Classifying Relational Data with Neural Networks

Werner Uwents and Hendrik Blockeel

Katholieke Universiteit Leuven,
Department of Computer Science,
Celestijnenlaan 200A, B-3001 Leuven
{werner.uwents, hendrik.blockeel}@cs.kuleuven.be

Abstract. We introduce a novel method for relational learning with neural networks. The contributions of this paper are threefold. First, we introduce the concept of relational neural networks: feedforward networks with some recurrent components, the structure of which is determined by the relational database schema. For classifying a single tuple, they take as inputs the attribute values of not only the tuple itself, but also of sets of related tuples. We discuss several possible architectures for such networks. Second, we relate the expressiveness of these networks to the 'aggregation vs. selection' dichotomy in current relational learners, and argue that relational neural networks can learn non-trivial combinations of aggregation and selection, a task beyond the capabilities of most current relational learners. Third, we present and motivate different possible training strategies for such networks. We present experimental results on synthetic and benchmark data sets that support our claims and yield insight in the behaviour of the proposed training strategies.

1 Introduction

Neural networks are a very popular learning method. However, their use is still mainly limited to propositional data. A number of approaches exist to extend them to structured domains, such as logical terms, trees and graphs [9, 18, 8]. However, none of them is specifically oriented to relational databases. Other research focuses on the combination of neural networks and first-order logic [3, 1], but the relational problem should be simpler to solve. In this paper, we will discuss a possible extension of propositional neural networks to relational databases.

In general, learning concepts over relational data can be considered as learning a combination of aggregation and selection. The distinction between aggregation and selection is basically a distinction between two different ways of handling sets. The difficulties for current relational learners to make combinations of both are an important motivation for our relational neural networks (RNNs).

This problem of combining aggregation and selection will be elaborated in section 2. In the context of neural networks, these combinations can be learned

S. Kramer and B. Pfahringer (Eds.): ILP 2005, LNAI 3625, pp. 384–396, 2005.

using a neural network consisting of feedforward and recurrent parts. The precise structure of relational neural networks will be explained in section 3. The training method for the networks is based on the well-known backpropagation algorithm. Some specific issues for training relational neural networks are discussed in section 4. To test our approach, four experiments were conducted and the results are presented in section 5. Finally, some conclusions will be formulated in section 6.

2 Aggregation Versus Selection

In propositional learning, an example is described by a single tuple of a fixed type (i.e., each example is described by the same attributes). In relational learning, an example is essentially described by a set of tuples that are somehow related to each other. The tuples may be of different types and the size of such a set is in general not constrained. Because of the latter property, the set cannot be reduced to a single tuple without loss of information. Thus, we can say that the essential difference between propositional and relational learning is that relational learners need to be able to handle sets in some way. They need to be able to construct tests on sets rather than on scalar attributes.

Some relational learners use what is called a propositionalisation approach: they transform the data into a propositional format using a number of predefined features, and let the propositional learner choose those features that are most relevant. Other relational learners integrate the construction of such features in the learning process.

Independent of the question whether feature construction happens before or during learning, we can also look at the type of features that are constructed. In relational algebra terminology, we can say that such features are of the form $\mathcal{F}(\sigma_C(S))$ where $\mathcal{F}$ is some aggregate function, σ_C maps the set S into its subset of elements that fulfill condition C, and S is the natural join of all the tuples linked by foreign keys to the tuple to be classified [2].

We can classify symbolic relational learners according to what kind of aggregate functions and selection conditions they consider. It then turns out that many propositionalisation approaches choose $\mathcal{F}$ from a predefined set of functions (typically count, sum, average, max, min; note that except for count, one has to specify an attribute in combination with the function, which means the actual number of features to be considered is linear in the number of attributes), and use for C a trivial or very simple condition.

Many propositionalisation approaches consider C to be true. The number of possible features then derived is still $O(fa)$ with f the number of aggregate functions and a the number of attributes. For instance, the RELAGGS approach [12, 13] considers several aggregate functions, and atomic conditions of the form $A\theta v$ with A an attribute, v a value, and θ a comparison operator. This makes the number of possible features $O(fa^2)$. Building a more complex C, for instance,

one involving multiple conjuncts, is difficult because the number of possible conjunctions grows exponentially in the number of conjuncts.

Inductive logic programming systems can be considered as constituting the other side of the spectrum: they build complex conditions C but a trivial aggregate function $\mathcal{F}$ that returns true if $\sigma_C(S)$ is non-empty. Indeed, a clause such as

```
pos(X) :- page(X), hub(X), linked(X,Y), hub(Y).
```

can be seen as constructing a boolean feature that expresses whether the page is linked to by a hub. (In other words, the set of pages linking to this page that are hubs, is non-empty.) An ILP system could add further conditions on Y to the clause, possibly introducing more variables somehow linked to Y, thus making the C condition arbitrarily complex.

As ILP systems focus on the construction of the selection condition, we call them selection-oriented. Systems that include aggregate functions with only very simple selection conditions, can be called aggregation-oriented. The question then arises whether systems could be built that look for patterns involving both aggregation and non-trivial selection conditions.

It turns out that this is difficult because of several reasons. First, clearly, the feature space that has to be searched becomes much larger. Second, it is more difficult to navigate this space in an efficient and structured way. One approach towards combining aggregation and selection is the work by Knobbe et al. [11]. They propose a method to search this more complex feature space for aggregations over complex selections. In order to keep the search well-behaved, however, they have to restrict the aggregate functions to monotone ones. Vens et al. [20] propose an approach where any aggregate functions can be combined with complex selections; their random forests [4] based approach involves a random sampling of the feature space, which makes the search feasible.

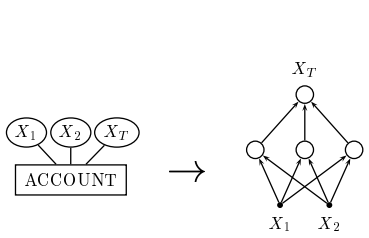
Perlich and Provost [16] provide an alternative characterization of relational learners in terms of probability distributions; what we call an aggregate over a complex conjunction, in their terminology boils down to summarizing statistics of a joint distribution over multiple variables. They essentially arrive at the same conclusion with respect to the position of ILP and aggregation-oriented relational learning approaches: both are at different sides of a spectrum that is very sparsely populated (if at all) in between.

The relational learning approach that we propose here, is a non-symbolic approach, and as such does not make a distinction between searching for aggregate functions and searching for complex conditions. It does both in parallel, and yields models that may be closer to selection-oriented models, or closer to aggregation-oriented ones, depending on what seems most fit for the dataset under consideration. In addition, they are not constrained to using only predefined aggregation functions, or to using a specific kind of conditions. Our approach is unique in this respect and makes it possible to learn patterns that none of the current relational learners can model.

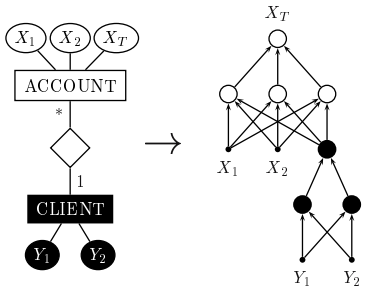
3 The Structure of Relational Neural Networks

The structure of a relational neural network (RNN) is based on the schema of the relational database. More specifically, it is influenced by the different types of tuples in the data set, the number of attributes for each tuple type and the relationships that are allowed. It is important that every attribute should be a real value, because these are the only values a neural network can process. Other types of attributes require a transformation to a fixed number of real values. Standard transformations for this are known.

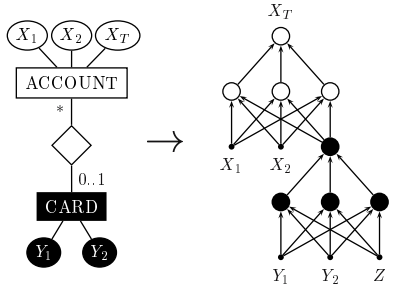
A good starting point to address the relational learning task, is the typical setting for solving propositional learning tasks. The usual method to construct a neural network for a propositional data set is illustrated in figure 1(a). In a propositional data set, only one type of tuple is present, in this case account tuples. All tuples of this kind are characterized by three attributes, X_1, X_2 and X_T, as is shown on the left side of the figure. On the right side, a corresponding neural network is depicted. More specifically, a standard feedforward neural network with two layers is used. X_1 and X_2 are used as inputs to this network. X_T is a special attribute, because it is the target attribute, which must be predicted. This value is used to train the neural network at the output.



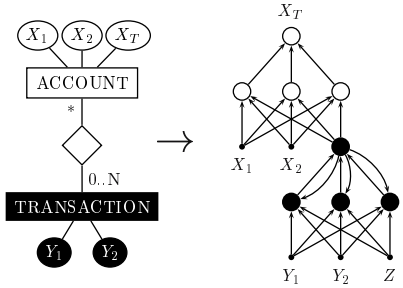
(a) Propositional case.



(b) One-to-one or many-to-one relationship, complete participation.



(c) One-to-one or many-to-one relationship, partial participation.



(d) One-to-many or many-to-many relationship.

Fig. 1.

Starting from this propositional case, an extension can now be proposed to solve relational tasks. This extension is developed in three steps. The first step is to handle one-to-one relationships with complete participation. Figure 1(b) shows how a new type of tuple, client, is added to the original data set. This new type of tuple has a one-to-one relationship with account tuples. It means that each account is related to exactly one client. This case could easily be transformed into a propositional case by adding some extra inputs for the related client tuple to the original network.

However, we will follow a different approach here, that is more similar to the use of combinations of aggregation and selection. When using such a combination, the related tuples are actually summarized and the result is used in predicting the target. Similarly, we can use a neural network, representing a combination of aggregation and selection, to summarize related tuples. The outputs of this network are then used as extra inputs into the original network so that it can predict the target.

In the case of a one-to-one relationship, this summarizing is performed by a feedforward network. On the right side of figure 1(b), the original propositional network, indicated by the white neurons, can still be distinguished. However, a new neural network, indicated by the black neurons, has been added. The attributes Y_1 and Y_2 of the client tuples are used as inputs to this network and the outputs are used as extra inputs to the original propositional network.

More generally, we are assuming that we have a data set with a target relation R_T and some other relations $R_1, \ldots, R_M$. The attribute set of R_i is denoted by U_i. $S_1(R)$ can now be defined as the set of all relations R_i with which R has a one-to-one or many-to-one relationship with complete participation. For all R_i in $S_1(R_T)$, where R_T is the target relation, a feedforward neural network N_i is created. The inputs I_i of N_i are equal to U_i in this case. The outputs O_i of N_i are used as inputs for N_T, so $I_T = U_T \cup (\cup_{i:R_i \in S_1(R_T)} O_i)$.

The second step is very similar to the first one. Instead of a one-to-one or many-to-one relationship with complete participation, a one-to-one or many-to-one relationship with only partial participation is considered here. This could be a relationship between account and card tuples for example, as shown in figure 1(c). Compared to the neural network for the first relational extension, only a new input Z is added. As there is only a partial participation between account and card, not every account has a card tuple related to it. The variable Z is therefore used to indicate whether there is a card tuple related to the account tuple or not. This variable has two possible values, for instance zero and one.

Again, this result can be described more formally for a target relation R_T and a number of other relations $R_1, \ldots, R_M$. Now we define $S_{01}(R)$ as the set of all relations R_i with which R has a one-to-one or many-to-one relationship with partial participation. For all R_i in $S_{01}(R_T)$ we define a feedforward network N_i, with inputs $I_i = U_i \cup \{Z\}$. The domain of Z is $\{0, 1\}$. The outputs O_i of N_i are used again as inputs for N_T, so $I_T = U_T \cup (\cup_{i:R_i \in S_1(R_T) \cup S_{01}(R_T)} O_i)$ so far.

A third step is required to facilitate handling sets, which involves recurrent neural networks. Instead of a one-to-one relationship, a one-to-many relationship

is now added to the data set in the form of a relationship between account and transaction tuples. An account can not only have one or zero transactions, as with cards, but also multiple transactions. An extra variable Z is used to indicate whether or not there is at least one transaction present. To be able to process multiple transactions, the added network is now a recurrent network. It contains recurrent connections which feed signals from the second layer back into the first layer. This enables the processing of a sequence of input vectors, so the set of transaction tuples is fed in the recurrent network as a sequence.

Formally, we define the set $S_M(R)$ of all relations R_i with a one-to-many or many-to-many relationship from R to R_i. This time a recurrent network N_i is constructed for each R_i in $S_M(R_T)$. The inputs for N_i are $I_i = U_i \cup \{Z\}$, where $Z \in \{0, 1\}$. The outputs O_i of N_i are added again to the inputs of N_T, resulting in $I_T = U_T \cup (\cup_{i:R_i \in S_1(R_T) \cup S_{01}(R_T) \cup S_M(R_T)} O_i)$.

The described method of constructing a new neural network and using its outputs as extra inputs to the original network, can also be applied to further relationships. If transaction tuples have a relationship with a bank, for instance, a new network to process this bank tuple can be added and the outputs used as extra inputs to the recurrent network for transaction tuples. This results in a tree structure, where every node is a network that processes tuples of some type and its children process tuples involved in some relationship with the parent tuple. The signals are propagated from bottom to top.

The most expressive recurrent networks are fully connected networks in which each neuron is connected to all other neurons. However, this makes the number of connections increase quadratically with respect to the number of neurons and therefore we prefer the Jordan recurrent network [10]. In the latter type of recurrent network, each neuron in the second layer is connected to all neurons in the first layer. This is the same as using the outputs of the neurons in the second layer as extra inputs for the network in the next update step. The number of recurrent connections is then $n_1 \times n_2$, with n_1 and n_2 the number of neurons in the first and second layer respectively. This results in a good trade-off between expressiveness and the number of neurons and connections in the network.

4 Training

The described relational neural network consists of feedforward as well as recurrent parts. Both are trained using the backpropagation algorithm. For the feedforward parts, standard backpropagation is used and the recurrent parts are trained with backpropagation through time. The latter is an adapted version of standard backpropagation for recurrent networks [21]. The key idea to backpropagation through time is to unfold the recurrent network into a feedforward network. As many folds or copies of the original network are created as there are instances in the input sequence. All recurrent connections are converted into feedforward connections between successive folds. The resulting feedforward network is trained using standard backpropagation, except for the fact that all weight updates are added to the original weights.

As explained above, the recurrent networks are used to process sets of tuples. This means that all tuples in the set are fed in the recurrent network as a sequence. The particular order of these sequences is arbitrary, as there is no real order in the set of tuples. To avoid imposing an artificial order on the data, it is possible to reshuffle or reorder these sequences randomly to train the network. Thus, the variation in the data presented to the network is increased, which should improve learnability. Reshuffling can also be done for testing where each sample is tested in different orders and the results are averaged, which should improve prediction.

There are two ways of reshuffling for training. In a first setting, simply reshuffle the training set after every iteration. This method presents a maximum of variation to the network, at least in the long term. Another possibility is to expand the original training set with a number of reshuffled copies. The latter method will increase the size of the training set initially, but during the training process the training samples remain the same. At first sight, reshuffling after every iteration would seem to give the best result as it produces maximal variation in the data presented to the network. However, it also changes the gradient from one iteration to the next, which can make it difficult for the training algorithm to converge.

5 Experiments

The described approach was tested on four different data sets. Experiments were conducted using ten-fold cross-validation and the results are averages over five different runs. Three different settings were used for training: without reshuffling, using continuous reshuffling and using ten reshuffled copies. When reshuffling is used for training, the accuracy on the test set is measured over twenty reshuffled tests for each sample. At least two questions should be answered by conducting these experiments. First, we want to know how the results obtained with relational neural networks compare to results for other systems, such as first-order random forests (FORFs) [20]. This should indicate whether our RNNs are indeed able to learn relational concepts. Second, the effect of reshuffling on learning such concepts should become clear.

5.1 Musk

Musk is actually a multi-instance data set, but multi-instance learning can be seen as a special, simple case of relational learning [5]. The data set consists of two parts, each containing a number of molecules and a bag of conformations for each molecule [14]. A conformation is described by 166 numerical attributes. Each molecule has to be classified as musk or not. There are 92 molecules in the first data set and 102 in the second one. A further difference between the two data sets is the average number of conformations per molecule. For the first data set there are 5 conformations per molecule on average, for the second data set the average is 65.

Table 1. Accuracies for relational neural networks on all tested data sets. Average accuracy and standard deviation over five runs are given for ten-fold cross-validation.

	no reshuffling	continuous reshuffling	10 reshuffled copies
musk 1	80±3%	82±3%	84±3%
musk 2	78±2%	79±2%	80±2%
trains 1	76±4%	91±3%	93±3%
trains 2	74±4%	86±3%	87±3%
trains 3	86±3%	94±3%	96±3%
trains 4	83±4%	85±3%	89±3%
mutagenesis	86±3%	88±3%	86±4%
diterpenes	81±2%	78±2%	79±2%

Table 2. Accuracies on musk data set compared to other methods. Results were obtained from [6] and [17].

	method	musk 1	musk 2
1	iterated-discrim APR	92.4%	89.2%
2	GFS elim-kde APR	91.3%	80.4%
3	GFS elim-count APR	90.2%	75.5%
4	GFS all-positive APR	83.7%	66.7%
5	all-positive APR	80.4%	72.6%
6	simple backpropagation	75.0%	67.7%
7	multi-instance neural networks	88.0%	82.0%
8	C4.5	68.5%	58.8%
9	1-nearest neighbor (euclidean distance)	/	75%
10	neural network (standard poses)	/	75%
11	1-nearest neighbor (tangent distance)	/	79%
12	neural network (dynamic reposing)	/	91%
13	relational neural networks	84%	80%

Table 1 shows the results obtained with relational neural networks for different settings. These results illustrate that reshuffling gives an improvement of the final accuracy and that copy reshuffling works better than continuous reshuffling in this case. Table 2 compares the best results for relational neural networks with the results for other methods. These other results come from [6], except for the results for multi-instance neural networks [17]. It should be noted that methods 11 and 12 require computation of the molecular surface, which cannot be done using the feature vectors in the data set.

Comparing the different neural network approaches, we see that RNNs do not perform as well as multi-instance neural networks, but substantially better than simple backpropagation. This method ignores the multi-instance character of the musk data set and treats all of the positive instances as positive examples. This is actually some kind of propositional approach. As RNNs perform clearly better than this method, it seems that they are able to learn a real multi-instance concept, which is also a relational concept.

5.2 Trains

The trains data set is an artificially created data set containing a number of trains. Every train consists of a number of cars, carrying some load. Some of the trains are eastbound, the others are westbound. This target concept is based on the properties of the cars of a train and their loads. A data generator for this train problem was used to create the data set [15]. A simple (trains 1 and 2) and a more complicated concept (trains 3 and 4) were defined to generate the data sets. The simple concept defines trains that are eastbound as trains with at least two circle loads, the other trains are westbound. The more complicated concept defines westbound trains as trains that have more than seven wheels in total but not more than one open car with a rectangle load, or trains that have more than one circle load; the other trains are eastbound. There is also a distinction between data sets without noise (trains 1 and 3) and those with 5% noise added (trains 2 and 4).

Training was done with learning rate 0.1 and during 5000 iterations. Results for the different settings can be found in table 1 and a comparison with first-order random forests (FORFs) [20] in table 5. Apparently, the first two data sets, containing 100 samples, are too small to train the network sufficiently. Therefore, better performance for these data sets is obtained with FORF. For data set 3 and 4, the results are very similar to those obtained with FORF. Using reshuffling clearly outperforms no reshuffling for this experiment. This indicates that reshuffling does indeed help to learn relational concepts.

5.3 Mutagenesis

Mutagenesis is a well-known ILP data set [19]. It consists of 230 molecules which have to be classified as mutagenic or not. A structural description of each molecule is given, stating all atoms of the molecule and the bonds between them. In this case, best results were achieved when using 20% of the training set as validation set to do early stopping. This means that after every training iteration the performance on this validation set and on the test set is computed and after training the iteration with the lowest validation error is used to select the test accuracy.

The network was trained for 20000 iterations with a learning rate of 0.5. The large number of training iterations was needed because convergence seems to be quite slow for this experiment. Results for different settings are shown in table 1. Best results are obtained using continuous reshuffling. The reason that this works better than copy reshuffling, could be the large number of training iterations. A comparison with FORF can be found in table 4. For this data set, RNNs achieve substantially better results than FORFs.

5.4 Diterpenes

For the last experiment, the diterpenes data set is used [7]. This data set contains information about 1503 diterpene structures. For each of the 20 carbon atoms

Table 3. Accuracy results for the diterpenes data set compared to other systems. Results for FOIL, RIBL and ICL come from [7], the result for FORF is obtained from [20].

RNN	FORF	FOIL	RIBL	ICL
81%	93%	78%	91%	86%

Table 4. Accuracy results for the mutagenesis data set compared to those for FORF [20]

RNN	FORF
88%	79%

Table 5. Results for trains data sets compared with FORF

	concepts	samples	noise	RNN	FORF
trains 1	simple	100	none	93%	100%
trains 2	simple	100	5%	87%	93%
trains 3	complex	800	none	96%	96%
trains 4	complex	800	5%	89%	90%

in the diterpene structure, multiplicity and frequency are given. The results are shown in table 1. Again, training was done over 20000 iterations and with 0.5 learning rate. A comparison with other results can be found in table 3. For this data set, relational neural networks do not perform very well and reshuffling gives worse results than using no reshuffling at all. It is not very clear why this is so.

5.5 Experimental Conclusions

One must be careful to draw straightforward conclusions from the four experiments as a whole. It seems to be partially problem dependent which training setting gives the best results. Copy reshuffling improves accuracy for musk and trains for instance, but for mutagenesis continuous reshuffling is better and best results for diterpenes are obtained without any reshuffling. Moreover, some results are rather sensitive to changes in the training setting. Even small changes for parameters can produce quite different results. This is also the reason why the training methodology is not uniform in the conducted experiments. There is not one setting that produces acceptable results for all experiments.

Another problem is that convergence tends to be slow. Probably, this is partially due to the use of recurrent neural networks. It is known that these networks are harder to train than feedforward networks. The fact that we increased the number of layers, decreases learnability further. These problems are related to the use of backpropagation as training method, which has problems to backpropagate an error signal over too long distances.

However, some conclusions can be made. For instance, the improvement of the accuracy when using reshuffling for the trains data set is remarkable. Because

this data set is artificially created, we are sure that the concept to be learned is a combination of aggregation and selection. As the results improve so much, this is a strong indication that reshuffling is indeed helping to learn this kind of concepts. If we look at the overall accuracies achieved for the different data sets and compare them to other approaches, we can also conclude that RNNs seem to be able to express relational concepts quite well.

6 Conclusions

In this paper, we presented a novel neural network approach to relational learning. The fact that current relational learners are very limited in making combinations of aggregation and selection is an important motivation for this work. By using neural networks, such combinations can be made in an implicit way and we should be able to avoid a bias to either aggregation or selection.

The structure of a RNN is based on the relational database schema. It is a combination of feedforward and recurrent networks to process a tuple together with its related tuples. The fact that sets of tuples are fed in the recurrent networks as sequences, is also used to improve training and testing. By reordering these sequences, the variation in the data can be increased, which should increase learnability.

Experiments on four different data sets give some insight in the capacities of RNNs. They seem to be able to learn relational concepts reasonably well. The beneficial effect of reshuffling in training and testing could also be demonstrated. But issues as verifying what concepts are actually learned and a training algorithm that is better suited to train this kind of neural networks, are worth further investigation. Understandability is an important issue in ILP, but it is a known problem for neural networks. A lot of work has been done in rule extraction from neural networks, but relational neural networks present some complications. With regard to the training method, a genetic algorithm could be a better method than backpropagation.

Acknowledgements

Hendrik Blockeel is a postdoctoral fellow of the Fund for Scientific Research of Flanders (FWO-Vlaanderen). Werner Uwents is supported by IDO/03/006 'Development of meaningful predictive models for critical disease'.

References

[1] R. Basilio, G. Zaverucha, and V. C. Barbosa. Learning logic programs with neural networks. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2001.

[2] H. Blockeel and M. Bruynooghe. Aggregation versus selection bias, and relational neural networks. In *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003, Acapulco, Mexico, August 11, 2003*, 2003.

[3] M. Botta, A. Giordana, and R. Piola. Fonn: Combining first order logic with connectionist learning. In *Proceedings of the 14th International Conference on Machine Learning*, pages 46–56. Morgan Kaufmann, 1997.
[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
[5] L. De Raedt. Attribute-value learning versus inductive logic programming: the missing links (extended abstract). In D. Page, editor, *Proceedings of the Eighth International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 1–8. Springer-Verlag, 1998.
[6] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71, 1997.
[7] S. Džeroski, S. Schulze-Kremer, K. R. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel. Diterpene structure elucidation from ^{13}C NMR spectra with inductive logic programming. *Applied Artificial Intelligence*, 12(5):363–384, July-August 1998.
[8] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE-NN*, 9(5):768–786, September 1998.
[9] C. Goller and A. Küchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN-96)*, pages 347–352, 1996.
[10] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Conference on Cognitive Science*, pages 531–546, 1986.
[11] A. Knobbe, A. Siebes, and B. Marseille. Involving aggregate functions in multi-relational search. In *Principles of Data Mining and Knowledge Discovery, Proceedings of the 6th European Conference*, pages 287–298. Springer-Verlag, August 2002.
[12] M.-A. Krogel and S. Wrobel. Transformation-based learning using multi-relational aggregation. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, pages 142–155, 2001.
[13] M.-A. Krogel and S. Wrobel. Facets of aggregation approaches to propositionalization. In T. Horváth and A. Yamamoto, editors, *Proceedings of the Work-in-Progress Track at the 13th International Conference on Inductive Logic Programming*, pages 30–39, 2003.
[14] C. Merz and P. Murphy. UCI repository of machine learning databases [`http://www.ics.uci.edu/~mlearn/mlrepository.html`], 1996. Irvine, CA: University of California, Department of Information and Computer Science.
[15] D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new east-west challenge. Technical report, Oxford University Computing Laboratory, Oxford, UK, 1994. Available at ftp.comlab.ox.ac.uk.
[16] C. Perlich and F. Provost. Aggregation-based feature invention and relational concept classes. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 167–176. ACM Press, 2003.
[17] J. Ramon and L. De Raedt. Multi instance neural networks. In *Proceedings of the ICML-Workshop on Attribute-Value and Relational Learning*, 2000.
[18] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, May 1997.
[19] A. Srinivasan, R. King, and D. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.

[20] C. Vens, A. Van Assche, H. Blockeel, and S. Džeroski. First order random forests with complex aggregates. In R. Camacho, R. King, and A. Srinivasan, editors, *Proceedings of the 14th International Conference on Inductive Logic Programming*, pages 323–340. Springer, 2004.

[21] P. J. Werbos. Back propagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.