Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Jan S t u d e n ý

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Learning Relevant Reasoning Patterns with Neuro-Logic Programming

Guidelines:

Incorporating common sense reasoning patterns into machine learning remains one of the challenges in Artificial Intelligence. The goal of this thesis is to practically demonstrate how, mainly symbolic, reasoning and, mainly statistical, learning may be tighly integrated with differentiable neuro-logic programming across diverse AI scenarios with different reasoning patterns, such as rule or similarity based reasoning with corresponding underlying learning patterns.

1. Get an overview of Artificial Intelligence methods and common underlying principles of continuous/discrete state space search. Focus mostly on learning and reasoning.
2. Get familiar with Statistical Relational Learning frameworks, primarily Lifted Relational Neural Networks.
3. Define suitable learning and reasoning problems, such as common sense classification where vague background knowledge may be given in addition to learning examples. Focus on extensions into relational setting.
4. Encode the diverse problems with relevant differentiable logic templates and showcase your learned solutions.
5. Discuss your findings, focus on generality of your approach, and compare with related work.

Bibliography/Sources:

[1] Getoor, Lise - Introduction to statistical relational learning - MIT press, 2007.
[2] De Raedt, Luc, Angelika Kimmig, and Hannu Toivonen - ProbLog: A Probabilistic Prolog and Its Application in Link Discovery - IJCAI. Vol. 7. 2007.
[3] Sourek, Gustav, et al. - Lifted Relational Neural Networks - NIPS 2015 Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches, 2015.

Bachelor Project Supervisor: Ing. Gustav Šourek

Valid until: the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 6, 2017

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's thesis

Learning Relevant Reasoning Patterns with Neuro-Logic Programming

Jan Studený

Supervisor: Ing. Gustav Šourek

26th May 2017

Acknowledgements

I would like to thank my supervisor Ing. Gustav Šourek so much for his invaluable support and optimism during the whole thesis. I would like to thank my entire family for their tolerance and sympathy during writing this thesis.

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague on 26th May 2017 ……

Czech Technical University in Prague
Faculty of Electrical Engineering

Citation of this thesis

Studený, Jan. *Learning Relevant Reasoning Patterns with Neuro-Logic Programming.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2017.

Abstrakt

Tato práce demonstruje schopnosti vylepšeného neuro-logického frameworku podchytit různé úlohy umělé inteligence, které jsou založeny na různorodých metodách uvažování. Základadem k tomuto frameworku je stávající engine nazvaný Lifted Relational Neural Networks.

V práci popisujeme nejčastější metody strojového uvažování používané ve statistických a symbolických metodách a také jak mohou být jednotlivé vzorce uvažování zakódovány do podoby navrženého neuro-logického programování. Dále se blíže zaměřujeme na schopnosti vyjadřování, které vzniknou kombinací obou přístupů.

Na vybraných příkladech z herního prostředí ilustrujeme, jak tento společný neurologický přístup rozšiřuje schopnosti již existujísích metod uvažování pracovat nad relačními strukturami při zachování výhod neurálního učení.

Klíčová slova metody uvažování, strojové učení relevantního uvažování,neuro-logické programování,strojové učení

Abstract

This thesis demonstrates the capability of an enhanced neuro-logic programming framework to capture diverse artificial intelligence tasks based on different reasoning patterns. The enhanced framework is building on existing engine called Lifted Relational Neural Networks.

We describe common reasoning patterns used in statistical and symbolic methods and demonstrate how each particular pattern may be captured from the perspective of the proposed neuro-logic programming framework.

We discuss the patterns in context of learning and reasoning and further focus more closely on abilities that arise from combination of both approaches. On selected examples from simple game environments, we illustrate how this joint neuro-logic programming approach broadens the scope of existing reasoning patterns through the ability to represent and reason with relational information while keeping the benefits of neural learning.

Keywords reasoning patterns, learning relevant reasoning patterns,neuro-logic programming,common sense patterns,machine learning

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Artificial intelligence (AI) has a goal for machines to exhibit rational, educated behaviour that is as good as, or superior to, human being. At the time of writing, this, rather philosophical, goal is far from fulfillment. Most researchers restrict themselves to pursue the mentioned goal only in some specific field instead. In some fields, the machines already surpassed human performance [1], but advancement in other fields such as image recognition or language translation is tough and unpredictable. Searching for a face in an image with hard-wired "if-then-else" structure proved infeasible, since detection of a face is not a task that has an explicit solution. Instead of studying the morphology of a face and hard-wiring every single aspect of it, researchers came up with a brilliant idea. Why couldn't a program *learn* to recognize a face by itself in a way similar to humans? To help with this elevated goal, statistical modeling and optimization techniques started to be employed to give a rise to the field of Machine Learning (ML).

In machine learning, we try to model complex systems $\mathcal{S}$ using generic architectures $\mathcal{M}$, such as neural networks or decision trees. In the process of learning we then try to optimize the fit of a model from $\mathcal{M}$ onto a system from $\mathcal{S}$. After some early attempts in symbolic AI, the majority of research in the field shifted towards vector representations of $\mathcal{S}$ and corresponding statistical methods for optimization of $\mathcal{M}$.

A considerable flaw of most of the current statistical machine learning tools is that the training data is the only source of knowledge. Moreover these training data have to be represented with real-valued vectors which limits the expressiveness scope of $\mathcal{S}$. Finally the interpretability of the learned models is often dubious. On the other hand, the symbolic AI approaches offer rich representation language, clear interpretability but a much weaker generalization through learning.

Thus to solve an AI task, we can typically either write a classical, structured, logical program explicitly encoding the solution, or rely purely on the power of statistical

[1] These are mostly simple axiomatic tasks, e.g., solving mathematical formulas, sorting numbers, finding shortest path in a graph, etc.

modeling. Approaches from in between these two worlds are thus subject to much of scientific inquiry.

1.2 Aim of the Work

The aim of this thesis is to show that there is a place for symbolic reasoning in statistical learning models, and to demonstrate that such a joint approach is able to capture variety of existing learning and reasoning patterns, as well as it allows for introduction of novel concepts that could not be expressed with the traditional, separate approaches. For the demonstrations, we build on a particular Statistical Relational Learning (SRL) framework combining logical reasoning with neural network learning, and instantiate the discussed reasoning patterns in simple and interpretable game environments.

1.3 Structure of the Thesis

The structure of the thesis can be divided into four parts.

Initially in chapter 2 - Background we discuss goals, views and approaches in Artificial Intelligence and introduce the concept of Machine Learning. Furthermore in section 2.4 - Statistical Relational Learning we discuss a particular field of machine learning - Statistical Relational Learning (SRL) into which this thesis might be classified. Chapter 3 - Related Work discusses different SRL frameworks.

The second part of this thesis, chapter 4 - Neuro-Logical Formalization discusses the rationale and defines an enhanced framework on top of existing SRL engine (Lifted Relational Neural network).

Next, in chapter 5 - Relevant Reasoning Patterns we firstly show the key reasoning patterns present in both statistical a symbolic learning models and their relation to our framework. Afterwards, we focus on combining the statistical and symbolic patterns with neuro-logic programming and show its benefits.

Lastly, in chapter 6 - Experiments the thesis illustrates the diversity of reasoning patterns on selected examples from simple game environments.

Chapter 2

Background

2.1 Artificial Intelligence

> *Artificial intelligence (AI) is intelligence exhibited by machines.*
>
> wikipedia.org, *Artificial learning*

Artificial intelligence has a goal for machines to exhibit rational, educated behaviour, that is as good as, or superior to human being. Beginning of Artificial intelligence dates back to post-war era, just right after the first computer was developed. It is a huge field containing different views on almost every possible non-trivial type of problem solved by humans. For the lack of any widely accepted definition, there is no easy answer to whether a particular problem solution belongs to the field of Artificial Intelligence.

There are even multiple approaches proposed on how to fulfill the AI goal. One approach is through Cognitive Science, where people try to do so by modeling human thinking. Cognitive scientists study human brain and try to incorporate that knowledge into a program that would correspond to human thinking.

A different approach is to build machines that act and think rationally. From this perspective, machines should make their best response to the environment, considering their actual and past observations, and the background knowledge they have. This approach appeals to most scientist because it is general, rationality is well defined, and the best response can be proved.

Historically, Artificial intelligence went through periods of success and optimism, but also lack of advancement which cut the funding and caused recession. Proposed ideas were often working well on toy examples, but generalization to bigger examples was typically computationally infeasible or inaccurate. A good example is the incentive from

1950's to translate Russian text to English. Small, simple sentences were translated quite well, but more difficult pieces went often out of control[2].

2.2 Machine Learning

In machine learning, we try to build models $\mathcal{M}$ that ideally respond as accurately as the original system $\mathcal{S}$ would given some input observation, without being explicitly programmed. Given this ability, the model may be seen as improving its performance P through time by the experience E it is presented in the form of the observations of the system $\mathcal{S}$ (i.e. its input-output data).

Machine learning is vital for AI because of this ability to adapt to a new environment - an ability that is impossible to achieve with any other existing approach. Also, as machine learning requires only the experience E, a program can learn to respond in problems that humans are unable to code. That includes, for example, the discussed face detection problem, but a variety of others, such as image classification, speech recognition, machine translation, etc.

Machine learning can be classified based on the type of E the program receives into 3 categories, (i) supervised learning (when the program is given a set of observations together with the responses of $\mathcal{S}$), unsupervised learning (when the program is given only the set of input observations of $\mathcal{S}$) and reinforcement learning (when the program receives a reward based on its own response).

2.2.1 Supervised Learning

Supervised Learning (also known as learning with a teacher) is probably the most commonly utilized type of learning. Let $T = \{(x_i, y_i)\}$ be a set of tuples, with x_i representing the input observation and y_i representing the $\mathcal{S}$'s response to it. This set T is called the *training set*. The goal of supervised learning is to learn an approximation of the transfer function $f : X \mapsto Y$ of the system $\mathcal{S}$ that outputs the best response $y \in Y$ given the observation $x \in X$ from the training set T. This is done via minimization of a given cost (also called objective or error) function, capturing the discrepancies between y_i, the actual response of $\mathcal{S}$, and the model's prediction. The aim of supervised learning is then to generalize from these known examples in T onto new, unseen observations of $\mathcal{S}$.

2.2.2 Unsupervised Learning

Unsupervised Learning (also known as learning without a teacher) is the type of learning where the training set is given without the $\mathcal{S}$'s responses as $T = \{(x_i)\}$. One example of such learning setting is the anomaly detection, where the underlying assumption is that most of the observations are normal and thus will be fitted by a generalizing learner M as opposed to the anomalous observations. Another example is cluster analysis with

[2]One of the "good" translations was: "the spirit is willing but the flesh is weak" translated as "the vodka is good but the meat is rotten"

the underlying assumption that similar observations will share the same, possibly latent, target concepts. For the lack of the ground-truth response from $\mathcal{S}$, there is no obvious way of measuring the quality P of an unsupervised learner.

2.2.3 Reinforcement Learning

This type of learning can be seen as a generalization of supervised learning. Learner doesn't receive any training set, but instead is rewarded based on his response in the environment he continuously observes. The learner tries to maximize a cumulative reward during a given period of learning. This type of learning is heavily used in robotic, because the reward can be extracted from the environment (e.g. whether robot moves towards the desired location or not), with the big advantage that there is no need to reward each response of an agent, but rather there can be a reward only for accomplishment of some high level goals (e.g. to get somewhere).

2.3 Symbolic Reasoning

Reasoning is the, equally important, counterpart to learning. Two distinct reasoning types are used prominently in the field of artificial intelligence. The first type, statistical reasoning, uses numerical features to reason about the environment. Second type of reasoning is symbolic reasoning that uses abstract symbols, expressions and processes that operate on top of expressions to produce other expressions that can be useful in a given context.

Symbolic reasoning is the foundation of logic and mathematics in general. For its formal nature, logical reasoning is a typical example of symbolic reasoning used in any field of artificial intelligence. Logical reasoning allows to unambiguously reason about facts, encode background knowledge and act rationally. We further discuss symbolic reasoning in more detail in section 5.2 - Symbolic Reasoning Patterns.

2.4 Statistical Relational Learning

Typical learning algorithms, such as Neural Networks or Support Vector Machines, require that each observation is a fixed-length vector of (real) numbers, as well as the output. While this assumption is common in the theory of modeling dynamic systems and it is a natural representation for many sources, e.g. images of the the same size, it is almost impossible to restrict more complex environments, e.g., words, to have a fixed length.

Relational Learning doesn't impose this fixed-length restriction, and while building on rich representation languages, such as the first order logic, the input can be a word, a tree structure, a graph, everything that can be represented in the given formal language. Aim of relational learning is not only to remove the fixed-length restriction but to exploit the structure of data. This means that if some objects relate to each other in some way, they should have similar properties. For example, if we want to classify a product, we

can do so not only based on its attributes, but also using the links and attributes of the related products.

Statistical Relational Learning then tries to add typical generalization properties from statistical learning on top of the rich representation languages in order to learn from complex, relational data that also exhibit uncertainty.

A direct advantage this strategy brings to common statistical models is parameter sharing. This occurs when there are hidden symmetries in the parameters of the system, and ergo the model, rendering the functionality of different parameters to be the same. This however may be only be captured on a higher level of abstraction than that of the original, statistical model such as neural network or SVM, resulting into often dramatic reduce of the number of parameters to estimate leading to faster and better generalization.

Chapter 3

Related Work

Statistical relational learning is still a rather unconventional field proposing multiple strategies on how to combine statistical and relational approaches to learning (and reasoning), and the body of related work is naturally vast. In this thesis, we will cover two representatives from the two most promising directions. The first direction is the strategy of lifting of graphical models, with Markov Logic Networks (MLN) as the most successful representative. Second, we will briefly introduce the strategy of probabilistic logic programming with the language of Problog as its core representative. Finally, we will introduce a SRL framework of Lifted Relational Neural Networks that is inspired by these strategies and which we will elaborate on more closely for the use in the rest of the thesis.

3.1 Markov Logic Networks

Markov Logic Networks [7] are a SRL framework that lifts Markov Networks to the expressiveness of first order logic (FOL). This means that they make it possible to use Markov Networks to learn from relational data and incorporate first order background knowledge. They do it by introducing a so called template, consisting of weighted FOL formulae representing the lifted model, which is then merged with (relational) data, also represented in (relational) logic, to finally create a Herbrand model [3] of the merged set, representing all possible worlds that can be inferred from the given situation. This process is commonly referred to as grounding, and in the ground model all atoms from the Herbrand model stand as nodes in a constructed markov network in which two nodes are connected iff their corresponding literals were grounded from the same FOL formula. The weights associated with the formulae then determine potentials of cliques in the constructed markov network and thus completely determine probabilities of all possible worlds.

The resulting ground markov network can then be used for inference and learning[3].

[3] In principle, for computational speed up only the smallest part of the whole network that can infer knowledge or learn weights is built

Advantage of MLN is the capability to work with the full expressiveness of FOL formulae. MLN allows existential quantifiers, usage of function symbols and is not limited to definite clauses, a common restriction of many other SRL frameworks. MLN has a well defined probability of formulae and if the weight of every formula goes towards infinity, then MLN infers all satisfiable formulas with probability 1. A big disadvantage is the tractability of learning, as even inference is NP-complete and is intractable in all but smallest domains. Instead of exact inference, probabilistic method are used. The same problems arises in learning, because maximizing the underlying log-likelihood is intractable as well.

3.2 ProbLog

Other promising SRL framework is ProbLog [6], which combines logic programming with probabilistic inference. ProbLog is a straightforward extension of Prolog which adds probability to each clause contained in a logical program. Each formula is then considered mutually independent. Probability of satisfying a query is then computed as a probability that the query will be satisfied in randomly sampled program. This is a neat idea that enables to express probabilities of complex events, beyond capabilities of ordinary probabilistic models, as they may be described in the first order setting. Again however, computing probability of a query is NP-hard, hence approximation algorithms are being used. For learning of the probabilities of clauses in a program from the training set, EM algorithm is used, which is again computationally expensive.

3.3 Lifted Relational Neural Networks

Lifted Relational Neural Networks (LRNN) [10] is another SRL framework that enhances neural networks to be able to handle relational data. In the lifting strategy, LRNNs are very similar in spirit to MLNs, and in the representation language to Problog. LRNN is a logical program described by a set of weighted first order definite clauses $\mathcal{N} = \{(C_i, w_i)|i \in \iota n\}$. Similarly to MLN, this logical program acts as a template for building ground models when presented with some relational data, only in this case, the ground models are neural networks instead of markov networks. To build a neural network from the set $\mathcal{N}$ and some given data, a least Herbrand model H is created and following steps are performed.

- For each ground fact from the data F_i create a *fact neuron* F^{θ}[4]. This neuron acts as an input and its output is always 1.
- Connect it to a corresponding *atom neuron* A^{θ} with the same signature and a weight w_i, corresponding to the truth value of the fact F_i.

[4] θ acts as a grounding substitution

- For every grounding of a definite clause, i.e. rule $R_i \in \mathcal{N}$, that has all atoms from the body satisfied, i.e. present in H, create a *rule instance neuron* RI^θ which has its inputs from the atom neurons (corresponding to all atoms in the body of R_i) with weight 1, and outputs to a *rule aggregation neuron* $Ragg^\theta$, also with weight 1. This aggregation neuron gathers all groundings of R_i that have the same grounded head literal.
- The aggregation neuron then connects again to an atom neuron A^θ corresponding to the R_i rule's head literal with weight w_j, associated with R_i in $\mathcal{N}$.
- These steps are repeated until all ground literals from H are connected in the network.

This is the procedure to build the ground structure of a network, but to fully describe the network behavior, activation functions are to be determined.

The fact neurons do not have any input so they do not need any activation function. Atom neurons gather all possible weighted explanations for an atom, so a corresponding activation function should have a high output whenever any of the rule-based explanations looks promising. For this purpose, authors [10] propose a sigmoidal approximation of Łukasiewicz disjunction.

Rule neurons represent instantiations of conjunctive rules, so they should behave as an approximation of a conjunction. In the LRNN [10], sigmoidal approximation of Łukasiewicz conjunction is used again.

Rule aggregation neurons gather the rule instance neurons and they should aggregate the individual instances of a rule into a single output. For that the authors propose two possible activation functions, maximum and average.

The built network can be used to estimate the truth value of any ground literal from H. To be able to estimate the truth value of the literal, given some observation in the form of weighted definite clauses $\mathcal{E}$, we just build a network from the set $\mathcal{N} \cup \mathcal{E}$. Because the underlying model is a feed-forward neural network, we can use simple stochastic gradient descend (SGD) based on back-propagation (BP) to learn the weights that correspond to the rules in $\mathcal{N}$. The most obvious difference from regular NNs is that for each example the ground NN may be different. This however does not cause any problem for generalization, because we tied all the weights using the single template $\mathcal{N}$.

Chapter 4

Neuro-Logical Formalization

We chose LRNN as a base for our proposed neuro-logical formalization of SRL problems because it combines two powerful modeling approaches, (i) neural networks that were theoretically proved to approximate every possible functional input-output correspondence, and First Order Logic that is expressive enough to formulate, and by resolution compute, wide range of fundamental problems in science. Restriction to Horn clauses does not impose any restriction on the structure of resulting NNs, and every possible NN can thus be represented using this formalization. Moreover, by building NNs using Horn clauses, every sub-network can be interpreted as proving a truth value of the atom in the sub-network's root. This sheds some light on how the output of the whole network is being recursively computed, which can be used to investigate and interpret the learned model.

To demonstrate the learning process for the model, encoding of the training examples and other core properties of our enhanced framework, we present a simple regression problem that illustrates the important features.

4.1 Proposed formalism

The following example aims to familiarize the reader with the proposed formalism and briefly illustrate its benefits. The goal for the learned model is to predict (compute) score of words in Scrabble[5].

The model is described as a weighted logic program, further referred to as a template. The template reflects the high-level structure of the resulting models and possibly some background knowledge about the problem. In Srabble for instance, we know that every letter affects the score of a word. To make the situation more interesting, we will consider that the same holds for bi-grams of letters. To encode the knowledge, we use Horn clauses

[5]Further, bonus points are added for selected bi-grams to demonstrate the use of rules with more than one literal in the body.

(also called rules). Each rule is in the following format:

```
0.0  score() :- letterA(X).   [lukasiewicz]
```
weight: 0.0 — rule head: score() — rule tail: letterA(X). — activation function (optional): [lukasiewicz]

Weight of the rule is either learnable (then the weight represents the initial value for learning with the exception of 0.0, where the initial weight value will be randomly generated) or fixed (fixed weights are enclosed in angle brackets). In our sample scenario, we do not know the weight of any letter, and so every rule in this example will have learnable weight. The template for this problem will look as follows[6].

```
0.0 score() :- letterA(X).
0.0 score() :- letterB(X).
...
0.0 score() :- letterE(X),letterR(Y),next(X,Y).
...
```

The training set consists of grounded rules. These rules act as examples to learn the right weights in the lifted (templated) model. Every example should be provable from the (learned) model. In this particular example, we know only scores for some words, the corresponding rules will be in the following format.

```
7.0  score() :- letterH(l1),letterI(l2),next(l1,l2).
```
desired output: 7.0 — query: score() — evidence: letterH(l1),letterI(l2),next(l1,l2).

4.1.1 Learning

Learning of the rule weights in a template works the same way as learning of a traditional neural network. We have to specify meta-parameters for learning such as the learning rate, number of epochs, number of restarts and so on (the full list of parameters can be found in Appendix A - Parameters for the learning). There is no definite guide on how to set them, but most of them have a default value that proved to be better than others in general. This is the only thing to set before running the program interpreter and start learning the weights.

4.2 Activation functions

Activation functions play important role in our neuro-logic formalism as they define how the antecedent of the rule determines the truth value of the consequent. Activation functions can be categorized into three groups w.r.t. to the logical operators they represent as $g_\wedge$,$g_\vee$,g_*. To see the rationale behind these groups, see Figure 4.1.

[6] for the whole template see Appendix B - Scrabble - template

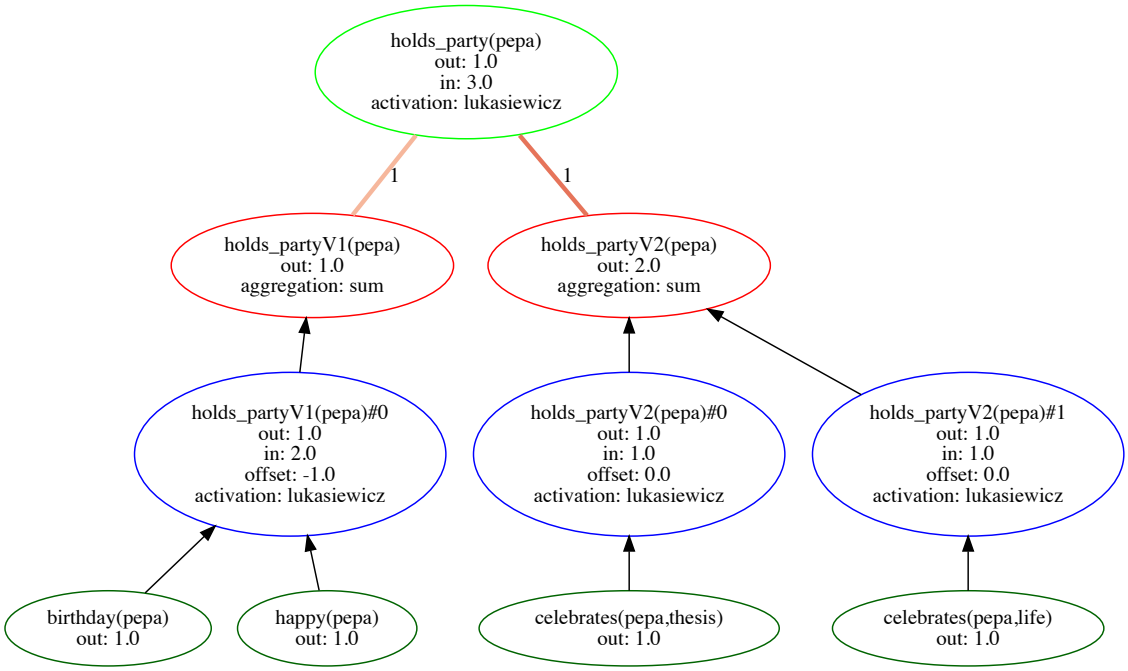


Figure 4.1: Neural network created from template with rules $holds\_party(X) \Longleftarrow birthday(X) \wedge happy(X)$ and $holds\_party(X) \Longleftarrow celebrates(X, Y)$. The blue neurons are rule instance neurons with activation function from g_λ. Red neurons are rule aggregation neurons with activation function from g_*. Green neurons are atom neurons with activation function from $g_\vee$.

4.2.1 g_λ

First group of activation functions, g_λ, defines how to calculate the value of a grounded body of a rule. Because we want to derive the truth value of the consequent, a useful property of g_λ would be to output high if all the input values are high. But what if some of the inputs are low? The right value depends on intention of the logical interpretation. If we assume Open World of the domain, we should output something neutral, maybe 0.5 if our atom truth values range from zero to one. If we assume Close World of the domain, we should probably output low value as we did not prove otherwise.

All possible functions for fuzzy conjunction are good representatives for this g_λ group. Beside interpretable meaning of the activation, we need to have a function that is differentiable because we need the derivative for the learning phase. This property rules out the standard fuzzy conjunction in the form of $\max_i(x_i)$ because the derivative is zero for all but the single value input with maximal value, which has a negative impact on the gradient descend. Good candidates for g_λ activation function we propose are as follows.

$$g_\lambda(x_1, \ldots, x_n) = sigm(\sum_i x_i - (n-1)) \qquad \text{(sigmoidal simple)}$$

$$g_\lambda(x_1, \dots, x_n) = sigm(6 * (\sum_i x_i - (n-1) - 0.5)) \qquad \text{(sigmoidal conjunction)}$$

$$g_\lambda(x_1, \dots, x_n) = max(\sum_i x_i - (n-1), 0) \qquad \text{(Łukasiewicz conjunction)}$$

4.2.2 g_*

Second group of activation functions, g_*, defines how to combine multiple proofs of the same grounded atom entailed by the same FOL rule. This aggregation intuitively corresponds to quantification over free variables used in the body of the rule. Two straightforward activation functions are inspired by existential quantification (described by max function) and universal quantification (described by min function). Another, more hybrid, approach between maximum and minimum is to use an average of the inputs. By using the average, every proof contributes to the final decision. This makes the average more robust to noisy inputs as opposed to max or min, where just a single noisy input may cause a noisy output. If used with caution, we also propose a sum function saturated between zero and one as another possible activation function. There, every bit of a proof supports the final output and even a lot of uncertain proofs can result in a high output.

$$g_*(x_1, \dots, x_n) = \min(x_1, \dots, x_n)$$
$$g_*(x_1, \dots, x_n) = \max(x_1, \dots, x_n)$$
$$g_*(x_1, \dots, x_n) = \frac{\sum(x_1, \dots, x_n)}{n}$$
$$g_*(x_1, \dots, x_n) = \min(\sum(x_1, \dots, x_n), 1)$$

4.2.3 $g_\vee$

Last group of activation functions, $g_\vee$, defines how deal with multiple FOL rules that entail the same grounded atom. Because the different rules act as alternatives to prove the same consequent, all types of fuzzy disjunctions are good candidates for this group. In situations where we want to satisfy a multitude of rules at the same time, we can substitute them with a single combined rule with the same head and merge of their bodies (this substitution will have the same interpretation as long as a proper fuzzy logic activation function is chosen for g_λ).

$$g_\vee(x_1, \dots, x_n) = sigm(\sum_i x_i) \qquad \text{(sigmoidal simple)}$$

$$g_\vee(x_1, \dots, x_n) = sigm(6 * (\sum_i x_i - 0.5)) \qquad \text{(sigmoidal disjunction)}$$

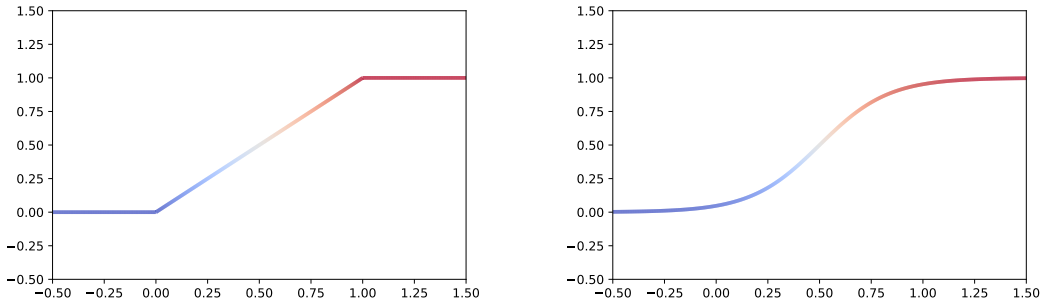
$$g_\vee(x_1, \dots, x_n) = min(\sum_i x_i, 1) \qquad \text{(Łukasiewicz disjunction)}$$

4.3 Activation functions' derivatives

Evaluation of created Neural Network doesn't impose any requirement on the shape of activation functions, but for the learning phase that is crucial for generalization we need to have a function where we know what happens if we shift the input by some ϵ. That's exactly the purpose of differentiation in the SGD strategy.

A central problem arises when, in the learning phase, we arrive at some plateau of the cost function that is to be optimized. At this point, the gradient is zero and SGD does not know where to move to lower the cost function. Since our objective function is generally not constant, certainly at the end of the plateau the function will rise or fall, but this information is not locally available for BP. This problem can be alleviated with a simple trick, where in the backpropagation phase we output a gradient of a smoothed variation of the original activation function instead. The reason why do not use this smoothed variant as the actual activation function as well is that it suppresses the logical interpretation where we would be missing the constant truth values of absolute *true* and *false*, respectively. As a result, a disjunction of "almost" false values may still result in a somewhat high output.

Particularly in this work, to help backpropagation from stucking at plateaus that can be found in both Łukasiewicz conjunction and disjunction, we utilized smoothed variants of these operators in the form of sigmoidal conjunction and sigmoidal disjunction defined in this chapter. The difference between those two functions can be seen in Figure 4.2 - Activation function plots.



(a) Łukasiewicz generalization of conjunction and disjunction

(b) Sigmoidal function used in disjunction and conjunction

Figure 4.2: Activation function plots

Chapter 5

Relevant Reasoning Patterns

In this chapter, we will introduce common learning and reasoning patterns in AI from the perspective of our neuro-logic programming framework.

5.1 Statistical Reasoning Patterns

Statistical reasoning patterns are typically based on some intuition from geometry. This is because the input (or the observation of $\mathcal{S}$) in statistical learning is restricted to the form of a fixed length vector of real numbers (or the observation have to be convertible to it). There are ways how to use variable length input (such as padding it with some constant or splitting it to number of fixed width inputs) but in its core form, the fixed width assumption must hold. With this assumption, every input can thus be described as a point in n-dimensional space.

5.1.1 Hyperplane Pattern

Let us first start with the simplest hyperplane reasoning pattern in the setting of binary classification. One of the easiest and most intuitive ways how to solve this problem is to pick a hyperplane (generalization of plane from 2D) and decide to which class the input point belongs, based on which side of the plane the point lies.

Even though this sounds as a rather naïve pattern, it is the basis for a number of statistical learning methods such as Logistic Regression, Perceptron and Support Vector Machines with a linear kernel. All of them try to find the best hyperplane that separates the corresponding points in the training set.

We can reason and learn in our neuro-logic framework with this pattern as well. The corresponding template for a generic separating hyperplane is simply as follows.

```
w_0 firstClass() :- coordinate(x_0).
...
w_(n-1) firstClass() :- coordinate(x_(n-1)).
w_n firstClass() :- true(). // offset variable
```

```
firstClass/0 [sigmoidal_disjunction_simple].
```

This correspondence brings a closer view on how we separate between positive and negative outputs through each of the rules in a template. Generally, if we choose appropriate cost function, we can even model some of the patterns mentioned more closely. For example, if we choose our cost function to be the cross-entropy loss ($C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln(f(x_i)) + (1 - y_i)\ln(1 - f(x_i)]$) we will learn the separating plane in a manner akin to Logistic Regression. If we decide to minimize the hinge loss ($C = \sum_{i=1}^{n} 1 - y_i f(x_i)$), we will emulate the rationale of the SVM's.

Extension to multiple classes can be done simply by creating a modified problem with a separating hyperplane for every class vs. the other classes, or with a pairwise separation. The resulting prediction can be taken as the one most far away from the separating plane.

The hyperplane pattern can be used as a regressor as well. Again, intuitively we should find a plane that minimizes the error between the prediction and the desired output value. The most commonly used cost function is the mean squared error (MSE) and in this particular linear case, the corresponding simple optimization method is named the Least Squares Method. For that purpose, almost exactly the same neuro-logic template is used as in the classification mode, with the exception that for the disjunction a simple sum is used (and with the exception that the output is not clamped between 0 and 1).

5.1.2 Transformation Pattern

Modelling an output value using only the hyperplane pattern works well if the data exhibit linear properties. If not, the separation or the regression doesn't work which results in poor training and testing errors. One way how to deal with this is to define more complex patterns and methods that build on top of these, or more easily, transform the input to a space where they do show linear properties. The transformation itself must be non-linear because the linear transformation only rotates and scales the input (affine transform also translates the input), resulting into a state where the same problem still remains. This is what SVM does with the kernel trick. For instance, using a polynomial kernel of degree d, the separating hyperplane in the transformed space corresponds to any polynomial function of degree d.

Transformation pattern is also the very essence of every deep neural network. They transform the input to a space where interesting (and finally linear) features emerge, and every the next layer combines these features from the previous layer into yet more complex concepts, until there is a clear correspondence to the target concept.

The neuro-logic framework takes essentially the same strategy. Activation functions that represent logical connectives are non-linear and we can infer the result from atoms that have been previously inferred (transformed) from the input data in a recursive fashion. There is no restriction on the length of a proof path and so the transformation can be stacked on top of another transformation and result into a multitude of deep neural networks. The following example illustrates this learning pattern.

5.1.2.1 Transformation example

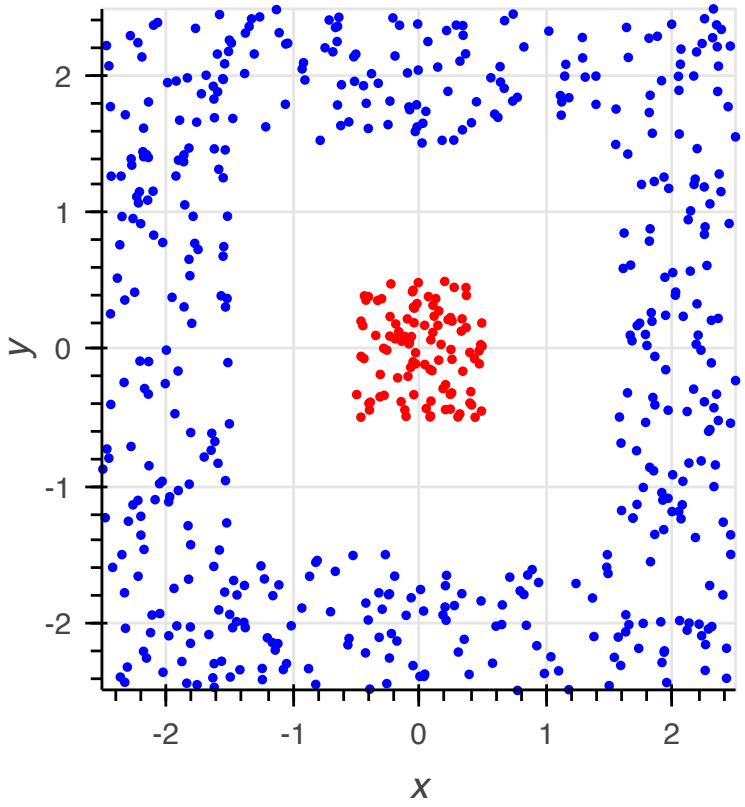


Figure 5.1: Image of red points in a center surrounded by blue points within a frame used as a motivation for the transformation pattern.

Given the Figure 5.1, learn a classifier that separates the blue points from the red. We can create a template that has 4 hidden neurons that transform the input image and then use the plane separation pattern in this newly created space.

```
w_0 hidden1()     :- coordinate(x).
w_1 hidden1()     :- coordinate(y).
...
w_7 hidden4()     :- coordinate(x).
w_8 hidden4()     :- coordinate(y).
w_9 firstClass()  :- hidden1().
...
w_12 firstClass() :- hidden4().
```

The hidden neurons are meant to learn the useful transformation that will make the points linearly separable. For this particular example, the hidden neurons learn whether a point is on the (left, right, upper, lower) part of the image or not. This transformation

will cause that a separating plane can be found, for example a plane that goes through a point on every axis that is at some distance ϵ from the origin. Thus a point that is on the border of the image will be classified as blue and in the center as red. A deeper network can learn such patterns in the first layer and combine them into more complex patterns in the next layer.

5.1.3 Similarity Pattern

As the inputs are situated in n-dimensional space, we can make use of some metric, or distance function (if a meaningful one exists) to compare a new input with the inputs from the training set in term of some inherent similarity given by the form of the metric used.

Probably the simplest (lazy) method in statistical learning, the k-Nearest Neighbours, uses this pattern. All it does is when it comes to evaluation a new input, it finds its k nearest neighbours from the training set. A combination of these neighbours (for classification the majority vote, for regression the weighted average with weights inversely proportional to their distances) gives the final decision.

For binary classification, more complex methods are typically used. Radial Basis Function (RBF) as a kernel for SVM is based on this neighbouring/similarity pattern, too. This is one of the kernel functions whose dimensionality of explicit feature space is infinite. As we cannot explicitly construct the separating plane, the learned model uses a biased weighted sum of RBF similarities between the input vector and the support vectors (points in the training set that determine the separating plane in the feature space). Intuitively, when we choose these support vectors (or rather centers) ahead, we can use some of the previous patterns to learn the final model. RBF NNs [9] use exactly this approach. As our framework can represent any FF NN, we can use the same principle as in RBF NN to handle the neighbouring pattern as well.

The neighbouring pattern also gives us the ability to search for neighborhoods, formally clusters, where objects in the same cluster are similar to each other. A generalization of this clustering pattern with the neuro-logic approach is described later in subsection 5.3.2.

5.2 Symbolic Reasoning Patterns

As opposed to statistical learning, symbolic approaches take a whole different view to model the world. Instead of numerical approximation of the original system, symbolic methods work with rules and symbols to derive the target concepts. One of the main advantages of this approach is the interpretability. We, as a human beings, are more used to reason about knowledge, derive conclusions, and test understandable hypotheses. All of this is even part of our natural language, the most powerful tool to reason about the surrounding world. On the other hand, most of us can't imagine nor reason in n-dimensional space of real numbers.

5.2.1 Relational Pattern

As we want to have a compact and unambiguous representation of statements, we have to use some formal language. Simplest formalization for symbolic reasoning is propositional logic. It allows for a statement to have the form of a proposition (e.g. "It is raining") or multiple propositions connected with logical connectives (or, and, not, if-then, …). Propositional logic naturally captured in the neuro-logic programming (if we restrict ourselves to Horn clauses) and so every propositional statement can be written in a template.

A central problem with propositional logic is that we cannot encode any statement that generalizes over sets of objects, i.e. that relates between different sets. An example of non-propositional statement is "All humans are mortal". The ability to take into account the sets of objects is important if we want to capture relations. Without this generalization, if we have, for example, n people that can go to m places, we would have to have $n * m$ statements of what can happen. Exactly this problem is targeted by first order logic via addition of predicates (relations), variables and quantifiers. We can then rephrase the previous sentence as: "All objects (quantifier) that are humans (relation) are mortal (implication to another relation)", or formally: $\forall x(\texttt{human}(x) \implies \texttt{mortal}(x))$.

This is the expressive power of the neuro-logic templates (again if we restrict ourselves to horn clauses) and most of the other SRL methods based on (subsets of) first order logic. With predicate logic we can encode variety of structures, such as graphs (edges between nodes will be interpreted as binary relation) and hypergraphs, and these structures can be used to encode molecules, relations between products, people, and so on.

5.2.2 Deductive Reasoning Pattern

Formal reasoning is the cause why logic as a formalism is so popular in symbolic approaches and what differentiates it from other representations (such as graph or special database formalisms).

Deductive reasoning is the process of arriving from premises (that are known to be true) to logical conclusions. To reach a conclusion, rules of inference are used. One of the rules is for example the "modus ponens", which states: "If P implies Q and P is asserted as true, then Q must be true as well". There is an infinite number of logical conclusions from premises. If P is true then by disjunction introduction rule the "P or Q" is one conclusion, "P or Q or Z", is another one, and we can continue endlessly. Rather than deriving all possible conclusions, we will focus on whether some conclusion follows directly from premises. For example if we have the following premises:

(A1): It is raining.
(A2): If it is raining I need an umbrella.

We can ask whether we need an umbrella.

One method for proving the validity of a consequent is the resolution method. It works by negating the consequent, adding it to premises and trying to deduce a contradiction. Resolution is refutation complete and so if a proof to the consequent exists, it

will be found. Resolution method is used in neuro-logic programming the same way as it is in Prolog [3]. In contrast to Prolog which stops when the first proof for the query is reached, we find all possible ways how to prove it. Subsequently, merging all of the proof trees will result into a directed acyclic graph (as parts of the different proof trees may be shared). This graph is then transformed (as described in section 3.3) and interpreted as a neural network.

The interpretation as a proof graph also sheds some light on how a generic neural network can be understood.

5.2.3 Inductive Reasoning Pattern

Deductive reasoning tries to prove a specific fact from some general knowledge. Inductive reasoning goes the opposite direction. It tries to "prove" a general statement from specific facts. For an example, if we see multiple times that the sun rises in the morning, we can conclude that sunrise *probably* happens in the morning. We can then use that general rule to predict that tomorrow the sun will rise in the morning as well. The word probably is important because, as opposed to deductive reasoning where everything we concluded was assured to be valid, here we are only assuming it is the case given some strong evidence.

One of the most successful methods for learning in the symbolic formalism, inductive logic programming (ILP) [5], uses inductive reasoning to learn the generic rules from a set of relational examples.

In its basic setting, ILP is given a background theory (B), set of positive examples (E^+), and a set of negative examples (E^-), and its goal is to derive a hypothesis (H) such that:

$$B \wedge H \models E^+$$

$$B \wedge H \wedge E^- \not\models false$$

If we restrict ourselves to the horn clauses, then the positive examples are given as facts and negative examples as headless rules.

ILP can be used to extend the neuro-logical templates with new rules (except for the weights that have to be optimized with different methods), or can be used to create the whole template from scratch.

5.3 Common Sense Reasoning Patterns

Both statistical and symbolic patterns lack the core advantages the other approach is offering. Statistical patterns lack the rich representation formalism and abstract reasoning from the domain of symbolic patterns. Symbolic patterns, on the other hand, lack the ability to capture uncertainty, robustness against noise, and strong generalization properties. Common sense reasoning patterns are patterns that, similarly to natural human reasoning, try to take the advantages of both approaches.

5.3.1 Fuzzy reasoning pattern

In section 5.1 - Statistical Reasoning Patterns we looked at neuro-logic programming as a statistical tool, where the presented rules did not have any interpretable meaning and we restricted ourselves to non-relational setting (all the used predicates were already ground, resulting into regular neural networks' functionality). On the contrary, in section 5.2 - Symbolic Reasoning Patterns we used the framework as a theorem prover and thus there was no generalization (i.e. learning). The rules had their meaning, but they were all of the same strength. In classical logic, a rule or a fact either holds true or does not. Therefore in the theorem proving setting, all the weights were kept to either 1 or 0.

If we allow the facts to have fuzzy truth values (i.e. to take on any value between 0 and 1), use max function as aggregation and max function also for disjunction, we obtain a standard fuzzy theorem prover. The output for any query is then a lower bound of its truth value.

Moreover, we can further relax the restrictions, and allow even the weights of the rules to be between 0 and 1. The rules will be fuzzy and their weight will act as a lower bound for their fuzziness.[7] Then we still obtain, now a more general, fuzzy theorem prover. Output for a query will be again a lower bound for its truth value.

Finally, if we remove all restrictions on the weights, we can encode both statistical and symbolic patterns and all their combinations.

5.3.2 Object similarity pattern

If we have a large number of objects (a fine description of the environment S), we may find that the variety and abundance of their parameters hides the underlying characteristics of S. If we do not restrict ourselves in some way, there can be a problem with overfitting (because the model will have too many learnable parameters compared to the number of input observations of S and we will fit to noise rather than to general characteristics of S).

One way to deal with this problem is to embed the objects into a much smaller set by extracting only the important characteristics that are useful for further reasoning. Object similarity pattern tries to group the objects into clusters, so that even when the objects are different, but for further reasoning have similar characteristics, they are treated similarly w.r.t. their correspondence to the clusters. As opposed to normal clustering, the characteristics that induce the clusters are often hidden, i.e. the similarity metric has yet to be learned. In common sense reasoning, this similarity pattern induces *soft clustering* which also differs from normal (crisp) clustering in that an object can belong to more than one cluster, and that the object has a soft (or fuzzy) membership to each of those.

[7] To understand the weight of a rule, one can view it as a strength of the rule. If the weight of the rule is small (below one), even a true body will not influence the head much, because the rule is unreliable. On the other hand if rule has a large weight (above one), proving only a bit of possibility for the tail will result in almost true head.

5.3.2.1 Soft clustering example

Suppose that we have a database of people with food they ate and whether they had (and how severe was) their allergic reaction. Obviously, we will want to predict whether some new people will have allergic reaction from each food and how severe it will be.

The training set in our framework will look like:

```
0.9 allergicReaction(pepa)          :- ate(ratatouille,pepa).
0.1 allergicReaction(franta)        :- ate(hamburger,franta).
...
```

This is the case where soft clustering is useful because we can group the foods (and therefore people as well) based on the unobserved allergic ingredients. The allergic ingredient is a hidden characteristic of each food and is not mentioned anywhere, but hopefully it will be extracted (learned) from the examples by the means of soft clustering. The grouping (if we have m different groups of meals and n different meals) will be encoded into a template as follows.

```
w_1 ateFromGroup1(Person)          :- ate(ratatouille,Person).
...
w_n ateFromGroup1(Person)          :- ate(hamburger,Person).
...
w_(n*m) ateFromGroupM(Person)      :- ate(hamburger,Person).
```

If we knew more information about the situation (such as whether a person was ill or whether he/she was stressed) we could further reason about the allergic reactions, but for the simplicity of this example, we will only consider the person being allergic to the specific types of food (those that have an ingredient the person is allergic to). The rest of the template will then be (if we have o people) as follows.

```
u_1 allergicReaction(pepa)        :- ateFromGroup1(pepa).
...
u_m allergicReaction(pepa)        :- ateFromGroupM(pepa).
...
u_(m*o) allergicReaction(franta) :- ateFromGroupM(franta).
```

The benefit of using this soft clustering pattern is that a meal can have more allergic ingredients and so be part of more clusters, and also a (hidden) quantity of that ingredient may define the allergic reaction. These nuances cannot be captured with normal clustering.

5.3.3 Structure similarity pattern

All of the similarity patterns share the same idea that similar inputs (under a given metric of similarity) should produce similar outputs (in a model based on that metric of similarity).

The incorporation of symbolic approaches allows us to also exploit the inner structure of examples, so that we can meaningfully measure similarity of objects in a much more

complex setting, e.g. based on their relations to other objects. A scientific field where this pattern is particularly useful is, for instance, molecular chemistry, where not only the contained atoms but mainly their structural conformations determine the final behavior of a molecule.

In common sense reasoning, the structural patterns that determine the similarity are again not crisp, generalizing regular pattern matching abilities of existing systems to *soft matching* where a pattern may be matched with different levels of confidence. Following example showcases this pattern with a learnable confidence level.

5.3.3.1 Soft matching example

Suppose that we have a company with lots of departments that is interested in knowing what makes their employees satisfied. We have an average satisfaction of every department. We think that in addition to various metrics, also relations between the employees affect their satisfaction. If they work with each other then their relationships are good, which should result in overall high satisfaction. On the other hand, if they have no relations between themselves, they may be dissatisfied with the colleagues and also with the working conditions. Here is how we may encode the mentioned situation:

```
<1.0> workTogether(Someone,Otherone) :- worksWith(Someone,Otherone).
<1.0> workTogether(Someone,Otherone) :- worksWith(Otherone,Someone).
 w_0  workTogether(Someone,Otherone).
<1.0> knowsEverybody(Someone)        :- workTogether(Someone,Otherone).
 w_1  overallSatisfaction()          :- knowsEverybody(Person).
 w_2  overallSatisfaction()          :- ... .
...
```

The soft matching occurs in the `knowsEverybody` rule[8]. We think that there should be a difference between not knowing any single person and knowing all (except one). The weight `w_0` varies the similarity between not knowing 0,1,2,… people. The lower the weight, the lower the similarity and faster the decay of satisfaction. The weight `w_1` represents the significance of the relationship between employees on the overall satisfaction. If the relationship assumption was false and the satisfaction does not correlate with their relationships, we would see the weight being learned (close to) zero value.

5.3.4 Incorporation of background knowledge

As we were writing the templates for the described examples, we were incorporating some form of background knowledge into the template to help the optimizer find the best representation of the model.

In the neuro-logic programming, we can incorporate variety of types of background knowledge and also control its strength. We can reason about the input, derive meaningful characteristics and use them further for more generic reasoning in the higher levels of abstraction. If we want to rather encode some vague intuition, we can keep the weights

[8] its aggregation activation is chosen to be the average function

learnable so that the optimizer decides on how to deal with our proposals to derive the output.

We do not need to reason straight on the input level of abstraction, but we can transform the data (e.g. group them as in the allergy example in subsubsection 5.3.2.1) and reason on top of that transformed representation. Also the transformation does not need to be done in a single step as we can, e.g., use a multi-layered (lifted) convolutional network to extract some important characteristics (lets denote it $Char$) and reason on the top of the classical neural network instead. Moreover, if we can additionally label the training examples with the right value of the $Char$, we can append them to the training set and learn the representation for $Char$ and the desired outputs simultaneously.

Chapter 6

Experiments

In the previous chapter, we have theoretically discussed key patterns used in purely statistical, purely symbolic and mixed approaches in AI. In this chapter, we will demonstrate in detail how these patterns may actually be learned with the proposed framework. For the demonstration, we chose simple, interpretable game environments of Scrabble, Tic-Tac-Toe and Poker. For mainly the statistical part, we chose a simple Scrabble score evaluator. For mainly the symbolic part, we chose Tic-Tac-Toe evaluator. Finally for the common sense patterns, we present a simple Poker cards figure score evaluator. In their basic form, both the Scrabble and Poker card scores present a regression task. However, we can either model the score itself or transform the problem into to a classification setting, which will correspond to a task of predicting the winner of a two player game.

The examples and their evaluation shown in this section are, by no means, meant to compete with state-of-the-art methods in terms of performance, but they are rather meant to practically demonstrate how the diverse concepts described in the previous Chapter 5 may be captured with our unified, neuro-logic programming framework.

At its core, the framework translates the template combined with an example to a Feed Forward NN and optimizes the weights using SGD, which is a standard technique taking place in almost all types of neural learning. From this follows that we could choose any common problem solvable with standard neural networks (including convolutional NNs) and achieve the same accuracy because in principle there is no difference between our architecture and these regular NNs for these problems, both in terms of network structure and learning[9]. As this would clearly not present any interesting insight, we rather focus of problems requiring solutions from the common sense reasoning category (Section 5.3) that cannot be captured with regular NNs[10].

[9]The grounding phase introduces some overhead but the overall complexity is still the same as building a normal NN

[10]With the exception of the Scrabble evaluator, which serves as a continuation of the introductory example to describe the very basics of the framework.

6.1 Scrabble

In this example, we continue the brief introduction of neuro-logic programming started in section 4.1 - Proposed formalism, to illustrate the first complete example of the whole neuro-logic learning workflow.

Each weighted rule in the template reflects the flow of information from the body of the rule to its head. In case of scrabble, every letter in a word modifies the score of the whole word. Because each letter has a different effect on the final score, each one also has a different rule in the template, too.

Next thing we need to describe are the activation functions used for each of the rules and each of the contained predicates. The activation functions for the rules reflect how the body truth value affects the head's value. Activation functions for predicates represent how the different rules that prove the same grounded predicate are being combined.

Because in this example all of the rules differ only in the particular letter they capture, we use the same activation function for all of the rules. As a conjunctive activation g_λ, we choose Łukasiewicz conjunction[11]. For the aggregative activation g_* we use the sum function, as we want to add the scores of each particular letter, even if it is used more than once. Finally for the disjunction $g_\vee$, we are again adding individual contributions of every letter so the sum activation function is used again.

6.1.1 Learning phase

First step of the learning phase is the construction of neural networks. In traditional neural network models, every example has fixed length input and output, and a single network is used to evaluate each example. In our neuro-logic (NL) programming framework, the example consists of a desired output value and a rule that is partitioned into head, representing the query we are asking to have the desired output value, and the body, representing the evidence we are given for it. The constructed NN in NL programming is defined as the merge of all the proof trees for the query query given the evidence, that may naturally differ for different examples. Therefore, instead of one main network, every example has its own neural network. See the Figure 6.1 - A ground neural network created from the word "enjoy" and Figure 6.2 - A ground neural network created from the word "career" for the example of two different neural networks (automatically) built from two different examples.

To be able to generalize from the training set, all the networks share the same set of weights. If the neural networks built from examples share some part of their proof path, they will also share all the weights on that path. In the words "career" (Figure 6.2) and "enjoy" (Figure 6.1), the "e" letter is shared which results in sharing the same weights in the path from the predicate `letterE` to the to root.

After the construction of networks, the optimization phase starts. Similarly to regular NNs, the algorithm starts with forward propagation in each of the networks to determine

[11] In this particular example, any function that goes through zero and is non-zero otherwise works, because we can compensate the difference from Łukasiewicz with the weight of the rule

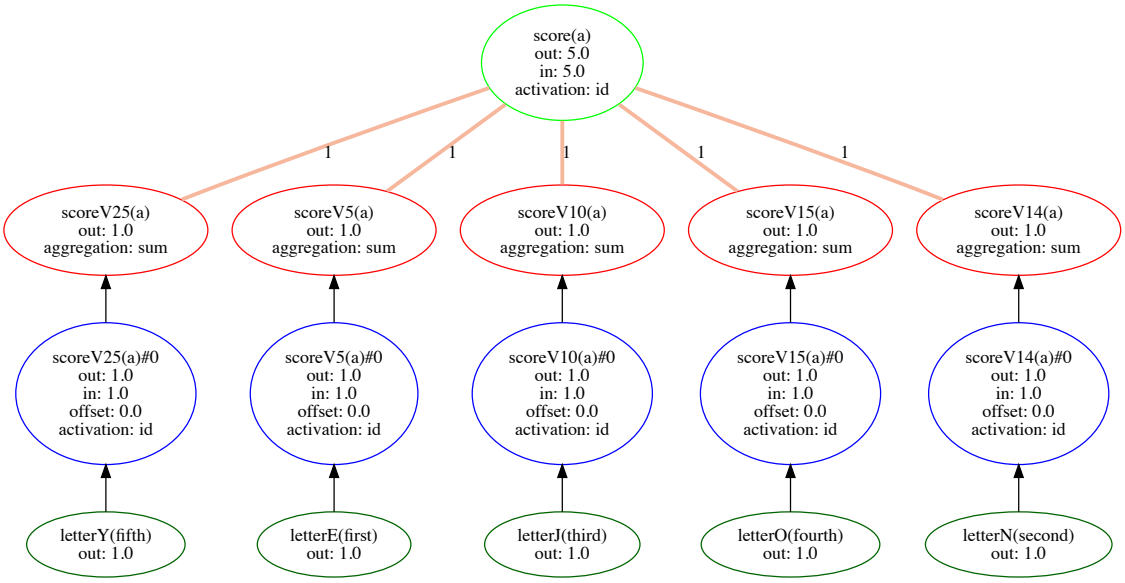


Figure 6.1: A ground neural network created from the word “enjoy”

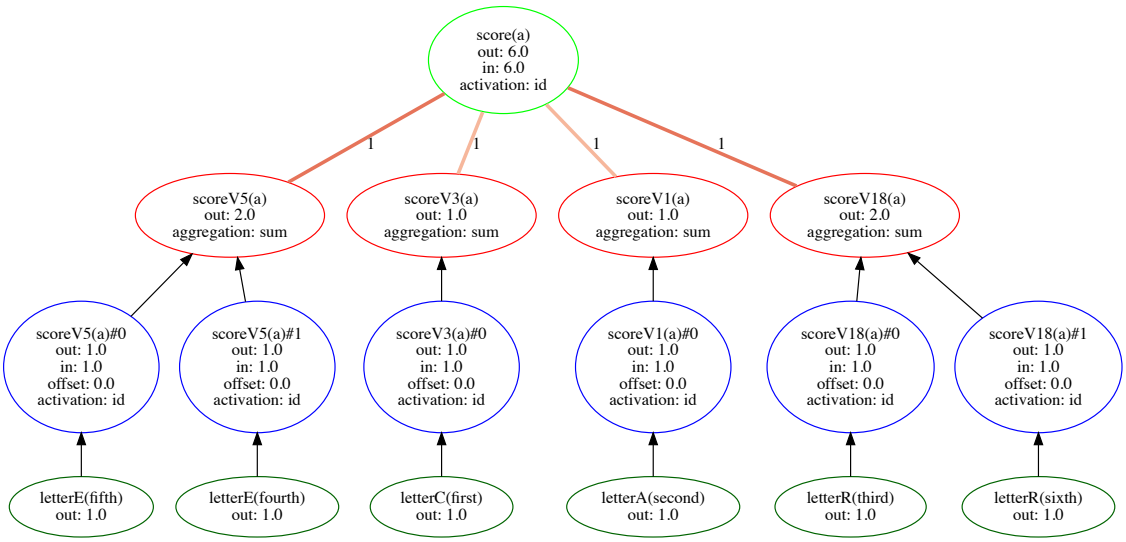


Figure 6.2: A ground neural network created from the word “career”

the truth value of every atom. In the back-propagation phase, the weight handling is slightly different as not all the weights that are used in the network are updated. This includes all edges that form inputs of the rule instance neurons (in Figure 6.2 the blue neuron), because change of these weights would change the interpretation of the rules. These neurons act as (Lukasiewicz) fuzzy conjunctions and therefore their activation should be a simple (non-weighted) sum. Just as in regular NNs, the weight update in a certain network corresponding to one example is automatically reflected throughout all examples via the shared template.

This procedure is repeated for `learningSteps` times and the learned weights will then fully determine the template which is the output model of learning. The learned template for the scrabble task looks as follows[12].

```
1.000 score() :- letterA(X).
3.000 score() :- letterB(X).
3.000 score() :- letterC(X).
2.000 score() :- letterD(X).
1.000 score() :- letterE(X).
4.000 score() :- letterF(X).
...
```

From the template it can be seen that the rule weights are learned to the correct values as expected and thus the model will respond to every word with the right answer. This is not surprising because the cost function of this problem is convex and the gradient descent will always reach global optimum (if the learning rate is low enough not to cause oscillation or divergence from the optimum).

6.2 Tic-Tac-Toe

This example is designed to demonstrate the symbolic reasoning part of our neuro-logic framework. As each template is a logic program written in terms of predicate logic (using Horn clauses), in principle every program representable in Prolog[13] can be used as an example of our symbolic reasoning capabilities.

We chose a simple Tic-Tac-Toe scenario to illustrate the relational reasoning as exhibited by the (learnable) neuro-logic programs. Tic-Tac-Toe is a simple two player game played on a 3x3 grid. The board has 9 spaces and every space is either empty, crossed or circled. In encoding of the game states, each cross on the board will be defined as:

```
cross(RockId),column(RockId,ColumnIndex),row(RockId,RowIndex).
```

and each circle as:

```
circle(RockId),column(RockId,ColumnIndex),row(RockId,RowIndex).
```

[12]The whole template can be found at Appendix C - Scrabble - learned template

[13]Out interpreter does not handle functors but, these are not important here and there in no principal problem in adding this feature.

The reasoning in the network can be seen from two opposing points of view. The first one is reasoning from the leaves to the root and the second one is from the root to the leaves. The leaves represents facts, or the evidence from the example. The root represents the knowledge we would like to induce.

In Tic-Tac-Toe, we can, for instance, represent whether two circles or crosses form a line anywhere in the grid[14], because that is a threat for the opponent. To demonstrate this reasoning in the proposed formalism we would use the following template[15].

```
 0.0  patternTwoLine(cross) :- cross(RockI),
                               cross(RockJ),
                               column(RockI,ColumnI),
                               row(RockI,RowI),
                               column(RockJ,ColumnJ),
                               row(RockJ,RowJ),
                               neighbours(RowI,ColumnI,RowJ,ColumnJ).
<1.0> neighbours(Row,XCol,Row,YCol)   :- successor(XCol,YCol).
<1.0> neighbours(XRow,Col,YRow,Col)   :- successor(XRow,YRow).
<1.0> neighbours(XRow,XCol,YRow,YCol) :- successor(XRow,YRow),
                                         successor(XCol,YCol).
<1.0> neighbours(XRow,XCol,YRow,YCol) :- successor(YRow,XRow),
                                         successor(XCol,YCol).
<1.0> successor(1,2).
<1.0> successor(2,3).
```

To get the truth value of the corresponding `patternTwoLine` (abbr. `pTL`), the reasoning part of the neuro-logic engine would need to create a proof path from the example, where the truth value is known, through all the rules that describe the related literals, all the way to the root which, in this case, is represented by the `pTL` literal. We know from the template that the `pTL` can be derived from the first rule. All of the rules in the template are in the form of implication. To get the truth value of a rule's head, we need to prove and obtain the truth value of all the atoms in the body. Following the backward chaining strategy from Prolog [3], we will try to satisfy the first predicate. Because we do not have any rules that can be used to prove this predicate, this predicate needs to be satisfied right in the example, otherwise we know that patternTwoLine can't be proven and we finish.

In case of an empty grid, the neural network built will also be empty as we do not have any evidence to reason from. Let us suppose that this is not the case and after unification of the first 6 predicates from the body of the first rule we try to prove that the crosses are neigbours. Until now, we have found all the support right in the example data, but here we can use the background knowledge to reason about the position of the crosses. We have the 4 rules in the template that describe whether positions at some

[14]This is a distinguishing feature from encoding with any standard ground model, including neural networks.

[15]for the whole template see Appendix D - Tic-Tac-Toe - template

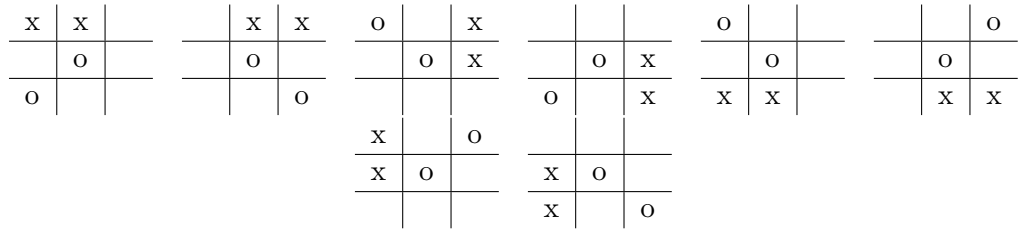


Figure 6.3: 8 different ground conformations of the "same" situation in Tic-Tac-Toe that are jointly captured by the corresponding lifted template.

indexes are neighbours. By applying the same backward chaining strategy we used to prove the `pTL`, we search for a proof to neighbours.

If the predicate body has some free variables (as in the case of `pTL`), we can find not only one proof, but several. To decide how the particular proofs (and their truth values) are combined to the single output corresponding to the head, every predicate has its aggregation function that aggregates the multiple proofs.

Generally, if we trace the described backward chaining strategy to prove any predicate in a (non-recursive) template and we merge all the resulting proof paths, we obtain an acyclic graph that we may further transform to a feed-forward neural network as described in Section 3.3.

This reasoning allows to extract the knowledge about the example from it relevant part without any restriction to the locality of particular feature. This is vital for reasoning on top of Tic-Tac-Toe board because the board itself is symmetrical in 8 directions. Different looking situations as the ones illustrated in the Figure 6.3 are identical. Because we defined our template accordingly the reasoning for the situations presented in Figure 6.3 will be identical.

6.3 Poker

This example demonstrates a complex common sense reasoning pattern combining intuition encoded into a background knowledge that is used to reason on top of an object similarity pattern, both of which are being simultaneously learned to generalize onto unseen relational patterns.

The task is as follows. We are given a figure made of n cards and we should say whether the figure is straight or not. If the figure is straight, we should further output the value of the highest card in it. The problem is that the cards are heavily illustrated in an unpredictable manner so we cannot decode neither cards' rank nor their suit. We can only distinguish when the cards are the same (have the same suit and the same rank) because they have the same illustration.

The training set consists of all figures that can be made, labeled with the value of the highest card if the figure is straight and zero otherwise. This is of how an example

of card pairs looks like:

```
4.0 score()            :- threes(),fourd().
```

The names of the predicates in the body of the rule (the observation) are just ids for identification of different cards for the illustration purpose. We chose the rank and suit as the name just to be easily interpreted for the reader[16], this information is not used in learning (otherwise the problem is trivial).

Next part is the design of the template. We do not know the card value nor its suit but we know that we are finding the straights. A straight depends only on the card value so we will cluster the card ids to m clusters where m is the number of different card values (which can be deduced, e.g., from labels in the training set). We know that a straight is built from n consecutive integers so we order them in any way and add a rule for every n consecutive clusters. These rules will determine the output. The neuro-logic template is then as follows[17].

```
 0.0  group2(X)        :- twoh(X).
...
 0.0  group2(X)        :- sixc(X).
...
 0.0  group6(X)        :- sixc(X).

<1.0> straight3()      :- group2(C0),group3(C1).
...
<1.0> straight6()      :- group5(C0),group6(C1).

<3.0> figureStrength() :- straight3().
...
<6.0> figureStrength() :- straight6().
```

6.3.1 Evaluation

Accuracy of the learned solution depends on the ability of our framework to correctly learn how to cluster the cards and combine them to obtain the right answer. Because the figure score doesn't depend on the card suit, the (soft) clustering pattern should therefore learn to ignore this information and merge cards together based only on their (unobserved) score. To see how the clusters emerge in the process of learning we display the membership of each of the cards to each of the groups. Because the membership to every group is a five dimensional vector we use Principle Component analysis (PCA) to reduce the dimensionality to 2 for the memberships to be plotted. The result can be seen in Figure 6.4 - Three phases of forming (soft) clusters during learning of the poker template..

[16] s = spades, d = diamonds, h = hearts, c = clubs

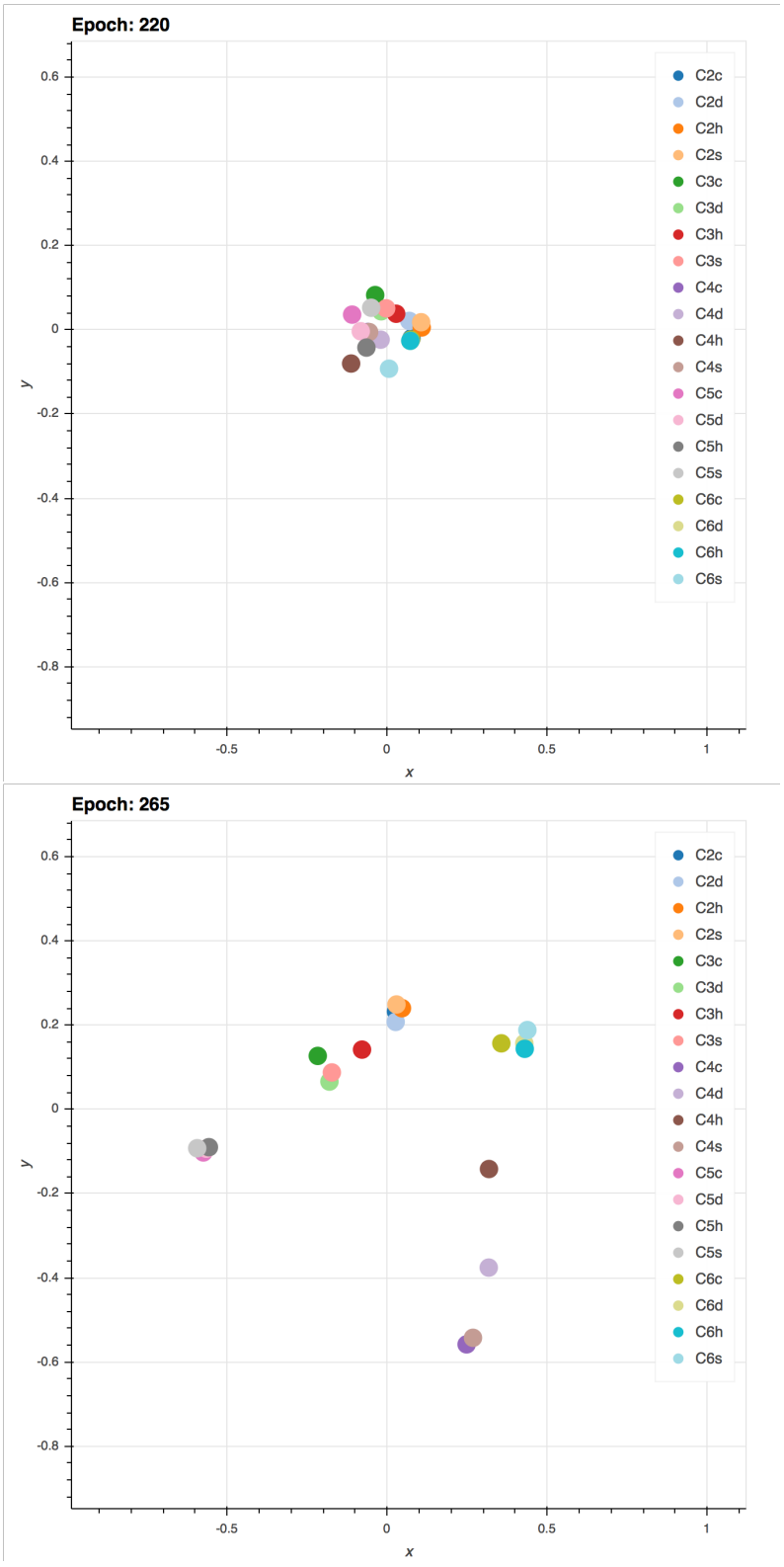
[17] A full template for 5 cards and figure doubles can be found in Appendix E - Poker - template

From the figures, it can be seen that the cards with the same rank but different suit indeed start to cluster together and in the final phase, there are five different clusters, which is the desired behaviour.

6.4 Comparison with Related Work

As well as our neuro-logical framework, both ProbLog and MLN are based on first order logic. MLN does not impose any restriction for the formulas so it can reason and encode rules with wider variety than our framework. To reason about uncertain knowledge, our framework chooses relaxed fuzzy logic as opposed to both ProBlog and MLN that use a well defined probability measures. The difference in terms of speed of reasoning as opposed to our framework is significant as both MLN and ProBlog methods for reasoning are NP-hard and they have to use the approximation methods.

Even though the probability measures in the related frameworks are well defined, each formula is either true or false as in boolean logic, and the methods only define the probability on top of this two valued logic. Therefore, they do not allow to directly capture regression tasks as they are not aimed to express real numbers, since every predicate is either true or false. On the other hand, our formalism, that is built on a relaxed version of fuzzy logic, is able to handle numerical data and allow modelling both classification and regression tasks as demonstrated in the experiments.


Epoch: 220
C2c
C2d
C2h
C2s
C3c
C3d
C3h
C3s
C4c
C4d
C4h
C4s
C5c
C5d
C5h
C5s
C6c
C6d
C6h
C6s
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
y
-0.5
0
0.5
1
x
Epoch: 265
C2c
C2d
C2h
C2s
C3c
C3d
C3h
C3s
C4c
C4d
C4h
C4s
C5c
C5d
C5h
C5s
C6c
C6d
C6h
C6s
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
y
-0.5
0
0.5
1
x

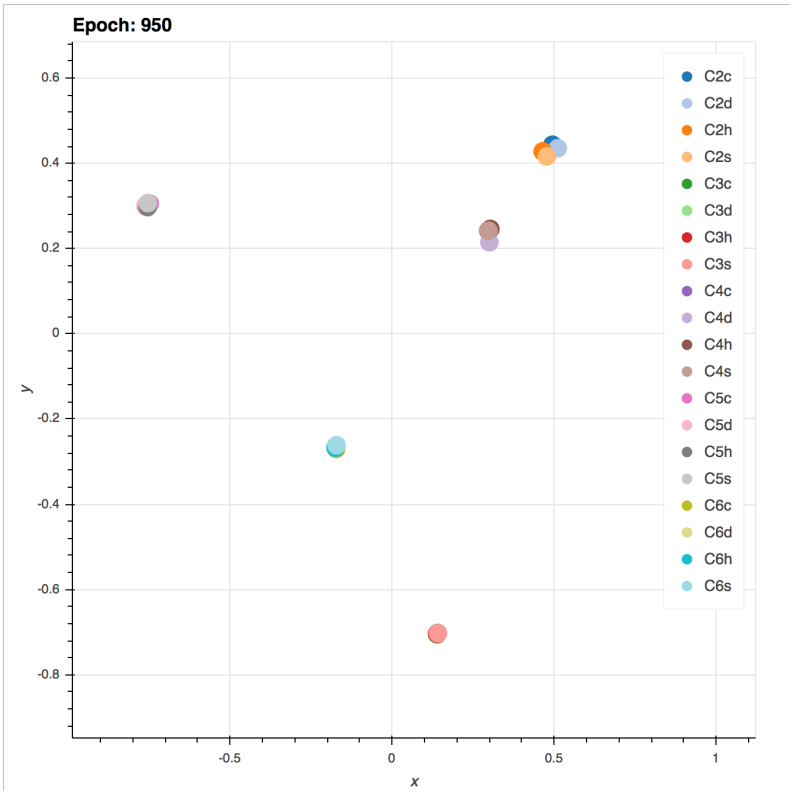


Figure 6.4: Three phases of forming (soft) clusters during learning of the poker template.

Chapter 7

Conclusion

This thesis illustrates a variety of diverse learning and reasoning patterns that can be captured within a unified framework of neuro-logic programming. For practical demonstration of the proposed concepts, we implemented an enhanced engine that builds on top of existing SRL method called Lifted Relational Neural Networks.

As a core part of the thesis, we abstracted key patterns from both statistical and symbolic approaches to AI and showed how they can be incorporated in the proposed formalism. Using neuro-logic programming allowed us to target common sense reasoning patterns, which combine both statistical and symbolic methods, and show their benefits. In the end, we demonstrated the use of these patterns with experiments in simple and interpretable game environments.

With demonstration of capabilities of the enhanced neuro-logic programming framework to capture diverse artificial intelligence tasks based on different reasoning patterns, we conclude that the neuro-logic programming has a potential to successfully model and reason about the world.

Bibliography

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

[2] Peter Flach. "Logical approaches to Machine Learning — an overview". In: *Think* (1992).

[3] Peter A. Flach. *Simply logical: intelligent reasoning by example*. Wiley, 1998.

[4] Lise Getoor. *Statistical Relational Learning: A Tutorial*. URL: https://www.cs.umd.edu/class/spring2008/cmsc828g/Slides/SRL-Tutorial-05-08.pdf (visited on 16/04/2017).

[5] Lise Getoor and Ben Taskar. *Introduction to statistical relational learning*. The MIT Press, 2007.

[6] A. Kimmig L. De Raedt and H. Toivonen. ProbLog. "A probabilistic Prolog and its application in link discovery". In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*. 2007.

[7] Matthew Richardson and Pedro Domingos. "Markov logic networks". In: *Machine learning* 62.1 (2006), pp. 107–136.

[8] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearson Education, 2003. ISBN: 0137903952.

[9] Friedhelm Schwenker, Hans A. Kestler and Günther Palm. "Three learning phases for radial-basis-function networks". In: *Neural Networks* 14.4–5 (2001), pp. 439–458. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(01)00027-2. URL: http://www.sciencedirect.com/science/article/pii/S0893608001000272.

[10] Gustav Sourek et al. *Lifted Relational Neural Networks*. 2015. eprint: arXiv:1508.05128.

Appendix A

Parameters for the learning

Parameter	Type	Default Value	Description
learning steps	positive integer	1000	Number of learning steps for the model
learning rate	positive real	0.05	Learning rate for back-propagation
restart count	positive integer	10	Number of restarts
stochastic gradient descent	boolean	true	Whether to use SGD (update weight after each example)

Appendix B

Scrabble - template

```
0.0 score() :- letterA(X).
0.0 score() :- letterB(X).
0.0 score() :- letterC(X).
0.0 score() :- letterD(X).
0.0 score() :- letterE(X).
0.0 score() :- letterF(X).
0.0 score() :- letterG(X).
0.0 score() :- letterH(X).
0.0 score() :- letterI(X).
0.0 score() :- letterJ(X).
0.0 score() :- letterK(X).
0.0 score() :- letterL(X).
0.0 score() :- letterM(X).
0.0 score() :- letterN(X).
0.0 score() :- letterO(X).
0.0 score() :- letterP(X).
0.0 score() :- letterQ(X).
0.0 score() :- letterR(X).
0.0 score() :- letterS(X).
0.0 score() :- letterT(X).
0.0 score() :- letterU(X).
0.0 score() :- letterV(X).
0.0 score() :- letterW(X).
0.0 score() :- letterX(X).
0.0 score() :- letterY(X).
0.0 score() :- letterZ(X).
```

Appendix C

Scrabble - learned template

```
10.000000000000000 score() :- letterZ(X).
 3.999999999999999 score() :- letterY(X).
 8.000000000000002 score() :- letterX(X).
 4.000000000000000 score() :- letterW(X).
 3.999999999999998 score() :- letterV(X).
 1.000000000000003 score() :- letterU(X).
 1.000000000000000 score() :- letterT(X).
 0.999999999999999 score() :- letterS(X).
 0.999999999999999 score() :- letterR(X).
 9.999999999999991 score() :- letterQ(X).
 3.000000000000001 score() :- letterP(X).
 0.999999999999999 score() :- letterO(X).
 1.000000000000000 score() :- letterN(X).
 2.999999999999999 score() :- letterM(X).
 1.000000000000000 score() :- letterL(X).
 4.999999999999999 score() :- letterK(X).
 7.999999999999999 score() :- letterJ(X).
 1.000000000000000 score() :- letterI(X).
 3.999999999999999 score() :- letterH(X).
 2.000000000000000 score() :- letterG(X).
 4.000000000000001 score() :- letterF(X).
 1.000000000000000 score() :- letterE(X).
 2.000000000000000 score() :- letterD(X).
 3.000000000000002 score() :- letterC(X).
 2.999999999999999 score() :- letterB(X).
 1.000000000000000 score() :- letterA(X).
```

Appendix D

Tic-Tac-Toe - template

```
0.0 patternTwoLine(cross) :- cross(RockI),
                             cross(RockJ),
                             column(RockI,ColumnI),
                             row(RockI,RowI),
                             column(RockJ,ColumnJ),
                             row(RockJ,RowJ),
                             neighbours(RowI,ColumnI,RowJ,ColumnJ).
0.0 patternTwoLine(circle) :- circle(RockI),
                             circle(RockJ),
                             column(RockI,ColumnI),
                             row(RockI,RowI),
                             column(RockJ,ColumnJ),
                             row(RockJ,RowJ),
                             neighbours(RowI,ColumnI,RowJ,ColumnJ).
0.0 patternTail(cross) :- cross(RockID).
0.0 patternTail(circle) :- circle(RockID).
<1.0> neighbours(Row,XCol,Row,YCol) :- successor(XCol,YCol).
<1.0> neighbours(XRow,Col,YRow,Col) :- successor(XRow,YRow).
<1.0> neighbours(XRow,XCol,YRow,YCol) :- successor(XRow,YRow),
                                         successor(XCol,YCol).
<1.0> neighbours(XRow,XCol,YRow,YCol) :- successor(YRow,XRow),
                                         successor(XCol,YCol).
0.0 score() :- patternTail(cross).
0.0 score() :- patternTail(circle).
0.0 score() :- patternTwoLine(cross).
0.0 score() :- patternTwoLine(circle).
```

Appendix E

Poker - template

```
0.0 group2(X) :- twoh(X).
0.0 group2(X) :- twod(X).
0.0 group2(X) :- twos(X).
0.0 group2(X) :- twoc(X).
0.0 group2(X) :- threeh(X).
0.0 group2(X) :- threed(X).
0.0 group2(X) :- threes(X).
0.0 group2(X) :- threec(X).
0.0 group2(X) :- fourh(X).
0.0 group2(X) :- fourd(X).
0.0 group2(X) :- fours(X).
0.0 group2(X) :- fourc(X).
0.0 group2(X) :- fiveh(X).
0.0 group2(X) :- fived(X).
0.0 group2(X) :- fives(X).
0.0 group2(X) :- fivec(X).
0.0 group2(X) :- sixh(X).
0.0 group2(X) :- sixd(X).
0.0 group2(X) :- sixs(X).
0.0 group2(X) :- sixc(X).
0.0 group3(X) :- twoh(X).
0.0 group3(X) :- twod(X).
0.0 group3(X) :- twos(X).
0.0 group3(X) :- twoc(X).
0.0 group3(X) :- threeh(X).
0.0 group3(X) :- threed(X).
0.0 group3(X) :- threes(X).
0.0 group3(X) :- threec(X).
0.0 group3(X) :- fourh(X).
0.0 group3(X) :- fourd(X).
0.0 group3(X) :- fours(X).
0.0 group3(X) :- fourc(X).
0.0 group3(X) :- fiveh(X).
0.0 group3(X) :- fived(X).
0.0 group3(X) :- fives(X).
0.0 group3(X) :- fivec(X).
0.0 group3(X) :- sixh(X).
0.0 group3(X) :- sixd(X).
0.0 group3(X) :- sixs(X).
0.0 group3(X) :- sixc(X).
0.0 group4(X) :- twoh(X).
0.0 group4(X) :- twod(X).
0.0 group4(X) :- twos(X).
0.0 group4(X) :- twoc(X).
0.0 group4(X) :- threeh(X).
0.0 group4(X) :- threed(X).
0.0 group4(X) :- threes(X).
0.0 group4(X) :- threec(X).
0.0 group4(X) :- fourh(X).
0.0 group4(X) :- fourd(X).
0.0 group4(X) :- fours(X).
0.0 group4(X) :- fourc(X).
0.0 group4(X) :- fiveh(X).
0.0 group4(X) :- fived(X).
0.0 group4(X) :- fives(X).
0.0 group4(X) :- fivec(X).
0.0 group4(X) :- sixh(X).
0.0 group4(X) :- sixd(X).
```

```
0.0 group4(X) :- sixs(X).
0.0 group4(X) :- sixc(X).
0.0 group5(X) :- twoh(X).
0.0 group5(X) :- twod(X).
0.0 group5(X) :- twos(X).
0.0 group5(X) :- twoc(X).
0.0 group5(X) :- threeh(X).
0.0 group5(X) :- threed(X).
0.0 group5(X) :- threes(X).
0.0 group5(X) :- threec(X).
0.0 group5(X) :- fourh(X).
0.0 group5(X) :- fourd(X).
0.0 group5(X) :- fours(X).
0.0 group5(X) :- fourc(X).
0.0 group5(X) :- fiveh(X).
0.0 group5(X) :- fived(X).
0.0 group5(X) :- fives(X).
0.0 group5(X) :- fivec(X).
0.0 group5(X) :- sixh(X).
0.0 group5(X) :- sixd(X).
0.0 group5(X) :- sixs(X).
0.0 group5(X) :- sixc(X).
0.0 group6(X) :- twoh(X).
0.0 group6(X) :- twod(X).
0.0 group6(X) :- twos(X).
0.0 group6(X) :- twoc(X).
0.0 group6(X) :- threeh(X).
0.0 group6(X) :- threed(X).
0.0 group6(X) :- threes(X).
0.0 group6(X) :- threec(X).
0.0 group6(X) :- fourh(X).
0.0 group6(X) :- fourd(X).
0.0 group6(X) :- fours(X).
0.0 group6(X) :- fourc(X).
0.0 group6(X) :- fiveh(X).
0.0 group6(X) :- fived(X).
0.0 group6(X) :- fives(X).
0.0 group6(X) :- fivec(X).
0.0 group6(X) :- sixh(X).
0.0 group6(X) :- sixd(X).
0.0 group6(X) :- sixs(X).
0.0 group6(X) :- sixc(X).
group2/1 [lukasiewicz]
group3/1 [lukasiewicz]
group4/1 [lukasiewicz]
group5/1 [lukasiewicz]
group6/1 [lukasiewicz]
<1.0> straight3() :- group2(X0),
                     group3(X1).
<1.0> straight4() :- group3(X0),
                     group4(X1).
<1.0> straight5() :- group4(X0),
                     group5(X1).
<1.0> straight6() :- group5(X0),
                     group6(X1).
<3.0> score() :- straight3().
<4.0> score() :- straight4().
<5.0> score() :- straight5().
<6.0> score() :- straight6().
```

Appendix F

Contents of enclosed CD

```
src ..... implementation sources
examples ..... evaluated experiments
thesis ..... the thesis text directory
    main.pdf ..... the thesis in PDF format
    main.tex ..... the thesis in LaTeX format
```

Appendix G

Glossary

PGM Probabilistic Graphical Model - Graph that expresses conditional probabilities between random variables.
SRL Statistical Relational Learning
CRF Conditional Random Fields
PRM Probabilistic Relational Models
RMN Relational Markov Network
MRF Markov Random Field
MLN Markov Logic Network
PER Probabilistic Entity-Relationship Models
RDN Relational Dependency Network
CPD Conditional Probability Distribution
Horn clause clause with at most one positive literal
Definitive clause clause with exactly one positive literal
DNN Deep Neural Network
ANN, NN Artificial Neural Network
FF NN Feed Forward Neural Network
CNN Convolutional Neural Network
RNN Recurrent Neural Network
SLD resolution Selective Linear Definite clause resolution
EM Expectation maximization
PSL Probabilistic Soft Logic
Factor graph bipartite graph representing the factorization of a function