KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Neural networks for relational data

Werner UWENTS

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

October 2012

Neural networks for relational data

Werner UWENTS

Jury:
Prof. dr. ir. P. Van Houtte, chair
Prof. dr. ir. H. Blockeel, supervisor
Prof. dr. ir. M. Bruynooghe
Prof. dr. ir. M. Van Hulle
Prof. dr. M. Gori
(University of Siena)
Prof. dr. B. Hammer
(Bielefeld University)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2012

© Katholieke Universiteit Leuven – Faculty of Engineering
Kasteelpark Arenberg 1, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2012/7515/115
ISBN 978-94-6018-581-6

Preface

I would like to express my gratitude to all people who contributed, one way or another, to this dissertation. First of all, I want to thank Hendrik, my supervisor. His patience and encouragement along the way were much appreciated. I am grateful for the opportunity he and Maurice gave me to do a PhD in the ML group. I want to thank the members of the jury, prof. Van Houtte, prof. Bruynooghe, prof. Van Hulle, prof. Hammer and prof. Gori, for their time and their helpful remarks. I also need to thank Anneleen and Celine for helping me with the experiments using FORFs, and Gabriele and Franco for their help with the experiments using GNNs.

Special thanks go to prof. Gori and the people of his group - Gabriele, Franco, Monica, Edmondo and all the others - for their hospitality during my stay in Siena. Furthermore, there are of course the many people that were part of the Leuven ML group during all these years and who made it a very vivid and interesting place to work. In particular, there are the people I shared an office with - Anneleen, Celine, Stefan, Daan, Joaquin, Fabrizio and Chris - and who made for many interesting and funny conversations.

I also want to mention my friends at Salvast and Servotte. Out of the office, I spent many good times with them during these years. And last but not least, I want to thank my parents and my sister and brother for their never-ending moral support.

Abstract

The topic of this dissertation is situated in the field of machine learning. In general, machine learning is concerned with the development of algorithms that can learn from empirical datasets containing input and target patterns. What will be considered within the context of this dissertation, is learning predictive models from relational data. While propositional data is represented by a single tuple for each input pattern, in relational data every input pattern consists of multiple tuples with relationships between the tuples. Learning approaches for this type of data should always take into account properties of the input pattern as a whole. This could be structural features, which are properties based on substructures of the input pattern, or aggregate features, which are properties based on subsets of the collection of input pattern tuples. The focus of this dissertation will be on aggregate features.

Artificial neural networks are one particular method in the field of machine learning. Originally, they were developed as a method for simple data domains in which every input pattern is represented by a single input vector of fixed size. There has also been quite some interest in the use of neural networks for relational data domains, but most of this research has focused on learning structural features. The aspect of learning aggregate features with neural networks has received very little attention so far, although this topic might be more promising. Learning structural features with neural networks gives rise to a number of problems but learning aggregate features is a task that suits them very well. Therefore, a number of neural network approaches will be developed in this dissertation that deal with learning aggregate features from relational data.

The first part of the dissertation focuses on methods for data containing only a single relation. This means that each input pattern can be represented as a single, homogenous collection of tuples, with all tuples of the same type. In the second part of the dissertation, methods for data containing multiple relations are considered. In this case, every input pattern is represented by multiple

collections of tuples, with different types of tuples for different collections. For both cases, similar neural network methods are developed and empirically tested on a number of datasets. These approaches are compared, both from a theoretical and an experimental point of view, to related approaches such as multi-instance methods or neural networks for graph structures. The experimental results show that the new methods can be viable options for tackling specific kinds of problems and have a performance that is in many cases comparable to the performance achieved with other methods or in some cases even better.

Beknopte samenvatting

Het onderwerp van deze dissertatie situeert zich in het onderzoeksveld machine learning. Dit onderzoeksdomein spitst zich toe op de ontwikkeling van algoritmes die in staat zijn te leren op basis van empirische gegevens, bestaande uit invoerpatronen met daaraan gekoppelde doelvariabelen. In de context van deze dissertatie wordt er voornamelijk gekeken naar het leren van voorspellende modellen op basis van relationele gegevens. Dit zijn gegevens waarbij elk invoerpatroon bestaat uit meerdere tupels met mogelijk bepaalde verbanden tussen de tupels, dit in tegenstelling tot propositionele gegevens waarbij elk invoerpatroon bestaat uit een enkel tupel. Leermethodes voor relationele gegevens zullen steeds rekening moeten houden met de eigenschappen van het invoerpatroon als geheel. Dat kunnen zowel structurele eigenschappen zijn, gebaseerd op substructuren van het invoerpatroon, als geaggregeerde eigenschappen, dat zijn eigenschappen die informatie uit een aantal tupels van het invoerpatroon samenvatten. De klemtoon in deze dissertatie zal liggen op geaggregeerde eigenschappen.

Artificiële neurale netwerken zijn een specifieke methode in machine learning. Ze werden in eerste instantie ontwikkeld als een methode voor simpele, propositionele gegevens waarbij elk invoerpatroon wordt voorgesteld door een enkele vector met vaste lengte. Er is daarnaast ook heel wat onderzoek verricht naar het gebruik van neurale netwerken in relationele domeinen, maar het grootste deel van dit onderzoek concentreerde zich op structurele eigenschappen. Het leren van geaggregeerde eigenschappen binnen de context van neurale netwerken is een onderwerp dat tot dusver weinig aandacht gekregen heeft, ondanks het feit dat net daarin heel wat potentieel zit. Het leren van structurele eigenschappen met neurale netwerken geeft immers aanleiding tot een aantal problemen, zodat ze geschikter zijn voor het leren van geaggregeerde eigenschappen. Om die reden worden er in deze dissertatie een aantal methodes ontwikkeld, gebaseerd op neurale netwerken, die zich richten op het leren van geaggregeerde eigenschappen uit relationele gegevens.

Het eerste deel van de dissertatie concentreert zich op methodes voor gegevens die slechts een enkele relatie bevatten. Dat wil zeggen dat elk invoerpatroon kan voorgesteld worden als een homogene groep tupels, allemaal van hetzelfde type. In het tweede deel worden methodes bekeken die kunnen omgaan met meerdere relaties. In dat geval bestaat elk invoerpatroon uit verschillende groepen van tupels, waarbij de types van de tupels tussen de groepen kunnen verschillen. Voor beide gevallen worden er gelijkaardige methodes uitgewerkt die empirisch getest worden op een aantal gegevensbanken. De verschillende methodes worden vergeleken, zowel in theoretisch als experimenteel opzicht, met verwante methodes zoals algoritmes voor multi-instance problemen of neurale netwerken voor grafen. De experimentele resultaten tonen aan dat deze nieuwe methodes een bruikbare optie zijn voor bepaalde problemen en dat hun performantie in vele gevallen vergelijkbaar is met die van andere methodes, in sommige gevallen zelfs beter.

Contents

Abstract iii

Contents vii

1 Introduction 1

1.1 Machine Learning . . . 1

1.1.1 Learning Tasks . . . 1

1.1.2 Relational Learning . . . 2

1.2 Motivation and Goals . . . 3

1.3 Outline . . . 4

1.4 Publications . . . 5

2 Neural Networks 7

2.1 Neural Networks . . . 7

2.1.1 Feedforward Neural Networks . . . 7

2.1.2 Backpropagation . . . 9

2.1.3 Resilient Propagation . . . 13

2.2 Neural Networks for Relational Domains . . . 14

2.2.1 Learning Structural Features . . . 14

2.2.2 Learning Aggregate Features . . . 15

3 Learning Aggregate Features **17**

3.1 Introduction 17

3.1.1 Learning from Bags 17

3.1.2 Bag Learning Settings 20

3.1.3 Relational Learning 23

3.1.4 Aggregation 24

3.1.5 Propositionalization and Direct Approaches 26

3.2 Aggregate Features 27

3.2.1 Aggregation and Selection 27

3.2.2 Aggregate Functions 28

3.3 Learning Approaches 30

3.3.1 Propositionalization 30

3.3.2 Inductive Logic Programming 32

3.3.3 Metric-Based Methods 35

3.4 Conclusions 37

4 Neural Networks for Learning Aggregate Features **39**

4.1 Simple Aggregate Network Structures 39

4.1.1 Padded Feedforward Networks 40

4.1.2 Symmetric Feedforward Networks 42

4.1.3 Aggregated Feedforward Networks 44

4.1.4 Recurrent Networks 48

4.2 Aggregate Cascade-Correlation Networks 49

4.2.1 The Original Cascade-Correlation Network 50

4.2.2 Cascade-Correlation with Aggregate Units 52

4.2.3 Aggregate Cascade-Correlation Training 54

4.3 Experiments 55

4.3.1 Simple Aggregates 55
4.3.2 Trains 62
4.3.3 Musk 65
4.3.4 Thioredoxin-Fold Proteins 70
4.3.5 Financial 71
4.4 Conclusions 72

5 Neural Networks for Structured Data 75

5.1 Recurrent Networks 75
5.1.1 Unfolding Networks 76
5.1.2 Shift Operator 78
5.2 Recursive Networks 78
5.2.1 Directed Positional Acyclic Graphs 78
5.2.2 Generalized Shift Operator 80
5.2.3 Encoding Network 80
5.3 Graph Neural Networks 81
5.3.1 Encoding Network 83
5.3.2 Contraction Mappings 86
5.3.3 Training Algorithm 86
5.4 Problems 88
5.4.1 Vanishing Gradient Problem 88
5.4.2 Approximation Capabilities 90
5.5 Conclusions 92

6 Relational Neural Networks 93

6.1 Relational Neural Networks 93
6.1.1 Relational Data 93
6.1.2 Encoding Network 95

6.1.3 Feeding and Training 99

6.1.4 Relational versus Recursive and Graph Networks 101

6.2 Experiments 103

6.2.1 Subtrees 104

6.2.2 Mutagenesis 106

6.2.3 Biodegradability 114

6.3 Conclusions 122

7 Relational Cascade-Correlation Networks 125

7.1 Relational Cascade-Correlation Networks 125

7.1.1 Cascade-Correlation for Relational Domains 125

7.1.2 Linearizing Relational Data 127

7.1.3 Relational Cascade-Correlation Algorithm 127

7.2 Experiments 128

7.2.1 Subtrees 128

7.2.2 Mutagenesis 133

7.2.3 Biodegradability 133

7.3 Conclusions 134

8 Conclusions 137

8.1 Summary 137

8.1.1 Aggregation 137

8.1.2 Relational Domains 139

8.2 Evaluation 140

8.3 Future Work 140

Bibliography 143

Curriculum Vitae 149

List of publications **151**

Chapter 1

Introduction

In this chapter, the topic of the dissertation is introduced and some context is given. To start with, machine learning is explained, the research field in which this work is situated. Next, the goals of this work will be discussed and an outline of the rest of text will be given.

1.1 Machine Learning

Machine learning is a subfield of artificial intelligence dealing with the development of algorithms that allow computers to learn. Learning in this context can be defined as the automatic acquisition of knowledge from some kind of experience about the problem at hand. In this way, a computer program can improve its own performance. Experience is typically provided as a number of empirically observed patterns and is referred to as the *data*. The knowledge that is obtained from the data throughout the learning process, is referred to as the *model*. It can have different forms, depending on the method.

1.1.1 Learning Tasks

There are different possible learning tasks [35]. In *descriptive learning* for instance, the goal is to find interesting regularities in the data. Another learning task is *predictive learning*, where the goal is to find a model that can predict some property of the data. This property is called the *target*. The data will typically provide a number of example patterns with associated, known targets

from which the model can be learned. Afterwards, this model can be used to make predictions for new patterns for which the target is not known. This dissertation will only be concerned with predictive learning, not with other kinds of learning.

A further distinction can be made between *classification* and *regression* tasks. In classification, each data pattern has an associated target class, which has to be predicted. So the prediction is a discrete value. As a consequence, a prediction by the model is simply right or wrong. The performance of a classification model can therefore be measured by the *accuracy*, the percentage of correctly classified patterns. In regression, a real target value is associated with each data pattern and the model has to predict this real value. The prediction will not just be right or wrong, but will have some error with regard to the target. This error is the difference between the predicted value and the target value. A performance measure that is often used in this case is the *mean squared error*.

1.1.2 Relational Learning

Nothing has been mentioned so far about the data patterns used as input for the model. A lot of research in machine learning has focused on the most simple data format. In this format, each pattern is described by a *tuple*. A tuple contains values for a fixed number of *attributes*. This setting is called *attribute-value learning* or *propositional learning*. This approach is probably still the most widely used approach in machine learning.

However, more complicated data structures might be needed in some cases. If the data under consideration is sequential for instance, it makes sense to represent the data patterns as sequences of tuples instead of single tuples and thus move from propositional learning to *learning from sequences*. In the next chapter, *learning from bags* will be discussed, which is very similar to learning from sequences, except that sequences are ordered while bags are not.

Sequences or bags are still very simple structures. It is possible to go further and consider learning from trees or graphs. All these settings belong to the domain of *relational learning* [8, 13], in which data patterns are described as collections of elements with relationships between the elements. More details will be given in later chapters. The topics discussed in Chapters 5, 6 and 7 involve relational learning settings.

1.2 Motivation and Goals

The key idea behind this research is to develop a method for dealing with relational data based on neural networks. This idea will be worked out step by step in the following chapters. More specifically, the focus will be on learning models that involve aggregation rather than structural features. Learning aggregate features is a very challenging task, but it is definitely worth studying in detail because it can be argued that it is one of the key elements of relational learning. However, there are still relatively few methods with an explicit focus on learning these aggregate features. In neural networks research for instance, there has been much more interest in adapting networks to structured data domains.

The approach taken in this dissertation, can be compared to existing work in the context of inductive logic programming where several strategies have been devised to embed aggregate features in the learned models. These features are built using special aggregate predicates that are included in the knowledge base. Of course, a neural network method will differ fundamentally from these inductive logic programming methods with regard to the learned model and the learning algorithm. The model will not be expressed in logic but in numerical parameters and learning will be done by numerical optimization instead of traversing a search space. It is interesting to explore this radically different approach because it provides a different point of view on this type of learning problems, with different advantages and disadvantages.

Although the popularity of neural networks has decreased over the years, mainly in favor of kernel methods, they are well suited for this particular learning setting. Basically, they approximate a function by using a composition of units representing simpler functions. This makes it rather straightforward to embed special units for aggregation in the network that can be adapted in a very gradual and smooth way. For most other machine learning methods, it is difficult to do something similar to this. There are for instance also kernels for sets, which rely on some form of aggregation as well. However, different types of kernels have been proposed with different advantages and disadvantages. Most of these kernels also have a number of parameters. The most suitable combination of type and parameters differs from case to case. The obvious path to tackle this kind of problems would be to conduct a search through different kernel types and parameters. But this is in fact very similar to the inductive logic programming setting where a search is conducted to find the right aggregate predicate. Therefore, this kernel approach would be closer to this kind of methods than to the approach taken here.

For the first chapters, the focus will be on methods for data containing only

a single relation. This means that each input pattern can be represented as a single, homogenous set of tuples, with all tuples of the same type. In the second part of the dissertation, methods for data containing multiple relations are considered. In this case, every input pattern is represented by multiple sets of tuples, with different types of tuples for different sets. For both cases, similar neural network methods are developed and empirically tested on a number of datasets.

1.3 Outline

Chapter 2 discusses the basics of neural networks and serves as a short introduction to the subject. The rest of the text falls apart in two main parts. The first part deals with neural networks for learning aggregate features. This topic is discussed in Chapters 3 and 4. The second part is a logical extension of this learning setting and will deal with neural networks for relational domains. This part comprises Chapters 5, 6 and 7.

In Chapter 3, the different aspects of learning aggregate features are discussed. After explaining learning from bags, different types of aggregate features and possible strategies to learn them are considered. At the end of the chapter, existing methods for learning aggregate features are discussed.

In Chapter 4, different types of neural network structures are presented that are able to learn aggregate features. First, a number of network types are proposed that are extensions of standard feedforward and recurrent networks. Second, a cascade-correlation network is considered in which several types of networks are combined. Important issues such as training are discussed as well. The performance of all these networks is also tested in a series of experiments.

Chapter 5 makes the transition towards learning from relational data and gives an overview of some existing neural network approaches for learning from structured data. It covers recurrent, recursive and graph neural networks. Details of these methods are explained and some advantages and disadvantages of the different methods are discussed.

In Chapter 6, extensions of standard feedforward and recurrent networks to relational domains are presented, similar to what was done in Chapter 4 for learning aggregate features. Aspects such as the encoding of the network from the data description and training of the network are discussed. A number of experiments are also conducted to assess the performance of this method.

Chapter 7 also deals with neural networks for relational domains, but in this chapter a cascade-correlation network is proposed. This is, again, similar to

what was done in Chapter 4 for learning aggregate features. Data conversion and training of the network are discussed. The same experiments as in Chapter 6 are conducted to compare the performance of the different methods.

Finally, Chapter 8 draws some general conclusions and discusses directions for future work.

1.4 Publications

Chapters 4, 6 and 7 contain original research and describe novel methods developed by the author. Most of this has been published before in conference and journal papers. The other chapters mainly serve as context and background. They also contain discussions of methods developed by other authors that relate in some way to this work.

The publications related to Chapter 4 are:

- Werner Uwents and Hendrik Blockeel. A comparison between neural network methods for learning aggregate functions. In *Discovery Science*, volume 5255 of *Lecture Notes in Computer Science*, pages 88–99. Springer, October 2008.
- Werner Uwents and Hendrik Blockeel. Learning aggregate functions with neural networks using a cascade-correlation approach. In *Inductive Logic Programming, 18th International Conference, ILP 2008, Proceedings*, volume 5194 of *Lecture Notes in Computer Science*, pages 315–329. Springer, September 2008.
- Werner Uwents, Celine Vens, Anneleen Van Assche, and Hendrik Blockeel. Learning aggregations and selections with relational neural networks. In M. Gori and P. Avesani, editors, *Proceedings of the Workshop on Sub-Symbolic Paradigms for Learning in Structured Domains*, pages 112–121, 2005.

The publications related to Chapters 6 and 7 are:

- Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Marco Gori, and Franco Scarselli. Neural networks for relational learning: An experimental comparison. *Machine Learning*, 82(3):315–349, 2011.
- Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Franco Scarselli, and Marco Gori. Two connectionist models for graph processing: An

experimental comparison on relational data. In T. Gärtner, G.C. Garriga, and T. Meinl, editors, *MLG 2006, Proceedings on the International Workshop on Mining and Learning with Graphs*, pages 211–220, 2006.

- Werner Uwents and Hendrik Blockeel. Classifying relational data with neural networks. In *Inductive Logic Programming, 15th International Conference, ILP 2005, Proceedings*, volume 3625 of *Lecture Notes in Computer Science*, pages 384–396. Springer, 2005.

- Werner Uwents and Hendrik Blockeel. Experiments with relational neural networks. In M. van Otterlo, M. Poel, and A. Nijholt, editors, *Proceedings of the Fourteenth Annual Machine Learning Conference of Belgium and the Netherlands*, pages 105–112, 2005.

- Hendrik Blockeel and Werner Uwents. Using neural networks for relational learning. In T. Dietterich, L. Getoor, and K. Murphy, editors, *ICML 2004 Workshop on Statistical Relational Learning and Its Connections to Other Fields*, pages 23–28, 2004.

Chapter 2

Neural Networks

The following chapters will focus on one specific method within machine learning, i.e., artificial neural networks. In this chapter, the basic concepts of neural networks are explained first. In the second part of the chapter, some general background is given about neural networks for structured and relational domains, which paves the way for the discussions in the following chapters.

2.1 Neural Networks

Artificial neural networks, or simply *neural networks*, are one example of a machine learning method. Because the rest of the dissertation will involve different types of neural networks, the basic notions about neural networks will be explained here. This explanation will also provide a good example of how machine learning methods, discussed in very general terms in the introduction, work in practice.

2.1.1 Feedforward Neural Networks

A neural network [22, 4, 43] is a non-linear model that can be trained to learn complex functions between input and output patterns. The name is due to the analogy with biological neural networks as present in the central nervous system. Although the actual functioning is rather different, there is a similarity between both in the sense that functions are performed collectively and represented distributively by the subparts of the networks.

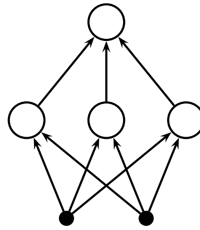

Figure 2.1: Simple feedforward network with two inputs, three hidden units and one output unit.

The basic idea is to represent a complex function by modeling it as a composition of simpler functions. This composition can be modeled as a directed graph or network. Each node of the network represents a unit, either an input unit or a *neuron*. The neurons all represent a simple function. Every edge represents a directed and weighted connection between two units. In principle, connections between units can be made in any way, but usually it is done in a regular fashion by organizing the units in *layers*. Typically, there is a layer with input units, a layer with output units and a layer in between with hidden units.

Example 2.1. *Figure 2.1 depicts an example of a simple feedforward network with two inputs, three hidden units and one output unit.*

The simplest type of networks are *feedforward* networks. In a feedforward network, the information only flows forward, from inputs to outputs. This implies that there are no cycles or loops in the network. It also means that it is possible to define an order in which the values of the units in the network should be computed to generate valid output values.

For each unit, an *activation value* is computed. For the input units, this is simply the corresponding value of the input vector. For the other units, this value is computed based on the activation values of units that have a connection to the considered unit. For a typical two layer network, the activation values x_j of the hidden layer units and the activation values y_j of the output layer

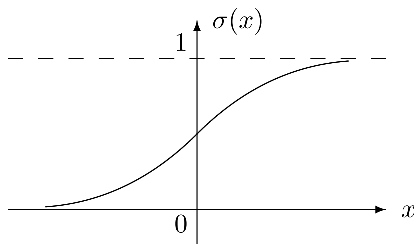

Figure 2.2: Example of a sigmoid function.

units are computed as follows:

$$net_{x_j} = \sum_{i=1}^{N} w_{ij} u_i + b_j \tag{2.1}$$

$$x_j = f(net_{x_j}) \tag{2.2}$$

$$net_{y_j} = \sum_{i=1}^{H} w'_{ij} x_i + b'_j \tag{2.3}$$

$$y_j = f(net_{y_j}) \tag{2.4}$$

with N the number of inputs, H the number of hidden units, u_i the input values, w_{ij} the weights of the hidden layer, w'_{ij} the weights of the output layer, b_j the bias weights of the hidden layer, b'_j the bias weights of the output layer and f an *activation function*. A typical activation function is the sigmoid function σ, shown in Figure 2.2, or a similar function. Networks can have more than one hidden layer, but in most cases, one hidden layer suffices.

2.1.2 Backpropagation

The *backpropagation* algorithm is probably the most popular training method for feedforward neural networks. Training a network is defined as a minimization problem, with the weights of the network connections as the parameters that need to be optimized. The *mean squared error* of the network, computed on all training examples, is the objective function that has to be minimized. It is a function of the weights of the connections in the network. After computing the gradient of this function, the weights can be adjusted in

order to reduce the mean squared error. By repeating this procedure, a series of weight adjustments can be made that will hopefully let the weight values converge to an optimum, minimizing the total output error.

The training data consists of P pairs $\{u^p, t^p\}$, with $u^p = \{u_1^p, \ldots, u_N^p\}$ the input vector and $t^p = \{t_1^p, \ldots, t_M^p\}$ the corresponding target vector. N is the size of the input vectors and M the size of the output vectors. When the output values of the network are computed for input vector u^p, an output vector $y^p = \{y_1^p, \ldots, y_M^p\}$ is obtained. For each pattern p, a squared error E_p between the output vector y^p and the target vector t^p can be computed:

$$E_p = \frac{1}{2}\sum_{k=1}^{M}(t_k^p - y_k^p)^2 \tag{2.5}$$

The total squared error is the sum of all squared errors on all the training patterns, while the mean squared error is the total squared error divided by the number of training patterns, P:

$$E = \frac{1}{P}\sum_{p=1}^{P} E_p \tag{2.6}$$

$$= \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(t_k^p - y_k^p)^2 \tag{2.7}$$

It is clear that E is a function of the weight parameters of the network. To minimize this error E, the gradient $\frac{\partial E}{\partial w_{ij}}$ is computed. After that, *gradient descent* is applied. The gradient vector points in the direction of the steepest ascent, so the negative gradient vector points in the direction of the steepest descent. By taking a small step in the direction of the negative gradient vector, a lower value for E is obtained, moving towards a minimum on the error surface. To make this step, adaptations for all weights should be computed:

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} \quad \text{and} \quad \Delta w'_{ij} = -\eta\frac{\partial E}{\partial w'_{ij}} \tag{2.8}$$

with η the step size of the gradient descent. The partial derivatives $\frac{\partial E}{\partial w_{ij}}$ and $\frac{\partial E}{\partial w'_{ij}}$ can be broken down by using equation 2.6. For $\frac{\partial E}{\partial w_{ij}}$ this results in:

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{P}\sum_{p=1}^{P}\frac{\partial E_p}{\partial w_{ij}} \tag{2.9}$$

Applying the chain rule to $\frac{\partial E_p}{\partial w_{ij}}$ results in:

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial net^p_{x_j}} \frac{\partial net^p_{x_j}}{\partial w_{ij}} \tag{2.10}$$

with $net^p_{x_j}$ the sum of all weighted input values:

$$net^p_{x_j} = \sum_{i=1}^{N} w_{ij} u^p_i + b_j \tag{2.11}$$

This means that:

$$\frac{\partial net^p_{x_j}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{i=1}^{N} w_{ij} u^p_i = u^p_i \tag{2.12}$$

If $\delta^p_{x_j}$ is defined as:

$$\delta^p_{x_j} = -\frac{\partial E_p}{\partial net^p_{x_j}} \tag{2.13}$$

then equation 2.10 can be rewritten as:

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial net^p_{x_j}} \frac{\partial net^p_{x_j}}{\partial w_{ij}} = -\delta^p_{x_j} u^p_i \tag{2.14}$$

To compute $\delta^p_{x_j}$, the chain rule can be applied again:

$$\delta^p_{x_j} = -\frac{\partial E_p}{\partial net^p_{x_j}} = -\frac{\partial E_p}{\partial x^p_j} \frac{\partial x^p_j}{\partial net^p_{x_j}} \tag{2.15}$$

Similar formulas can be derived for the output units, defining the partial derivatives and the $\delta^p_{y_j}$ values in the same way:

$$\frac{\partial E_p}{\partial w'_{ij}} = \frac{\partial E_p}{\partial net^p_{y_j}} \frac{\partial net^p_{y_j}}{\partial w'_{ij}} = -\delta^p_{y_j} x^p_i \tag{2.16}$$

$$\delta^p_{y_j} = -\frac{\partial E_p}{\partial net^p_{y_j}} = -\frac{\partial E_p}{\partial y^p_j} \frac{\partial y^p_j}{\partial net^p_{y_j}} \tag{2.17}$$

The second part of Equations 2.15 and 2.17 is the derivative of activation function f:

$$\frac{\partial x_j^p}{\partial net_{x_j}^p} = f'(net_{x_j}^p) \tag{2.18}$$

$$\frac{\partial y_j^p}{\partial net_{y_j}^p} = f'(net_{y_j}^p) \tag{2.19}$$

For a hidden unit, the effect of a change in the activation value on the final output values, can only be propagated through the M outgoing connections of that unit. Using the chain rule, this results in the following equation:

$$\frac{\partial E_p}{\partial x_j^p} = \sum_{k=1}^{M} \frac{\partial E_p}{\partial net_{y_k}^p} \frac{\partial net_{y_k}^p}{\partial x_j^p} = \sum_{k=1}^{M} \frac{\partial E_p}{\partial net_{y_k}^p} w'_{jk} = -\sum_{k=1}^{M} \delta_{y_k}^p w'_{jk} \tag{2.20}$$

By using Equations 2.20 and 2.18, Equation 2.15 can be rewritten as:

$$\delta_{x_j}^p = f'(net_{x_j}^p) \sum_{k=1}^{M} \delta_{y_k}^p w'_{jk} \tag{2.21}$$

The $\delta_{y_j}^p$ values are even easier to determine because the contribution to the output error is known directly. Equation 2.5 implies that:

$$\frac{\partial E_p}{\partial y_j^p} = -(t_j^p - y_j^p) \tag{2.22}$$

By using Equations 2.22 and 2.19, Equation 2.17 can be rewritten as:

$$\delta_{y_j}^p = (t_j^p - y_j^p) f'(net_{y_j}^p) \tag{2.23}$$

Together, Equations 2.23 and 2.21 provide a way to calculate the δ values for all neurons by proceeding from the outputs to the inputs. In this way, the error signal is backpropagated from the outputs towards the inputs, hence the name of the algorithm. The δ values are then used to compute the weight adaptations from Equation 2.8:

$$\Delta w_{ij} = \eta \frac{1}{P} \sum_{p=1}^{P} \delta_{x_j}^p u_i^p \tag{2.24}$$

$$\Delta w'_{ij} = \eta \frac{1}{P} \sum_{p=1}^{P} \delta_{y_j}^p x_i^p \tag{2.25}$$

The bias weights should be adapted in the same fashion, the reasoning is exactly the same as for the other weights.

2.1.3 Resilient Propagation

The backpropagation algorithm provides an elegant way to adjust the weight parameters of a neural network, but one of its drawbacks is the learning parameter η. The choice of this parameter determines the time needed until the algorithm converges to an optimum because it determines the step size of the gradient descent. If it is set too small, it will take a large number of iterations before an acceptable solution is reached, but if it is set too large, it will be difficult to come close enough to the optimum because the algorithm takes too large steps.

Different solutions have been proposed to determine the learning rate dynamically during training. One of these methods is *resilient propagation* [42, 41]. The idea is to increase the step size as long as steps are taken in the same direction, and to decrease it otherwise. This allows to move rapidly on long monotone parts of the error curve, but also to take smaller steps in the neighborhood of a local optimum to approximate it well.

For each weight w_{ij} a value Δ_{ij} is computed, which determines the size of the weight update. The computation depends on the sign of the partial derivative of the weight w_{ij}. If this sign has changed with regard to the previous iteration, it means that the last update was too big and the algorithm has jumped over a local minimum. Therefore, the update value is reduced with a factor η^-. If the sign is still the same, the value is increased with a factor η^+. So the update value grows when all updates keep on being positive or negative, and it shrinks when the updates alternate between being positive and negative. This results in Algorithm 2.1.

A number of parameters have to be determined still, and the same values will be used as in [42]. The Δ_{ij} values are limited by a minimum and a maximum. These values are set to $1e^{-6}$ and 50 respectively. The initial values Δ_0 of the Δ_{ij} values also have to be determined, 0.1 is a reasonable choice for that. And also the scaling factors η^- and η^+ have to be set. Reasonable values are $\eta^- = 0.5$ and $\eta^+ = 1.2$.

The resilient propagation training algorithm will be used as the basic training algorithm for the networks in the next chapters.

Algorithm 2.1 The resilient propagation training algorithm for adapting the weights w_{ij}, T is the number of training iterations.

$\Delta_{ij}(0) = \Delta_0$
$\frac{\partial E}{\partial w_{ij}}(0) = 0$
for $t = 1$ to T **do**
  for each w_{ij} **do**
    compute $\frac{\partial E}{\partial w_{ij}}(t)$
    if $\frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) > 0$ **then**
      $\Delta_{ij}(t) = \text{minimum}(\Delta_{ij}(t-1) \times \eta^{+}, \Delta_{max})$
      $\Delta w_{ij}(t) = -sign\left(\frac{\partial E}{\partial w_{ij}}(t)\right) \times \Delta_{ij}(t)$
      $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$
    else if $\frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) < 0$ **then**
      $\Delta_{ij}(t) = \text{maximum}(\Delta_{ij}(t-1) \times \eta^{-}, \Delta_{min})$
      $\frac{\partial E}{\partial w_{ij}}(t) = 0$
    else if $\frac{\partial E}{\partial w_{ij}}(t-1) \times \frac{\partial E}{\partial w_{ij}}(t) = 0$ **then**
      $\Delta w_{ij}(t) = -sign\left(\frac{\partial E}{\partial w_{ij}}(t)\right) \times \Delta_{ij}(t)$
      $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$
    end if
  end for
end for

2.2 Neural Networks for Relational Domains

The standard neural network is only suitable for simple input patterns consisting of single vectors. But more advanced types of neural networks have been developed to deal with relational data domains. There are however two new aspects that come into play when using neural networks - or other machine learning methods - in relational domains.

2.2.1 Learning Structural Features

There has been quite some research in the past on using neural networks for relational learning [46, 15, 18, 21]. From the beginning of the development of neural networks, the possibility of using recurrent instead of feedforward

networks has been explored. This kind of networks can process sequences of input vectors instead of single vectors. From there, it was only a small step to expand this to treelike structures and acyclic graphs. More recently, methods have been proposed that can also process cyclic graphs. The details of some of these approaches will be discussed in Chapter 5.

Most of the existing methods focused mainly on learning structural features, i.e., they are designed to learn concepts based on some characteristic structural pattern in the input pattern. This could for instance be the occurrence of a specific combination of attributes, distributed over a number of connected vectors in the input pattern. A good example of this is a learning problem where some property of a molecule has to be predicted, a property that is determined by the existence of a certain substructure, e.g., a chain of three carbon atoms.

All these different neural network models for relational domains are distributed models where the network consists of several connected parts, with each part a small neural network on itself. The structure of the network as a whole typically mirrors the structure of the input pattern quite closely. This is a very natural way of applying neural networks to this kind of domains, but it comes with some disadvantages, especially when the focus lies on learning structural features.

First of all, it means that these methods are local methods in the sense that each part of the network only sees its own neighborhood, i.e., its own inputs and the information coming from directly connected network parts. There is no global view. This introduces some theoretical restrictions on what can be learned, as will be discussed in Chapter 5. So not every concept can be learned. Another disadvantage is that even if the concept can be learned in principle, it might be difficult because of problems related to the learning process. It is known that learning relations between vectors of the input pattern that are far from each other, is difficult with this kind of methods. The fact that information has to be propagated through a lot of different parts of the network to reach the other vector, influences the ability to recognize the relation between the two vectors in a negative way. These two disadvantages have as effect that these methods might actually not be the best choice for learning structural features, especially when features get more complicated or comprehensive.

2.2.2 Learning Aggregate Features

Little attention has been given so far to neural network methods that focus on learning aggregate features. Aggregate features are features that summarize information from different elements of the relational input pattern into one

single feature. This is very similar to the process of aggregating a set of values, resulting in a single value, e.g., the maximum or the average. Learning this kind of features can also play an important role in relational domains [6]. Consider the problem of learning to predict the property of a molecule again. It could be that the prediction does not depend on the existence of some specific substructure, but rather on some aggregate properties of the atoms of the molecule. Such aggregate properties could be the count of all carbon atoms in the molecule or the highest charge value of all atoms in the molecule.

The distinction between structural and aggregate features is a bit vague because there is some overlap between both concepts. In the existing neural network models for relational domains, this problem of learning aggregate features has never been considered very explicitly. However, these methods do also learn aggregate features, albeit rather simple ones in most cases. In fact, every distributed model has to be able to learn aggregate features, because every part of the network needs to process information coming from the connected parts. This processing naturally involves some kind of information reduction to single features that can be used as input. Learning how to perform this reduction is the same as learning aggregate features.

In this sense it seems very logical to dedicate some attention to this problem of learning aggregate features on itself and to focus more on problem settings where this ability of learning aggregate features is more important than the ability to learn structural features. That will also be the main theme throughout the rest of this dissertation.

Chapter 3

Learning Aggregate Features

This chapter provides some background on learning aggregate features, the topic of this chapter and the next one. The idea of learning from bags is explained and the aggregate features that are typically involved in this process are considered in more detail. A number of existing approaches will be discussed at the end of the chapter.

3.1 Introduction

First, learning from bags will be defined, the learning setting that will be the subject of this and the next chapter. Also the relationship with other settings, such as multi-instance learning and relational learning, is discussed.

3.1.1 Learning from Bags

Every machine learning algorithm for supervised learning is supposed to learn some model, i.e., a mapping function between an input and an output domain, that can be used for prediction. There are however big differences between machine learning algorithms with regard to the considered domains and models. In the simplest case, the input domain consists of simple tuples or data vectors[1]

[1] The terms tuple and vector are almost the same in this context. They both denote a fixed-sized list of simple features. When talking about relational data, the term tuple is more common, the term vector is generally used for neural networks. The only difference is the assumption that a vector has only real values while a tuple can also have nominal values for

Loan

	age	**gender**	**amount**	**class**
l_1	31	male	5000	pos
l_2	25	female	7000	pos
l_3	39	male	9000	neg
l_4	28	male	4000	pos
l_5	42	female	8000	neg

Figure 3.1: Propositional data example. The dataset is given in a single table, the loan relation, where each row represents a single data example described by three attributes - age, gender and amount - and a class label.

of the same type and the output domain is either a set of labels or the domain of the real numbers, depending on whether it is a classification or a regression problem. This setting is called *propositional* or *attribute-value learning*. The fact that the input data are all tuples of the same type means that they contain values for the same attributes.

Example 3.1. *A simple example of a propositional dataset is a database describing loans which have to be classified as positive or negative depending on whether they are paid back correctly or not. Each loan corresponds to one person, the borrower, and is described by three attributes, i.e., age and gender of the borrower and the amount borrowed. This means that for each loan, a tuple or vector with values for these three attributes is given together with a class label. From a given set of loans, the right model can then be learned empirically by the algorithm. A concrete example of such a loan dataset is shown in Figure 3.1. This propositional format can easily be represented as a single table where every row stands for a tuple. A model learned from these examples could predict for instance that a loan is positive if the amount is lower than 7500, and negative otherwise.*

It is clear that propositional learning uses a simple representation. However, in many cases it makes sense to look for more expressive representations, which enables dealing with more complicated input domains. A simple but substantial extension, which will be the focus of this chapter, is to make it possible that an input pattern is described by a set or bag[2] of tuples instead of a single one. Each tuple in the bag consists of values for a number of attributes, just as the

instance. However, it is in most cases quite easy to transform these other types of values into real values and thus obtain a vector from a tuple.

[2]Throughout the rest of the text, the term bag will be used instead of set because, strictly speaking, each element in a set can occur only once. In the setting considered here, it is

Loan

	bag	class
l_1	$\{a_1, a_2, a_3\}$	pos
l_2	$\{a_4, a_5\}$	pos
l_3	$\{a_6, a_7\}$	neg
l_4	$\{a_8\}$	pos
l_5	$\{a_9, a_{10}, a_{11}\}$	neg

Account

	type	amount
a_1	checking	400
a_2	savings	250
a_3	savings	350
a_4	savings	600
a_5	checking	300
a_6	savings	150
a_7	checking	100
a_8	savings	500
a_9	savings	100
a_{10}	checking	150
a_{11}	savings	300

Figure 3.2: Bag data example. Each row in the loan table represents a data example, described by references to the corresponding tuples from the account table and a class label.

single tuples in the propositional setting. All tuples in all the bags should be of the same type, which means that they contain values for the same attributes.

Example 3.2. *In the example of the loan classification, loans could be described in a different way. Instead of representing a loan by the three attributes of Example 3.1, a loan could be represented by a bag of tuples. This bag of tuples could for instance represent the different accounts the borrower owns at the bank. Suppose that each borrower can have a number of savings and checking accounts. Each account is described by attributes for the account type and the amount on it. Each loan is described by a bag of accounts and a class. An example dataset is shown in Figure 3.2. The same dataset in Prolog format, which will be useful later on, is shown in Figure 3.3. A valid model learned from this data could for instance predict that a loan is positive if the total amount on all savings accounts in the bag is equal to or greater than 500, otherwise the loan is negative.*

sometimes possible that the same tuple occurs multiple times in the collection. This will for instance be the case with the networks from Chapters 6 and 7.

```
loan(l1, pos).
loan(l2, pos).
loan(l3, neg).
loan(l4, pos).
loan(l5, neg).

account(l1, a1, checking, 400).
account(l1, a2, savings, 250).
account(l1, a3, savings, 350).
account(l2, a4, savings, 600).
account(l2, a5, checking, 300).
account(l3, a6, savings, 150).
account(l3, a7, checking, 100).
account(l4, a8, savings, 500).
account(l5, a9, savings, 100).
account(l5, a10, checking, 150).
account(l5, a11, savings, 300).
```

Figure 3.3: The bag data example from Figure 3.2 in Prolog.

3.1.2 Bag Learning Settings

Learning from bags can be further differentiated into separate learning settings depending on the characteristics of the model that has to be learned. These settings are, from simple to complex, multi-instance learning, multi-tuple learning, multi-join learning and multi-relational learning [7].

Multi-instance learning. The simplest form of learning from bags is multi-instance learning. For this specific machine learning problem, an extensive literature already exists, because it is often not so difficult to extend propositional machine learning methods to cover this class of problems as well. The distinguishing mark of multi-instance problems is that the target value depends on the occurrence of at least one specific type of tuple in the bag. Dependencies or connections between the different tuples in a bag do not play any role. This allows to keep the model simple and close to propositional models.

Example 3.3. *Using the same data as in Example 3.2, the learning problem would be a multi-instance problem if the model that has to be learned, classifies the loan as negative if at least one of the accounts in the associated bag of accounts satisfies a condition C, and positive otherwise. In this case, the*

condition C could be 'the account is a savings account and the amount on it is less than or equal to 150', giving a correct classification of the example in Figure 3.2.

Multi-tuple learning. The multi-tuple learning setting is very similar to multi-instance learning, the only difference is that in multi-tuple learning the target value depends on the occurrence of a combination of two or more specific tuples in the same input bag. The target depends on more than just single tuples, as in multi-instance learning, but the interdependence is still very limited. Only the concurrence of specific tuples as such plays a role.

Example 3.4. *Using the same data as in Example 3.2, the learning problem would be a multi-tuple problem if the model that is learned, classifies the loan as negative if and only if at least one of the accounts in the associated bag of accounts satisfies a condition C and at least one of the accounts in the same bag satisfies a condition D. In this case, the condition C could be 'the account is a savings account and the amount on it is less than 300' and the condition D could be 'the account is a checking account and amount on it is less than 200', giving a correct classification for the example data of Figure 3.2.*

Multi-join learning. In the multi-join learning setting, the target also relies on an interdependence between two or more tuples in the bag but the interdependence goes further than just concurrence. This means that there is some relation between attributes of different tuples involved in the underlying model. In relation databases, this typically involves the use of a join, hence the name.

Example 3.5. *Using the same data as in Example 3.2, the learning problem would be a multi-join problem if the model that is learned, classifies the loan as negative if and only if the loan has an associated account, savings or checking, with an amount of exactly 100 and an another account of the opposite kind with an amount of exactly 150. This results in a correct classification for the example data of Figure 3.2.*

Multi-relational learning. The multi-relational learning setting stands a bit apart from the rest. It is different from the previous ones because it allows different types of tuples in the bag, with different numbers and types of attributes. So the crucial difference in this case is not the characteristics of the model that is learned, but the representation of the input data. It is very close to the relational learning setting that will be discussed in the next section.

Loan

	bag	**class**
l_1	$\{a_1, a_2, a_3, c_1, c_2\}$	pos
l_2	$\{a_4, a_5, c_3\}$	pos
l_3	$\{a_6, a_7\}$	neg
l_4	$\{a_8, c_4, c_5, c_6\}$	pos
l_5	$\{a_9, a_{10}, a_{11}, c_7\}$	neg

Account

	type	**amount**
a_1	checking	400
a_2	savings	250
a_3	savings	350
a_4	savings	600
a_5	checking	300
a_6	savings	150
a_7	checking	100
a_8	savings	500
a_9	savings	100
a_{10}	checking	150
a_{11}	savings	300

Card

	type	**limit**
c_1	credit	50
c_2	debit	100
c_3	debit	100
c_4	credit	80
c_5	debit	50
c_6	credit	75
c_7	credit	100

Figure 3.4: Extension of the dataset from Figure 3.2. An extra table with information about credit and debit cards is added to the dataset. The result is that each example consists of a heterogenous bag with account tuples as well as card tuples.

Example 3.6. *The dataset of Example 3.2 could be extended with an extra table containing information about the credit and debit cards associated with the borrower. In this case, the input bags would not only consist of account tuples but also contain card tuples. See Figure 3.4.*

3.1.3 Relational Learning

Representing input patterns as bags of tuples is a substantial representational extension of the propositional setting, but it still has some limitations. In general, two very important capacities are needed for *relational learning*, i.e., the capacity to deal with different types of tuples and the capacity to deal with relationships between the tuples of an input pattern. Together they provide a way to represent complex structured input patterns. In this sense, multi-relational learning is the last setting that can be considered as a form of learning from bags, although it already deals with different types of tuples. But relational learning takes it one step further because structure is added to the data representation. Adding structural aspects clearly renders the representation of data by bags of tuples inadequate.

Learning from bags can also be seen as a simple case of relational learning, in which only one type of tuples is allowed, and only one type of relationships, namely the relationship that expresses that tuples all belong to the same bag. However, one of the more difficult problems in relational learning, how to handle multiple tuples, has to be addressed already by taking this step. So in this respect, learning from bags is a crucial part of relational learning. Relational learning could also be seen as some form of learning from bags of bags, where tuples that are of the same type and take part in the same type of relationship are grouped together in bags.

Example 3.7. *The loan database example would for instance be a relational dataset if each loan is described by a single loan tuple, a bag of account tuples and a bag of card tuples. There are three different types of tuples now, i.e., loans, accounts and cards. Just as in the propositional case, a loan has attributes for age, gender and amount. Furthermore, a bag of account and card tuples are associated with each loan. Accounts are described by two attributes, type and amount, and cards by type and limit. See Figure 3.5. This could be extended further, for example by adding a table with all transactions related to the accounts. Once multiple levels of relationships are added, the structural aspects of the dataset become more important.*

An overview can be made, from simple to more complicated settings, as shown in Figure 3.6. Propositional learning is the starting point, learning from bags is an extension with multi-instance, multi-tuple and multi-join learning as subdomains, and relational learning is the most complex and complete setting. Between learning from bags and relational learning, there is the special case of multi-relational learning. For the rest of this chapter, only learning from bags will be considered. The multi-relational setting will be ignored because it involves different types of tuples. In the context of this dissertation, this will

Loan

	age	gender	amount	accounts	cards	class
l_1	31	male	5000	$\{a_1, a_2, a_3\}$	$\{c_1, c_2\}$	pos
l_2	25	female	7000	$\{a_4, a_5\}$	$\{c_3\}$	pos
l_3	39	male	9000	$\{a_6, a_7\}$	$\{\}$	neg
l_4	28	male	4000	$\{a_8\}$	$\{c_4, c_5, c_6\}$	pos
l_5	42	female	8000	$\{a_9, a_{10}, a_{11}\}$	$\{c_7\}$	neg

Account

	type	amount
a_1	checking	400
a_2	savings	250
a_3	savings	350
a_4	savings	600
a_5	checking	300
a_6	savings	150
a_7	checking	100
a_8	savings	500
a_9	savings	100
a_{10}	checking	150
a_{11}	savings	300

Card

	type	limit
c_1	credit	50
c_2	debit	100
c_3	debit	100
c_4	credit	80
c_5	debit	50
c_6	credit	75
c_7	credit	100

Figure 3.5: Relational data example. There are three relations in the dataset: loan, account and card. The loan table contains references to the tuples of the account and card tables.

be considered to be a form of relational learning and be separated from the learning from bags setting. The topic of relational learning will come back in later chapters.

3.1.4 Aggregation

The learned mapping between input and output patterns is defined by a model that is constructed from a set of example data. The essential difference between propositional learning and learning from bags is that for the latter setting some form of *aggregation* will have to be involved in the model. In general,

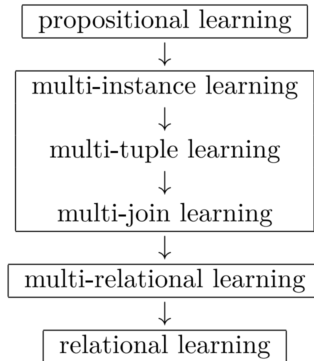

Figure 3.6: Overview of learning settings, from simple to more complex.

aggregation is the process by which a set of items is summarized into a single piece of information. In this case, a bag of tuples will have to be reduced to a single output value. In general, this aggregation will be found in the model in the form of *aggregate features*, as defined below. These are features of the bag as a whole and will be discussed in more detail in the next section.

Definition 3.1 (Aggregate Feature). *If D_I denotes an n-dimensional input domain (e.g. $\mathbb{R}^n$), D_O is a one-dimensional output domain (e.g. $\mathbb{R}$) and $\mathcal{B}(D)$ represents the domain of all finite bags of elements from a domain D, then an aggregate feature f is defined as a function $f : \mathcal{B}(D_I) \mapsto D_O$ that maps a bag of tuples from an input domain D_I to an output domain D_O.*

The result is that the whole model will have two parts and the process of constructing it will involve two steps:

- First of all, a number of aggregate features need to be generated. A very simple example of such a feature is a count of the tuples in the bag. Another example is the maximum value of one specific attribute over all the tuples in a bag. This part is the *feature construction*.
- In a second stage, the aggregate features will be combined to construct the model. How this combination is made, will depend on the considered representation and learning method. This part is the *model construction*.

The details of how this works in practice will become clear in Section 3.3 when some existing approaches are explained. The following example gives a first idea of what such a model could look like.

Example 3.8. *Consider the data from Example 3.2 again. A model covering all the negative examples would be 'the amount on checking accounts is 150 or the amount on savings accounts is 150'. In this case the aggregate features would be a sum of the amounts on checking accounts (feature* f_1*) and a sum of the amounts on savings accounts (feature* f_2*). The combination of the features is then '*$f_1 == 150$ *or* $f_2 == 150$*', which is a disjunction of two equality checks on single features.*

Example 3.9. *The same example can also be expressed in Prolog, using the representation from Figure 3.3. In this case, the aggregate features are represented as:*

```
sum(M, account(L, A, checking, M), X)
sum(M, account(L, A, savings, M), X)
```

These features are then embedded in clauses:

```
loan(L, neg) :- sum(M, account(L, A, checking, M), X), X=150.
loan(L, neg) :- sum(M, account(L, A, savings, M), X), X=150.
```

3.1.5 Propositionalization and Direct Approaches

The question remains how a model containing aggregate features is constructed. Learning the right model is essentially all about constructing the right aggregate features. Although there is a lot of difference between learning methods with respect to how this problem is tackled, it is possible to divide methods into two separate classes with regard to the construction of the aggregate features:

- *Propositionalization approaches:* in these methods, the features are constructed before actual learning takes place [27, 28]. It can be regarded as a data transformation that has been predefined by the user. This means that the aggregate feature construction is not part of the learning process. The RELAGGS system will be discussed in Section 3.3.1 as an example. In a nutshell, RELAGGS uses the average, maximum, minimum, etcetera, of each attribute of the tuples in the input bag as features, instead of using the input bag directly. The advantage of this kind of approach is that it is simple and it makes it possible to rely on existing propositional learning techniques. Methods for feature selection, meta learning, etcetera, can also be used. The distinction between feature and model construction is very clear in this case.

- *Direct approaches:* in this case, the construction of the aggregate features is done during the learning process. This yields a more flexible but also a more complicated method. The fact that the learning algorithm has to construct both features and model, often at the same time, makes this approach very challenging. Inductive logic programming (ILP) falls into this category [25]. Examples of existing systems will be discussed in Section 3.3.2 and 3.3.3. The distinction between feature and model construction can become much vaguer in this case.

3.2 Aggregate Features

In this section, a closer look will be taken at the aggregate features that need to be constructed to learn from bags of data. A general form of these features is given and the aggregate functions that are involved are discussed in more detail.

3.2.1 Aggregation and Selection

To be able to discuss the aggregate features in more detail, a general aggregate feature form can be proposed consisting of two different parts. As has been pointed out in [6], the features typically have an aggregation and a selection part. The aggregation part provides a way to reduce a number of elements to a single value while the selection part operates on individual elements. The distinction between aggregation and selection will be useful when discussing the different methods.

Definition 3.2 (General Aggregate Feature Form). *The general form of an aggregate feature is* $\mathcal{F}(\sigma_C(B))$ *with* B *a bag of tuples,* C *a condition defined over single objects,* σ *the selection operator from relational algebra, and* $\mathcal{F}$ *an aggregate function, which maps a set of objects to a scalar.*

Example 3.10. *A simple example will help to clarify this. Consider the dataset of Figure 3.2, where each input bag represented a number of accounts. Each account is represented by a tuple with an attribute indicating the type and another attribute for the amount. Many different aggregate features could be constructed then. The simplest example is a feature that just counts the number of tuples in the bag, which corresponds to the number of accounts a person has. In this case there is no selection condition needed, only an aggregate function* $\mathcal{F}$ *that counts the number of objects in the bag. Another feature could count the number of savings accounts in the bag. This involves the count aggregate*

function again, but also a selection condition that filters the tuples for which the type attribute equals 'savings' out of the original bag.

Example 3.11. *More complicated features can be constructed by considering combinations of two attributes. An example is the feature that counts the number of savings accounts with more than 250 on it:*

$$count(\sigma_{type='savings' \wedge amount \geq 250}(Account))$$

The selection condition selects the tuples for which the type attribute equals 'savings' and the amount attribute is greater than 250, the aggregate function is still the count function.

With regard to this aggregation and selection part of the features, another distinction between learning systems can be made:

- On the one hand there are methods that focus on the construction of the selection condition and keep the aggregate function very simple. These are *selection-oriented methods*. It has been pointed out [6] that for instance ILP systems typically construct structurally complex conditions but often use the same, trivial, aggregate function, namely $\exists$ (the exists operator, checking for non-emptiness of the set). The reason for this is computational complexity, which will be explained further in Section 3.3.2.
- The importance of using more complex aggregate functions has however been recognized by many people [24, 29]. This is the other side of the spectrum, *aggregation-oriented methods*, where the focus is on the aggregate function while the selection condition is kept simple. One way of achieving this, is by restricting the selection condition to a single attribute. This is done for instance in the RELAGGS system, explained in Section 3.3.1. Most existing systems that do handle such complex aggregate functions, follow the propositionalization approach.

3.2.2 Aggregate Functions

One of the interesting aspects of having aggregate features of the form $\mathcal{F}(\sigma_C(B))$, is that the selection condition corresponds to a part of a propositional model. Features that are learned in propositional learning typically have the same form as these selection conditions. The aggregate function on the other hand is new, something that is not found in propositional

methods. So the most challenging part of the feature construction are the aggregate functions.

Two well-known examples of aggregate functions are maximum and average. They can be defined on a bag of real numbers as the maximum and the average of these numbers. Typically, it is this kind of aggregate functions that are thought of first. But in general, aggregate functions can be very diverse and one could think of much more complicated and exotic functions. Consider for instance a function that works as a statistical distribution detector, giving 1 as value when a set of numbers comes from a gaussian distribution, and 0 otherwise. The only constraint that these functions always have to satisfy, is that they are invariant to permutations, which means that there is no order between the elements of the bag on which they operate. The considered aggregate functions are defined as follows:

Definition 3.3 (Aggregate Function). *An aggregate function $\mathcal{F}$ is defined as a function $\mathcal{F} : \mathcal{B}(D_I) \mapsto D_O$ that maps a bag of elements from a one-dimensional input domain D_I to a one-dimensional output domain D_O.*

These aggregate functions add two complications to the aggregate features. The first problem is that the class of aggregate functions is very broad. This makes it hard to find a general, parameterized expression that, with the right parameters, is able to represent any aggregate function. Unfortunately, this is exactly what is highly desirable in a machine learning algorithm because it would allow to simply adjust the parameters to create the right aggregate function. In some cases the aggregate function can be reduced to a simpler form, like for the additive aggregate functions defined below, but this is not always possible. An additive aggregate function defines an aggregate function as the sum of the values of a simpler function that is applied to the individual elements in the bag separately.

Definition 3.4 (Additive Aggregate Function). *An additive aggregate function is an aggregate function $\mathcal{F}(B) : \mathcal{B}(D_I) \mapsto D_O$ that can be defined as $\sum_{x \in B} \phi(x)$, where $\phi(x) : D_I \mapsto D_O$.*

Another problem is that it is not always clear what the aggregation part is and what the selection part. The reason for this is that an aggregate feature is not necessarily defined in a unique way, as will be demonstrated in the next example. How it can be defined, depends for instance on the aggregate functions that are available for defining it.

Example 3.12. *Consider the following aggregate feature in Prolog:*

```
count(AccId, account(LoanId, AccId, savings, 150), X)
```

which counts the number of savings accounts with an amount of 150 a borrower identified by LoanId has. This feature could also be written using the sum function:

```
sum(1, account(LoanId, AccId, savings, 150), X)
```

This demonstrates that the same feature can be defined in different ways.

3.3 Learning Approaches

It should be clear from the explanation above that the problem of learning aggregate features is far from easy. In this section, some existing approaches to tackle the problem are discussed.

3.3.1 Propositionalization

A simple approach to dealing with bags of tuples is to convert them first to a propositional representation. After that, the problem has been reduced to a propositional learning problem and so a propositional learner can be applied. Of course, the problem is how to define the right transformation. The choice of the transformation will determine whether the propositional learner is able to learn the desired model or not.

A good example of this kind of approach is the RELAGGS system [29, 30]. It is meant to be a fully relational system, able to deal with multiple relations, but what is relevant here is only how it deals with bags of tuples coming from a single relation. With the RELAGGS approach, the tuples in the bag would be summarized according to the types of the tuple attributes. If an attribute is a numeric value, the average, maximum, minimum and sum of this attribute are computed for each bag and added to the propositional representation. If the attribute is a nominal value, the occurrences of the different possible values are counted for each bag and added to the propositionalization. Finally, also the number of tuples in the bag is added to the new representation.

Example 3.13. *If the input bags consist of tuples with two attributes, X_1 and X_2, with X_1 a numeric value and X_2 a nominal value with possible values a and b, then the propositional representation contains $avg(X_1)$, $max(X_1)$, $min(X_1)$, $sum(X_1)$, $count(X_2 = a)$, $count(X_2 = b)$ and the cardinality of the bag. On this propositional representation, propositional learners like decision trees or support vector machines can be used.*

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	**class**
l_1	3	2	1	333.3	400	250	1000	pos
l_2	2	1	1	450	600	300	900	pos
l_3	2	1	1	125	150	100	250	neg
l_4	1	1	0	500	500	500	500	pos
l_5	3	2	1	183.3	300	100	550	neg

$$
\begin{aligned}
f_1 &= count(Account) \\
f_2 &= count(\sigma_{type='savings'}(Account)) \\
f_3 &= count(\sigma_{type='checking'}(Account)) \\
f_4 &= average(\sigma_{amount}(Account)) \\
f_5 &= maximum(\sigma_{amount}(Account)) \\
f_6 &= minimum(\sigma_{amount}(Account)) \\
f_7 &= sum(\sigma_{amount}(Account))
\end{aligned}
$$

Figure 3.7: RELAGGS transformation for the loan example from Figure 3.2.

It is clear that this representation can be very limiting in some cases. For instance, if a bag should be classified as positive if the number of tuples where $X_1 > 0$ and $X_2 = a$ is greater than 2, and negative otherwise, then the right model would be impossible to learn for the propositional learner. The reason is that the propositionalization discarded all information about any relationship between the values for X_1 and X_2. But even if only a single attribute is considered, there can be a problem. If a bag is classified as positive if the number of tuples where $X_1 > 0$ is greater than 2, and negative otherwise, then the same problem arises. The concept can not be learned correctly, although the learner can possibly learn an approximation from the average, maximum, minimum and sum values and still achieve a somewhat acceptable predictive performance in this way.

Example 3.14. *For the loan dataset from Example 3.2, RELAGGS would propositionalize the data with features counting the number of savings and checking accounts, and features for the average, maximum, minimum and sum of the amounts on the accounts. So the dataset of Figure 3.2 would be transformed into the one in Figure 3.7, which is a propositional dataset.*

In an improved version of the RELAGGS system, new aggregate functions were included for numeric values, namely standard deviations, medians, quartiles and ranges. Also combinations of attributes were considered, with one of them

nominal. This leads to a propositionalization that preserves more information and it helps to learn the problematic concepts mentioned before. However, for other concepts it will not result in improved performance at all. The question remains what aggregate functions to include and how far to go with combinations of attributes, as it enlarges the propositionalized input space very rapidly.

3.3.2 Inductive Logic Programming

Inductive logic programming (ILP) can also be used to learn from bags [7]. In fact, ILP can do more than just learning from bags, as it supports fully relational learning, but here the discussion will be limited to how it handles learning from bags.

Contrary to the RELAGGS method, ILP is a direct approach, so the features are constructed during learning, as they are needed. In the case of ILP, features are expressed as clauses of a logic program. The generated clauses are typically of the form $h \leftarrow l_1, \ldots, l_m$, where the head atom or predicate h is true when all literals l_i of the body are true. The data is represented as a collection of facts, i.e., clauses without literals in the body. Now the aim in ILP is to find a set of clauses that entail all positive examples and none of the negative ones. These clauses define the predictive model.

In fact, standard ILP techniques can be used to work with bags of tuples, without further modifications. A standard clause in ILP represents a selection condition and this condition can be tested on multiple tuples as well as on one tuple. If there is at least one tuple for which the condition holds, then the clause will become true, otherwise it is false. With regard to the aggregate feature form $\mathcal{F}(\sigma_C(B))$, this means that the aggregate function $\mathcal{F}$ is implicit and it is always the $\exists$ operator.

Example 3.15. *Consider the following standard clause in ILP:*

```
loan(L, pos) :- account(L, A, savings, M).
```

It says that a loan is classified as positive if there exists a savings account for the person to which the loan belongs. It does not matter whether there is only one account for each person, or multiple accounts.

It is clear that constructing more advanced aggregate features, with other aggregate functions than the $\exists$ operator, would be useful. This is possible for instance by generating clauses that include a literal $F(X, Q, R)$, where F

is an aggregate function, Q a query, X a variable from the query Q and R the result of applying F to all substitutions of X in the result of the query Q. The function F from this aggregate literal corresponds to the $\mathcal{F}$ function from the aggregate feature form, while the query Q corresponds to the selection condition C. The query can contain complicated conditions, depending on what refinements are allowed.

Example 3.16. *Consider the following aggregate clause:*

```
loan(L, pos) :- sum(M, account(L, A, savings, M), X), X >= 500.
```

This clause covers all the positive examples from the example dataset of Figure 3.3, so together with a clause that says that all other examples are negative, it provides a correct model for the example dataset. In this case, the aggregate function F is the sum function and the query Q selects all saving accounts that are related to a loan L.

However, the introduction of aggregate predicates gives rise to problems with complexity and monotonicity [50]. First of all, introducing aggregate functions increases the computational complexity of the learning task. Typically, ILP systems explore the search space in a greedy manner. They start from the current best clause, generate a number of refinements and select the best one to start all over again. The complexity of this procedure depends on the number of possible refinements at each step. The problem with constructing features of the form $\mathcal{F}(\sigma_C(B))$ is that the number of possible refinements grows very rapidly. Different aggregate functions can be used, each of the aggregate functions can be applied to any of the variables, and within these aggregate functions complicated selection conditions can be constructed. Together, these factors give rise to a very large number of possible combinations and blow up the search space.

This computational complexity is determined by the *branching factor*, the number of possible refinements generated from a clause. If C is the number of standard literals that can be added to a standard clause in ILP, then C is also the branching factor of the search. However, when aggregate literals $F(X, Q, R)$ are allowed in the clause, the branching factor is much larger. If the query Q can only be a simple literal, the branching factor for this part is C again. But all of the variables in the added literal can be used as aggregate variable, so if there are V variables, the branching factor becomes $V \times C$. Furthermore, different aggregate literals can be used for the aggregate function F. If there are N different aggregate literals, the branching factor grows to $N \times V \times C$. The total branching factor will be $C + N \times V \times C$. It gets even worse if the query Q can be refined to more complicated expressions. Then the total branching

factor will be as large as $C + N \times V \times C + N \times V \times C \times (C + N \times V \times C)^L$, with L the number of added language elements in the query.

Another problem is that not all possible refinements are valid. Many ILP systems rely on the property that refinements always happen from more general to more specific, or vice versa, but this is not automatically guaranteed when aggregate functions are introduced. To ensure valid refinements, the aggregate functions should be monotone (or anti-monotone) for all possible refinements. As long as this condition is met, valid refinements can be defined and the normal refinement procedure in ILP can be used.

Example 3.17. *Suppose that the start clause is:*

```
loan(L, pos) :- sum(M, account(L, A, T, M), X), X >= 100.
```

This can be refined to:

```
loan(L, pos) :- sum(M, account(L, A, savings, M), X), X >= 100.
```

This refinement is valid because it will always have at most the same coverage as the original clause, at least if $M \geq 0$. In this case the refined clause is a specialization. But if the refined clause is:

```
loan(L, pos) :- sum(M, account(L, A, T, M), X), X >= 50.
```

then the coverage is at least the same as for the original clause, which means that the refined clause is a generalization.

Looking at the clauses with aggregate literals, three parts can be refined. The aggregation function F can be changed, the query Q, and the check on the result of the aggregation function. Refinement can happen along these three dimensions, so it must be guaranteed that in all three dimensions refinement happens in a monotone way. To facilitate this, the available aggregate functions should be ordered, as well as the query refinements and the tresholds of the result check. For the query and threshold, this is rather straightforward. For the aggregate functions, this is also possible by considering parameterized classes of aggregate functions in which the desired functions are included. If these conditions are ensured, a monotone path along these three dimensions can be followed during the refinement procedure.

So if aggregation is used in ILP, complexity should be kept within reasonable bounds and monotonicity properties should be preserved. But if these

restrictions are met, it is possible to use ILP methods with aggregate functions to learn from bags. It is clear that this method is much more powerful and flexible than the propositionalization method, especially with regard to the selection conditions. The problem of choosing the right aggregate functions persists however.

3.3.3 Metric-Based Methods

Metric-based methods, such as the k-nearest neighbors algorithm (k-NN) or support vector machines (SVMs), can also be used to learn from bags, although in this case the link with the general aggregate features defined in Section 3.2.1 is lost. In general, the problem of learning aggregate features is reformulated in terms of a similarity or distance measure between bags. The most popular metric-based approach is probably support vector machines.

The basic idea of a support vector machine is to construct a separating hyperplane between the positive and negative vectors in a dataset, maximizing the margin between the vectors of the two classes. In order to handle non-linear classification problems as well, a transformation function is introduced to perform a non-linear mapping of the vectors into some feature space. In fact, this function itself is not needed but only the dot products between the vectors in feature space. These dot products are computed by a *kernel function* $k(x, y)$, which can be interpreted as a similarity or distance measure between the input patterns x and y.

Instead of defining the kernel function between single vectors, a kernel function can also be constructed between bags of vectors. This bag kernel function will typically be defined in terms of a simple kernel function between the single vectors from the bags. An example of such a bag kernel function is the average linkage kernel, which is defined as:

$$K_{AL}(A, B) = \frac{\sum_{i=1}^{|A|} \sum_{j=1}^{|B|} k(a_i, b_j)}{|A||B|} \tag{3.1}$$

with $A = \{a_i\}$ and $B = \{b_j\}$ bags of vectors and $|A|$ and $|B|$ the sizes of bags A and B respectively. This function can be interpreted as the average distance between all vectors of the two bags. Many other kernel functions could be defined as well. For instance, if the maximum possible distance between vectors

of the two bags is taken, the Hausdorff distance is obtained:

$$K_H(A, B) = \max[\max_{a_i}(\min_{b_j}(k(a_i, b_j))), \max_{b_i}(\min_{a_j}(k(b_i, a_j)))] \quad (3.2)$$

Two problems arise when using kernels to learn from bags. As already mentioned, the link with the aggregate features described in Section 3.2.1 is lost in these kernel functions. It is not immediately clear how each similarity measure corresponds to an aggregate feature. Essentially, if two bags are very similar, it is implied that there exists an aggregate feature $\mathcal{F}(\sigma_c(B))$ for which the values are very close to each other for these two bags, and vice versa:

$$B_1 \sim B_2 \Leftrightarrow \mathcal{F}(\sigma_C(B_1)) \approx \mathcal{F}(\sigma_C(B_2)) \quad (3.3)$$

The aggregate feature is implicitly defined by the similarity measure, but it is not constructed explicitly. On the other hand, the bag kernel function can also be broken up into two parts, as can be seen in Equations 3.1 and 3.2. There is the simple kernel function between single vectors from the bags and then there is the function which combines the results of this simple kernel applied to all pairs of vectors. The simple kernel looks a bit similar to the selection part σ_C and the combining function bears some resemblance to the aggregate function $\mathcal{F}$ in the aggregate feature, but there is no direct correspondence. One similarity is that the simple kernel could also be used to perform some kind of feature selection, just as the selection operator σ_C.

The second problem is very similar to the problem with propositionalization or ILP. The right kernel has to be selected, just like the right propositionalization or aggregate predicates had to be selected, to achieve a good performance. This requires some background knowledge about the problem. In fact, this is related to the two parts of which the bag kernel consists, as explained above. The right simple kernel together with the right combination function should be found. If they are selected by the user before learning is performed, which is a typical practice with SVMs, then they might be unsuited for the considered problem, a problem which could also occur with propositionalization. If they are selected automatically during training, a search will have to be conducted to find the right combination. This gives rise to the same kind of problems as with ILP methods because the search space can be quite large, especially because the simple kernels typically also have parameters that need to be optimized.

Example 3.18. *Consider the bag data example from Figure 3.2 again. A measure based on the average linkage kernel will be used as an example here.*

It is not sure that this measure would be very useful for this example, it only serves as an illustration. First, some encoding has to be provided for nominal attributes like the account type. Savings accounts can be encoded as -100 and checking accounts as 100 for instance. Ideally, some scaling should be done to bring all attributes of the vectors in the same interval, but this will be left out of consideration here. As vector kernel, the simple dot product is taken. In this case, the average linkage kernel between the first and second loan gives $(230000+130000+160000+65000+220000+95000)/6 = 150000$ *as result. The same kernel between the first and the third loan gives* $(50000+50000+47500+15000+62500+25000)/6 = 41667$ *as result. This implies that the first and second loan are more similar than the first and the third, at least according to this measure.*

3.4 Conclusions

It is clear from the discussion above that learning aggregation is an interesting and challenging topic. In the rest of the dissertation, novel methods will be proposed, based on neural networks, that deal with this aspect of machine learning. It is also clear that from a machine learning perspective, a direct approach is preferable to a propositionalization approach, because it relies much less on prior knowledge about the problem at hand. The existing direct approaches typically rely on a discrete search. Neural network methods are different because they are based on a gradient optimization. In this respect, they provide a new perspective on the aggregation learning problem. In the next chapters, the focus will be first on learning from simple bags of data. This allows to consider aggregation in its most elementary form. After that, an attempt will be made to broaden this to more complex relational domains.

Chapter 4

Neural Networks for Learning Aggregate Features

In this chapter, approaches to learning from bags based on neural networks are discussed. There are different ways to adapt neural networks in order to let them process bags of vectors instead of single vectors. They will be explained in the following sections. First, a number of novel approaches are developed that are based on standard neural networks. After that, a hybrid approach based on cascade-correlation will be proposed. In the last part of the chapter, experiments with these different types of networks will be discussed and the results will be compared with results for other approaches.

4.1 Simple Aggregate Network Structures

In the following subsections, extensions of feedforward networks are discussed. Most of these extensions are rather simple and straightforward. To ease the discussion further on, some terminology is explained here first.

Consider a standard feedforward network with one hidden layer. This network consists of a number of N input units, H hidden units with activation values x_i and M output units with activation values y_i. The connections in a layered network are between subsequent layers, so there are connections from each input unit to all hidden units, and from each hidden unit to all output units. The weight associated with a connection from the input unit u_i to the hidden

unit x_j is denoted by w_{ij}, the bias weight is denoted by b_j. For the output units, w'_{ij} and b'_j are used.

Computation of the activation values of hidden and output units is done in two steps. First, the net value $net_{x_i} = \sum w_{ji}u_j + b_i$ is computed. After that, an activation function σ is applied to the net value to compute the actual activation value $x_i = \sigma(net_{x_i})$. The σ-function is the sigmoid function, although other functions could be used as well. So the computation of all activation values for a data pattern p happens in the following way:

- For the hidden units:

$$net^p_{x_i} = \sum_{j=1}^{N} w_{ji}u^p_j + b_i \quad (4.1)$$

$$x^p_i = \sigma(net^p_{x_i}) \quad (4.2)$$

- For the output units:

$$net^p_{y_i} = \sum_{j=1}^{H} w'_{ji}x^p_j + b'_i \quad (4.3)$$

$$y^p_i = \sigma(net^p_{y_i}) \quad (4.4)$$

Explanations of the network structures in the following subsections will always start from this standard feedforward network with one hidden layer. Because these network structures will deal with multiple input vectors, an extra index is needed. An input value $u_i(j)$ now denotes the i-th component of the j-th vector. The indexing of the vectors is arbitrary because there is no real order among them. When different activation values are computed for the different input vectors, these will also be denoted as $x_i(j)$ or $y_i(j)$. The number of input vectors for a data pattern p is $\#^p$.

4.1.1 Padded Feedforward Networks

Probably the simplest and most straightforward way to extend normal feedforward neural networks to bags, is to use a feedforward network F with $\#_{max} \times N$ inputs. In this formula, $\#_{max}$ is the maximum number of vectors a bag can contain, while N is the number of values per vector. When a bag of size $\#^p$ is fed into the network, the first $\#^p \times N$ inputs are filled with the

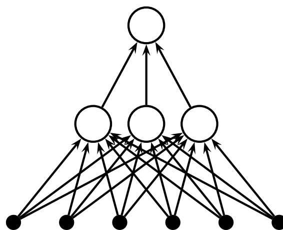

Figure 4.1: Example of a padded network. The network has one output unit, three hidden units and six input units. Each vector contains two values. In this case, the maximum number of input vectors ($\#_{max}$) is three.

values of the $\#^p$ vectors in the bag. The order in which this happens, is not important. The remaining $(\#_{max} - \#^p) \times N$ inputs are filled up with padding values. For these padding values, a fixed, given value is used.

Another way to look at it, is that every bag is extended with padding vectors u_{pad} until it has cardinality $\#_{max}$. After that, all bags can be converted into a long vector with size $\#_{max} \times N$, by concatenating all vectors from the bag. This vector can then be used as input to the network F. This procedure can also be carried out with other learning algorithms than neural networks. The padding vector u_{pad} can be a vector filled with zero values, but, depending on the problem, other choices might be better. Most importantly, there should be no confusion between a padding vector and a regular input vector so that the learning algorithm can make the distinction.

Example 4.1. *Suppose that all tuples in the bag are represented by two-dimensional vectors, the maximum number of tuples encountered in a bag is 3 and the default value for unused inputs is 0. In this case, a network should be used with 6 inputs. For a bag with tuples* $[0.5, 0.3]$*,* $[0.1, -0.2]$ *and* $[0.7, 0.4]$*, an input vector* $[0.5, 0.3, 0.1, -0.2, 0.7, 0.4]$ *can be constructed to feed into the network. If there are only two tuples, for instance* $[0.2, 0.8]$ *and* $[-0.3, 0.1]$*, then an input vector* $[0.2, 0.8, -0.3, 0.1, 0, 0]$ *can be constructed. Other orders, like* $[-0.3, 0.1, 0, 0, 0.2, 0.8]$*, are also valid. In Figure 4.1, an example of such a padded network is shown for input tuples with two variables and a maximum of three tuples in the input bag.*

The main advantage of this approach is that it is simple. One of the disadvantages is that $\#_{max}$ can be quite large. This results in a network with a lot of connections and corresponding weights to train, which is inconvenient

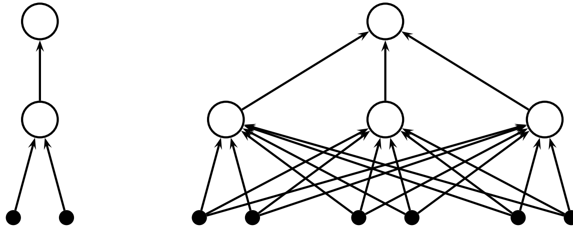

Figure 4.2: Example of a symmetric network. The original feedforward network is shown at the left, the unfolded network at the right. The network has two inputs, one hidden and one output unit and has been unfolded to handle three input vectors.

because it increases the complexity of the model and the danger of overfitting. If there is a large variation in the sizes of the bags, a lot of these network connections will also be irrelevant most of the time because they will only be used for padding values. It is also not guaranteed that the F network is really not sensitive to the order of the vectors. To relieve these problems somewhat, one could change the order of the bag and padding vectors constantly during training.

Another problem is that the number of hidden neurons for F has been fixed when determining the network architecture, just as for any other feedforward neural network. This is however not ideal. It seems more logical that the number of hidden neurons would be somehow proportional to $\#^p$, the cardinality of the input bag, because the complexity of the function is probably more or less proportional to the size of the bag.

In the experimental results, this type of network structure is denoted as FF because it is basically a standard feedforward network.

4.1.2 Symmetric Feedforward Networks

To avoid the mentioned problems with padded feedforward networks, a similar but more sophisticated approach can be taken. Consider a feedforward network structure having N inputs, with N the number of attributes in each vector. However, the network does not work like a normal feedforward network. It uses some kind of *folding* procedure, which means that the network, or at least some parts of it, are copied as many times as there are input vectors. So the network is dynamically adapted to the input pattern. This folding method is

used in neural networks for structured domains, and will be explained in detail in Section 5.1.1 for recurrent networks.

Here it means that the inputs and hidden units are copied $\#^p$ times, one set for each input tuple. The connections are also copied. Each input is connected with all $\#^p \times H$ hidden units. In principle, weights are shared between copies of the same connection. When a change is made to a weight, it should also be propagated to the weights of the other copies of that connection. In this case, two different weights per connection will be used for the hidden units. The weight for the connection between $u_i(k)$ and $x_j(k)$ is w_{ij}, for all other connections between $u_i(k)$ and $x_j(l)$, the weight is w^*_{ij}. The reason for this is that some kind of one-against-all evaluation is done. Each copy of the hidden neurons computes a contribution based on the corresponding input vector versus all other input vectors. The other input vectors are indistinguishable from each other because there is no order among them, so the same weights are used there.

Mathematically, the activation values are now calculated as follows:

- For the hidden units:

$$\begin{aligned} net^p_{x_i}(j) &= \sum_{k=1}^{N} w_{ki} u^p_k(j) + \sum_{k=1}^{j-1} \sum_{l=1}^{N} w^*_{li} u^p_l(k) \\ &\quad + \sum_{k=j+1}^{\#^p} \sum_{l=1}^{N} w^*_{li} u^p_l(k) + b_i \end{aligned} \tag{4.5}$$

$$x^p_i(j) = \sigma(net^p_{x_i}(j)) \tag{4.6}$$

- For the ouptut units:

$$net^p_{y_i} = \sum_{j=1}^{H} \sum_{k=1}^{\#^p} w'_{ji} x^p_j(k) + b'_i \tag{4.7}$$

$$y^p_i = \sigma(net^p_{y_i}) \tag{4.8}$$

Example 4.2. *An example of how this network and its folding works, is shown in Figure 4.2. The vectors all have two attributes. One hidden and one output unit is used. The folding is shown for an input bag with three vectors.*

Making the input and hidden layer variable, depending on the size of the input bag, and introducing a lot of weight sharing, makes this type of network more

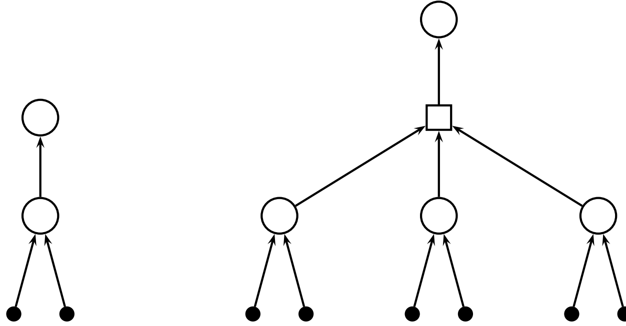

Figure 4.3: Example of an aggregated network with an aggregate unit after the hidden unit. The original network is shown at the left, the unfolded network at the right. The network has two inputs, one hidden and one output unit. The square unit represents the aggregate function used to combine the activation values from the different folds. The weights of the connections to this unit are just 1.

suitable for learning aggregate functions than the first, naive one. It reduces the number of parameters to be learned and it ensures that this type of network is symmetric. In the experimental results, this type of network is denoted by SYM.

4.1.3 Aggregated Feedforward Networks

Another possibility is to introduce special aggregate units in the network. Outputs for these units are computed for all vectors in the input bag and the output values are combined with an aggregate function g that maps a collection of real values to a single real value. The function g should be differentiable to allow computation of the gradient for the training of the network. For functions that are not differentiable, it is possible to consider approximations that are differentiable. If the aggregate units are used in the hidden layer, the computation of the activation values goes as follows:

- For the hidden units:

$$net^p_{x_i}(j) = \sum_{k=1}^{N} w_{ki} u^p_k(j) + b_i \tag{4.9}$$

$$x^p_i = g(\sigma(net^p_{x_i}(j))) \tag{4.10}$$

- For the output units:

$$net^p_{y_i} = \sum_{j=1}^{H} w'_{ji} x^p_j + b'_i \quad (4.11)$$

$$y^p_i = \sigma(net^p_{y_i}) \quad (4.12)$$

Two possible choices for g will be considered here, the sum and the maximum function. If a sum is used for the function g, and the aggregate units are used at the level of the hidden neurons, then the network is evaluated as follows:

- For the hidden units:

$$net^p_{x_i}(j) = \sum_{k=1}^{N} w_{ki} u^p_k(j) + b_i \quad (4.13)$$

$$x^p_i = \sum_{j=1}^{\#^p} \sigma(net^p_{x_i}(j)) \quad (4.14)$$

- For the output units:

$$net^p_{y_i} = \sum_{j=1}^{H} w'_{ji} x^p_j + b'_i \quad (4.15)$$

$$y^p_i = \sigma(net^p_{y_i}) \quad (4.16)$$

When the max function is used instead of the sum, the computation for the hidden units changes as follows:

$$net^p_{x_i}(j) = \sum_{k=1}^{N} w_{ki} u^p_k(j) + b_i \quad (4.17)$$

$$x^p_i = max\{\sigma(net^p_{x_i}(j))\} \quad (4.18)$$

Example 4.3. *Figure 4.3 shows how this works for an input bag with three vectors.*

If a sum is used for g but the aggregate units are placed at the output layer instead of the hidden layer, the network becomes:

- For the hidden units:

$$net^p_{x_i}(j) = \sum_{k=1}^{N} w_{ki} u^p_k(j) + b_i \tag{4.19}$$

$$x^p_i(j) = \sigma(net^p_{x_i}(j)) \tag{4.20}$$

- For the output units:

$$net^p_{y_i}(j) = \sum_{k=1}^{H} w'_{ki} x^p_k(j) + b'_i \tag{4.21}$$

$$y^p_i = \sum_{j=1}^{\#^p} \sigma(net^p_{y_i}(j)) \tag{4.22}$$

When the max function is used instead of the sum, the computation for the output units changes as follows:

$$net^p_{y_i}(j) = \sum_{k=1}^{H} w'_{ki} x^p_k(j) + b'_i \tag{4.23}$$

$$y^p_i = max\{\sigma(net^p_{y_i}(j))\} \tag{4.24}$$

Example 4.4. *Figure 4.4 shows how this works for an input bag with three vectors. The approach is visualized using the same unfolding procedure as for the symmetric networks.*

When a maximum function is used for the g function, there is a complication because the derivative has to be calculated to perform gradient based learning, but it can not be calculated directly for the max function. However, the max function could easily be approximated by a softmax function, which is differentiable. This is the approach taken in [40], where the same kind of network is described for use in a multi-instance learner. The softmax function and its derivative are:

$$smax_C\{z_1, \ldots, z_n\} = \frac{1}{C}\, ln\left(\sum_{i=1}^{n} e^{Cz_i}\right) \tag{4.25}$$

$$\frac{\partial}{\partial z_j} smax_C\{z_1, \ldots, z_n\} = \frac{e^{Cz_j}}{\sum_{i=1}^{n} e^{Cz_i}} \tag{4.26}$$

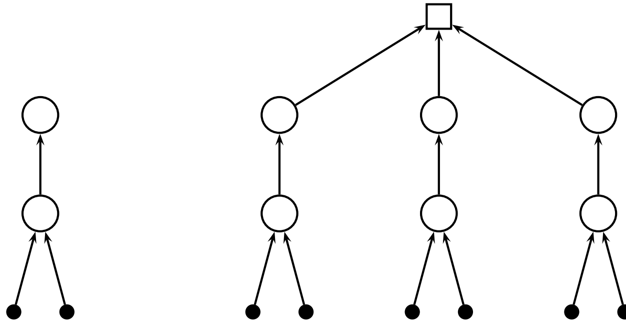

Figure 4.4: Example of an aggregated network with an aggregate unit after the output unit. At the left, the original network with an aggregate unit is shown. The network processes input vectors with two attributes. There is one hidden and one output unit. At the right, the unfolded network is shown for an input bag with three tuples. The square node represents the aggregate function g that is used. Connections to this node have weight 1.

The larger the value C, the closer this function will approximate the real max function. It can be shown that:

$$|smax_C\{z_1, \ldots, z_n\} - max\{z_1, \ldots, z_n\}| < \frac{ln(n)}{C} \tag{4.27}$$

Instead of choosing a value for C that is large enough to give a good approximation, one could take the limit for C going to infinity, which results in the real max function and its derivative:

$$max\{z_1, \ldots, z_n\} = \lim_{C \to \infty} \frac{1}{C}\, ln\left(\sum_{i=1}^{n} e^{Cz_i}\right) \tag{4.28}$$

$$\frac{\partial}{\partial z_j} max\{z_1, \ldots, z_n\} = \begin{cases} 1 & \text{if } x_j = max\{z_1, \ldots, z_n\} \\ 0 & \text{otherwise} \end{cases} \tag{4.29}$$

The disadvantage with this exact function is that the derivative is not continuous anymore, which can deteriorate learning. In the experimental results, these networks will be named MAX and HMAX when the real max function is used, and SMX and HSMX when the softmax function is used. The H indicates that the aggregate function is used at the level of the hidden units,

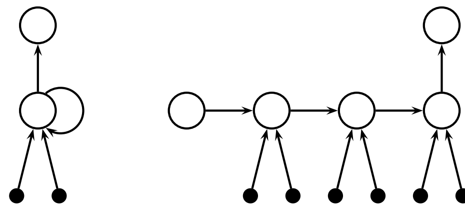

Figure 4.5: Example of a recurrent network. The original network is shown at the left, the unfolded network for three input vectors at the right. The network has two inputs, one hidden and one output unit. There is one recurrent connection, from the hidden unit to itself. The leftmost of the hidden units in the unfolded network represents the initial state of the hidden unit. For more details about the unfolding process, see Section 5.1.1.

otherwise it is used at the output units. In the same way, the names SUM and HSUM will be used for the sum networks.

4.1.4 Recurrent Networks

Instead of feedforward networks, recurrent networks could be used as well. Typically, these networks are used to process sequences of data, but they can also be used to process bags. The difference with a feedforward network is that some connections feed information back into the network instead of propagating it towards the outputs. This gives the network a memory or internal state while processing multiple input vectors. The question is what kind of recurrent connections should be used. Two different types of recurrent networks will be discussed here.

The simplest type of recurrent neural networks, are locally recurrent networks. In this case, each neuron in the hidden layer has exactly one recurrent connection, namely to itself. Fully recurrent networks work in the same way as locally recurrent networks, but now the hidden neurons do not only have a recurrent connection to themselves, but also to all the other hidden neurons. This enhances the expressive power of the network, but it also increases the complexity of the network quadratically with respect to the number of hidden neurons.

The $\#^p$ vectors of the input pattern are fed into the network one after another. For each vector, the activation values of the hidden neurons are updated. After

processing all input vectors, the last activation values are used to compute the activation value of the output neurons. Assuming that the weights of the recurrent connections are represented by w^r_{ij}, the activation values for all units of a fully recurrent network are computed as follows:

- For the hidden units:

$$net^p_{x_i}(0) = 0 \tag{4.30}$$

$$x^p_i(0) = \sigma(net^p_{x_i}(0)) \tag{4.31}$$

$$net^p_{x_i}(j) = \sum_{k=1}^{N} w_{ki} u^p_k(j) + \sum_{k=1}^{H} w^r_{ki} x^p_k(j-1) + b_i \tag{4.32}$$

$$x^p_i(j) = \sigma(net^p_{x_i}(j)) \tag{4.33}$$

- For the output units:

$$net^p_{y_i} = \sum_{k=1}^{H} w'_{ki} x^p_k(\#^p) + b'_i \tag{4.34}$$

$$y^p_i = \sigma(net^p_{y_i}) \tag{4.35}$$

Example 4.5. *Graphically, this procedure is equivalent again to some kind of folding procedure, which is shown in Figure 4.5.*

4.2 Aggregate Cascade-Correlation Networks

Cascade-correlation networks are a special kind of neural networks, constructed one unit at a time. In the next subsection, the original cascade-correlation algorithm will be discussed. After that, a number of new units, capable of aggregation, will be presented. These units will then be integrated in an adapted version of the cascade-correlation network, resulting in a cascade-correlation algorithm that can learn concepts involving aggregation. The resulting networks are called aggregate cascade-correlation networks (ACCNs).

4.2.1 The Original Cascade-Correlation Network

The idea behind the original cascade-correlation algorithm [14] is to learn not only the weights, but also the structure of the network at the same time. This is done in a constructive way, meaning that only one neuron at a time is trained and then added to the network. At first, a network without any hidden unit is created, and then hidden neurons are added, one by one, until some stopping criterion is satisfied. Once a hidden neuron has been added to the network, its weights remain fixed throughout the rest of the procedure. This also means that, besides the actual input vector, the output values of these existing hidden units can be used as extra inputs for any new hidden neuron. At the output, a sigmoid or a linear function can be used. A schema of the network is shown in Figure 4.6. Assume that the network has grown to a network with H hidden units, the weights of the connections from the existing hidden units to the new one are denoted by w^*_{ij} and the weights of the connections from the inputs to the outputs by w''_{ij}. Then the activation values of the network are computed as follows:

- For the hidden units:

$$net^p_{x_i} = \sum_{j=1}^{N} w_{ji} u^p_j + \sum_{j=1}^{i-1} w^*_{ji} x^p_j + b_i \tag{4.36}$$

$$x^p_i = \sigma(net^p_{x_i}) \tag{4.37}$$

- For the output units:

$$net^p_{y_i} = \sum_{j=1}^{H} w'_{ji} x^p_j + \sum_{j=1}^{N} w''_{ji} u^p_j + b'_i \tag{4.38}$$

$$y^p_i = \sigma(net^p_{y_i}) \tag{4.39}$$

The training of the network is done in two alternating phases. Before a new hidden neuron is added, its weights are trained while keeping the weights of all other hidden units fixed. This training is not done by minimizing the squared error between target and output, but by maximizing the correlation with the residual error. The residual error is defined as the difference between the actual target value and the output of the existing network, before adding the new neuron. Instead of the real correlation, a slightly different measure S is taken,

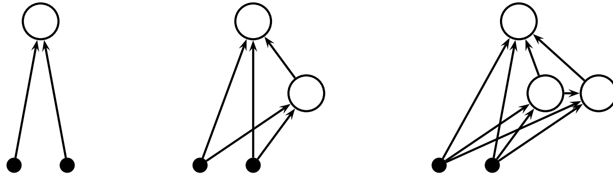

Figure 4.6: Example of the construction of a feedforward cascade-correlation network. At the left, the original network with two inputs and one output unit is depicted. There are no hidden units in this original state. In the middle, the next step step is shown when one hidden unit has been added to the network. At the right, another hidden unit has been added to the network.

in which some normalization factors are omitted and the absolute value is taken:

$$S = \sum_{k=1}^{M} |s_k| \tag{4.40}$$

$$s_k = \sum_{p} \left(x^p_{H+1} - \overline{x^p_{H+1}} \right) \left(e^p_k - \overline{e^p_k} \right) \tag{4.41}$$

$$e^p_k = t^p_k - y^p_k \tag{4.42}$$

with M the number of outputs, x_{H+1} the activation value of the new candidate unit and t^p the target vector for pattern p. When this S value is maximized, the output of the new hidden neuron will correlate well with the residual error. The key idea here is that a unit that correlates well with the residual error, will help to reduce the output error when added to the network. The maximization is done by computing the gradient and performing gradient ascent. This gradient is computed as follows:

$$\frac{\partial S}{\partial w_{ij}} = \sum_{k=1}^{M} \sum_{p} sign(s_k)\, \sigma'(net^p_{x_j})\, (e^p_k - \overline{e^p_k})\, u^p_i \tag{4.43}$$

The gradient for the w^*_{ij} weights and the b_i bias weights can be computed in the same way. Instead of training only one candidate neuron at a time, a pool of neurons, initialized with random weights, can be trained. At the end, the

best one is selected. This increases the chance that a good candidate will be found. Once the best candidate is selected and added to the network, the output weights for the updated network can be trained. If a linear function is used at the outputs instead of a sigmoid, the output weights can be obtained by simple linear regression.

4.2.2 Cascade-Correlation with Aggregate Units

The concept of cascade-correlation networks can be extended to networks for learning from bags of vectors. The crucial difference is that instead of the simple hidden neurons, aggregate units that can process bags are used. For the rest, the network and the training of it works in the same way as for the feedforward cascade-correlation networks. The dataset now consists of patterns with an input bag and an associated target vector t^p. Because the input pattern is a bag of vectors instead of one single vector, it can no longer be used as direct input for the output units, and so this part is dropped from the equation. Each input bag will be processed by the hidden units. Each time a vector of the bag for pattern p has been processed, an intermediate output value for the hidden units can be computed, yielding a sequence of $\#^p$ values for each hidden unit. The final value is used by the output units, but if this network works in the same fashion as the feedforward one, the whole sequence of values can also be used by new hidden units. A schema of an aggregate cascade-correlation network for 2 input vectors is shown in Figure 4.7. The formulas for the network activation values then become:

- For the hidden units:

$$net^p_{x_i}(j) = \sum_{k=1}^{N} w_{ki} u^p_k(j) + \sum_{k=1}^{i-1} w^*_{ki} x^p_k(j) + b_i \quad (4.44)$$

$$z^p_i(j) = \{\sigma(net^p_{x_i}(1)), \ldots, \sigma(net^p_{x_i}(j))\} \quad (4.45)$$

$$x^p_i(j) = g_i(z^p_i(j)) \quad (4.46)$$

- For the output units:

$$net^p_{y_i} = \sum_{j=1}^{H} w'_{ji} x^p_j(\#^p) + b'_i \quad (4.47)$$

$$y^p_i = \sigma(net^p_{y_i}) \quad (4.48)$$

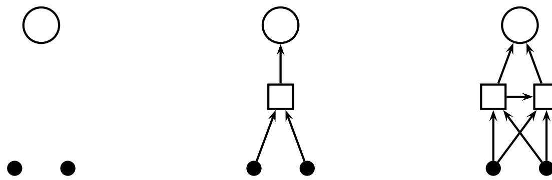

Figure 4.7: Schema for an aggregate cascade-correlation network with 2 input vectors. The circle unit is the feedforward output unit, the square units are aggregate units. The network is constructed in the same fashion as the feedforward network. At the left, the initial situation is shown without any aggregate units, in the middle the network after adding one aggregate unit and at the right the same network after adding a second aggregate unit.

with g_i an aggregate function. Different functions can be used for g_i, as long as they perform some kind of aggregation. This is similar to the aggregate units for the networks defined in Section 4.1. One could take the sum or the max function for instance. Another possibility is to use a locally recurrent neuron instead. The only condition imposed on the g_i function is that it must be differentiable to allow gradient training. Different types of units can easily be combined in the network. For every new unit, several candidates of different types can be trained and the best one selected and added to the network. The four different types that will be considered here, are explained below:

1. The *sum* unit:

$$g_i^{sum}(z_i^p(j)) = \sum_{k=1}^{j} \sigma(net_{x_i}^p(k)) \quad (4.49)$$

$$\frac{\partial}{\partial z_i(j)} g_i^{sum}(z_i^p(j)) = \sum_{k=1}^{j} \sigma'(net_{x_i}^p(k)) \quad (4.50)$$

2. The *smx* (softmax) unit:

$$g_i^{smx}(z_i^p(j)) = \frac{1}{C}\, ln\left(\sum_{k=1}^{j} e^{C\sigma(net_{x_i}^p(k))}\right) \quad (4.51)$$

$$\frac{\partial}{\partial z_i(j)} g_i^{smx}(z_i^p(j)) = \sum_{k=1}^{j} \frac{e^{C\sigma(net_{x_i}^p(k))}}{\sum_{l=1}^{j} e^{C\sigma(net_{x_i}^p(l))}} \sigma'(net_{x_i}^p(k)) \quad (4.52)$$

3. The *max* unit:

$$g_i^{max}(z_i^p(j)) = max(z_i^p(j)) \quad (4.53)$$

$$\frac{\partial}{\partial z_i(j)} g_i^{max}(z_i^p(j)) = \sigma'(net_{x_i}^p(l)) \quad (4.54)$$

$$\text{with } \sigma(net_{x_i}^p(l)) = g_i^{max}(z_i^p(j))$$

4. The *lrc* (locally recurrent) unit:

$$g_i^{lrc}(z_i^p(0)) = 0 \quad (4.55)$$

$$g_i^{lrc}(z_i^p(j)) = \sigma(net_{x_i}^p(j) + w_i^r g_i^{lrc}(z_i^p(j-1))) \quad (4.56)$$

$$\frac{\partial}{\partial z_i(j)} g_i^{lrc}(z_i^p(j)) = \sigma'(net_{x_i}^p(j) + w_i^r g_i^{lrc}(z_i^p(j-1))) \quad (4.57)$$

4.2.3 Aggregate Cascade-Correlation Training

With all parts of the aggregate cascade-correlation network explained, only the training of the network is left for discussion. Each time a new unit should be added to the hidden layer, a pool of units is created containing the four types discussed in the previous section. Weights are initialized randomly. After that, all units in the pool are trained for a number of iterations. This training is basically a gradient ascent, maximizing the correlation with the outputs as defined in Equation 4.40. The computation of the gradient depends of course on the type of unit. The gradient ascent itself is actually done by using resilient propagation, as described in [42] and discussed in Section 2.1.3. This method has the advantage that the step size is determined automatically and convergence is faster than for a fixed step size. The basic idea is to increase the step size when the sign of the gradient remains the same, and decrease the step size when the sign changes.

When all units in the pool have been trained, the best one is chosen. In this case, the best unit is the one with the highest correlation. To be able to compare units of different types with each other, the absolute value of the real correlation has to be computed, and not the S-value from Equation 4.40 in which some normalization constants were omitted. When the unit with the highest correlation has been chosen, it is installed in the network and the output weights have to be trained again. When linear activation functions are used for the output units, the output weights can be determined with least squares linear regression.

In the ideal case, when there is enough data available, a validation set can be used to determine when to stop adding new units. If this is difficult, there is also an alternative stopping criterion. Typically, the first units added to the network will have a high correlation. When more units are added, the correlation will decrease until no more reduction can be made. Training can be stopped when the correlation is below a certain threshold or does not decrease significantly anymore.

4.3 Experiments

To assess the performance of the different simple network structures and the cascade-correlation approach, a number of experiments will be discussed in the next sections. Because there are not that many real-world datasets available that represent data as bags of vectors and because the focus for existing datasets has been mostly on multi-instance problems, a large part of these experiments will be carried out on artificial datasets. The datasets discussed in the following subsections are two artificial datasets, i.e., simple aggregates and trains, two multi-instance datasets, i.e., musk and Thioredoxin-fold proteins, and a simplified version of the financial dataset, which is a relational dataset.

4.3.1 Simple Aggregates

A first, simple experiment to examine the capacity of the new networks, is to create artificial datasets, using predefined aggregate functions to define the target concepts, and then train the networks on these datasets. The data consists of bags with a variable number of elements. Each element of the bag is a vector with five components. Only the first or the first and second component are relevant for the target value, depending on the aggregate function under consideration. The values of these components are randomly generated, but in such a way that the target values are uniformly distributed over the possible target values. All the other components are filled with uniformly distributed random numbers from the interval $[-1, 1]$. It is very likely that the number of vectors in the bags influences the difficulty of the learning task, so different sizes are tested. The datasets denoted as small contain 5 to 10 vectors per bag, the medium datasets 20 to 30 and the large ones 50 to 100. Each dataset contains 3000 bags. A range of different target concepts are used to construct the datasets:

1. **count:** the target is the number of vectors in the bag.

		small	**medium**	**large**
MSE	count	0.06 (0.10)	0.26 (0.01)	0.23 (0.01)
	sum	0.00 (0.00)	0.09 (0.10)	0.23 (0.01)
	max	0.02 (0.00)	0.01 (0.00)	0.01 (0.01)
	avg	0.02 (0.00)	0.01 (0.01)	0.01 (0.01)
	stddev	0.03 (0.00)	0.04 (0.00)	0.04 (0.01)
	cmpcount	0.02 (0.01)	0.01 (0.01)	0.11 (0.07)
	corr	0.07 (0.01)	0.08 (0.04)	0.04 (0.01)
accuracy	even	52.64 (0.01)	50.19 (0.02)	50.23 (0.01)
	distr	51.62 (0.05)	63.49 (0.02)	58.70 (0.02)
	select	90.55 (0.03)	63.47 (0.03)	50.42 (0.02)
	conj	94.43 (0.01)	68.09 (0.10)	49.11 (0.05)
	disj	72.08 (0.03)	51.24 (0.05)	50.32 (0.02)

Table 4.1: Results for the simple aggregate datasets with fully recurrent networks. Accuracies or mean squared errors are given, depending on whether the target is numerical or nominal, together with the standard deviations. The columns small, medium and large refer to the size of the bags in the datasets.

2. **sum:** the target is the sum of all values of the first component of the bag vectors.
3. **max:** the target is the maximum value of the first component of the bag vectors.
4. **avg:** the target is the average value of the first component of the bag vectors.
5. **stddev:** the target is the standard deviation of the values of the first component of the bag vectors.
6. **cmpcount:** the target is the number of bag vectors for which the value of the first component is smaller than the value of the second component.
7. **corr:** the target is the correlation between the first two components of the bag vectors.
8. **even:** the target is one if the number of positive values for the first component is even, and zero if it is odd.
9. **distr:** the target is one if the values of the first component come from a Gaussian distribution, and zero if they are from a uniform distribution.

		small	medium	large
MSE	count	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	sum	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	max	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	avg	0.04 (0.03)	0.01 (0.01)	0.01 (0.01)
	stddev	0.02 (0.02)	0.01 (0.01)	0.00 (0.00)
	cmpcount	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
	corr	0.04 (0.03)	0.02 (0.02)	0.01 (0.01)
accuracy	even	51.59 (0.05)	51.10 (0.04)	51.34 (0.10)
	distr	66.48 (0.05)	77.97 (0.07)	84.23 (0.09)
	select	99.21 (0.07)	99.13 (0.10)	98.83 (0.10)
	conj	100.00 (0.04)	100.00 (0.00)	99.89 (0.10)
	disj	99.13 (0.06)	98.57 (0.10)	96.54 (0.11)

Table 4.2: Results for the simple aggregate datasets with aggregate cascade-correlation networks. Accuracies or mean squared errors are given, depending on whether the target is numerical or nominal, together with the standard deviations. The columns small, medium and large refer to the size of the bags in the datasets.

10. **select:** the target is one if at least one of the values of the first component lies in a given interval, and zero otherwise.
11. **conj:** the target is one if there is at least one vector in the bag for which the first and the second component lie in a certain interval.
12. **disj:** the target is one if there is at least one vector in the bag for which the first or the second component lies in a certain interval.

The first 7 datasets have a numerical target, the other 5 a nominal target. In case of a nominal target, the number of positive and negative examples are equal, while in the case of a numerical target, the target values are uniformly distributed over the target interval. The experiments are done using 10-fold cross-validation. One fold is used as test set, 7 folds are used for training and 2 folds are used as validation set to determine when to stop training. For the simple network structures, 3 hidden units are used. Because the number of inputs is only five and the concepts to be learned are not that complicated, a small number of units should be sufficient. Training is done using resilient propagation and 1000 training iterations. The validation set is

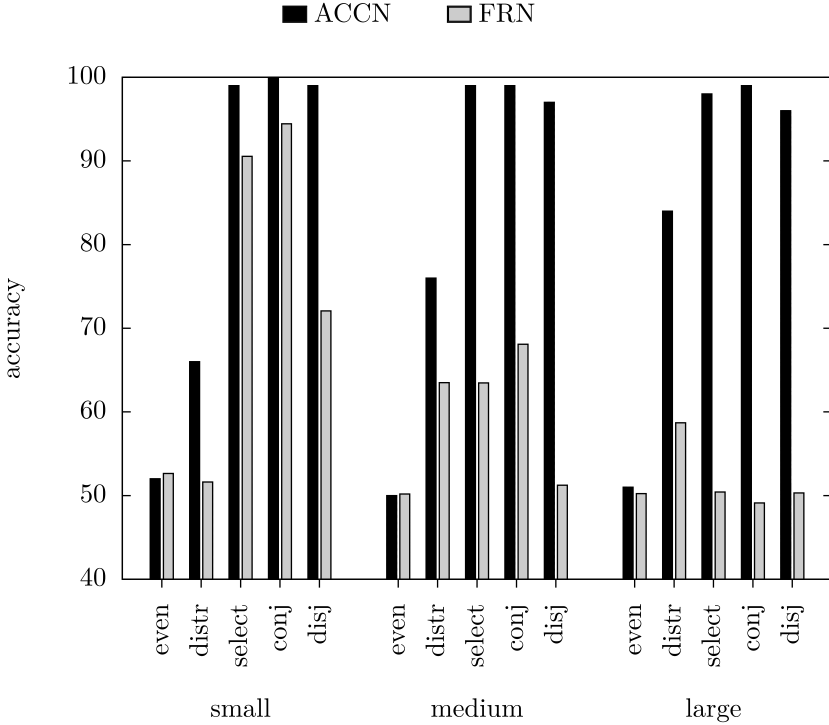

Figure 4.8: Comparison between fully recurrent networks (FRN) and aggregate cascade-correlation networks (ACCN) on the aggregate classification datasets.

used to determine the optimal number of iterations. For the cascade-correlation networks, the maximum number of hidden units is limited to 10. The number of candidate units trained in every step is 20, which means that there are five units of every type. Each unit is trained for 500 iterations, which should be enough to have converged to optimal weights. For the datasets with nominal target, the accuracy is reported and for the sets with numerical targets the mean squared error is given. Standard deviation is reported as well.

Table 4.1 shows the results for fully recurrent networks. On some datasets, e.g. the small version of *select* and *conj*, the results are reasonably good, but on others the performance is not that convincing. Results vary for other network types, but the overall impression is quite the same. The results for the aggregate cascade-correlation networks are shown in Table 4.2. From these results, it is clear that most concepts can be learned reasonably well, some very well. Only the *even* concept seems really impossible to learn. It should be noted that this

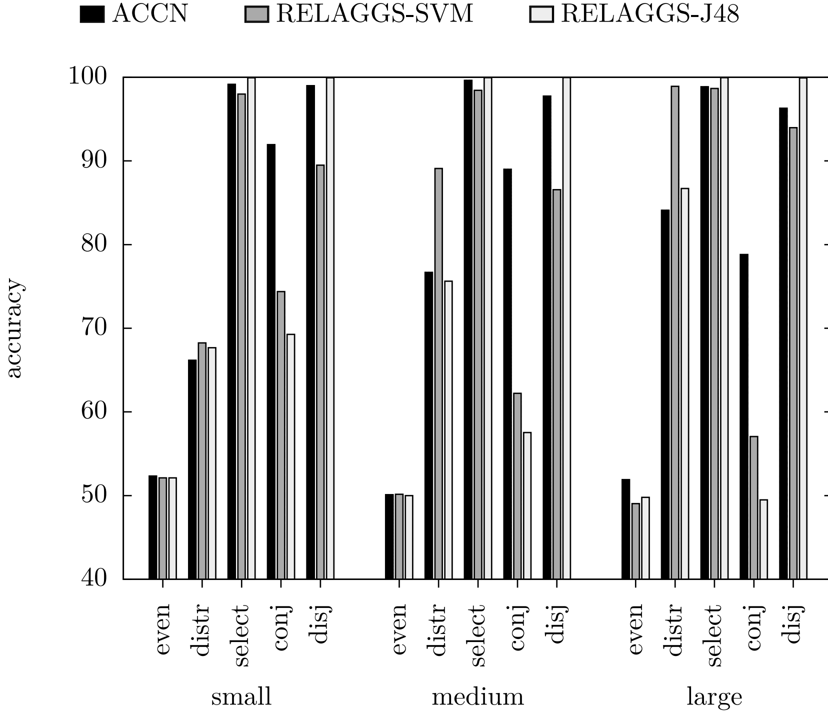

Figure 4.9: Comparison between aggregate cascade-correlation networks (ACCN) and the RELAGGS system on the aggregate classification datasets. Two different possibilities are considered: RELAGGS in combination with a support vector machine (RELAGGS-SVM) and RELAGGS with a decision tree (RELAGGS-J48).

concept can be seen as the aggregate version of the XOR concept, which is known to pose problems for neural networks with only one layer. A similar situation occurs here, which explains the bad performance for this dataset. For the *distr* concept, the number of vectors must be large to be able to learn it well. This makes sense because it is easier to say whether a bag of values comes from a normal or uniform distribution if the bag is larger than when it is rather small. Compared with the results for the fully recurrent networks, it is clear that ACCNs perform better. One of the major problems with the recurrent networks, is the decreasing performance on larger bags. Looking at the results for the *select* datasets for instance, the accuracy on the dataset with small bags is still reasonable for the recurrent networks, although the accuracy for the ACCNs is better. But for the datasets with larger bags, the accuracy goes

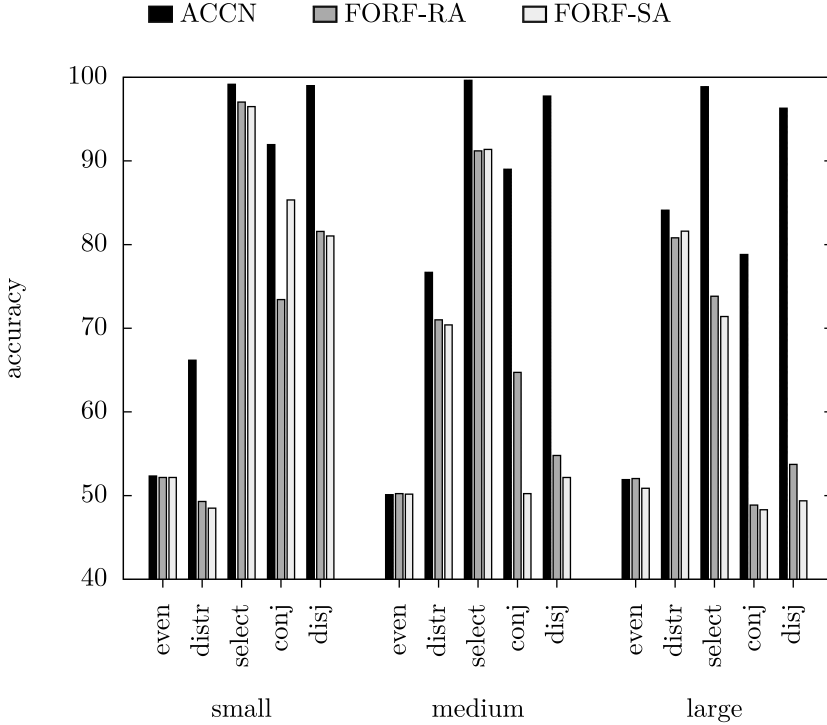

Figure 4.10: Comparison between aggregate cascade-correlation networks (ACCN) and first-order random forests (FORF) on the aggregate classification datasets. Two different possibilities are considered: FORF with simple aggregates (FORF-SA) and FORF with more complex, refined aggregates (FORF-RA).

down for the recurrent networks while it remains about the same for the ACCNs. Overall, it is clear that ACCNs are a better choice than the recurrent networks or the other simple network types. This is clearly illustrated in Figure 4.8 where the accuracies are plotted for both ACCNs and fully recurrent networks.

A comparison can also be made with other methods, e.g. RELAGGS or ILP methods. For RELAGGS, the Weka [19] implementation is used and two different machine learning algorithms are considered, i.e., support vector machines (the SMO algorithm in Weka, using an RBF kernel) and decision trees (the J48 algorithm in Weka). The performances are shown in Figure 4.9. Results vary somewhat, but in general ACCNs perform very well compared with RELAGGS. Only for the *distr* dataset it is clear that RELAGGS-SVM is

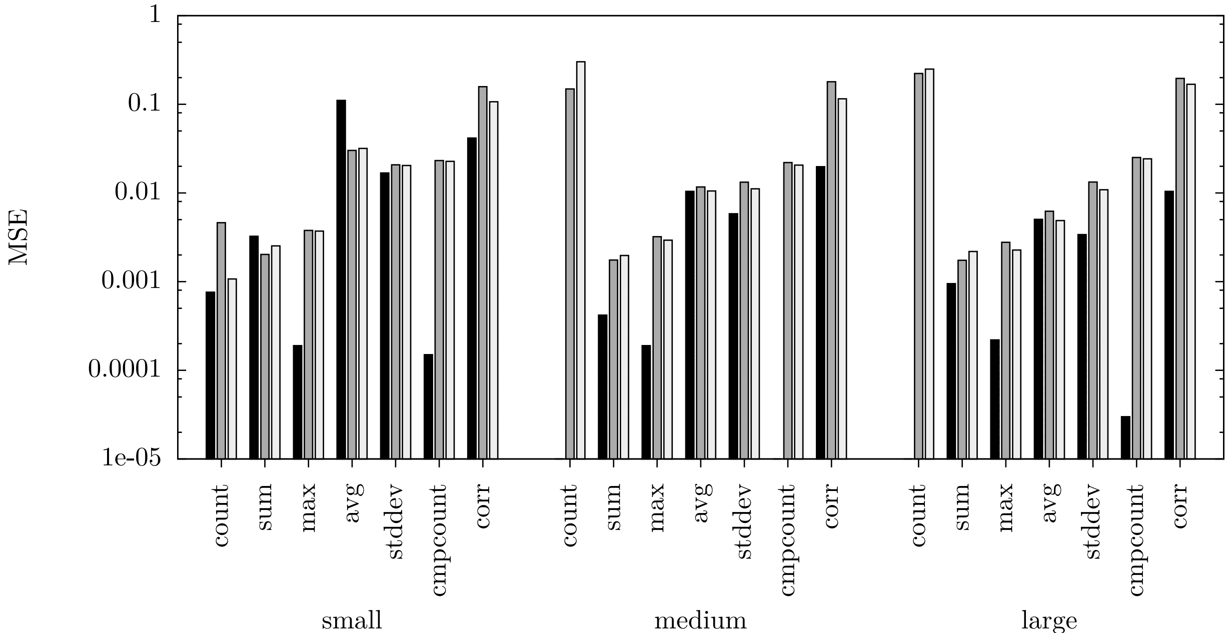

Figure 4.11: Comparison between aggregate cascade-correlation networks (ACCN) and first-order random forests (FORF) on the aggregate regression datasets. Two different possibilities are considered: FORF with simple aggregates (FORF-SA) and FORF with more complex, refined aggregates (FORF-RA).

a better choice. For the *conj* datasets on the other hand, ACCNs perform a lot better than RELAGGS. These large differences in performance are typical when the propositionalization is not very suitable for the considered problem. As these datasets only consider simple aggregate concepts in which only one vector component is involved, RELAGGS still performs quite well overall, but this will probably no longer be the case when more complicated aggregate concepts are considered.

Another comparison is made with an ILP system that includes aggregate predicates. The considered system is based on TILDE [5], which builds decision trees with predicates in the nodes. Here, an extended version of TILDE is used in which aggregate predicates are added to the system and random forests are built instead of single trees. Two different options are considered, i.e. first-order random forests with simple aggregates (FORF-SA) and first-order random forests with refined aggregates (FORF-RA) [50, 49]. The results are shown in Figures 4.10 and 4.11. Overall, ACCNs perform better than FORFs, in many cases the difference is even quite large. The difference also tends to grow for the datasets with larger bags of vectors. It is not really clear why FORFs have more problems dealing with this than ACCNs. It is clear however, that in general, the learning problem becomes more difficult when the bags are larger, because more data means more noise and more irrelevant information.

4.3.2 Trains

The trains datasets are also artificially created datasets representing trains and their direction. Every train consists of a number of cars, carrying some loads. Some of the trains are eastbound, the others are westbound. The direction of the trains is what has to be predicted and this target concept is based on the properties of the cars of a train and their loads. The cars of the train constitute a bag of tuples for each train. A data generator [34] for this train problem was used to create 12 datasets with different properties. Sets 1 to 4 consist of short trains, having 2 to 6 cars. Datasets 5 to 8 are similar to sets 1 to 4, except that they contain longer trains. Each of these trains consists of 20 to 29 cars. The used concepts are the same as for sets 1 to 4, except that the numbers in the aggregations are adapted to the longer length of the trains. Datasets 9 to 12 contain noisy data. This means that a number of samples have been mislabeled. The used concepts for the different datasets are as follows:

1. Trains having at least one circle load are eastbound, the others are westbound.

2. Trains having at least one circle or rectangle load and at least one car with peaked roof or 3 wheels are eastbound, the others are westbound.
3. Trains having more than 8 wheels in total are eastbound, the others are westbound.
4. Trains having more than 3 wheels in total and at least 2 rectangle loads and maximum 5 cars are eastbound, the others are westbound.
5. Same concept as for set 1.
6. Same concept as for set 2.
7. Trains having more than 53 wheels in total are eastbound, the others are westbound.
8. Trains having more than 45 wheels in total and at least 10 rectangle loads and maximum 27 cars are eastbound, the others are westbound.
9. Same concept as for set 1, but with 5% noise.
10. Same concept as for set 1, but with 15% noise.
11. Same concept as for set 3, but with 5% noise.
12. Same concept as for set 3, but with 15% noise.

The training setup is the same as for the simple aggregate datasets. The results for ACCNs and fully recurrent networks are given in Figures 4.12 and 4.13. It is clear that most concepts can be learned well with ACCNs. Most of the datasets without noise have an accuracy very close to 100%. Only for set 8, which has the most difficult concept, it is impossible to get close to perfect accuracy. For the datasets with noise, the accuracies are all close to 100% minus the percentage of noise, which means that the method is noise-resistant. Compared with the fully recurrent networks, the results for ACCNs are always better again. Sometimes the difference is quite spectacular. For these datasets, the size of the bags also has an important influence on the accuracy of the recurrent networks.

Again, a comparison is made with other methods. In Figure 4.14, accuracies for ACCNs and RELAGGS are compared with each other. In general, ACCNs perform better than RELAGGS. The difference can be quite large, e.g. for dataset 6 where ACCNs have an accuracy of almost 100% while the RELAGGS accuracy is below 80%. For one dataset, number 8, RELAGGS performs better than ACCNs, but overall ACCNs seem to be the better choice.

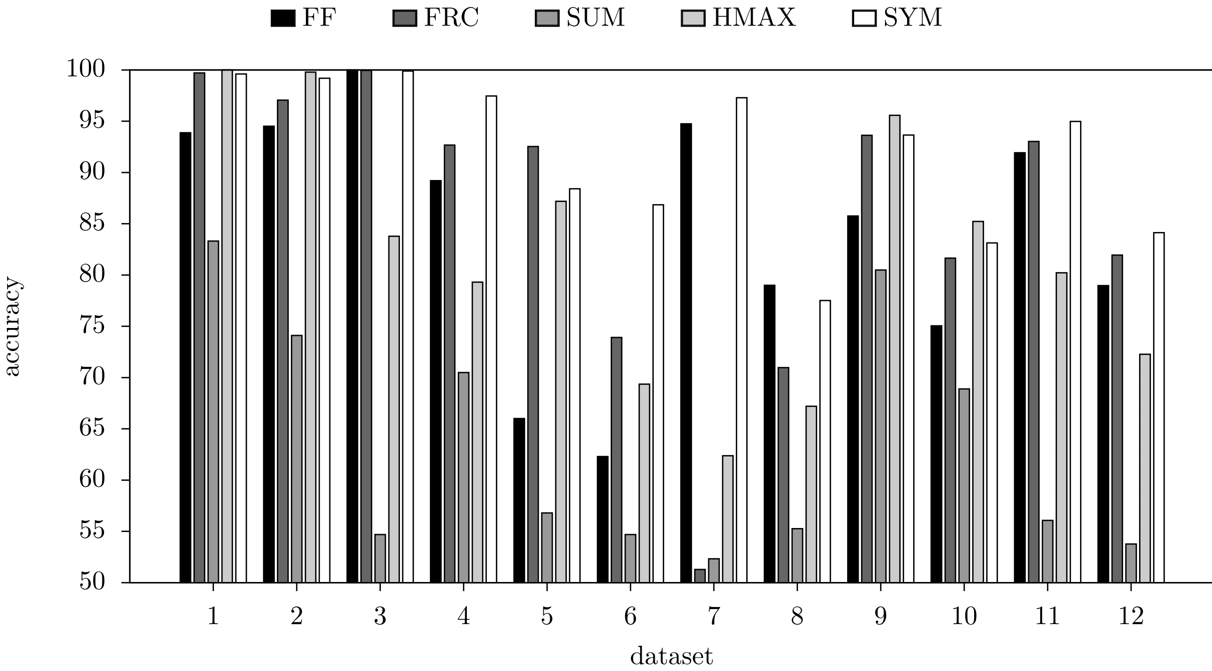

Figure 4.12: Accuracies on the train datasets for different simple network structures. The considered network structures are standard feedforward networks (FF), fully recurrent networks (FRC), aggregate networks with a sum function placed after the output units (SUM), aggregate networks with a max function placed after the hidden units (HMAX) and symmetric networks (SYM).

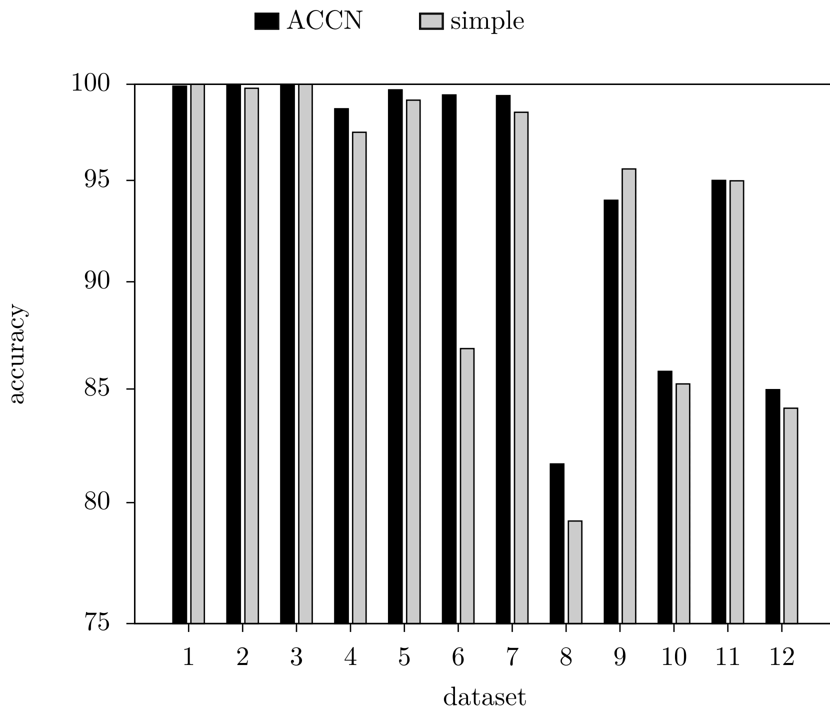

Figure 4.13: Accuracies on the train datasets for aggregate cascade-correlation networks (ACCN) and the best simple network structure.

Just as for the aggregate datasets, a comparison with first-order random forests (FORF) is made in Figure 4.15. Overall, ACCNs perform better than FORFs. For most datasets, the accuracies are very close to each other, but for dataset 6 for instance, the accuracy of FORFs is between 80 and 90%, while the accuracy of ACCNs was almost 100%. So again, ACCNs seem to be the better choice.

4.3.3 Musk

Musk is a well-known multi-instance dataset [11]. Each data instance stands for a molecule, represented by a bag of all its possible conformations. A conformation is described by 166 numerical features. The molecules have to be classified as musk or non-musk. Because it is a multi-instance problem, the target value is determined by the existence of a specific conformation. If there is at least one conformation in the bag with certain properties, then the molecule

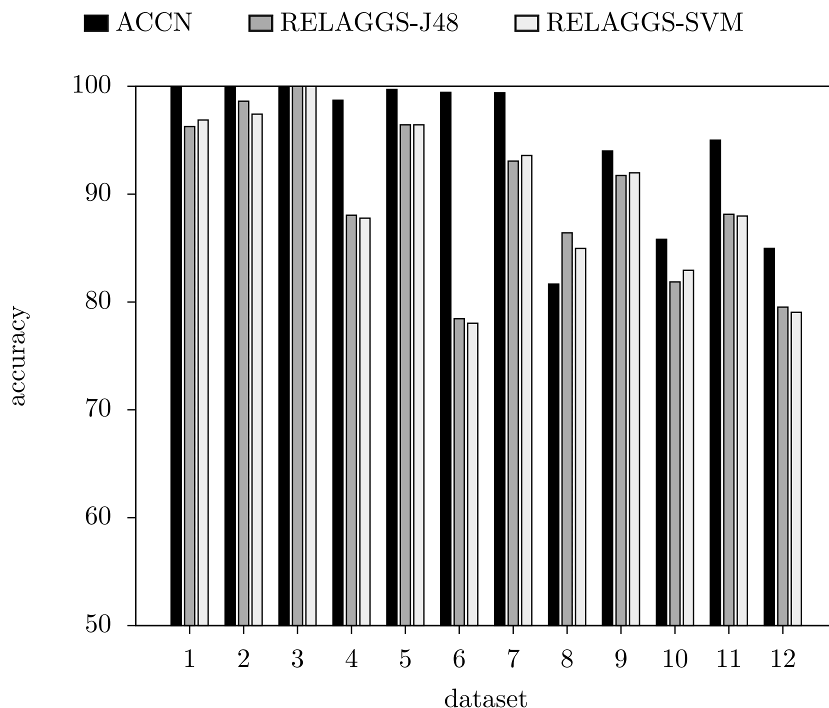

Figure 4.14: Comparison between aggregate cascade-correlation networks (ACCN) and the RELAGGS system on the train datasets. Two different possibilities are considered: RELAGGS in combination with a support vector machine (RELAGGS-SVM) and RELAGGS with a decision tree (RELAGGS-J48).

can be classified as positive, otherwise it is negative. The dataset consists of two parts. The first part contains 92 molecules, the second part 102. In each bag, there are between 2 and 40 conformations for the first part, and between 1 and 1044 for the second part.

Experiments are carried out using 10-fold cross-validation. For the ACCNs, a pool of 20 neurons and 500 training iterations are used in every step. The value of the correlation is used as stopping criterion. The results for the musk datasets can be found in Table 4.3 and Figures 4.16 and 4.17.

For the simple network structures, the results vary quite a lot. Overall, the best accuracies seem to be achieved with aggregate networks with max functions

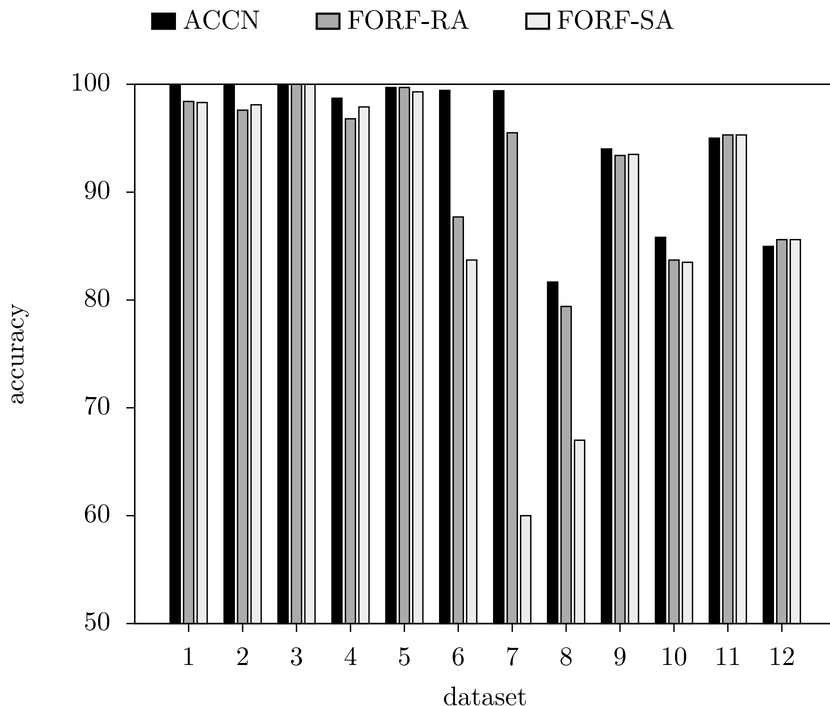

Figure 4.15: Comparison between aggregate cascade-correlation networks (ACCN) and first-order random forests (FORF) on the train datasets. Two different possibilities are considered: FORF with simple aggregates (FORF-SA) and FORF with more complex, refined aggregates (FORF-RA).

after the hidden units (HMAX and HSMX). This makes sense because this would be the natural choice in the case of multi-instance problems. The accuracies for the ACCNs are not bad but not excellent either compared with the other methods. Some of these methods were specifically designed for multi-instance problems while the neural networks described here are more general. This makes the multi-instance methods better suited for this task and so it is normal that their performance is better. The neural networks might also suffer from poor learning because of relatively few data or skewness in the data. The final ACCNs are for instance very small, in most cases with just one hidden unit, because training is stopped early. The selected unit during training is almost always a MAX or SMX unit, the most logical choice in case of a multi-instance problem.

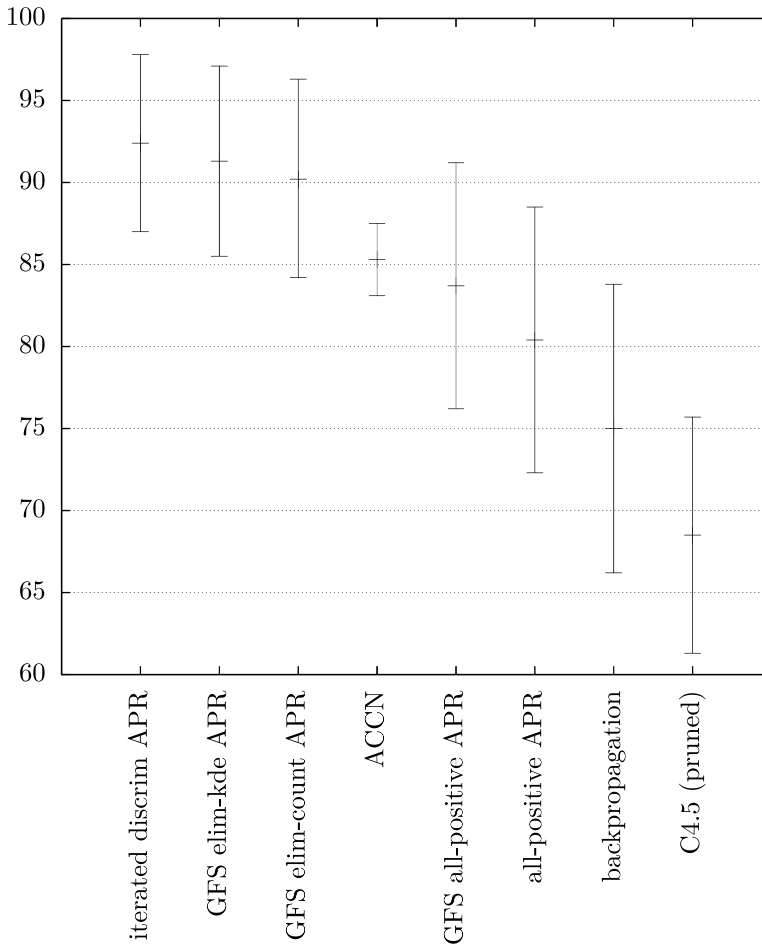

Figure 4.16: Accuracies and 95% confidence intervals for the musk dataset 1. Results are the same as in Table 4.3. For the sake of clarity, methods are sorted from highest to lowest accuracy. It can be seen that ACCNs lie somewhere in the middle.

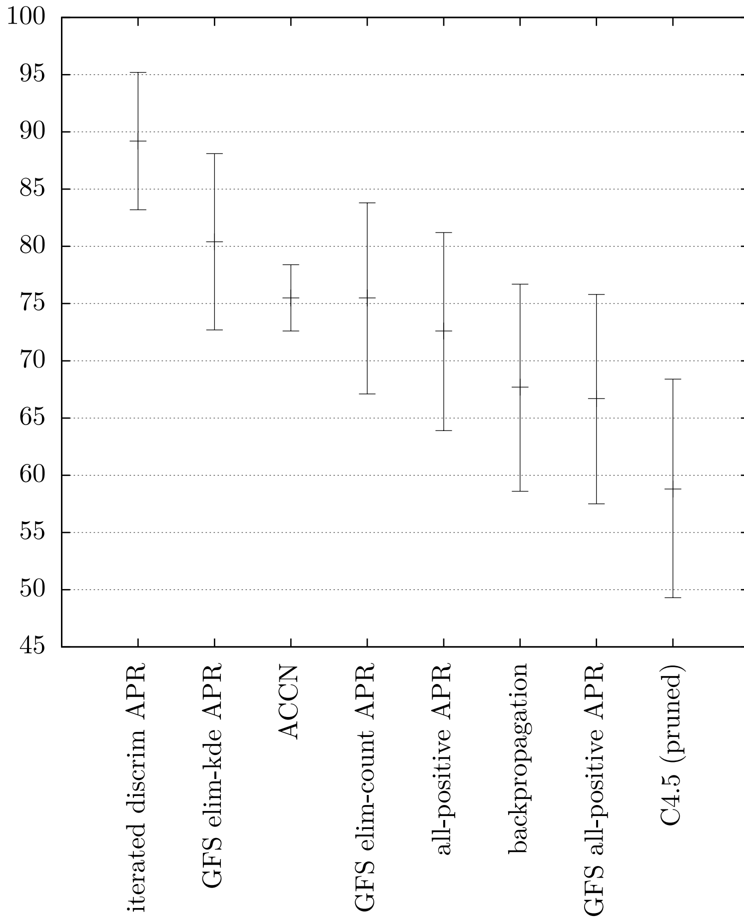

Figure 4.17: Accuracies and 95% confidence intervals for the musk dataset 2. Results are the same as in Table 4.3. For the sake of clarity, methods are sorted from highest to lowest accuracy. It can be seen that ACCNs lie somewhere in the middle.

	musk 1	**musk 2**
iterated discrim APR	92.4 [87.0-97.8]	89.2 [83.2-95.2]
GFS elim-kde APR	91.3 [85.5-97.1]	80.4 [72.7-88.1]
GFS elim-count APR	90.2 [84.2-96.3]	75.5 [67.1-83.8]
GFS all-positive APR	83.7 [76.2-91.2]	66.7 [57.5-75.8]
all-positive APR	80.4 [72.3-88.5]	72.6 [63.9-81.2]
backpropagation	75.0 [66.2-83.8]	67.7 [58.6-76.7]
C4.5 (pruned)	68.5 [40.9-61.3]	58.8 [49.3-68.4]
SYM	75.8 [74.1-77.5]	71.1 [68.8-73.4]
LRC	77.2 [75.6-78.8]	75.2 [72.7-77.7]
FRC	74.9 [73.4-76.4]	73.3 [70.8-75.8]
SUM	56.6 [54.2-59.0]	49.0 [46.3-51.7]
HSUM	78.2 [76.6-79.8]	65.5 [63.9-67.1]
MAX	73.1 [71.5-75.7]	65.6 [63.4-67.8]
HMAX	79.0 [77.9-80.1]	77.4 [75.7-79.1]
SMX	72.2 [71.0-73.4]	62.8 [60.8-64.8]
HSMX	77.3 [75.7-78.9]	78.1 [76.7-79.5]
ACCN	85.3 [83.1-87.5]	75.5 [72.6-78.4]

Table 4.3: Accuracies and 95% confidence intervals for the musk datasets using 10-fold crossvalidation. Results for other methods than ACCN are obtained from [11].

4.3.4 Thioredoxin-Fold Proteins

In this classification task, proteins have to be classified as belonging to the Thioredoxin-fold family or not [51]. It is difficult to do this based on the primary sequence of the proteins, for instance by using hidden Markov models, because there is a low conservation of the primary sequence in this family of proteins. One approach to deal with this problem has been to transform the data into bags of vectors. In [51], this transformation is done in three steps. First, the primary sequence motif, which is known to exist in all Thioredoxin-fold proteins, is identified. Around this motif, aligned subsequences are extracted. Finally, these windows are mapped to 8-dimensional numerical properties. For further details about this transformation, see [51]. The relevant transformation is referred to as motif-based alignment.

The result of this transformation is a dataset containing 193 proteins, each

	TP	**TN**
MIL	0.74	0.88
ACCN	0.614 (0.147)	0.838 (0.071)

Table 4.4: Results for 3-fold cross-validation on the Thioredoxin-fold proteins dataset. True positive and true negative rates are given, together with the standard deviation over ten runs for ACCNs. MIL denotes a multi-instance learner and this result was reported in [51].

described by a bag of 8-dimensional feature vectors. Of these 193 proteins, 25 are labeled positive and 168 negative. The bags contain 35 to 189 vectors. Two different experiments were carried out, using the same settings as in [51]. In the first, simple setting, the dataset is divided in three parts, using two of them for training and one for testing. It is clear by now that the results obtained with ACCNs are as good or better than the results obtained with the simple network structures, so the focus is on ACCNs only here. The training setup is the same as for the musk datasets. The results are given in Table 4.4. One can see that the results are worse than for a multi-instance learner. Standard deviations are also quite large, especially for the true positive rate. The skewness of the dataset is probably the cause of these problems.

Results were also obtained for a second, more difficult experiment carried out in [51], in which 5 of the 25 positive proteins were removed, retaining only the 20 most dissimilar ones. The negative examples are divided in 8 folds, the positive examples in 20 folds of 1 example each. A jack-knife test is performed, taking one of the negative folds and one of the positive folds as test set, the other 19 positive folds and one of the negative folds as training set. This yields a total of 160 different runs. Results for this experiment are shown in Table 4.5. The true positive rate is better now than for MIL, but the true negative rate is worse. Overall, the situation seems to be more or less the same as for the musk datasets where it was also not possible to achieve the same accuracy as multi-instance methods.

4.3.5 Financial

For this last experiment, the financial dataset of the discovery challenge organized at PKDD'99 and PKDD'00 is used [3]. The dataset contains 8 different relations in total and the goal is to classify loans as good or bad. There are 234 loans, 203 of which are good and 31 bad. This means that the default accuracy is already 86.8%. Each loan is linked to an account and for

	TP	**TN**
MIL	0.75	0.75
ACCN	0.850 (0.035)	0.681 (0.028)

Table 4.5: Results for jack-knife tests on the Thioredoxin-fold proteins dataset. True positive and true negative rates are given, together with the standard deviation over all 160 experiments for ACCNs. MIL stands for a multi-instance learner and this result was reported in [51].

	accuracy
FORF-LA	90.8 (1.7)
DINUS-C	85.1 (10.3)
RELAGGS	88.0 (6.5)
PROGOL	86.3 (7.1)
ACCN	87.2 (2.3)

Table 4.6: Results on the financial dataset using 10-fold cross-validation. Results for FORF-LA, DINUS-C, RELAGGS and PROGOL are obtained from [50] and [29].

every account there is a bag of transactions. The number of transactions varies between 93 and 594. Only these bags will be used by the ACCNs to learn a classification of the loans, all the other information is ignored because we only work with a single bag for each data instance. It is clear that this approach will only work if these transactions contain the most important information to learn the target function. The experiments were done using 10-fold cross-validation. Results are summarized in Table 4.6. The accuracy for the ACCNs is not significantly higher than the default accuracy, but neither is the accuracy of the other methods, except FORF-LA. It should also be noted that these other methods use more information than only the bags of transactions. Overall, it is hard to draw final conclusions from this experiment.

4.4 Conclusions

At the start of this chapter, a number of simple network structures were proposed, adapting standard neural networks to enable them to process bags of vectors. After that, the ideas behind these simple network structures were

combined in a cascade-correlation approach. The general idea was to create a learning method with parameterized aggregate features that can be tuned, with the right parameters, to the appropriate aggregate features to predict targets on bags of vectors. The experiments that were carried out, showed three things. First of all, the cascade-correlation approach outperformed the simple network structures convincingly. The results were almost always at least as good or better than with the best simple network structure. Second, the results with ACCNs turned out to be as good or better in most cases than the results for other methods such as RELAGGS and FORFs. These results were mainly achieved on artificial datasets, because there were not that many real-world datasets of this kind available. Third, for multi-instance problems, the ACCNs do perform reasonably well but not as good as some other methods. This is however not a surprise as these other methods are designed specifically for this class of problems while ACCNs are more general and have a broader scope.

Chapter 5

Neural Networks for Structured Data

In the previous chapters, the focus was on largely unstructured data, with input domains consisting of bags of unordered tuples. The topic of this chapter will be neural networks for structured data like graph structures. Contrary to learning from bags, this topic has received a lot of attention in neural network research. Therefore, three existing approaches will be discussed, starting from recurrent networks for sequences, to recursive networks for directed, acyclic graphs and graph neural networks for graphs in general. Each of these approaches can be seen as an extension of the previous one. This chapter contains no novel work by the author, but it provides an overview of some existing methods for structured and relational domains, as this is in some regards closely related to the methods discussed in the previous and the next chapters. The given overview is not meant as a survey and it is not exhaustive, as the literature on this topic is quite extensive and varied, but it should provide sufficient background and context. It should also help to understand connections between these methods and the ones discussed in the next chapters.

5.1 Recurrent Networks

Recurrent networks are the most straightforward extension of feedforward networks to structural domains. They are simply created by adding recurrent connections to the feedforward networks, enabling them to process sequences. Recurrent networks and the folding procedure were already used in the previous

chapter, but these concepts are first restated here because they serve as the starting point of the discussion in this chapter.

5.1.1 Unfolding Networks

The simplest form of structured data a neural network can deal with, are sequences. For this, a recurrent network is needed. A recurrent network is a network in which the connections between the neurons introduce at least one cycle in the network. This means that the computation of the activation values of some neurons depends on previous activation values of one or more neurons. It implies that the network gets some kind of memory. Only discrete time recurrent networks are considered here, where in each time step one vector of the sequence is presented to the network, followed by a new computation of the activation values, proceeding from inputs to outputs. The last activation values are also the final output values for the sequence as a whole.

The recurrent network can be split up in two parts. The part with the recurrent connections is the *transition network*, implementing a *transition function* f_w, with w the weight parameters. This part of the network must be updated for each tuple in the sequence. The activation values of the units in this part can be gathered in a *state vector*. For each tuple presented to the network, a new state vector is computed. Then the transition network computes the transition from one state to another. The other part is the feedforward *output network*, implementing an *output function* g_w. This part only has to be updated after the last tuple has been presented to the network, because it is only used to compute the final output for the total sequence.

This procedure can also be looked at as *unfolding* the network for a given input sequence. This involves creating folds or copies of the network, at least the transition part of the network, without the recurrent connections. There is one fold for each input vector in the sequence. The recurrent connections are then turned into feedforward connections between the successive network copies. This yields a standard feedforward network, with the constraint that the connection weights are shared between the copies in the unfolded network. So when backpropagation is used to train the unfolded network, weight adaptation made to the weights of one copy should also be made to the other copies. Training in this way corresponds to the backpropagation through time (BPTT) algorithm that is often used to train these networks [52]. All the folds of the transition network f_w together will be referred to as the function F_w in the remainder of the text.

Example 5.1. *An example of an input sequence, represented as a linear graph, is given in Figure 5.1. The unfolding procedure for this example is shown in*

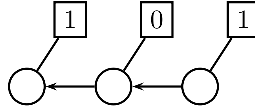

Figure 5.1: Example of an input sequence, represented as a linear graph. Each node stands for one tuple or vector in the sequence and to each node a label is attached representing the attributes of that node. In this case, only one attribute is given. The edges in the graph are drawn from the last tuple in the sequence towards the first one.

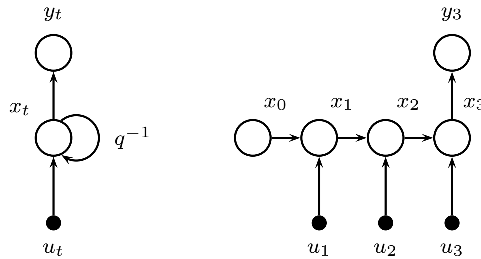

Figure 5.2: The unfolding procedure for a simple network. The original recurrent network is shown at the left with one input unit, one recurrent hidden unit and one output unit. The unfolded network for a sequence with three input vectors is shown at the right. Each input vector has only one attribute so there is only one input. The units u_t and x_t belong to the transition network, y_t to the output network. For this reason, there are multiple copies of x_t in the unfolding network, and only one for y_t.

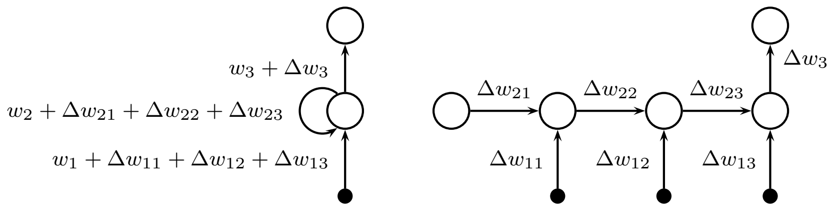

Figure 5.3: The backpropagation through time training algorithm. At the left, the recurrent network is shown with formulas for the new weights, at the right the corresponding unfolded network for a sequence with three input vectors. The delta values can be computed for each connection in the unfolded network with standard backpropagation, and then applied to the weights of the original network.

Figure 5.2. The corresponding weight updating with backpropagation through time is illustrated in Figure 5.3.

5.1.2 Shift Operator

With regard to the unfolding procedure, the recurrent connections actually correspond to a shift operator, a discrete-time operator that can be applied to a temporal variable.

Definition 5.1 (Shift Operator). *If $x_0, \ldots, x_T$ is a sequence of $T+1$ values for a variable X at successive times, then the shift operator q^{-1} applied to x_t returns x_{t-1}, the value for the variable x at time $t-1$, i.e., $q^{-1}x_t = x_{t-1}$.*

When considering the sequence as a directed graph, this shift operator can be regarded as an operator that returns the child node when applied to a node. Shift operators can be composed, for instance $q^{-2} = q^{-1}q^{-1}$, and $q^0 = 1$ is the neutral operator defined as $q^0 x_t = x_t$.

In the case of recurrent networks, the considered variable is the output of a neuron in the network. The sequence of values are the activation values. Recurrent connections correspond to a shift operator on this sequence of activation values.

5.2 Recursive Networks

Recurrent networks can only be applied to sequences, but they can easily be extended to graphs without a strictly linear structure. Networks for these graphs are called recursive networks.

5.2.1 Directed Positional Acyclic Graphs

Recurrent networks can be generalized to recursive networks that operate on directed positional acyclic graphs (DPAGs), a superset of sequences. The edges in these graphs are directed, just as for the sequences, but in a DPAG each node can have more than one incoming or outgoing edge. The only limitation is that the graph should be without cycles. The graph is also positional, which means that all outgoing edges of each node are ordered.

Definition 5.2 (Directed Positional Acyclic Graph). *A directed positional acyclic graph (DPAG) $G(V, E)$ is a graph with vertices V and edges E*

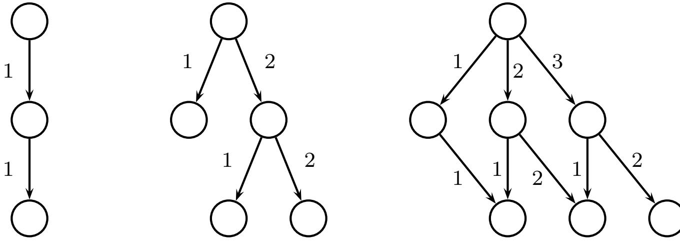

Figure 5.4: Examples of directed positional acyclic graphs. The numbers on the edges indicate the position of the edge. To each node a vector with attributes is attached, just as for the sequences, but they are left out in this picture.

where there is no directed path in E that connects any vertex $v \in V$ to itself, and where all outgoing edges $(v_1, v_2) \in out(v_1)$ of each vertex v_1 are ordered. So there exists an ordering function $pos(v_1, v_2) : E \mapsto \mathbb{N}_0^+$ which maps an edge (v_1, v_2) to its position within $out(v_1)$, the set of all outgoing edges of v_1.

Example 5.2. *Figure 5.4 shows some examples of DPAGs. All trees and sequences are also DPAGs.*

Definition 5.3 (Supersource). *A supersource of a graph $G(V, E)$ is a vertex $v_1 \in V$ from which all other vertices $v_2 \in V$ can be reached and that has no incoming edges.*

Definition 5.4 (Supervised DPAG). *A DPAG $G(V, E)$ is supervised if it has at least one supervised node, i.e., a node that has a target vector.*

For the classification and regression tasks under consideration, all data instances can now be represented by DPAGs instead of sequences. Only graphs with a single supervised node will be considered, because there is only one target vector for each data pattern. The supervised node must also be a *supersource.*

Moreover, the considered DPAGs are also bounded. This means that for all vertices the number of outgoing edges is limited to a certain number m.

Definition 5.5 (Bounded DPAG). *A graph $G(V, E)$ is an m-bounded DPAG if $G(V, E)$ is a DPAG and $\#out(v) \leq m$ for all vertices $v \in V$.*

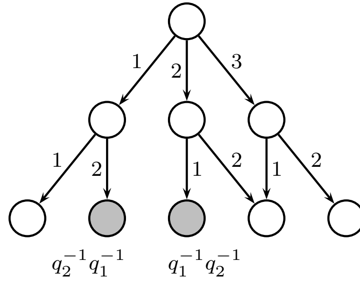

Figure 5.5: Example of the generalized shift operator on a DPAG. The gray nodes are the nodes given by the shift operators below them when applied to the root node of the graph. Note that the order of the shift operators is important.

5.2.2 Generalized Shift Operator

The shift operator for sequences, discussed in Section 5.1.2, can be generalized to an operator for DPAGs. The following example illustrates how this works.

Example 5.3. *If a binary tree structure is considered,* q_1^{-1} *would refer to the left child of a node and* q_2^{-1} *would refer to the right child. Combinations are also possible,* $q_1^{-1}q_2^{-1}$ *refers to the left child of the right child for instance. See Figure 5.5.*

The generalized shift operator is not defined on a sequence anymore, but on a graph. This means that the variable X to which the operator is applied, does not have a sequence of values but a value for each vertex of the graph. So let x_v be the value of the variable X for vertex v of the graph. Then the generalized shift operator can be defined as follows:

Definition 5.6 (Generalized Shift Operator). *If* x_{v_1} *is the value of a variable* X *for vertex* v_1 *of a graph* G *with vertices* V *and edges* E*, then the generalized shift operator* q_k^{-1} *gives the value of* X *for the* k*-th outgoing vertex as a result, so* $q_k^{-1}x_{v_1} = x_{v_2}$ *with* $(v_1, v_2) \in E$ *and* $pos(v_1, v_2) = k$*.*

5.2.3 Encoding Network

Using the generalized shift operator, recursive networks can be defined that are very similar to recurrent networks. In recurrent networks, all recurrent connections represented the q^{-1} shift operator. In recursive networks, all recursive connections will represent a generalized shift operator.

Definition 5.7 (Recursive Network). *A recursive network is a feedforward network extended with a number of recursive connections that introduce cycles in the network and are all labeled with a* q_k^{-1} *generalized shift operator.*

Example 5.4. *An example of a recursive network is shown in Figure 5.6, in the top left corner.*

A recursive network can be unfolded on a graph just as a recurrent network was unfolded on a sequence. The network, without the recursive connections, is copied for all nodes in the graph. Then the recursive connections are replaced by feedforward connections between the different copies. Suppose there is a recursive connection from neuron n_1 to neuron n_2, labeled with a generalized shift operator q_k^{-1}. Then, for a DPAG $G(V, E)$ each vertex $v_1 \in V$ with a vertex $v_2 \in out(v_1)$ for which $pos(v_1, v_2) = k$, results in a feedforward connection between the neuron n_1 in the copy for vertex v_2 and neuron n_2 in the copy for vertex v_1. As a result, the graph structure is mirrored in the unfolded network in a way defined by the recursive network. The unfolded network is also called *encoding network*.

Example 5.5. *An example of a recursive network and its corresponding encoding network on a given input graph is shown in Figure 5.6. To keep the example simple, there is only one hidden unit and all recursive connections are therefore from that unit to itself, but more complicated settings with recurrent connections between different units are perfectly possible.*

It should be noted that the encoding network is again a feedforward network. Weights are also shared between the different copies of the same original connections, in the same way as for the recurrent networks. This means that the same backpropagation through time training algorithm can be applied as before.

5.3 Graph Neural Networks

The third approach discussed here, is also the most general. It is called graph neural networks (GNNs) and is explained in depth in [45]. It can be seen as a further extension of recursive networks. The key ideas behind it will be discussed in this section.

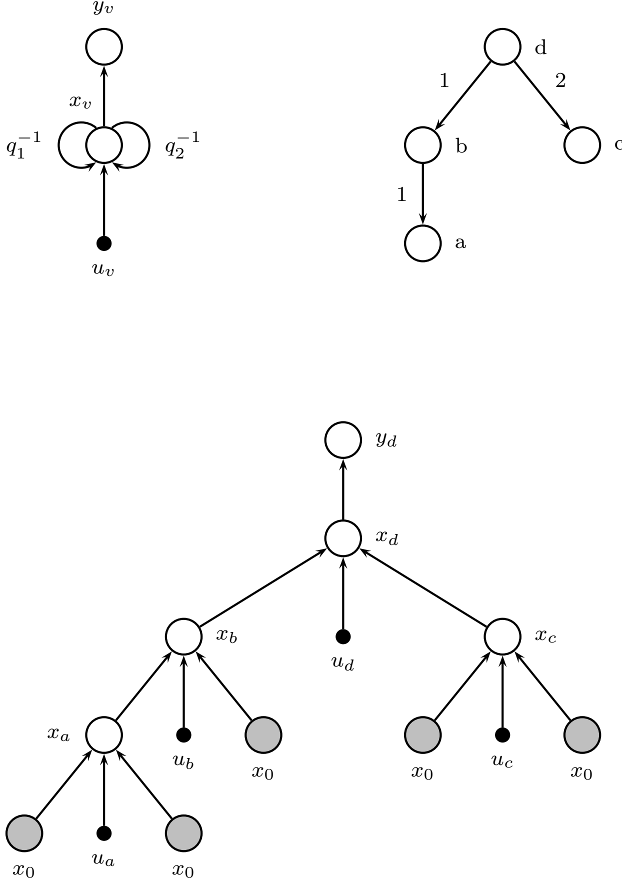

Figure 5.6: Example of network encoding for a 2-bounded DPAG. In the top left corner, the recursive network is shown, in the right corner the input graph and at the bottom the corresponding encoding network. The gray nodes in the encoding network are nodes with a default value x_0 for positions where there is no child present.

5.3.1 Encoding Network

For graph neural networks (GNNs), the basic approach is not that different from recursive networks. But some of the restrictions of recursive networks will no longer hold. The goal is to create an extension and generalization of the recursive networks of the previous section, just as recursive networks were an extension and generalization of the recurrent networks.

First of all, the input graphs do no longer have to be acyclic. This means that, if the procedure for creating the encoding network is the same as for the recursive networks, then the encoding network will contain cycles for some input graphs. This raises some problems for computing the activation values and gradient information, which will be addressed in the next sections. But essentially, it does not change the procedure of unfolding a recursive network structure on the input graph.

The graphs do no longer have to be directed either. Directed as well as undirected edges can be used in the graph. Typically, in GNNs all edges of the input graph are replaced by directed edges going in both directions. Extra attributes can be added to the edges to indicate the type of edge, although this will be ignored here for the sake of simplicity.

Definition 5.8 (Bounded Positional Graph). *A graph $G(V, E)$ is an m-bounded positional graph if $\#out(v_1) \leq m$ for all vertices $v_1 \in V$ and the vertices $v_2 \in out(v_1)$ are ordered.*

Example 5.6. *Figure 5.7 shows an example of how the encoding network is created for a GNN applied to a bounded positional graph containing cycles.*

A last extension is that it is also possible now that the graphs are not positional. In this case, the generalized shift operator q_k^{-1} is not needed anymore because referring to the k-th child does no longer make sense. The simpler shift operator q^{-1} will be used again, this time referring to all children together. Because the order among the outgoing vertices is no longer important, contributions from the different children can simply be summed up and used as one value. When the contributions are summed up, the upper bound m on the number of outgoing edges is also no longer relevant and so these graphs do no longer have to be bounded.

Example 5.7. *Figure 5.8 shows an example of the encoding network for a non-positional graph.*

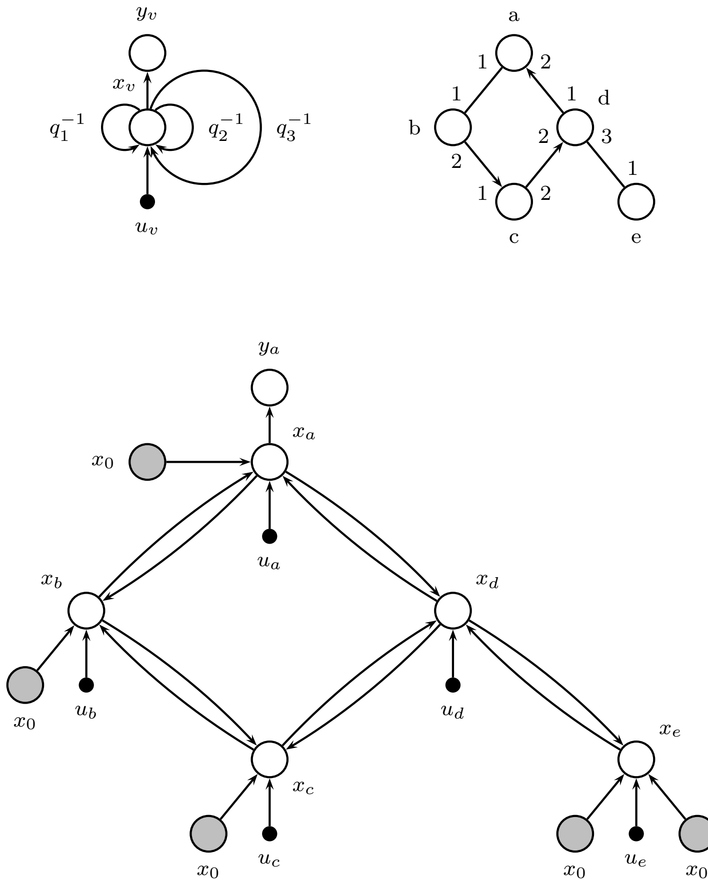

Figure 5.7: Example of an encoding network when a recursive network structure is applied to a 3-bounded positional graph to create a GNN. The recursive network structure is shown at the top left corner, the input graph at the top right corner and the encoding network at the bottom. Note that all connections in the GNN go in both directions, whether the corresponding edge in the input graph is unidirectional or bidirectional.

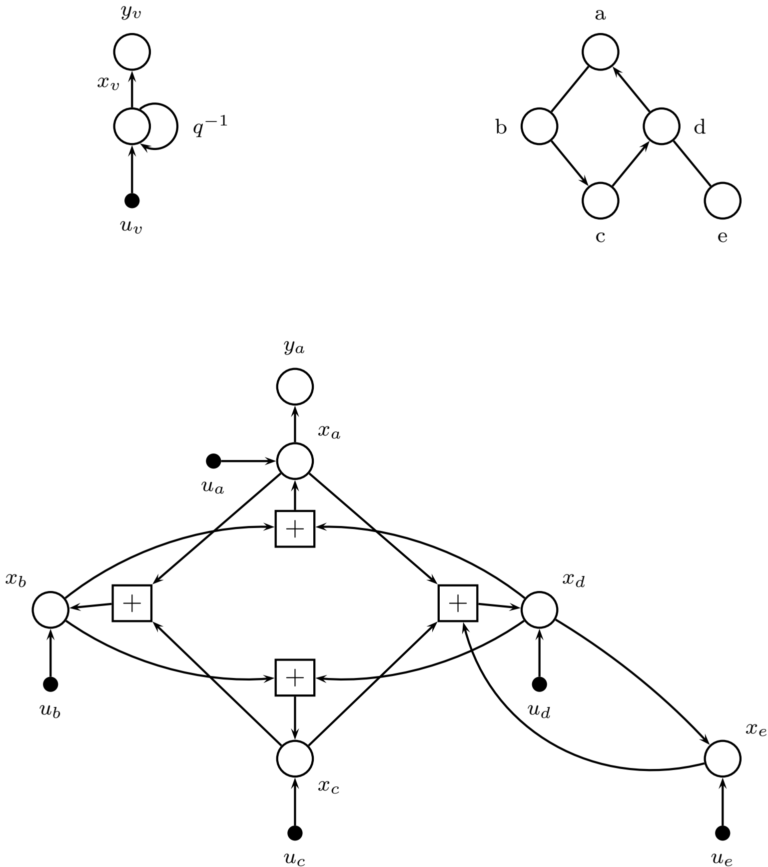

Figure 5.8: Example of an encoding network when a recursive network structure is applied to a non-positional graph to create a GNN. The recursive network is shown at the top left corner, the input graph at the top right corner and the encoding network at the bottom.

5.3.2 Contraction Mappings

The introduction of cycles in the encoding network creates a problem. It implies that there is no topological order any more between the neurons in the encoding network, and so the updating sequence of the neurons can no longer be defined as before. The activation values of some neurons will mutually depend on each other, which means that it is impossible to calculate these values in a serial order.

An elegant solution to this problem is to compute the activation values not in one step, but iteratively in many updating steps. The key idea is that the activation values will gradually evolve to stable values, regardless of the order in which the activation values are computed in each step. However, this only works if there is indeed convergence, which is not automatically guaranteed. Mathematically, it means that the function represented by the network needs to be a contraction mapping.

Definition 5.9 (Contraction Mapping). *A function $\phi : \mathcal{M} \mapsto \mathcal{M}$ on a metric space $\mathcal{M}$ is a contraction mapping with regard to a vector norm $\|\cdot\|$, if there exists a real value $0 \leq \eta < 1$ such that $\|\phi(x) - \phi(y)\| \leq \eta\|x - y\|$ for any $x, y \in \mathbb{R}^n$.*

Functions that are contraction mappings have interesting properties. They have a stable state and they are even guaranteed to converge to that state when the function is applied iteratively.

Theorem 5.1 (Banach Fixed Point Theorem). *If ϕ is a contraction mapping on a complete metric space, then the equation $x = \phi(x)$ has one and only one solution x^*. Moreover, for any initial state x_0, the sequence $x_k = \phi(x_{k-1})$ converges exponentially to x^*, i.e., $\|x^* - x_k\| \leq \eta^k\|x^* - x_0\|$ for any $k \geq 0$, where η is the contraction constant.*

This means that if the transition function F_w implemented by the encoding network is a contraction mapping, then the states can simply be computed iteratively by updating them until they have reached stable values.

5.3.3 Training Algorithm

As long as the network implementing the function F_w is a contraction mapping, convergence will be guaranteed for any initial state. Whether F_w is a contraction mapping depends of course on the choice of weights w. The weights are determined by the training algorithm, so it will be the responsibility of the

training algorithm to ensure that F_w is a contraction mapping. There exists a condition that guarantees that a function is a contraction mapping.

Theorem 5.2 (Contraction Mapping Condition). *If ϕ is a function in $C^1(\mathbb{R}^n, \mathbb{R}^n)$, $\|\cdot\|_v$ is a vector norm and $\|\cdot\|_m$ is the corresponding matrix norm*

$$\|M\|_m = max_{x \in \mathbb{R}^n} \frac{\|Mx\|_v}{\|x\|_v}, \tag{5.1}$$

then ϕ is a contraction mapping with regard to $\|\cdot\|_v$ if and only if there exists some $0 \leq \eta < 1$ such that

$$\left\| \frac{\partial \phi}{\partial x}(x) \right\|_m \leq \eta \tag{5.2}$$

with $\frac{\partial \phi}{\partial x}$ the Jacobian matrix of ϕ.

A typical way to enforce an additional constraint on a neural network, is to add a penalty term to the error expression that is minimized during training. In this case, the extra penalty term has to ensure that the norm of the Jacobian is kept smaller than 1. The formula for the error becomes:

$$e = \sum_{i=1}^{n} (t_i - y_i)^2 + \beta L \left(\left\| \frac{\partial F_w}{\partial x} \right\| \right) \tag{5.3}$$

The first part of this expression is the sum of all squared errors between output and target values, which is typically minimized when training a network, the second part is an added penalty term to keep the Jacobian small. The parameter β defines the balance between the error on the patterns and the penalty term, $L(y)$ equals $(y-\mu)^2$ if $y > \mu$ and 0 otherwise, and the parameter μ defines the desired contraction constant of F_w.

For computing the gradient information, backpropagation through time is combined with the Almeida-Pineda training algorithm [1, 36]. This latter algorithm relies on the settling behavior of recurrent networks, which implies that computation of the activation and δ values can be carried out in an iterative fashion, until convergence to stable values occurs. In principle, because the activation values of the transition network are computed in multiple time steps, the total encoding network should be unfolded in time to compute the gradient with backpropagation through time. But since the system has reached a stable point $x^* \approx x_T \approx x_{T-1}$, it can be assumed that the state of the units remains more or less constant for each time step and backpropagation through time can be carried out using only the last, stable state values. This saves memory and makes the computation more efficient.

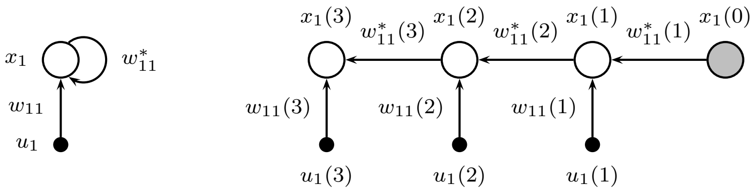

Figure 5.9: Example network to illustrate the vanishing gradient problem. At the left, a very simple recurrent network is shown, at the right the unfolded network for an input sequence of three inputs. The node $x_1(0)$ is just the initial value for the recurrent unit.

5.4 Problems

There are also some general problems associated with these neural networks for structural data. Two limitations will be discussed here. The first one is the vanishing gradient problem, which is a problem concerning the training of the networks with gradient descent. The second limitation is about what kind of functions can be approximated by this kind of networks.

5.4.1 Vanishing Gradient Problem

Although recurrent networks are in principle able to learn from sequences, there is a problem that occurs during training these networks that can make it difficult to learn the right model. This is usually called the *vanishing gradient* problem, and it has been explained for instance in [2]. It does not only apply to recurrent nets, but also to networks on graph structures because it is related to the unfolding procedure. In this section, the basic problem will be explained.

With standard backpropagation, the weight update Δw_{ij} between unit i and j is computed as $\Delta w_{ij} = \delta_j x_i$. The δ value is the gradient information computed for each network unit, according to the following formula:

$$\delta_j = \begin{cases} \sigma'(net_{y_j})(t_j - y_j) & \text{if } j \text{ is an output unit} \\ \sigma'(net_{x_j}) \sum_{k=1}^{M} \delta_{y_k} w_{jk} & \text{if } j \text{ is a hidden unit} \end{cases} \tag{5.4}$$

It is illustrative to look at how this calculation works in practice when applied to a very simple recurrent network. The considered network is shown in Figure 5.9, it consists of one input and only one recurrent unit. Suppose that this network is unfolded for a sequence of three inputs. This implies that the recurrent connection is converted into three feedforward connections in the unfolded network. The different copies of a unit x_i will be denoted as $x_i(j)$, with j indicating the fold to which this unit belongs, the weights will be indexed in the same way. Now the weight updates can be computed for the unfolded recurrent connections:

$$\Delta w^*_{11}(3) = (t - x_1(3))\, \sigma'(net_{x_1}(3))\, x_1(2) \tag{5.5}$$

$$\Delta w^*_{11}(2) = (t - x_1(3))\, \sigma'(net_{x_1}(3))\, \sigma'(net_{x_1}(2))\, w^*_{11}(3)\, x_1(1) \tag{5.6}$$

$$\Delta w^*_{11}(1) = (t - x_1(3))\, \sigma'(net_{x_1}(3))\, \sigma'(net_{x_1}(2))\, \sigma'(net_{x_1}(1))\, w^*_{11}(3)\, w^*_{11}(2)\, x_1(0) \tag{5.7}$$

If the value for $x_1(0)$, the initial state value, is zero, then $\Delta w^*_{11}(1) = 0$. With n the number of folds, these weight updates can be written as:

$$\Delta w^*_{11}(j) = (t - x_1(3))\, x_1(j-1)\, \sigma'(net_{x_1}(n)) \prod_{k=j}^{n-1} \sigma'(net_{x_1}(k))\, w^*_{11}(k+1) \tag{5.8}$$

The problem here is the product term. If $\rho_k = |\sigma'(net_{x_1}(k))\, w^*_{11}(k+1)| < 1$, then this product term will decrease exponentially with respect to n. So the weight updates for the folds further away from the final output unit, will tend to go to zero. Giving large initial values to the weights does not help to avoid this problem because the σ' function goes to zero for large values, neutralizing the effect of larger weights. Increasing the learning rate does not help either, because it does not change the ratio between the errors for the first and the last fold. So for long sequences, the more recent inputs will have a substantial impact on the weight update, while the influence of the older inputs will be negligible.

The same effect applies to recursive and graph networks. When learning from tree structures for instance, it will only be possible to train the network effectively on trees that are not too deep. Backpropagation of the error signal only succeeds for the top of the tree, for the leaves that are far away at the bottom of the tree, the error signal will disappear as well.

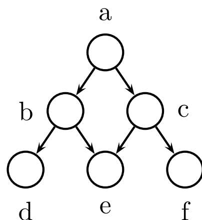

Figure 5.10: An example graph for which it can be a problem to learn the right function. This is for instance the case if the target is the number of vertices in the whole graph.

5.4.2 Approximation Capabilities

An interesting question is what models can be learned with these networks. For recursive networks, it has been proven that they are able to approximate any function on trees up to any degree of precision [20]. The problem is that this result only holds for tree structures. For input graphs that are not trees, it can not always be guaranteed that the desired model can be learned correctly. The following example illustrates this.

Example 5.8. *Suppose that the input graphs are acyclic directed graphs, but not trees, like the example graph shown in Figure 5.10. The target is the total number of vertices in the graph. For the example graph, this is 6. When the unfolded network is created, all vertices are replaced by a network implementing the function f_w. So the question is whether there is any such function that can compute the desired result. It could for instance be a function that adds up the numbers from the children of that node and gives that sum plus one as a result. But in this case the problem is that vertex e will be counted twice. It will be included in the result both at vertex b and c and so the final result will be 7 instead of 6. Unfortunately, there is no way to avoid this at vertex b and c because there is no information available that indicates that vertex e is already counted at another vertex. These kind of problems do not occur with tree structures.*

To discuss this problem more formally, some concepts need to be defined. First of all, the *unfolding tree* of a graph $G(V, E)$, denoted as T_v^d where d is the depth of the tree and v the root vertex.

Definition 5.10 (Unfolding Tree). *An unfolding tree T_v^d of a graph $G(V,E)$ is a tree starting from vertex v and with depth d, defined recursively as*

$$T_v^d = \begin{cases} tree(l_v, \emptyset) & \text{if } d = 1 \\ tree(l_v, \{T_{v'}^{d-1} : (v,v') \in E\}) & \text{if } d > 1 \end{cases} \tag{5.9}$$

with $tree(v,S)$ an operator that creates a tree from a given vertex v and a set S of subtrees and l_v the vector attached to vertex v.

One should take care here whether positional or non-positional graphs are considered. For positional graphs, the position of the subtrees will have to be the same as the position of the vertices for which they were constructed. So in this case, two unfolding trees are only equal if they have the same vector and subtrees, and these subtrees also have the same positions. For non-positional graphs, the positions of the subtrees are not important and they are not taken into account for the equality between trees. In a second step, the definition of unfolding trees leads to an equivalence relationship on vertices.

Definition 5.11 (Unfolding Equivalence). *Two vertices v_1 and v_2 of a graph $G(V,E)$ are unfolding equivalent, denoted as $v_1 \sim v_2$, if and only if $T_{v_1}^d = T_{v_2}^d$ for all $d \geq 1$.*

A third definition is about preserving unfolding equivalence. Functions that have this property will compute the same output values for all equivalent input graphs.

Definition 5.12 (Unfolding Equivalence Preservation). *A function $f : \mathcal{G} \mapsto \mathbb{R}^n$, with $\mathcal{G}$ the domain of input graphs, preserves the unfolding equivalence on $\mathcal{G}$ if $v_1 \sim v_2$ implies that $f(G_{v_1}) = f(G_{v_2})$ for all vertices v_1 and v_2 from an input graph $G(V,E)$, where G_{v_1} is the graph G with vertex v_1 as root and G_{v_2} the same graph G but now with vertex v_2 as root.*

Considering the set of all functions that preserve the unfolding equivalence on $\mathcal{G}$, denoted as $\mathcal{F}(\mathcal{G})$, it turns out that this is exactly the set of functions that can be learned by GNNs.

Theorem 5.3 (Approximation Capability of GNNs). *For any function $f : \mathcal{G} \mapsto \mathbb{R}^n$ which preserves the unfolding equivalence and is measurable on $\mathcal{G}$, any probability measure P on $\mathcal{G}$ and any $\epsilon > 0$, there exists a GNN implementing a function ϕ so that $P(\|f(G) - \phi(G)\| \geq \epsilon) \leq \epsilon$.*

A proof of this theorem is not given here but can be found in [44].

5.5 Conclusions

The explanation of recurrent, recursive and graph networks clearly showed how neural networks can be used on graph domains. The networks were gradually extended with regard to the input domain, starting from linear graphs over acyclic graphs to cyclic graphs. The key idea in the first transition was to generalize the recurrent connection of the recurrent networks. The transition from acyclic to cyclic graphs could be made by using contraction mappings.

The vanishing gradient problem, which possibly occurs for all these networks on structured domains, makes it difficult to learn on large graphs where relevant information is at large distances. This is why methods like recursive or graph networks must be seen as *local* learning methods. They can learn a function that relies on information close to the supervised node of the graph, but not on information that is too far away from the supervised node. However, for many problems, the assumption that information nearby is most important, is reasonable. There is also a limit on what models can be learned by these methods. Not all structural information can be taken into account.

The discussion of these existing neural network approaches for dealing with structured data serves as an introduction and background for the next chapter, which presents a novel approach for dealing with similar relational data. This new approach resembles the discussed methods but will be more focused on the aggregation aspect.

The overview given in this chapter is not exhaustive. There are for instance contextual cascade-correlation networks [33], an extension of cascade-correlation networks to structured domains, and neural networks for graphs (NN4G) [32]. There are also many different kernels for graphs [17, 16]. Because they typically focus on structural features only, being based on substructure matching for instance, they have less in common with the methods proposed in this dissertation.

Chapter 6

Relational Neural Networks

In the previous chapter, existing approaches for learning from structured data with neural networks were discussed. In this chapter, a neural network method for learning from relational data is proposed. This method resembles recursive networks, but it includes the possibility of multiple types and it focuses on aggregation, something that was never considered on itself in recursive networks. A number of experiments to determine the capabilities of this new type of networks are discussed as well.

6.1 Relational Neural Networks

The new type of networks introduced in this chapter are relational neural networks (RelNNs). First, the kind of relational data for which these networks are developed, is discussed, then the encoding network is explained and how these networks work with regard to input feeding and training.

6.1.1 Relational Data

The goal is to create neural networks that can deal with relational input data. Relational data can be represented as graphs, but the graphs will be heterogeneous, meaning that there are different types of nodes or edges. Each type can have different attributes. This case was not considered in the networks for structured domains discussed in the previous chapter. The data type of the

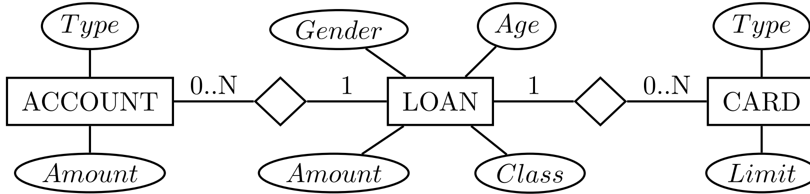

Figure 6.1: ER schema for a simple relational dataset. The entities are loan, account and card. For each entity, the attributes are shown: gender, age, amount and class for loan, type and amount for account, and type and limit for card. Between loan and account, a relationship is defined with exactly one loan for each account and zero, one or more accounts for each loan. A similar relationship is defined between loan and card.

input structures can be described using an entity-relationship (ER) diagram, as is common in the field of relational databases.

Example 6.1. *Reconsider example 3.7, in which a relational dataset was described with loans which had to be classified, and accounts and cards tuples attached to each loan. The ER schema for this dataset is given in Figure 6.1.*

In general, the ER schema consists of the following elements:

- The *entities* are depicted as rectangular nodes in the schema.
- The *attributes* are attached to the entity with oval nodes.
- Between the entities, *relationships* can be defined. These are shown as diamond shaped nodes between two entities.
- For each relationship, the *connectivity* is shown on the edges with the entities. These connectivities indicate how many instances of one entity can be connected to an instance of the other entity.

Now each input graph must follow the structure of this ER schema:

- Each node of the graph is an instance of one entity from the ER schema. As a result, the node has a tuple attached to it with attributes that correspond to the attributes associated with that specific entity in the ER schema.

- Each edge in the graph is an instance of one of the relationships given in the ER schema. This means that it must be an edge between nodes having the right type, as described by the ER schema. The connectivity constraints of the relationship also have to be obeyed.

6.1.2 Encoding Network

The ER schema of the input graphs is used for the construction of the encoding network. This encoding network is built as a tree structure, in a top-down fashion. The root of the tree is a feedforward network with inputs for the attributes of the target entity. The target or root entity is the entity that has the target attribute attached to it. After that, relationships in which this root entity participates are explored, adding network structures for the related entities to the tree. How this is done, depends on the kind of relationship under consideration. All possibilities are discussed below:

- The situation for the root node is depicted in Figure 6.2. In this example, a feedforward network is created for the two input attributes U_1 and U_2 of the root entity. The target attribute T is used at the output of this network to train it. This situation is the starting point.
- Another entity B could be related to the root entity A through a strict one-to-one or many-to-one relationship. This means that there is exactly one B tuple connected to each A tuple. In the other direction, one B tuple could be connected to more than one A tuple though. In this case, the solution is very simple. The inputs for the attributes of the related B tuple are just added to the feedforward network. See Figure 6.3.
- A very similar situation occurs when there is a non-strict one-to-one or many-to-one relationship, where there could be *at most* one tuple related to the root tuple. So in this case, there can be zero or one tuple connected to the root tuple through the considered relationship. The root network is now expanded with inputs for the attributes of the related entity B, as before, but also with an extra dummy attribute to indicate whether the related tuple is present or not. Figure 6.4 shows this.
- The last situation involves a one-to-many or many-to-many relationship between root entity A and another entity B. This situation is obviously more complicated because it is possible that there are multiple tuples related to the root tuple through the same relationship. To deal with this, networks with the capability to process bags of tuples, like the network structures from Chapter 4, need to be used. This situation is depicted in Figure 6.5, in this example with a locally recurrent network.

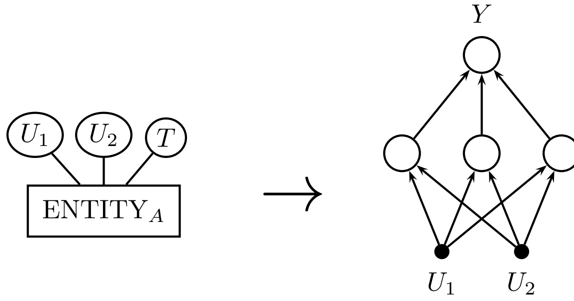

Figure 6.2: Example of an encoding network for the root entity alone. The root entity A is shown at the left with two input attributes, U_1 and U_2, and a target attribute T. The encoded network is shown at the right. It is a feedforward network with two inputs and one output.

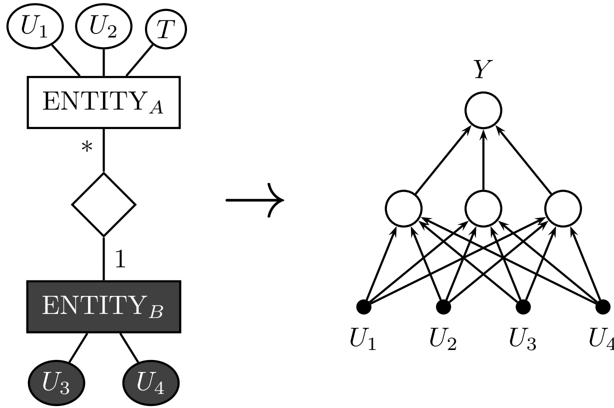

Figure 6.3: Example of an encoding network for a strict one-to-one or many-to-one relationship. The root entity A has a related entity B with two attributes, U_3 and U_4. They are added to the network for A as extra inputs.

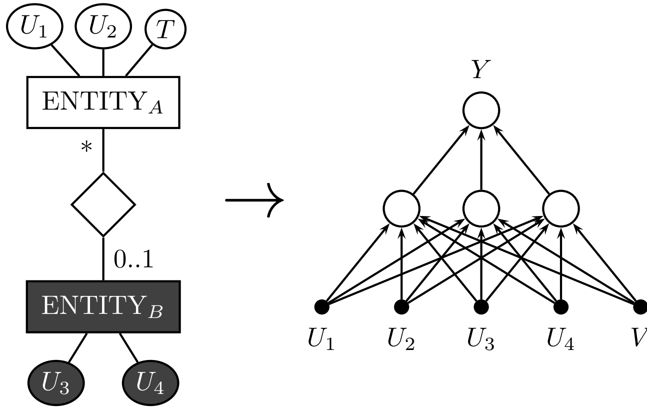

Figure 6.4: Example of an encoding network for a non-strict one-to-one or many-to-one relationship. The two attributes U_3 and U_4 of entity B are added to the network for A as extra inputs, plus an extra input V to indicate whether a B tuple is present or not.

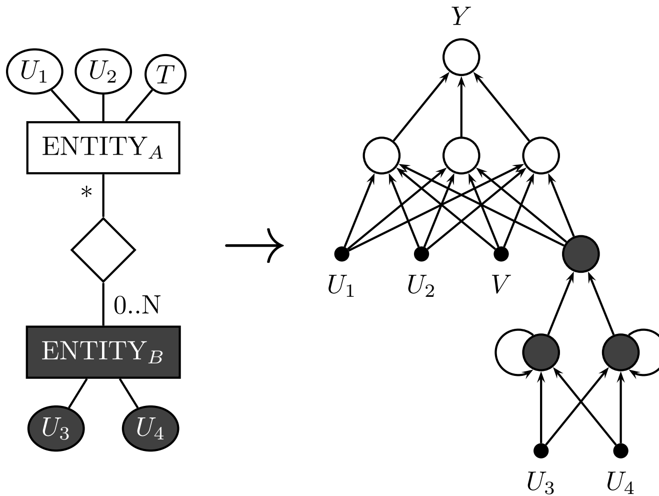

Figure 6.5: Example of an encoding network for a one-to-many or many-to-many relationship. To process the bag of B tuples related to the A tuple, a recurrent network is added to the network for entity A, plus an extra input V to indicate whether at least one B tuple is present or not.

This process can of course be repeated for other entities than the root entity. In practice, a breadth-first encoding is conducted, starting from the root entity, down to a given depth d. The encoding network is built step by step by adding extra elements to the network for each new related entity. This results in Algorithm 6.1.

Algorithm 6.1 The algorithm for constructing the encoding network for relational neural networks. Construction happens top down. X is the root entity in the ER schema from which construction starts, Relationships is the set of all relationships in the ER schema, R(A,B) is the relationship R between entities A and B in the ER schema and #attr(A) is the number of attributes of an entity A.

```
Root = new feedforward network with #attr(X) inputs and label X
Nodes = [Root]
for i = 1 to d do
  NewNodes = []
  for each N in Nodes do
    let A be the label of N
    for each R(A, B) in Relationships do
      if R is strict one-to-one or many-to-one then
        add #attr(B) inputs to N
        add N labeled with B to NewNodes
      else if R is non-strict one-to-one or many-to-one then
        add #attr(B) + 1 inputs to N
        add N labeled with B to NewNodes
      else if R is one-to-many or many-to-many then
        G = new aggregate network with #attr(B) inputs and labeled B
        add an extra input to N
        add G to NewNodes
      end if
    end for
  end for
  Nodes = NewNodes
end for
```

Example 6.2. *In Figures 6.6 and 6.7, an ER schema and its corresponding encoding networks for depths 1, 2 and 3 is shown, according to Algorithm 6.1.*

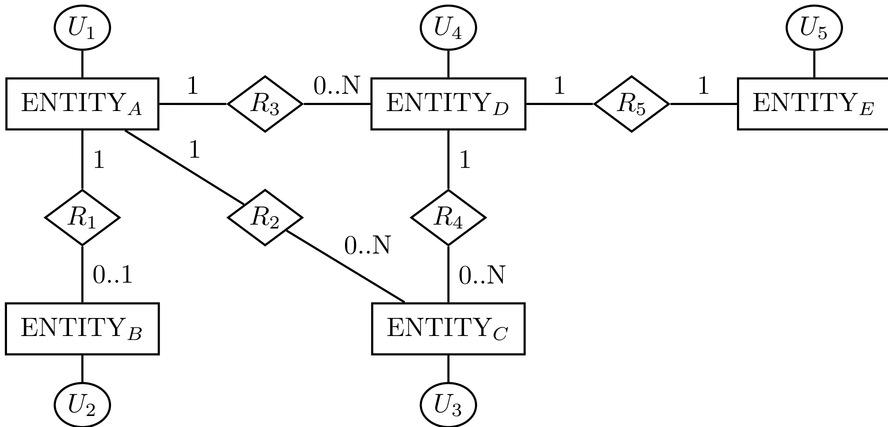

Figure 6.6: The ER schema that serves as a basis for the example encoding networks of Figure 6.7.

6.1.3 Feeding and Training

As the encoding network is a tree based on the ER schema, the input graphs will also have to be transformed into tree structures, similar to the encoding network.

Example 6.3. *An example of how an input graph is transformed in the right input structure, is shown in Figure 6.9. The input graph is shown in Figure 6.8, the corresponding ER schema in Figure 6.6.*

Feeding the input structures into the encoding network happens in a bottom-up way. The bags at the leaves of the input structure are fed into the lowest parts of the encoding network. The state values of these lower network parts are then combined with the inputs from the bags in the nodes one level higher in the tree structure, and so on towards the top of the network. Finally, the output values of the network are computed. This procedure is very similar to the unfolding of the network in the case of recurrent or recursive networks. In this case, the encoding network is unfolded according to the input tree structure.

Training of the network happens in the opposite direction. As long as there are no cycles, standard backpropagation can be used. In this case, there is an order in which the units should be updated to propagate the signals from inputs to outputs. Going in the reverse direction, from outputs to inputs, makes it possible to backpropagate the error signal and compute the gradient. Since the encoding network for RelNNs is a tree and therefore acyclic, the unfolded

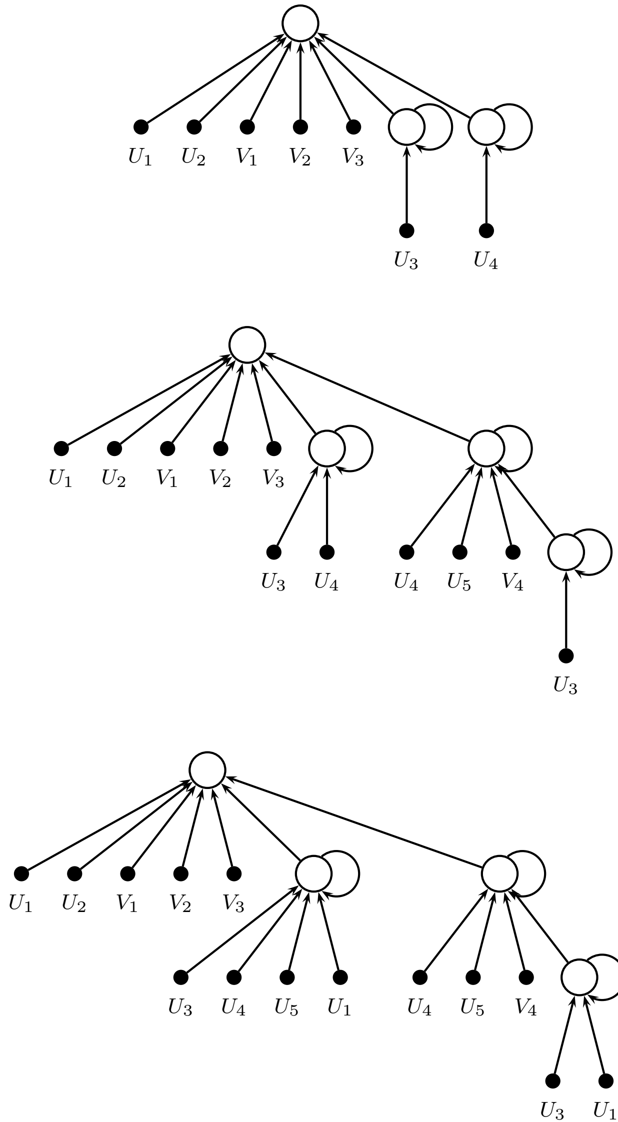

Figure 6.7: Example of how the ER schema of Figure 6.6 can be encoded into networks of different depths. At the top, the corresponding encoding network of depth 1 is shown, in the middle the encoding network of depth 2 and at the bottom the encoding network of depth 3.

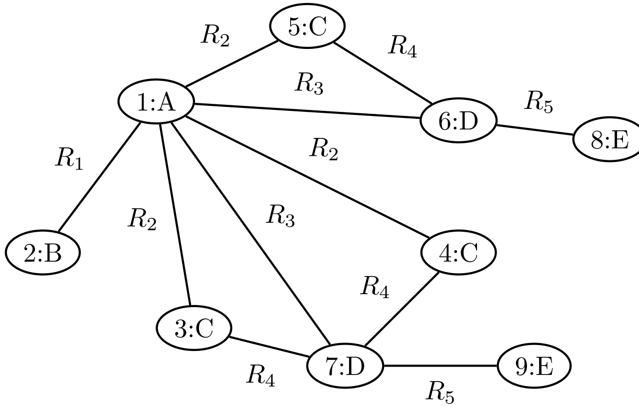

Figure 6.8: Example of an input graph according to the ER schema of Figure 6.6. The numbers in the nodes identify the tuples while the letters denote the types of the tuples.

encoding network turns out to be a large feedforward neural network. Applying backpropagation to this unfolded network results in a backpropagation through structure algorithm [46, 15], similar to the algorithm used for recursive neural networks. Since the parameters are shared among nodes of the same type, the corresponding gradients are accumulated.

6.1.4 Relational versus Recursive and Graph Networks

RelNNs look very similar to recursive networks because they use tree input structures, similar to the DAG input structures of recursive networks. There are two main differences however. First of all, the input trees can contain different types of nodes. Therefore, different networks are used for the nodes of the encoding network, according to the type of the node. In recursive networks, all nodes in the encoding network are assigned the same network structure. But more important is the second difference. In RelNNs, special aggregate networks are used to combine the contributions from the different child nodes. As explained in the previous section, the child nodes of the input graph are grouped in bags per type, depending on the relationship involved. Each of these bags is collapsed into a single state vector by the aggregate network. This can be done in different ways, as explained in Chapter 4. In recursive networks, this process of aggregating bags of child nodes is not considered as such. The

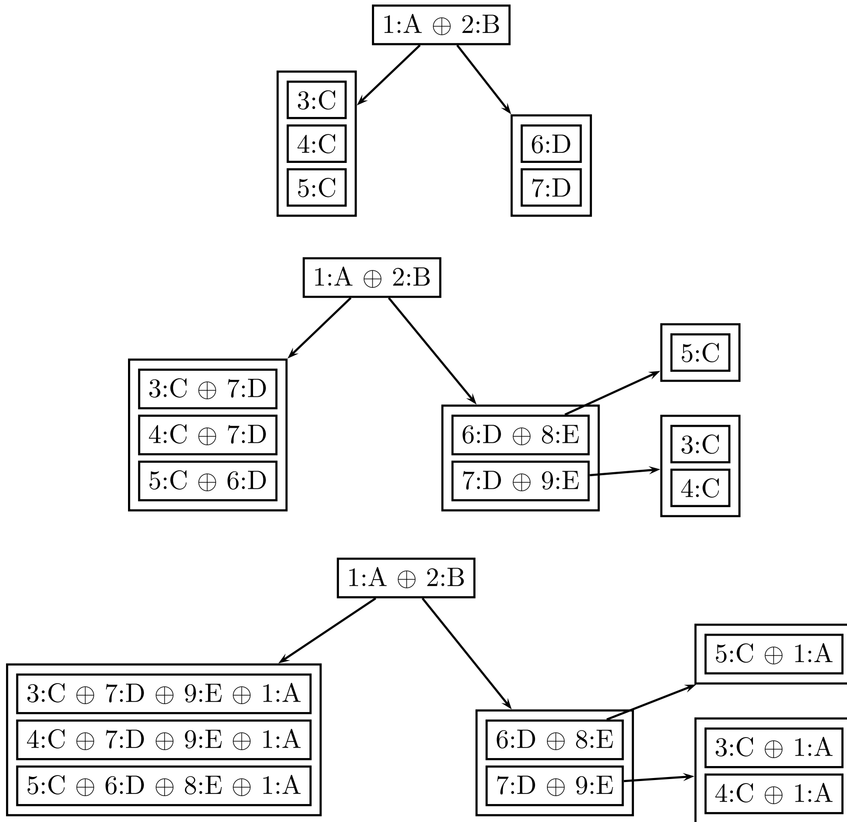

Figure 6.9: Example of a transformation of an input graph into the input pattern. The corresponding input patterns for depths 1, 2 and 3 are shown for the example given in Figure 6.8. The operator $\oplus$ indicates that the vectors of two tuples are concatenated.

cardinality of the bags is assumed to be small and recursive connections are used to process the state vectors for the children in the bag directly.

The most important difference between RelNNs and GNNs is that GNNs can process graphs with cycles directly while RelNNs cannot. But even though cyclic graphs cannot be elaborated directly by RelNNs, they can be transformed into trees by the preprocessing step. As shown, the procedure consists of unfolding the graphs into trees according to a breadth-first visit. The visit starts from the node for which the output is evaluated and may traverse the nodes several times in the case of cycles. The resulting tree is the one defined by the visited edges and nodes, and thus it may contain several copies of the original nodes. The depth of the tree may change according to the depth of the visit. It might look as if this is a limitation of the representational power of RelNNs that does not apply to GNNs. But, as explained in Section 5.4.2, GNNs have the same kind of problem even though they can handle cycles directly.

Like recursive networks, GNNs do not take into account the possibility of different types. Although in principle, it is possible to incorporate this in the GNNs, it has not been considered in the original GNN method. With regard to the aggregation of bags of child nodes, two possibilities were considered in GNNs. The first one is to use the same method as in the recursive networks. An alternative is to make the sum of all contributions of the child nodes. This is more similar to the aggregation methods used in RelNNs, where summing up contributions was also one of the possibilities, among others. In principle, it is also possible to use some of these other methods in GNNs, but it has not been considered so far.

6.2 Experiments

A first experiment is conducted on an artificial dataset to examine the ability of RelNNs to learn to recognize small substructures in relational data. Furthermore, experiments on two relational datasets are conducted to make a comparison between GNNs and RelNNs, and also to compare them with other machine learning methods. The considered datasets contain descriptions of molecular structures and a target property of these molecules which has to be predicted by the model. A variety of methods has already been applied to the datasets, so these problems can serve as some kind of benchmark.

6.2.1 Subtrees

A first, simple experiment is to test whether RelNNs are able to detect simple subtree structures in relational data. This is the counterpart of the simple aggregate datasets of Chapter 4. Artificial data is generated according to the ER schema of Figure 6.10. For each tuple in the input pattern, a vector containing five values is generated. Only the first value is important, the other four are generated randomly, from a uniform distribution in the interval $[-1, 1]$. The first value is also generated randomly in the interval $[-1, 1]$, but it is important whether the value falls in the $[-1, 0.75[$ or the $[0.75, 1]$ interval because this will determine the target class.

Datasets with different concepts are generated. For instance, if the concept is 'A', an input pattern is a positive example if and only if the first attribute value of the A tuple falls in the $[0.75, 1]$ interval. If the concept is 'A&B', a pattern is a positive example if and only if the first attribute for the A tuple is in $[0.75, 1]$ and there is at least one B tuple connected to the A tuple for which the first attribute also falls in that same interval. If the concept is 'A&B&C', then a pattern is positive if and only if the first attributes of the A tuple, at least one of the connected B tuples and at least one of the C tuples connected to that B tuple, are all in the $[0.75, 1]$ interval. An example of an input pattern is shown in Figure 6.11.

For these experiments, datasets for a range of different concepts were generated. Each dataset contains 3000 patterns. The datasets were generated in such a way that the different bags of B, C and D tuples contain between 5 and 10 tuples. They are also generated in such a way that the number of positive and negative examples are equal.

The experiments are conducted using ten-fold cross-validation. Seven folds are used for training, two for validation and one for testing. Three hidden units are used in the network, on all levels. Results for RelNNs using locally recurrent networks (LRC) and aggregate networks with maximum (MAX) and sum (SUM) units are shown in Table 6.1 and Figure 6.12. The LRC, MAX and SUM network structures correspond to the networks discussed in Section 4.1. From these results, it is clear that the more a concept depends on information far away from the root tuple, the more difficult it becomes to learn it. The concept 'A' is of course easy to learn, for 'B' it is still possible to achieve good accuracy, but concepts involving C or D are much harder. This confirms the hypothesis that this kind of method is only suitable for learning local concepts, depending only on a root tuple and information coming from its direct environment. Learning structural concepts spanning over very distant tuples in the structure is much harder.

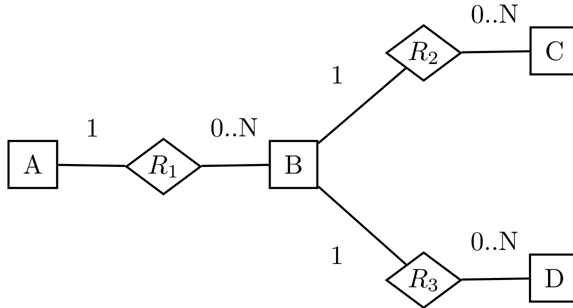

Figure 6.10: ER schema for the subtrees dataset. A, B, C and D are entities, R_1, R_2 and R_3 are relationships. The attributes have been omitted from the schema, but every entity has five attributes. Entity A also has an extra target attribute.

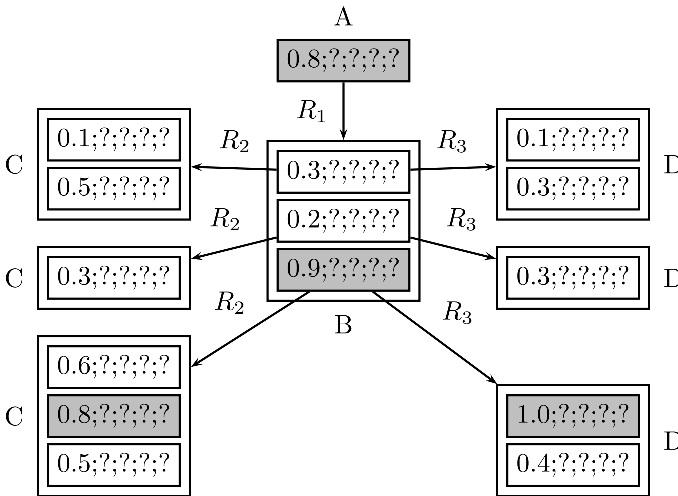

Figure 6.11: Example of an input pattern of the subtrees dataset. It is a positive example for the concept 'A&B&C&D' because of the subtree with first values greater than or equal to 0.75. This subtree consists of the gray nodes. The question marks stand for values that are not important.

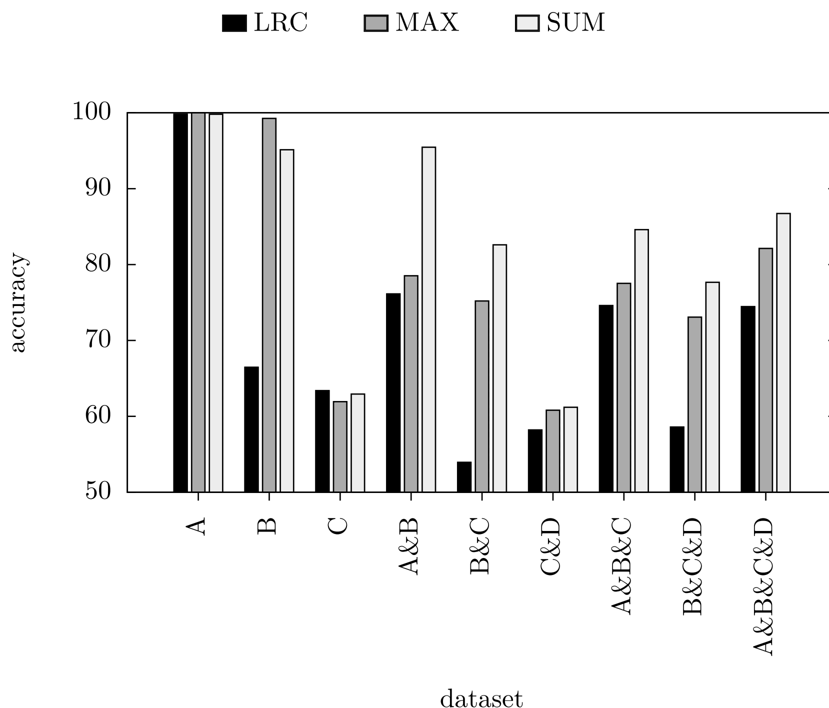

Figure 6.12: Accuracies on the subtrees dataset for RelNNs using LRC, MAX and SUM networks.

6.2.2 Mutagenesis

The mutagenesis dataset [10] is a small dataset, publicly available and often used as a benchmark in the ILP literature [31]. It contains the descriptions of 230 nitroaromatic compounds that are common intermediate subproducts of many industrial chemical reactions. The goal is to predict whether compounds are mutagenic on Salmonella typhimurium or not. In the original dataset, the value to be predicted was a real-valued measure of the mutagenicity of each compound. However, the problem considered in the literature is typically a classification problem where it has to be predicted whether a compound is mutagenic (mutagenicity is larger than one) or not (mutagenicity is smaller than one). In [10] it is shown that 188 molecules out of 230 are amenable to a regression analysis. This subset is called regression–friendly, while the remaining 42 compounds are named regression–unfriendly.

	LRC	**MAX**	**SUM**
A	99.80 [0.16]	100.00 [0.00]	99.80 [0.16]
B	66.47 [1.02]	99.27 [0.33]	95.13 [1.42]
C	63.40 [1.65]	61.93 [1.93]	62.93 [0.44]
A&B	76.13 [1.24]	78.53 [10.26]	95.47 [0.75]
B&C	53.93 [1.55]	75.20 [1.51]	82.60 [4.29]
C&D	58.20 [1.85]	60.80 [3.10]	61.20 [3.95]
A&B&C	74.60 [1.45]	77.53 [4.66]	84.60 [1.95]
B&C&D	58.60 [2.04]	73.07 [9.17]	77.67 [5.14]
A&B&C&D	74.47 [0.62]	82.13 [6.28]	86.73 [2.58]

Table 6.1: Accuracies on the subtrees datasets for RelNNs using LRC, MAX and SUM networks. Standard deviations on the results for the different folds are shown between brackets.

Despite the fact that the dataset is rather small, it has been used intensively to evaluate statistical and relational learning techniques. For historical reasons, many authors have reported their results only on the regression–friendly part, that is often referred to as the mutagenesis dataset. Moreover, the comparison is complicated by the fact that there are different possibilities with regard to the knowledge base, i.e., the set of attributes used for building the model.

The ER schema of the dataset is given in Figure 6.13. Each compound has four global attributes [10]. Two of these attributes are chemical measurements (denoted as C), namely LUMO, the lowest unoccupied molecule orbital[1], and log P, the water/octanol partition coefficient[2]. The other two attributes are pre-coded structural attributes (denoted as PS). Besides these global attributes of the compounds, the dataset also contains information about the atoms and bonds in each compound (denoted as AB). There are two relationships, one between molecules and atoms to represent that an atom is part of a compound and one between different atoms to represent a bond between two atoms. For the atoms, the type and the charge of the atom are given. For the bonds, the type of bond is given. Finally, attributes indicating the presence of functional groups, e.g., methyl groups, have been used in some papers as higher level attributes (denoted as FG). However, these functional groups will not be used in the experiments conducted here. For these experiments, discrete values are

[1] The LUMO of a molecule is the lowest energy orbital that has room to accept electrons.

[2] Octanol is a fatty alcohol with eight carbon atoms that is immiscible with water. Water/octanol partitioning, measured in logarithmic scale, is a relatively good approximation of the partitioning between the cytosol and lipid membranes of living systems.

Figure 6.13: The ER schema for the mutagenesis dataset.

represented by a one–hot encoding of the different possible values. All other attributes are represented by the corresponding real values, scaled to the $[-1, 1]$ interval.

The experiments are conducted with relational neural networks as well as graph neural networks, which makes it possible to compare both approaches. The results are also compared with previously reported results on this dataset. The experimental settings for RelNNs and GNNs are the same as much as possible, but in some respects the two methods require different setups. The most important difference exists with regard to the encoding of the information:

- **RelNNs:** different types of nodes can be represented in RelNNs. Each compound is first of all represented by a top node that stands for the compound as a whole and contains the global attributes. Besides this top node, a different type of nodes is used to represent the atoms belonging to the compound. These atom nodes contain the type and charge of the atom. They can be further connected to other nodes representing the atoms that have a bond with the considered atom. Because RelNNs cannot process cycles in the relational information, the atom–bond structure is unfolded down to a certain depth, as explained in Section 6.1.2. Depths 0, 1 and 2 will be considered for the experiments. The target value is associated with the compound node.

- **GNNs:** with the current implementation of GNNs, it is not possible to have different types in the encoding network. Therefore, the compound should be represented as a graph with a single type of nodes. The nodes in the graph are chosen to represent the atoms of the compound. The global attributes of the compound are then added to each atom tuple for all atoms in the compound. So the global compound is not represented as a separate node in the graph, but its information is added to all the

nodes in the input graph. The supervised node is just the first node of each compound in the dataset.

RelNNs and GNNs also differ with regard to the types of neural networks used for the different parts of the encoding network. This results in further differences between both methods concerning the internal structure of the network nodes after the encoding step:

- **RelNNs:** the top node for the global compound is a feedforward network. The nodes for processing the atom information are locally or fully recurrent networks or aggregate networks, like the simple network structures from Chapter 4. All networks use sigmoid activation functions.
- **GNNs:** two different functions need to be implemented, i.e., the transition and the output function. The output function is a standard feedforward neural networks. The transition function is a feedforward network for which the output values are accumulated for the different contributing nodes. Both networks have one hidden layer. Hyperbolic tangent activation functions are used for the hidden units and linear activation functions for the output units.

Following the most common experimental procedure for this dataset, a 10-fold cross-validation scheme is used. The dataset is randomly split into 10 folds of equal size. Ten different experiments are run, each experiment using one fold for testing, one fold for validation and 8 folds for training. Throughout these 10 experiments, each fold should have been used once as test set, once as validation set and 8 times as part of the training set. The final results are averages on all 10 folds and on 3 different runs of the whole experiment.

The validation sets are used to select the optimal GNN and RelNN parameters, i.e., the number of units in the different parts of the network. There are three different numbers that have to be chosen:

- The number of hidden units in the top node of the RelNN, which corresponds to the number of hidden units in the output network of the GNN.
- The number of output units of the atom node network of the RelNN, which corresponds to the number of output units of the transition network of the GNN.
- The number of hidden units of the atom node network of the RelNN, which corresponds to the number of hidden units of the transition network of the GNN.

All these numbers are chosen to be 2, 5 or 10. In order to keep the number of experiments small, the number of hidden units in the top node and the atom node network are required to be the same. This yields a total of 9 possible configurations. For RelNNs, the unfolding depth and the type of the atom node network should also be determined. The unfolding depth will be restricted to be 0, 1 or 2 and the considered network types are locally recurrent networks (LRC), fully recurrent networks (FRC) and aggregate networks using sum units (SUM). Another issue is the number of training iterations. For this, two different possibilities are considered:

- A fixed number of 500 iterations is used.
- Every 20 iterations, the network is evaluated on the validation set. After 500 iterations, training is stopped and the network corresponding to the iteration with the best performance on the validation set is selected.

The first strategy may be better, provided there is no overfitting. The second strategy is preferable when a sufficiently large validation set is available and it provides a good estimation of the error on the test set. Table 6.2 shows the performance of GNNs and RelNNs on the two parts of the mutagenesis dataset and on the whole dataset. The average accuracy and the sample standard deviation are displayed. Several conclusions can be drawn from these results.

The results indicate that a fixed number of 500 iterations works better than using the validation set for GNNs, whereas the converse holds for RelNNs. In this case, both the validation and the training sets are probably too small. In GNNs this gives rise to an early stop of the learning, whereas in RelNNs overfitting is observed. That RelNNs suffer from overfitting, can be confirmed by observing the difference between the performance on the training set and on the test set. This difference is small for GNNs but large for RelNNs (see Tables 6.3 and 6.4). The reason why overfitting occurs for the RelNNs and not for the GNNs is hard to determine with certainty. It might be due to the number of parameters, which is larger than for GNNs, but it could also be caused by differences with regard to information encoding or training. GNNs use for instance some form of regularization to ensure that the network implements a contraction mapping. Using different parameters for GNNs and RelNNs or increasing the number of folds might improve the performance, but such possibilities have not been explored further.

Table 6.2 shows results for different knowledge bases. More precisely, three cases are considered: AB (only atom and bond information is used), AB+C (atom and bond information is used but also the global chemical measurements of the compound) and AB+C+PS (atom and bond information, chemical

measurements and the precoded structural attributes of the compound are used). It is important to notice that the relational information is in AB while the propositional information is in C and PS. The results indicate that GNNs and RelNNs can merge the propositional and relational information. The best performances are achieved when pure graph information (AB) is combined with global information (C and PS).

The results for AB alone are interesting because they indicate whether RelNNs and GNNs are able to learn from relational information, which is the most important question here. The results for AB are worse than the results for AB+C+PS, but they are not that bad. The baseline accuracy, which is computed by always predicting majority class, on the regression-friendly, the regression-unfriendly and the whole dataset is 66.49%, 69.05% and 60.00%, respectively. This means that both methods are effective in learning from the atom and bond information alone, without using any pregenerated features. Also note that the sample standard deviations are for the most part small, suggesting that the network initialization has a small effect on the accuracy and that the results are statistically significant. The results for AB are also interesting because in this case, the data representations for RelNNs and GNNs are closest to each other, because the GNNs do not attach global information to the atom nodes. The accuracies of GNNs and RelNNs are closer in this case, which suggests that the better GNN results may be partially due to the different information encodings used by the two models.

Tables 6.3 and 6.4 compare the performance of the models when different numbers of hidden and state units are used. The table displays the performance on the whole mutagenesis dataset using the knowledge base AB+C+PS. Similar results were obtained using the separate parts of the dataset and different knowledge bases. The results show that the number of hidden and state units does affect the performance somewhat, but most results are close to each other.

A review of the published results on the regression–friendly part can be found in [31], whereas [39] and [48] present a selection of results using the full set of compounds. Tables 6.5, 6.6 and 6.7 report the accuracies achieved by different techniques on the regression–friendly part, the regression–unfriendly part and the whole mutagenesis dataset, respectively. The best results for RelNNs and GNNs are shown as well.

The comparison is not straightforward because different methods use different knowledge bases. However, focusing on the best performance of each method and disregarding the used knowledge base, it turns out that GNNs outperform other methods on the regression-unfriendly part, while RelNN results are in line with other results. For the regression–friendly part and the whole dataset, best results for GNNs are very similar to the best results for other methods,

model	KB	best architecture				accuracy	
		state dim.	**hidden dim.**	**unf. depth**	**agg. type**	**500 epochs**	**best on val.**
whole dataset							
GNN	AB	10	10	–	–	81.74 [2.42]	79.13 [1.99]
GNN	AB+C	5	2	–	–	88.12 [0.50]	85.51 [0.66]
GNN	AB+C+PS	10	2	–	–	87.54 [1.00]	86.38 [0.25]
RelNN	AB	2	10	2	SUM	79.57 [2.01]	78.26 [0.64]
RelNN	AB+C	5	10	2	SUM	77.10 [1.47]	79.57 [1.70]
RelNN	AB+C+PS	5	2	2	SUM	80.87 [2.51]	83.04 [1.13]
regression–friendly part							
GNN	AB	10	10	–	–	80.49 [0.81]	79.59 [0.63]
GNN	AB+C	2	5	–	–	94.83 [0.83]	93.61 [1.07]
GNN	AB+C+PS	2	2	–	–	95.92 [0.32]	93.06 [0.93]
RelNN	AB	2	10	2	SUM	87.77 [2.48]	84.75 [1.82]
RelNN	AB+C	2	2	1	SUM	87.77 [1.22]	88.30 [0.45]
RelNN	AB+C+PS	10	10	1	SUM	88.30 [1.27]	91.49 [0.53]
regression–unfriendly part							
GNN	AB	10	5	–	–	79.67 [2.75]	79.83 [1.44]
GNN	AB+C	2	10	–	–	95.83 [1.44]	89.83 [1.61]
GNN	AB+C+PS	2	10	–	–	95.83 [1.44]	94.33 [1.15]
RelNN	AB	2	2	2	SUM	78.57 [7.72]	79.37 [2.71]
RelNN	AB+C	5	10	2	SUM	70.63 [4.64]	80.95 [2.71]
RelNN	AB+C+PS	5	10	2	SUM	73.02 [4.26]	80.16 [3.98]

Table 6.2: The performance of RelNNs and GNNs on the mutagenesis dataset. KB is the knowledge base. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The accuracy is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the accuracies are shown between brackets.

architecture			training acc.		test acc.	
state dim.	**hidden dim.**	**agg. type**	500 **epochs**	**best on val.**	500 **epochs**	**best on val.**
2	2	LRC	89.62	83.37	81.01	80.72
2	2	FRC	89.64	87.54	81.30	81.45
2	2	SUM	86.81	82.54	78.70	81.01
2	5	LRC	94.28	83.71	76.67	81.30
2	5	FRC	95.49	84.24	79.13	82.17
2	5	SUM	92.36	83.24	79.13	80.72
2	10	LRC	96.88	82.64	76.38	80.58
2	10	FRC	98.24	85.92	74.78	79.86
2	10	SUM	94.71	85.63	77.25	78.99
5	2	LRC	91.18	82.14	80.87	80.29
5	2	FRC	89.75	86.41	80.87	83.04
5	2	SUM	85.43	83.55	78.26	77.97
5	5	LRC	95.24	89.15	80.29	81.16
5	5	FRC	97.26	83.93	77.83	81.45
5	5	SUM	91.68	84.22	78.70	78.70
5	10	LRC	98.15	88.91	76.81	82.61
5	10	FRC	99.31	84.67	76.38	81.74
5	10	SUM	95.07	84.67	78.55	80.72
10	2	LRC	90.25	87.21	81.30	81.30
10	2	FRC	90.14	86.39	78.99	81.30
10	2	SUM	76.18	74.35	71.88	71.88
10	5	LRC	95.05	85.07	79.28	81.88
10	5	FRC	96.92	88.03	78.99	82.46
10	5	SUM	84.20	80.20	76.09	76.23
10	10	LRC	98.39	84.91	77.68	82.46
10	10	FRC	99.37	84.31	75.65	80.43
10	10	SUM	91.74	85.38	77.68	79.13

Table 6.3: Overview of the performance of RelNNs on the whole mutagenesis dataset for different network types and dimensions. Accuracies are shown on both training and test set for a fixed number of 500 training iterations as well as for the optimal number of training iterations as determined by using the validation set.

architecture		**training acc.**		**test acc.**	
state dim.	**hidden dim.**	**500 epochs**	**best on val.**	**500 epochs**	**best on val.**
2	2	89.84	86.90	87.97	87.25
2	5	90.96	87.67	88.70	87.10
2	10	91.57	87.68	88.70	86.38
5	2	90.17	87.63	87.39	85.80
5	5	91.07	88.22	88.41	86.81
5	10	91.72	87.54	87.97	86.09
10	2	89.82	87.48	87.54	86.38
10	5	90.94	87.41	88.99	87.10
10	10	91.56	88.08	89.13	86.23

Table 6.4: Overview of the performance of GNNs on the whole mutagenesis dataset for different network dimensions. Accuracies are shown on both training and test set for a fixed number of 500 training iterations as well as for the optimal number of training iterations as determined by using the validation set.

RelNNs perform a bit worse but still in line with the majority of the results.

6.2.3 Biodegradability

In this section, results of experiments on the biodegradability dataset [12] are presented. Since the experimental procedure is very similar to the one for mutagenesis, only the differences are discussed. The reader is referred to the previous section for the representation of the compounds, the model architectures and any other detail that is not explicitly reported here.

The biodegradability dataset is very similar to the mutagenesis dataset. The aim is to predict the degree of biodegradability of 328 chemical compounds. The compounds are described by their atom and bond structure, and some global information, i.e., molecular weight and log P. This information gives a full description of the molecules, but in earlier experiments conducted on this dataset, extra descriptors were built. These extra descriptors include a vector with occurrences of functional groups and a vector with counts of small substructures in the molecules.

model	KB	reference	accuracy
GNN	AB		80.49
GNN	AB+C		94.83
GNN	AB+C+PS		95.92
RelNN	AB		84.75
RelNN	AB+C		88.30
RelNN	AB+C+PS		91.49
RS	AB	[31]	88.9
RDBC	AB	[23]	84
$1nn(d_m)$	AB	[39]	83
FOIL	AB	[38]	76
MFLOG	AB+C	[26]	95.7
$1nn(d_m)$	AB+C	[39]	91
RDBC	AB+C	[23]	83
P-Progol	AB+C	[47]	82.0
RS	AB+FG	[31]	89.9
Neural Networks	C+PS	[47]	89.0
RSD	AB+C+FG	[28]	92.6
RELAGGS	AB+C+FG	[28]	88.0
P-Progol	AB+C+FG	[47]	88.0
SINUS	AB+C+FG	[28]	84.5
RS	AB+C+PS+FG	[31]	95.8
boosted-FOIL	not available	[37]	88.3

Table 6.5: A comparison of the performance of GNNs and RelNNs with other techniques on the regression–friendly part of the mutagenesis dataset. KB is the used knowledge base.

method	**KB**	**reference**	**accuracy**
GNN	AB		79.67
GNN	AB+C		95.83
GNN	AB+C+PS		95.83
RelNN	AB		79.37
RelNN	AB+C		80.95
RelNN	AB+C+PS		80.16
TILDE	AB	[9]	85
RDBC	AB	[23]	79
$1nn(d_m)$	AB	[39]	72
TILDE	AB+C	[9]	79
RDBC	AB+C	[23]	79
$1nn(d_m)$	AB+C	[39]	72

Table 6.6: A comparison of the performance of GNNs and RelNNs with other techniques on the regression–unfriendly part of the mutagenesis dataset. KB is the used knowledge base.

method	**KB**	**reference**	**accuracy**
GNN	AB		81.74
GNN	AB+C		88.12
GNN	AB+C+PS		87.54
RelNN	AB		78.26
RelNN	AB+C		79.57
RelNN	AB+C+PS		83.04
RDBC	AB	[23]	83
$1nn(d_m)$	AB	[39]	81
TILDE	AB	[9]	77
$1nn(d_m)$	AB+C	[39]	88
RDBC	AB+C	[23]	82
TILDE	AB+C	[9]	82

Table 6.7: A comparison of the performance of GNNs and RelNNs with other techniques on the whole mutagenesis dataset. KB is the used knowledge base.

This means that five different knowledge bases can be used for learning. Three of them (P0, P1 and P2) contain global information of the whole molecule, two of them (R0 and R1) contain relational information of the molecule atoms:

- **P0:** contains the molecular weight and log P.
- **P1:** consists of the counts of functional groups in the molecule.
- **P2:** consists of the counts of common substructures in the molecule.
- **R0:** consists of raw atom and bond information.
- **R1:** consists of predicates detecting the presence of functional groups and substructures. They are only relevant for ILP systems and they are defined on top of R0, so they are considered to be part of the relational information.

A more detailed description of the information encoded by these knowledge bases can be found in [12]. For the experiments, atom and bond types are represented by one–hot encoded vectors. Other features are represented directly by their respective values. P0 is a 2–dimensional vector, and P1 and P2 are 30-dimensional sparse vectors.

For this dataset, both regression and classification problems are considered. For the regression problem, the half–life time has to be predicted directly. For the classification problem, four categories are defined by using thresholds on the half–life time. This results in four classes: chemicals that degrade fast, moderately fast or slowly, and chemicals that are resistant.

For the classification task, RelNNs and GNNs with four output units are used, one for each biodegradability class. A one–hot encoding schema is used to represent each class, i.e., the vector $t_1 = [1, -1, -1, -1]$ stands for the first class, $t_2 = [-1, 1, -1, -1]$ stands for the second class, etcetera. The models are trained to output the vector that represents the class to which the processed molecule belongs to. During testing, the predicted class is determined by looking at the output with the highest value. The performance is measured with two different measures. The first measure is standard accuracy, the percentage of the correctly classified compounds. The other measure is accuracy up to one error (accuracy ± 1), which consists of the percentage of examples whose classification is at most one class up or down from the correct classification. For instance, a molecule that belongs to the class fast and is classified as moderately fast, still counts as correctly classified for accuracy ± 1. For the regression task, the RelNN and GNN models have only one output and are trained using the half-life time value as target. Just as in the original paper [12], the performance

is evaluated using the correlation score between the output of the model and the target to be predicted.

Tables 6.8 and 6.9 show the results for the biodegradability dataset. It must be noted that the biodegradability dataset is not an easy learning task so the performance is not that great in general. The best accuracy on the classification task and the best correlation on the regression task are obtained by GNNs with knowledge base R0+P0. On the other hand, the accuracy ±1 on the classification task is close to 90% for almost all the methods and the differences are not very significant from a statistical point of view.

Interestingly, the maximal knowledge base R0+P0+P1+P2 is not automatically the knowledge base that yields the best performance. This can be explained by the fact that a larger number of attributes requires a larger number of parameters and makes the networks prone to overfitting problems. Actually, the fact that P1 and P2 are sparse vectors, since some of the substructures are rather infrequent in the data, can worsen the problem. It is also interesting to note that in some cases the performance of the models is reasonably good without any precoded attributes, i.e., using R0 alone, which includes only the graph connectivity and the atom and bond types.

Whether it is best to use a fixed number of training iterations or the validation set to determine the optimal number of validations, depends. A fixed number of iterations gives best results for GNNs on the classification task. Using the validation set seems to be the better choice for RelNNs on the regression task. In the other cases, there is no clear difference between both settings.

Table 6.10 illustrates the effect of using different numbers of hidden and state units. The table shows results for the classification task using features R0+P0. For RelNNs, only sum networks are considered. For the GNNs, it is clear that it is best to use a fixed number of training iterations. The results also improve when the number of hidden units is larger, the number of state units is less important. For RelNNs, the number of iterations and the number of hidden and state units do not have a decisive impact on the results, but most importantly, the accuracy is significantly lower than for GNNs.

Tables 6.11 and 6.12 show a comparison with other results achieved on the classification and regression task, as published in [12]. Note that R0, which contains the graph connectivity, is needed by GNNs and RelNNs and has always been used in the experiments, whereas such information is not useful for most of the other methods, since R0 can not be used by propositional methods and is partially and implicitly contained in some of the propositional attributes. The results show that the performance of the GNNs and some of the RelNNs is comparable to the other results, but not better. GNNs achieve the second

model	**KB**	**best architecture**				**correlation**	
		state dim.	**hidden dim.**	**unf. depth**	**agg. type**	**500 epochs**	**best on val.**
GNN	R0	10	10	–	–	0.6023 [0.0137]	0.6035 [0.0149]
GNN	R0+P0	5	10	–	–	0.6823 [0.0056]	0.6822 [0.0049]
GNN	R0+P0+P1+P2	2	10	–	–	0.4449 [0.0411]	0.4452 [0.0413]
RelNN	R0	5	5	2	SUM	0.6521 [0.0186]	0.6304 [0.0140]
RelNN	R0+P0	2	5	2	SUM	0.6516 [0.0203]	0.6499 [0.0210]
RelNN	R0+P0+P1	5	2	2	LRC	0.6066 [0.0238]	0.6268 [0.0237]
RelNN	R0+P0+P2	5	5	1	LRC	0.5619 [0.0427]	0.6417 [0.0120]
RelNN	R0+P0+P1+P2	2	5	1	FRC	0.5287 [0.0244]	0.6729 [0.0213]

Table 6.8: The performance of RelNNs and GNNs on the biodegradability regression task. The performance is measured using the correlation score. KB is the knowledge base. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The correlation is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the correlations are shown between brackets.

		best architecture				accuracy		accuracy ±1	
model	**KB**	**state dim.**	**hidden dim.**	**unf. depth**	**agg. type**	500 **epochs**	**best on val.**	500 **epochs**	**best on val.**
GNN	R0	2	10	–	–	52.60 [0.97]	45.01 [1.36]	89.42 [0.94]	89.01 [1.70]
GNN	R0+P0	5	10	–	–	58.34 [0.97]	53.66 [3.51]	92.96 [0.91]	91.65 [0.18]
GNN	R0+P0+P1	10	10	–	–	54.85 [1.31]	42.66 [0.63]	91.55 [2.45]	89.60 [1.33]
GNN	R0+P0+P2	2	10	–	–	53.31 [1.63]	41.68 [1.97]	88.60 [1.37]	91.34 [1.53]
GNN	R0+P0+P1+P2	2	10	–	–	51.09 [1.85]	41.17 [0.02]	87.38 [2.00]	88.32 [1.55]
RelNN	R0	5	2	2	LRC	39.33 [3.16]	40.00 [1.37]	87.87 [1.30]	92.26 [0.63]
RelNN	R0+P0	2	2	1	FRC	43.23 [1.60]	43.60 [0.43]	91.95 [1.56]	92.99 [0.83]
RelNN	R0+P0+P1	2	5	1	LRC	52.01 [4.01]	50.73 [1.62]	91.04 [1.19]	92.01 [1.02]
RelNN	R0+P0+P2	5	5	2	FRC	48.72 [2.43]	47.62 [1.47]	88.60 [1.02]	90.67 [0.88]
RelNN	R0+P0+P1+P2	2	5	2	LRC	49.94 [1.77]	51.46 [2.09]	89.39 [2.19]	91.22 [0.85]

Table 6.9: The performance of RelNNs and GNNs on the biodegradability classification task. The performance is measured using accuracy and accuracy up to one error (accuracy ±1). KB is the knowledge base. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The accuracy is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the accuracies are shown between brackets.

architecture		train accuracy		test accuracy		test acc. ± 1	
state dim.	**hidden dim.**	500 epochs	**best on val.**	500 epochs	**best on val.**	500 epochs	**best on val.**
GNNs on the classification task using R0+P0							
2	2	66.96	56.33	53.11	49.10	92.26	90.22
2	5	74.57	58.03	56.79	48.27	92.16	92.25
2	10	78.39	58.86	57.09	47.48	91.64	91.03
5	2	67.03	57.75	53.33	51.42	90.22	89.61
5	5	75.61	61.06	57.82	49.97	92.56	90.92
5	10	79.52	64.58	58.34	53.66	92.96	91.65
10	2	66.75	54.72	52.93	48.87	90.63	91.02
10	5	75.71	59.96	57.30	49.49	92.46	89.80
10	10	79.12	64.71	56.71	52.64	93.17	91.35
RelNNs on the classification task using R0+P0							
2	2	50.79	45.43	43.23	43.60	91.95	92.99
2	5	61.71	46.10	41.16	42.50	88.29	93.66
2	10	69.43	47.33	42.87	43.29	86.28	91.95
5	2	51.49	46.33	43.60	43.54	91.71	93.66
5	5	62.82	46.06	42.68	43.96	87.80	93.23
5	10	73.40	45.56	39.02	42.50	85.85	93.05
10	2	52.23	46.18	42.56	42.93	92.62	93.60
10	5	65.71	47.93	44.39	43.17	88.66	92.99
10	10	78.82	46.62	39.09	41.71	85.67	92.68

Table 6.10: Performance of RelNNs and GNNs on the biodegradability dataset for different dimensions. Experiments for the RelNNs were all carried out using SUM networks and an unfolding depth of 2. The number of state units, the number of hidden units, and the accuracy after 500 iterations and after the optimal number of iterations as determined by using the validation set, are shown. For the training set, accuracy is shown, for the test set accuracy and accuracy up to one error (acc. ± 1).

best performance on the classification and regression task. The performance of the RelNNs varies a lot and depends on the used knowledge base. The good results obtained by methods that do not exploit R0 suggest that the counts of functional groups and substructures are probably a good propositionalization of the relational data, as already mentioned in [12]. On the other hand, the relatively good performance obtained by GNNs on the classification task and by RelNNs on the regression task, using only R0, suggests that these methods can actually extract a large part of the information in P0, P1, P2 and R1 directly from the original graph.

6.3 Conclusions

The relational neural networks presented in this chapter are a logical extension of the networks for bags from Chapter 4 to relational domains. They are very similar in structure to recursive neural networks but focus much more on the aggregation aspect than on learning structural features. The first experiments on the subtrees datasets showed that these RelNNs work well on datasets with shallow tree structures, but performance degrades when datasets with deeper tree structures are considered. This was to be expected and it confirms the assumption that these models are best suited for learning in local environments, i.e., in situations where the relevant information is concentrated and extensive structural information is not relevant for the learning task at hand. On the mutagenesis and biodegradability datasets, performances can be achieved that are more or less in line with other approaches, but not better. Overall, GNNs seem to do better on these datasets. It might be that their preservation of cycles in the data pays off, but there are also some other differences in data encoding and network training that might account for this difference in performance. To enable a fair comparison between methods, experiments were all carried out in settings that are as similar as possible for the different methods. But these settings might advantage or disadvantage one method with regard to the others and might not be optimal for all methods.

In general, one should also be careful to draw definitive conclusions from experiments with this kind of real-world datasets, because these datasets tend to have very peculiar characteristics. It might also be the case that these problems depend too much on structural features while the focus, at least for RelNNs, should definitely be on aggregation. But it is hard to find datasets like that, datasets also for which a fair number of results have been reported and that can be used as a benchmark. But at least the experiments show that RelNNs are capable of extracting relational information.

models	**KB**	**accuracy**	**accuracy** ± 1
GNN	R0	52.60	89.42
GNN	R0+P0	58.34	92.96
GNN	R0+P0+P1	54.85	91.55
GNN	R0+P0+P2	53.31	88.60
GNN	R0+P0+P1+P2	51.09	87.38
RelNN	R0	40.00	92.26
RelNN	R0+P0	43.60	92.99
RelNN	R0+P0+P1	50.73	92.01
RelNN	R0+P0+P2	47.62	90.67
RelNN	R0+P0+P1+P2	51.46	91.22
C4.5	P0+P1	55.2	86.2
C4.5	P0+P2	56.9	82.4
RIPPER	P0+P1	52.6	89.8
RIPPER	P0+P2	57.6	93.9
M5'	P0+P1	53.8	94.5
M5'	P0+P2	59.8	94.7
FFOIL	P0+R0	53.0	88.7
ICL	P0+R1	55.7	92.6
SRT-C	P0+P1	50.8	87.5
SRT-C	P0+P1+R1	55.0	90.0
SRT-R	P0+P1	49.5	91.9
SRT-R	P0+P1+R1	51.6	92.8
TILDE-C	P0+R1	51.0	88.6
TILDE-C	P0+P1+R1	52.0	89.0
TILDE-R	P0+R1	52.6	94.0
TILDE-R	P0+P1+R1	52.4	93.9

Table 6.11: A comparison of the performance of GNNs and RelNNs with other methods on the classification task of the biodegradability dataset. The performance is measured using accuracy and accuracy up to one error (accuracy ± 1). KB is the used knowledge base.

models	KB	correlation
GNN	R0	0.6023
GNN	R0+P0	0.6823
GNN	R0+P0+P1+P2	0.4449
RelNN	R0	0.6511
RelNN	R0+P0	0.6329
RelNN	R0+P0+P1	0.6253
RelNN	R0+P0+P2	0.5228
RelNN	R0+P0+P1+P2	0.5515
M5'	P0+P1	0.666
M5'	P0+P2	0.693
SRT-R	P0+P1	0.580
SRT-R	P0+P1+R1	0.632
TILDE-R	P0+R1	0.622
TILDE-R	P0+P1+R1	0.623

Table 6.12: A comparison of the performance of GNNs and RelNNs with other methods on the regression task of the biodegradability dataset. KB is the used knowledge base.

Chapter 7

Relational Cascade-Correlation Networks

The relational networks from the last chapter can be seen as an extension of the simple aggregate network types from Chapter 4 to relational domains. It is only a logical step to try to extend the cascade-correlation approach from that chapter to relational domains as well and that is exactly the topic of this chapter. Also, experimental results for the same datasets as in the previous chapter will be discussed.

7.1 Relational Cascade-Correlation Networks

The goal is to create a cascade-correlation approach for relational domains, RelCCNs, similar to RelNNs. This is however not straightforward with the treelike network structure considered in the previous chapter, so a somewhat alternative approach will be developed.

7.1.1 Cascade-Correlation for Relational Domains

Using cascade-correlation turned out to be quite successful for learning from bags, as discussed in Chapter 4. It would also be a useful technique to apply to relational data, for the same reasons, i.e., it allows the algorithm to choose the right number and types of neurons and to construct hybrid network structures in a step by step fashion. In fact, it could be even more useful for learning from

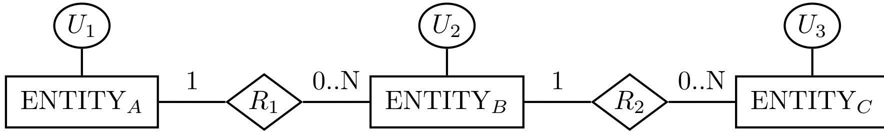

Figure 7.1: ER schema for Example 7.1.

relational domains because the choices to be made about numbers and types of units depend on the number of entity and relationship types, as explained in the previous chapter. The number of possibilities grows rapidly when the number of tuple and relationship types increases, so it becomes impracticable to try out all possibilities. A method to choose the right option automatically during training would be very helpful.

Using cascade-correlation in this setting is complicated however by the fact that it is unclear how to train units deep in the network with this method. The problem is that for each trained unit the correlation with the output error needs to be maximized, but for the deeper levels of the network, this correlation also depends on the units in between the trained unit and the output unit. The following example will make this more clear.

Example 7.1. *Consider the ER schema in Figure 7.1. The corresponding relational network would consist of three parts, one feedforward part for entity A, and two aggregation parts for entities B and C. When a unit is added to the network part for C, the question is how to maximize the correlation of this unit with the output error signal. First of all, there are multiple output values for this new unit within the same data pattern, because each B tuple of the data pattern can have its own bag of related C tuples. For each of these bags, there is a different output value for the C network part. To maximize the correlation of a new unit, it is necessary to determine its final, total effect on the output of the network. But this effect can only be determined by combining the different output values for the C tuple bags in the network layers above. In this case, it means that the network part for B should be involved in maximizing the correlation. To maximize that correlation, the B network part should be retrained during the training of the new unit, but this defeats the basic idea of the cascade-correlation approach to train the network one unit at a time.*

It is clear from this example that using cascade-correlation is getting more complicated with this kind of relational networks. The original simplicity of the cascade-correlation approach is lost. And the problem related to the training

of deep networks persists. Therefore, a different solution is chosen here. The treelike structure is given up and a linearized data format, more similar to the bag data of Chapter 4, is adopted. After that, an algorithm similar to the cascade-correlation algorithm for the bag data can be used.

7.1.2 Linearizing Relational Data

The first step is to linearize the relational data. This means that each input pattern is converted into a simpler pattern, removing part of the structural information. The chosen format has one single vector and a number of bags of vectors.

Definition 7.1 (Linearized Relational Dataset). *A linearized relational dataset is a collection of pairs of input and target patterns, where the target patterns are simple vectors and the input patterns are given by a tuple* $\langle V, B_1, \ldots, B_N \rangle$*, with* V *a single vector and* B_1 *to* B_N *bags of vectors. All vectors in a bag are of the same type and the types of* V *and* B_1 *to* B_N *are the same for all input patterns.*

In practice, this means that an input pattern is described by a single vector for all the global information of the pattern and a number of bags of vectors for information that is related to the data example but that cannot be transformed into a single attribute. How the transformation of an input pattern is carried out in practice, is deliberately left open. In general, it depends on the considered problem which transformation is most appropriate for that problem. It will become more clear how this is done in the experiments section.

Sometimes it will be possible to transform data into this format without loss of information, but in other cases, some of the structural information will not be preserved during the transformation. This can be a problem when learning some kind of concepts, but even with models such as RelNNs and GNNs, which try to preserve the structural information, some kind of concepts turn out to be impossible to learn. In any case, the focus is, again, more on aggregation than on structural aspects.

7.1.3 Relational Cascade-Correlation Algorithm

Using cascade-correlation on this linearized relational data is quite straightforward. It is a combination of the standard cascade-correlation method for feedforward networks and the aggregate cascade-correlation algorithm of Chapter 4. In each iteration step, units are generated for the single vector

Algorithm 7.1 The relational cascade-correlation training algorithm. The input patterns for this algorithm are the linearized data patterns consisting of a single vector and a number of bags of vectors.

N = new cascade-correlation network with no hidden units
train output weights of N
for $i = 1$ **to** maxNbOfNeurons **do**
  Pool = {}
  add m feedforward units to Pool for the single input vector
  for $j = 1$ **to** nbOfBags **do**
    for bag j, add m aggregate units of each type to Pool
  end for
  for each unit x **in** Pool **do**
    maximize correlation for x
  end for
  select unit x with the highest correlation
  add unit x to N
  retrain output weights of N
end for

and all the bags of vectors. The unit with the highest correlation is chosen to be added to the network. A more formal description is given in Algorithm 7.1. Besides the fact that multiple bags are used instead of a single one, the cascade-correlation approach works exactly the same as the approach described in Chapter 4.

7.2 Experiments

To compare RelCCNs with RelNNs, experiments on the same datasets as in Chapter 6 are conducted, i.e., the subtrees dataset and the mutagenesis and biodegradability datasets.

7.2.1 Subtrees

First of all, the subtrees experiment is repeated, but with RelCCNs instead of RelNNs. The considered datasets are exactly the same as the ones in Section 6.2.1, but the tree data needs to be transformed in the linearized relational data format. The single vector for each input pattern is of course the single tuple of type A for that input pattern. Moreover, three bags are added for each input pattern. The first bag contains all B tuples, the second bag is filled with

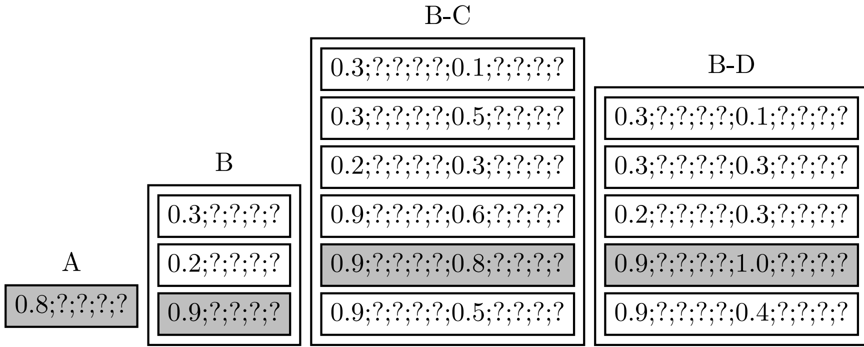

Figure 7.2: Example of a transformed input pattern of the subtrees dataset. This pattern is the transformed version of the example depicted in Figure 6.11 and has one simple input vector and three bags of input vectors. It is a positive example for the concept ‘A&B&C&D’ because of the subtree with first values greater than or equal to 0.75. This subtree consists of the gray nodes. The question marks stand for values that are not relevant for the concept under consideration.

	RelNN/LRC	**RelNN/MAX**	**RelNN/SUM**	**RelCCN**
A	99.80 [0.16]	100.00 [0.00]	99.80 [0.16]	99.63 [0.38]
B	66.47 [1.02]	99.27 [0.33]	95.13 [1.42]	99.27 [0.55]
C	63.40 [1.65]	61.93 [1.93]	62.93 [0.44]	97.03 [0.80]
A&B	76.13 [1.24]	78.53 [10.26]	95.47 [0.75]	98.57 [0.67]
B&C	53.93 [1.55]	75.20 [1.51]	82.60 [4.29]	90.77 [2.29]
C&D	58.20 [1.85]	60.80 [3.10]	61.20 [3.95]	67.23 [3.87]
A&B&C	74.60 [1.45]	77.53 [4.66]	84.60 [1.95]	91.33 [1.56]
B&C&D	58.60 [2.04]	73.07 [9.17]	77.67 [5.14]	87.80 [2.32]
A&B&C&D	74.47 [0.62]	82.13 [6.28]	86.73 [2.58]	89.23 [1.40]

Table 7.1: Results for RelCCNs on the subtrees dataset compared with different types of RelNNs. Average accuracies are shown with standard deviations between brackets.

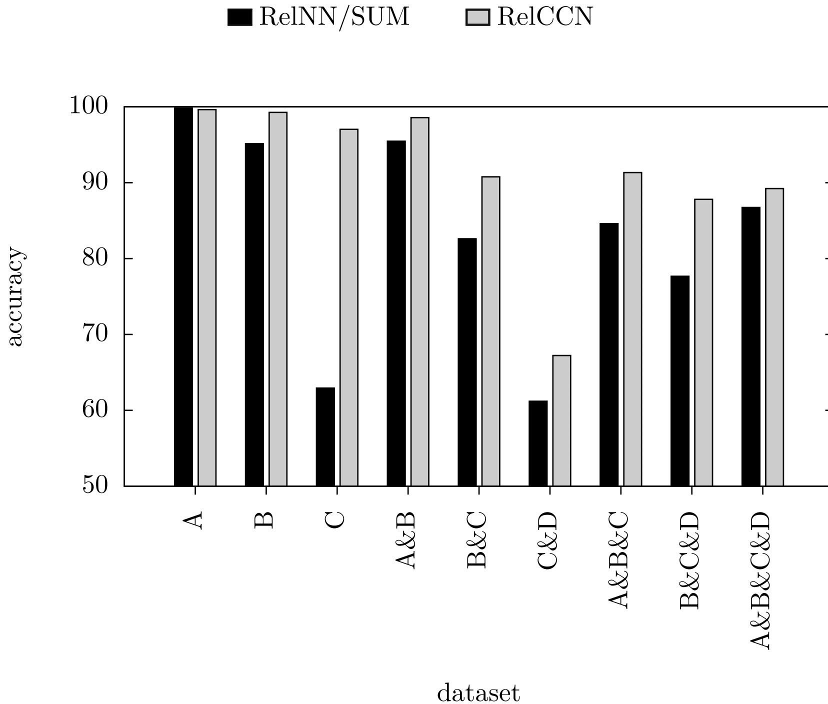

Figure 7.3: Accuracies for relational cascade-correlation networks on the subtrees datasets. Results for relational networks with the sum network type are also shown for comparison.

the Cartesian product of the B and C tuples and the third bag consists of the Cartesian product of the B and D tuples. An example is shown in Figure 7.2.

With this transformation, some structural information has been lost. It will be difficult to learn the concept 'C&D' for instance, because the information in the bags does not tell which C and D tuples are connected to the same B tuple. This could be solved by adding a bag with the Cartesian product of the B, C and D tuples, but this kind of solution tends to add quite large bags to the input pattern and may become practically infeasible if there are a lot of relationships in the data. Therefore, this possibility is not explored any further here and only the simple transformation is considered. After all, the focus is not so much on learning from the structural information, but on learning from the bag data. Apart from the data transformation, the experimental setup is similar to the setup for RelNNs.

model	KB	best architecture				accuracy	
		state dim.	hidden dim.	unf. depth	agg. type	500 epochs	best on val.
GNN	AB	10	10	–	–	81.74 [2.42]	79.13 [1.99]
GNN	AB+C	5	2	–	–	88.12 [0.50]	85.51 [0.66]
GNN	AB+C+PS	10	2	–	–	87.54 [1.00]	86.38 [0.25]
RelNN	AB	2	10	2	SUM	79.57 [2.01]	78.26 [0.64]
RelNN	AB+C	5	10	2	SUM	77.10 [1.47]	79.57 [1.70]
RelNN	AB+C+PS	5	2	2	SUM	80.87 [2.51]	83.04 [1.13]
RelCCN	AB	–	–	2	–	78.42 [3.05]	79.31 [3.68]
RelCCN	AB+C	–	–	2	–	77.89 [2.69]	78.21 [2.81]
RelCCN	AB+C+PS	–	–	2	–	80.54 [2.68]	79.89 [3.01]

Table 7.2: Results on the whole mutagenesis dataset for GNNs, RelNNs and RelCCNs. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs and RelCCNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The accuracy is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the accuracies are shown between brackets.

The network is trained with a maximum of 10 hidden units. In each iteration, 3 units of each type are constructed and trained for each bag in the data and the single vector. Each unit is trained for 500 iterations to maximize its output correlation. Experimental results are obtained using 10-fold cross-validation. If a validation set is used, 7 folds are used for training, 2 for validation and 1 for testing. Results are shown and compared with RelNNs in Table 7.1 and Figure 7.3. From these results, similar conclusions can be drawn as in Chapter 4. The cascade-correlation approach performs at least as good as the simple network approach, and in some cases substantially better. As long as no crucial information is lost in the transformation, the linearized data also enables to learn from more distant data tuples because they can be converted into more nearby data by the transformation. The results for dataset ‘C’ illustrate this clearly.

model	**KB**	**best architecture**				**accuracy**	
		state dim.	**hidden dim.**	**unf. depth**	**agg. type**	**500 epochs**	**best on val.**
regression–friendly part							
GNN	AB	10	10	–	–	80.49 [0.81]	79.59 [0.63]
GNN	AB+C	2	5	–	–	94.83 [0.83]	93.61 [1.07]
GNN	AB+C+PS	2	2	–	–	95.92 [0.32]	93.06 [0.93]
RelNN	AB	2	10	2	SUM	87.77 [2.48]	84.75 [1.82]
RelNN	AB+C	2	2	1	SUM	87.77 [1.22]	88.30 [0.45]
RelNN	AB+C+PS	10	10	1	SUM	88.30 [1.27]	91.49 [0.53]
RelCCN	AB	–	–	2	–	86.30 [2.76]	86.95 [2.54]
RelCCN	AB+C	–	–	2	–	85.31 [2.26]	87.44 [2.89]
RelCCN	AB+C+PS	–	–	2	–	89.30 [2.25]	91.05 [2.77]
regression–unfriendly part							
GNN	AB	10	5	–	–	79.67 [2.75]	79.83 [1.44]
GNN	AB+C	2	10	–	–	95.83 [1.44]	89.83 [1.61]
GNN	AB+C+PS	2	10	–	–	95.83 [1.44]	94.33 [1.15]
RelNN	AB	2	2	2	SUM	78.57 [7.72]	79.37 [2.71]
RelNN	AB+C	5	10	2	SUM	70.63 [4.64]	80.95 [2.71]
RelNN	AB+C+PS	5	10	2	SUM	73.02 [4.26]	80.16 [3.98]
RelCCN	AB	–	–	2	–	80.65 [3.46]	79.93 [3.29]
RelCCN	AB+C	–	–	2	–	78.87 [2.11]	80.21 [2.86]
RelCCN	AB+C+PS	–	–	2	–	76.19 [3.76]	80.24 [3.47]

Table 7.3: Results on the mutagenesis dataset for GNNs, RelNNs and RelCCNs. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs and RelCCNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The accuracy is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the accuracies are shown between brackets.

7.2.2 Mutagenesis

Experiments with RelCCNs on the mutagenesis dataset are also conducted. The dataset itself and experiments with RelNNs and GNNs were already discussed in Section 6.2.2. The experimental setup is as similar as possible to the one used in Section 6.2.2, only the differences will be stated explicitly.

One issue that needs further explanation, is the transformation of the data for use with the RelCCNs. As explained before, the input patterns consist of some global molecular information and the atom-bond structure of the considered molecule. This has to be transformed into a single attribute vector and a number of bags of attribute vectors. It follows naturally that the global molecular information is encoded in the single attribute vector. A first bag can be constructed by encoding all atoms that are part of the molecule. For each atom, type and charge can be encoded in a vector. This yields a first bag of vectors for each input pattern. Another bag can be constructed containing all bonded atom couples in the molecule. For each couple, a vector is constructed by concatenating the attributes of the first atom, the attributes of the bond and the attributes of the second atom. Which atom is the first and which the second is not important, each combination is only represented once. The same process can be carried out with all combinations of three atoms and two bonds, etcetera. Each time a new bag is created. The user needs to determine the maximal combination of atoms that is still considered. As a result, this will also be the number of bags in the transformed input pattern and it corresponds to the depth of the RelNNs in Section 6.2.2. For the experiments, the network is trained with a maximum of 10 hidden units. In each iteration, 3 units of each type are constructed and trained for each bag in the data and the single vector. Each unit is trained for 500 iterations to maximize its output correlation.

Results are shown in Tables 7.2 and 7.3. The results for RelCCNs are very similar to those for RelNNs. There is still a difference with the best results for GNNs. As in Section 6.2.2, an explanation for that difference could be the peculiarities of GNNs with regard to data representation and network encoding and training.

7.2.3 Biodegradability

Experiments on the biodegradability are also repeated. This dataset has already been discussed in Section 6.2.3, all details about the data can be found there. The data transformation is similar to the one for the mutagenesis data in the previous section. The molecule information is encoded in the global vector, the atom-bond information in the bags. For the experiments, the network is

trained with a maximum of 10 hidden units. In each iteration, 3 units of each type are constructed and trained for each bag in the data and the single vector. Each unit is trained for 500 iterations to maximize its output correlation.

The results are summarized in Tables 7.4 and 7.5. In general, RelCCNs perform at least as good or better than RelNNs. For the regression task, there seem to be less problems with overfitting when using RelCNNs. Compared to the GNNs, it depends on the setting which method performs best, there is no overall best.

7.3 Conclusions

The relational cascade-correlation approach discussed in this chapter completes the range of methods developed to process relational data. This method is the equivalent of the cascade-correlation method for learning from simple bags in Chapter 4. It can be seen as an extension from the single bag setting to a multiple bag setting. With a simple transformation, relational data can be converted to this format if it does not directly correspond to this format. Some structural information will be lost in most cases, but, for the considered problem setting which focuses on aggregation, the most relevant information that is spread out over multiple tuples should be preserved.

Experimental results confirm the conclusions already drawn in Chapter 4. Especially for the artificial subtrees dataset, it is clear that the cascade-correlation approach works better. For the mutagenesis and the biodegradability datasets, this is not so clear. It should be noted again that these datasets serve as a good benchmark because there are many other results available, but that they are not necessarily the most suited datasets for the methods developed in this dissertation.

model	KB	best architecture				correlation	
		state dim.	**hidden dim.**	**unf. depth**	**agg. type**	500 **epochs**	**best on val.**
GNN	R0	10	10	–	–	0.6023 [0.0137]	0.6035 [0.0149]
GNN	R0+P0	5	10	–	–	0.6823 [0.0056]	0.6822 [0.0049]
GNN	R0+P0+P1+P2	2	10	–	–	0.4449 [0.0411]	0.4452 [0.0413]
RelNN	R0	5	5	2	SUM	0.6521 [0.0186]	0.6304 [0.0140]
RelNN	R0+P0	2	5	2	SUM	0.6516 [0.0203]	0.6499 [0.0210]
RelNN	R0+P0+P1	5	2	2	LRC	0.6066 [0.0238]	0.6268 [0.0237]
RelNN	R0+P0+P2	5	5	1	LRC	0.5619 [0.0427]	0.6417 [0.0120]
RelNN	R0+P0+P1+P2	2	5	1	FRC	0.5287 [0.0244]	0.6729 [0.0213]
RelCCN	R0	–	–	2	–	0.6535 [0.0412]	0.6617 [0.0369]
RelCCN	R0+P0	–	–	2	–	0.6614 [0.0322]	0.6629 [0.0415]
RelCCN	R0+P0+P1	–	–	2	–	0.6486 [0.0503]	0.6528 [0.0476]
RelCCN	R0+P0+P2	–	–	2	–	0.6579 [0.0464]	0.6781 [0.0351]
RelCNN	R0+P0+P1+P2	–	–	2	–	0.6758 [0.0539]	0.6781 [0.0455]

Table 7.4: Results on the regression task of the biodegradability dataset. The performance is measured using the correlation score. KB is the knowledge base. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs and RelCCNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The correlation is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the correlations are shown between brackets.

model	**KB**	**best architecture**				**accuracy**		**accuracy (±1)**	
		state dim.	**hidden dim.**	**unf. depth**	**agg. type**	500 **epochs**	**best on val.**	500 **epochs**	**best on val.**
GNN	R0	2	10	–	–	52.60 [0.97]	45.01 [1.36]	89.42 [0.94]	89.01 [1.70]
GNN	R0+P0	5	10	–	–	58.34 [0.97]	53.66 [3.51]	92.96 [0.91]	91.65 [0.18]
GNN	R0+P0+P1	10	10	–	–	54.85 [1.31]	42.66 [0.63]	91.55 [2.45]	89.60 [1.33]
GNN	R0+P0+P2	2	10	–	–	53.31 [1.63]	41.68 [1.97]	88.60 [1.37]	91.34 [1.53]
GNN	R0+P0+P1+P2	2	10	–	–	51.09 [1.85]	41.17 [0.02]	87.38 [2.00]	88.32 [1.55]
RelNN	R0	5	2	2	lrc	39.33 [3.16]	40.00 [1.37]	87.87 [1.30]	92.26 [0.63]
RelNN	R0+P0	2	2	1	frc	43.23 [1.60]	43.60 [0.43]	91.95 [1.56]	92.99 [0.83]
RelNN	R0+P0+P1	2	5	1	lrc	52.01 [4.01]	50.73 [1.62]	91.04 [1.19]	92.01 [1.02]
RelNN	R0+P0+P2	5	5	2	frc	48.72 [2.43]	47.62 [1.47]	88.60 [1.02]	90.67 [0.88]
RelNN	R0+P0+P1+P2	2	5	2	lrc	49.94 [1.77]	51.46 [2.09]	89.39 [2.19]	91.22 [0.85]
RelCCN	R0	–	–	2	–	44.57 [4.67]	45.16 [3.89]	88.02 [4.96]	88.91 [3.26]
RelCCN	R0+P0	–	–	2	–	45.13 [3.77]	45.43 [4.01]	89.51 [3.48]	90.10 [3.91]
RelCCN	R0+P0+P1	–	–	2	–	51.23 [2.45]	51.02 [2.67]	91.87 [2.93]	92.01 [3.02]
RelCCN	R0+P0+P2	–	–	2	–	50.94 [2.53]	51.28 [3.14]	90.52 [2.31]	89.85 [2.47]
RelCCN	R0+P0+P1+P2	–	–	2	–	52.03 [3.09]	51.46 [2.95]	91.69 [2.72]	90.34 [3.17]

Table 7.5: Results on the classification task of the biodegradability dataset. The performance is measured using accuracy and accuracy up to one error (accuracy ±1). KB is the knowledge base. The best architecture is the one selected by using the validation set. The number of state units, the number of hidden units, the unfolding depth (only for RelNNs and RelCCNs) and the type of aggregate network (only for RelNNs) are shown for this architecture. The accuracy is shown for a fixed number of 500 training iterations as well as for the optimal number of iterations, as determined by using the validation set. Standard deviations on the accuracies are shown between brackets.

Chapter 8

Conclusions

To conclude, a brief summary of the dissertation is given, followed by an evaluation of the developed ideas and some pointers for future work.

8.1 Summary

The dissertation consists of two main parts. In the first part, the discussion focused on aggregation, considering learning from bags of tuples. In the second part, this was broadened to more diverse and complex relational learning settings.

8.1.1 Aggregation

The use of aggregate features in machine learning algorithms was the starting point of this dissertation. It is motivated by the need to process bags of data tuples instead of single tuples. Since the early years of machine learning, there has been interest in extending the input data patterns from the simple and well-known propositional domain to more complex and expressive data domains. It can be argued that the step from single to multiple tuples, without considering other aspects like structural relationships, is one of the most fundamental steps in this process.

In Chapter 3, an important distinction was made between propositionalization and direct approaches. In a propositionalization approach, aggregate features

are constructed before the actual learning process takes place, while in a direct approach these features are constructed during learning. Propositionalization has often been the first and natural choice in the past because it allows to use familiar propositional methods. It reduces a more complicated setting to a simpler one. But how this reduction is done and whether it will be a reduction that results in good performance or not, will depend entirely on the chosen propositionalization. This can be a very hard choice to make, especially when little is known about the considered problem and the targeted concept. Direct approaches are more advanced, but, as a consequence, also more complicated. They will typically involve traversing some search space, but this can be far from trivial for these aggregate features.

Another distinction was made between the aggregation and the selection part of aggregate features. When the general form of aggregate features is considered, it can be split up into two different aspects. There is typically a part of the feature that operates on single tuples. This is the selection part, which is not that different from the type of features found in propositional methods. The second part is an aggregate function that collects and aggregates the results of this selection operator on all tuples of a bag of data. This is the aggregation part and this is what distinguishes these features from propositional features.

In Chapter 4, several adaptations of known neural network structures were created to enable them to process bags of data. The advantage of using neural networks is that they can easily be adapted for such a setting. Different network structures were considered based on feedforward networks, networks with special aggregate units and recurrent networks.

Each of these structures has its own advantages and disadvantages. They will work well in some cases, but not in others. Therefore, a combined approach is desirable in which different types of structures are available to the learning algorithm. This is possible with aggregate cascade-correlation networks. These networks follow the cascade-correlation network approach, in which the network is constructed one unit at a time. At each step, several types of units can be considered and the best unit is selected to be added to the network. This allows a hybrid network structure that should deliver a more robust performance.

All these different methods were tested on several datasets, artificial and real-world, to assess them and to compare them with other approaches. The artificial datasets turned out to be most insightful. Results confirmed that in general, the cascade-correlation approach performs better than the simple networks. On multi-instance datasets, specialized methods turned out to have a better performance, but that is not a surprise. Nevertheless, the cascade-correlation approach achieves a reasonable performance on these problems as well. From the other experiments on real-world datasets, it is hard to draw

definitive conclusions. The main problem is that there are not that many real-world datasets readily available that focus on this particular learning setting and that can serve as a benchmark.

8.1.2 Relational Domains

In the second part of the dissertation, the focus shifted towards relational domains that still rely heavily on learning from bags of data but also involve some other aspects. In Chapter 5, known approaches are discussed that use neural networks on structured data like sequences, trees and graphs. An extensive literature on this topic already exists. There are however problems associated with the use of neural networks for this kind of domains, because of learning problems and limitations of the approximation capabilities in some cases.

These neural networks for structured data are however in some respects similar to the relational networks proposed in Chapter 6. Instead of focusing on learning structural features, as these existing approaches mostly did, the relational networks are meant in the first place to deal with the aggregation aspect of learning from relational domains. They provide an extension to the networks for bag data because they allow for heterogenous input patterns with multiple types of tuples and relationships. This allows to process data from relational domains like the data domain used in typical inductive logic programming settings.

Experiments with relational networks on an artificial dataset confirmed the presumption that these networks are less suited for learning from data when structural features should be learned or when the relevant information is further away from the target tuple. Results for the real-world datasets mutagenesis and biodegradability are reasonable, often within the same range as results obtained with different methods, but not outstanding. Better results were achieved with graph neural networks, the most advanced neural network approach for structured data. There does not seem to be a clear reason why graph neural networks perform better, but they do not seem to be plagued by the potential problems mentioned earlier, at least for these datasets.

In a similar way as for the bag learning setting, a cascade-correlation approach was created in Chapter 7 for the relational domain. Essentially, this can be seen as a multi-bag approach where the relational data is considered to be an input pattern consisting of multiple bags of tuples. On these multiple bags, a cascade-correlation approach can be used that is almost identical to the one for single bag data.

Experiments on the artificial dataset showed that the relational cascade-correlation approach yields better results than the relational networks. It turns out the cascade-correlation is a more robust method again, just as for the simple bag setting. For the mutagenesis and biodegradability datasets, results are close to previous results with relational networks, though the cascade-correlation approach seems to be slightly better overall.

8.2 Evaluation

The key idea behind this research was to develop a method for dealing with relational data, from simple bags of tuples to more heterogenous collections of tuples. Two basic principles are important here. First of all, the approach should focus on the aggregation aspect of this learning setting. Second, the aggregate features involved in the process should be constructed during learning and not in advance. It is clear that constructing these features before learning is not desirable because it relies heavily on prior knowledge about the problem at hand. In this sense, the new method stands clearly apart from approaches like propositionalization or kernels.

With respect to the two basic principles, the new method resembles existing work in the context of inductive logic programming. The approach taken here is however radically different with regard to the model that is learned and the learning algorithm. The model is not expressed in logic but in numerical parameters and learning is done by numerical optimization instead of traversing a search space. Both methods have their advantages and disadvantages. What is most attractive about a numerical model is that it is flexible. When the model is expressed in logic, the construction of the model still depends on predefined aggregate functions that are provided in the background knowledge. Numerical models can be more flexible and adaptable. In the case the aggregate feature form is not perfectly suited for the problem at hand, it can still be stretched and adapted to fit as well as possible. This results in a method that is more gradual and smooth in its performance and does not suffer too much from blind spots.

8.3 Future Work

The question remains whether the proposed approach can be further improved. Embedding the construction of aggregate features in the learning process is definitely an interesting and valuable idea. Doing this with some general,

parameterized feature form that is adapted during the learning process, is also the most promising way to do it. But there might be better ways to express these features than by using neural networks. Ideally, a more general and flexible aggregate feature form should be found that can be integrated in machine learning methods and fits seamlessly in the learning process.

One of the problems encountered throughout the research, is that real-world datasets suitable for the considered problems are often hard to find. Obviously, a lot of research has focused on propositional data, which encourages data representation in this format, even if the underlying problem is not optimally represented in this way. On the other hand, there has also been a lot of interest in structured data, and as a consequence, there are some interesting datasets for this setting. But the focus there has always been on structural aspects and there has not been that much consideration for the aggregation aspect. Experiments on artificial datasets do provide good insights about the performance of the developed methods and they confirm many of the a priori assumptions with empirical evidence, but it remains a challenge to identify real-world problems for which this approach would be a perfect choice.

Bibliography

[1] L.B. Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In M. Caudill and C. Butler, editors, *Proceedings of the IEEE International Conference on Neural Networks*, volume 2, pages 609–618. IEEE, New York, 1987.

[2] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5:157–166, 1994.

[3] P. Berka. Guide to the financial data set. In A. Siebes and Berka P., editors, *The ECML/PKDD 2000 Discovery Challenge*, 2000.

[4] C. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[5] H. Blockeel. *Top-down induction of first order logical decision trees.* PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 1998.

[6] H. Blockeel and M. Bruynooghe. Aggregation versus selection bias, and relational neural networks. In L. Getoor and D. Jensen, editors, *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003*, Acapulco, Mexico, 2003.

[7] L. De Raedt. Attribute-value learning versus inductive logic programming: The missing links (extended abstract). In *Proceedings of the 8th International Workshop on Inductive Logic Programming (ILP98)*, pages 1–8, London, UK, 1998. Springer-Verlag.

[8] L. De Raedt. *Logical and Relational Learning.* Springer, 2008.

[9] L. De Raedt and H. Blockeel. Using logical decision trees for clustering. In *Proceedings of the 7th International Workshop on Inductive Logic*

Programming ILP 1997, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 133–141. Springer-Verlag, 1997.

[10] A.K. Debnath, R.L. Lopex de Compandre, G. Debnath, A.J. Schusterman, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797, 1991.

[11] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Perez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71, 1997.

[12] S. Džeroski, H. Blockeel, B. Kompare, S. Kramer, B. Pfahringer, and W. Van Laer. Experiments in predicting biodegradability. *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, 1634:80–91, 1999.

[13] S. Džeroski and N. Lavrač. *Relational Data Mining*. Springer-Verlag, 2001.

[14] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532. Morgan Kaufmann, 1990.

[15] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998.

[16] T. Gärtner. A survey of kernels for structured data. *SIGKDD Explorations*, 5(1):49–58, 2003.

[17] T. Gärtner, P. A. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In Bernhard Schölkopf and Manfred K. Warmuth, editors, *COLT*, volume 2777 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2003.

[18] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN2005)*, pages 729–734, 2005.

[19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[20] B. Hammer. Approximation capabilities of folding networks. In *Proceedings of the 7th European Symposium on Artificial Neural Networks (ESANN 1999)*, pages 33–38, 1999.

[21] B. Hammer and J. Jain. Neural methods for non-standard data. In M. Verleysen, editor, *Proceedings of the 12th European Symposium on Artificial Neural Networks*, pages 281–292, 2004.

[22] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Macmillan Publishing, 1994.

[23] M. Kirsten. *Multirelational Distance-Based Clustering.* PhD thesis, School of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2002.

[24] A.J. 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, 2002.

[25] S. Kramer. Relational learning vs. propositionalization: investigations in inductive logic programming and propositional machine learning. *AI Communications*, 13(4):275–276, 2000.

[26] S. Kramer and L. De Raedt. Feature Construction with Version Spaces for Biochemical Applications. *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 258–265, 2001.

[27] S. Kramer, N. Lavrač, and P. Flach. Propositionalization approaches to relational data mining. In S. Džeroski, editor, *Relational Data Mining*, pages 262–286. Springer-Verlag, 2000.

[28] M.-A. Krogel, S. Rawles, F. Železný, P. Flach, N. Lavrač, and S. Wrobel. Comparative evaluation of approaches to propositionalization. In *Proceedings of the 13th International Conference on Inductive Logic Programming (ILP03)*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 194–217. Springer-Verlag, 2003.

[29] M.-A. Krogel and S. Wrobel. Transformation-based learning using multirelational aggregation. In *Proceedings of the 11th International Conference on Inductive Logic Programming (ILP01)*, pages 142–155. Springer-Verlag, 2001.

[30] 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.

[31] H. Lodhi and S. H. Muggleton. Is mutagenesis still challenging? In *Proceedings of the 15th International Conference on Inductive Logic Programming, ILP 2005, Late-Breaking Papers*, pages 35–40, 2005.

[32] A. Micheli. Neural network for graphs: a contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[33] A. Micheli, D. Sona, and R. Sperduti. Contextual processing of structured data by recursive cascade correlation. *IEEE Transactions on Neural Networks*, 15(6):1396–1410, 2004.

[34] 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.

[35] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[36] F.J. Pineda. Generalization of back–propagation to recurrent neural networks. *Physical Review Letters*, 59:2229–2232, 1987.

[37] J.R. Quinlan. Boosting first-order learning. In *Proceedings of the 7th International Workshop on Algorithmic Learning Theory*, volume 1160 of *Lecture Notes in Computer Science*, pages 143–155. Springer, 1996.

[38] J.R. Quinlan and R.M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, volume 667 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 1993.

[39] J. Ramon. *Clustering and instance based learning in first order logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2002.

[40] J. Ramon and L. De Raedt. Multi instance neural networks. In L. De Raedt and S. Kramer, editors, *Proceedings of the ICML-2000 workshop on attribute-value and relational learning*, pages 53–60, 2000.

[41] M. Riedmiller and H. Braun. Rprop - a fast adaptive learning algorithm. In *Proceedings of the International Symposium on Computer and Information Science VII*, pages 279–286, 1992.

[42] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 586–591, 1993.

[43] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[44] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20:81–102, 2009.

[45] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80, 2009.

[46] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8:429–459, 1997.

[47] A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In *Proceedings of the 4th International Workshop on Inductive Logic Programming*, pages 217–232, 1994.

[48] W. Uwents and H. Blockeel. Classifying relational data with neural networks. In *Proceedings of the 15th International Conference on Inductive Logic Programming*, volume 3625 of *Lecture Notes in Computer Science*, pages 384–396. Springer, 2005.

[49] A. Van Assche. *Improving the applicability of ensemble methods in data mining*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2008.

[50] C. Vens. *Complex aggregates in relational learning*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2007.

[51] C. Wang, S. D. Scott, J. Zhang, Q. Tao, D. E. Fomenko, and V. N. Gladyshev. A study in modeling low-conservation protein superfamilies. Technical Report UNL-CSE-2004-0003, University of Nebraska, 2004.

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

Curriculum Vitae

Werner Uwents was born on May 24, 1981 in Leuven, Belgium. After finishing high school at the Sint-Jozefscollege in Herentals in 1999, he studied engineering, philosophy and economics at the Katholieke Universiteit Leuven. He received a Master's degree of Science in Engineering in Computer Science in 2004, a Master's degree of Arts in Philosophy in 2007 and a Master's degree of Science in Economic Policy in 2009.

From 2004 till 2009, he worked in the Declarative Languages and Artificial Intelligence research group of the Katholieke Universiteit Leuven and conducted PhD research under the supervision of prof. Hendrik Blockeel. In 2006, he also spent several months in the research group of prof. Marco Gori at the University of Siena, Italy. He defended his PhD on October 19, 2012.

List of publications

Articles in internationally reviewed journals

- Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Marco Gori, and Franco Scarselli. Neural networks for relational learning: An experimental comparison. *Machine Learning*, 82(3):315–349, 2011.

Papers at international conferences and symposia, published in full in proceedings

- Werner Uwents and Hendrik Blockeel. A comparison between neural network methods for learning aggregate functions. In *Discovery Science*, volume 5255 of *Lecture Notes in Computer Science*, pages 88–99. Springer, October 2008.

- Werner Uwents and Hendrik Blockeel. Learning aggregate functions with neural networks using a cascade-correlation approach. In *Inductive Logic Programming, 18th International Conference, ILP 2008, Proceedings*, volume 5194 of *Lecture Notes in Computer Science*, pages 315–329. Springer, September 2008.

- Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Franco Scarselli, and Marco Gori. Two connectionist models for graph processing: An experimental comparison on relational data. In T. Gärtner, G.C. Garriga, and T. Meinl, editors, *MLG 2006, Proceedings on the International Workshop on Mining and Learning with Graphs*, pages 211–220, 2006.

- Werner Uwents and Hendrik Blockeel. Classifying relational data with neural networks. In *Inductive Logic Programming, 15th International*

Conference, ILP 2005, Proceedings, volume 3625 of *Lecture Notes in Computer Science*, pages 384–396. Springer, 2005.

- Werner Uwents and Hendrik Blockeel. Experiments with relational neural networks. In M. van Otterlo, M. Poel, and A. Nijholt, editors, *Proceedings of the Fourteenth Annual Machine Learning Conference of Belgium and the Netherlands*, pages 105–112, 2005.

- Werner Uwents, Celine Vens, Anneleen Van Assche, and Hendrik Blockeel. Learning aggregations and selections with relational neural networks. In M. Gori and P. Avesani, editors, *Proceedings of the Workshop on Sub-Symbolic Paradigms for Learning in Structured Domains*, pages 112–121, 2005.

- Hendrik Blockeel and Werner Uwents. Using neural networks for relational learning. In T. Dietterich, L. Getoor, and K. Murphy, editors, *ICML 2004 Workshop on Statistical Relational Learning and Its Connections to Other Fields*, pages 23–28, 2004.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Declarative Languages and Artificial Intelligence
Kasteelpark Arenberg 1
B-3001 Heverlee

KATHOLIEKE UNIVERSITEIT
LEUVEN

ASSOCIATIE K.U.LEUVEN