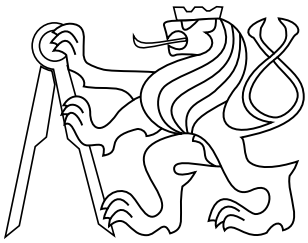
diploma thesis

Structure learning of neural-symbolic architectures

Martin Svatoš



May 2016

Thesis advisor: Ing. Gustav Šourek

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Computer Science

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Martin Svatoš**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Structure learning of neural-symbolic architectures**

Guidelines:

Get familiar with the area of neural-symbolic integration.
Research the state-of-the-art techniques of structure learning in the respective domain.
Create propositional testbed for existing approaches.
Select suitable methods and design transition into the first-order setting.
Incorporate with the method of Lifted Relational Neural Networks (with guidance).
Evaluate your approach on relational benchmarks, assess interpretability, demonstrate results.

Bibliography/Sources:

Sebastian Bader and Pascal Hitzler - Dimensions of Neural-symbolic Integration - A Structured Survey
Sourek, Aschenbrenner, Zelezny, Kuzelka - Lifted Relational Neural Networks

Diploma Thesis Supervisor: Ing. Gustav Šourek

Valid until the end of the summer semester of academic year 2016/2017

prof. Ing. Filip Železný, Ph.D.
Head of Department

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 11, 2016

Acknowledgement

I would like to thank my advisor Gustav Šourek, for his patience and guidance while getting familiar with the theory behind Lifted Relational Neural Networks, its computational engine and advices concerning writing this thesis. An acknowledgment belongs also to all the people from the neural-symbolic community which have responded my questions in that field.

Besides, I would like to thank all the family members, dear ones and fellow colleagues for all kinds of support while studying at Czech Technical University in Prague and Katholieke Universiteit Leuven.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague May 27, 2016 .....

Poděkování

Rád bych poděkoval mému vedoucímu Gustavu Šourkovi za jeho trpělivost a vedení během seznamování se s Lifted Relational Neural Networks, jejich výpočetním modelem a rady týkajících se obsahu této diplomové práce. Poděkování patří zároveň i všem kolegům z komunity neurálně-symbolické integrace za jejich odpovědi v otázkách týkajících se této domény.

Současně bych rád poděkoval celé rodině, blízkým a kolegům za všechny projevy podpory během studijních let na Českém Vysokém Učení Technickém v Praze a Katholieke Universiteit Leuven.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze 27. 5. 2016 .....

Abstract

Artificial neural networks are nowadays widely exploited in many research areas ranging from object and speech recognition to machine translation. Their applications vary from helping visually impaired people to recognize surroundings scenes, operating self-driving cars, or playing Go on grand master level. These successful applications are typically based on combinations of neural networks with other techniques.

Even though neural networks are used in so many areas, little work has been done regarding the theory behind learning of their structures. The aim of this thesis is to firstly gain knowledge on structure learning used in standard neural networks and the domain of neural-symbolic integration, which is a field encompassing hybrid approaches combining neural networks and mathematical logic. Secondly, to use this knowledge to design a principal extension of the existing propositional structure learning techniques to the first-order logic based neural-symbolic integration approaches.

As a result, several existing structure learning methods were selected, deeply elaborated, and extended for the use in a method combining first-order predicate logic and neural networks. Performed experiments displayed improvements over the original method on selected datasets.

Keywords

neural-symbolic integration, structure learning, lifted models, relational learning

Abstrakt

Neuronové sítě jsou dnes hojně využívány v mnoha výzkumných oblastech sahající od rozpoznávání objektů a textu až k automatickému překladu. Jejich aplikace pokrývají rozpoznávání okolních scén pro lidi se zrakovým postižením, přes řízení autonomních aut, po hraní Go na velmistrovské úrovni. Tyto úspěšné aplikace jsou typicky založeny na kombinaci neuronových sítí s jinými přístupy.

Přestože jsou neurální sítě používány v tolika oblastech, teorie jejich strukturního učení nebyla příliš rozvinuta. Cílem této práce je nejprve získat znalost strukturního učení ze standardních neurálních sítí a z domény neurálně-symbolické integrace, což je oblast, která zahrnuje hybridní přístupy kombinující neuronové sítě a matematickou logiku. Za druhé se zaměřujeme na použití těchto znalostí k návrhu principiálního rozšíření existujících strukturních učících přístupů z výrokové logiky do predikátové logiky prvního řádu v přístupech neurálně-symbolické integrace.

Několik existujících metod strukturního učení bylo vybráno, popsáno, otestováno a rozšířeno pro použití s metodou kombinující predikátovou logiku prvního řádu a neuronové sítě. Provedené experimenty ukazují zlepšení výsledků oproti původní metodě na vybraných datasetech.

Klíčová slova

neurálně-symbolická integrace, strukturní učení, pozdvihnuté modely, relační učení

Contents

Abbreviations

ANNs artificial neural networks
CNF conjunctive normal form
FOL first-order predicate logic
GA genetic algorithm
ILP Inductive Logic Programming
LP linear program
NSI neural-symbolic integration
RBF radial basis function
RBFNNs radial basis function neural networks
RL relational learning

1. Introduction

– "A machine cannot think, can it?"
– "If you describe me exactly what it is that the machine cannot do, I can always construct a machine that does exactly that."

John Von Neumann, Princeton, 1948

1.1. Motivation

Artificial Neural Networks (ANNs) a a framework which, especially when combined with other techniques, can produce interesting results in a wide range of tasks. The main motivation for this thesis is an extension of ANNs with capabilities similar to those provided by relational learning (RL), sometimes also called Inductive Logic Programming (ILP) [1].

Let us imagine that we are having a set of chemical compounds with labels denoting whether a given compound, represented as a atom to atom binding structure, is carcinogenic. Our job is to develop a classifier providing an answer to a question *Is the given compound carcinogenic?*. How to approach such a task with an attribute-value classifier? We may create a fixed-length vector representation of these structures to be employed by a standard attribute-vale machine learning model, such as SVM. But we will never be able to fully describe all the possible variations of the compound structures within this fixed-length vector. For a more interested reader we refer to [2].

One approach capable to effectively solve the presented situation is relational learning, where a logic program stands as a model for the learner. There is a huge gap in expressiveness between the former and the later; it is similar to the gap between propositional and first-order predicate logic (FOL) section 2.1, while relational learning methods typically use FOL to describe the inputs as well as the learned model itself.

The aim of this thesis is to voyage undiscovered areas of neural-symbolic integration (NSI) with focus on structure learning of neural networks within this domain, on both propositional and FOL level. NSI is a field combining all kinds of logics with neural networks. At the moment, there are several NSI methods combining propositional logic with ANNs, but there are very few utilizing FOL. Moreover, total majority of structure learning approaches within NSI focus purely on propositional logic. Thus, the novelty of this thesis lies in transferring of these existing propositional-level structure-learning approaches to the more expressive level of first-order logic.

1.2. Problem formulation

The problem is formulated as follows: investigate possible enhancements of first-order predicate neural-symbolic integration approaches with structure learning.

Structure learning in propositional NSI reduces to adding or removing set of neurons or edges. In first-order predicate NSI, structure learning is principally a relational learning task, because a structure of a neural network is given by a first-order predicate theory. There are several ways of constructing or extending these theories, for example by using crisp relational learners [1]. Instead of using crisp learners, we decided to investigate already known propositional NSI approaches and transfer them to the first-order predicate level. We further denote this process as *lifting*.

1.3. Thesis structure

This thesis is structured in the following way: this introduction chapter (Chapter 1) is followed by Chapter 2 which contains basic definitions and brief theoretical foundations of concepts used in this work. Although some of the definitions are clear, some of them are adjusted for the sake of a more economical text. Chapter 3 reviews literature in respective domains of the thesis.

The rest of the thesis can be divided into two main branches: a propositional and a first-order predicate one. Chapter 4 describes propositional NSI methods which were implemented within the thesis and experimentally compared in Chapter 5. Chapter 6 defines LRNNs, a method combining FOL and ANNs, on which the proposed lifted structure learning approaches are experimentally compared in Chapter 9. These lifted approaches are proposed in Chapters 7 and 8. Finally, Chapter 10 summarizes results and propose interesting ideas found during the thesis development.

Appendix A contains implementation's notes for methods from Chapters 4 and 8. Appendix B contains brief settings used in presented experiments. Appendix C serves as a brief guide for attached DVD.

2. Theoretical foundations

This chapter contains concept used in this thesis together with several definitions for the sake of a more economical text. Some of the used definitions were taken from [3].

2.1. Logic

A logic is a mathematical tools used e.g. for modeling situations and proving facts within these situations. In this thesis, propositional and first-order predicate logic is used. We restrict ourself to only function and negation free logic, which is denoted as *first-order* in the text.

A first-order logic theory is a set of formulae formed from constant, variables and predicates. Constant symbols (e.g. *adam*) represent object in the domain. Variables (e.g. X) ranges over the domain's objects. Predicate symbols (e.g. *sibling*) express relations among objects in the domain of their attributes. A *term* may be either constant or variable. An *atom* is a predicate symbol applied to a tuple of terms (e.g. $sibling(X, adam)$); note that a predicate symbol with arity zero will be written without parenthesis. Formulae are constructed from atoms using logical connectives and quantifiers. A *ground term* does not contain any variable. A *ground atom* contains only ground terms as arguments. A *literal* is an atom or its negation. A clause is a universally quantified disjunction of literals, which will not be written in the thesis because only those clauses will be used in the thesis. A *definite clause* has got exactly one positive literal. A definite clause with no negative literals is called a *fact*. A definite clause $h \Longleftarrow \neg b_1 \vee \cdots \vee \neg b_k$ can also be written as an implication $h \Longleftarrow b_1 \wedge \cdots \wedge b_k$. The literal h is then called *head* and the conjunction $b_1 \wedge \cdots \wedge b_k$ is called *body*. Definite clauses, which are not facts, are *rules*.

Given a first-order logic theory, the set of all ground atoms which can be constructed using the constants and predicates present in the theory is its *Herbrand base*. A *Herbrand interpretation*, assigns a truth value to each possible ground atom from a given Herbrand base. A set of formulae is satisfiable if there exists at least one world in which all formulae from the set are true; such a world is its *Herbrand model*. A satisfiable set of definite clauses has a least Herbrand model and this model is unique. The least Herbrand model of a function-free set of definite clauses (i.e. a Datalog theory) can be constructed in finite number of steps using the *immediate-consequence operator* [4]. Immediate consequence operator T_p maps the space of Herbrand interpretations over some Herbrand base $\mathcal{B}$ back to itself as $T_p : \mathcal{I}(\mathcal{B}) \mapsto \mathcal{I}(\mathcal{B})$. The mapping of T_p is directly prescribed by the theory $\mathcal{P}$ such that for $I \in \mathcal{I}(\mathcal{B})$ the $T_p(I) = \{h | (h \leftarrow b_1 \wedge \cdots \wedge b_k) \in \mathcal{P}\}$ and $b_1 \wedge \cdots \wedge b_k \subseteq I$. In other words, the operator T_p expands the current set of true atoms (interpretation I) with their immediate consequences as prescribed by the rules in $\mathcal{P}$.

Propositional logic can be seen as a special case of function and variable free first-order logic with predicates arities res arities.

An *initial* or *background theory* denotes a theory given to a method as an input. While denoting predicates without assigned variables or constant to the arguments, Prolog-

like notation denoting name and arity of a predicate will be used, e.g. $sibling/2$. By a *fresh variable* is meant a variable that does not occur in a given clause or set.

Function $pred(S)$ returns set of used predicates in a given set of clauses S. Function $|S|$ is overloaded for two cases. It returns a number rules in a given set of clauses S, or an absolute value for a real number S.

2.1.1. $\lambda\kappa$ notation

A translation process of a logic theory, a rule set in other words, into neural networks uses two types of nodes – $\wedge$ and $\vee$. To simplify notation and the transfer process, λ-κ notation is used.

Thus, two types of predicates are defined – λ and κ expressing $\wedge$ and $\vee$ respectively. This notation will be used in the rest of this thesis for clear readability which type of predicate is used. If there is a λ predicate inside a rule's head, then we say that the rule is λ *rule*; similar we say κ *rule* for the opposite case. Restrictions given over λ and κ are following: a λ predicate can be used only as a head of a λ rule or inside a body of a κ rule. Contrary, a κ predicate can be used only as a head of κ rule or inside a λ rule.

Example 2.1. Having defined $\lambda - \kappa$ notation, let us have two rules

$$\begin{aligned} foal(A) &\Longleftarrow parent(A,P) \wedge horse(P) \\ foal(A) &\Longleftarrow sibling(A,S) \wedge horse(S) \end{aligned}$$

which are translated into $\lambda - \kappa$ notation as

$$\begin{aligned} \kappa_{foal}(A) &\Longleftarrow \lambda_{foal_1}(A) \\ \kappa_{foal}(A) &\Longleftarrow \lambda_{foal_2}(A) \\ \lambda_{foal_1}(A) &\Longleftarrow \kappa_{parent}(A,P) \wedge \kappa_{horse}(P) \\ \lambda_{foal_2}(A) &\Longleftarrow \kappa_{sibling}(A,S) \wedge \kappa_{horse}(S) \end{aligned}$$

There are two more restriction over those predicates. λ predicate can be implied by only one rule, which can have multiple literals in its body. κ predicate can be implied by multiple bodies, but each body must contain exactly one literal.

Moreover, we define *base predicate* which is predicate that occur in given set of samples; mathematically this function $pred$ is used to return set of predicates that occur in given theory; e.g. $pred(\{sibling(jan, marcela), female(marcela), male(jan)\}) = \{sibling/2, female/1, male/1\}$. Correspondingly, *lambdas* and *kappas* are functions returning only set of λ and κ predicates respectively, given a theory.

We say that rule is *terminal* if each predicate inside its body is a base predicate. The *longest path* of a predicate is defined as follow: if the a predicate is terminal, then the value is zero, otherwise the value is given by 1 plus the maximal value of longest paths of predicates implying the original one.

2.2. Relational learning

Relational learning, sometimes also called as Inductive Logic Programming [1], is a machine learning area, which aims to learn a first-order logic program that would solve a problem. The main advantage of this approach is better capability of learning models concerning more expressive level than propositional level.

2.3. Artificial Neural Networks

An artificial neural network (ANN) is a biologically inspired mathematical model, consisting of interconnected processing units called *neurons*, each of which is associated with an activation function $g_i \in \mathcal{G}$ from some predefined family of differentiable functions. Neural network then defines a mapping $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ of input space to target space vectors, parameterized by a set of weights $w_{ij} \in \mathbb{R}$. By adapting weights, a process that is referred as *weight learning*, of connections (*edges*) of a neural networks, its mapping can be learnt to approximate some target function $t : \mathbb{R}^m \mapsto \mathbb{R}^n$. All methods used in this thesis uses some variant of stochastic gradient descent or Backrpopagation with momentum [5]. *MSE* represents mean squared error, widely concept used in weight learning processes.

We say that a neural network is composed from three type of layers and neurons – input, corresponding to input data, hidden, and output, corresponding to output values. Neurons form a architecture of a network, which can be also referred as structure or topology. By *output weights* or *connections* only edges leading to output nodes are meant. We say that an edge is an oriented connection from a source to target (both being neurons). Given two adjacent, neighboring, layers of a neural networks, we say that the one, closer to the input layer, is a *previous* layer w.r.t. the second; contrary, we say that the layer, closer to the output layer, is the *next* one w.r.t. to the other.

For the sake of readability, we define several operations with ANNs that are executed automatically. If the very last neuron of a layer is removed, the whole layer is removed from a neural network. If an edge is added to the network without any note of its weight, a random weight is assigned to the edge.

2.4. Threshold classifier

Neural networks are evaluated on binary classification problems in this thesis. In order to decide whether a network's output is positive or negative, e.g. the first or the second class, a threshold classifier is needed. The threshold classifier is aimed to produce a threshold, within range from zero to one (because a network's outputs is produced by a sigmoid function), to achieve the best possible accuracy on train samples.

To state the classification explicitly process, after each weight learning phase is done, e.g. by Backrpopagation, a new threshold is learned to be coherent with just learned model. The learning process of the threshold is straightforward: network's outputs values, corresponding to a train samples set, are sorted and each value between these two adjacent values, together with zero and one, is investigated. A sample below this value is classified as negative; otherwise it is classified as positive. A value producing the best accuracy is selected as the final threshold.

Considering this threshold classifier, two well known concepts are to be defined. An negative sample, classified as a positive one, is said to be *false positives*. Contrary, a positive sample, classified as a negative one, is said to be *false negatives*.

2.5. Search

Generally speaking, a search is a process of iteratively expanding states by a successor generator function, starting from an initial state, aimed to find a solution satisfying some criteria, e.g. accuracy high enough. In this thesis, two types of searches is used – local and beam search. The local search, used here, picks only the best, with respect

to the accuracy value, successor among its generated children. Contrary beam search traverses the search space, in other words expanded states, by prioritized order of a heuristic values. Moreover, beam search does not expands all of its successors but only a part of them. Its priority queue, where expanded successors are stored, has limited length. Thus, if there are more successors in the queue than its limit, the ones with lowest heuristic values are thrown away.

2.6. Genetic algorithms

Genetic algorithms (GA) is a computational model, inspired by natured, using multiple individuals that correspond to states in parallel search. Individuals, state's successors in other words, are generated by two operations called *mutation*, which represents generation of a successor state from a given one, and *crossover*, which corresponds to mixing properties of two states, producing one or two new states. Genetic algorithms uses a process called *selection* to decide which individuals will be transfered to the next search loop, called *population*. Tournament selection was used within all experiments in this thesis [6].

3. Related work

This chapter contains survey of literature across NSI and ANNs fields. Both NSI and structure learning within NSI and ANNs are reviewed in separate sections for better readability.

One of the initial goals was to make a survey of NSI approaches across exhaustive number of experiments. But there are two obstacles in doing so. Firstly, only a few executable NSI approaches were found; after dozens of emails to NSI methods' authors, only four of them were found in a workable state. Secondly, NSI community still lacks unified set of benchmarks and selecting appropriate ones would take time and effort that was put into extension of current techniques.

3.1. Propositional neural-symbolic integration

The very fist combination of logic and ANNs dates back to [7]. Since then, the field has been investigated in multiple ways, even extended to FOL case as it is reviewed in the following section. Several methods arose from the transition process of propositional rules to neural network structure described in [7].

Knowledge-Based Neural Networks (KBANN) [8, 9], a successor of earlier *EBL-ANN* [10], is based on the idea of translating a set of non-recursive propositional rules (*initial theory*) into neural network structure; thereafter weighs of such network are learned. The initial rules may be produced by an expert or by a random generator. This approach outperformed other NSI methods, which do not possess initial background knowledge [9]. KBANN was extended in multiple ways by capability of processing a set of recursive finite-state grammars [11], improving *rule to network* translation [12], improving a network's structure by adding neurons to mimic rules addition to an already constructed network (*TopGen*) [13] or by genetic algorithm search for better structure (*REGENT*) [14]. Last known successor of KBANN is *INSS* [15, 16], which contains the whole neural-symbolic cycle. It works by mining rules from a neural network, inserting them back into the network structure in case that an expert evaluates these found rules as helpful. In spite of this, the process is not fully automatic.

KBCNN [17] is very similar to KBANN in its network construction procedure. The main discrepancy with KBANN lies in the used family of activation functions. Meanwhile KBANN and its extensions uses logistic sigmoid in most of the cases, KBCNN uses *certainty factor* function [18], based on the MYCIN-like systems [19]. Drawback of this function is that it handles only binary inputs. KBCNN was extended in multiple ways resulting in *Certainty Factor Network* [18], *CLNet* [20] and its special case *Induce-Net* [21]. The first of these extension, Certainty Factor Network, does not need initial theory as KBANN.

Following the Core method [22, 23], *Connectionist Inductive Learning and Logic Programming System* (C-IL^{2}P) [24] was the one of the first systems realizing neural-symbolic learning cycle. The method takes an initial knowledge as KBANN, but it allows recurrent connections within a network, but having these backward weights fixed to 1. C-IL^{2}P also differs with the former by the rule to network translation approach.

The later produces always a three layers network to overcome possible problems of a huge network's structure. They claim that KBANN is not able to learn efficiently, because of vanishing gradient and similar properties while having huge network in sense of numbers of layers.

In [25] *ERANN* system was presented, which aimed to both rule extraction and network construction at the same time. Moreover, some pruning techniques are used to simplify the rule extraction phase.

Enormous work has been done in the field of rule extraction from neural networks. For interested readers in this area, we recommend the survey part of [26].

3.2. First-order predicate neural-symbolic integration

Several methods, each based on different idea, were developed in aim to connect ANNs and FOL. Unfortunately, only very little of them focus on structure learning. Some of the approaches used encoding of logic to numerical values; but such a mapping is hard to be preserved, e.g. to have two formulae that differ very little in both FOL and numerical encoding is not an easy task.

FONN [27, 28] is a method having a four-layer network representing only flat theory with negations and function; it is based on RBFNNs. Its aim is to represent and learn an existential quantifier semantics and fuzzy logic.

CILP++ [29, 30] was created by integrating *Bottom Clause Propositionalization* (BCP) with C-IL^{2}P. In short, BCP generates bottom clauses of samples to transform them to propositional logic. A subset of initial theory, here called *background knowledge*, can be used to initialize structure and weights of the network.

Relational Neural Networks (RelNNs[1]) [31, 32] is a feed forward network with recurrent components, based on the scheme of relational database from which data come from. The aim of this approach is to learn pattern from the data as well as statistical information of the data described by a pattern. Later, RelNNs were enhanced with *Cascade Correlation* [33], which together gave birth to *Aggregation Cascade Correlation Networks* (ACCN) [34, 35]. So, the power of ACCNs is that it also builds structure of a neural network; RelNNs itself is incapable of that. Later, it was seen as a special case of *GNN* [36, 37].

PAN [38] system can learn first-order relations from an initial theory and a set of samples with output predictions. It is based on several types of modules that act as a neurons inside a network. Structure of these modules can be arbitrary, e.g. one can be represented by a recurrent network. Invertible encoding was developed in order to use PAN.

In [39, 40] method based on Topos theory [41] was presented. The idea is to translate FOL to a variable-free representation, from which homogeneous equations function can be generated. Then, a neural network is learned by these equations. Similarly, *FOLNN* [42] constructs three layer feed forward neural network from an initial theory and samples, which is thereafter learned in spirit of *multi-instance learning* [43, 44, 45].

FineBlend system [46, 47, 48] was developed as another connectionist system for an acyclic logic program and for some level of accuracy. This method is the only one NSI method, to our knowledge, capable of structure learning in sense of removing and adding neurons to increase flexibility of the learning process. The main difference [49] and between [46] is that the first one uses *fibring neural networks* [50]; the goal

[1] They were originally referred as *RNN*, but in spite of Recurrent Neural Networks, we used this shortcut.

of approximating an acyclic logic program is common in both approaches. Similar approach to FineBlend was proposed in [51].

There has been other methods developed in this field, for example based on first-order abductive inference [52], first-order reasoning system [53, 54, 55, 56], later enhanced by structure learning [57], and its more expressive version based on *bottom-up reduction calculus* [58]. In spirit of theorem proving and unification connected to ANNs, there are several works, e.g. approximation of least general generalization [59, 60], SDL resolution [61] or unification [62, 63, 64]. An extension of connectionist system to handle multi-valued logic was investigated in [65].

3.3. Structure learning in ANNs approaches

Construction of neural networks' structures can be categorized in the following way: constructive, destructive or genetic algorithms approaches, as mentioned in [66]; there is a possibility of someone else's earlier categorization of these, however, this is the earliest to our knowledge. Constructive approaches are based on searching minimal network's structure that is able to fit data within given error. These approaches start from a minimal or some small structure, incrementally expanding the network according to data. Destructive approaches reverse the previous idea, thus starting with a large network's structure, followed by iteratively pruning insignificant edges, neurons or both. Genetic algorithms may combine both approaches by augmenting and pruning during the search process.

3.3.1. Constructive approaches

This subsection contains constructive approaches, which start with a small network and gradually extend it. Some of the earliest methods, concerning this idea, were driven by the need of faster learning algorithms than Backpropagation, leaving the goal of precise network's structure behind. This approach can be faster than destructive as shown in [67].

Cascade Correlation (CasCor) [33] is well known constructive method that has been extended to numerous ANNs variants and problems [68, 69, 70, 71, 72]. The main idea is to construct a hierarchical network evolving one useful neuron correlated with outputs at the time. Moreover, instead of learning just one neuron, more candidates may be tried and the best neuron selected; which exploits search space simultaneously and mimics GMDH [73] a little bit. Similar method, driven by error curve of weight learning algorithm, is called *Dynamic Node Creation* (DNC) [74]; it also reuses weights from previous iterations. The later differs mainly by the resulting structures, only one hidden node is presented, and by addition of non-correlated neurons to outputs.

There are more constructive methods based on the same ideas, for example extending them by edge pruning techniques [75], using another weight learning methods [67] or trying to develop neurons in a systematical way by inserting layers of neuron as well [76].

Other techniques are based on linear programming (LP). One of the approach proposed an idea of constructing neural network's structures in polynomial time [77]. However, those were special cases for classification problems. The method, similar to [78, 79, 80], is based on mining a pattern describing a part of the dataset. Then, this pattern is translated and incorporated into a neural network and correctly classified examples are removed from the train set; such loop is repeated until error drops below a threshold.

3.3.2. Destructive approaches

This section investigates some destructive methods. Those methods start with large networks and aims to prune or trim them in some way to either get better error rate or simplify its structure. However, constructive methods tend to be faster than destructive; this is caused mainly by two aspects. Firstly, weight learning algorithms are computational demanding on larger networks than on smaller ones. Secondly, weight learning algorithms may not find a suitable setting of weights while given a large network, in a case that the instance need only a small. On the other hand, constructive approach does not possess any ability to remove redundant neurons or edges.

Aim of changing neural network's structure has also been investigated by the mean of incorporating these changes directly into a weight learning algorithm, e.g. Backpropagation. Two of these methods are *successive learning with forgetting* (*SLF*) [81, 82] and *MLP2NL* [83]. The common goal is to enhance the objective function of a weight learning algorithm by a term describing penalization of a weight from a predefined value. In the work of [84] similar methodology is introduced; in this case, any arbitrary weight can have predefined value, which represents its attractor. Forgetting was used as the key part in SLF; we note it here although it was primarily aimed to simplify the network architecture to be faster while extracting rules from a network. Similarly, ERANN uses penalty functions responding negatively to weights having large or small, almost zero but non-zero, values. After the weight learning phase, edges with zero weights are removed as well as neurons having no incoming nor outgoing edges.

Another method developed to simplify network structure is based on pruning the least relevant neuron [85] or the least sensitive change in error [86] in a network. A pruning method capable of removing single neuron or edge is presented in [87].

3.3.3. Genetic algorithms

Only few approaches, from the area of GA, are presented here, because we did not investigate this area much in depth; this was mainly caused by our requirement of interpretability of made changes during the structure learning process. One, previously described, member of this group is REGENT. An evolutionary programming based approach called *EPNet* is investigated in [88]. A well known GA approach for constructing ANNs' structures is *Group method of data handling* (GMDH) [89]. GMDH is data-driven approach which constructs the network in order to minimize given criterion. Another well known GA approach is *Hypercube-based Neuroevolution of Augmenting Topologies* (HyperNeat) [90]. HyperNEAT is a neuroevolution based search evolving a CPPN, which constructs the network.

3.3.4. Others

The idea of escaping from a local minimum, in the weight learning phase, gave birth to a method that inserts rules into the learnt network [91]. This does not fit into structure learning directly. Although, they have claimed the first of such experiments, the superiority of architecture based on prior knowledge over a randomly generated one is well known since [8]. Also, the approach needs another source of rules, either an expert or a software producing rules. On the other hand, they found that inserting a rule, which is done by adding a proper set of neurons and weights, may increase accuracy on examples not described by the rule as well. There are other works concerning structure learning in different domains, e.g. regression neural networks [92].

4. Structure learning approaches for propositional neural-symbolic integration

The main goal of this thesis is to transfer already known structure learning algorithms from propositional NSI and standard to first-order predicate level. Our first subgoal is to compare structure learning approaches in the field of propositional logic, in order to gain some directions and experience. This chapter describes selected algorithms from that scope in depth; their comparison is done in Chapter 5. Besides that, this chapters also servers as a theoretic part for method implemented in our propositional testbed; e.g. main parameters are explained here. Although one may only cites original papers of each algorithm, we decided to make detailed description because some sources are not explicitly defining every used parameter. Also, our implementations may differ a little bit from original sources, yet original sources' functionalities are preserved.

For the comparison, we selected five constructive and one destructive methods. Firstly, constructive approaches based purely on numerical values of neural networks are described (sections 4.1 and 4.2). Secondly, approaches based on propositional rules are described (sections 4.3 and 4.4), together with a genetic algorithm (section 4.5). Finally, the chapter ends with one destructive algorithm (section 4.6) based only on changing the optimization criterion of weight learning algorithms. We restrict ourself to only feed forward neural networks; there are many reasons for that. Firstly, it is always better to start with simpler task. Secondly, feed forward neural networks are capable to approximate every function. Thirdly, there bigger potential of using non-recurrent version, since there are very few first-order predicate NSI method using recurrent connections; thus application of such extension would be limited.

4.1. Dynamic Node Creation

Weight learning algorithms, for example Backpropagation, can struggle while learning weights of a huge network given a simple problem. One option to solve such a problem is to start with a small network and in repetition learn weights and extend the network by some neurons in that case that the error after the weight learning phase is not small enough. Based on the universal approximation theorem [93], *Dynamic Node Creation* (DNC) [74] works in this manner using and gradually extending only single hidden layer.

The method starts with a three layer neural network having a single hidden neuron. DNC gradually extends the hidden layer by a neuron each time the error curve of weight learning process flattens. Each hidden neuron possess one incoming edge from each input neuron and a bias, and one outgoing edge for each output neuron; other connections are forbidden. Each new edge's weight is randomly initialized. Weights from previous weight learning phase are used fo the next phase to boost the search, instead of random reinitialization. The method terminates when mean squared error and maximal squared error, over samples, are below user specified thresholds.

The rest of this section contains description of DNC method in details. The algorithm is shown in Algorithm 1. Firstly, network with one hidden neuron, input and output

layers corresponding to samples is created (line 1). Thereafter, a cycle of weight learning phase (line 4) is followed by addition of a new neuron (line 8), in the case that a neuron limit has not been reached; otherwise the cycle runs until stopping criterion is met. Note that limit of hidden neurons is our extension. The stopping criterion is satisfied if maximal squared error is below C_m threshold and also mean squared error is below a threshold C_a (line 5).

One can see that weight learning algorithm takes more arguments than a neural network and a set of samples (line 4). The two next arguments w and Δ_T ensure that the weight learning terminates when the error curve flattens. The method returns squared errors on given samples after the last epoch. Flattening is detected when there has been at least w epochs and equation (4.1) holds; where m_i stands for mean squared error at time, or epoch, i, t stands for current time index. Time index 0 stands for first epoch and Δ_T is a triggering value that is provided by a user.

$$\frac{|m_t - m_{t-w}|}{m_0} < \Delta_T \tag{4.1}$$

In the original paper, DNC uses Backpropagation as weights learning algorithm. In the pseudocode, we changed notation to empathize that arbitrary weight learning algorithm may be used.

Algorithm 1: Dynamic Node Creation

Input: Set of samples *samples*, desired mean squared error C_a, desired maximal squared error C_m, limit of added neurons *neuronsLimit*, width of time window w and trigger slope Δ_T

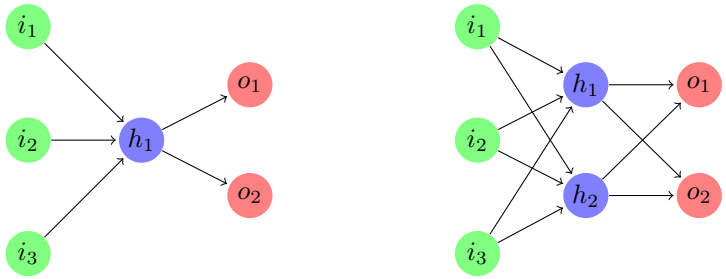
Output: Learned neural network

```
1  network ← constructNetwork(samples)
2  hiddenNodes ← 1
3  while true do
4      errors ← weightsLearningAlg(network, samples, w, Δ_T)
5      if max(errors) ≤ C_m ∧ mean(errors) ≤ C_a then
6          break
7      else if hiddenNodes < neuronsLimit then
8          addHiddenNode(network)
9          hiddenNodes ← 1 + hiddenNodes
10 return network
```

For illustration, Figure 4.1 shows initial network's architecture (Figure 4.1a) and its structure after adding the second hidden neuron (Figure 4.1b) on a problem with three inputs and two outputs. Note that weights are not shown as well as biases.

4.2. Cascade Correlation

Cascade Correlation (CasCor) [33] was originally developed to overcome *step size* and *moving target* problems while using Backpropagation, but we focus only on the later. The method gradually builds up neural network of neurons that should behave as useful detectors correlated with network's outputs. Since each new hidden neuron possesses one incoming edge from each input and hidden neuron, the structure mimics a cascade



(a) constructed neural network from samples with one hidden neuron

(b) neural network after adding second hidden neuron

Figure 4.1. Example of constructed networks by DNC method on dataset having three inputs and two outputs. Input, hidden and output neurons are displayed in green, blue and red color respectively. Biases and weights are not shown.

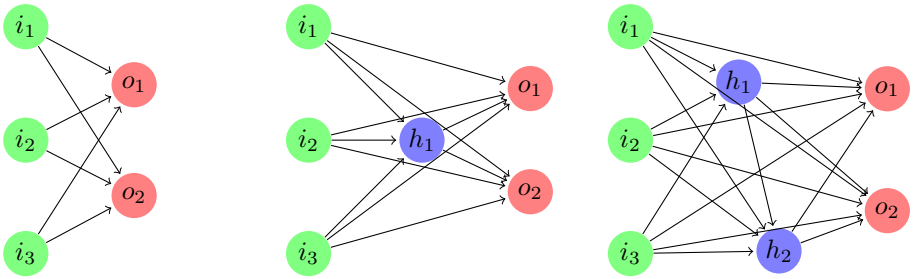
(see Figure 4.2c), thus the name. Each output neuron possesses one incoming edge from each hidden and input neuron.

In short, step size is connected to gradient optimization, where one would like to have automatically adjusting size of step instead of a constant. Several solutions exist for this kind of problem, for example momentum [5]. Cascade Correlation resolve this problem by using Quickprop [94]. Instead of using Quickprop, we use Backpropagation in all cases of weight learning, since our goal is structure learning.

The second problem is the *moving target* problem. In short, each neuron in a network would like to become a useful detector of some feature in a given problem, but the inability of communication between units while learning weights may result in moving target problem. Let us have two hidden neurons A and B at the same layer. The A is producing bigger error than B, thus the previous layer's neurons would act according to *herd effect* to compensate A's error. After A's error is below B's one, the same herd effect may arise for those previous layer's neurons – they aim to compensate B's error. This may take a long time before previous layer's neurons split to two groups, one compensating A's and the second B's error.

Now, we describe the algorithm, which is displayed in Algorithm 2, in details. The method starts by constructing a neural network with input and output layers corresponding to samples (line 1). Each output neuron possesses one incoming edge from input neuron. Thereafter, outputs' weight learning phase and a new hidden neuron generation, and addition, repeats until mean squared error is below given threshold $errorThreshold$, or the number of hidden neurons reaches given limit $neuronsLimit$. During learning of outputs weights, only edges incoming to output neurons are learnt (line 4). Originally, this was done by Quickprop, but we use Backpropagation instead. Learning only output weights means that no propagation of error is done; this fact quite speed ups the learning. The original idea is that all edges expect output ones are frozen (they are not taken into account during weight learning), which is the same thing as we describe.

Next phase in the loop is focused on generating a new neuron (line 9). The algorithm creates a number of candidates according to $candidatesSize$, each one of them can be initialized with different activation function, incoming edges or weights. These neurons, called *candidates*, can then be trained in parallel to maximize their correlation to output neurons; to be precise incoming weights of a candidate are updated to maximize its correlation. The only constrain over all candidates is that each one can have incoming edges only from already added network's input and the hidden neurons.



(a) constructed neural network from samples **(b)** neural network after adding first hidden neuron **(c)** neural network after adding first hidden neuron

Figure 4.2. Example of constructed networks by Cascade Correlation method on dataset having three inputs and two outputs. Input, hidden and output neurons are displayed by green, blue and red color respectively. Biases and weights are not shown.

Finally, one *candidate* with maximal correlation (line 12) is selected and added to the network (line 13). Weights of incoming edges to *candidate* are set up to weights learnt during the correlation maximization process. For each output neuron o, new edge with random initial weight is added from *candidate* to o. For illustration, the network construction is shown in Figure 4.2.

The one and only part, which is left to be described, is the correlation maximization process (line 10). The correlation C is given by the sum over all output neurons o of the magnitude of the correlation between residual error E_o and the value of candidate V, which is given by equation (4.4), where s stands for a sample from samples, o stands for an output neuron of the network, V_s is the value of the candidate neuron given sample s, $E_{s,o}$ is the residual output error given a sample s and an output neuron o. $\overline{V_s}$ and $\overline{E_{s,o}}$ are averaged values over samples; equations (4.2) and (4.3). In fact, equation (4.4) is not correlation but covariance, but we stick to the used naming convention.

$$\overline{V} = \frac{1}{|samples|} \sum_s V_s \tag{4.2}$$

$$\overline{E_o} = \frac{1}{|samples|} \sum_s E_{s,o} \tag{4.3}$$

$$C = \sum_o |\sum_s (V_s - \overline{V})(E_{s,o} - \overline{E_o})| \tag{4.4}$$

In order to maximize C, its partial derivations equation (4.5) have to be computed, which can then be used in delta rule to perform gradient ascent. Although Quickprop is used in the original paper, we use adjusted Backpropagation for the gradient ascent process. Speaking of equation (4.5), full description comes handy; $correlation_o$ is the correlation between the candidate neuron and an output, f'_s is derivative of the candidate's activation function given the input s w.r.t. the sum of its inputs, $I_{s,i}$ is the input received by the candidate from a neuron i given a sample s.

$$\frac{\partial C}{\partial w_i} = \sum_o \text{sign}(correlation_o) \sum_s (E_{s,o} - \overline{E_o}) f'_s I_{s,i} \tag{4.5}$$

Besides the speed up of learning candidates in pool in parallel, another speed up arises when the values of the network's neurons, except the current candidate, are cached. This is possible since weights of incoming edges to these neurons do not change (they are frozen), thus values of these neurons do not change as well.

Algorithm 2: Cascade Correlation

Input: Set of samples *samples*, desired mean squared error *desiredError*, limit of added neurons *neuronsLimit* and number of candidates *candidatesSize*

Output: Learnt neural network

```
1  network ← constructNetwork(samples)
2  hiddenNodes ← 0
3  while true do
4      MSE ← learnOutputWeights(network, samples)
5      if hiddenNodes ≥ neuronsLimit ∨ MSE ≤ desiredError then
6          return network
7      candidates ← ∅
8      for i ∈ 1 ... candidatesSize do
9          neuron ← generateCasCorNode(network)
10         maximizeCorrelation(neuron, network, samples)
11         candidates ← {neuron} ∪ candidates
12     bestCandidate ← argmax_{candidate ∈ candidates} correlation(candidate)
13     addCascadeHiddenNode(bestCandidate, network)
14     hiddenNodes ← 1 + hiddenNodes
15 return network
```

4.3. Knowledge Based Artificial Neural Networks

One of the first hybrid approach in neural-symbolic integration is *Knowledge Based Artificial Neural Networks* (KBANN) [8, 9]. The method constructs a neural network's structure from an acyclic propositional theory, then it uses arbitrary weights learning algorithm (Backpropagation in original paper). The network structure is created in a way preserving logical conjunctives ($\wedge$, $\vee$); this translation mechanism was originally presented in [7]. They showed that such approach outperform purely connectionist and symbolic systems. Firstly, the method's pseudocode is sketched. After that, the KBANN's construction process is described, since it is the most important idea behind the method.

The original system from [9] is capable of processing besides binary, also nominal, hierarchical, linear and ordered features but these extensions were not implemented, thus their are not discussed. In [8], they proposed and tested a way of inserting rules to an already constructed KBANN network as an option to escape from local minimum. Even though the process is trivial, the rules should be provided by an expert, which is not always available.

The method's pseudocode is shown in Algorithm 3. Firstly, if given propositional theory is cyclic , the algorithm terminates with *null* (line 2). Secondly, the network is constructed from the given dataset and rules (line 3). Finally, the network's weights are learnt by an arbitrary algorithm (line 4).

The rest of this section deals with the construction of a KBANN network given rules, samples and interpretation parameter ω. We describe all steps of the process on a running example. Let us suppose, for the running example, that we have a dataset with three inputs a, b, c and two outputs x and y. Given theory is described by $T = \{x \Longleftarrow a, m \Longleftarrow a, m \Longleftarrow \neg a \wedge b, p \Longleftarrow b, w \Longleftarrow a \wedge p\}$.

Firstly, the given theory is transformed so that each atom a_{head}, being implied by

Algorithm 3: KBANN

Input: Set of samples *samples*, neuron interpretation parameter ω and initial rules *rules*

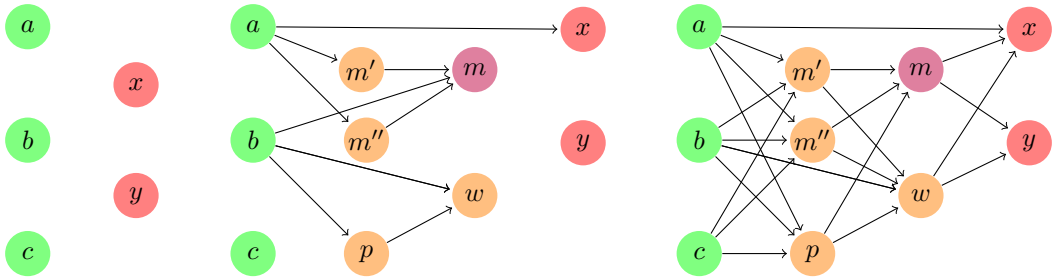
Output: Learned neural network or *null* given theory is cyclic

```
if isTheoryAcyclic(rule) then
    return null
network ← constructNetwork(rules, samples, ω)
weightLearningAlg(network, samples)
return network
```

more than one rule in the theory, is replaced by a fresh different atoms a_i in those rule definitions and set of new rules in the form $a_{head} \Longleftarrow a_i$ for each original rule definitions. This transformation roots from the inability of a single neuron to behave as a head of multiple rule definitions at the same time; this is the same reason why λ and κ notation were presented. For example, take rules $m \Longleftarrow a$ and $m \Longleftarrow \neg a, b$. The heads in those rules will be renamed, $\lambda_{m'} \Longleftarrow a$, $\lambda_{m''} \Longleftarrow \neg a \wedge b$, and two new rules will be added $\kappa_m \Longleftarrow \lambda_{m'}$ and $\kappa_m \Longleftarrow \lambda_{m''}$. This transformation preserves $\wedge$ and $\vee$ behavior given at the beginning, because, as one can see, m' and m'' (a_i in general) preserves their $\wedge$ behavior and m (a_head) preserve its $\vee$ behavior. Thus theory T after transformation will look like $T' = \{x \Longleftarrow a, \lambda_{m'} \Longleftarrow a, \lambda_{m''} \Longleftarrow \neg a \wedge b, \kappa_m \Longleftarrow \lambda_{m'}, \kappa_m \Longleftarrow \lambda_{m''}, p \Longleftarrow b, w \Longleftarrow a \wedge p\}$. To be clear, λ and κ were used here for illustration; in the propositional level, there is no restriction over their bodies and occurrences in heads.

Secondly, the network with corresponding input and output neurons to the dataset is created; this is shown in Figure 4.3a. Then, the transformed rules T' are added to the network so that each neuron corresponding to a rule head is in the closest layer to the input layer, preserving the non-recursiveness. In other words, layer index of a neuron is given by the maximal layer index of its predecessors plus one; in case that all predecessors are input neurons, the neuron is in the first hidden layer. For each pair body atom b_a implying a head atom h_a, there is an outgoing edge from b_a to h_a with weight 1, or -1 in case that b_a is negated literal in the body. The last step of this network construction phase, is adjusting biases of neurons to preserve the $\wedge$ and $\vee$ behavior. For each $\vee$ neuron (we remember them from the transformation process), its incoming edge's weight from bias is set to $-\frac{\omega}{2}$. For each $\wedge$ neuron (all others), its incoming edge's weight from bias is set to $-\frac{\omega l}{2}$, where l is number of literals in body of rule implying the head atom represented by the neuron. The idea, on which, those biases' setting is created, arisen from the requirement of activation of a neuron when at least one predecessor, and all predecessors are active to simulate $\vee$ and $\wedge$ behavior respectively. So, at the end of this construction, the network constructed upon T' is shown figure 4.3b.

Thirdly, for each two neurons from adjacent layers, a feed forward edge with a zero weight is added in case that there is no edge between those neurons. Finally, each weight is perturbed by a small, randomly generated number; this is done to preserve the problem connected to symmetry in neural networks [95]. Structure of the final KBANN network is shown in Figure 4.3c.



(a) initial KBANN network constructed from inputs and outputs

(b) KBANN network constructed from transformed rules

(c) final structure of KBANN network

Figure 4.3. Example of constructed networks by KBANN method on the running example. Input, output, ∧ and ∨ neurons are displayed by green, red, orange purple and color respectively. Biases and weights are not shown.

4.4. Topology Generator

KBANN only constructs single neural network from an initial theory, as described in the previous section. It does not possess any structure learning at all. Focused on the topology (structure) of KBANN's network, [13] developed a heuristically driven search called *Topology Generator* (TopGen), which aims to insert neurons into a KBANN network. The novelty of this approach lies in the neurons insertion itself – adding a neuron should preserve the same meaning as extending propositional theory by a rule (or modifying a subset of the theory).

TopGen uses beam search with a cutoff on maximal length of open list (see section 2.5), which is not as interesting. TopGen was designed to insert new neurons into a KBANN network, since KBANN is capable only of removing antecedents from existing rules, e.g. zero weight on particular edge, but unable to induce new rules into the network. The neuron extension method arose from the assumption that in large neural networks an output value of a hidden neuron is always close to its maximum or minimum, thus a neuron's activation function can be seen as a step-wise. Based on this, TopGen classifies whether a hidden neuron is false positive or false negative with respect to the network's output values. The one and only unclear thing from [13] is the classification process of a hidden neuron's behavior to a step function. So, we have set up boundaries for which a hidden neuron is classified as inactive or as active; e.g. for sigmoid function – inactive and active for the output values ranging from zero to 0.1 and 0.9 to one respectively. TopGen firstly splits a dataset to two parts, using one (train set) for weight learning and the second one (validation set) for false positive and false negative neurons classification. The main idea is to find neurons producing error because of a bad generalization on the validation dataset rather than memorization, which would be yielded by classification on train set. Thereafter, for each neuron, new neural network is constructed in order to decrease false positives or false negatives producing by that neuron.

TopGen uses beam search, starting from an initial state, which is given by a network produced by KBANN. In spite of space, we describe only the part of hidden neurons extension, because other parts were already described sections 2.5 and 4.3. Pseudocode for hidden neurons classification and extension is shown in Algorithm 4. Firstly, TopGen counts each hidden neuron with bounded activation function, e.g. from 0 to 1, a number

Algorithm 4: TopGen's hidden neuron classification and extension

Input: Set of samples *samples*, number of successors *successorsSize*, in/active parameter *inActive* and learnt network by KBANN *network*

Output: List of learnt neural networks

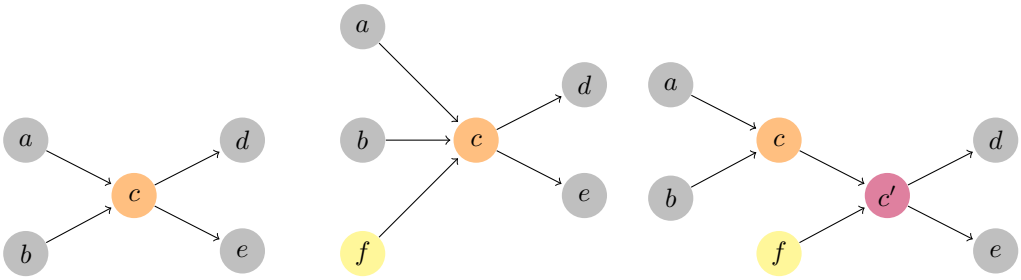
1 $candidates \leftarrow produceFPCounters(samples, network, inActive)$
2 $candidates \leftarrow produceFNCounters(samples, network, inActive) \cup candidates$
3 $sorted \leftarrow sortAccordingToCounters(candidates)$
$sorted \leftarrow takeFirstN(sorted, successorsSize)$
4 $extended \leftarrow \emptyset$
5 **for** $(neuron, isFP, count) \in sorted$ **do**
6 $\quad child \leftarrow insertNode(network, neuron, isFP)$
7 $\quad weightLearning(child, samples)$
8 $\quad extended \leftarrow \{child\} \cup extended$
9 **return** *extended*

of false positives and false negatives situation of that neuron (lines 1 and 2) over all samples w.r.t. the network outputs, given a threshold value *inActive*, which creates two intervals, e.g. from 0 to inActive and from $1 - inActive$ to 1; this is stored as a triple containing the neuron, FP or FN count and a boolean expressing whether the counter is FP or not. Then, these records are sorted according to FP/FN counters in descending order (line 3), breaking ties in favour of neurons closer to the input layer; only first triples of size *successorsSize* (line 3) are taken for later processing. For each of the selected triples, new network is constructed base on two fact – given neuron and whether the counter was FP or not. The new network is learnt by weight learning algorithm with a learning rate decay that is the product of *learningDecay* constant and learning rate which the predecessor's network was learnt with (line 7). Finally, list of such extended networks is returned (line 9).

4.4.1. Rule-like neuron insertion

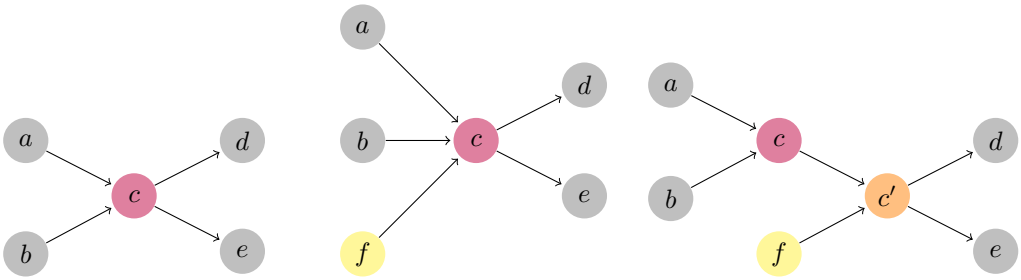
Probably the most important TopGen's part, neuron insertion, is to be explained in the rest of this section. Firstly, the algorithm needs to recognize, whether a given neuron is $\wedge$ or $\vee$ one. The problem reduces to another one: classify whether a given hidden neuron is closer to $\wedge$ or $\vee$ neuron. A $\wedge$ neuron is active only if all of its positive antecedents are active and negative antecedents are inactive. Thus its bias must be slightly less than sum of active inputs. A $\vee$ neuron is active if at least on of its antecedents is active. Thus the neuron's bias must be slightly greater the sum of negative incoming weights. Thus, a neuron is considered to be $\wedge$ in case that the neuron's bias is closer to the sum of positive incoming weights than to the sum of negative weights; otherwise it is a $\vee$ neuron.

Secondly, after the neuron classification, insertion of new rules into a network can be presented. The aim is at decrease of false positives and false negatives. If a $\wedge$ neuron produces false positives, to decrease its firing rate, let us add one antecedent to its body. This antecedent is represented by a fresh new neuron, called *base neuron*, which possess one new incoming edge for each input neuron. If a $\wedge$ neuron n produces false negatives, the aim is to increase its firing rate. Thus new neuron $\wedge$ neuron n_a is constructed. n_a takes two incoming edges – one from n and one from fresh new base neuron. All previous output connections of n are moved to be outgoing edges of n_a. See Figure 4.4 for illustrative example.



(a) original network's part before extension **(b)** FP decrease **(c)** FN decrease

Figure 4.4. Example of structure extension after detecting a ∧ neuron (Figure 4.4a) being false positive (Figure 4.4b) and being false negative (Figure 4.4c). ∧, ∨, base and arbitrary neurons are displayed by orange, purple, yellow and grey color respectively. Biases and weights are not shown.



(a) original network's part before extension **(b)** FN decrease **(c)** FP decrease

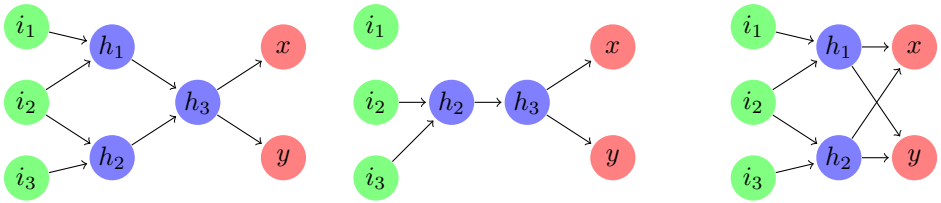
Figure 4.5. Example of structure extension after detecting a ∨ neuron (Figure 4.5a) being false positive (Figure 4.5c) and being false negative (Figure 4.5b). ∧, ∨, base and arbitrary neurons are displayed by orange, purple, yellow and grey color respectively. Biases and weights are not shown.

In the case of ∨ neuron n_o, the case is similar, but opposite. To decrease false negatives a fresh new base neuron is added to the network and outgoing edge to n_o is added. To decrease false positives, new ∧ neuron n_a is added. n_a takes two incoming weights, one from n_o and one from fresh new base neuron; all previous outgoing edges from n_o are moved, so they are outputs of n_a neuron.

The process described here is aimed to decrease firing power of neurons that are probably negatively correlated to the output. Thus, this ideas should not work on several cases, for example a problem that has go multiple outputs that are negatively correlated between each other.

4.5. Refining, with Genetic Evolution, Network Topologies

Although TopGen searches for better network's structure by gradually extending the initial one, in [14] they search for a bigger number of possibilities. They presented genetic algorithm called *Refining, with Genetic Evolution, Network Topologies* (REGENT), which is driven by the same goal – to find better network's structure corresponding to data, operating on KBANN network as well. The only parts of genetic algorithm, described in section 2.6, that are to be designed specially for REGENT are mutation and crossover operators. The algorithm uses train and validation set in the same way



(a) network before mutation **(b)** network after neuron h_1 deletion **(c)** network after neuron h_3 deletion

Figure 4.6. Example of structure changes to initial neural network (Figure 4.6c) after neuron deletion mutation on neuron h_1 (Figure 4.6b) and h_3 (Figure 4.6c). Inputs, hidden and output neurons are displayed by green, blue and red color respectively. Biases and weights are not shown.

as TopGen.

4.5.1. Mutations

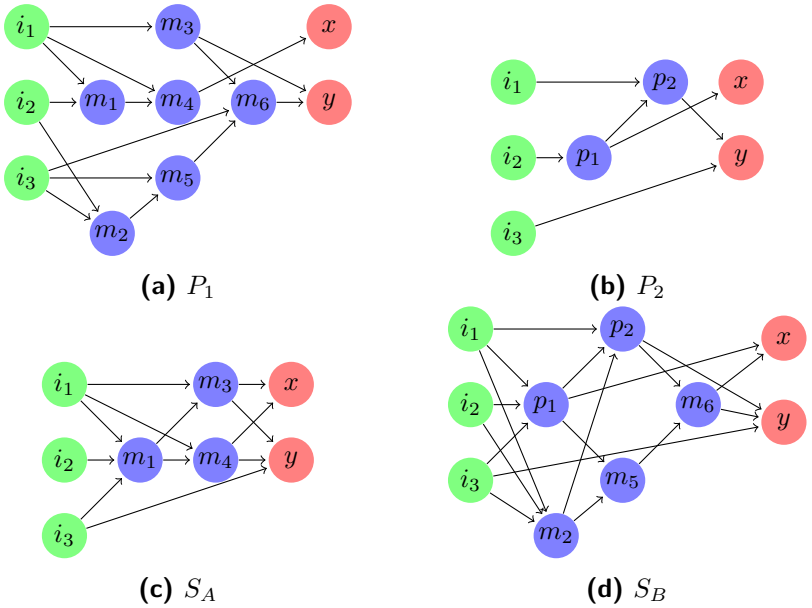
REGENT uses two mutation operators – neuron deletion and TopGen's neuron insertion. One can see the later mutation as directly using Algorithm 4 with $successorsSize$ equal to 1. The neuron deletion mutation is quite simple – randomly select a hidden neuron, then remove it from the network together with incoming and outgoings edges of the neuron. The one and only trick to resolve is the case when the removed neuron is the only one in that layer. In such a case, an edge is added for each neuron in the previous layer to each neuron in the following layer, w.r.t. the removed neuron's layer; for illustration see Figure 4.6. Otherwise the generated network could splitted to multiple disconnected components. This process is called *neuron deletion*. The initial population is constructed from the given initial theory by both of these mutations. Thereafter, during search across populations, only TopGen's rule extension mutation is used.

4.5.2. Crossover

There are multiple ways to crossover two neural networks. REGENT's crossover is based on the idea of grouping densely connected neurons into one network and adjusting neurons biases to preserve their original $\wedge$ or $\vee$ meanings. Firstly, given two parents networks, the algorithm traverses over hidden layers from input to output layer, for each parent, and splits neurons to two sets A and B. Probability that a given neuron n will occur in A is given by equation (4.6); where w_{in} stands for weight between neuron i and n. If a randomly generated number is withing given probability of n belonging to A, then n is added to A, otherwise to B. Node sets A and B are shared for both parent networks.

$$p(n \in A) = \frac{\sum\limits_{i \in A} |w_{in}|}{\sum\limits_{i \in A} |w_{in}| + \sum\limits_{i \in B} |w_{in}|} \tag{4.6}$$

From each set A and B, one new network is constructed in following way. Network's structure is constructed from aset of neurons; input and output layers are taken from parent networks and each neuron lies in layer which index is given by the longest path to the input layer (the same principle as in KBANN network's construction). For each two neurons from the same parent, which shared an edge in the parental network, the



(a) P_1 **(b)** P_2

(c) S_A **(d)** S_B

Figure 4.7. Example of crossover of two parent networks P_1 (figure 4.7a) and P_2 (figure 4.7b) producing two offspring S_A (figure 4.7c) and S_B (figure 4.7d). Offspring' indices denote original set of neuron. Nodes were splitted so that $A = \{m_1, m_3, m_4\}$ and $B = \{p_1, p_2, m_2, m_5, m_6\}$. Input, hidden and output neurons are displayed by green, blue and red color respectively. Edges' weights and biases are not shown.

edge, with the corresponding weight, is added to the newly constructed one; this holds for input to hidden neuron edges, since the input and output layers are same for all network during the REGENT run. The same holds for hidden to output and input to output edges; in such cases, an edge's weight is set to average of both edges' weights from both parents, taking zero weight in case that the edge is not presented in a parent.

Next step of construction of a offspring network lies in adjusting the biases. Output neuron's biases are computed as average over their values in both parents. In hidden neurons, the process is little bit complicated, because the aim is to preserve the neuron $\wedge$ or $\vee$ behavior. For each neuron, we remember its type ($\wedge$, $\vee$) at its network of origin; classification to these types is discussed in section 4.4. For $\wedge$ neuron and every positively weighted incoming edge, which was not transfered from the parent network to the offspring, its bias is decreased by the product of the weight and average activation value of the edge's source. In contrast, for $\vee$ neuron and every negatively weighted incoming edge, which was not transfered from the parent network to the offspring, its bias is increased by the product of the weight and average activation function value of the edge's source. Both these extension are driven by the idea of classification of $\wedge$ and $\vee$ neuron that is based purely on incoming weights and bias.

Finally, between unconnected neurons of each two adjacent layers, edges with a small initial weight are added. For illustration see Figure 4.7, where the crossover, given two parental networks, produces two offspring networks.

4.6. Structure Learning with Selective Forgetting

So far, only constructive approaches were discussed. At the end of this chapter, a destructive algorithm is presented. It is called *Structure Learning with Selective Forgetting* (SLSF) [81], based on the idea of forcing selected edges to have zero weights; so name

zeroing used in [25] sounds similar.

Even though the SLSF is a part of a little bit wider method called *Structure Learning with Forgetting* (SLF) [81], but we decided to implement only SLSF to test a simple regularization technique. The method extends the weight learning's optimization criterion by an expression corresponding to penalization of a weight, which are within a certain distance to zero. The final optimization criterion is shown in equation (4.7), where J stands for sum of squared errors, ϵ is amount of forgetting and θ is the maximal distance from zero within which the forgetting is applied. Derivation of such criterion is approximated by equation (4.8), because of the occurrence of absolute value in the extended criterion.

$$J' = J + \epsilon \sum_{|w_{ij}|<\theta} |w_{ij}| \tag{4.7}$$

$$\frac{\partial J'}{\partial w_{ij}} = \frac{\partial J}{\partial w_{ij}} + \epsilon \text{sign}(w_{ij}) \tag{4.8}$$

5. Comparison of selected propositional approaches

This chapter contains comparison of method described in chapter 4.

5.1. Datasets

Because there is no unified set of NSI benchmarks, set of datasets based on propositional logical formulae was created. Iris dataset [96] could be used as in other works [33, 8], but comparing neural network's based method on a dataset with three classes, having one of them linearly separable from the other two, is not very interesting.

Thus, 56 new datasets based of artificially created propositional logic formulae were created. The rationale behind these datasets is based on the idea of comparing the methods on datasets with gradually increasing complexity to see results of each particular method. Unfortunately, there is no standard measure of *complexity* of a propositional formula. Thus, *CNF score* is proposed as a complexity indicator of a propositional formula. CNF score is a number of clauses in CNF of a given formula. The rationale behind this score comes from the fact that in a single-hidden-layer neural network, the number of hidden nodes corresponds to the CNF score of a formula prescribing its structure and thus such a network should theoretically be able to learn any formula having such a score. Under this assumption, CNF score corresponds to the number of hidden nodes of the resulting neural network. Besides CNF score, other criteria were tested as well, but CNF score makes the most sense credible approach.

Each dataset composes of seven inputs (a, b, c, d, e, f, g), one output (x) and a formula ϕ, from which x was generated according to given inputs. Restrictions over ϕ were following:

- possible logical operators inside ϕ: $\wedge$, $\vee$, $\neg$ and $\oplus$
- $\oplus$ can be applied only to two arguments
- $\neg$ can be applied only to one argument
- $\neg$ can be used only within the first and second depth
- maximal depth of ϕ was 4
- given all 2^7 input possibilities, ϕ must produce at least 20% of each class (true, false)

Given these restriction, 11000 ϕ, respective datasets, were generated. The generation was done in bottom-up way, firstly generating all possible ϕ of depth 1 satisfying given constrains, before continuing to formulae with higher depths. During this generation, a formula, which truth table can be explained by some already generated formula, was removed and not computed into the amount of 11000.

Thus, for each ϕ of the 11000, a CNF score was computed; for this purpose [97] was used to transform a formula to its CNF. Thereafter, generated formulae were sorted according its CNF score to 56 clusters. From each cluster single formula was randomly selected. These 56 selected formulae correspond to our datasets.

5.2. Methodology

Having only dataset of size 2^7 and comparing methods containing non-determinism, e.g. randomly generation of weights, we decided to run 5-folds *stratified crossvalidation* 40 times with random folds creation for each run. Stratified splits represent a splitting process that takes into account the requirement of similar distributions of classes in both the whole dataset and each fold. Train and test MSE and accuracy were collected from each run as well as time, number of added neurons and resulted networks.

5.3. Parameters

Since SLSF, KBANN, TopGen and REGENT need an initial theory, or its corresponding network, three such inputs were created. For SLSF, three, six and nine hidden layers, each layer having 15 neurons fully connected to adjacent layers, were created. For the rest, three theories resulting in three, six and nine hidden layers networks, were constructed. The thee, six and nine hidden layers networks were created from theories having 7, 14 and 18 rules respectively. Note that networks produced by theories had edges even between neurons of non-adjacent layers; that is the main difference in comparison with the network given to SLSF. Those algorithms will be in experiments indexed by a number expressing the number of hidden layers of the corresponding initial network.

Although modularity allows to use different activation functions than logistic sigmoid, only that one was used in all cases. There are two simple reasons for this. Firstly, the methods were originally developed with usage of sigmoids. Secondly, possible number of activation functions is vast and evaluation would take a long time.

Note that each method learns a threshold classifier as well; it was done at the end of every single weight learning phase. All methods were run with the same weight learning setting. For detailed settings of parameters see appendix B.1.

5.4. Evaluation

Results are presented in order of descending average of crossvalidation accuracy over given methods, because sorting them according to CNF score did not resulted in monotone curves. Only results based on three layers initial networks are presented, because accuracy curves possessed the same decay tendency in all cases. The minor difference with the other initial networks is that their accuracy curves are shifted towards 0 a little bit.

Although, one may come up with an idea that datasets generated by formulae where $\oplus$ is most frequent one, are the *hardest* ones, such results were observed; it is caused by these methods or their initial structure. There are several formulae having multiple $\oplus$ in the left most (simplest) and the right most (hardest) end of the resulting graph, see Figure 5.1. Also, there is a seemingly simple formula at the 39th place that is composed from $\wedge$, $\vee$ and $\oplus$ having only depth two; $(a \wedge c) \vee (e \oplus f)$.

Train and crossvalidation accuracy, time requirements, train and crossvalidation MSE are shown in Figures 5.1 to 5.5. On the simpler datasets, given by the selected order, CasCor, DNC and REGENT are the most successful ones; in the second half of datasets, CasCor does not produce as good results as the two firstly named. As easily seen on train accuracy (Figure 5.2), both REGENT and DNC are overfitted.

Interestingly, KBANN is superior to TopGen in wide number of cases. This may be given by two facts. The first possible explanation is that this is caused by the split of a dataset inside TopGen. This may cause that the network does not get sufficient number of samples to be well trained. The second possible explanation is that the TopGen's idea is based on inaccurate assumptions.

Finally, SLSF does not produce any produce any accuracy results. Its capability of producing high accuracy results degrades with harder datasets. It is probably given by the fact that the given network was to big for this kind of dataset. When comparing SLSF across initial networks, the one having three layers has highest accuracy over the two other.

Figure 5.3 shows learning times of all methods. The fact that KBANN, SLF and DNC are faster than the rest is quite predictable. The two firstly named depend only on Backpropagation; the last one uses Backpropagation on different structures but reusing already learned weights. The time superiority of TopGen in comparison with REGENT is simply given by the fact that the later is a genetic algorithm. CasCor was in most of the cases slower than REGENT; probably it took long time to evolve some useful detectors within the network.

5.5. Conclusion

To sum up presented experiments, DNC and REGENT are favorites in most of the cases. Superiority of REGENT over KBANN and TopGen was expected, but superiority of TopGen over KBANN is little bit odd. Nonetheless, it could be caused by wrongly assumptions on top of which TopGen arisen. Since datasets can be generated by a formula containing negation, TopGen's decreasing of false positive is the precise opposite behavior with respect to the wanted goal.

DNC, REGENT and CasCor tend to overfit, but the last one probably rather remember given samples than create a generalization pattern; the two former ones are a little bit better in the generalization. SLSF capability of learning a pattern vanishes with harder formulae.

Order of CNF score, over given datasets, was not matched by decreasing accuracy order of selected methods.

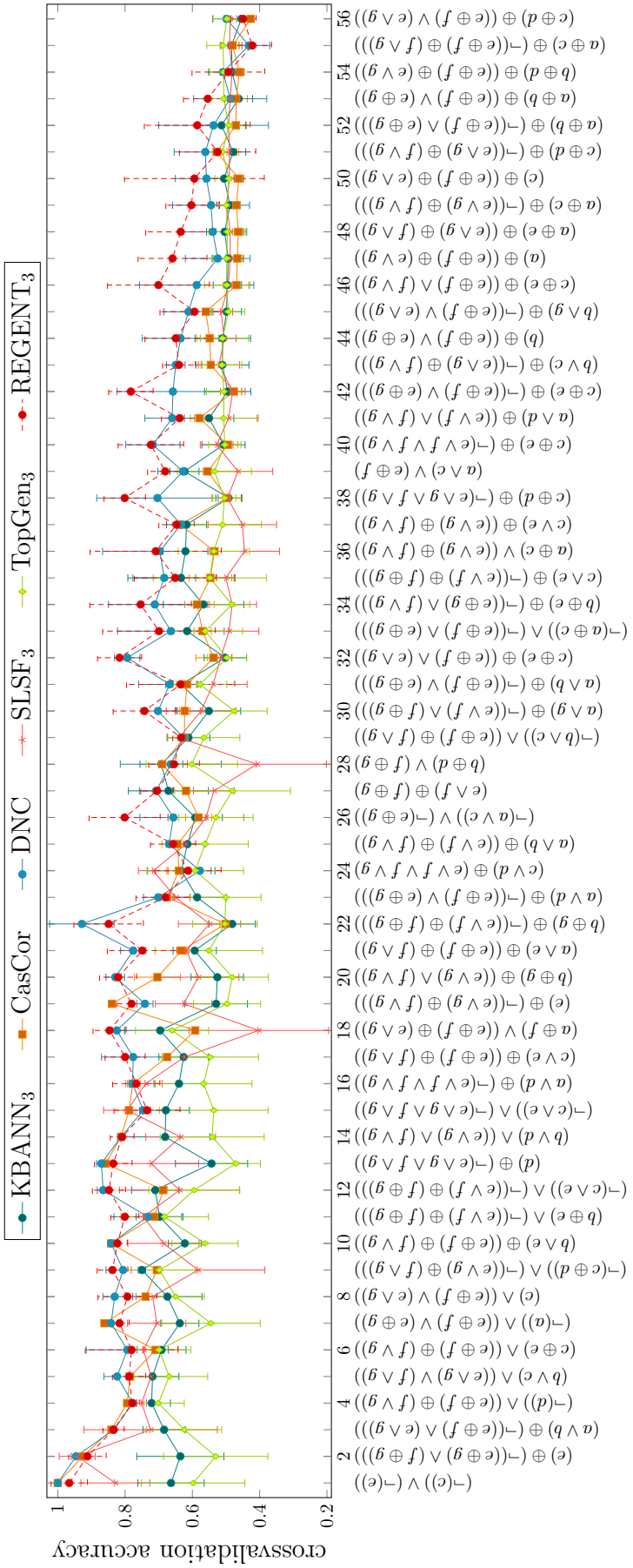


Figure 5.1. Averages and variances of crossvalidation accuracy of selected methods on 56 logical datasets.

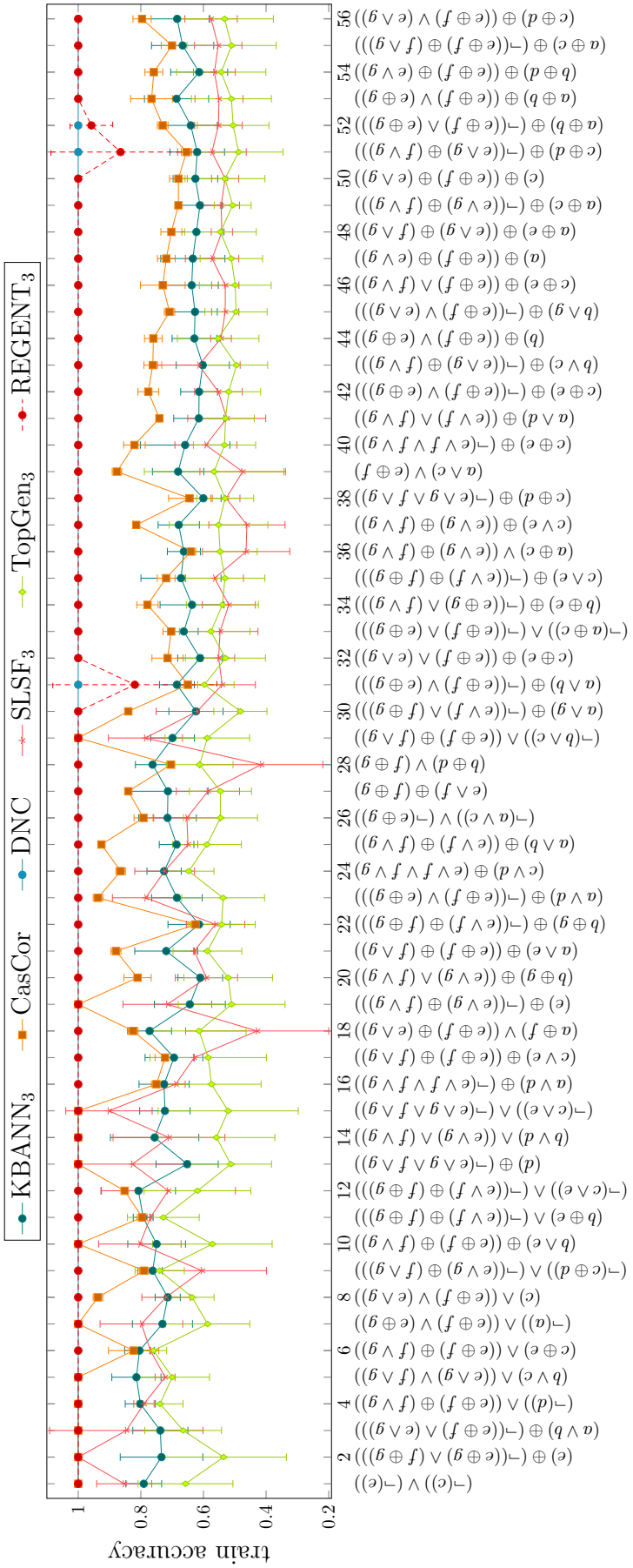


Figure 5.2. Averages and variances of train accuracy of selected methods on 56 logical datasets.

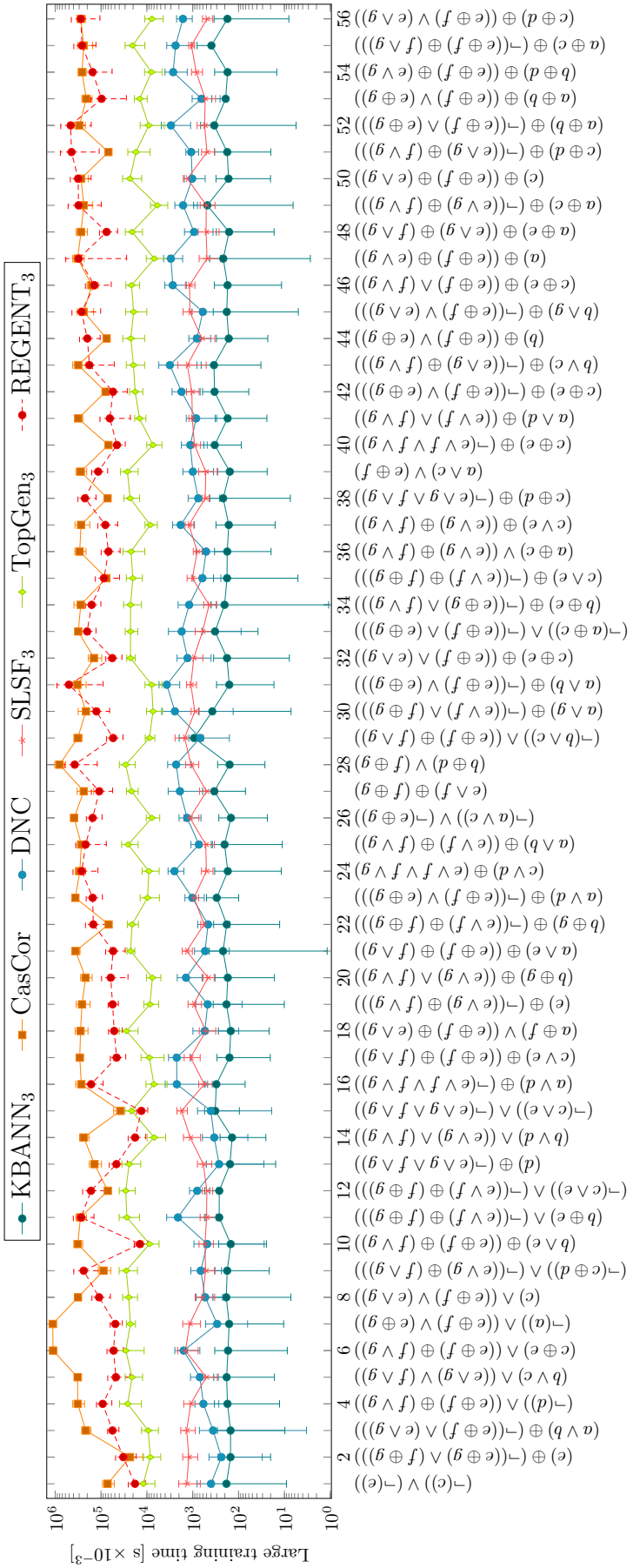


Figure 5.3. Averages and variances of training time of selected methods on 56 logical datasets.

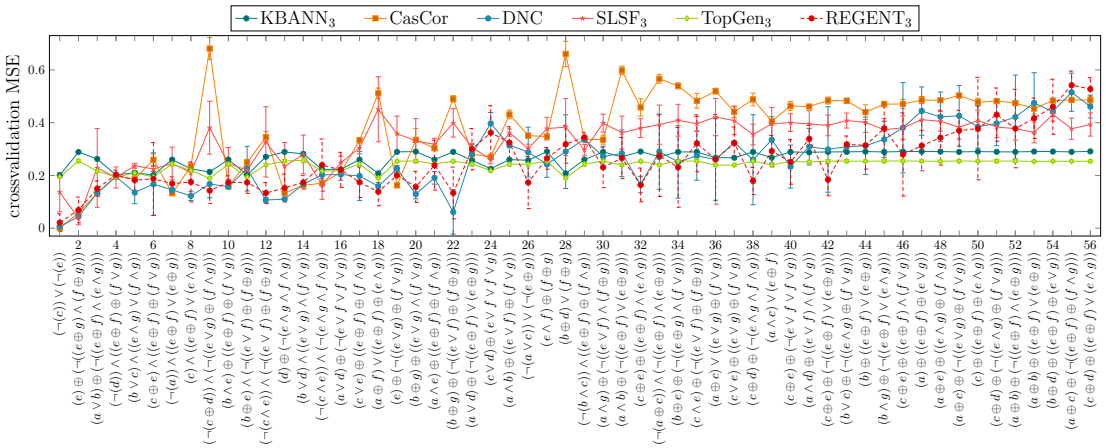


Figure 5.4. Averages and variances of crossvalidation MSE of selected methods on 56 logical datasets.

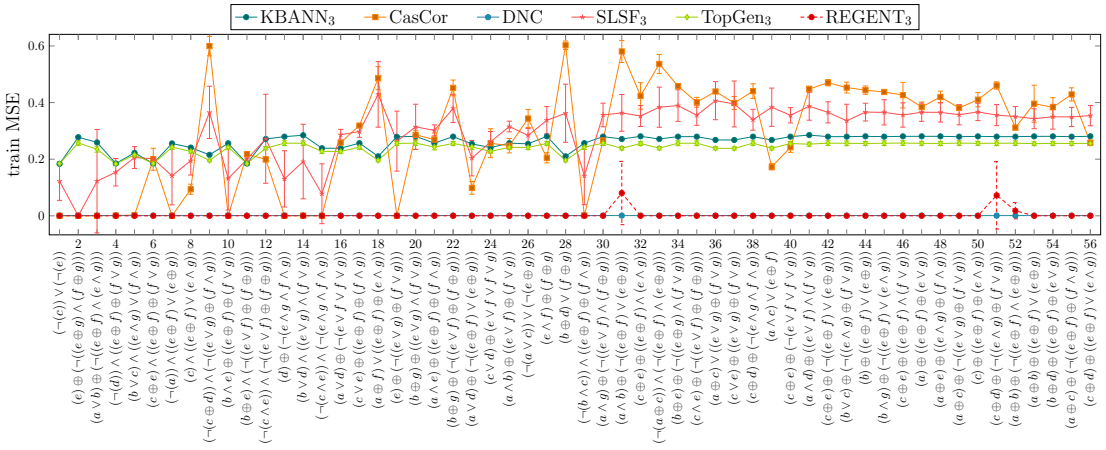


Figure 5.5. Averages and variances of train MSE of selected methods on 56 logical datasets.

6. Lifted Relational Neural Networks

This chapter describes *Lifted Relation Neural Networks* (LRNNs) [98, 3], which is a NSI approach combining first-order logic and neural networks. LRNNs is described in this thesis, because selected propositional structure learning techniques, which lifting is designed in the following chapters, are experimentally evaluated with the usage of LRNNs engine. Note that some definitions, examples and illustrations are taken from [3].

6.1. Main ideas of Lifted Relational Neural Networks

Let us firstly recall KBANN's idea; not the construction process in details, but its idea. KBANN tries to generalize a propositional pattern over a set of examples and an initial theory. In order to achieve this goal, it constructs a network similar to an initial theory, adding some possibilities for generalization, by adding some edges. LRNNs, although it has arisen from different idea than propositional logic, it does similar process in principle. It also aims to generalize a template, which is described in first-order logic. Thus, expressivenesses and complexities of each approach differ a lot.

LRNNs arose on top of lifted models [99], which define patterns from which specific (*ground*) models can be *unfolded*. These patterns are called *templates*. LRNNs combines lifting and grounding to learn weights of weighted relational clauses by firstly creating a set of *ground neural networks*, which is done by grounding (unfolding) a template over a set of examples. Thereafter, clauses' weights can be learned using gradient descent method adjusted to the case of shared weights, since edges are shared among grounded neural networks, and an edge may occur multiple times in one grounded network.

Novelty of the method lies in learning outputs values, e.g. classification problems, as well as latent concepts simultaneously. This is caused by the weight learning process. Since, LRNNs lives in the level of weighted clauses and each input example is given by a set of grounded facts in first-order predicate logic, the grounding is done by creating the least Herbrand model, which is very costly. For further reading, let us recall that we restricted ourself to only non-recursive first-order function-free predicate logic without negation.

6.2. Ground neural networks

A lifted relational neural network $\mathcal{N}$ is a set of weighted definite clauses in form (R_i, w_i), where R_i is a function-free definite clause and w_i is a real number. Then, $\mathcal{N}^*$ is used for denoting the set of clauses of $\mathcal{N}$ without weights; $\mathcal{N}^* = \{C : (C, w) \in \mathcal{N}\}$. Given a LRNN $\mathcal{N}$, let $\mathcal{H}$ be the least Herbrand model of $\mathcal{N}^*$. We define *grounding of the LRNN* N as $\overline{\mathcal{N}} = \{(h\theta \iff b_1\theta \wedge \cdots \wedge b_k\theta, w) : (h \iff b_1 \wedge \cdots \wedge b_k, w) \in \mathcal{N}$ and $\{h\theta, b_1\theta, \ldots, b_k\theta\} \subseteq \mathcal{H}\}$. In other words, $\overline{\mathcal{N}}$ is set of ground definite clauses which can be obtained by grounding rules from the LRNN and which are active in the least Herbrand model of $\mathcal{N}^*$.

Definition 6.1. Let $\mathcal{N}$ be a LRNN, and let $\overline{\mathcal{N}}$ be its grounding. Let $g_\vee$, $g_\wedge$ and $g^*_\wedge$ be families of multivariate functions with exactly one function for each number of argu-

ments. The *ground neural network* of $\mathcal{N}$ is a feed forward neural network constructed as follows.

- For every ground atom h occurring in $\overline{\mathcal{N}}$, there is a neuron A_h, called *atom neuron*. The activation functions of atom neurons are from the family $g_\vee$.
- For every ground fact $(h, w) \in \overline{\mathcal{N}}$, there is a neuron $F_{(h,w)}$, called *fact neuron*, which has no input and always outputs a constant value.
- For every ground rule $h\theta \impliedby b_1\theta \wedge \cdots \wedge b_k\theta \in \overline{\mathcal{N}}^*$, there is a neuron $R_{h\theta \impliedby b_1\theta \wedge \cdots \wedge b_k\theta}$, called *rule neuron*. It has the atom neurons $A_{b_1\theta}, \ldots, A_{b_k\theta}$ as inputs, all with weight 1. The activation functions of rule neurons are from the family $g_\wedge$.
- For every rule $(h \impliedby b_1 \wedge \cdots \wedge b_k, w) \in \mathcal{N}$ and every $h\theta \in \mathcal{H}$, there is a neuron $Agg^{h\theta}_{(h \impliedby b_1 \wedge \cdots \wedge b_k, w)}$, called *aggregation neuron*. Its inputs are all rule neurons $R_{h\theta' \impliedby b_1\theta' \wedge \cdots \wedge b_k\theta'}$ where $h\theta = h\theta'$ with all weights equal to 1. The activation functions of the aggregation neurons are from the family $g^*_\wedge$.
- Inputs of an atom neuron $A_{h\theta}$ are the aggregation neurons $Agg^{h\theta}_{(h \impliedby b_1 \wedge \cdots \wedge b_k, w)}$ and fact neurons $F_{(h\theta,w)}$. The weights of the input neurons are the respective w's.

Example 6.1. Let us consider the following LRNN

$$\begin{aligned}\mathcal{N} = \{&(\mathit{foal}(A) \impliedby \mathit{parent}(A,P) \wedge \mathit{horse}(P), w_m), (\mathit{foal}(A) \impliedby \mathit{sibling}(A,S) \wedge \mathit{horse}(S), w_n),\\ &(\mathit{horse}(\mathit{dakotta}), w_1), (\mathit{horse}(\mathit{cheyenne}), w_2), (\mathit{horse}(\mathit{aida}), w_3),\\ &(\mathit{parent}(\mathit{star}, \mathit{aida}), w_6), (\mathit{parent}(\mathit{star}, \mathit{cheyenne}), w_5), (\mathit{sibling}(\mathit{star}, \mathit{dakotta}), w_4)\}.\end{aligned}$$

The LRNN $\mathcal{N}$ and its ground neural network are shown in Figure 6.1.

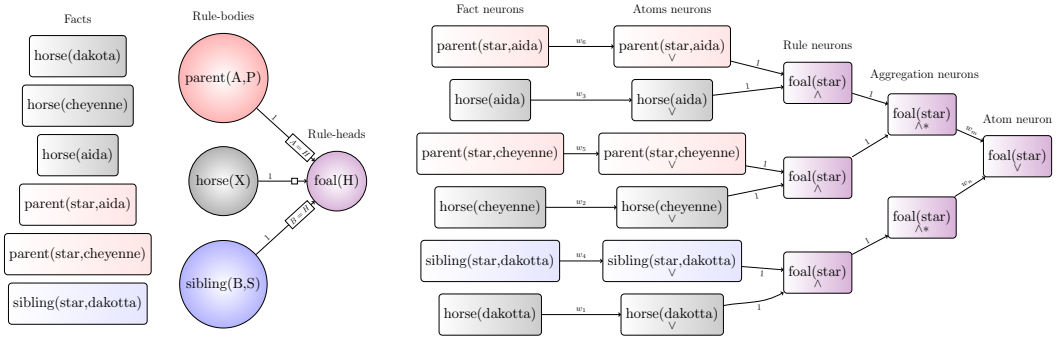


Figure 6.1. Depiction of the rule-based template (left) of LRNN $\mathcal{N}$ from Example 6.1, and its corresponding ground neural network $\overline{\mathcal{N}}$ (right), with colors denoting the predicate signatures, rectangular nodes corresponding to ground and circular to lifted literals, respectively.

Example 6.2. Let show $\lambda - \kappa$ notation within the LRNNs. The example above:

$$\begin{aligned}(\mathit{foal}(A) &\impliedby \mathit{parent}(A,P) \wedge \mathit{horse}(P), w_m)\\ (\mathit{foal}(A) &\impliedby \mathit{sibling}(A,S) \wedge \mathit{horse}(S), w_n)\end{aligned}$$

is translated into $\lambda - \kappa$ notation as

$$\begin{aligned}(\kappa_{\mathit{foal}}(A) &\impliedby \lambda_{\mathit{foal}_1}(A), w_m)\\ (\kappa_{\mathit{foal}}(A) &\impliedby \lambda_{\mathit{foal}_2}(A), w_n)\\ (\lambda_{\mathit{foal}_1}(A) &\impliedby \kappa_{\mathit{parent}}(A,P) \wedge \kappa_{\mathit{horse}}(P), 1)\\ (\lambda_{\mathit{foal}_2}(A) &\impliedby \kappa_{\mathit{sibling}}(A,S) \wedge \kappa_{\mathit{horse}}(S), 1)\end{aligned}$$

By looking at Examples 6.1 and 6.2, one can easily see that rule, aggregation and atom neuron corresponds to λ rule body, λ predicate and κ predicate respectively. Such observation is important, one predicate may be grounded multiple times in a ground network.

We restrict LRNNs to have only one output, which is denoted by λ_f. Only κ_f/a predicates may imply λ_f, but this restriction is only for clearer notation through the rest of the thesis. Because our aim is to search in the level of templates, the notation of rules without corresponding weights is used in the following chapters.

6.3. Activation functions and weight learning algorithm

Since the main aim of this thesis is structure learning, this section contains only a brief description of used activations functions and basic concept of its weight learning algorithm. For more interested reader we recommend original source of LRNNs [3].

Behavior of grounded neural networks heavily depends on used families of activation functions $g_\vee$, $g_\wedge$ and $g^*_\wedge$. Intuitively, based on the same idea as KBANN, having a λ ($\wedge$) rule, one would like to be this rule active if and only if all the inputs of literals from the rule body of the rule have hight outputs. Based on the same idea, having and κ ($\vee$) rule, one would like to be this rule active if at least one of its body literals output is active.

In our experiments, we used identity for κ rules and sigmoid function for λ rules. Current state of LRNNs engine allows to use different activation functions for different predicates. This interferes with structure learning as well, but there was a lack of time for experiments of those kinds, since we investigated other approaches to structure learning.

The most important message of this section is, that only κ rules weights are learnt. The process is based on gradient descent method capable of learning shared weights. Shared weights arose from the fact that given a sample and a template, produced grounded neural network my contain several atom neurons, corresponding to the same κ predicate. Weights of incoming edges from a κ predicate precisely correspond to its weights in the weighted clauses level. Therefore, the weight is shared among all of the κ's atom neurons in samples.

7. Predicate rule generation

Having defined LRNNs, in the previous chapter, we can move to the main contribution of this thesis, which is described in the present and the following chapter. Speaking of constructive methods described in Chapter 4, they are mostly presented as a combination of two parts – *neuron generation* and *neuron insertion*. These can be modularized and taken into deep, independent study, done in separate ways. For example, TopGen's base neuron addition is a valid approach, since redundant antecedents of the neuron vanish during weight learning phase. However, we aim for a modularized approach, because we believe that such a way can enhanced our knowledge by discovering generalization of several tasks.

In the level of first-order predicate logic, the neuron generation transforms to a *rule generation* task while the neuron insertion to a *rule insertion*. Note that rule insertion is a subtask of a more general problem called *template change strategy*, which covers both destructive and constructive approaches for possible template change. This chapter is focused on rule generation, while Chapter 8 aims to elaborate on lifted rule insertion and template change strategy techniques. In this chapter, firstly, several concepts of transferring neuron generation from Chapter 4 to first-order predicate level are presented (sections 7.1 and 7.2); with zero arity for the beginning. Secondly, non-zero arity extensions of former approaches are described. Thirdly, a constraints rule generation is presented section 7.4. Finally, some interesting ideas from the machine learning are presented as a possible base for another rule generators (section 7.5).

In this work, we restrict ourself to extending the template by adding only new non-recursive rules without negation. The reasons for this have been presented earlier.

For clarity and prevention of misunderstanding, some LRNNs' properties are changed for simpler notation. Although each κ rule possesses a weight, this is not a concern of this chapter, thus κ rules will be displayed without weights. Each λ and κ rule denoted in this chapter will start with its corresponding symbol; so one can easily recognize the rule's type.

7.1. Base rules

Several of the propositional constructive approaches gradually add base neuron after another; recall that it is a neuron having one incoming edge from every input node. In order to lift base neurons, *base rules* are to be defined. But firstly, before defining base rules, a decision whether base predicates are to be considered as λ or κ ones has to be made. Since now, each time a base predicate is mentioned, it is a λ predicate by default. This decision was backed by the aim of creating rather narrow LRNNs networks. Categorization of base predicates to the other group is possible as well, but resulting rules would be different.

Generally speaking, a set of base rules consist of one rule for each base predicate, each one in form $\kappa/a \Longleftarrow basePredicate/b$, where κ is a predicate of arity a and b is $basePredicate$'s arity; for sure, $basePredicate$ possesses different fresh variables in its arguments. It is said that κ, which is head of every rule in the set, is a *base* κ. For

clarity see Algorithm 5. Note, that in this section is focused on generation of base rules with kappa having zero arity; extension to non-zero arity is done in section 7.3.3.

Algorithm 5: Base rules generator

Input: Template *template* and set of samples *samples*
Output: A base rule set

```
1 i ← |pred(template) ∪ pred(samples)|
2 rules ← ∅
3 for l ∈ pred(samples) do
4 |   rule ← (κ_i ⟸ l)
5 |   rules ← {rule} ∪ rules
6 return rules
```

The rationale behind generating such a set of rules arises from the requirement of having one layer of learnable weights, thus mimicking one neuron in the propositional word. In order to use this rule, and since a ground network structure is composed of alternating λ and κ layers, only a λ rule can be entailed by a κ. Therefore, in order to use values of such created rule set, its corresponding κ head is entailed by a λ; e.g., the following rule $\lambda \Longleftarrow \kappa$ must be used.

Example 7.1. Given the following set of samples

$$S = \{\{male(adam), female(diana), sibling(adam, diana)\},$$
$$\{male(joseph), male(adam), parent(joseph, adam)\}\}$$

and a template having $\lambda_f/0$ and $\kappa_f/0$

$$T = \{\lambda_f \Longleftarrow \kappa_f\}$$

the following set of base rules is created:

$$B = \{\kappa_2 \Longleftarrow male(X), \kappa_2 \Longleftarrow female(X),$$
$$\kappa_2 \Longleftarrow sibling(X, Y), \kappa_2 \Longleftarrow parent(X, Y)\}$$

When referring to generated base rule set by κ_f, following rules must be added:

$$\kappa_f \Longleftarrow \lambda_2$$
$$\lambda_2 \Longleftarrow \kappa_2$$

7.2. Cascade rules

Cascade Correlation gradually builds up a network by adding a new hidden neuron having one incoming edge from each input and a hidden neuron. Lifting such approach is straightforward as the previous case. *Cascade rule* is a base rule set extended by adding $\kappa \Longleftarrow \lambda_j$ for every $\lambda_j \in lambdas(template)$ such that $\lambda_j \neq \lambda_f$, $lambda_j$; having fresh variables in its arguments, where κ is the base κ of the extended rule set. Such created rule set is called *cascade rule set* and the κ is called *cascade* κ. For clarity, see Algorithm 6.

The rationale behind this lifting is the same as in the previous case. Rules appended to the base rule set mimic the cascade effect of a cascade neuron. Rule $\kappa_i \Longleftarrow \lambda_f$ cannot be added, since λ_f is the root predicate; a recursion would occur in the opposite case.

Algorithm 6: Cascade rules generator

Input: Template *template* and set of samples *samples*
Output: A cascade rule set

```
1 i ← |pred(template) ∪ pred(samples)|
2 rules ← ∅
3 for l ∈ (pred(samples) ∪ lambdas(template)) \ {λ_f} do
4 |   rule ← (κ_i ⟸ l)
5 |   rules ← {rule} ∪ rules
6 return rules
```

Example 7.2. Given the following set of samples

$$S = \{\{\}male(adam), female(diana), sibling(adam, diana)\},$$
$$\{male(joseph), male(adam), parent(joseph, adam)\}\}$$

and a template

$$T = \{\lambda_f \Longleftarrow \kappa_f, \kappa_f \Longleftarrow \lambda_2, \lambda_2 \Longleftarrow \kappa_2, \kappa_2 \Longleftarrow male(X),$$
$$\kappa_2 \Longleftarrow female(X), \kappa_2 \Longleftarrow sibling(X, Y), \kappa_2 \Longleftarrow parent(X, Y)\}$$

the following cascade rule set is created:

$$C = \{\kappa_4 \Longleftarrow male(X), \kappa_4 \Longleftarrow female(X), \kappa_4 \Longleftarrow sibling(X, Y),$$
$$\kappa_4 \Longleftarrow parent(X, Y), \kappa_4 \Longleftarrow \lambda_2\}$$

When referring to generated base rule set by κ_f, following rules must be added:

$$\kappa_f \Longleftarrow \lambda_4$$
$$\lambda_4 \Longleftarrow \kappa_4$$

7.3. Non-zero arity rules

Previous attempts are straightforward lifting of propositional neurons creation to first-order predicate logic level. This section is focused on creating predicate rules having a non-zero arity predicate in head. A question of order, a sequence , of variables in the rule head and the whole rule is a new task to be taken care of. Firstly, variables' ordering approaches are discussed. Thereafter an extension of base and cascade rules to non-zero arity is presented.

Before further investigation, let us define several concepts for easier manipulation with variables in a rule. Although a rule is a set of literals, each one having arbitrary variables in its arguments, the rule is here redefined as a sequence of literals of order given by writing of the particular rule or rule's body. This section is heavily based on operation of variables within the rule, thus we define a *variable sequence* to be composed of a ordered set, of variables occurring in the literals sequence according to the order of appearance. Literals sequence can be either given by writing of a whole rule, starting with head, or a rule's body only.

Example 7.3. Let us have the following rule: $familiar(X, Z) \Longleftarrow familiar(X, Y) \wedge familiar(X, Z)$. Its sequence is X, Z, Y while rule's body sequence is X, Y, Z.

7.3.1. Rule head's variable

Let us firstly focus on the meaning, not the order, of arities of λ and κ predicates. For example, having a rule $\lambda(X) \impliedby \kappa(X, Y)$ is useful, because it holds in the case that X is represent a object, which is in relation given by κ. In the reversed order of arities, e.g. $\lambda(X, Y) \impliedby \kappa(X)$, is a malformed case, since X must be in relation given by the κ, but there are no restriction on Y variable. From computation point of view, such a rule is expensive, because Y may bind to everything in the domain. Also, it does not express any interesting property of Y. Supported by these facts, following idea is proposed – in a rule head, only variables, occurring in the body, may appear. The most straightforward way is to take rule's body variables repeatedly. Others constraints may be added, for instance rule's head arity must be equal or lower to number of variables occurring in its body.

7.3.2. Variable order

Firstly, let us focus on variable ordering within a rule body. Let have a following rule with more than one literal in its body: $\lambda \impliedby male(W) \wedge sibling(X, Y) \wedge female(Z)$. Such a rule is very general, more general than the following $\lambda \impliedby male(X) \wedge sibling(X, Y) \wedge female(Y)$. The later states that there λ holds only if there is a brother and a sister. The former holds whenever there is at least one male, one female and something that is in sibling relationship.

The problem here lies in the independent literals within the rule body. The later is more specific and more suitable for our case. Thus, the main idea of this example is motivation of creating only rules that have dependent literals within a rule body.

To tackle this problem we propose *chaining body* variable order within a rule's body. Having a sequence of literals, the chaining body is defined as follow: each literal possesses arguments order such that the first argument is the same variable as the last argument of the previous non-zero arity literal in the sequence; rest of the arguments are filled with fresh variables not occurring in any other literal of the sequence. The term of *subsequence* is used for a sequence of a continuous ordered list.

Example 7.4. Given the following rule body

$$human/1 \wedge work/0 \wedge salary/3 \wedge month/1$$

the following variable order will be produced by chaining body:

$$human(X) \wedge work \wedge salary(X, Y, Z) \wedge month(Z)$$

This approach does not solve every problem. For example having known that arguments of $salary/3$ represent *who*, *how many* and in *which* month one get a salary, rule in form $human(X) \wedge work \wedge salary(X, Y, Z) \wedge human(Z)$ should not bind a single time, unless there is an entity being a *human* and a *month* at once. Therefore we, making a body literals dependent is domain dependent. The most straightforward way to deal with this, is to firstly find possible dependencies between literals withing train samples, thereafter use this knowledge to make only some variable ordering. In fact, this is the idea of least general generalization; more on this in section 7.5. On the other hand, there may be vast number of such possibilities.

Secondly, the question of variable ordering in rule head arises. It is the same problem for λ and κ rule, the only difference is that the former possesses only one literal in its

body. Thus description of our proposed approach will be given for κ rule, the second one is a special case.

The simplest variable order in a rule head, beside a random one, is to take set of body variables sequence. In case that the sequence's length is greater or equal to the body head's arity, a subsequence of length head's arity is selected as head variable order; otherwise the sequence is used multiple times till the length of the head's arity is achieved. This head's variable order will be referred as *chain root order*.

Example 7.5. Given the following rule with head only in the Prolog-like description

$$\kappa/2 \Longleftarrow salary(X, Y, Z)$$

the following variable order for head predicate will be created by chain root order:

$$\kappa(X, Y)$$

Another, a computationally expensive, extension of the previous is creation of a new rule for each variation of the body's variables set of size equal the head's arity; in case that head's arity is bigger than the cardinality of the set, the set is filled with fresh variables to size of the head's arity. This rule generation will be referred as *variation order*.

Example 7.6. Given the following rule with head only in Prolog-like description

$$\kappa/2 \Longleftarrow salary(X, Y, Z)$$

following rules with a variable order of the head predicate will be created by the variation order:

$$\begin{aligned}
\kappa(X, Y) &\Longleftarrow salary(X, Y, Z) \\
\kappa(X, Z) &\Longleftarrow salary(X, Y, Z) \\
\kappa(Y, X) &\Longleftarrow salary(X, Y, Z) \\
\kappa(Y, Z) &\Longleftarrow salary(X, Y, Z) \\
\kappa(Z, X) &\Longleftarrow salary(X, Y, Z) \\
\kappa(Z, Y) &\Longleftarrow salary(X, Y, Z)
\end{aligned}$$

As one can easily see, the blow up of variations order is substantial. Thus we propose a relaxation that combines both variation and chain root order called *sliding window order*. Sliding window order generates a set of rules, such that for each subsequence of a sequence used in rule chain order, having the same length as head's arity, a new rule having such head variable order is generated; in case that the sequence is smaller than the head's arity, it is extended by itself till the its length is long enough.

Example 7.7. Given the following rule with head only in Prolog-like description

$$\kappa/2 \Longleftarrow salary(X, Y, Z)$$

following rules with variable order for head predicate will be created by sliding window order:

$$\begin{aligned}
\kappa(X, Y) &\Longleftarrow salary(X, Y, Z) \\
\kappa(Y, Z) &\Longleftarrow salary(X, Y, Z)
\end{aligned}$$

Since each λ rule can have only one definition, chain root order is used, but others are possible as well; more on this in section 7.5.

7.3.3. Base and cascade rule with non-zero arity

As described in the previous subsection, generation of non-zero arity rules can be done in multiple ways. Now, an extension of base and cascade rules to non-zero arity level is to be presented. We propose an extension for this case, which arises from the task of appending base or cascade rule to an existing κ one. The proposed extension will be simulated for a base rule.

Given a κ_a with arity a a κ_b of the arity a is created. Thereafter for each base predicate p of arity m a set of rules of form $\kappa_b/a \Longleftarrow p/m$ according to a variable ordering strategy is created. The only problem, which may occur here, is the creation of λ_b implied by κ_{a_b} for each created kappas with arity a_b. Here, sliding window, variation or root chain order can be used. For illustration see Example 7.8.

Example 7.8. Given the $\kappa_a/1$ and the following set of samples

$$S = \{\{male(adam), female(diana), sibling(adam, diana)\},\\ \{male(joseph), male(adam), parent(joseph, adam)\}\}$$

the following set of base rules with non-zero arity and sliding window order will be created:

$$\begin{aligned}
\kappa_b(X) &\Longleftarrow male(X)\\
\kappa_b(X) &\Longleftarrow female(X)\\
\kappa_b(X) &\Longleftarrow sibling(X, Y)\\
\kappa_b(Y) &\Longleftarrow sibling(X, Y)\\
\kappa_b(X) &\Longleftarrow parent(X, Y)\\
\kappa_b(Y) &\Longleftarrow parent(X, Y)
\end{aligned}$$

Of course that in order to use this base κ_b, an rule, entailing this κ_b, must be created, for example:

$$\lambda_b(X) \Longleftarrow \kappa_b(X)$$

Extension of cascade rules to non-zero arity is done in the same way.

7.4. Random rule generator with constraints

Beside generation of base κ, other approaches are valid. One of these is generation of a random rule satisfying set of constraints. Let us start with description of constraints, then continue with randomized rule generation.

There are several parameters that may be useful while generating a rule, these are:

- λ rule
  - body length
- κ rule
  - number of definitions
- common for λ and κ rules
  - head's arity
  - variable order
  - longest path to terminal rule

Firstly, start from the top of this list to describe the constraints. As we known, a λ rule can possesses only one definition, but its body length can vary. To this part, constraints based on the minimal and maximal body length are given, reducing the space of possible rules. In contrast, κ rule can be defined multiple times, but each time the rule body must contains exactly one literal. Thus constrains on the minimal and maximal number of κ definitions arise.

For both λ and κ rules, constraints over head's arity, variable order and longest path to terminal rule are possible. The head's arity can be constrained from both sides – minimal and maximal. There can be constraints on variable ordering, but instead of this, one of the following strategies is used: chaining body, sliding window, chain root and random assignment of variables to arguments. The longest path to terminal rule is constrained only by an upper bound. As in the previous case, the arity of a rule's head predicate can be constrained by a bottom and an upper bound.

Generating new rules may be done in different ways, for example κ rule may be added, λ rule body may be extended, both new λ and κ predicate may be inserted. Our random generator, displayed in Algorithm 7, is capable of all of these; note that the pseudocode does not display all constraints parameters for the sake of the space. The generator firstly decides whether to extend λ or κ predicate and selects an appropriate one (lines 1 to 4). Then, either new predicate is created or not. When decision is made not to create new predicate, new rule is generated for the selected predicate (line 13). The rule method $randomBody(predicate, template, samples)$ constructs a new κ rule, in case that *predicate* is of the same type; otherwise it extends the λ rule with a head *predicate*. Method $randomBodyWithPredicate$ does the same thing, only forcing newly created predicate to be inside the body.

When a decision is made that a predicate should be created, the algorithm creates new κ or λ predicate; always the created predicate is of the other type than the selected predicate *head*. Thereafter, new rule is created for the newly created predicate and new rule is created for the firstly selected predicate (lines 9 and 11). The similarly named methods work as in the previous case when no predicate is created.

While extending or creating new rules, several constraints must hold. When creating a λ rule, the old one has to be removed from the template since only one λ definition can be in the template. Contrary to that, if given κ has more definition than the corresponding upper bound, randomly chosen old rule of the given κ is removed from the template. When generating a fresh new predicate, it creates a predicate that does not occur in the template nor in the base predicates with arity within the given bounds. When generating new body rule, only κ predicates can be inserted into a λ body; contrary, κ body can contains only λ and base predicates. For every predicate given to a newly created body, two constraints must hold. Firstly, the maximal path to terminal predicate, from that rule, summed with the maximal path to head of the body, does not exceed the longest path constraint's bound. Secondly, the rule body predicate must preserve the non-recursive property of the template.

7.5. Predicate rule generation extensions

So far, several possibilities of predicate rule generation have been presented, but this was mainly inspired by lifting the propositional approaches. Since the field of relation learning is vast, only several extensions are mentioned in this section. The first such extension was proposed in section 7.4. Instead of approaching to a base or cascade rule generation as to a totally different processes to each other, they have in common the

Algorithm 7: Random rule generator

Input: Template *template*, set of samples *samples*, probability of predicate creation p_c, probability of λ change p_λ

Output: A set of generated rules

```
if isProbable(p_λ) then
    head ← lambdas(template)
else
    head ← kappas(template)
rules ← ∅
if isProbable(p_c) then
    newPredicate ← randomArityFreshPredicate(head, template, samples)
    body_i ← randomBody(newPredicate, template, samples)
    rules ← {randomRule(head, body_i)} ∪ rules
    body_h ← randomBodyWithPredicate(head, newPredicate, template, samples)
    rules ← {randomRule(head, body_h)} ∪ rules
else
    body_h ← randomBody(template, samples)
    rules ← {randomRule(head, body_h)} ∪ rules
return rules
```

rule generation property. A rule generator may have multiple definitions.

Constructing a template could be based on frequent pattern, e.g. association rules, in first-order level. In fact, such an approach is very intuitive – instead of building template by random rule generation, it can be used in a semi-supervised way. Approaches based on this were already proposed, for example in KBANN [9], aimed to escape from local minimum during the weight learning phase. But they used insertion of rules provided by experts. Similar approach, while during the search of neural-symbolic cycle, was proposed in INSS [15], also with the need of an expert, who examined the extracted rules. Novelty of our proposition is automatically mining of such patterns (rules). This can be done by several exiting methods, for example *Warmer* [100], *Farmer* [101] or others [102, 103].

In section 7.3, variable ordering techniques were discussed. A variable order within a rule is the rule's property. Thus, one can search for frequent rules together with corresponding variable order within the samples, or even within the grounded neural networks (the least Herbrand model). For such approaches, a least general generalization based method may be used, for instance *Golem* [104].

In fact, in search of frequent patterns, aimed at variable order, one could run LRNNs with different variable order of given rule and then count support, which is in fact given by sizes of generated grounded networks. Of course, this brings computational blow up, because of possible variable orders. Thus one may find useful some other methods in search of such rules.

Another relational learning techniques can be used for creation the initial template [105, 106]. The aim here is not to run a relation learner to full depth and use its outputs as an input to LRNNs. We believe that a small portion of found theory by some of these approaches, could serve as initial template for constructive structure learning approaches.

Leaving the field of automatically derived rules, a grammar describing which rules

can be generated, is also a valid approach. Naturally that such grammar would need an expert, who would create it for each domain. On the other hand, the search space of possible rules could be meaningfully pruned; for example, grammar constraining isomorphism could prevent generation of a huge number of meaningless rules in some domains.

In fact, we can look at the grammar approach from another perspective. Instead of using an expensive expert and convincing him that a domain specific grammar from his field is really important, numerous runs with different grammars, corresponding to different properties, may be tried. Upon results from such runs, one may find useful properties of the data.

8. Transfer of selected structure learning approaches to first-order logic

In the previous chapter was shown that lifting of propositional structure learning approaches to first-order predicate logic can be seen as changing the template, and that this can be separated into two independently investigated areas. This chapter investigates lifted template changing strategies. The strategies can be divided into separate categories, e.g. to a rule insertion part which corresponds to node insertion in the propositional techniques.

Note that incorporating of KBANN to the first-order predicate case would result in something similar to LRNNs, thus it it is not investigated here. Both of these approaches arose from different ideas.

8.1. Local search

The most straightforward constructive method for structure learning within first-order level is local search over the template. Evaluation of a template is quite simple – its accuracy after grounding and weight learning. The one and only thing that is to be defined is successors generation. For this purpose, the proposed random rule generator from section 7.4. Other possibilities are using *false decrease* (section 8.4.2) or *rule deletion* (section 8.5.1).

This approach uses LRNNs engine as a black-box and the whole process of learning could be stalled because of repeatedly grounding and a random, useless, rule. Our aim here is threefold: to reduce computational demands, add useful rules and use learnt neuron insertion strategies from Chapter 4. Computational demands can be reduced by adding only rules that do not change structures of already ground neural networks. For example, adding a fact or a rule producing already used predicate in another rule's body from the template are such cases. This idea is behind sections 8.2 and 8.3.

8.2. Lifted Dynamic Node Creation

Lifting DNC method is quite straightforward. The idea is to gradually construct flat theory (template), so the DNC's one-hidden-layer property is preserved. The flat template is constructed gradually by the following process: for given κ_f/a a set of base rules entailed by fresh κ_i/a is created; thereafter the set is added to the template. To finalize this template's extension, κ_i/a must be entailed by κ_f/a; this is done via fresh predicate λ_i/a and two rules: $\lambda_i/a \impliedby \kappa_i/a$ and $\kappa_f/a \impliedby \lambda_i/a$, both having the same variable order in rule's body and head. Such constructed template is ensured to have two layers of learnable weights. Pseudocode for this algorithm is not shown here, because it is the same algorithm as Algorithm 1. The main difference with the former is that the number of added predicates is compared to the given bound instead of the number of added neurons. In fact, in the pseudocode, *rule* and *template* would

be written instead of *neuron* and *network structure* respectively; informally speaking, this holds for every lifted algorithm.

Example 8.1. Given the following set of samples

$$S = \{\{male(adam), female(diana), sibling(adam, diana), happy\}, \{male(joseph), male(adam), parent(joseph, adam)\}\}$$

and a template

$$T = \{\lambda_f \Longleftarrow \kappa_f\}$$

the resulting template after first rule addition would look like:

$$B = \{\kappa_f \Longleftarrow \lambda_2, \lambda_2 \Longleftarrow \kappa_2, \kappa_2 \Longleftarrow happy, \kappa_2 \Longleftarrow male(X), \kappa_2 \Longleftarrow female(X), \kappa_2 \Longleftarrow sibling(X, Y), \kappa_2 \Longleftarrow parent(X, Y)\}$$

Note that having κ_f/a with $a > 0$, usage of sliding window, chain root or another variable ordering would be necessary.

8.3. Lifted Cascade Correlation

The incorporation of Cascade Correlation to the first-order predicate level is also straightforward, as in the previous case. The main idea of gradually constructing hierarchical structure is preserved together with the insertion of a predicate that is firstly learnt so its output correlates to outputs; this is followed by learning only the output weights.

The aim is to create a cascade architecture within the template, such that there is a at least one learnable edge between the newly added *candidate predicate*, which is lifted candidate neuron from propositional level, and each base predicate and previously added predicates. As one can easily see, having non-learneable edges does not allow weight learning process, thus Cascade Correlation would have no effect.

Pseudocode for the method is not presented, because it is very similar to Algorithm 2. The method firstly creates an initial template by a using the cascade rule generator. Given λ_f and κ_f/a, the initial template is constructed by generating cascade rule set entailed by κ_f/a. Rule $\lambda_f/0 \Longleftarrow \kappa_f/a$ is added.

Example 8.2. Given the following set of samples

$$S = \{\{male(adam), female(diana), sibling(adam, diana), happy\}, \{male(joseph), male(adam), parent(joseph, adam)\}\}$$

and final predicates $\lambda_f/0$, $\kappa_f/1$, the initial template, produced with usage of sliding window order, follows:

$$T_{init} = \{\lambda_f \Longleftarrow \kappa_f(X), \kappa_f(X) \Longleftarrow \lambda_2(X), \lambda_2(X) \Longleftarrow \kappa_2(X), \kappa_2(X) \Longleftarrow happy, \kappa_2(X) \Longleftarrow male(X), \kappa_2(X) \Longleftarrow female(X), \kappa_2(X) \Longleftarrow sibling(X, Y), \kappa_2(Y) \Longleftarrow sibling(X, Y), \kappa_2 \Longleftarrow parent(X, Y), \kappa_2(Y) \Longleftarrow parent(X, Y)\}$$

Secondly, when extending a template with κ_f/a, a fresh new κ_i/a and a cascade rule set entailed by κ_i/a are created. Then, the first layer of learnable weights, w.r.t. the candidate predicate κ_i/a, is learnt in order to maximize correlation of the candidate predicate w.r.t. outputs. After weights are learnt, the template is extended by the generated cascade rule set, $\kappa_f/a \Longleftarrow \lambda_i/a$ and $\lambda_i/a \Longleftarrow \kappa_i/a$, both having the same variable order in rule's head and body.

Example 8.3. Given S and T_{init} from Example 8.2 and using slide window order, the template after adding another cascade predicate, is extended by:

$$\begin{gathered} T_e = \{\kappa_f(X) \Longleftarrow \lambda_4(X), \lambda_4(X) \Longleftarrow \kappa_4(X), \kappa_4(X) \Longleftarrow \lambda_2(X), \\ \kappa_4(X) \Longleftarrow happy, \kappa_4(X) \Longleftarrow male(X), \kappa_4(X) \Longleftarrow female(X), \\ \kappa_4(X) \Longleftarrow sibling(X,Y), \kappa_4(Y) \Longleftarrow sibling(X,Y), \kappa_4 \Longleftarrow parent(X,Y), \\ \kappa_4(Y) \Longleftarrow parent(X,Y)\} \end{gathered}$$

Such rule insertion results in demanded structure.

8.3.1. Correlation maximization

The only process that is left to be lifted is the correlation maximization process. The task, which arisen from the lifting, is equal to maximization of predicate w.r.t. the output. The output used in LRNNs, as defined before, is one λ predicate with zero arity, thus the sum over residual output neurons' errors vanishes and each time E is used, residual error given by the only one output LRNNs' neuron is meant; E_s then means average of those errors equation (4.3).

Recall that the added cascade rules set is entailed by candidate predicate of type λ. Therefore the maximization process should be rooted in this predicate, or corresponding ground neurons, and run only through one layer of learnable weights. Since each predicate may be grounded multiple times within a ground network, possible having different bindings each time, multiple neurons may corresponding to a single predicate. The task, which has to be solves, is to aggregate these values in order to run the correlation maximization. We propose to firstly compute $\overline{V_s}$ equation (8.1), and average of output values of ground candidate neurons corresponding to candidate predicate; then to use these values to gradient ascent. The correlation is now given by equation (8.4), which is the change we had discussed, and its partial derivative is given by equation (8.5).

$$\overline{V_s} = \frac{1}{|bindedHead \in S|} \sum_{bindedHead \in S} V_{s,bindedHead} \tag{8.1}$$

$$\overline{V} = \frac{1}{|samples|} \sum_s \overline{V_s} \tag{8.2}$$

$$\overline{E} = \frac{1}{|samples|} \sum_s E_s \tag{8.3}$$

$$C = |\sum_s (\overline{V_s} - \overline{V})(E_s - \overline{E})| \tag{8.4}$$

$$\frac{\partial C}{\partial w_i} = \text{sign}(correlation) \sum_s (E_s - \overline{E}) f'_s I_{s,i} \tag{8.5}$$

Having defined those equations, arbitrary gradient ascent can be run. For this purpose stochastic gradient descent, developed for LRNNs, was taken and modified to perform correlation maximization of the given candidate predicate.

8.4. Lifted TopGen

The lifting of TopGen algorithm into the first-order level can be done in a few changes within the algorithm. Only the parts, where non-trivial lifting of TopGen's techniques occur, are described here in details. The rest of the method stays the same, only, informally speaking, each time a *neuron* is replace by a *rule*.

One of the simpler differences is deciding whether a predicate is *and* or *or*. This is given by the template, thus instead of biases' comparison, the question reduces to whether a given predicate is λ or κ. There are two more differences to be processed – counting of false positives or negatives, and rule extension technique to decrease these values.

8.4.1. Counting false positives and negatives

There is a major difference between propositional and ground networks. In the former, since there is only one neural network, it is ensured that a given neuron is there for ever sample's input. This does not hold in the later, since a predicate can bind in multiple ways. In each of these ways, it may produce different output value. Thus, while computing whether a predicate is false positive or negative, an aggregation technique has to be decided before further computation. One approach, finally applied, is to take an average over the predicate's ground neurons within one sample; it is the same aggregation technique as the one during correlation maximization.

The rest of incrementing false positives and negatives counters is the same as in propositional case. Recall that such counting is useful, and is done, only for neurons having bounded activation functions. The descent order for computed counters is the same; ties are broken in favour of the predicate with shortest maximal length to terminal predicate.

8.4.2. Decrease of false positives and negatives

Decreasing a false positives or negatives splits to four possible cases given by predicate meaning and whether it is a false positive or negative case. Recall that each predicate is strictly to be either λ or κ, thus mimicking either $\wedge$ or $\vee$. Therefore no comparison of a neuron's bias is done.

Before deeper analysis of the extensions, the idea of addition of new predicates is to be mentioned, because it occurs in all of these extensions. In these cases, a base rule set is generated for each new base κ that is to be added somewhere in the rule producing false negatives or positives. One may come up with an idea of reusing already constructed base rules, but such approach would be insufficient, because instead of giving freedom of weights of the newly created neurons, those would be shared with some already existing used neurons. This lifting is based on TopGen, although instead of creating base rules all the time, some other rule generator can be used. For the parallel with TopGen, a base rule set are used to describe the lifted case.

Let us start with λ predicate entailing a set of κ; we do not consider arities right now. In order to decrease its false negatives, outputs of its κ body predicates should be increased. Therefore, for every κ_i/a in its body, one new λ_{i_B}/a, for which new base

rules set, is generated; to be precise, a the set is constructed to be entailed by κ_{i_B}/a. Of course that corresponding rules in the form of $\kappa_i/a \Longleftarrow \lambda_{i_B}/a$ and $\lambda_{i_B}/a \Longleftarrow \kappa_{i_B}/a$ are created. Note that arbitrary variable ordering technique can be used to generate those rules, and it depends on user specification.

Example 8.4. In order to decrease false negatives of λ/a given $\lambda(X,Y) \Longleftarrow \kappa_a(X) \wedge \kappa_b(Y)$ and base predicates

$$B = \{male/1, female/1, sibling/2\}$$

the following set of rules will be created using root chain order:

$$\begin{gathered}\{\kappa_a(X) \Longleftarrow \lambda_a(X), \lambda_a(X) \Longleftarrow \kappa_{a_b}(X), \kappa_{a_b}(X) \Longleftarrow male(X),\\ \kappa_{a_b}(X) \Longleftarrow female(X), \kappa_{a_b}(X) \Longleftarrow sibling(X,Y), \kappa_{a_b}(Y) \Longleftarrow sibling(X,Y),\\ \kappa_b(X) \Longleftarrow \lambda_b(X), \lambda_b(X) \Longleftarrow \kappa_{b_b}(X), \kappa_{b_b}(X) \Longleftarrow male(X),\\ \kappa_{b_b}(X) \Longleftarrow female(X), \kappa_{b_b}(X) \Longleftarrow sibling(X,Y), \kappa_{b_b}(Y) \Longleftarrow sibling(X,Y)\}\end{gathered}$$

To end the extension of λ/a, let us take the former instantiation and describe reduction of false positives. Driven by the same idea, as in the propositional case, the rule body implying λ should be extended by a fresh new base κ/a.

Example 8.5. In order to decrease false positives of $\lambda/2$ given $\lambda(X,Y) \Longleftarrow \kappa_a(X) \wedge \kappa_b(Y)$ and base predicates

$$B = \{male/1, female/1, sibling/2\}$$

a base rule set entailed by fresh new $\kappa_1/2$ is created and the former λ rule, using root chain order, is replaced by:

$$\lambda(X,Y) \Longleftarrow \kappa_a(X) \wedge \kappa_b(Y) \wedge \kappa_1(Y,Z)$$

The created base rules set entailed by $\kappa_1/2$ is not shown, since it is similar to the ones generated in Example 8.4.

So far, λ extensions were covered, thus next κ extensions are described. Let us start with the simpler one, decrease of false negatives. Given κ/a, new base rule set, entailed by fresh new λ_b/a, is created. Thereafter, rule entailing λ_b/a by κ/a is created.

Example 8.6. In order to decrease false positives of $\kappa/1$ and base predicates

$$B = \{male/1, female/1, sibling/2\}$$

base rules set entailed by fresh new $\lambda_1/1$ is created and the former and new rule is produced:

$$\kappa(X) \Longleftarrow \lambda_1(X)$$

A base rules set entailed by $\lambda_1/$ is not shown, since it is similar to the ones generated in Example 8.4.

The last extension aims to decrease false positives of κ/a predicate. In order to do that, replace all occurrences of κ in rules' heads by a fresh new κ_i/a. Then create a base rules set entailed by fresh new κ_b. Thereafter, create a new λ/a rule that is implied by both κ_i/a and κ_b/a. Finally, entail λ/a by κ/a.

Example 8.7. In order to decrease false positives of $\kappa/1$, base predicates

$$B = \{male/1, female/1, sibling/2\}$$

and part of template

$$T' = \{\kappa(X) \impliedby \lambda_1(X), \kappa(X) \impliedby \lambda_2(X)\}$$

following predicates will be created: $\kappa_i/1$ as copy of $\kappa/1$, intermediate $\lambda_i/1$ and base κ_b. Both rules from T' will be removed. The following rules, beside the base ones, using chain root order, will be created:

$$\{\kappa(X) \impliedby \lambda_i(X), \lambda_i(X) \impliedby \kappa_i(X) \wedge \kappa_b(X),$$
$$\kappa_i(X) \impliedby \lambda_1(X), \kappa_i(X) \impliedby \lambda_2(X)\}$$

Base rules set entailed by $\kappa_b/1$ is not shown, since it is similar to the ones generated in Example 8.4.

A question arises to investigate real impact of this extension to decrease of false positives. By looking at the process, one can easily see that the κ_i is in the place where κ was originally, thus the problem may stay there. Contrary, κ' false positives may decrease.

8.5. Lifted REGENT

Lifting of REGENT differs from its propositional case only by lifted mutation and crossover operators. The only parts left to be lifted are neuron deletion and crossover, which are not direct lifting of the propositional world. Note that in lifted version of REGENT, mutation on freshly crossovered individuals was not take into account.

8.5.1. Rule deletion

Rule deletion arises by lifting neuron deletion. Lifting the process in every detail, a predicate should be removed from a template, however, we decided to delete only one rule. Moreover, the task of template mutation can be seen as nothing else than genetic programming fitted for this application.

The process is done in a very straightforward way. Given a template T, a randomly chosen rule from T with head p, which does not imply λ_f nor κ_f, is removed from T. After the rule is removed, a *reachability check* for every predicate is executed from root λ_f. The reachability check is a process, during which the theory is traversed as a tree, here given by clauses, from given root, returning a set of reachable and unreachable predicates.

8.5.2. Crossover

As noticed in the previous case of rule deletion, the task of changing template can be seen as a genetic programming. Lifting REGENT's crossover strategy in one to one relation is not an easy task. This is can be cause mainly by adding feed forward edges between neurons of two adjacent layers in the final crossover step. Splitting neurons to two different sets is harder for in the lifted level as well.

In spite of these facts, we propose simpler crossover based on genetic programming. The key idea is to switch parts of templates between two given parents producing two

new offspring; let call these parent templates T_1 and T_2. In order to do that, the crossover firstly selects either λ or κ, both having the same arity, from each template; let us refer them as p_i according to the parent index. Note that it is important to take the same predicate type from both templates, otherwise inconsistent templates would be created. Then, from each template a part rooted by the p_i is copied; let us call it S_{p_i}. Thereafter from template T_i all predicates and rules that are reachable from λ_f only through p_i are removed. Then names of both predicates are switched and S_{p_1} is added to T_2 with fresh names of predicates, except of the base ones; the second offspring is created in the same way using T_1 and S_{p_2}. Valid non-recursive templates are produced by this approach.

8.6. Structure Learning with Selective Forgetting in first-order level

SLSF is in the first-order predicate level, the same as in the propositional (section 4.6). General equations in that section correspond to the case for SLSF run on LRNNs. Looking in more details, the difference is in derivation of J' where J is may be defined differently than in LRNNs, but its derivation is different from the propositional view naturally; the reasons for that are shared weights in LRNNs. but this is no concern of us, since the structure learning backed by the regularization is hidden in the term containing ϵ.

8.7. On mixing strategies

Till now, design of transferring structure learning approaches from propositional to the first-order predicate logic was investigated, by lifting rule generation and insertion strategies. These two parts can be combined, resulting in new approaches.

Beautiful example of generalization is lifted TopGen's template extension. Instead of adding base rules set, multiple ways are possible. The idea behind TopGen's template extension is to add new learneable weights, so base rules set is right choice. However, there is a possibility of decrease of false negatives in a different way. For example, by adding a frequent pattern instead of base rules set. The rationale behind this roots from weight learning phase. Given a frequent pattern, rather a smaller number of learnable weights, than all of the possible combinations, created by adding a base rule, arise in the ground networks. Of course, this template change strategy assumes that no neuron is negatively correlated to the output. Thus limiting the capability of the produced template.

Speaking of Lifted REGENT, different mutation operations may be used. One such mutation is just using the random rule generator. Other techniques from relation learning may be used as well.

Both beam and local search can take any of these template change strategies as successors generators components.

9. Comparison of proposed first-order structure learning approaches

This chapter describes an experimental comparison of the proposed first-order NSI structure learning approaches. These approaches were incorporated into LRNNs engine and tested against original LRNN method and one standard relational learning method.

9.1. Datasets and methodology

Two datasets were used for the comparison. The first one is *Mutagenesis* (MUT) [107], a dataset containing 188 molecules labeled according to their mutagenicity. The second is *Predictive Toxicology Challenge on Mouse Rats* (PTC-MR) [108], a dataset containing 344 chemical compounds labeled according to their carcinogenicity.

Errors were estimated by 10-fold stratified crossvalidation. All experiments, except the kFoil's ones, used the same folds.

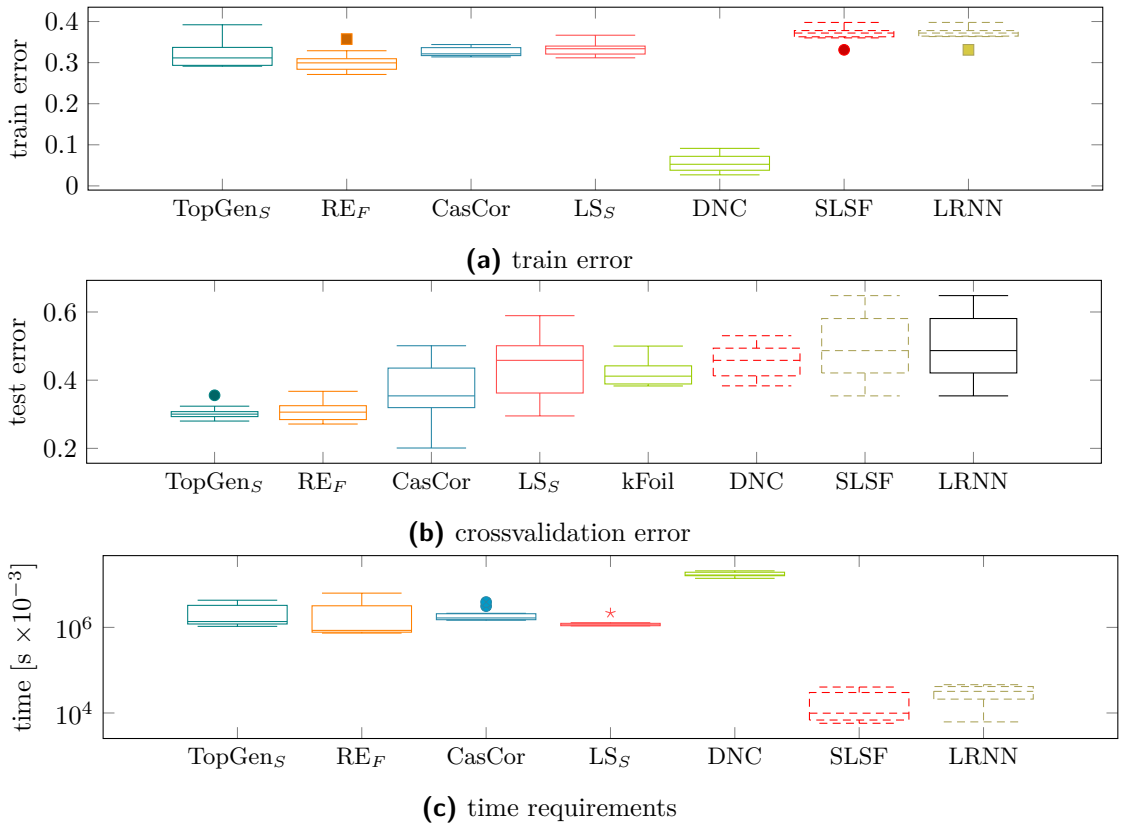
9.2. Compared methods

All of the proposed first-order structure learning approaches, as well as the original LRNN method, were tested on the two datasets. Methods are named by their propositional origin for simplicity of legends, instead of writing *lifted* everywhere; thus DNC, CasCor, TopGen, REGENT and LRNN corresponds to section 8.2, section 8.3, section 8.4, section 8.5 and Chapter 6 respectively. For the sake of readability, in some legends *CC*, *TG*, *RE*, *LR* and *LS* denote CasCor, TopGen, REGENT, LRNN and local search respectively. *Local search* corresponds to local search section 8.1, which generates successors by random rule generator with constraints section 7.4. Parameters used within these experiments are described in Appendix B. For comparison with standard relational methods, results of kFoil [109] were taken from [3].

9.3. Initial templates and parameters

All methods, except DNC and CasCor, need an initial template, therefore one for PTC-MR and three for MUT were constructed. The MUT's first template is very similar to the PTC-MR one; it composes of one layer of three base λ entailed by κ_f. The two next MUT's templates were based on the idea of having a layer of λ predicates corresponding each possible combination of three base κ. The difference between these two lies in the number of literals of these λ rules' bodies; the simpler and the complex one allow 4 and 7 literals respectively. These three templates will be denoted by subindex in order of appearance in this paragraph.

All methods were tested with chain root order, and sliding window or first window only options. Thus sliding window and first window only will be denoted by subindices S or F respectively. The only exception is local search which was run with random variable order. Another parameter, which can be tested only on DNC and CasCor, is



(a) train error

(b) crossvalidation error

(c) time requirements

Figure 9.1. Train and crossvalidation errors, together with time requirements of selected methods on PTC-MR dataset.

arity of κ_f; two values, 0 and 1, were tested. The subindex A will be used to denote that κ_f with arity 1 was used in the experiment. See appendix B.2 for more information about parameters' settings.

Note that LRNNs parameters were not tuned so much as in [3] for the sake of saving computational resources.

9.4. Evaluation

After all experiments were completed, we decided to show only the best result for each of the methods among their possible settings. However, the initial theory is not taken into account as a setting. Crossvalidation and train errors, together with time requirements for PTC-MR and MUT dataset are shown in Figures 9.1 and 9.2. Some methods, for example REGENT or local search, wasted all computational resources on some initial templates, and thus they are not shown.

Let us firstly start with methods that do not use initial templates, i.e. with DNC and CasCor. Considering MUT dataset, one can see that DNC found similar solution to kFoil, but it is prone to overfitting. Similar behavior of DNC can be found on the PTC-MR dataset, see Figures 9.1a and 9.1b. CasCor produced different results than DNC. It was outperformed by every other method on MUT, also having high train error. Contrary, on the PTC-MR dataset, CasCor produced similar error to kFoil. This is quite an interesting observation, however, more datasets would be probably needed for its proper confirmation.

Let us now compare results of methods that do start with some initial template.

The regularization based method SLSF did not produce any interesting results. More or less, it gives the same results as LRNN. This can be given by two facts: either the selective forgetting threshold was too low to be applied, or another regularization technique should be used in the context of neural networks with shared weights.

By observing results of TopGen, REGENT, local search and LRNN, one can notice that the results depend on the initial template. Each of these methods, except LRNN, should produce a better result than LRNN, because they provide some sort of heuristic extension of the original template in order to find a better one. Some of the initial templates used for MUT were extensively manually tuned in order to get better accuracy than standard relational learners. The second of these initial templates produces average accuracy around 15%, which is a comparably hard result to beat, even more difficult since the dataset is quite small. However, as one can see in Figure 9.2b, several of our newly proposed structure-learning methods found a template producing better accuracy.

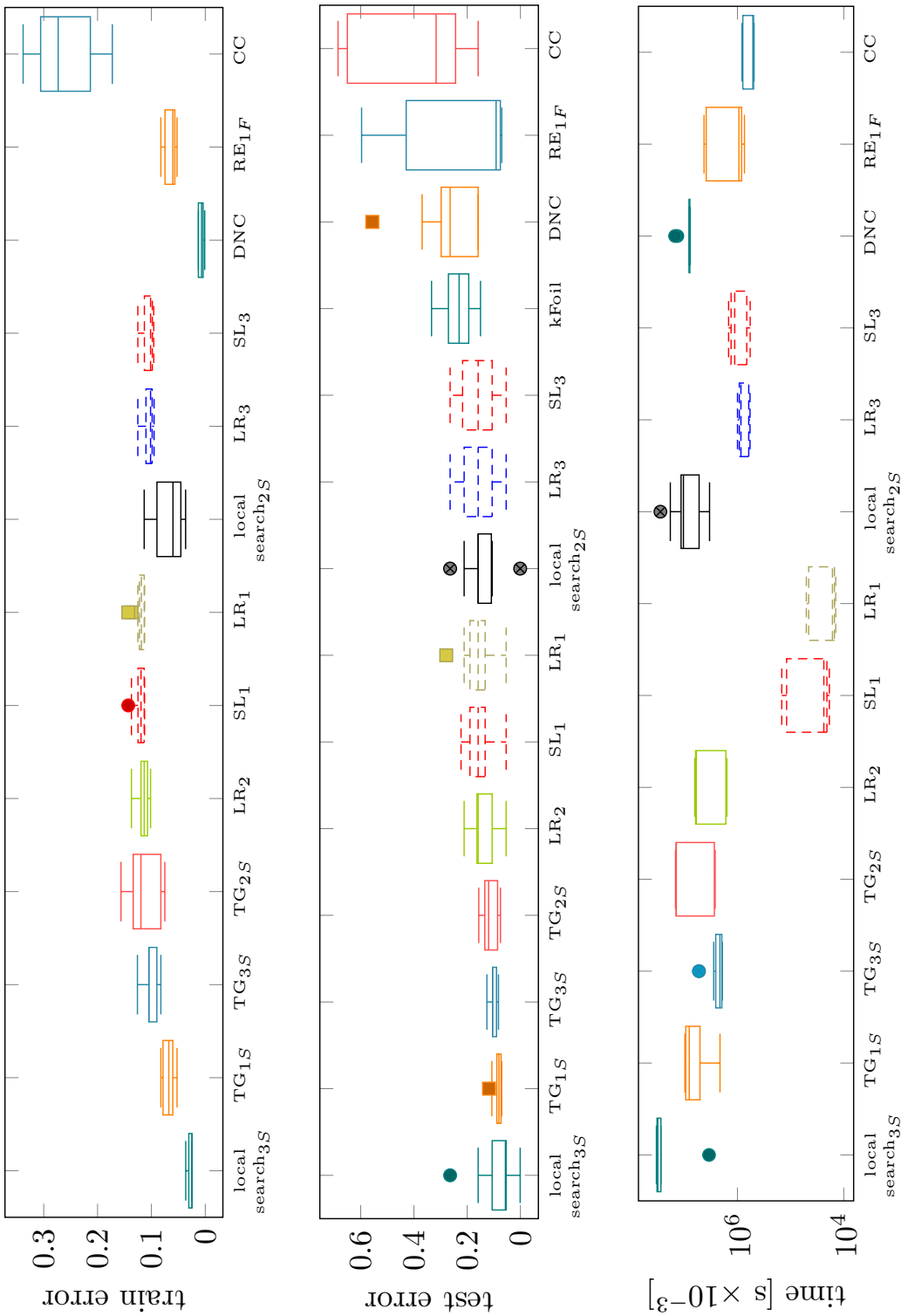
More interesting results are the ones in Figure 9.1b, because quite simple methods as TopGen and REGENT produced templates with app. 20% better accuracy than the original one learned by LRNN; having app. 8% better accuracy than kFoil. The interesting fact here is that LRNN produces worse accuracy than kFoil, given the same initial template as TopGen and REGENT. The results produced by TopGen are surprising because of two facts: the idea that backs TopGen does not hold in all cases, and TopGen did not produce any interesting results within propositional experiments.

In order to compare how much better the heuristic extensions can do as compared to the more simple random strategies, the local search, driven by random rule generator with constraints, was designed and implemented. One can see that the heuristic extensions, e.g. TopGen, really produce better results than such a simple random approach.

Comparing the methods from the computational side, it was expected that SLSF's and LRNN's computational times would be equal. Also, given the fact local search, TopGen and REGENT rely on complex search-based strategies, one can easily conclude that their training times would be high, since they can often produce computationally expensive templates. It was expected that CasCor and DNC would have higher training time than LRNN, but smaller than REGENT for example, but this does not hold for DNC. It is caused by a non-optimal implementation. For example, *incremental regrounding* would help a lot; this term stands for grounding only a certain extension of a template while reusing previous grounding of that template.

9.5. Conclusion

To sum up this chapter, three facts can be concluded. Firstly, DNC tends to overfit in the first-order NSI as well as in the propositional case. Secondly, results given by LRNN are dependent on given template. Thirdly, the idea behind TopGen seems to work well in the first-order NSI. In fact, TopGen produces better results on PTC-MR than state-of-the-art relational methods. Rest of the methods tend to behave differently from dataset to dataset and for their deeper comparison more dataset would be needed.



(a) train error **(b)** crossvalidation error **(c)** training time

Figure 9.2. Train and crossvalidation errors, together with time requirements of selected methods on MUT dataset.

10. Conclusion

This thesis presents a review of neural-symbolic integration methods from both propositional and first-order logic level. Several propositional methods were described and experimentally compared on 56 datasets. A transition to first-order predicate level was proposed for each of these methods. Proposed designs were incorporated with Lifted Relational Neural Networks and experimentally evaluated on two relational datasets. The results show that some of these approaches improve LRNNs accuracy, eventually leading to better accuracy than general state-of-the-art relational learners.

10.1. Future work

Although all the main goals of this thesis were accomplished, a set of new questions and ideas has arisen. Considering the developed propositional testbed, two interesting ideas can be further investigated. Firstly, it can be used, after some enhancements, to test and compare activation functions with respect to behaviors of logical operators within NSI.

Secondly, the propositional testbed can be used to test whether some of the methods is able of deep learning. More precisely, whether a given method is capable of learning latent concepts by introducing multiple dependent dataset's outputs.

For instance, let us have a dataset with a, b, c, d, e, f, g as inputs and two outputs. Let $\phi \Leftrightarrow a \wedge b$ and $\omega \Leftrightarrow \neg f \oplus (g \vee e)$. The first dataset's input will match $\phi \wedge \neg\omega$ while the second $\neg\phi \wedge \omega$. After learning of both the weights and the structure, the produced networks should be examined whether they have developed neurons corresponding to ϕ and ω.

The network examination can be done in multiple ways; e.g. by rule mining algorithm *TREPAN* [110], since it is the only one rule mining algorithm with existing implementation to our knowledge.

Considering the first-order predicate part, the most interesting question is to investigate TopGen deeper, because it worked in the FOL so well, but did not produce any interesting results on the propositional level. Other ideas that arose during the work in the FOL level can be described as follows:

- Incorporation of standard crisp techniques from relational learning, while using them as rule generators, might provide promising results. We believe that using e.g. frequent first-order rules could be more beneficial than inserting base rules all the time.
- Using a constraint, probably domain dependent, grammar each time, instead of proposed variable orders, is another potential source of speed up of the learning process.
- Learning a hyper-heuristic [111] using described rule creation and insertion strategies could be investigated.
- Invention of *incremental regrounding* to speed up learning process could be introduced. The term describes following situation. Given a template and its extension, only the difference between the two would be grounded as the rest would be taken

from the previously grounded templates, respectively. Although this would be theoretically and memory demanding process, it could result in huge improvements in speed up.

We believe that some of these extensions could achieve better accuracy or provide a speed up for learning process.

Appendix A.

Implementation notes

Both projects, propositional testbed and lifted structure learning approaches, have been implemented in Java 8. The implementation is focused on two things: modularity, allowing further extensibility, and parallelization, allowing speed up while executing experiments. Both codes are heavily based on usage of Java 8 streams and lambda functions. This technique often allows easy parallelization as well as coding style. On the other hand, source code reader who is not familiar with functional programming approach (especially streams) may struggle while reading the code.

A.1. Propositional testbed

Even though several parallelization enhancements have been implemented, they are no discussed in the text; for example, one of them is used in correlation computation. Moreover, everything what could be easily parallelized, was parallelized. While applying this approach, the fact that testbed has been evolved with heavy emphasis on stateful and stateless side of implemented classes, helped much.

The project, within which the propositional testbed is implemented, also contains the described formulae generator (it is dependent on [97]) as well as parameters setting generators.

A.2. Lifted structure learning approaches

Lifted structure learning techniques have been implemented to the already existing project [3]; the code is probably to be rewritten in following months in order to perform another experiments. There has been one big goal while designing the architecture needed for the lifted structure learning approaches. The goal was to separate predicate rule generation and rule insertion process apart. This was fully accomplished. Thus, one can easily extend Lifted TopGen to insert not only base rules, but arbitrary set of rules that is desired by a new structure learning approach.

Appendix B.

Experiments' parameters

B.1. Propositional experiments

All methods were tested using the same weight learning algorithm, Backpropagation with momentum in this case; its setting is in Table B.1. Other important parameters are shown in tables B.1 to B.3. Note that KBANN's and TopGen's parameters used in REGENT are the same as in the original methods; the same holds for TopGen's parameters that are used in REGENT.

During the experiments, all runs of one method shared the same random generator over one dataset; each of the runs was invoked separately in parallel inside the application. The application shared Java's *java.util.Random* random generator with seed equal to 13.

Finally, stopping criterion of each method should be mentioned. Each of these methods, except KBANN and SLSF, works in a cycle. At the end of each cycle, currently best solution is updated in case that the new solution is better. The loop is stopped when the learning has *converged*. The process converges if its error curve flattens. This is mathematically given by equation (B.1), where user specified l stands for long time window, s for short time window, ϵ_c for a threshold given by user, and the rest corresponds to the current loop: e_i is MSE error in time i. All experiments were run with $l = 30$, $s = 10$ and $\epsilon_c = 0.1$.

$$|average(e_{t-l}, \ldots, e_{t-1}, t) - average(e_{t-s}, \ldots, e_{t-1}, t)| < \epsilon_c \tag{B.1}$$

Table B.1. Parameters of Backpropagation, Dynamic Node Creation and KBANN

Backpropagation	value	DNC	value	KBANN	value
epoch	2500	δ_T	0.05	ω	4
learning rate	0.7	w	100	perturbation	0.3
momentum	0.3	neurons limit	200		
		c_a	10^{-3}		
		c_m	10^{-2}		

Table B.2. Parameters of Cascade Correlation and Structure Learning with Selective Forgetting

CasCor	value	SLSF	value
neurons limit	200	ϵ	10^{-4}
candidate size	4	θ	10^{-1}

Table B.3. Parameters of TopGen and REGENT

TopGen	value	REGENT	value
successors size	10	individuals selection	tournament
open list length	60	tournament size	2
learning decay	0.9	population size	40
neuron activation threshold	0.15	# mutations	24
neuron validation set : train set	1:4	# mutations of crossovered	8
		# crossovers	20
		# elites	1
		neuron deletion probability	0.5
		edge weight limit after crossover	0.2

B.2. First-order experiments

Lifted DNC and Lifted CasCor were allowed to add at most 20 base rules. All methods were allowed to 55 cycles of dataset groundings. Since the process of grounding grows exponentially, only templates having the longest path from λ_f equal to 5 and up to 3000 rules were allowed; bigger templates were trimmed in each dimension to satisfy this constrain. This constraint did not hold for Lifted CasCor, since it would limit the hierarchical structure too much.

Lifted REGENT and Lifted TopGen, were run with unActive threshold set to 0.15. Local search ran with both probability of λ and a new predicate creation equal to 0.5. The random rule generator could generate up to 2 rules in each run, with head arities ranging from 1 to 4. Detailed settings of parameters can be found in attached source codes.

Appendix C.

Content of DVD

Structure of the attached DVD is following:

- *codes/NESISL* – implementation of the propositional testbed
- *codes/Neurogical* – project containing lifted structure learning methods incorporated in LRNNs engine
- *thesis* – folder containing thesis in PDF
- *thesis/source* – LaTeX sources of the thesis

Bibliography

[1] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[2] M Bongard and Joseph K Hawkins. *Pattern recognition*. Trans. from the Russian, Moscow, 1967. New York, NY: Spartan, 1970.

[3] Gustav Sourek et al. "Lifted Relational Neural Networks". In: *arXiv preprint arXiv:1508.05128* (2015).

[4] Maarten H Van Emden and Robert A Kowalski. "The semantics of predicate logic as a programming language". In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 733–742.

[5] DE Rumelhart, GE Hinton, and RJ Williams. *Learning Internal Representations by Error Propagation, Parallel Distributed Processing, Explorations in the Microstructure of Cognition, ed. DE Rumelhart and J. McClelland. Vol. 1. 1986*. 1986.

[6] Tobias Blickle and Lothar Thiele. "A Mathematical Analysis of Tournament Selection." In: *ICGA*. 1995, pp. 9–16.

[7] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[8] Geofrey G Towell, Jude W Shavlik, and Michiel O Noordewier. "Refinement of approximate domain theories by knowledge-based neural networks". In: *Proceedings of the eighth National conference on Artificial intelligence*. Boston, MA. 1990, pp. 861–866.

[9] Geoffrey G Towell and Jude W Shavlik. "Knowledge-based artificial neural networks". In: *Artificial intelligence* 70.1 (1994), pp. 119–165.

[10] Jude W Shavlik and Geoffrey G Towell. "An approach to combining explanation-based and neural learning algorithms". In: *Connection Science* 1.3 (1989), pp. 231–253.

[11] Richard Maclin and Jude W Shavli. "Refining algorithms with knowledge-based neural networks: Improving the chou-fasman algorithm for protein folding". In: *in Computational Learning Theory and Natural Learning Systems*. Citeseer. 1992.

[12] Geoffrey G Towell and Jude W Shavlik. "Using symbolic learning to improve knowledge-based neural networks". In: *AAAI*. 1992, pp. 177–182.

[13] David W Pitz and Jude W Shavlik. "Dynamically adding symbolically meaningful nodes to knowledge-based neural networks". In: *Knowledge-based systems* 8.6 (1995), pp. 301–311.

[14] David W. Opitz and Jude W. Shavlik. "Connectionist theory refinement: Genetically searching the space of network topologies". In: *Journal of Artificial Intelligence Research* (1997).

[15] Fernando S Osorio and Bernard Amy. "INSS: A hybrid system for constructive machine learning". In: *Neurocomputing* 28.1 (1999), pp. 191–205.

[16] Fernando S Osorio and Bernard Amy. "Constructive Machine Learning: A New Neuro-Symbolic Approach". In: *UNISINOS–Computer Science Dept.-C6 Av. Unisinos* (), pp. 930022–000.

[17] Li Min Fu. "Knowledge-based connectionism for revising domain theories". In: *Systems, Man and Cybernetics, IEEE Transactions on* 23.1 (1993), pp. 173–182.

[18] LiMin Fu. "A neural-network model for learning domain rules based on its activation function characteristics". In: *Neural Networks, IEEE Transactions on* 9.5 (1998), pp. 787–795.

[19] Jadzia Cendrowska and Max A Bramer. "A rational reconstruction of the MYCIN consultation system". In: *International journal of man-machine studies* 20.3 (1984), pp. 229–317.

[20] LiMin Fu. "A concept learning network based on correlation and backpropagation". In: *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 29.6 (1999), pp. 912–916.

[21] LiMin Fu. "Knowledge discovery by inductive neural networks". In: *Knowledge and Data Engineering, IEEE Transactions on* 11.6 (1999), pp. 992–998.

[22] Steffen Holldobler, Yvonne Kalinke, Fg Wissensverarbeitung Ki, et al. "Towards a new massively parallel computational model for logic programming". In: *In ECAI'94 workshop on Combining Symbolic and Connectioninst Processing*. Citeseer. 1991.

[23] Pascal Hitzler, Steffen Hölldobler, and Anthony Karel Seda. "Logic programs and connectionist networks". In: *Journal of Applied Logic* 2.3 (2004), pp. 245–272.

[24] Artur S Avila Garcez and Gerson Zaverucha. "The connectionist inductive learning and logic programming system". In: *Applied Intelligence* 11.1 (1999), pp. 59–77.

[25] SM Kamruzzaman, Md Abdul Hamid, and AM Jehad Sarkar. "Erann: An algorithm to extract symbolic rules from trained artificial neural networks". In: *IETE Journal of Research* 58.2 (2012), pp. 138–154.

[26] Eduardo R Hruschka and Nelson FF Ebecken. "Extracting rules from multilayer perceptrons in classification problems: a clustering-based approach". In: *Neurocomputing* 70.1 (2006), pp. 384–397.

[27] Marco Botta, Attilio Giordana, and Roberto Piola. "FONN: Combining first order logic with connectionist learning". In: *ICML*. Citeseer. 1997, pp. 46–56.

[28] Marco Botta, Attilio Giordana, and Roberto Piola. "Refining first order theories with neural networks". In: *Foundations of Intelligent Systems*. Springer, 1997, pp. 84–93.

[29] Manoel VM Franca, Gerson Zaverucha, and Artur S d'Avila Garcez. "Fast relational learning using bottom clause propositionalization with artificial neural networks". In: *Machine learning* 94.1 (2014), pp. 81–104.

[30] Manoel VM Franca, Gerson Zaverucha, and Artur S d'Avila Garcez. "Fast Relational Learning using Bottom Clauses in Neural Networks". In: ().

[31] Hendrik Blockeel and Werner Uwents. "Using neural networks for relational learning". In: *ICML-2004 Workshop on Statistical Relational Learning and its Connection to Other Fields*. 2004, pp. 23–28.

[32] Werner Uwents and Hendrik Blockeel. "Classifying relational data with neural networks". In: *Inductive Logic Programming*. Springer, 2005, pp. 384–396.

[33] Scott E Fahlman and Christian Lebiere. "The cascade-correlation learning architecture". In: (1989).

[34] Werner Uwents and Hendrik Blockeel. "Learning aggregate functions with neural networks using a cascade-correlation approach". In: *Inductive Logic Programming*. Springer, 2008, pp. 315–329.

[35] Werner Uwents and Hendrik Blockeel. "A comparison between neural network methods for learning aggregate functions". In: *Discovery Science*. Springer. 2008, pp. 88–99.

[36] Franco Scarselli et al. "The graph neural network model". In: *Neural Networks, IEEE Transactions on* 20.1 (2009), pp. 61–80.

[37] Werner Uwents et al. "Neural networks for relational learning: an experimental comparison". In: *Machine Learning* 82.3 (2011), pp. 315–349.

[38] Mathieu Guillame-Bert, Krysia Broda, and Artur D'Avila Garcez. "First-order logic learning in artificial neural networks". In: *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE. 2010, pp. 1–8.

[39] Helmar Gust and Kai-Uwe Kühnberger. "Learning symbolic inferences with neural networks". In: *CogSci*. 2005, pp. 875–880.

[40] Helmar Gust, Kai-Uwe Kuhnberger, and Peter Geibel. "Learning models of predicate logical theories with neural networks based on topos theory". In: *Perspectives of Neural-Symbolic Integration*. Springer, 2007, pp. 233–264.

[41] R Goldblatt. *Topoi: The Categorical Analysis of Logic, Studies in Logic and the Foundations of Mathematics, Vol. 98*. 1979.

[42] Thanupol Lerdlamnaochai and Boonserm Kijsirikul. "First-order logical neural networks". In: *Hybrid Intelligent Systems, 2004. HIS'04. Fourth International Conference on*. IEEE. 2004, pp. 192–197.

[43] Yann Chevaleyre and Jean-Daniel Zucker. "A framework for learning rules from multiple instance data". In: *Machine Learning: ECML 2001*. Springer, 2001, pp. 49–60.

[44] Zhi-Hua Zhou and Min-Ling Zhang. "Neural networks for multi-instance learning". In: *Proceedings of the International Conference on Intelligent Information Technology, Beijing, China*. 2002, pp. 455–459.

[45] Xin Huang, Shu-Ching Chen, and Mei-Ling Shyu. "An open multiple instance learning framework and its application in drug activity prediction problems". In: *Bioinformatics and Bioengineering, 2003. Proceedings. Third IEEE Symposium on*. IEEE. 2003, pp. 53–59.

[46] Sebastian Bader et al. "A Fully Connectionist Model Generator for Covered First-Order Logic Programs." In: *IJCAI*. 2007, pp. 666–671.

[47] Sebastian Bader et al. "The core method: Connectionist model generation for first-order logic programs". In: *Perspectives of neural-symbolic integration*. Springer, 2007, pp. 205–232.

[48] Sebastian Bader, Pascal Hitzler, and Steffen Holldobler. “Connectionist model generation: A first-order approach”. In: *Neurocomputing* 71.13 (2008), pp. 2420–2432.

[49] Sebastian Bader, Artur S d’Avila Garcez, and Pascal Hitzler. “Computing First-Order Logic Programs by Fibring Artificial Neural Networks.” In: *FLAIRS Conference*. 2005, pp. 314–319.

[50] Artur S d’Avila Garcez and Dov M Gabbay. “Fibring neural networks”. In: *AAAI*. 2004, pp. 342–347.

[51] Mona F Ahmed and Nevin M Darwish. “FORN: First order rule based neural network”. In: *2010 2nd International Conference on Computer Technology and Development*. 2010.

[52] Oliver Ray and Bruno Golenia. “A neural network approach for first-order abductive inference”. In: *Proceedings of the fifth international workshop on neural-symbolic learning and reasoning (NeSy 09)*. 2009, pp. 2–8.

[53] Lokendra Shastri and Venkat Ajjanagadde. “From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony”. In: *Behavioral and brain sciences* 16.03 (1993), pp. 417–451.

[54] Lokendra Shastri. “Advances in Shruti—A neurally motivated model of relational knowledge representation and rapid inference using temporal synchrony”. In: *Applied Intelligence* 11.1 (1999), pp. 79–108.

[55] Carter Wendelken and Lokendra Shastri. “Acquisition of concepts and causal rules in shruti”. In: *Proceedings of Cognitive Science, Boston, MA* (2003).

[56] Lokendra Shastri. “SHRUTI: A neurally motivated architecture for rapid, scalable inference”. In: *Perspectives of Neural-Symbolic Integration*. Springer, 2007, pp. 183–203.

[57] Joe Townsend, Ed Keedwell, and Antony Galton. “Artificial development of biologically plausible neural-symbolic networks”. In: *Cognitive Computation* 6.1 (2014), pp. 18–34.

[58] Steffen Holldobler, Yvonne Kalinke, and Jorg Wunderlich. “A recursive neural network for reflexive reasoning”. In: *Hybrid Neural Systems*. Springer, 2000, pp. 46–62.

[59] Rodrigo Basilio, Gerson Zaverucha, and Valmir C Barbosa. “First-order Cascade ARTMAP”. In: (2001).

[60] Rodrigo Basilio, Gerson Zaverucha, and Valmir C Barbosa. “Learning logic programs with neural networks”. In: *Inductive Logic Programming*. Springer, 2001, pp. 15–26.

[61] Ekaterina Komendantskaya. “First-order deduction in neural networks.” In: *LATA*. Citeseer. 2007, pp. 307–318.

[62] Ekaterina Komendantskaya. “Unification by error-correction”. In: *Proceedings of NeSy* 8 (2008).

[63] Ekaterina Komendantskaya. “Unification neural networks: Unification by error-correction learning”. In: *Logic Journal of IGPL* 19.6 (2011), pp. 821–847.

[64] Gadi Pinkas, Priscila Lima, and Shimon Cohen. "Representing, binding, retrieving and unifying relational knowledge using pools of neural binders". In: *Biologically Inspired Cognitive Architectures* 6 (2013), pp. 87–95.

[65] Ekaterina Komendantskaya, Maire Lane, and Anthony Karel Seda. "Connectionist representation of multi-valued logic programs". In: *Perspectives of Neural-Symbolic Integration*. Springer, 2007, pp. 283–313.

[66] Marghny H Mohamed. "Rules extraction from constructively trained neural networks based on genetic algorithms". In: *Neurocomputing* 74.17 (2011), pp. 3180–3192.

[67] Rudy Setiono and Lucas Chi Kwong Hui. "Use of a quasi-Newton method in a feedforward neural network construction algorithm". In: *Neural Networks, IEEE Transactions on* 6.1 (1995), pp. 273–277.

[68] Mitchell Potter et al. "A genetic cascade-correlation learning algorithm". In: *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*. IEEE. 1992, pp. 123–133.

[69] C Lee Giles et al. "Constructive learning of recurrent neural networks: Limitations of recurrent cascade correlation and a simple solution". In: *Neural Networks, IEEE Transactions on* 6.4 (1995), pp. 829–836.

[70] Nicholas K Treadgold and Tamas D Gedeon. "Exploring constructive cascade networks". In: *Neural Networks, IEEE Transactions on* 10.6 (1999), pp. 1335–1350.

[71] Barbara Hammer, Alessio Micheli, and Alessandro Sperduti. "Universal approximation capability of cascade correlation for structures". In: *Neural Computation* 17.5 (2005), pp. 1109–1159.

[72] Sudhir Kumar Sharma and Pravin Chandra. "Constructive neural networks: a review". In: *International journal of engineering science and technology* 2.12 (2010), pp. 7847–7855.

[73] AG Ivakhnenko and GA Ivakhnenko. "The review of problems solvable by algorithms of the group method of data handling (GMDH)". In: *Pattern Recognition And Image Analysis C/C Of Raspoznavaniye Obrazov I Analiz Izobrazhenii* 5 (1995), pp. 527–535.

[74] Timur Ash. "Dynamic node creation in backpropagation networks". In: *Connection Science* 1.4 (1989), pp. 365–375.

[75] Liying Ma and Khashayar Khorasani. "New training strategies for constructive neural networks with application to regression problems". In: *Neural networks* 17.4 (2004), pp. 589–609.

[76] M Iqbal Bin Shahid et al. "A new algorithm to design multiple hidden layered artificial neural networks". In: *SCIS & ISIS*. Vol. 2006. 0. 2006, pp. 463–468.

[77] Somnath Mukhopadhyay et al. "A polynomial time algorithm for generating neural networks for pattern classification: Its stability properties and some test results". In: *Neural Computation* 5.2 (1993), pp. 317–330.

[78] K Mangasarian. "Neural network training via linear programming". In: *Advances in Optimisation and Parallel Computing* (1992), pp. 56–67.

[79] Asim Roy and Somnath Mukhopadhyay. "A polynomial time algorithm for generating neural networks for classification problems". In: *Neural Networks, 1992. IJCNN., International Joint Conference on*. Vol. 1. IEEE. 1992, pp. 147–152.

[80] Asim Roy and Somnath Mukhopadhyay. “Iterative generation of higher-order nets in polynomial time using linear programming.” In: *IEEE transactions on neural networks/a publication of the IEEE Neural Networks Council* 8.2 (1996), pp. 402–412.

[81] Masumi Ishikawa. “Structural learning with forgetting”. In: *Neural Networks* 9.3 (1996), pp. 509–521.

[82] Masumi Ishikawa. “Rule extraction by successive regularization”. In: *Neural Networks* 13.10 (2000), pp. 1171–1183.

[83] Wlodzislaw Duch, Rafal Adamczak, and Krzysztof Grabczewski. “Extraction of logical rules from training data using backpropagation networks”. In: *The 1st Online Workshop on Soft Computing*. 1996, pp. 19–30.

[84] Volker Tresp, Jurgen Hollatz, and Subutai Ahmad. “Network structuring and training using rule-based knowledge”. In: *Advances in neural information processing systems* (1993), pp. 871–871.

[85] Michael C Mozer and Paul Smolensky. “Skeletonization: A technique for trimming the fat from a network via relevance assessment”. In: *Advances in neural information processing systems*. 1989, pp. 107–115.

[86] Ehud D Karnin. “A simple procedure for pruning back-propagation trained neural networks”. In: *Neural Networks, IEEE Transactions on* 1.2 (1990), pp. 239–242.

[87] Giovanna Castellano, Anna Maria Fanelli, and Marcello Pelillo. “An iterative pruning algorithm for feedforward neural networks”. In: *Neural Networks, IEEE Transactions on* 8.3 (1997), pp. 519–531.

[88] Xin Yao and Yong Liu. “A new evolutionary system for evolving artificial neural networks”. In: *Neural Networks, IEEE Transactions on* 8.3 (1997), pp. 694–713.

[89] AG Ivakhnenko. “Polynomial theory of complex systems”. In: *Systems, Man and Cybernetics, IEEE Transactions on* 4 (1971), pp. 364–378.

[90] Jason Gauci and Kenneth Stanley. “Generating large-scale neural networks through discovering geometric regularities”. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM. 2007, pp. 997–1004.

[91] Sebastian Bader, Steffen Holldobler, and N Marques. “Guiding backprop by inserting rules”. In: *Procs. 4th Intl. Workshop on Neural-Symbolic Learning and Reasoning*. Citeseer. 2008.

[92] Tin-Yau Kwok and Dit-Yan Yeung. “Constructive feedforward neural networks for regression problems: A survey”. In: (1995).

[93] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5 (1989), pp. 359–366.

[94] S Fahlaman. “Fast-learning variations on backpropagation: An imperical study”. In: *Connecionist Models Summer School. Proc* (1998).

[95] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. DTIC Document, 1985.

[96] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: `http://archive.ics.uci.edu/ml`.

[97] Tyler Sorensen. *PyBool*. May 2016. URL: `https://github.com/tyler-utah/PBL/commits/master`.

[98] Vojtech Aschenbrenner. "Deep Relational Learning with Predicate Invention". English. MA thesis. Prague, CZ: Czech Technical University in Prague, 2013, p. 80.

[99] Angelika Kimmig, Lilyana Mihalkova, and Lise Getoor. "Lifted graphical models: a survey". In: *Machine Learning* 99.1 (2015), pp. 1–45.

[100] Luc Dehaspe and Hannu Toivonen. *Frequent Query Discovery: A Unifying ILP Approach to Association Rule Minging*. Katholieke Universiteit Leuven. Departement Computerwetenschappen, 1998.

[101] Siegfried Nijssen and Joost Kok. "Faster association rules for multiple relations". In: *International Joint Conference on Artificial Intelligence*. Vol. 17. 1. Citeseer. 2001, pp. 891–896.

[102] Nada Lavrac, Filip Zelezny, and Peter A Flach. *RSD: Relational subgroup discovery through first-order feature construction*. Springer, 2002.

[103] Amanda Clare, Hugh E Williams, and Nicholas Lester. "Scalable multi-relational association mining". In: *null*. IEEE. 2004, pp. 355–358.

[104] Stephen Muggleton, Cao Feng, et al. *Efficient induction of logic programs*. Citeseer, 1990.

[105] Luc De Raedt and Luc Dehaspe. "Clausal discovery". In: *Machine Learning* 26.2-3 (1997), pp. 99–146.

[106] J. Ross Quinlan. "Learning logical definitions from relations". In: *Machine learning* 5.3 (1990), pp. 239–266.

[107] Huma Lodhi and Stephen Muggleton. "Is mutagenesis still challenging". In: *ILP-Late-Breaking Papers* 35 (2005).

[108] Christoph Helma et al. "The predictive toxicology challenge 2000–2001". In: *Bioinformatics* 17.1 (2001), pp. 107–108.

[109] Niels Landwehr et al. "kFOIL: Learning simple relational kernels". In: *Aaai*. Vol. 6. 2006, pp. 389–394.

[110] Mark W Craven and Jude W Shavlik. "Extracting tree-structured representations of trained networks". In: *Advances in neural information processing systems* (1996), pp. 24–30.

[111] Edmund K Burke et al. "A classification of hyper-heuristic approaches". In: *Handbook of metaheuristics*. Springer, 2010, pp. 449–468.