the network to obtain correcting rules.
   (c) Embed the rule(s) into the network.

In the sequel, we will show that this works using a simple classification task. This is only meant to be a proof of concept, and not to show the superiority of this method in general.

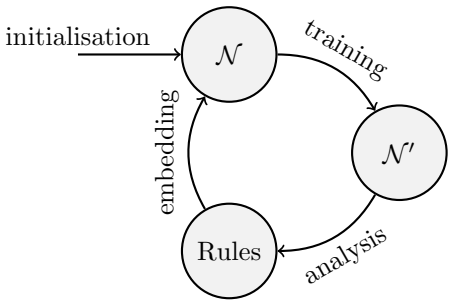

Figure 1. The rule-insertion cycle.

4 The "Tic-Tac-Toe" Classification Task

We used the "Tic-Tac-Toe Endgame Data Set" of the UCI Machine Learning Repository [1] for our experiments described below. The data set contains all 958 possible board configurations that can be reached while playing Tic-Tac-Toe starting with player X. Each configuration contains a description of the board and is classified as win or no-win for player X. A player wins the game, if he manages to place 3 of his pieces in a line. The board contains 3×3 cells and each cell can be marked by either x or o, or can be blank (b). Possible samples are $[\mathsf{x}, \mathsf{b}, \mathsf{o}, \mathsf{o}, \mathsf{x}, \mathsf{o}, \mathsf{x}, \mathsf{b}, \mathsf{x}]^+$, $[\mathsf{x}, \mathsf{x}, \mathsf{o}, \mathsf{o}, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{o}, \mathsf{x}]^-$ and $[\mathsf{x}, \mathsf{o}, \mathsf{x}, \mathsf{o}, \mathsf{o}, \mathsf{b}, \mathsf{x}, \mathsf{o}, \mathsf{x}]^-$. The corresponding boards are shown in Figure 2 from left to right. The goal of the task is to decide whether a given board is a win-situation for player X or not.

x		o
o	x	o
x		x

x	x	o
o	o	x
x	o	x

x	o	x
o	o	
x	o	x

Figure 2. Three board configurations, showing a win for player X, a draw and a win for player O, i.e., this are one positive and two negative examples.

4.1 A Network to Classify Board States

We used a simple 3-layer feed-forward network with $\tanh$ as activation function in all units. It contains $9 \times 3 = 27$ units in the input layer, that is three for each of the nine position on the board, corresponding to the three possible states. For each of this triples we used an 1-out-of-3 activation schema. If a cell contains for example an "x" the x-unit for this cell will be activated by setting the to output 1, while the o and b-units are set to -1. Initially, the network contains one hidden units and one output unit. A board state is fed into the network by activating the appropriate input units. This activation pattern is propagated through the network and the networks decision can be read from the output unit. If its activation is greater than 0 the network evaluates the board as a win for player X, and as a no-win otherwise. Figure 3 shows the network used in our experiments. We will now continue by detailing the general steps of our method as introduced in Section 3.

Initialising the Network We initialised the network with a single hidden units and all connections as well as the bias of the units were randomised to values between -0.2 and 0.2.

Training the Network The network was trained using standard back-propagation in the SNNSimulator [10]. We used simple back-propagation without any additions. The learning rate was set to $\eta = 0.1$. During each epoch we presented all samples 100 times to the network.

Analysis of the Errors to Obtain Correcting Rules After the training, we presented all samples again to the network and compared the networks output to the desired output. All samples where the difference between the computed and the desired output was greater than 0.5 were collected and grouped into positive and negative samples. Depending on whether there have been more errors on positive or on negative samples, we continued with the bigger set. From this set, we constructed a template that is as accurate as possible and at the same time as general as possible. This was done by repeatedly selecting a

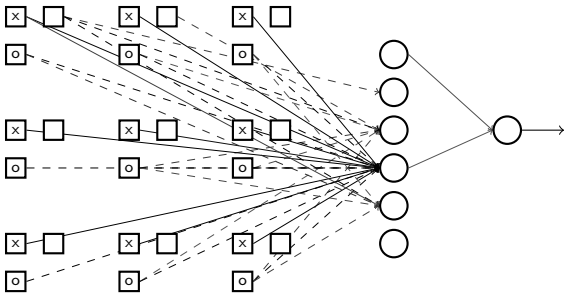

Figure 3. A 3-layer fully connected feed-forward network with 27 input (rectangular), 6 hidden (cyclic) and a single output (cyclic) unit to classify Tic-Tac-Toe boards. The 27 input units are arranged like the board itself. Positive connections are shown as solid lines, while negative ones dashed. The width corresponds to the strength and small connections are omitted.

cell and a value such that most erroneous samples agree on this. This is a variant of the "Learn one rule"-algorithm as used in sequential covering algorithms, but considers only one class [8].

The following rule was constructed from 99 wrongly classified sample: $[\mathsf{b}_{13}, \mathsf{b}_3, \mathsf{o}_{23}, \mathsf{x}_{43}, \mathsf{x}_{34}, \mathsf{x}_{70}, \mathsf{o}_7, \mathsf{b}_1, \mathsf{b}_1] \mapsto +$. The subscripts show the percentage of the 99 samples which agree on the value. E.g., 13% of those samples had an empty upper left corner and 34% an x in the centre. We constructed the rule template by ignoring all entries with a support $< 20\%$. This resulted in the following template $[?, ?, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{x}, ?, ?, ?] \mapsto +$, which actually covers 38 of the samples.

Embedding Rules into the Network The rules constructed above, are embedded into the network following the general ideas of the *Core Method* as specified in [5], [4] and [3]. A new hidden unit is inserted into the network and the connections and the bias are set up such that this unit becomes active if and only if the input of the network coincides with the rules precondition.

Let us use the rule $[?, ?, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{x}, ?, ?, ?] \mapsto +$ to exemplify the idea. For all entries different from ?, a connection with weight ω from the input unit corresponding to the value and connections with weight $-\omega$ from the other two units are created. E.g., the x-unit for the central cell is connected to the new hidden unit with weight ω, while the corresponding o and b-units are connected with weight $-\omega$. All connections from input units for cells marked ? in the template have been initialised to 0.0. And the connection to the output unit has been set to ω for positive rules and to $-\omega$ otherwise. The result is shown in Figure 4. Finally, all connections have been slightly disturbed by adding some small random noise from $[-0.05, 0.05]$. In the experiments described below we used different values for ω.

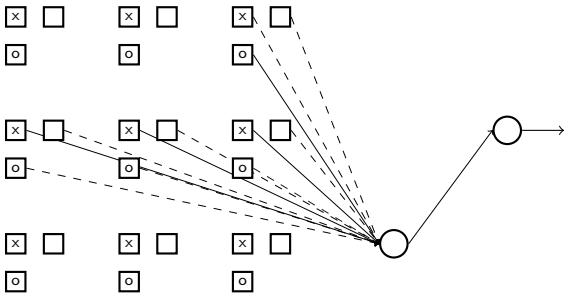

Figure 4. The embedding of the rule $[?, ?, \mathsf{o}, \mathsf{x}, \mathsf{x}, \mathsf{x}, ?, ?, ?] \mapsto +$. Connections are depicted as described in Figure 3.

4.2 Experimental Evaluation

We performed several experiments to evaluate our method, varying the parameter ω. We used $\omega \in \{0.0, 0.5, 1.0, 2.0, 5.0\}$. By setting it to 0.0, we can emulate the simple addition of a free hidden unit during the training, i.e., the new unit will be initialised randomly. Therefore, this case should be the baseline for our experiments. Small values would only push the network slightly into the direction of a rule, while a value of 5.0 would strongly enforce the rule. In particular, if a rule was embedded with $\omega = 5.0$, then as soon as the corresponding hidden unit became active it's output will very likely dominated the sum of the outputs of the other hidden unit. This is due to the fact that its activation of approximately 1.0 is propagated through a connection with a weight of 5.0 or -5.0 to the output unit. Therefore, the output unit will usually simply follow the rule. Consequently, as a rule of thumb, only rules where one is absolutely certain should be embedded with very large ω.

Figure 5 shows the result of the experiment. It shows the mean squared error over the time for the different settings of ω. For each value, we repeated the experiment 50 times and showed the averages. It also shows a zoom into the lower right corner, i.e., into the final cycles. Here, not only the averages, but also the standard deviation is shown. There are a few points to notice in those plots, which will be discussed in more detail below:

- For $\omega = 0$ the network stops to improve at some point, even if units are added.
- For $\omega > 0$ the network outperforms the network where ω was set to 0 in the later cycles.
- For $\omega = 5$ the network did not learn very well.

Due to local minima in the error surface, back-propagation can get stuck in non-optimal solutions. Usually this is the case if there are only errors on few samples, because back-propagation follows in principle the majority vote. This also happens in our experiments. For $\omega = 0$, the network fails to improve further and remains at the 0.01-level. Our method, proposed here, can help to solve this problem by inserting a unit which covers those few samples. Thus, we allow back-propagation to jump to better solutions.

The bad performance for $\omega = 5$, is due to the above mentioned fact about big values for ω and the fact that our rules are not necessarily correct. It happened several times in our experiments that the same (only partially correct) rule was embedded again and again. E.g., the rule $[?, ?, \mathsf{x}, ?, ?, ?, \mathsf{x}, ?, ?] \mapsto +$ was embedded, because most of the positive samples on which there were errors agreed on those values. But at the same time there are 42 negative samples with those values. As the network blindly followed the rule, those negative samples are wrongly classified afterwards. During the training phase, the network basically unlearnt the rule but could not improve on the original wrong positive samples. And thus, the same bad rule was embedded again and again. This problem could be solved using e.g., tabu-lists preventing the re-insertion. But we believe that better rule-generation should be used, which construct only valid rules.

There is another finding in our experiments which is worth to be mentioned. E.g., the rule discussed in Section 4.1, covers only 38 samples, but the number of errors of the network on the training data went down from 99 to 26, i.e., inserting the rule apparently helped to better classify 73 samples. We think this is due to the better starting point provided by the insertion of the rule, that helps the network to correct errors more efficiently. This effect will be subject to further studies because it seems to improve the performance to a level which cannot be achieved by symbolic or connectionist learning alone.

5 Conclusions and Further Work

We presented a first experiment to support our claim that we can positively influence the training process of a feed-forward neural network by incorporating knowledge in the form of rules into the training process. Thus, we make a first attempt to solve the challenge-problem number 4 mentioned in [2], i.e., the improvement of established learning algorithms by using symbolic rules. Here, the rules were obtained from wrongly classified examples after an epoch of training the network using back-propagation.

One of the main problems was to find good rules. So far, we applied ID3 and C4.5 techniques for rule learning on all available data prior to the training and embedded the rules. On the other hand, we found that artificial neural networks with proper encoding learnt such (very general) rules easily. The main problems are usually related with infrequent data patterns. Therefore, we believe that the analysis of the errors as proposed here is better than an a-priory generation of rules. But the problem of finding good rules remains. In this paper, we used a very naive approach to construct the correcting rules, but more powerful methods like ID3 or others could improve the performance even more [8]. But the method of choice will probably depend a lot on the application domain. And it is actually the point where background knowledge about the domain can be incorporated into the training process. Alternatively, error-analysis and rule creation could be done by a human expert. One should observe that this expert is only required once the network has learnt most of the rules. I.e., the human intervention is done only for the difficult cases and thus the expert does not need to explicitly state the simple rules.

In this work, we did not use separate training and validation sets, because we wanted to study the improvement during learning. But we observed in other experiments, that rule insertion can also lead to better generalisation. But this is again very much dependant on the quality of the rules. If the rules generalise, their embedding will also lead to a better generalisation of the neural network. We will study this problem in our follow-up work on natural language processing in more detail.

We have also found that rule insertion improves results beyond the knowledge directly expressed in rules. Indeed, similarly to work done when trying to improve neural networks by pruning irrelevant weights, rule insertion can also, indirectly help in the task of overriding irrelevant information. We believe this relation should be made clear in further work. Indeed the relevance of magnitude based pruning methods is well known in methods such as the ones related with optimal brain damage [6]. In this sense, we believe that there should be some resemblance between these methods. Therefore, we will investigate the relations with other pruning and growing techniques in the future. This will be done empirically by comparing the performance of different methods, but the general methodology of the core method may even serve as a basis for a theoretical comparison between the different approaches.

We understand that this paper presents only a starting point. But we believe that the methodology presented here can be developed into a full fledged training paradigm that allows to incorporate domain-dependent background knowledge in a concise way. We are currently applying our algorithm to larger data-sets. As pointed out in the beginning, natural language processing problems seem to provide a good challenge where neuro-symbolic integration should be advantageous. As already reported in [7], we can improve the performance of a connectionist Part-of-Speech tagger by inserting using prior to the training. Next, we will study whether we can further imrove it by guiding the training using error-correcting rules as proposed here.

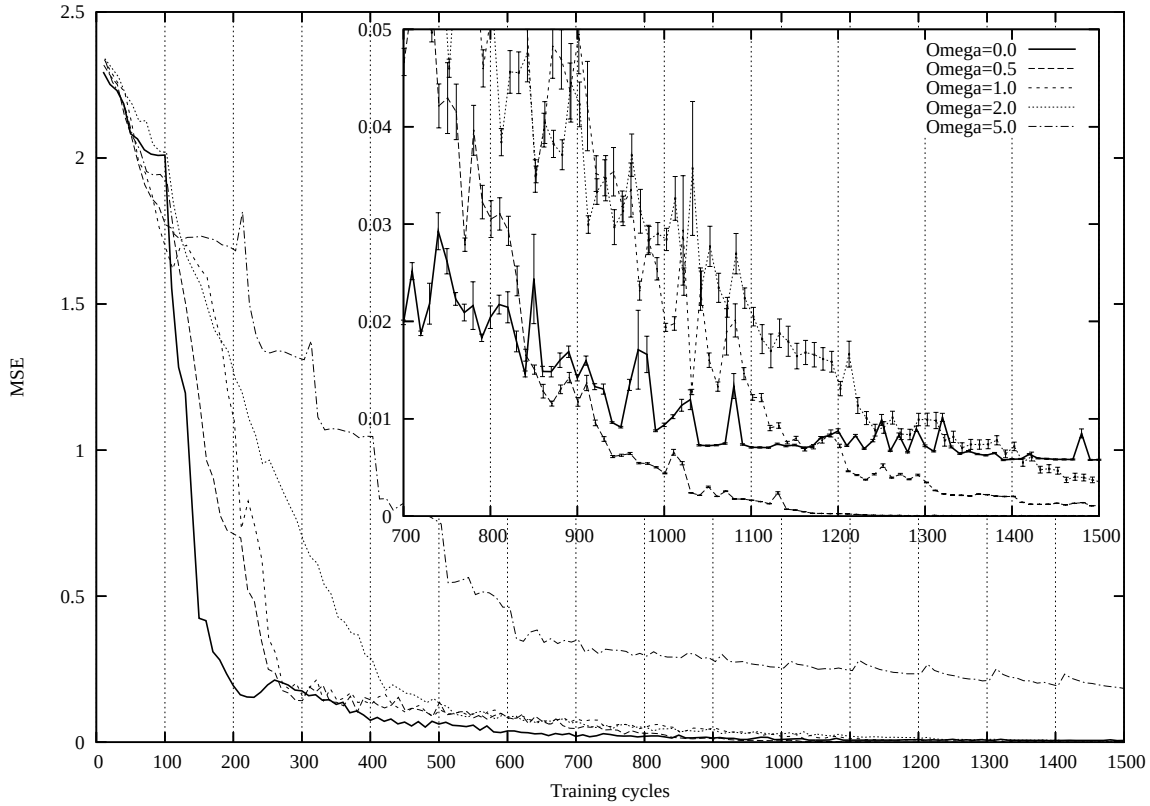

Figure 5. The development of the mean squared error over time for different values of ω (averaged over 50 runs). The vertical lines show the timepoints where rules have been inserted (every 100 cycles). The zoom in the upper right corner shows also the standard deviation over all 50 runs.

ACKNOWLEDGEMENTS

We would like to thank the referees for their comments which indeed helped to improve the paper.

REFERENCES

[1] A. Asuncion and D. J. Newman. UCI machine learning repository, 2007.

[2] Sebastian Bader, Pascal Hitzler, and Steffen Hölldobler, 'The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence', *Journal of Information*, **9**(1), 7–20, (January 2006).

[3] Sebastian Bader, Pascal Hitzler, and Steffen Hölldobler, 'Connectionist model generation: A first-order approach', *Neurocomputing*, (2008). accepted, in press.

[4] Artur S. d'Avila Garcez, Krysia B. Broda, and Dov M. Gabbay, *Neural-Symbolic Learning Systems — Foundations and Applications*, Perspectives in Neural Computing, Springer, Berlin, 2002.

[5] Steffen Hölldobler and Yvonne Kalinke, 'Towards a massively parallel computational model for logic programming', in *Proceedings ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pp. 68–77. ECCAI, (1994).

[6] Y. LeCun, J. Denker, S. Solla, R. E. Howard, and L. D. Jackel, 'Optimal brain damage', in *Advances in Neural Information Processing Systems II*, ed., D. S. Touretzky, San Mateo, CA, (1990). Morgan Kauffman.

[7] Nuno C. Marques, Sebastian Bader, Vitor Rocio, and Steffen Hölldobler, 'Neuro-symbolic word tagging', in *New Trends in Artificial Intelligence*, ed., José Machado José Neves, Manuel Filipe Santos, pp. 779–790. APPIA - Associação Portuguesa para a Inteligência Artificial, (12 2007).

[8] Tom M. Mitchell, *Machine Learning*, McGraw-Hill, March 1997.

[9] Geoffrey G. Towell and Jude W. Shavlik, 'Knowledge-based artificial neural networks', *Artificial Intelligence*, **70**(1–2), 119–165, (1994).

[10] Andreas Zell, 'SNNS, stuttgart neural network simulator, user manual, version 2.1', Technical report, Stuttgart, (1992).