Universiteit Utrecht

Bachelor's Thesis Artificial Intelligence
(7.5 ECTS)

Inserting Symbolic Knowledge to Improve Learning in Neural Networks

Two Test Cases

Cornelis Bouter

First supervisor:
dr. ir. Jan Broersen

Second supervisor:
prof. dr. Daniel Cohnitz

August 25, 2016

Contents

Chapter 1

Introduction

The symbolic and subsymbolic approaches to Artificial Intelligence have often been seen as complementary. On the one hand, symbolic systems present us formal reasoning and human readability. On the other hand, subsymbolic systems, especially Artificial Neural Networks (ANN), have the capability to learn from data and are very resistant to noise. Therefore, we will try to combine the strengths of both to improve their performance on binary classification problems, compared to the performance of standard subsymbolic systems. Binary classification is the problem to, given an example containing n input values, determine whether this example has a certain property or does not. We expect that the performance of ANNs will indeed increase by adding background knowledge.
The symbolic background knowledge about the problem will be in form of a logic program. So, it is a set of clauses "if $A_1, ..., A_n$, then B", where $A_1, ..., A_n$ and B represent some property. The human readability is necessary for human experts to write their background knowledge as a logic program.
The subsymbolic model of computation we will use is an Artifical Neural Network. It is a model designed for n-ary classification, in our case binary, so it gives us the learning capability, typical of subsymbolic systems. The precise learning algorithm will be presented in section 2.1. For now it is sufficient to say that after obtaining a set of input values and incorrectly classfying it, the network can adjust its structure to be more likely to classify the next example correctly.
Usually, an ANN is initiated randomly. We take another approach. We will initiate the ANN by inserting background knowledge into the network. To perform the translation of logic program to neural network, we use the Connectionist Inductive Learning and Logic Programming (CILP) translation algorithm [5]. This way, we aim to combine a subsymbolic system with

symbolic background knowledge.
The theoretical part of this thesis will use the Immediate Consequence Operator (ICO) as a tool to prove that a translation of logic program to ANN is theoretically possible. Any logic program has a unique ICO associated with it, so if an ANN is functionally equivalent to the ICO, we believe the ANN has learned the logic program. Firstly, we will show that the operator has the properties needed for an ANN to be able to approximate it. Then, we will give a sketch of the proof that the CILP translation algorithm does indeed produce an ANN that approximates the Immediate Consequence Operator of the inserted logic program.
To conclude this thesis we will perform some experiments to test our hypothesis that inserting symbolic background knowledge into an ANN will improve its performance on a binary classification problem. We will present experimental results for both a network constructed with background knowledge and a randomly initiated neural network. For this task we selected two classification problems.

1.1 Datasets

Firstly, we got the problem commonly known as the Muddy Children puzzle. Imagine a situation where a particular number of children has played outside and may have gotten dirty. For some reason a child cannot know if he or she is muddy, but the child can see whether another child is muddy. Let us just assume their hair is muddy. We observe the situation with three children. When they get home, a parent or guardian makes the public announcement that at least one of them is muddy. He also tells them to step forward if he or she knows whether he or she is muddy. If no child steps forward, the parent announces that one more child is muddy. How does a child reason to learn whether he or she is muddy?
Let m of the n children be muddy. Then, immediatly after the m^{th} announcement the muddy children will know that they are muddy [15]. Take for example the situation with only child 1 being muddy. It can see that the other children are clean. So, it only has to hear the initial announcement to deduce that he must be muddy. The other children learn that the first public announcement was enough for child 1 to learn whether it is muddy. With the information available to child 1 he cannot have learned to be clean, so the other children can infer that child 1 has learned that he or she is indeed muddy. So children 2 and 3 learn they are clean. By seeing the muddiness of each child as a binary variable, we can construct a binary list for each state of the puzzle.

The other dataset we use is of passengers on board the Titanic. The data is readily available online through the data science website kaggle.com. It is well suited to our problem, as it only has one binary output variable: survival. The dataset presents for each person his or her name, age, ticket price, place of embarkment and more. We need to reason about the relation between these properties and survival. Maybe women and children survive, but not if they were third class passengers. However, contrary to the Muddy Children problem, the clauses we may derive for the Titanic problem will not be deterministic. A rule only tells us that certain properties increase the likelihood of survival. This is a major difference between the Titanic and the Muddy Children problem, but this is intentionally. We want to investigate whether we can increase the performance of ANNs with both kinds of background knowledge.
Our research is primarily based on [5] and [7]. It is an expansion of Knowledge Based Artificial Neural Networks [14]. The authors have already obtained significant results on the Muddy Children problem. First of all, we want to reproduce those results. Secondly, we want to apply the method to the Titanic problem where we reason about likelihoods, to see if the method increases the performance of ANNs in both situations.

1.2 Structure

We structure our thesis as follows. Firstly, to conduct our research we need to define some background concepts: feedforward neural networks trained with backpropagation, logic programs and the immediate consequence operator. In the next part we describe the combination of those three concepts: the Connectionist Inductive Learning and Logic Programming translation algorithm. We also present an outline of its proof. In chapter 4 we state our testing conditions and present the results. To conclude, we explain the relevance for general AI research and give suggestions for further research.

Chapter 2

Background

2.1 Artificial Neural Networks

The Artificial Neural Network (ANN) is the model of computation we use for our learning problem. It is inspired by the structure of a brain. Just as a brain consists of interconnected neurons, an ANN consists of several layers of interconnected so-called perceptrons. The earliest mention of the brain as a computational model occured in the 1940s with the McCulloch-Pitts neuron. The perceptron model, wich will be described in the next paragraph, was first proposed in the 1960s. It took another twenty years until a network of perceptrons was proposed [8]. With the advent of Deep Learning, ANNs have recently become a state-of-the-art AI-technique again. For my purposes, however, it is enough to describe the *feedforward backpropagation ANNs* of the early 1990s. With feedforward we mean the absence of loops in the network. Backpropagation refers to the learning algorithm.

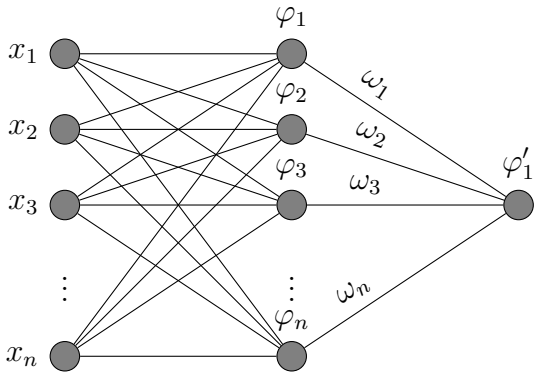

Figure 2.1: The structure of a neural network.

2.1.1 Perceptron

As the figure shows, an ANN consists of three layers: an input layer, a hidden layer and an output layer. Both the hidden- and the output layer consist of perceptrons. Each input unit simply outputs a value (for example the greyness of a pixel or the truthvalue of a propositional variable) to every perceptron in the hidden layer. The behavior of a perceptron is a different.

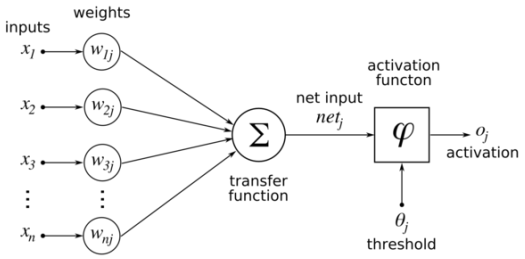

Figure 2.2: A perceptron [12].

An individual perceptron takes values x_0,...,x_n, where n is the number of inputs. The perceptron computes its net input value by taking the inproduct of the input vector and the weight vector: $net_j = \vec{w}_j \bullet \vec{i}_j$. So, we take the sum over all inputs, where each input value is multiplied by its associated weight. The netto input minus the threshold is the input of the perceptron's output function: $o_j = \varphi(net_j - \theta_j)$. Still, we need to decide on the exact output function to use. The function originally proposed is the sign-function.

$$sgn(x) = \begin{cases} 1 \text{ if } x > 0 \\ -1 \text{ otherwise} \end{cases}$$

The sign-operation, however, can only output either 1 or -1, so the perceptron can only learn a linear classifier [11]. Even several linked linear classifiers can still only produce another linear classifier [2]. With ANNs we aim to learn a more flexible classifier. Enter the sigmoid unit. The only difference between the perceptron and the sigmoid unit is the latter's output function. To be precise, the output function of a sigmoid unit is:

$$o(\vec{w}, \vec{i}) = \sigma(\vec{w} \cdot \vec{i})$$
where
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

As the image shows the sigmoid function gives a smooth increase in the range (0,1).

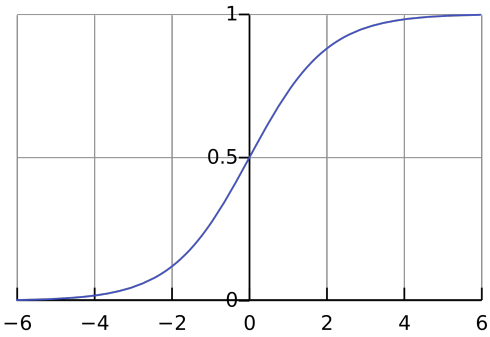

Figure 2.3: The curve of a sigmoid function.

2.1.2 Backpropagation with momentum

Having constructed a network, we can, given an input vector, compute an output value. Both the Muddy Children and the Titanic problem are supervised, so we know the ideal output. If the network output differs from the ideal output, we need to change the weights of the network. The backpropagation algorithm is very efficient for this task and not difficult. It computes the error at the output level. The error is then propagated backwards through the network to adjust the weights of all connections. The complete algorithm is presented in the following table [11].

Data: trainingExamples is a list of tuples $< \vec{x}, \vec{t} >$ where $\vec{x}$ is the input and $\vec{t}$ is the desired output; n_i is the number of input values; n_h is the number of perceptrons in the hidden layer; n_o is the number of output values; η is the learn factor, ξ is the momentum factor

Result: A trained ANN

1 Create a feedforward network with n_i input units, n_h hidden units and n_o output units;
2 Initialise all weights with small random values (e.g. between 0.05 and -0.05);
3 **while** *termination condition not met* **do**
4 **for** $< \vec{x}, \vec{t} >$ *in trainingExamples* **do**
5 *Forward:* ;
6 Compute output o_p of each perceptron p for this particular training example; ;
7 *Backward:* ;
8 *Compute error for each output node* k ;
9 $d_o \Leftarrow o_k(1 - o_k)(t_k - o_k)$;
10 *Compute error for each hidden node* h ;
11 $d_h \Leftarrow o_h(1 - o_h) \sum_{o \in outputnodes} w_{oh} d_o$;
12 *Update weights*;
13 **for** *weight* w_{ji} *in network* **do**
14 $w_{ji} \leftarrow w_{ji} + \eta d_j x_{ji} + \xi \Delta w_{ji}(n-1)$;
15 where x_{ji} is the output from unit j to unit i;
16 **end**
17 **end**
18 **end**

The only part of the algorithm that needs some clarification is the third line. The termination condition of an ANN is not trivial at all. Terminating too soon may cause suboptimal results because of too little training. Terminating too late may also cause inferior results caused by overfitting [11]. The termination condition we use will be described per case in the result section. Also, the backpropagation algorithm does not necessarily end in the global minimum, as the search space may contain several local minima. Therefore, we have implemented momentum in line 14: $\eta \delta_j x_{ji} + \xi \Delta w_{ji}(n-1)$, with $0 \leq \xi \leq 1$. The first term of the sum is just the backpropagation algorithm. The second part is the momentum. It adds the weight update of the previous

iteration to the current. Intuitively, it is just like a ball descending along a slope. Its movement does not suddenly stop in a local minimum, but rather it ascends a little along the next slope.
In the rest of the thesis we will use neural networks with backpropagation and momentum. The number of input and output neurons will decided by the number of propositional variables in logic program. It is the number of hidden neurons, the thresholds and the weights that we will set to approximate the immediate consequence operator.

2.2 Modal Logic

Formal logic was originally developed to reason about knowledge and arguments. While natural language contains lots of ambiguities, logic aims to present a formal approach to define the truth and falseness of statements. In this case, we want to formally describe the knowledge of an agent. How do we say that a child knows it is muddy, when it sees the other two children being clean?
The most simple form of logic is propositional logic, which deals only with truth and falsehood. Propositional logic, however, is not a sufficient tool to reason about the Muddy Children problem. We need to discriminate between different states of knowledge. For example, each agent only knows the muddiness of the other agents. Therefore, we use another logic: modal logic. This logic introduces different states of knowledge. Each state is connected to every other state thats indistinguishable from it for a particular agent. For example, given that all children are muddy, and that the parent has announced that at least one child is muddy, each child cannot distinguish the actual situation from the situation where he himself would be clean.
We provide an inductive definition for the minimal set of syntactically legal modal logic formulas:

Definition 1. *Syntax of modal logic (FOR)*

1. $p_i \in FOR$ *for all* $i \in \mathbb{N}$
2. $\bot \in FOR$
3. *If* $A \in FOR$ *then* $\neg A, \Box A, \diamond A \in FOR$
4. *If* $A, B \in FOR$ *then* $(A \wedge B), (A \rightarrow B) \in FOR$[1]

[1]We omitted some commonly used connectives, as those don't increase representational power. The implication does not either, but we kept it because it is relevant for section 1.3.

For an agent to know $\Box\phi$ means, he is certain ϕ is the case. So, in every world visible for the agent, ϕ is the case. For an agent to know $\diamond\phi$ means he thinks it is possible ϕ is the case. This means the current world is connected to at least one world where ϕ is the case. The state of the world is represented in a frame.

Definition 2. *Kripke Frame. A Kripke frame is a tuple $< W, R >$, where W is a set of worlds (ie. nodes) and $R \subseteq (W \times W)$, the accessability relation, is a binary relation on W [10].*

When valuations are added to the nodes of the frame, it is called a Kripke model.

Definition 3. *A Kripke model is an extension of a Kripke frame with a valuation: $< W, R, V >$, where $< W, R >$ is a Kripke frame. V is a function $V : W \to \mathcal{P}(var)$, where var is the set of atoms in the model. A proposition p is true in world $w \in W$ if $p \in V(w)$ and false if $p \notin V(w)$ [10].*

To conclude, we give the semantics of modal logic.

Definition 4. *The truth of a formula ϕ in a given model $M =< W, R, V >$ is defined as follows, with $w \in W$:*

- $M, w \vDash p$ *iff* $p \in V(w)$
- $M, w \vDash \bot$ *iff* $1 = 0$
- $M, w \vDash \neg A$ *iff not* $M, w \vDash A$
- $M, w \vDash (A \wedge B)$ *iff* $M, w \vDash A$ *and* $M, w \vDash B$
- $M, w \vDash (A \to B)$ *iff (not* $M, w \vDash A$*) or* $M, w \vDash B$
- $\Box\phi$ *is true in world w iff for all $v \in W$ with wRv it holds that $M, v \vDash \phi$.*
- $\diamond\phi$ *is true in world w iff there exists a $v \in W$ with wRv such that $M, v \vDash \phi$.*

2.2.1 Modal Logic Programs

When we combine the knowledge of an agent to a set of rules $A_0 \leftarrow A_1, ..., A_n$ with A and $A_1, ..., A_n$ atoms, we obtain a definite logic program. It is definite, because no classical negations ($\neg A$) are allowed. We do, however, allow default negation (A). For $\neg A$ to be true, we have to explicitly know that A

is not the case, whereas for A to be true it is enough to have no knowledge whether A is true. Although the difference is an interesting topic on its own, it is beyond the scope of this thesis.
For each rule, A_0 is called the head (or consequent) and the possibly empty set of rules $A_1, ..., A_n$ is called the body (or antecedent). If $A_0, ..., A_n$ may be modal atoms (ie. of the form MA with $M \in \Box, \Diamond$) it is called a modal program. When every rule is of the form $w_i : A_0 \rightarrow A_1, ..., A_n$, where w_i designates a world in the frame, it is called an extended logic program [13].
Take for example the following modal program P'. It is a part of the modal logic program for the muddy children problem (three children) for child 1:

1. $\Box p_1 \leftarrow \Box q_3$
2. $\Box q_3$

Propositional variable p_i is interpreted as "child i is muddy". Variable q_i means that at least i children are muddy. So, the first rule says that if an agent knows at least three children are muddy, then it knows itself to be muddy. Note that the second rule is syntactically correct, since the body may be empty.
To give meaning to a logic program we need to interpret it. We can give it a Herbrand interpretation, for example. A Herbrand interpretation is nothing more than a set of atoms: possibly $\emptyset$ or $\{p_1, q_3\}$. These atoms are interpreted as true variables and the other atoms are interpreted as false. An interpretation of P is also a model of P if every formula in P is true. An interpretation is supported if every atom exists as a head in one of the rules of P and if for each of these rules the body is true [13]. So, for example interpretation $\emptyset$ is neither a model, nor stable, because q_3 is true in the model. Model $\{p_1, q_3\}$ *is* a supported model.

2.2.2 Immediate Consequence Operator

Given an interpretation we want to find the added knowledge (or consequences) of that interpretation. In program P', for example, if we know $\{q_3\}$ to be true, we can add p_1 to our knowledge. Similarly, if we know $\emptyset$ to be true, we can q_3 to our interpretation. It is the immediate consequence of what we already regard as true. So, the immediate consequence operator T_p is a mapping of subsets of atoms to subsets of atoms.

$$T_p : \mathcal{P}\left(atoms(P)\right) \rightarrow \mathcal{P}\left(atoms(P)\right)$$
where $\mathcal{P}$ is the power set notation,
and P is a logic program.

Definition 5. *Let P be a definite logic program and I a Herbrand interpretation (ie. a set of ground atoms) of P.*

$$T_p(I) = \{\alpha \in atoms(P) | \alpha \leftarrow B_1, ..., B_m (m \geq 0) \text{ is an instance of a clause in } P \text{ and } \{B1, ..., B_m\} \subseteq I\}$$

So, what is significant about this function? When the output of T_p equals its input (ie. at a fixed point, $f(x) = x$), then I is a model of P. If an $x \in I$ would not be true in P, then x would not be an element of $T_p(I)$. If some truths of the model would not be captured in I, then $T_p(I)$ would not be equal to I. So, the operator can express the same statements the logic program can.
Also, T_p is a continuous function [13]. This roughly means that it has no gaps or jumps. More importantly, though, this means an ANN can in theory approximate T_p arbitrarily well [3]. How we do this in practice will be the subject of the following section.

Chapter 3

CILP translation algorithm

In this section we present the Connectionist Inductive Learning and Logic Programming (CILP) translation algorithm. It takes an (extended) logic program and outputs an artificial neural network that computes its immediate consequence operator. The algorithm is described based on [5] and [4]. The only difference is that our implementation can translate *extended* logic programs, while our sources split this explanation in two parts. We will describe the algorithm both intuitively and formally. We will not provide a complete proof of its correctness, but we present only the idea and refer to the formal proof.

3.1 Intuitive definition

Firsly, we need to create the appropriate structure of the neural network. That means we need to set the amount of nodes in the input, hidden and output layer. An input node is created for each unique literal in the body of a rule, where we regard $\Box p$ and $\diamond p$ as different literals than p. For each clause a hidden node is constructed and each literal in its body is connected to the hidden node. If the literal in the body is negative (negation as failure) the connection will have a negative weight. An output node is created for each unique literal in the head of a rule. Each hidden node is connected to the output node that represents its head. Each connection just prescribed will receive weight W, or $-W$ for negative literals. The exact value of W will be defined in a moment. All other connections will be initialised with 0 as their weight, so the network will be fully connected.
Given this structure, we need to set the value W and each threshold value such that the network is equivalent to the original logic program. Therefore,

we need to implement two conditions.

1. A hidden neuron may not fire unless all its connected positive neurons fire and none of its connected negative neurons fire.

2. An output neuron may not fire unless at least one of its connected hidden neurons fires.

3.2 Formal definition

To provide a formal definition of the algorithm we need to define several variables. Let:

- q denote the number of clauses in P,
- v denote the number of literals in P,
- A_{min} denote the minium value for a neuron to be active. In the equivalent logic program it means the literal is true. $A_{min} \in (0, 1)$,
- $A_{max} = -A_{min}$ denote the maximum value for a neuron to be inactive. In the equivalent logic program it means the literal is false. $A_{max} \in (0, -1)$,
- $h(x) = \frac{2}{1+\epsilon^{-\beta x}} - 1$, where β is the steepness factor,
- $g(x) = x$, the standard linear activation function,
- W and $-W$ denote the weights of connections associated with positive and negative literals, respectively,
- θ_l denote the threshold of hidden neuron N_l, associated with clause r_l,
- θ_A denote the threshold of the output neuron A,
- k_l denote the number of literals in the body of clause r_l,
- p_l denote the number of positive literals in the body of clause r_l,
- n_l denote the number of negative literals in the body of clause r_l,
- μ_l denote the number of clauses in P with the same atom in the head, including itself,
- $\text{MAX}_{rl}(k_l, \mu_l)$ denote the maximum element of k_l and μ_l for clause r_l, and

- $\text{MAX}_P(k_1, ..., k_q, \mu_1, ..., \mu_l)$ denotes the maximum element of any k_i and μ_i for the whole logic program.

For example, let P be the logic program $\{A \leftarrow B, C; A \leftarrow D, E; D \leftarrow E; E\}$. So, $q = 4$, $v = 5$, $k_1 = 2$, $k_2 = 2$, $k_3 = 1$, $k_4 = 0$, $p_i = k_i$ and $n_i = 0$ for all $1 \leq i \leq 4$, $\mu_1 = 2$, $\mu_2 = 2$, $\mu_3 = 1$, $\mu_4 = 1$, $MAX_{r1} = 2$, $MAX_{r2} = 2$, $MAX_{r3} = 1$, $MAX_{r4} = 1$ and $MAX_P = 2$. So the translation gives us the following network. Note that the network is fully connected, but connections with weight value 0 are not shown.

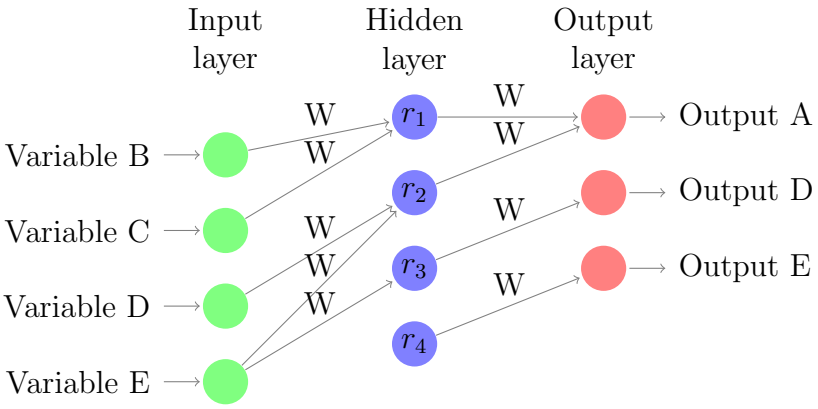

Figure 3.1: The structure of the translated logic program.

These definitions allow us to define the algorithm which takes a logic program P as its input and returns a neural network N.

1. Calculate A_{min} and W such that the following are satisfied:
$$W \geq \frac{2}{\beta} \cdot \frac{\ln(1+A_{min}) - \ln(1-A_{min})}{\text{MAX}_P(\vec{k},\vec{\mu})(A_{min}-1)+A_{min}+1}$$
$$A_{min} > \frac{\text{MAX}_P(\vec{k},\vec{\mu})-1}{\text{MAX}_P(\vec{k},\vec{\mu})+1}$$
2. For each rule r_l of P as $A \leftarrow L_1, ..., L_k$ with $k \geq 0$:
   (a) Create input neurons $L_1, ..., L_k$ and an output neuron A in N, unless such a neuron already exists.
   (b) Add a neuron N_l to the hidden layer of N.
   (c) Connect each neuron L_i $(0 \leq i \leq k)$ to neuron N_l in the hidden layer. Assign weight W to the connection if L_i is a positive literal, otherwise set the connection weight to $-W$.

(d) Connect N_l to output neuron A and set the connection weight to W.

(e) Define the threshold of N_l:
$$\theta_l = \frac{(1+A_{min})(k_l-1)}{2}W.$$

(f) Define the threshold of the output neuron A as:
$$\theta_A = \frac{(1+A_{min})(1-\mu_l)}{2}W.$$

3. Set $h(x)$, the bipolar semilinear function, as the activation function of all nodes in the hidden layer and in the output layer.

4. Set $g(x)$, the linear function, as the activation function of all nodes in the input layer.

5. Set the weight of all other connections to zero.

3.3 Proof of correctness

Theorem 1. *For each propositional general logic program P, there exists a feedforward artificial neural network N with exactly one hidden layer and semilinear neurons such that N computes the fixed-point operator T_P of P.*

We only show the idea, but the complete proof can be found in [5] or [4]. We need to show that, given an input vector i, output neuron A is active (ie. its output is greater than or equal to A_{min}) if and only if there exists a clause $A \rightarrow L_1, ..., L_k$ such that all L_i with $1 \leq i \leq k$ are satisfied by interpretation i. The converse direction states that the output of neuron A is less than $-A_{min}$, if the clause is not satisfied. The proof consists of two directions:

'$\leftarrow$' *To proof:*If the body $L_1, ..., L_k$ of rule h is satisfied by i, then $o(A) \geq A_{min}$.
Proof: Starting with the hidden layer, we want to compute the minimal possible activation value for hidden node N_h, given that its associated rule is satisfied. As the activation function is monotonically increasing, for all higher activation values the output will be greater than A_{min} and for all lower values the output will be less than A_{min}. The minimal activation value is reached when all connected positive literals output A_{min} and all negative literals output $-A_{min}$.[1] So, at that point the input potential of N_h equals $p_h W A_{min} + n_h(-W)(-A_{min}) - \theta_h$. So,

[1]Not when the output is -1. The weight of the connection is $-W$ and $-1 * -W$ is greater than $-A_{min} * -W$.

the equation $h(p_h W A_{min} + n_h(-W)(-A_{min}) - \theta_h) \geq A_{min}$ has to be satisfied.
We compute the same minimal input potential for the output neuron A, tiven that A is true. Then, only one connected hidden neuron should be active. The other connected neurons (numbering $\mu - 1$) have to be inactive. So, the minmial input potential equals $WA_{min} + -1(\mu - 1)W - \theta_h$. So, for the output layer, the equation $h(WA_{min} + -1(\mu - 1)W - \theta_h) \geq A_{min}$ has to be satisfied.

'$\rightarrow$' *To proof:* If the body $L_1, ..., L_k$ of rule h is not satisfied by i, then $o(A) \leq -A_{min}$. *Proof:* The converse direction uses the same idea. This time, we compute the maximal possible input potential when the body is not satisfied. So, either one of the neurons that should be active is inactive, or one of the neurons that should be inactive is active. Either way, the maximum possible input potential equals $(k-1)W - A_{min}W - \theta_h$. The neuron in the hidden layer should output no higher value than $o((k-1)W - A_{min}W - \theta_h)$. The same way, we can find that neuron A should output no higher value than $o(\mu(-A_{min})W - \theta_h)$.
Finally, we need to solve all four inequalities and will find that both directions result in the same two constraints for respectively the hidden neuron and for the output neuron. These constraints, however, differ for every rule, as different clauses may have different values for k and p. We want to obtain a unique value for W and for A_{min}, so with the MAX_P-function we find the highest k or p value. Then, we apply the strictest constraint to the whole network.

3.4 Adapting the network for the learning problem

This result allows us to easily simulate the immediate consequence operator. Let (-1,-1,-1,-1) be the input vector for our example network. This means we believe atoms B, C, D and E to be false. After computing the output, we will find that only the output value of the neuron associated with atom E has an output value higher than A_{min}. Intuitively, this is correct, as our original logic program states E as a fact. In the next iteration we can insert (-1,-1,-1,1) as input vector and will find the output of the neuron associated with D to be higher than A_{min}. And so on until we reach a fixed point.
The proof also gives us reason to believe that we have correctly inserted the logic program into the network and that we have retained the knowledge it represented. The complete dataset, however, may contain several properties

that were not present in the background knowledge. For example, in the Titanic problem we have may knowledge of the relation between sex and survival, but not about the relation between family size and survival. So, we need to expand the network. Afterall, the translated logic program will only be part of the whole neural network. In the same way, background knowledge is only part of the learning process.
To prepare the generated network for a learning process, we need to perform three steps [5].

1. We need to add input nodes for each property represented in the dataset. Also, we may need to add output nodes.

2. We need to add hidden nodes for the network to learn new rules. The background knowledge is only part of the complete solution, after all. Needless to say, we add weights so the network stays fully connected.

3. We need to perturb the weights. We add to each weight a Gaussian value with mean 0 and standard deviation 0.1. The learning ability of a neural network is negatively influenced by symmetry, so we make sure the network is not symmetric.

Chapter 4

Implementation and tests

Having defined the translation algorithm in the previous chapter, we can at last test whether it does indeed improve the performance of a neural network. Therefore, this section will open with a short explanation of our Java implementation. Secondly, we will describe the datasets in detail and define our test parameters. For both learning problems we construct two neural networks: one with background knowledge and one without. Finally, we present our results.

4.1 Implementation

We originally wanted to program a complete framework that could take as its input a logic program and a data set. Then it would output a trained neural network with accuracy scores. This project, however, turned out to be too time consuming. Therefore we chose to simply construct a neural network per test case. For the networks we used the Neuroph framework [1]. It provided us with sufficient classes to represent the connections, neurons and layers the network consists of. It also provided Java-interfaces that we implemented to write our own activation function and evaluation class.

4.2 Datasets

4.2.1 Muddy Children

To present the Muddy Children as a learning problem we need to format it as a set of numerical input vectors with a corresponding output value. The following four rules describe the a priori knowledge of agent 1 in the puzzle

with three agents. The atoms p_i and q_i represent that agent i is muddy and that at least i agents are muddy, respectively.

1. When the agent knows the other two agents are not muddy and at least one person is muddy, then it knows itself to be muddy:
   $\Box p_1 \leftarrow \Box q_1 \wedge \Box\neg p_2 \wedge \Box\neg p_2$

2. When at least two other agents are muddy and another agent is clean, the agent knows he is muddy:
   $\Box p_1 \leftarrow \Box q_2 \wedge \Box\neg p_2$

3. Same as above:
   $\Box p_1 \leftarrow \Box q_1 \wedge \Box\neg p_3$

4. When the agent knows that at least three agents are muddy, then it must be muddy itself:
   $\Box p_1 \leftarrow \Box q_3$

So, we end up with nine different input variables: "$\Box q_i$", "$\Box p_i$" and "$\Box\neg p_i$ for all $0 \leq i \leq 3$. From these inputs we construct the dataset for agent 1. We do not permit contradictions in the dataset. Clearly, both $\Box p_2$ and $\Box\neg p_2$ cannot be true at the same time. Also, since we construct the dataset for agent 1, "$\Box_1\neg p_1$" and "$\Box_1 p_1$ are false. This leaves us five binary variables, so the dataset consists of $2^5 = 32$ entries. The output variable represents $\Box p_1$. We consider two slightly different versions, as we think the data is not unambiguously defined in [5]. We presume that data set contains only four rows that should output the value 1. Namely, exactly the four scenario's enumerated in the previous alinea. We call this the strict approach. Another option would be that an entry should output 1, whenever all atoms of any of the four clauses are true. This, however, may lead to situations that cannot occur in the original Muddy Children puzzle. For example, the child may know that at least two other children are muddy, but it does not know that at least one other child is muddy. Because of the fourth clause, any data set entry where "$Kq3$" is true, would output 1. This we will call the soft approach.
The background knowledge we want to insert is the first rule of above the enumeration: $\Box p_1 \leftarrow \Box q_1\Box\neg p_2 \wedge \Box\neg p_3$. This means the network will be structured as shown in figure 4.1. Note that connections with weight 0 are not shown. When not inserting background knowledge, all connections will be iniated with weight value 0.

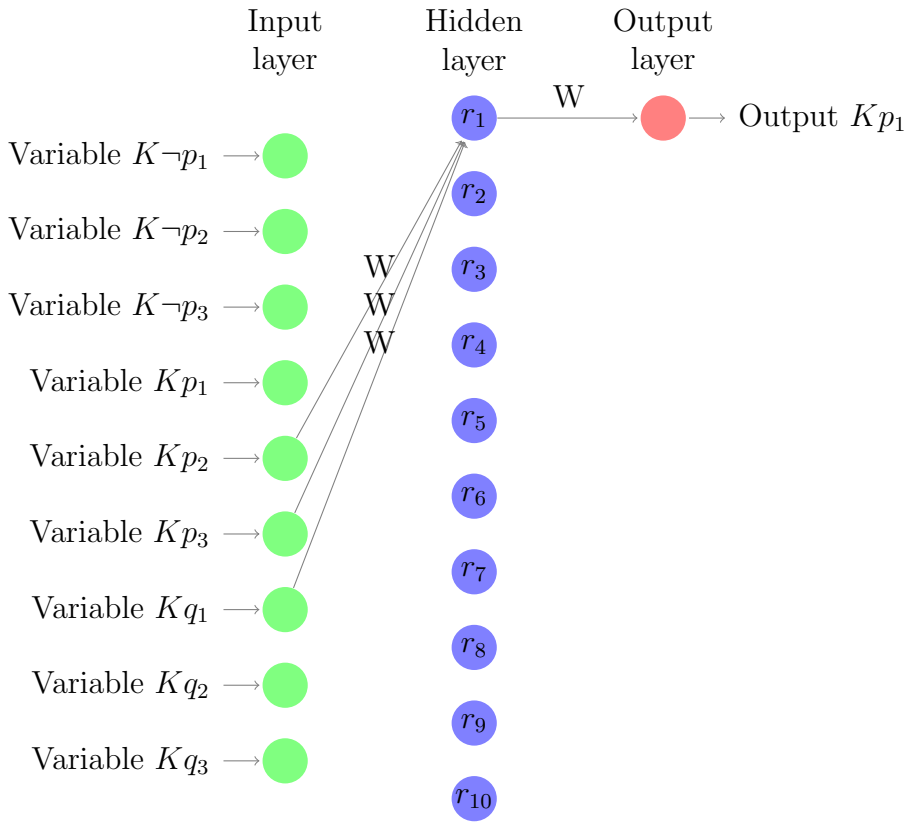

Figure 4.1: The structure of the Muddy Children network.

4.2.2 Titanic

For 892 passengers the dataset gives, among others, his or her name, age, place of embarkment, family and ticket price. The survival for the other passengers is not publicly available on the site, so we use only these 892 passengers for our tests. We discarded the variable representing the place of embarkment, as it does not have an ordering. Also, the ticket price variable seems to give incorrect values for some passengers, so we disregard it. We split the numerical "family size" variable in two binary variables: "Singleton" for passengers travelling alone, and "LargeFamily" for members of a family larger than five persons. Also, several age values are missing, so we insert likely substitutes [12]. We end up with the following attributes for each dataset entry:

1. Pclass: a discrete variable that represents the class the person travelled in. It has possible values -1, 0, and 1 for respectively first, second and

third class.

2. Sex: a binary variable with value 1 for men and value -1 for women.

3. Age: a continuous variable with value -1 representing a child just born and value 1 representing someone of a hundred years old. The variable increases linearly with the person's age.

4. RareTitle: most of the passengers are just called "Mr." or "Ms.", but there are some with a special title, like "Jonkheer". This indicates a rich person, so we do not want to lose this information. A value of 1 indicates that this person has some special title.

5. Singleton: a binary variable to indicate whether a person travelled alone.

6. LargeFamily: a binary variable to indicate whether a person was member of a family consisting of five or more persons.

7. Adult: a binary variable indicating whether the person is older than eighteen.

8. Mother: a binary variable indicating whether the person is a mother.

9. Survived: the binary output variable which indicates whether the person survived the Titanic disaster.

We want to insert two clauses of background knowledge: Survived $\leftarrow$ ˜Pclass$\wedge$ ˜Sex and Survived $\leftarrow$ RareTitle. So, the network will look as shown in figure 4.2.

4.3 Testsetting

For learning both datasets we aimed to use as much the same settings as possible. Also, we want to use the parameters of [5] to check whether we obtain the same results on the Muddy Children data set. All neural networks used backpropagation with momentum as their learning rule. We set the learn factor η at 0.2 and momentum factor ξ at 0.1. We train a single neural network until either the complete train set has passed through the network for 10.000 iterations, or the mean squared error of the output nodes is less than or equal to 0.01.
We evaluate the performance of the Muddy Children networks with k-fold crossvalidation, where $k = 8$. So, we divide the dataset in k subsets of equal

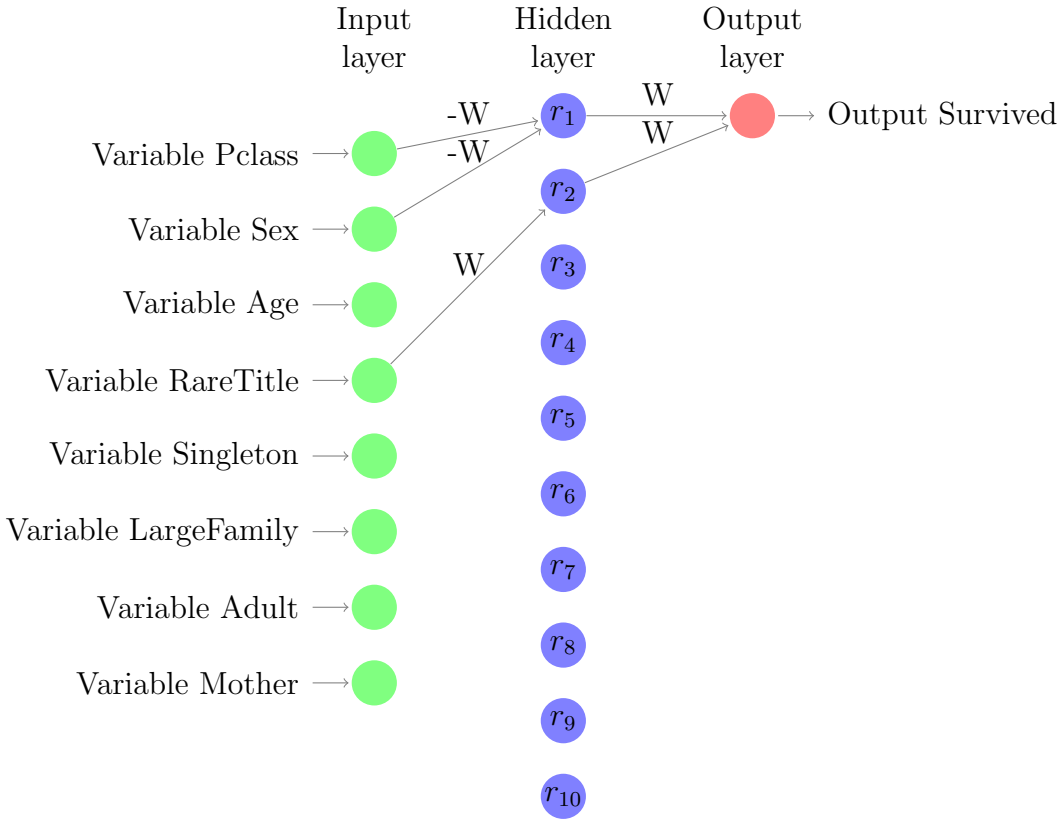

Figure 4.2: The structure of the Titanic network.

size. For each division we train a network on the $k-1$ data sets and test it on the remaining data set. In the end, the results are averaged over the networks [2]. This method is indispensible for small datasets, like the Muddy Children dataset. If we were to use several Muddy Children entries exclusively for testing, we might miss some important features during training. The Titanic data set, however, consists of almost 900 entries. Therefore, it is not necessary to use cross validation. Still, for the sake of retaining our test conditions, we do use five-fold cross validation on the Titanic data set.

4.4 Testresults

For the Muddy Children problem we trained 32 networks, being four times eight-fold cross validation. For the Titanic problem we trained 20 networks, being four times five-fold cross validation. Our results present the average accuracy, where accuracy equals the number correct divided by the size of the test set. Figure 4.3 presents our test results.

Data set	With background	Without
Muddy Children (soft)	0.999	0.999
Muddy Children (strict)	0.664	0.633
Titanic	0.745	0.767

Figure 4.3: Testresults

Chapter 5

Discussion and further research

Sadly, we were not able to produce a significant increase in performance by inserting background knowledge into the neural network. In some cases we even obtained inferior results. This is a very peculiar outcome indeed, and we are left to wonder the exact cause. Therefore, our most important suggestion is to explain the difference in our test results and those published in [5]. We do have some suggestions.

1. Realistically speaking, our program may be at fault. We do not think the CILP algorithm was implemented incorrectly, but rather that the backpropagation algorithm in the Neuroph library was different from the implementation used by the authors of [5]. In that case, our assumptions about the framework would have been incorrect. Also, we want to advise other researchers to try to use a different framework. Encog is an alternative in Java, but an implementations in Python is another option.

2. The problem may also lay in the parts of the algorithm that are dependent on the data set. For example, we need to add as much hidden neurons as neccesary for the convergence of the network [5]. Maybe our number of hidden numbers was not correct. However, a small test of training with 50 hidden neurons did not give a significant increase within 10000 epochs.

3. Maybe our experiment differs on one of more points from the experiment in [5]. In the previous section we already stated that, in our opinion, the authors have not defined Muddy Children data set clearly. We did inquire with the authors, and it turns out they used the soft approach. Also, the authors published the average accuracy over eight networks, while the data set may be divided in a lot more ways. Maybe

the accuracy is very dependent on the folds. Thirdly, we are not sure when the authors classify a train example as correct. The translation algorithm guarantees the output to be higher than A_{min} or lower than $-A_{min}$, but the expanded network does not. So, is a test example that should output 1, correctly classified when the network output $0,001$?

Despite these issues, we still think the research is very relevant for general Artificial Intelligence research, as it combines two major paradigms. Also, the theoretical result of having a neural network approximate the knowledge represented by a logic program, gives us reason to believe that we should, theoretically, be able to improve the performance of neural networks. Also, even if we are be able to reproduce the results published in [5], still more research has to be conducted.

5.1 Further research

First of all, we can investigate the extraction of rules from a neural network. Whereas other machine learning algorithms often are black boxes, this may not be the case for three-layer feedforward networks. The idea is to extract rules after training the network. This way, we can obtain a human readable format of the rules the network has learned. These can be compared to the originally inserted rules. The authors mention this in the last chapter of their book as further research. Still in 2015 it is a challenge, because of the computational complexity and the "need for compact representation" [6].
Apart from the theoretical research, we also need to compare the CILP algorithm to other algorithms. The following tests come to mind:

1. Firstly, we need to compare the performance of our neural network with various other neural networks. In other words, we need to investigate the optimal parameters for both learning problems. The performance of a neural network is very dependent on its learning factor η, its momentum factor ξ, the number of hidden neurons and the stopping conditions. In chapter 3 we already noted the importance of the latter. If we learn for too many epochs, we risk overfitting. If we terminate learning too early, we have not learned all features yet. Also, in recent years various improvements of backpropagation have been proposed.

2. Secondly, we need to compare the performance of our neural network to the performance of various other learning algorithms. In the world of k-Nearest Neighbours, Random Forests and Support Vector Machines,

a neural network constructed with the CILP algorithm is just one fish in the sea. Another algorithm may be more suited to this particular problem. Take for example the family size variable from the Titanic dataset. Investigating the data shows that both people from large families and single persons were more likely to drown. This correlation cannot be translated to a single input neuron, so we created two input neurons to respectively represent the binary variables single person and large family. We may have needed to treat other input neurons the same way. A Random Forest, for example, *can* learn this correlation without special treatment.

3. To conclude, we need to apply the CILP algorithm to several other Machine Learning problems. A co-author of *Neural Symbolic Cognitive Reasoning* already used the algorithm for two problems of DNA sequence analysis [4]. We even encountered a paper that proposed to equip police officers with a digital assistant, appropriatly called Sherlock, programmed as a neural network that received its input as a logic program [9]. Only this way we will understand the strenghts and weaknesses of Neural Symbolic computing and the CILP translation algorithm.

Bibliography

[1] Neuroph: Jave neural network framework. `http://neuroph.sourceforge.net`.

[2] Yaser Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning fom Data*. `www.amlbook.com`, 2012.

[3] George Cybenco. Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[4] Artur d' Avila Garcez and Gerson Zavaruche. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11:59–77, 1999.

[5] Artur S. d' Avila Garcez, Luis C. Lamb, and Dov M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Springer, 2009.

[6] Artur d'Avila Garcez, Tarek Besold, Luc de Raedt, and Peter F editors. *Neural-Symbolic Learning and Reasoning: Contributions and Challenges*.

[7] Artur S. d'Avila Garcez, Luis C. Lamb, and Dov M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag, 2002.

[8] Gary William Flake. *Computational Beauty of Nature*. Bradford Book, 1998.

[9] Ekaterina Komendantskaya and Qiming Zhang. Sherlock - a neural network software for automated problem solving. In *Proceedings of Seventh International Workshop on Neural-Symbolic Learning and Reasoning*, 2011.

[10] Rosja Mastop. Modal logic for artificial intelligence. `http://www.phil.uu.nl/~rumberg/infolai/Modal_Logic.pdf`.

[11] Tom Mitchell. *Machine Learning*. McGraw-Hill Science.

[12] Megan Risdal. Exploring survival on the titanic. `https://www.kaggle.com/mrisdal/titanic/exploring-survival-on-the-titanic/notebook`.

[13] Mark Sergot. Minimal models and fixpoint semantics for definite logic programs. `https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Fixpoint_Definite_461-2x1.pdf`.

[14] Geoffry G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70:119–165, 1994.

[15] Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic Epistemic Logic*. Springer.