Experiments with Relational Neural Networks

Werner Uwents WERNER.UWENTS@CS.KULEUVEN.AC.BE
Hendrik Blockeel HENDRIK.BLOCKEEL@CS.KULEUVEN.AC.BE
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract

The fundamental difference between propositional and relational learners is the ability to handle sets. Most current relational learners handle sets either by aggregating over them or by testing the occurrence of elements with specific properties, but a non-trivial combination of both remains a challenge. In this paper, we present a neural network approach to solve relational learning tasks. These relational neural networks are in principle able to make such a combination. We will discuss some experiments that we conducted to test the capacity of our approach.

1. Introduction

Neural networks have been applied to solve many different learning tasks, but their use is still limited to relatively simple data types. Feedforward neural networks, for example, only deal with propositional data, where each tuple consists of a fixed-size vector of real values. Recurrent networks are able to process sequences. However, few attempts have been made to extend the data domain of neural networks beyond this point. Allowing different types of relations in the dataset and relationships between tuples would be a powerful extension. We will present an approach, based on standard neural networks, to learn concepts over such relational data.

The most fundamental difference between propositional and relational learning is the ability to handle sets. These sets are the result of following one-to-many and many-to-many relationships in the dataset. Some approaches to deal with these sets already exist, but they are often biased as will be explained in the next section. There are also some approaches, based on neural networks, that deal with problems very similar to the relational learning task.

The existing work that is probably closest to our approach, is a line of work in the neural networks community on learning from structured data using recursive neural networks or folding architecture networks (Goller & Küchler, 1996; Sperduti & Starita, 1997; Frasconi et al., 1998). These authors describe how to learn from structured data (e.g. logical terms, trees, graphs) and discuss tasks like the identification of substructures. Those tasks relate to the tasks we consider, more or less as inductive logic programming (ILP) relates to our approach and some existing results on learnability of recursive neural networks may carry over to our setting. However, they do not specifically consider the problem of learning aggregate functions over sets and the problem of different types of data. They also focus on learning in graph structures instead of learning in relational databases.

There has also been some research in using neural networks for multi-instance problems (Ramon & De Raedt, 2000). These problems can be seen as a special case of relational learning because they deal with learning over a single set. If one instance in the set is positive, the sample as a whole will be classified as positive. The basic idea behind multi-instance networks is to use a feedforward network to feed all the instances into and to combine all the results with an aggregation function, namely the maximum function.

Neural logic programs (Ramon et al., 2002) are also somewhat similar to our relational neural networks, with as main differences that they are described in a first order logic framework and that, just like for multi-instance neural networks, specific aggregate functions are encoded in advance by the user, instead of learned. Typically, they represent logical conjunctions and disjunctions.

Our approach is also based on neural networks, but it is oriented specifically towards relational data domains and it does not rely on predefined aggregate functions or concepts. We believe that from the point of view of relational learning, the ability to learn aggregate functions is a crucial advantage of this approach.

In the next section, we will discuss the difference

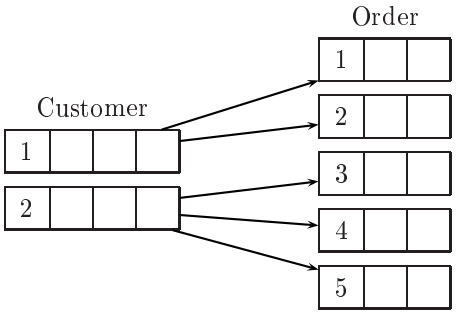


Figure 1. Example of a relational dataset.

between selection and aggregation over sets. Section 3 gives a definition of the structure and training of relational neural networks. The results of some experiments with these networks will be presented in section 4. We end with some conclusions about the presented approach in section 5.

2. Combining Selection and Aggregation

The learning task that we are considering, is a relational learning task. This means that we have a dataset with a number of different relations. For each relation, a set of tuples is given. Each tuple has a number of attribute values and can also have relationships with other tuples. We want to classify all tuples belonging to some target relation R_T, based on their own attribute values and the attribute values of related tuples.

The specific problem that arises in relational datasets, is how to deal with sets. These sets are the result of one-to-many or many-to-many relationships. An example dataset containing the relations customer and order is given in figure 2. Customer tuples can be linked to a number of order tuples. These linked tuples form a set. The first customer is linked to two orders, the second to three, so the number of tuples in these sets can vary. This is the reason why we can not reduce this dataset to a propositional dataset.

Relational learners can be divided into two categories, depending on how they handle one-to-many and many-to-many relationships, or, equivalently, how they handle sets of tuples (Blockeel & Bruynooghe, 2003). Most current relational learners are restricted to one of these categories. This imposes a significant, possibly undesirable bias on these learners.

Methods in the first category, selective methods, handle sets by looking at properties of their elements. A set S is examined by testing a condition of the form $\exists\, x \in S : P(x)$. This $P(x)$ can be a complicated criterium, but it only considers the attributes of a single tuple. Using this method, we can learn a concept like 'people with at least one son'.

Aggregating methods, the second category, compute a function $F(S)$ over a set S of tuples. This reduces the set to a single value. One example of such a function is the cardinality function that simply counts the number of elements in the set. With such a function, we could express a concept like 'people with two children'.

Many approaches to relational learning rely on some kind of propositionalization of the relational data. On the resulting propositional dataset, a propositional learner can be used. An example of this is the RELAGGS system (Krogel & Wrobel, 2001). The propositional data is extended with some extra attributes, which are the result of evaluating predefined aggregate functions over the related data. This method, however, cannot learn undefined aggregate functions. How combinations of aggregation and selection could be learned, is not explained by the authors.

This is a problem when we want to express a concept like 'people with two sons'. This concept clearly combines aggregation and selection: we have to select all males from the set of children and then count them to check this criterium. Other concepts may require different kinds of combinations of selection and aggregation. As aggregation and selection are both very natural operations, a relational learning system should be able to combine both in the models it builds. However, these combinations can be quite complicated and diverse, and they may depend on the structure of the dataset and the relations in it.

For instance, probabilistic relational models (PRMs), as defined by (Getoor et al., 2001), cannot learn the concept of 'people having two sons' without having separate relations for sons and daughters. Manually introducing these separate relations of course presupposes that the user is aware of the possible importance of these concepts. Alternatively, one could predefine a large number of aggregate functions that have appropriate selection conditions built in. In that case, a search through a space of aggregate functions is needed. The power of this approach largely depends on which aggregate functions are defined.

In inductive logic programming (ILP), one could for instance define aggregate functions as background knowledge. Then, e.g., the rule `p(X) :- count(Y, (child(X,Y), male(Y)), 2)` expresses the concept of people having two sons. The main difficulty here is that the second argument of the count metapredicate is itself a query that is the result of a search

through some hypothesis space. It is not obvious how such a search should be conducted. The many results in ILP on how to search a first-order hypothesis space efficiently (Nienhuys-Cheng & De Wolf, 1997), do not consider the case where the resulting hypothesis will be used as the argument of a metapredicate.

ILP-like approaches that do not include aggregate functions, can still express the concept as, e.g., 'the person has a male child x and a male child y and $x \neq y$ and there does not exist a child z such that z is male and $z \neq x$ and $z \neq y$'. But in practice, the length of this rule, as well as the occurrence of a negation, make it difficult to learn. The comprehensibility of the result is also negatively influenced.

Knobbe et al. (2002) are, to our knowledge, the first to present a method that performs a systematic search in a hypothesis space (in this case, that of 'selection graphs') where hypotheses combine aggregation and selection. Their approach is however limited to monotone aggregate functions, which limits its applicability somewhat (for instance, sum and average are not monotone), and to selecting aggregate functions from a limited set given by the user.

Our relational neural networks would have as advantage over the other approaches that they can learn an aggregate function, without that function being predefined and with selection possibly integrated in it. Training the relational neural network automatically constitutes a search through aggregations and selections simultaneously.

3. Relational Neural Networks

Assume that we have a dataset with a target relation R_T and some other relations $R_1, \ldots, R_N$. We denote the attribute sets of R_i by U_i. For any relation R, we define

- $S_1(R)$: $R_i \in S_1(R)$ iff each tuple $t \in R$ is connected to exactly one tuple in R_i. This means R has a one-to-one or many-to-one relationship with R_i, in which R participates completely.
- $S_{01}(R)$: $R_i \in S_{01}(R)$ iff each tuple $t \in R$ is connected to at most one tuple in R_i. This is, again, a one-to-one or many-to-one relationship between R and R_i, but now with partial participation.
- $S_N(R)$: $R_i \in S_N(R)$ iff each tuple $t \in R$ is connected to zero, one or more tuples in R_i. This is a one-to-many or many-to-many relationship between R and R_i, with complete or partial participation.
- $S_U(R)$: $R_i \in S_U(R)$ iff R_i is a relation of the relational dataset, but not in $S_1(R)$, $S_{01}(R)$ or $S_N(R)$. This means R is not directly connected to R_i.

Given a tuple $t \in R_T$, we want to classify it based on the information contained in the tuple and in any tuples linked to this tuple. For a relation R_i, we use U_i to denote the original attribute set of that relation. All attributes in U_i must be real values, as this is the only type of input a neural network can process. Other types of attributes need to be converted to real values first. We use I_i to denote the attribute set actually used as input to our neural network. One might expect that $I_i = U_i$, but there will be some small differences:

- For R_T, the target relation, $I_T = U_T - \{C\}$, where C is the class attribute.
- For any $R_i \in S_{01}(R)$, there can be a tuple $t \in R$ for which there exists no tuple $s \in R_i$ that t is directly connected to. As neural networks do not have a distinguished encoding for null values, we will use an extra attribute E_i that indicates whether the link to R_i yielded a tuple or not. $I_i = U_i \cup \{E_i\}$.
- The same problem arises for $R_i \in S_N(R)$, so here also $I_i = U_i \cup \{E_i\}$.

Based on the above, we can construct a relational neural network that classifies $t \in R_T$ based on its own attribute values as well as those of related tuples. For each tuple $t \in R_T$, we construct a tuple t' with attributes

$$I_T \cup \Big(\bigcup_{i:R_i \in S_1(R_T) \cup S_{01}(R_T)} O_{1i} \Big) \cup \Big(\bigcup_{i:R_i \in S_N(R_T)} O_{Ni} \Big)$$

with I_i as defined above, O_{1i} a set of attributes that are the output values of a feedforward neural network taking I_i as input values and O_{Ni} a set of attributes that are the output values of a recurrent neural network taking I_i as input values. This tuple t' has then a fixed set of attributes which can be used to feed into a feedforward neural network. The output of this network gives us the final result of our classifier.

In the described approach, sets resulting from relations in $S_N(R_T)$ are processed using recurrent neural networks. These networks are able to process tuple sequences of indefinite length. However, we are presenting the tuples to the network in some imposed order while the sets are actually unordered. As we will see, this fact can be exploited in training the network.

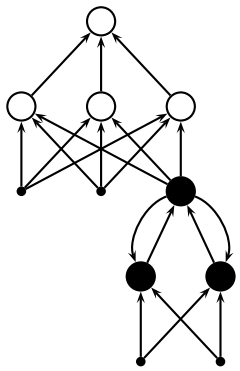


Figure 2. Example of the structure of a relational neural network.

The precise structure of the different neural networks in our classifier must be defined now. We take both the feedforward and recurrent networks to have two layers. The ideal number of neurons in each layer needs to be tuned by conducting experiments, there is no straightforward rule to determine this.

For the recurrent networks, we also have to define which recurrent connections are allowed. The most expressive recurrent network is a fully connected network in which each neuron has connections with all other neurons. But as this makes the number of connections increase quadratically when the number of neurons increases, we prefer the Jordan recurrent network (Jordan, 1986).

In this kind of recurrent network, each neuron in the second layer is connected with all neurons in the first layer. 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 gives us a good trade-off between expressiveness and the number of neurons and connections in the network.

A small example of a relational network is given in figure 3. The two attributes of the target relation are fed into the feedforward part of the network (white neurons). The two attributes of the tuples of another relation, linked to the target relation, are fed into the recurrent part of the network (black neurons). The output of this recurrent part is used as extra input to the feedforward part.

The technique of adding to t the O_{1i} and O_{Ni} attributes that summarize related tuples, can be repeated for those tuples, thus also incorporating information in indirectly linked tuples. In the end, this yields a hierarchical structure where each node is a neural network that takes the attributes of a relation and the outputs of its children as input and propagates the result to its parent node.

Training this relational neural network can be done with an adapted form of the standard backpropagation algorithm. The feedforward neural networks in the relational network are trained with standard backpropagation. The recurrent networks are trained with backpropagation through time (BPTT) (Werbos, 1990). The key idea to BPTT is the unfolding of the recurrent network into a feedforward network.

As many folds (copies of the original network) are created as there are instances in the input sequence and recurrent connections are converted into feedforward connections between successive folds. The resulting feedforward network is trained using the standard backpropagation algorithm, but with one important restriction: since all folds have been created by replicating the original network, weights in all folds should be the same.

The fact that sets are fed into the recurrent network in some imposed order, can be used to improve our training algorithm. This can be done by reshuffling the sequence and presenting the set to the recurrent network in a different order. Two possibilities can be considered: reshuffling after every training iteration and expanding the training set by adding reshuffled copies of the initial instances.

4. Experiments

To evaluate this approach, we have performed experiments on the musk and trains datasets. The musk dataset, available from UCI (Merz & Murphy, 1996), is an example of a multi-instance learning task. As mentioned above, this can be seen as a special case of relational learning. In this dataset, each example describes a molecule. For each example, several poses (instances) are given, each with 166 attributes. If at least one of these poses has some property, the molecule is said to be musk.

There are two versions of this dataset, musk 1, containing 92 molecules, and musk 2, containing 102 molecules, which differ in size. Musk 2 has more conformations per molecule than musk 1. Several learning approaches have been compared on this dataset (Dietterich et al., 1997). To be able to compare our results with these results, we conduct our experiments in the same setting, namely ten-fold cross-validation.

Overall results are summarized in table 1 (results for multi-instance neural networks come from Ramon and De Raedt (2000), other results from Dietterich et al. (1997)). The tangent distance and dynamic reposing

Table 1. Classification accuracies on the musk dataset.

method	musk 1	musk 2
iterated-discrim APR	92.4%	89.2%
GFS elim-kde APR	91.3%	80.4%
GFS elim-count APR	90.2%	75.5%
GFS all-positive APR	83.7%	66.7%
all-positive APR	80.4%	72.6%
simple backpropagation	75.0%	67.7%
multi-instance neural networks	88.0%	82.0%
C4.5	68.5%	58.8%
1-nearest neighbor (euclidean distance)	/	75%
neural network (standard poses)	/	75%
1-nearest neighbor (tangent distance)	/	79%
neural network (dynamic reposing)	/	91%
relational neural networks	89.1%	85.3%

Table 2. Training configurations for the musk dataset (n_1 = number of neurons in first layer of the recurrent component, n_2 = number of neurons in second layer of the recurrent component, η = learning rate, μ = momentum term).

	dataset	n_1	n_2	η	μ	reshuffle	iterations	accuracy
1	musk 1	50	10	0.5	0.2	every it.	190	84.8%
2	musk 1	50	10	0.5	0.2	none	110	88.0%
3	musk 1	50	10	0.5	0.2	30 copies	10	89.1%
4	musk 2	80	20	0.5	0.2	every it.	40	76.5%
5	musk 2	80	20	0.5	0.2	none	240	85.3%
6	musk 2	40	15	0.5	0.2	none	50	77.5%
7	musk 2	40	15	0.5	0.2	5 copies	20	80.4%

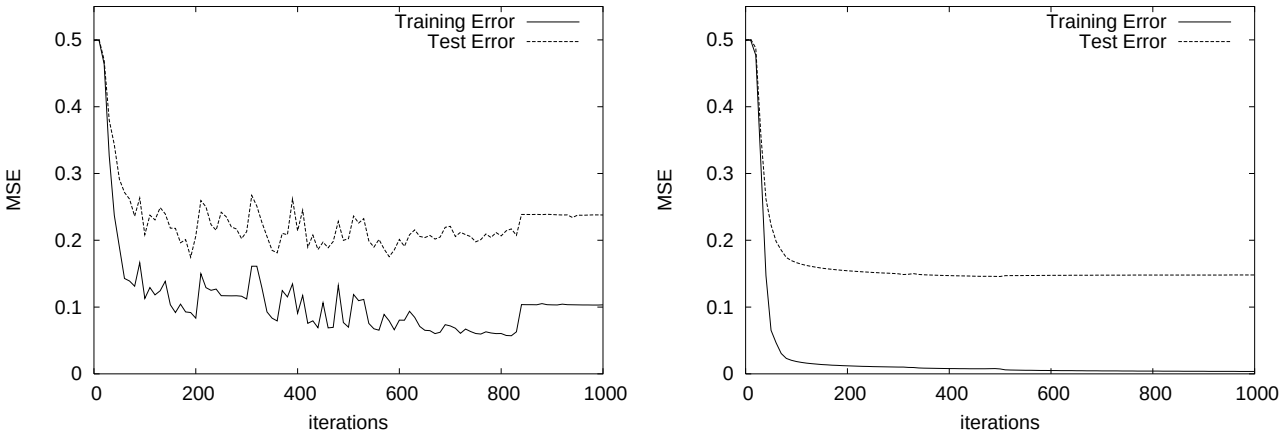
technique require computation of the molecular surface, which cannot be done using the feature vectors included in the dataset. A comparison of different configurations for the relational neural networks, is shown in table 2.

These results show that relational neural networks are performing quite well. They give results that are a lot better than simple backpropagation and even better than multi-instance neural networks. The latter can only be the result of better parameters because the hypothesis space searched by relational neural networks, H_{RNN}, is a superset of the hypothesis space H_{MINN} searched by multi-instance neural networks. This means that the hypothesis in H_{RNN} that best approximates the target hypothesis, must also be in H_{MINN}. The accuracy of iterated-discrim axis-parallell rectangles (iterated-discrim APR), the best method, is still significantly higher, but this method is specifically designed for this kind of problems.

When we look at the differences between different configurations for training the relational neural networks, we see that reshuffling the training data after every iteration gives poor results. However, expanding the training set with reshuffled copies does improve training. Although the training set becomes larger, the accuracy is better and the necessary number of iterations is reduced. It also helps to avoid overfitting.

Figure 3 shows us why reshuffling after every iteration is not working well. Reshuffling after every iteration actually changes the training set continuously. This means that the error function also changes continuously, and what was the gradient in the previous iteration may be unrelated to the gradient in this iteration. This makes it difficult for the backpropagation algorithm to converge. The mean square error on training and test set is very jagged in this case (see figure 4(a)). This is not the case for adding reshuffled copies to the training set (see figure 4(b)). Adding reshuffled copies has a similar effect as enlarging the dataset with new samples.

Figure 3. Mean square error (MSE) for training and test set in function of the number of training iterations.



(a) With reshuffling after every iteration.

(b) With adding reshuffled copies.

Table 3. Accuracies on the trains dataset, comparing relational neural networks with first order decision trees using random forrests (RNN 1 is without reshuffling, RNN 2 is with reshuffled copies).

dataset	concept	samples	noise	RNN 1	RNN 2	FORF
trains 1	simple	100	none	80%	95%	100.0%
trains 2	simple	100	5%	78%	89%	92.8%
trains 3	simple	1000	none	100%	100%	100.0%
trains 4	complex	800	none	89.4%	97.8%	96.1%
trains 5	complex	800	5%	84.5%	89.8%	90.3%

The second tested dataset is the train dataset based on the Michalski train problem. This problem was invented by Ryszard Michalski around 25 years ago (Michalski, 1980). The aim is to find a concept which explains why trains are travelling eastbound or westbound. Every train consists of a number of cars, carrying some load. The concept is based on the properties of these cars and their loads.

We used a generator for this train problem to create a dataset (Michie et al., 1994). Two different concepts are used for the generation, a simple and a more complicated. 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.

First-order decision trees using random forests (Vens et al., 2004) are another approach to tackle the problems combining selection and aggregation. We compare our results with results obtained with the latter method. A complete overview of the results is given in tables 3 and 4. The used datasets are generated to test different settings: a simple versus a more complicated concept and noise versus no noise. The tests were done with five-fold cross-validation.

What we see for the simple concept (datasets 1, 2 and 3), is that 100 samples is not enough to learn the concept completely. Taking a dataset with 1000 samples, however, we can reach 100% accuracy. Adding 5% noise results in an accuracy that is a little more than 5% lower, so our classifier seems to be rather noise resistant.

For the more complicated concept (datasets 4 and 5), the results are quite similar to those for first-order decision trees using random forests. The effect of noise is the same as for the simple concept. For both the simple and complicated concept, there is a remarkable difference between training without reshuffling and with added reshuffled copies.

Table 4. Training configurations for the trains dataset (n_1 = number of neurons in first layer of the recurrent component, n_2 = number of neurons in second layer of the recurrent component, η = learning rate, μ = momentum term).

dataset	n_1	n_2	η	μ	reshuffle	iterations	accuracy
trains 1	20	10	0.1	0.0	none	250	80%
trains 1	20	10	0.1	0.0	10 copies	100	95%
trains 2	20	10	0.1	0.0	none	300	78%
trains 2	20	10	0.1	0.0	20 copies	150	89%
trains 3	20	10	0.1	0.0	none	40	100%
trains 3	20	10	0.1	0.0	10 copies	10	100%
trains 4	60	40	0.2	0.1	none	300	89.4%
trains 4	60	40	0.2	0.1	20 copies	200	97.8%
trains 5	60	40	0.2	0.1	none	200	84.5%
trains 5	60	40	0.2	0.1	20 copies	130	89.8%

5. Conclusions

We have presented a novel, neural network based approach to relational learning. The approach consists of constructing a neural network that reflects the structure of the relational dataset. This relational neural network may contain both feedforward and recurrent parts. The training algorithm for this relational network is based on the standard backpropagation (through time) algorithm.

Our approach was tested on two datasets. It turned out that relational neural networks are performing quite well compared with other methods. They can deal with noisy data and the expressive power of Jordan recurrent networks seems to be sufficient to learn the desired concepts.

While many open questions remain regarding the optimal architecture of these relational neural networks, their learning behaviour, the optimal learning methodology, etcetera, we did obtain a first important result from these experiments: the 'copy reshuffling' method to temove the effect of set ordering is clearly superior to the other reshuffling method. Its effects include a higher final accuracy, less iterations needed to train the network and reducing the risk of overfitting. These advantages outweigh the disadvantage of having a larger training set.

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'. We thank Celine Vens for generating the train dataset and obtaining results with first-order random forests.

References

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

Dietterich, T. G., Lathrop, R. H., & Lozano-Pérez, T. (1997). Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, *89*, 31–71.

Frasconi, P., Gori, M., & Sperduti, A. (1998). A general framework for adaptive processing of data structures. *IEEE-NN*, *9*, 768–786.

Getoor, L., Friedman, N., Koller, D., & Pfeffer, A. (2001). Learning Probabilistic Relational Models. In S. Dzeroski and N. Lavrac (Eds.), *Relational data mining*, 307–334. Springer-Verlag.

Goller, C., & Küchler, A. (1996). Learning task-dependent distributed representations by backpropagation through structure. *Proceedings of the IEEE International Conference on Neural Networks (ICNN-96)* (pp. 347–352).

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference on Cognitive Science* (pp. 531–546).

Knobbe, A., Siebes, A., & Marseille, B. (2002). Involving aggregate functions in multi-relational search. *Principles of Data Mining and Knowledge Discovery, Proceedings of the 6th European Conference* (pp. 287–298). Springer-Verlag.

Krogel, M.-A., & Wrobel, S. (2001). Transformation-based learning using multi-relational aggregation. *Proceedings of the Eleventh International Conference on Inductive Logic Programming* (pp. 142–155).

Merz, C., & Murphy, P. (1996). UCI repository of machine learning databases [`http://www.ics.uci.edu/~mlearn/mlrepository.html`]. Irvine, CA: University of California, Department of Information and Computer Science.

Michalski, R. (1980). Pattern Recognition as Rule-Guided Inductive Inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *2*, 349–361.

Michie, D., Muggleton, S., Page, D., & Srinivasan, A. (1994). *To the international computing community: A new east-west challenge* (Technical Report). Oxford University Computing Laboratory, Oxford, UK. Available at ftp.comlab.ox.ac.uk.

Nienhuys-Cheng, S.-H., & De Wolf, R. (1997). *Foundations of Inductive Logic Programming*, vol. 1228 of *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*. New York, NY, USA: Springer-Verlag.

Ramon, J., & De Raedt, L. (2000). Multi instance neural networks. *Proceedings of the ICML-Workshop on Attribute-Value and Relational Learning*.

Ramon, J., Driessens, K., & Demoen, B. (2002). Neural logic programs. `http://www.cs.kuleuven.ac.be/~janr/nlptechrep.ps`, to be published as a technical report.

Sperduti, A., & Starita, A. (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, *8*, 714–735.

Vens, C., Van Assche, A., Blockeel, H., & Džeroski, S. (2004). First order random forests with complex aggregates. *Proceedings of the 14th International Conference on Inductive Logic Programming* (pp. 323–340). Springer.

Werbos, P. J. (1990). Back propagation through time: What it does and how to do it. *Proceedings of the IEEE* (pp. 1550–1560).