Neural Probabilistic Logic Programming in DeepProbLog[☆]

Robin Manhaeve[a,*], Sebastijan Dumančić[a], Angelika Kimmig[a], Thomas Demeester[b,1], Luc De Raedt[a,1]

[a] *KU Leuven*
[b] *Ghent University - imec*

Abstract

We introduce DeepProbLog, a neural probabilistic logic programming language that incorporates deep learning by means of neural predicates. We show how existing inference and learning techniques of the underlying probabilistic logic programming language ProbLog can be adapted for the new language. We theoretically and experimentally demonstrate that DeepProbLog supports (i) both symbolic and subsymbolic representations and inference, (ii) program induction, (iii) probabilistic (logic) programming, and (iv) (deep) learning from examples. To the best of our knowledge, this work is the first to propose a framework where general-purpose neural networks and expressive probabilistic-logical modeling and reasoning are integrated in a way that exploits the full expressiveness and strengths of both worlds and can be trained end-to-end based on examples.

Keywords: logic, probability, neural networks, probabilistic logic programming, neuro-symbolic integration, learning and reasoning

1. Introduction

Many tasks in AI can be divided into roughly two categories: those that require low-level perception, and those that require high-level reasoning. At the same time, there is a growing consensus that being capable of tackling both types of tasks is essential to achieve true (artificial) intelligence [2]. Deep learning is empowering a new generation of intelligent systems that excel at low-level perception, where it is used to interpret images, text and speech with unprecedented accuracy. The success of deep learning has caused a lot of excitement and has also created the impression that deep learning can solve any problem in artificial intelligence. However, there is a growing awareness of the limitations of

[☆] This is an extended and revised version of work previously published at NeurIPS 2018 [1].
[*] Corresponding author
Email address: `robin.manhaeve@cs.kuleuven.be` (Robin Manhaeve)
[1] Joint last authors.

deep learning: deep learning requires large amounts of (the right kind of) data to train the network, it provides neither justifications nor explanations, and the models are black-boxes that can neither be understood nor modified by domain experts. Although there have been attempts to demonstrate reasoning-like behaviour with deep learning [3], their current reasoning abilities are nowhere close to what is possible with typical high-level reasoning approaches. The two most prominent frameworks for reasoning are logic and probability theory. While in the past, these were studied by separate communities in artificial intelligence, many researchers are working towards their integration, and aim at combining probability with logic and statistical learning; cf. the areas of statistical relational artificial intelligence [4, 5] and probabilistic logic programming [6].

The abilities of deep learning and statistical relational artificial intelligence approaches are complementary. While deep learning excels at low-level perception, probabilistic logics excel at high-level reasoning. As such, an integration of the two would have very promising properties. Recently, a number of researchers have revisited and modernized ideas originating from the field of neural-symbolic integration [7], searching for ways to combine the best of both worlds [8, 9, 10, 3], for example, by designing neural architectures representing differentiable counterparts of symbolic operations in classical reasoning tools. Yet, joining the full flexibility of high-level probabilistic and logical reasoning with the representational power of deep neural networks is still an open problem. Elsewhere [11], we have argued that neuro-symbolic integration should: 1) integrate neural networks with the two most prominent methods for reasoning, that is, logic and probability, and 2) that neuro-symbolic integrated methods should have the pure neural, logical and probabilistic methods as special cases.

With DeepProbLog, we tackle the neuro-symbolic challenge from this perspective. Furthermore, instead of integrating reasoning capabilities into a complex neural network architecture, we proceed the other way round. We start from an existing probabilistic logic programming language, ProbLog [12], and introduce the smallest extension that allows us to integrate neural networks: the neural predicate. The idea is simple: in a probabilistic logic, atomic expressions of the form $q(t_1, ..., t_n)$ (aka tuples in a relational database) have a probability p. We extend this idea by allowing atomic expressions to be labeled with neural networks whose outputs can be considered probability distributions. This simple idea is appealing as it allows us to retain all the essential components of the ProbLog language: the semantics, the inference mechanism, as well as the implementation.

Therefore, *one should not only integrate logic with neural networks in neuro-symbolic computation, but also probability.*

This effectively leads to an integration of probabilistic logics (hence statistical relational AI) with neural networks and opens up new abilities. Furthermore, although at first sight, this may appear as a complication, it actually can greatly simplify the integration of neural networks with logic. The reason for this is that the probabilistic framework provides a clear optimisation criterion, namely the probability of the training examples. Real-valued probabilistic quantities are also well-suited for gradient-based training procedures, as opposed to discrete logic quantities.

Example 1
Before going into further detail, the following example illustrates the possibilities of this approach. Consider the predicate $\texttt{addition}(\texttt{X}, \texttt{Y}, \texttt{Z})$, where $\texttt{X}$ and $\texttt{Y}$ are images of handwritten digits from the MNIST dataset [13], and $\texttt{Z}$ is the natural number corresponding to the sum of these digits. The goal is that after training, DeepProbLog allows us to make a probabilistic estimate on the validity of, for example, addition(, , 8). While such a predicate can be learned directly by a standard neural classifier, such an approach cannot incorporate background knowledge such as the definition of the addition of two natural numbers. In DeepProbLog such knowledge can easily be encoded in rules such as

$$\texttt{addition}(\texttt{I}_\texttt{X}, \texttt{I}_\texttt{Y}, \texttt{N}_\texttt{Z}) :- \texttt{digit}(\texttt{I}_\texttt{X}, \texttt{N}_\texttt{X}), \texttt{digit}(\texttt{I}_\texttt{Y}, \texttt{N}_\texttt{Y}), \texttt{N}_\texttt{Z}\ \texttt{is}\ \texttt{N}_\texttt{X} + \texttt{N}_\texttt{Y}$$

with `is` the standard operator of logic programming to evaluate arithmetic expressions. All that needs to be learned in this case is the neural predicate *digit* which maps an image of a digit I_D to the corresponding natural number N_D. The trained network can then be reused for arbitrary tasks involving digits. Our experiments show that this leads not only to new capabilities but also to significant performance improvements. An important advantage of this approach compared to standard image classification settings is that it can be extended to multi-digit numbers without additional training. We note that the single digit classifier (i.e., the neural predicate) is not explicitly trained by itself: its output can be considered a latent representation, as we only use training data with pairwise sums of digits.

To summarize, we introduce DeepProbLog which has a unique set of features: (i) it is a programming language that supports neural networks and machine learning and has a well-defined semantics (ii) it integrates logical reasoning with neural networks; so both symbolic and subsymbolic representations and inference; (iii) it integrates probabilistic modeling, programming and reasoning with neural networks (as DeepProbLog extends the probabilistic programming language ProbLog, which can be regarded as a very expressive directed graphical modeling language [4]); (iv) it can be used to learn a wide range of probabilistic logical neural models from examples, including inductive programming.

This paper is a significantly extended and completed version of our previous work [1]. This extended version now contains the necessary deep learning and probabilistic logic programming background and a more in depth theoretical explanation. It also contains additional experiments (see Section 6): the MNIST addition experiments from the short version are completed with the new experiments **T3** and **T4**, and we designed new experiments (**T8** and **T9**) to further investigate the use of DeepProbLog on combined probabilistic learning and deep learning. We have added a new experiment introducing embeddings that shows that DeepProblog really provides an interface between the neural and the probabilistic logic part that is wide and powerful (**T10**). Finally, we have also added an experiment on the CLUTRR dataset (**T11**).

The code is available at `https://bitbucket.org/problog/deepproblog`.

2. Background

2.1. Logic programming concepts

In this section, we briefly summarize basic logic programming concepts; see e.g., Lloyd [14] for more details. Atoms are expressions of the form $q(t_1, ..., t_n)$ where q is a predicate (of arity n, or q/n in shorthand notation) and the t_i are terms. A literal is an atom or the negation $\neg q(t_1, ..., t_n)$ of an atom. A term t is either a constant c, a variable V, or a structured term of the form $f(u_1, ..., u_k)$ where f is a functor and the u_i are terms. We follow the Prolog convention and let constants, functors and predicates start with a lower case character and variables with an upper case. A rule is an expression of the form $h \coloneq b_1, ..., b_n$ where h is an atom, the b_i are literals, and all variables are universally quantified. Informally, the meaning of such a rule is that h holds whenever the conjunction of the b_i holds. Thus :− represents logical implication ($\leftarrow$), and the comma (,) represents conjunction ($\wedge$). Rules with an empty body $n = 0$ are called facts. A logic program is a finite set of rules.

A substitution $\theta = \{V_1 = t_1, ..., V_n = t_n\}$ is an assignment of terms t_i to variables V_i. When applying a substitution θ to an expression e we simultaneously replace all occurrences of V_i by t_i and denote the resulting expression as $e\theta$. Expressions that do not contain any variables are called ground. The *Herbrand base* of a logic program is the set of ground atoms that can be constructed using the predicates, functors and constants occurring in the program.[2] Subsets of the Herbrand base are called *Herbrand interpretations*. A Herbrand interpretation is a *model* of a clause $h \coloneq b_1, \ldots, b_n$. if for every substitution θ such that the conjunction $(b_1, \ldots, b_n)\theta$ holds in the interpretation, $h\theta$ is in the interpretation. It is a model of a logic program if it is a model of all clauses in the program.

For negation-free programs, the semantics is given by the minimal such model, known as the least Herbrand model, which is unique. General logic programs use the notion of negation as failure, that is, the negation of an atom is true exactly if the atom cannot be derived from the program. These programs are not guaranteed to have a unique minimal Herbrand model, and several ways to define a canonical model have been studied. We follow the well-founded semantics here [15].

The main inference task in logic programming is to determine whether a given atom q, also called *query* (or *goal*), is true in the canonical model of a logic program P, denoted by $P \models q$. If the answer is yes (or no), we also say that the query *succeeds* (or *fails*). If such a query is not ground, inference asks for the existence of an *answer substitution*, that is, a substitution that grounds the query into an atom that is part of the canonical model.

[2]If the program does not contain constants, one arbitrary constant is added.

2.2. Deep Learning

The following paragraphs introduce some key concepts in the field of deep learning, needed for understanding the training mechanism of DeepProbLog. We however do not provide details on standard architectural building blocks such as convolutional or recurrent neural networks which are used in some of the experiments in Section 6; instead, we refer the reader to excellent texts such as [16].

An artificial neural network is a highly parameterized and therefore very flexible non-linear mathematical function that can be 'trained' towards a particular desired behavior, by suitably adjusting its parameters. During training, the model learns to capture from the input data the most informative 'features' for the task at hand. The need for 'feature engineering' in classical (or rather, non-neural) machine learning methods has therefore been replaced by 'architecture engineering', since a wide variety of neural network components are available to be composed into a suitable model.

Deep neural networks are often designed and trained in an 'end-to-end' fashion, whereby only the raw input and the final target are known during training, and all components of the model are jointly trained. For example, for the task of hand-written digit recognition, an input instance consists of a pixel image of a hand-written digit, whereas its target denotes the actual digit.

Consider a supervised learning problem, with a training set $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ containing N i.i.d. input instances $\boldsymbol{x}_i$ and corresponding outputs $\boldsymbol{y}_i$. A model represented by a mapping function $\mathcal{M}$ with parameters Θ, maps an input item $\boldsymbol{x}$ to the corresponding predicted output $P(Y|X = \boldsymbol{x}) = \mathcal{M}(\boldsymbol{x}|\Theta)$.

To quantify how strongly the predicted output $P(Y|X = \boldsymbol{x})$ deviates from the target output $\boldsymbol{y}$, a loss function $\mathcal{L}(P(Y|X = \boldsymbol{x}), \boldsymbol{y})$ is defined. Training the model then comes down to minimizing the expected loss

$$\bar{\mathcal{L}} = \frac{1}{N} \sum_i \mathcal{L}\big(\mathcal{M}(\boldsymbol{x}_i|\Theta), \boldsymbol{y}_i\big)$$

over the training set. In the specific setting of multiclass classification, each input instance corresponds to one out of a fixed set of M output categories. The target vectors $\boldsymbol{y}$ are typically represented as one-hot vectors: all components are zero, except at index m of the corresponding category. The predicted counterpart $P(Y|X = \boldsymbol{x})$ at the model's output is often obtained by applying a so-called softmax output function to intermediate real-valued scores $\mathbf{s}$ obtained at the output of the neural network.

The i-th component of the softmax is defined as

$$\text{softmax}(\mathbf{s})_i = \hat{y}_i = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

The softmax outputs are well-suited to model a probability distribution (i.e., $0 < \hat{y}_i < 1$ and $\sum_i \hat{y}_i = 1$). The standard corresponding loss function is the cross-entropy loss, which quantifies the deviation between the empirical output distribution $\hat{\boldsymbol{y}}$ (i.e., the softmax outputs) and the ground truth distribution (i.e., the one-hot target vector $\boldsymbol{y}$) defined as

$$\mathcal{L} = -\sum_j y_j \log \hat{y}_j$$

The neural network models we will focus on in this work, are *discriminative* classifiers, which directly model the probability distribution $P(Y|X = \boldsymbol{x}_i)$ of the target classes, given the input feature representation $\boldsymbol{x}_i$ of the i-th example. We will show how these can be used to build neural predicates as a probabilistic logic programming component (see Section 3). We currently do not consider *generative* classification models. These would also allow modeling the output labels given the inputs, by applying Bayes' law on the prior class probabilities and the inputs distribution given the labels. Yet, we argue that taking into account background information such as prior class probabilities is more naturally handled by the reasoning component, and that modeling the inputs distribution may be overly complex and is not needed for the purpose of these neural predicates.

The most widely used optimization approaches for neural networks are variations of the gradient descent algorithm, in which the parameters Θ are iteratively updated by taking small steps along the negative gradient of the loss. An estimate Θ_n at iteration n is updated as $\Theta_{n+1} = \Theta_n - \lambda \nabla_\Theta \bar{\mathcal{L}}$, in which the step size is controlled by the learning rate λ. Typically, training is not performed by averaging over the entire dataset per iteration, but instead over a smaller 'mini-batch' of instances. This is computationally more efficient and allows for a better exploration of parameter space. Importantly, the loss gradient can only be calculated if all components of the neural network are differentiable.

A deep neural network typically has a layer-wise architecture: the different layers correspond to nested differentiable functions in the overall mapping function $\mathcal{M}$. The 'forward pass' through the network corresponds to consecutively applying these layer functions to a given input to the network. The intermediate representations obtained by evaluating these layer functions are called hidden states. After a forward pass, the gradient with respect to all parameters can then be calculated by applying the chain rule. This happens during the so-called 'backward pass': the gradients are calculated from the output back to the first layer. As an illustration of how the chain rule is applied, consider the network function $\mathcal{M}(\boldsymbol{x}|\Theta) = \mathbf{g}\big(\mathbf{f}(\boldsymbol{x}, \theta_f), \theta_g\big)$, which contains a first layer represented by the vector function $\mathbf{f}$, and a second layer $\mathbf{g}$. For simplicity, say each layer has one trainable parameter, respectively written as θ_f and θ_g. The derivative with respect to these parameters of a scalar loss function applied to the network output, becomes

$$\nabla_\Theta \mathcal{L}\big(\mathcal{M}(\boldsymbol{x}|\Theta)\big) = \left[\frac{d\mathcal{L}}{d\theta_f}, \frac{d\mathcal{L}}{d\theta_g}\right] = \left[\sum_i \frac{\partial \mathcal{L}}{\partial g_i} \sum_j \frac{\partial g_i}{\partial f_j} \frac{\partial f_j}{\partial \theta_f}, \sum_i \frac{\partial \mathcal{L}}{\partial g_i} \frac{\partial g_i}{\partial \theta_g}\right]$$

in which the individual derivatives are evaluated based on the considered input $\boldsymbol{x}$ and current value of the parameters. The entire procedure to calculate the gradients is called the backpropagation algorithm. It requires a forward pass to calculate all intermediate representations up to the value of the loss. After that, in the backward pass, the gradients corresponding to all operations applied during the forward pass, are iteratively calculated, starting at the loss (i.e., with $\partial\mathcal{L}/\partial g_i$ in the example). As such, the gradients with respect to parameters at a given layer can be calculated as soon as the gradients due to all operations further in the network are known, as governed by the chain rule.

To summarize, a single iteration in the optimization happens as follows: 1) A minibatch is sampled from the training data. 2) The output of the neural network is calculated during the forward pass. 3) The loss is calculated based on that output and the target. 4) The gradients for the parameters in the neural network are calculated using backpropagation. 5) The parameters are updated using a gradient-based optimizer.

3. Introducing DeepProbLog

We now recall the basics of probabilistic logic programming using ProbLog (see De Raedt and Kimmig [6] for more details), and then introduce our new language DeepProbLog.

3.1. ProbLog

Definition 1 (ProbLog program)
A ProbLog program consists of a set of probabilistic facts $\mathcal{F}$ of the form $p :: f$ where p is a probability and f an atom, and a set of rules $\mathcal{R}$. $\triangleleft$

For instance, the following ProbLog program models a variant of the well-known alarm Bayesian network [17]:

$$\begin{aligned}
0.1 &:: \texttt{burglary}.\\
0.5 &:: \texttt{at\_home}(\texttt{mary}).\\
0.2 &:: \texttt{earthquake}.\\
0.4 &:: \texttt{at\_home}(\texttt{john}).
\end{aligned}$$

$$\begin{aligned}
\texttt{alarm} &:- \texttt{earthquake}.\\
\texttt{alarm} &:- \texttt{burglary}.\\
\texttt{calls}(\texttt{X}) &:- \texttt{alarm}, \texttt{at\_home}(\texttt{X}).
\end{aligned}$$

Each ground instance $f\theta$ of a probabilistic fact f corresponds to an *independent Boolean random variable* that is true with probability p and false with probability $1-p$. Let us denote the set of all ground instances of probabilistic facts in $\mathcal{F}$ as $\mathcal{F}\Theta$. Every subset $F \subseteq \mathcal{F}\Theta$ defines a possible world $w_F =$

$F \cup \{h\theta | \mathcal{R} \cup F \models h\theta$ and $h\theta$ is ground$\}$, that is, the world w_F is the canonical model of the logic program obtained by adding F to the set of rules $\mathcal{R}$, e.g.,

$$w_{\{\texttt{burglary},\texttt{at\_home(mary)}\}} = \{\texttt{burglary}, \texttt{at\_home(mary)}\} \cup \{\texttt{alarm}, \texttt{calls(mary)}\}$$

To keep the presentation simple, we focus on the case of finitely many ground probabilistic facts, but note that the semantics is also well-defined for the countably infinite case. The probability $P(w_F)$ of such a possible world w_F is given by the product of the probabilities of the truth values of the probabilistic facts:

$$P(w_F) = \prod_{f_i \in F} p_i \prod_{f_i \in \mathcal{F}\Theta \setminus F} (1 - p_i) \tag{1}$$

For instance,

$$P(w_{\{\texttt{burglary},\texttt{at\_home(mary)}\}}) = 0.1 \times 0.5 \times (1 - 0.2) \times (1 - 0.4) = 0.024$$

The probability of a ground atom q, also called *success probability of* q, is then defined as the sum of the probabilities of all worlds containing q, i.e.,

$$P(q) = \sum_{F \subseteq \mathcal{F}\Theta : q \in w_F} P(w_F) \tag{2}$$

The probability of a query is also equal to the weighted model count (WMC, see [18] for more details) of the worlds where this query is true.

For ease of modeling, ProbLog supports non-ground probabilistic facts as a shortcut for introducing a set of ground probabilistic facts, as well as annotated disjunctions (ADs), which are expressions of the form

$$p_1 :: h_1 \ ; \ ... \ ; \ p_n :: h_n \ \text{:-} \ b_1, ..., b_m.$$

where the p_i are probabilities that sum to at most one, the h_i are atoms, and the b_j are literals. The meaning of an AD is that whenever all b_i hold, the AD causes one of the heads h_j to be true, or none of them with probability $1 - \sum p_i$. Note that several of the h_i may be true at the same time if they also appear as heads of other rules or ADs. This is convenient to model choices between different categorical variables, e.g. different severities of the earthquake:

$$0.4 :: \texttt{no\_earthquake} \ ; \ 0.4 :: \texttt{mild\_earthquake} \ ; \ 0.2 :: \texttt{severe\_earthquake}.$$

or without explicitly representing the event of no earthquake:

$$0.4 :: \texttt{mild\_earthquake} \ ; \ 0.2 :: \texttt{severe\_earthquake}.$$

In which neither `mild_earthquake` nor `severe_earthquake` will be true with probability 0.4. Annotated disjunctions do not change the expressivity of ProbLog, as they can alternatively be modeled through independent facts and logical rules; we refer to De Raedt and Kimmig [6] for technical details.

To obtain some intuitions about the probabilistic logic program representation, it is instructive to show how they can represent Bayesian networks. Let us show this for Bayesian networks involving Boolean random variables. Each node without a parent then corresponds to a probabilistic fact. Observe that both nodes without any parents in a Bayesian network and probabilistic facts in ProbLog are marginally independent. Furthermore, each entry in a conditional probability table would be mapped onto an annotated disjunction. Assume the parents of the node n are x and y. Then there would be four annotated disjunctions of the form $p_i :: n \,{:}{-}\, x_v, y_v$, where x_v and y_v are the positive or negative literals corresponding to x, resp. y. This shows that annotated disjunctions can be used to specify conditional probabilities.

3.2. DeepProbLog

In ProbLog, the probabilities of all random choices are explicitly specified as part of probabilistic facts or annotated disjunctions. DeepProbLog extends ProbLog to basic random choices whose probabilities are specified through external functions implemented as neural networks.

Definition 2 (Neural annotated disjunction)
A *neural annotated disjunction* (nAD) is an expression of the form

$$nn(m_r, [X_1, ..., X_k], O, [y_1, ..., y_n]) :: r(X_1, ..., X_k, O)$$

where nn is a reserved functor, m_r uniquely identifies a neural network model (i.e., its architecture as well as its trainable parameters) that defines a probability distribution $p_{m_r}(O|X = \boldsymbol{x})$ over the domain $O \in \{y_1, ..., y_n\}$ given the input $\boldsymbol{x} = [x_1, .., x_k]$, $X_1, ..., X_k$ are variables representing the inputs to the neural network, O is the output variable, the ground terms $y_1, ..., y_n$ define the domain of the output distribution, and r is a predicate symbol. Note that the arguments of predicate r can be in an arbitrary order.

Formally, such a neural AD represents a set of *ground neural ADs* of the following form, one for every sequence of ground terms $x_1, ..., x_k$ representing inputs to the neural network:

$$\begin{aligned} & p_{m_r}(O = y_1 | X_1 = x_1, \ldots, X_k = x_k) :: r(x_1, ..., x_k, y_1) \ ; \\ & ... \ ; \\ & p_{m_r}(O = y_n | X_1 = x_1, \ldots, X_k = x_k) :: r(x_1, ..., x_k, y_n) \end{aligned}$$

The neural network thus represents a discriminative classifier, which naturally maps onto an annotated disjunction for each input. For instance, in the MNIST addition example, we would specify the nAD

$$\texttt{nn(m\_digit}, [\texttt{X}], \texttt{Y}, [\texttt{0}, \ldots, \texttt{9}]) :: \texttt{digit(X}, \texttt{Y)}.$$

where `m_digit` is a network that classifies MNIST digits. For input image , the ground nAD is

$\texttt{p}_{\texttt{m\_digit}}(\texttt{Y} = \texttt{0}|\texttt{X} =$ $) :: \texttt{digit(}$ $, \texttt{0)}$; ... ; $\texttt{p}_{\texttt{m\_digit}}(\texttt{Y} = \texttt{9}|\texttt{X} =$ $) :: \texttt{digit(}$ $, \texttt{9)}$.

The neural network could take any shape, e.g., a convolutional network for image encoding, a recurrent network for sequence encoding, etc. However, its output layer, which feeds the corresponding neural predicate, needs to be normalized.

We consider an output domain size of two as a special case. Instead of the neural network having two probabilities at the output that sum to one, we can simplify this to a single probability, with the second one the complement of that probability. This difference coincides with the difference between a softmax and single-neuron sigmoid layer in a neural network. We call such an expression a neural fact.

Definition 3 (Neural fact)
A neural fact is an expression of the form

$$nn(m_r, [X_1, ..., X_k]) :: r(X_1, ..., X_k).$$

where nn is a reserved functor, m_r uniquely identifies a neural network model that defines a probability distribution $p_{m_r}(O|X = \boldsymbol{x})$ over the Boolean domain $\{1, 0\}$, given the input $\boldsymbol{x} = [x_1, .., x_k]$, $X_1, ..., X_k$ are variables representing the inputs to the neural network, and r is a predicate symbol. Note that the arguments of predicate r can be in an arbitrary order.

Formally, such a neural fact represents a set of *ground neural facts* of the following form, one for every sequence of ground terms $x_1, ..., x_k$ representing inputs to the neural network:

$$p_{m_r}(O = 1|X_1 = x_1, ..., X_k = x_k) :: r(x_1, ..., x_k).$$

The neural network can be viewed as a discriminative classifier that determines the truth value of the fact.

To exemplify, we use a neural network that gives a measure of the similarity between two input images. We can encode this with the following neural fact:

$$\texttt{nn(m, [X1, X2]) :: similar(X1, X2)}.$$

For input and , the ground neural fact is

$p_m(O = 1|X1 =$ $, X2 =$ $) :: similar($ $,$ $).$

Definition 4 (DeepProbLog Program)
A DeepProbLog program consists of a set of probabilistic facts $\mathcal{F}$, a set of neural facts $\mathcal{N}_f$, a set of neural ADs $\mathcal{N}_{ad}$, and a set of rules $\mathcal{R}$. $\triangleleft$

The semantics of a DeepProbLog program with respect to a fixed set of possible inputs to every neural network used in the neural facts and neural ADs is the semantics of the ProbLog program that replaces each neural fact and neural AD by the corresponding sets of ground neural facts and ground neural ADs for these inputs.

It is worth elaborating on the fact that this includes the inputs to the neural networks, such as MNIST images, into the Herbrand domain of the program. We discuss two possible ways to view neural inputs and outputs (e.g. MNIST images). Both explicitly introduce neural inputs through type predicates, but they differ on how the inputs are represented. For instance, we could add MNIST images through the type `image`:

$$\texttt{nn}(\texttt{m}, [\texttt{X1}, \texttt{X2}]) :: \texttt{similar}(\texttt{X1}, \texttt{X2}) : -\texttt{image}(\texttt{X1}), \texttt{image}(\texttt{X2}).$$

In the first view, the `image` predicate simply introduces one constant per input. This implies, however, that neural inputs, just like other constants in the Herbrand domain, are part of the program, and have to be specified upfront. Furthermore, the possible worlds are then, so to speak, conditioned on the possible inputs to the neural networks and the possible worlds will only contain these neural inputs and no other ones. This perspective is in line with the discriminative training that we propose in Section 5. This excludes the possibility of using neural networks in a generative fashion where "new" images or constants could be generated.

In the second view, neural inputs are represented as structured terms that provide an abstract language to define the possible inputs, rather than enumerating them explicitly. For instance, an image is just a matrix of numbers. Under this view, the predicate `image` could simply allow for lists containing exactly 28×28 numbers between 0 and 255, which would include all possible images in the Herbrand domain. This view would also allow one to represent generative neural networks as neural predicates, and would consider possible worlds that contain all possible images. One approach going in that direction is that of SPLog [19], which combines DeepProbLog with Sum-Product networks to represent variational and probabilistic auto-encoders as neural predicates.

4. Inference

This section explains how a DeepProbLog model is used for a given query at prediction time. First, we provide more detail on ProbLog inference [20]. Next, we describe how ProbLog inference is adapted in DeepProbLog.

4.1. ProbLog Inference

ProbLog inference proceeds in four steps. The first step is the grounding step, in which the logic program is grounded with respect to the query. This step uses backward reasoning to determine which ground rules are relevant to derive the truth value of the query, and may perform additional logical simplifications that do not affect the query's probability.

The second step rewrites the ground logic program into a formula in propositional logic that defines the truth value of the query in terms of the truth values of probabilistic facts. We can calculate the query success probability by performing *weighted model counting* (WMC) on this logic formula (cfr. Fierens et al. [20]). However, performing WMC on this logical formula directly is not efficient.

The third step is knowledge compilation [21]. During this step, the logic formula is transformed into a form that allows for efficient weighted model counting. The current ProbLog system uses Sentential Decision Diagrams (SDDs, Darwiche [22]), the most succinct suitable representation available today. SDDs, being a subset of deterministic decomposable negational normal forms (d-DNNFs) allow for polytime model counting ([21]). However, they also support polytime conjunction, disjunction and negation while being more succinct than OBDDs (Darwiche [22]).

The fourth and final step transforms the SDD into an arithmetic circuit (AC). This is done by putting the probabilities of the probabilistic facts or their negations on the leaves, replacing the OR nodes with addition and the AND nodes by multiplication. The WMC is then calculated with an evaluation of the AC.

Example 2
In Figure 1, we apply the four steps of ProbLog inference on the earthquake example with query `calls(mary)`.
In the first step, the non-ground program (Figure 1a) is grounded with respect to the query `calls(mary)`. The result is shown in Figure 1b: the irrelevant fact `at_home(john)` is omitted and the variable `X` in the `calls` rule is substituted with the constant `mary`. The resulting formula in the second step is

$$\texttt{calls}(\texttt{mary}) \leftrightarrow \texttt{at\_home}(\texttt{mary}) \land (\texttt{burglary} \lor \texttt{earthquake})$$

The WMC of this formula is shown in Figure 1c. However, it is not calculated by enumeration as shown here, but an AC is used instead. The AC derived in step four is shown in Figure 1d, where rounded grey rectangles depict variables corresponding to probabilistic facts, and the rounded red rectangle denotes the query atom defined by the formula. The white rectangles correspond to logical operators applied to their children. The intermediate results are shown in black next to the nodes in Figure 1d.

4.2. DeepProbLog Inference

The only change required for DeepProbLog inference is that we need to instantiate nADs and neural facts to regular ADs and probabilistic facts. This is done in two steps. During grounding, we obtain ground nADs and ground neural facts with a symbolic representation of the probabilities. In a separate step after grounding, the concrete parameters are determined by making a forward pass on the relevant neural network with the ground input.

```
0.2::earthquake.
0.1::burglary.
0.5::at_home(mary).
0.4::at_home(john).
alarm :- earthquake.
alarm :- burglary.
calls(X):-alarm,at_home(X).
```

(a) The ProbLog program.

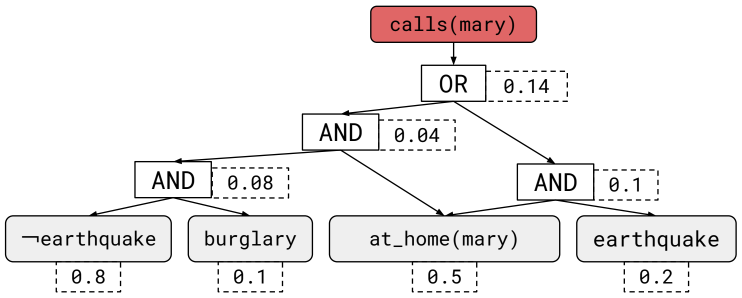
```
0.2::earthquake.
0.1::burglary.
0.5::at_home(mary).

alarm :- earthquake.
alarm :- burglary.
calls(mary):-alarm,at_home(mary).
```

(b) The relevant ground program.

Models of `calls(mary)` ↔ `at_home(mary)` ∧ (`burglary` ∨ `earthquake`)	w
{}	0.36
{`at_home(mary)`}	0.36
{`earthquake`}	0.09
{`earthquake, at_home(mary),calls(mary)`}	**0.09**
{`burglary`}	0.04
{`burglary, at_home(mary),calls(mary)`}	**0.04**
{`burglary, earthquake`}	0.01
{`burglary, earthquake, at_home(mary),calls(mary)`}	**0.01**
$\sum_{\texttt{calls(mary)}\in\text{model}}$	**0.14**

(c) The weighted count of the models where `calls(mary)` is true.



(d) The AC for query `calls(mary)`.

Figure 1: Inference in ProbLog using query `calls(mary)` and the program in (a). (Example 2)

Example 3
We illustrate this by evaluating the MNIST addition example (Figure 2a). The DeepProbLog program requires two lines: the first line defining the neural predicate, and the second line defining the addition. We evaluate it on the query `addition(` , , `1)`. In the first step, the DeepProbLog program is grounded into a ground DeepProbLog Program (Figure 2b). Note that the nADs are now all ground. As ProbLog only grounds the relevant part of the program, i.e. the part that can be used to prove the query, only the digits 0 and 1 are retained as the larger digits cannot sum to 1. The next step is the only difference between ProbLog and DeepProbLog inference: instantiating the ground nADs into regular ground ADs, which could, for instance, produce an AD as shown in Figure 2c. The probabilities in the instantiated ADs do not sum to one, as the irrelevant terms (`digit(` , `2)`, ...,`digit(` , `9)` and `digit(` , `2)`, ..., `digit(` , `9)`) have been dropped in the grounding process, although the neural network still assigns probability mass to them. Inference then proceeds identically to that of ProbLog: the ground program is rewritten into a logical formula, this formula is compiled and transformed into an AC. Finally, this AC is evaluated to calculate the query probability.

```
nn(m_digit, [X], Y, [0...9]) :: digit(X,Y).
addition(X,Y,Z) :- digit(X,N1), digit(Y,N2), Z is N1+N2.
```

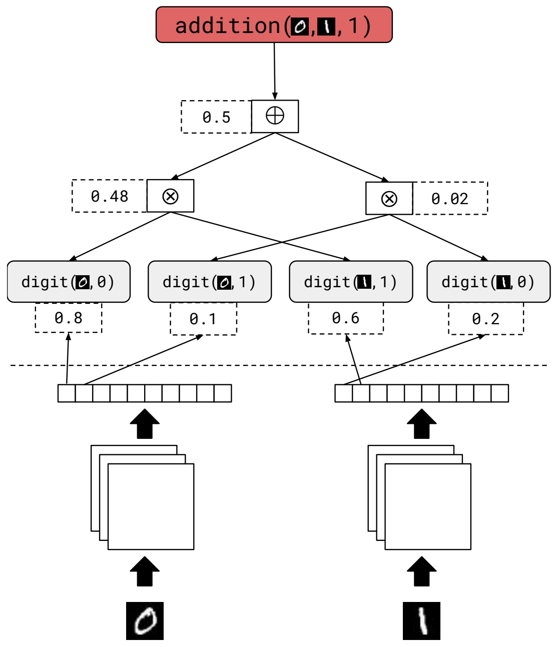
(a) The DeepProbLog program.

```
nn(m_digit,[ ],0)::digit( ,0);nn(m_digit,[ ], 1)::digit( ,1).
nn(m_digit,[ ],0)::digit( ,0);nn(m_digit,[ ], 1)::digit( ,1).
addition( , ,1) :- digit( ,0), digit( ,1).
addition( , ,1) :- digit( ,1), digit( ,0).
```

(b) The ground DeepProbLog program.

```
0.8 :: digit( ,0); 0.1 :: digit( ,1).
0.2 :: digit( ,0); 0.6 :: digit( ,1).
addition( , ,1) :- digit( ,0), digit( ,1).
addition( , ,1) :- digit( ,1), digit( ,0).
```

(c) The ground ProbLog program.



(d) The AC for query `addition( , , 1)`.

Figure 2: Inference in DeepProbLog (Example 3)

5. Learning in DeepProbLog

We now introduce our approach to learn the parameters in DeepProbLog programs. The parameters include the learnable parameters of the neural network (which we will call neural parameters from now on) and the learnable parameters in the logic program (which we will refer to as probabilistic parameters).

ProbLog, just like any other statistical relational AI model such as Markov Logic [4, 23], can be trained discriminatively as well as a generatively. In the generative setting, the examples are (partial) possible worlds or interpretations, while in the discriminative setting they correspond to facts for a specific target predicate with the evidence residing in the background theory. In the present paper, we use the discriminative training setting, which is called zw4zw*learning from entailment* [24] in ProbLog.

Definition 5
Learning from entailment Given a DeepProbLog program with parameters Θ, a set $\mathcal{Q}$ of tuples $(q, \mathcal{X}, p)$ with q a query, $\mathcal{X}$ the neural input for this query and p its desired success probability , and a loss function $\mathcal{L}$, compute:

$$\arg\min_{\Theta} \frac{1}{|\mathcal{Q}|} \sum_{(q,\mathcal{X},p)\in\mathcal{Q}} \mathcal{L}(P(q|\mathcal{X},\Theta), p)$$

In most of the experiments, unless mentioned otherwise, we only use positive examples for training (i.e., with desired success probability $p = 1$). The model then needs to adjust the weights to maximize query probabilities $P_\Theta(q|\mathcal{X})$ for all training examples. This can be expressed by minimizing the average negative log likelihood of the query, whereby Definition 5 reduces to:

$$\arg\min_{\Theta} \frac{1}{|\mathcal{Q}|} \sum_{(q,p)\in\mathcal{Q}} -\log P_\Theta(q|\mathcal{X})$$

The presented method however works for other choices in the loss function. For example, in experiment **T9** (Section 6.3) the mean squared error (MSE) is used.

5.1. Gradient-based learning in ProbLog

In contrast to the earlier approach for ProbLog parameter learning in this setting by Gutmann et al. [25], we use gradient based learning instead of EM. This allows for seamless integration with neural network training. The key insight here is that we can use the same AC that ProbLog uses for inference for gradient computations as well. This AC is a differentiable structure, as it is composed of addition and multiplication operations.

In this work, we rely on the automatic differentiation capabilities already available in ProbLog to derive these gradients. More specifically, to compute the gradient with respect to the probabilistic logic program part, we rely on Algebraic ProbLog (aProbLog [26]), a generalization of the ProbLog language and inference to arbitrary commutative semirings, including the gradient semiring

[27]. Whereas ProbLog is confined to only calculating probabilities, the use of this gradient semiring in aProbLog allows the system to calculate the gradient alongside the probabilities. In the following, we provide the necessary background on aProbLog, discuss how to use it to compute gradients with respect to ProbLog parameters and extend the approach to DeepProbLog.

aProbLog and the gradient semiring. ProbLog annotates each probabilistic fact f with the probability p that f is true, which implicitly also defines the probability $1-p$ that its negation $\neg f$ is true. It then uses the probability semiring with regular addition and multiplication as operators to compute the probability of a query on the AC constructed for this query, cf. Figure 1d. The probability semiring is defined as follows:

$$a \oplus b = a + b \tag{3}$$
$$a \otimes b = ab \tag{4}$$
$$e^{\oplus} = 0 \tag{5}$$
$$e^{\otimes} = 1 \tag{6}$$

And the accompanying labeling function as:

$$L(f) = p \qquad \text{for } p :: f \tag{7}$$
$$L(\neg f) = 1 - p \qquad \text{with } L(f) = p \tag{8}$$

This idea is generalized in aProbLog to compute such values based on arbitrary commutative semirings. Instead of probability labels on facts, aProbLog uses a labeling function that explicitly associates values from the chosen semiring with both facts and their negations, and combines these using semiring addition $\oplus$ and multiplication $\otimes$ on the AC. We use the gradient semiring, whose elements are tuples $(p, \frac{\partial p}{\partial \theta})$, where p is a probability (as in ProbLog), and $\frac{\partial p}{\partial \theta}$ is the partial derivative of that probability with respect to a parameter θ, that is, the probability p_i of a probabilistic fact with learnable probability, written as $t(p_i) :: f_i$. This is easily extended to a vector of parameters $\vec{\theta} = [\theta_1, \ldots, \theta_N]^T$, the concatenation of all N probabilistic parameters in the ground program, as it is easier and faster to process all gradients in one vector. Semiring addition $\oplus$, multiplication $\otimes$ and the neutral elements with respect to these operations are defined as follows:

$$(a_1, \vec{a_2}) \oplus (b_1, \vec{b_2}) = (a_1 + b_1, \vec{a_2} + \vec{b_2}) \tag{9}$$
$$(a_1, \vec{a_2}) \otimes (b_1, \vec{b_2}) = (a_1 b_1, b_1 \vec{a_2} + a_1 \vec{b_2}) \tag{10}$$
$$e^{\oplus} = (0, \vec{0}) \tag{11}$$
$$e^{\otimes} = (1, \vec{0}) \tag{12}$$

Note that the first element of the tuple mimics ProbLog's probability computation, whereas the second simply computes gradients of these probabilities using derivative rules.

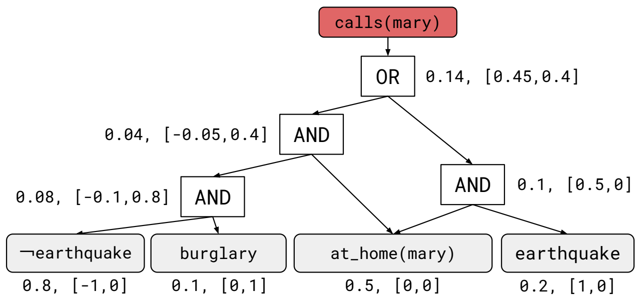


Figure 3: The AC evaluated using the gradient semiring. (Example 4)

Gradient-based learning with aProbLog. To use the gradient semiring for gradient based parameter learning in ProbLog, we first transform the ProbLog program into an aProbLog program by extending the label of each probabilistic fact $p :: f$ to include the probability p as well as the gradient vector of p with respect to the probabilities of all probabilistic facts and ADs in the program, i.e.,

$$L(f) = (p, \vec{0}) \qquad \text{for } p :: f \text{ with fixed } p \tag{13}$$
$$L(f_i) = (p_i, \mathbf{e}_i) \qquad \text{for } t(p_i) :: f_i \text{ with learnable } p_i \tag{14}$$
$$L(\neg f) = (1 - p, -\nabla p) \qquad \text{with } L(f) = (p, \nabla p) \tag{15}$$

where the vector $\mathbf{e}_i$ has a 1 in the i-th position and 0 in all others. For fixed probabilities, the gradient does not depend on any parameters and thus is 0. Note that after each update step, the probabilistic parameters are clipped to the $[0, 1]$ range, and the parameters of an AD are re-normalized to ensure that they sum to one. For the other cases, we use the semiring labels as introduced above.

Example 4
Assume we want to learn the probabilities of `earthquake` and `burglary` in the example of Figure 1, while keeping those of the other facts fixed. Figure 3 shows the evaluation of the same AC as in Figure 1d, but with the gradient semiring. The nodes in the AC now also contain the gradient (the second element of the tuple). The result on the top node shows that the partial derivative of the query is 0.45 and 0.4 with respect to the earthquake and burglary parameters respectively.

5.2. Gradient-based learning for DeepProbLog

Just as the only difference between inference in ProbLog and DeepProbLog is the evaluation of the neural facts and nADs, the only difference between gradient-based learning in ProbLog and DeepProbLog is optimizing the neural parameters alongside the probabilistic parameters. As mentioned in the previous section, the probabilistic parameters p_i in the logic program can be optimized by using the gradient semiring, which allows us to calculate $\partial P(q)/\partial p_i$. This gradient is then used to perform the update by using gradient-based learning. Note that since the outputs of the neural networks are used as probabilistic facts in the logic program, they are leafs in the AC. Conceptually, it might be easiest to think of the AC being connected with the differentiable structure of the neural network at these leaves (cfr. Figure 4c). This creates a single differentiable structure in which standard differentiation techniques can be used to derive gradients.

However, in this work, we do not explicitly join the AC and neural networks. Instead, we rely on the aProbLog method as described above. This makes the joint learning of probabilistic and neural parameters more straightforward. Note however, that the gradients for the internal parameters inside of the neural networks are still derived using backpropagation. The outputs of the neural networks can be considered *abstract* parameters. Although we can derive a gradient for these abstract parameters, we cannot optimize them directly, as the logic is unaware of the neural parameters that determine the value of these abstract parameters. Recall from Equation (1) that the gradient of the internal (neural) parameters in standard supervised learning can be derived using the chain rule in backpropagation. Below, we show how we can derive the gradient for these neural parameters of the loss applied to $P(q|\mathcal{X})$ (Definition 5), rather than a loss function defined directly on the output of the neural network.

Specifically, consider the case of a single neural annotated disjunction, with probabilities $\hat{p}_i$ (i.e., the aforementioned abstract parameters), calculated by evaluating a neural network with softmax output. The predicted probability that the query holds true, based on the current values of the neural and probabilistic parameters, is written $P(q|\mathcal{X})$. While training, true examples should yield a predicted query probability close to the expected query probability, which is expressed by means of a loss function $\mathcal{L}$ as introduced in Definition 5.

Application of the chain rule leads to

$$\frac{d\mathcal{L}}{d\theta_k} = \frac{\partial \mathcal{L}}{\partial P(q|\mathcal{X})} \sum_i \frac{\partial P(q|\mathcal{X})}{\partial \hat{p}_i} \frac{\partial \hat{p}_i}{\partial \theta_k}$$

where the derivative of the loss with respect to any trainable parameter θ_k in the neural network is decomposed into the partial derivative of the loss with respect to the predicted output $P(q|\mathcal{X})$, the latter's derivative $\partial P(q|\mathcal{X})/\partial \hat{p}_i$ with respect to each head in the annotated disjunction as obtained with the gradient semiring, and finally $\partial \hat{p}_i/\partial \theta_k$, the derivative of the neural network's output components with respect to the considered parameter. The latter is obtained by the standard application of the chain rule in the neural network. The backpropagation procedure in the neural network can thus be started by providing $\partial P(q|\mathcal{X})/\partial \hat{p}_i$, to systematically obtain the loss gradients for all neural parameters.
Extending this approach to the situation of multiple neural predicates is straightforward. If the same neural network is used for different neural predicates (e.g. in Example 3), the final derivative is obtained by summing over the contributions of each neural predicate.

Then, standard gradient-based optimizers (e.g. SGD, Adam [**?**], ...) are used to update the parameters of the network. During gradient computation with aProbLog, the probabilities of neural ADs are kept constant. Furthermore, updates on neural ADs come from the neural network part of the model, where the use of a softmax output layer ensures a normalized distribution, hence not requiring the additional normalization as for non-neural ADs.

To extend the gradient semiring to DeepProbLog programs, we define it for nADs and neural facts. The label for the nAD is defined as:

$$L(f_i) = (\hat{p}_j, \mathbf{e}_j) \quad \text{for } nn(m, [x_1, .., x_k], y_j) :: r(x_1, .., x_k, y_j) \text{ a ground nAD head} \tag{16}$$

Where y_j is the j-th domain element, $\hat{p}_j$, is the j-th element of the output of the neural network m evaluated on input $[x_1, .., x_k]$. The label for a neural fact is defined as:

$$L(f_i) = (\hat{p}, \mathbf{e}_j) \quad \text{for } nn(m, [x_1, .., x_k]) :: r(x_1, .., x_k) \text{ a ground neural fact} \tag{17}$$

where $\hat{p}$ is the output of the neural network m evaluated on input $[x_1, .., x_k]$. Since the first element of the tuple for nADs and neural facts is the evaluation of the neural networks as in Section 4.2, this change remains semantically equivalent.

Example 5
To demonstrate the learning pipeline (Figure 5), we will apply it on the MNIST addition example show in Section 4.2 with a small extension: some of the labels have been corrupted and are picked randomly from a uniform distribution over $[0, 18]$. The goal is to also learn the fraction of noisy examples. The DeepProbLog program is given in Figure 4a. Grounding on the query `addition(` `,` `, 1)` results in the ground DeepProbLog program shown in Figure 4b. The arithmetic circuit corresponding to the ground program is shown in Figure 4c. As can be seen, the neural networks already have a confident prediction for both images (being 0 and 1 respectively). The top right shows how the different partial derivatives that are calculated: one with respect to the noisy parameter, ten for the evaluation of the neural network on input a and ten for the evaluation on input b.

```
nn(classifier, [X], Y, [0 .. 9]) :: digit(X,Y).
t(0.2) :: noisy.

1/19 :: uniform(X,Y,0) ; ... ; 1/19 :: uniform(X,Y,18).

addition(X,Y,Z) :- noisy, uniform(X,Y,Z).
addition(X,Y,Z) :- \+noisy, digit(X,N1), digit(Y,N2), Z is N1+N2.
```

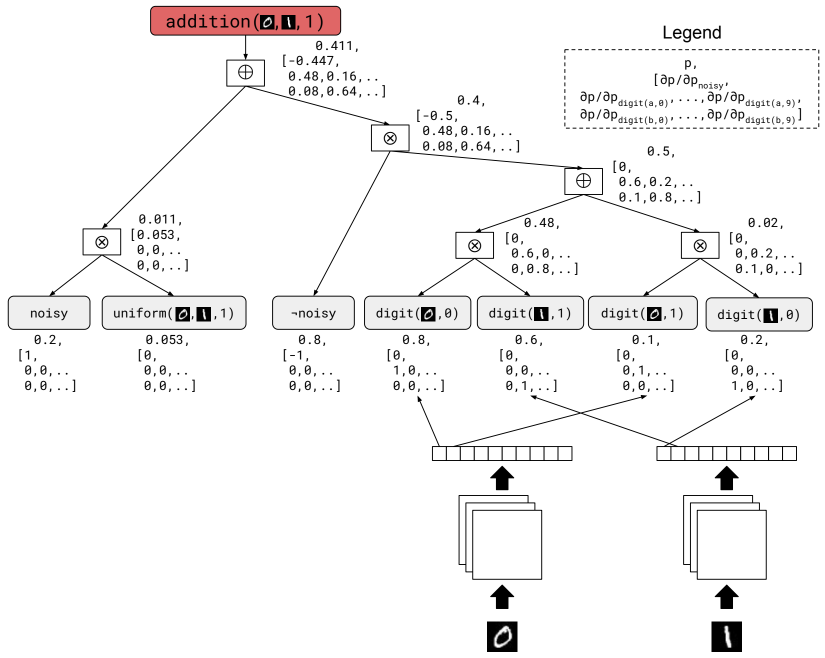
(a) The DeepProbLog program.

```
nn(classifier,[ ],0)::digit( ,0); nn(classifier,[ ],1)::digit( ,1).
nn(classifier,[ ],0)::digit( ,0); nn(classifier,[ ],1)::digit( ,1).
t(0.2)::noisy.

1/19::uniform( , ,1).
addition( , ,1) :- noisy, uniform( , ,1).

addition( , ,1) :- \+noisy, digit( ,0), digit( ,1).
addition( , ,1) :- \+noisy, digit( ,1), digit( ,0).
```

(b) The ground DeepProbLog program.



(c) The AC for query addition(, , 1).

Figure 4: Parameter learning in DeepProbLog. (Example 5)

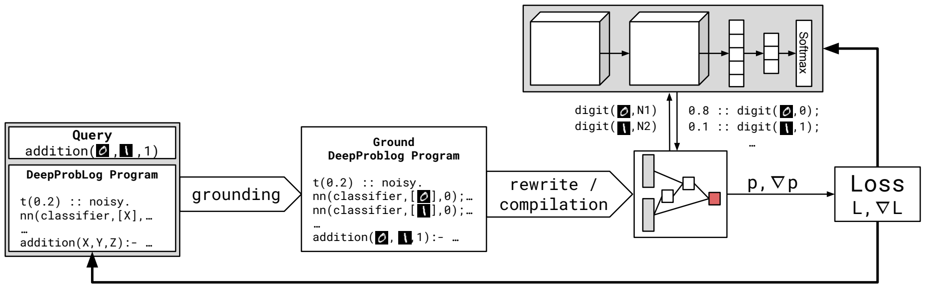


Figure 5: The learning pipeline.

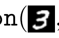
6. Experimental Evaluation

We perform four sets of experiments to demonstrate that DeepProbLog supports (i) logical reasoning and deep learning; (ii) program induction; (iii) probabilistic inference and combined probabilistic and deep learning; and (iv) natural language reasoning and embeddings

We provide implementation details at the end of this section and list all programs in Appendix A.

6.1. Logical reasoning and deep learning

To show that DeepProbLog supports both logical reasoning and deep learning, we extend the classic learning task on the MNIST dataset [**?**] to four more complex problems that require reasoning:

T1: `addition(``,` `,8)`

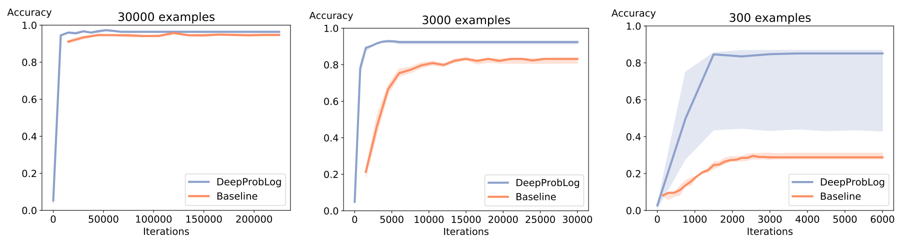
Instead of using labeled single digits, we train on pairs of images, labeled with the sum of the individual labels. This is the same as Example 3. The DeepProbLog program consists of the clause

$$\texttt{addition(X,Y,Z)}{:}{-}\texttt{digit(X,X2)},\texttt{digit(Y,Y2)},\texttt{Z is X2}+\texttt{Y2}$$

and a neural AD for the `digit/2` predicate, which classifies an MNIST image. We compare to a CNN baseline [3] classifying the two images into the 19 possible sums.

Results. Figure 6 shows the learning curves for the baseline (orange) and DeepProbLog (blue) on the single-digit addition. We evaluated on 3 levels of data availability: 30 000 examples, 3 000 and 300 examples. As can be seen in the figures, DeepProbLog converges faster and achieves a higher accuracy than the baseline. In the case for N = 30 000 (Figure 6a), the difference between the baseline and DeepProbLog is significant, but not immense. However, for N = 3000 and especially N = 300, the difference

[3]We'd like to thank Paolo Frasconi for the interesting discussion and idea for a new baseline.



(a) 30 000 examples (b) 3 000 examples (c) 300 examples

Figure 6: MNIST Single-Digit Addition (**T1**). The graphs show the accuracy on the validation set during training for different training set sizes.

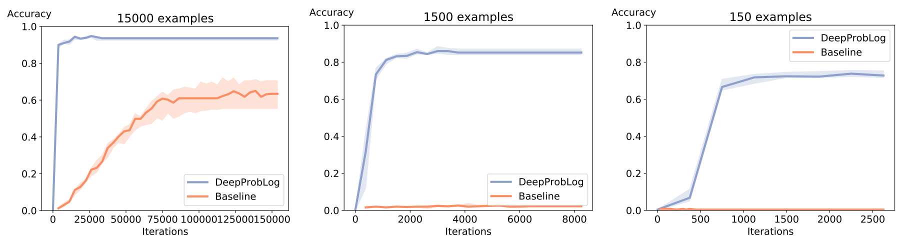
	Number of training examples		
Model	30 000	3 000	300
Baseline	93.46 ± 0.49	78.32 ± 2.14	23.64 ± 1.75
DeepProbLog	97.20 ± 0.45	92.18 ± 1.57	67.19 ± 25.05

Table 1: The accuracy on the test set for **T1**.

becomes more apparent.
The reason behind this disparity is that the baseline needs to learn making a decision for the combined input digits (and there are a 100 different sums possible), whereas the DeepProbLog's neural predicate only needs to recognize individual digits (with only 10 possibilities). Table 1 shows the average accuracy on the test set for the different models for different training set sizes.

T2: `addition([` `,` `],[` `,` `],63)`
The input consists of two lists of images, each element being a digit. Each list represents a multi-digit number. The label is the sum of the two numbers. The neural predicate remains the same. Learning the new predicate requires only a small change in the logic program. Because the CNN baseline cannot handle numbers of varying size, we fixed the size of the input to two-digit numbers.

Results. First, we perform an experiment where we take the neural network trained in **T1** and use it in this model without any further training. Evaluating it on the same test set, we achieve an accuracy that is not significantly different from training on the full dataset of **T2**. This demonstrates that the approach used in DeepProbLog causes it to generalize well beyond training data. Figure 7 shows the learning curves for the baseline (orange) and DeepProbLog (blue) on the multi-digit addition. DeepProbLog achieves a somewhat lower accuracy compared to the single digit problem due to the compounding effect of the classification error on



(a) 15 000 examples (b) 1 500 examples (c) 150 examples

Figure 7: MNIST Multi-Digit Addition (**T2**). The graphs show the accuracy on the validation set during training for different training set sizes.

	Number of training examples			
Model	15 000	1 500	150	**T1** (30 000)
Baseline	60.85 ± 9.77	1.34 ± 0.53	0.80 ± 0.14	–
DeepProbLog	95.16 ± 1.70	87.21 ± 1.92	72.73 ± 3.03	93.36 ± 1.18

Table 2: The accuracy on the test set for **T2**.

the individual digits, but the model generalizes well. The baseline fails to learn from few examples (150 and 1 500). It is able to learn with 15 000 examples, but converges very slowly. Table 2 shows the average accuracy on the test set for the different models for different training set sizes.

T3: `addition(` , , `)`
The input consists of 3 MNIST images such that the last is the sum of the first two. This task demonstrates potential pitfalls of only providing supervision on the logic level. Namely, without any regularization, the neural network quickly learns to predict 0 for all digits, i.e., the model collapses to always predicting $0 + 0 = 0$, as it is a valid logical solution. To avoid this, we add a regularisation term based on entropy maximization (Equation 18, Section 6.5). The intuition behind this regularisation term is that it penalizes mode collapse by requiring the entropy of the average output distribution per batch to be high. As such, this term encourages exploration, but is only necessary to start the training of the neural networks. If they are sufficiently trained, this term can be dropped. This additional regularization loss is multiplied by a factor λ and added to the cross-entropy loss. We run the experiment for different values of λ.

Results. Figure 8 shows the accuracy of the neural predicate on classifying single digits for different levels of the regularization parameter. As can be seen, for $\lambda = 2$, the neural predicate converges on the trivial solution. For $\lambda = 4$, the neural predicate sometimes converges on the correct solution, but can also converge on the wrong solution. For $\lambda = 8$, the neural network

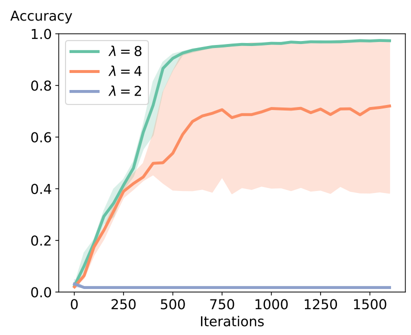


Figure 8: The accuracy on the MNIST test set for individual digits while training on (**T3**).

	Fraction of noisy labels					
	0.0	0.2	0.4	0.6	0.8	1.0
Baseline	93.46	87.85	82.49	52.67	8.79	5.87
DeepProbLog	97.20	95.78	94.50	92.90	46.42	0.88
DeepProbLog w/ explicit noise	96.64	95.96	95.58	94.12	73.22	2.92
`noisy` parameter	0.000	0.212	0.415	0.618	0.803	0.985

Table 3: The accuracy on the test set for **T4**.

.

consistently converges on the correct solution.

T4: `addition(` , , `14)`

This experiment is the example shown in Figure 4. It is the same as **T1**, but with noise introduced in the labels. Namely, a fraction of the labels is replaced by a uniformly selected number between 0 and 18. We compare three models: the CNN baseline from **T1**, the DeepProbLog model from **T1**, and a DeepProbLog model where the noise is explicitly modeled as in Figure 4. We train with different levels of noise.

Results. Table 3 shows the accuracy on the test set which has no noise. The baseline is not tolerant to noisy labels, quickly dropping in accuracy as the fraction of noisy labels increases. The DeepProbLog model from **T1** is more tolerant, but also drops noticeably in accuracy as the fraction of noise goes over 0.5. Explicitly modeling the noise makes the model very noise tolerant, even retaining an accuracy of 73.2% with 80% noisy labels. The last row shows the value of the learned `noisy` parameter, stating the estimated likelihood of a noisy label. As shown in the last row of the table, this value is close to the actual fraction of noisy labels. This shows that the model is able to recognize which examples have noisy labels.

```
nn(m_result,[D1,D2,Carry],Y,[0,...,9])::result(D1,D2,Carry,Y).
nn(m_carry,[D1,D2,Carry],Y,[0,1])::carry(D1,D2,Carry,Y).

hole(I1,I2,Carry,NewCarry,Result) :-
    result(I1,I2,Carry,Result),
    carry(I1,I2,Carry,NewCarry).

add([],[],[C],C,[]).
add([H1|T1],[H2|T2],C,Carry,[Digit|Res]) :-
    add(T1,T2,C,NewCarry,Res),
    hole(H1,H2,NewCarry,Carry,Digit).

forth_addition(L1,L2,C,[Carry|Res]) :- add(L1,L2,C,Carry,Res).
```

Figure 9: The code for experiment **T5**.

6.2. Program Induction

The second set of problems demonstrates that DeepProbLog can perform program induction. We follow the program sketching [28] setting of differentiable Forth ($\partial 4$) [8], where holes in given programs need to be filled by neural networks trained on input-output examples for the entire program. As in their work, we consider three tasks: addition, sorting [29] and word algebra problems (WAPs) [30]. Although **T5** and **T6** do not involve sub-symbolic inputs, they do show the potential of neural networks to learn the behaviour needed to fill in holes in the program.

T5: `forth_addition`$([4], [8], 1, [1, 3])$
The input consists of two numbers, represented as lists of digits, and a carry. The output is the sum of the numbers and the carry. The program (Figure 9) specifies the basic addition algorithm in which we go from right to left over all digits (`add/5`), calculating the sum of two digits and taking the carry over to the next pair. The hole in this program (`hole/5`) corresponds to calculating the resulting digit (`result/4`) and carry (`carry/4`), given two digits and the previous carry.

Results. The results are shown in Table 4. Similarly to $\partial 4$, DeepProbLog achieves 100% on all training sizes.

T6: `forth_sort`$([8, 2, 4], [2, 4, 8])$
The input consists of a list of numbers, and the output is the sorted list. The program implements bubble sort, but leaves open what to do on each step in a bubble (i.e. whether to swap or not: `swap/2`).

Results. The results are shown in Table 5. Similarly to $\partial 4$, DeepProbLog achieves 100% on training sizes 2 and 3. However, whereas $\partial 4$ fails to converge on training sizes larger than 3, DeepProbLog stills achieves 100%

		Training length		
	Test length	2	4	8
∂4 [8]	8	100.0	100.0	100.0
	64	100.0	100.0	100.0
DeepProbLog	8	100.0	100.0	100.0
	64	100.0	100.0	100.0

Table 4: Accuracy on the addition (**T5**) problem (results for ∂4 reported by Bošnjak et al. [8]).

		Training length				
	Test length	2	3	4	5	6
∂4 [8]	8	100.0	100.0	49.22	–	–
	64	100.0	100.0	20.65	–	–
DeepProbLog	8	100.0	100.0	100.0	100.0	100.0
	64	100.0	100.0	100.0	100.0	100.0

Table 5: Accuracy on the sorting (**T6**) problem (results for ∂4 reported by Bošnjak et al. [8]).

accuracy. As Bošnjak et al. [8] mention, the failure of ∂4 is due to computational issues arising from the long program trace resulting from sorting long lists. DeepProbLog does not suffer from these issues. As shown in Table 6, DeepProbLog runs faster and scales better with increasing training length.

T7: `wap(`*'Robert has 12 books . ... How many does he have now ?'*`,12,3,1,10)`
The input to the word algebra problems (WAPs) consists of a natural language sentence describing a simple mathematical problem. These WAPs always contain three numbers, which are extracted from the string and are given as part of the input. The output is the solution to the question. Every WAP can be solved by chaining the following 4 steps: permuting the three numbers (`permute/2`), applying an operation on the first two numbers (addition, subtraction or product `operation_1/2`), potentially swapping the intermediate result and the last digit (`swap/2`), and performing a last operation (`operation_2/2`). The hole in the program is in deciding which of the alternatives should happen on each step.

Results. DeepProbLog reaches an accuracy of up to 96.5%, similar to the results for ∂4 reported by Bošnjak et al. [8] (96%).

	Training length				
	2	3	4	5	6
$\partial4$ on GPU	42 s	160 s	–	–	–
$\partial4$ on CPU	61 s	390 s	–	–	–
DeepProbLog	11 s	14 s	32 s	114 s	245 s

Table 6: Time until 100% accurate on test length 8 for the sorting (**T6**) problem.

6.3. Probabilistic programming and deep learning

In this section we introduce two experiments that show the intricacies involved in combining probabilistic logic programming and deep learning.

T8: *Coin classification and comparison*

In this experiment we train two neural networks using distant supervision. The input consists of a synthetic image containing two coins (an example is shown in Figure 10). They are either heads or tails. The image is labeled either with *same* or *different*. We train a neural network for each coin to predict either *heads* or *tails* from the entire image. Solving this task requires solving two problems. On the one hand, the neural networks have to learn to recognize and separate the two different coins; on the other hand, they also have to each classify a different coin as heads or tails. The first question we ask is whether the neural networks can recover the latent structure imposed by the logic program. We expect the two neural networks to agree on which side of the coin is heads and which is tails, however, this might be the inverse of what is generally considered heads and tails. Furthermore, we expect the two neural networks to each pick one coin to label, but which network classifies which coin will vary between runs. As such, there are four possible solutions that the neural networks can converge on. The second question we ask is how many additionally labeled examples (with both the label for same/different and heads/tails of one of the coins given) we need for the neural network to recover the desired latent representation.

Results. We ran each experiment 100 times. The fraction of runs that converged on either no solution, the expected solution or a logically equivalent solution is shown in Table 7. We see that with no additionally labeled examples, DeepProbLog does not converge on a satisfactory solution in about half of all runs. When it does converge on a solution, it converges on the *expected* solution 25% of the time, and on different solutions 75% of the time, which is conform with our expectations. We can also see that as the number of additionally labeled examples increases, DeepProbLog converges more reliably, and more on the expected solution. Starting with 10 additionally labeled examples, DeepProbLog reliably converges on the desired solution. Beyond 20 additionally labeled examples, we do not see any further improvements.

Labeled examples	Not solved	Expected solution	Other solution
0	56%	11%	33%
5	39%	40%	21%
10	7%	92%	1%
20	4%	96%	0%
50	3%	97%	0%
100	4%	96%	0%

Table 7: The fraction of runs that converged to different outcomes for the Coins experiment (**T8**).

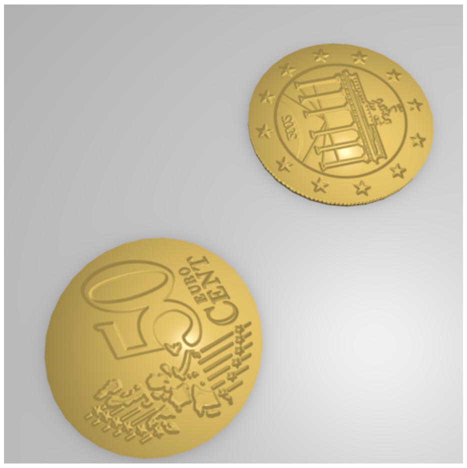


Figure 10: An example input image for the Coins (**T8**) experiment.

T9: `0.8::poker([Q♡, Q♢, A♢, K♣],loss).`

In this experiment we demonstrate that DeepProbLog can perform combined probabilistic reasoning, probabilistic learning and deep learning.

We do this by playing a simplified poker game: there are two players that are dealt two cards from several decks. There is also one community card. Each player then makes a poker hand (e.g. pair, straight, ...) with their two cards and the community card. The best combination wins.

The task is to learn to correctly predict the probability that the first player wins. To do this, the model also has to learn the distribution of the community card, which is not observed and is considered a latent variable. By modelling the task with knowledge of its structure, we are able to construct an interpretable model in which the distribution of this latent variable is made explicit. This contrasts with a neural model, which, although it might be able to perform the same regression, would not provide the user with information of this distribution.

Distribution	Jack	Queen	King	Ace
Actual	0.2	0.4	0.15	0.25
Learned	0.203 ± 0.002	0.396 ± 0.002	0.155 ± 0.003	0.246 ± 0.002

Table 8: The results for the Poker experiment (**T9**).

For simplicity, we only use the jack, queen, king and ace. We also do not consider the suits of the cards. The input consists of 4 images that show the cards dealt to the two players. Additionally, every example is labeled with the chance that the game is won, lost or ended in a draw, e.g.:

$$0.8 :: \texttt{poker}([\texttt{Q}\heartsuit, \texttt{Q}\diamondsuit, \texttt{A}\diamondsuit, \texttt{K}\clubsuit], \texttt{loss})$$

We expect DeepProbLog to:

- train the neural network to recognize the four cards
- reason probabilistically about the non-observed card
- learn the distribution of the non-observed community card

To make DeepProbLog converge more reliably, we add some examples with additional supervision. Namely, in 10% of the examples we additionally specify the community card, i.e.

$$\texttt{poker}([\texttt{Q}\heartsuit, \texttt{Q}\diamondsuit, \texttt{A}\diamondsuit, \texttt{K}\clubsuit], \texttt{A}\diamondsuit, \texttt{loss}).$$

This also showcases one of the strengths of DeepProbLog, namely, it can make use of examples that have different levels of observability. The loss function used in this experiment is the MSE between the predicted and target probabilities.

Results. We ran the experiment 10 times. Out of these 10 runs, 4 did not converge on the correct solution. The average values of the learned parameters for the remaining 6 runs are shown in Table 8. As can be seen, DeepProbLog is able to correctly learn the probabilistic parameters. In these 6 runs, the neural network also correctly learned to classify all card types, achieving a 100% accuracy. The other runs did not converge because some of the classes were permuted (i.e., queens predicted as aces and vice versa) or multiple classes mapped onto the same one (queens and kings were both predicted as kings).

6.4. Embeddings and Natural Language Reasoning

T10: `successor(` , , 2`)`

In this experiment, we explore representing embeddings directly in DeepProbLog programs, again using the MNIST images as data instances. A straightforward way to obtain image embeddings, is by optimizing a clustering objective. That is, we minimize the distance between the embeddings of images of the same number, while maximizing the distance between the embedding of images of different numbers. As a similarity metric, the radial basis function $\varphi(x, y)$ (RBF) can be used, also used in the neural theorem prover [9], which is defined as

$$\varphi(x, y) = e^{-||x-y||_2}$$

This can be included in DeepProbLog as a predicate `rbf/2` that succeeds with probability $p = \varphi(x, y)$ where x and y are embeddings that are the arguments of the predicate. The `rbf/2` predicate provides DeepProbLog with the power of soft unification from the Neural Theorem Prover (NTP) [9]. With soft unification, literals can be unified if they are similar (e.g. `grandPa(abe,bart)` can be unified with `grandFather(abe,bart)`), which was proven to be a powerful concept in Rocktäschel and Riedel [9]. By implementing it explicitly in DeepProbLog, we have additional control over the soft unification, as we can decide to only apply it to parts of the logic program, whereas in the NTP, it is applied to the entire program. For example, here, soft unification is only applied to MNIST images, and the remainder of the program uses standard reasoning and unification.

The corresponding DeepProbLog program is shown in Figure 11a, in which `cnn_encode/2` unifies the second argument with the embedding of the first argument by using a convolutional neural network, and `rbf/2` leads to clustering in embedding space.

As a proof of concept of the wider possibilities of implicitly training embeddings through DeepProbLog, we extend the clustering objective towards inducing an order relation in embedding space. The successor relationship between two MNIST images is defined as `successor(I1,I2,R)` where `I1` and `I2` are MNIST images, and `R` is the difference of the digits represented by the two images (e.g. `successor(` , , -2)).

We model the successor relationship as a translation (cfr. TransE [31]). That is, we learn an embedding r_s of the successor relationship in this embedding space such that we minimize the distance between $e_x + nr_s$ and e_y, where e_x and e_y are the embeddings of the two images and n is the difference of the image labels. If $n = 0$, then the `successor/3` relationship (Figure 11b) becomes identical to the `similarity/2` relationship (Figure 11a). We also specify that r_s should not be zero to avoid collapsing on a trivial solution. Note that this constraint on the size of the embedding can be seen as a form of regularization defined in the logic. We define a predicate `embed/2` that unifies the second argument with a learnable embedding for the first argument. Here, `rbf/2`, `cnn_encode/2` and `embed/2` are as described above, `mul/3` is the product between a scalar and a vector and `add/3` an element-wise addition. These create the mathematical definition of the successor relationship. The number 0 is used as a zero vector of the same dimension as the embeddings.

We generate the dataset for this experiment by labeling each pair of subsequent images in the original MNIST dataset with the correct successor label, resulting in 30000 instances for training, and 5000 for testing. Although the embedding dimensionality can be chosen freely, we know that for the successor relation a single dimension should suffice. For the benefit of visual interpretation, we therefore use an embedding size of 1.

```
similar(I1,I2) :-
    %Encode images I1 and I2 into E1 and E2
    cnn_encode(I1,E1), cnn_encode(I2,E2),
    rbf(E1,E2).
```

(a) Similarity between MNIST images

```
successor(I1,I2,N) :-
    %Encode images I1 and I2 into E1 and E2
    cnn_encode(I1,E1), cnn_encode(I2,E2),
    %Embed the successor relation into embedding S
    embed(successor, S),
    %The relation should not be trivial (i.e. 0)
    \+rbf(S,0).
    %E = E1 + N*S should be similar to E2
    mul(S,N,S2), add(E1,S2,E),
    rbf(E,E2).
```

(b) The code in the MNIST successor task (**T10**)

Figure 11: Embeddings in DeepProbLog

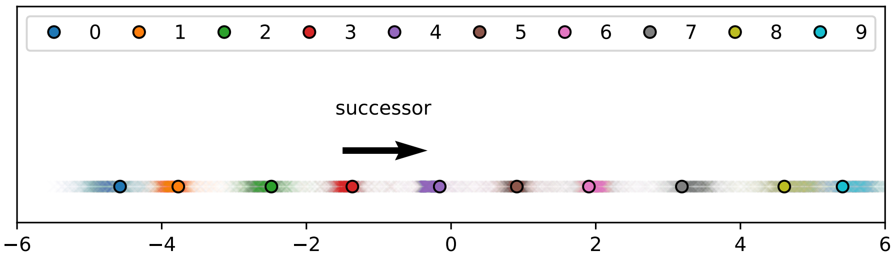


Figure 12: The final embedding space for **T10** after training the encoder and successor relation embedding. Individual MNIST image embedding are shown in crosses, with the mean for each label as a solid dot. The direction of the successor relation is marked by the arrow.

Result. Figure 12 shows the embeddings of the MNIST test set (crosses) and the mean of each of these images, grouped by label (solid dots). It shows that the model has learned to embed images into clusters corresponding to the labels of these images, which are positioned sequentially along the embedding dimension. As can be visually inspected, the cluster centers can be (approximately) linearly mapped to the actual image labels. This result confirms that, with the inclusion of embeddings, DeepProbLog is able to learn structured embedding spaces and perform soft unification (which can be considered one of the key strengths of the NTP[9]).

T11: *Compositional language understanding and reasoning*

The final experiment further demonstrates DeepProbLog's ability to manipulate distributed representations (embeddings), effectively leading to a wider interface between the neural and logic components. The aim is to illustrate the framework's potential for real-world applications beyond the toy tasks in the previous examples. In particular, this experiment explores to what extent the framework allows reasoning over natural language sentences. For this purpose, we will use the Compositional Language Understanding with Text-based Relational Reasoning (CLUTRR) dataset [32].

The CLUTRR dataset consists of short natural language texts of several sentences containing facts about family relations between people. The goal is to infer the relation between two people mentioned in the text. These relations are not directly mentioned, but have to be inferred from other relations in the text. The task in this experiment is thus to train the neural networks to perform relation extraction. That is, we have to extract logical facts from natural language sentences, as shown in Figure 13. In contrast to the original setup of the CLUTRR experiment, we do not solve the rule learning aspect. Instead, we follow the idea that is central to DeepProbLog that we can provide the model with background knowledge when it is available, and only learn those parts that we cannot define easily in logic. In this case, we provide the family relations shown in Listing 1, and focus on the relation extraction task from the sentences. Thus, one can expect good generalization capabilities in the model, if the individual relations are extracted correctly.

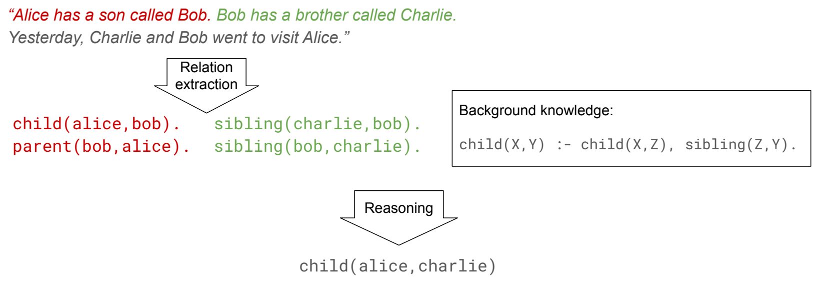
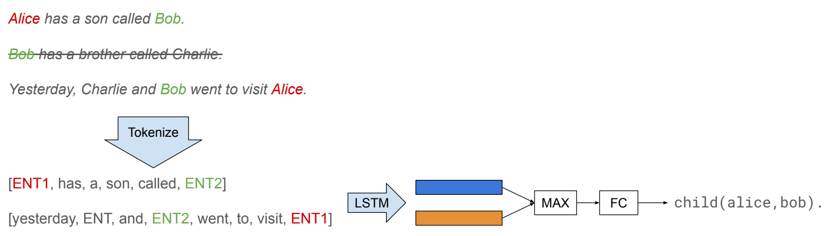

"Alice has a son called Bob. Bob has a brother called Charlie.
Yesterday, Charlie and Bob went to visit Alice."
Relation extraction
child(alice,bob).
parent(bob,alice).
sibling(charlie,bob).
sibling(bob,charlie).
Background knowledge:
child(X,Y) :- child(X,Z), sibling(Z,Y).
Reasoning
child(alice,charlie)

Figure 13: An example inference task from the CLUTRR dataset.

Embeddings. In this experiment, we further explore the use of embeddings in DeepProbLog. A simplified version of the code can be seen in Listing 2.



Alice has a son called Bob.
~~Bob has a brother called Charlie.~~
Yesterday, Charlie and Bob went to visit Alice.
Tokenize
[ENT1, has, a, son, called, ENT2]
[yesterday, ENT, and, ENT2, went, to, visit, ENT1]
LSTM
MAX
FC
child(alice,bob).

Figure 14: The implementation of the neural predicate for relation extraction, illustrated on the relation between Alice and Bob.

There are two ways to prove relations between entities: reasoning using background knowledge, and extracting relations from the text. The relation extraction happens in the `neural_relation/4` predicate. First, we select all relevant sentences for the given entities, that is, the sentences that contain both entities. Then we embed these using the `embed/3` predicate, which embeds a single sentence (e.g., encodes the sequence of tokens into a vector) using a gated recurrent unit encoder (GRU [33], a recurrent architecture similar to the LSTM) on top of randomly initialized token embeddings. The final state of the GRU for each sentence is used as the embedding. Afterwards, the `max_tensor/2` predicate performs a max pooling over the sequence of embeddings. The relations are predicted through a logistic regression classifier (a single linear layer with a softmax activation function) on top of the final text representation (`nn_rel/2`). A diagrammatic overview is given in Figure 14. A similar process happens for the `neural_gender/3` predicate that predicts the gender for a single entity. The benefit of being able to manipulate embeddings inside of the logic itself is that the neural model itself becomes simpler and the logic becomes more expressive. This shows that DeepProbLog is more than a simple pipeline where logical reasoning follows neural network evaluation. Instead, DeepProbLog is an interface between the logic and neural sides.

The model as it is currently shown relies on the fact that all rules are present. However, if this were not the case, we could replace the gaps in the knowledge by a neural counterpart (i.e., a neural component that tries to directly predict the missing relation directly from the entire text).

Experiments. The first experiment on the CLUTRR dataset investigates the innate ability of DeepProbLog to generalize. There are several facts that connect the two query entities. The number of facts needed to connect the two will be called k (e.g. in the example in Figure 13, $k = 2$). In particular, we show how the model generalizes to longer texts and an increasing number of reasoning steps than seen during training. To do this, it is trained for smaller k (e.g. $k < 4$) and evaluated on larger k (e.g. $k < 10$). The model is trained until no further improvements occur for two consecutive steps on the accuracy on a held-out set with $k = 2$. We hypothesize that the neural network mainly needs to learn to recognize elementary relations, whereas the logic component will tackle the reasoning required to deal with larger k values.

The second experiment investigates the robustness of the reasoning component. To test this, the CLUTRR dataset adds unnecessary sentences and facts to the text. These include supporting facts, irrelevant facts and disconnected facts. Supporting facts are facts that provide an alternate explanation for (part of) the relation between the two query entities. Irrelevant facts are facts that mention relevant entities, but cannot be used to explain the query relation. Disconnected facts are about entities and relations that are not related at all to the query or other entities. In this experiment, we train on clean data ($k = 2, 3$) and test how well it performs on the other data ($k = 3$). We also see what the impact of training on the supporting, irrelevant or disconnected data has on the accuracy. We trained until we saw no further improvement for 2 steps on the accuracy on the test set for $k = 2$.

```
grandchild(X,Y) :- child(X,Z), child(Z,Y).
grandchild(X,Y) :- so(X,Z), grandchild(Z,Y).
grandchild(X,Y) :- grandchild(X,Z), sibling(Z,Y).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
grandparent(X,Y) :- sibling(X,Z), grandparent(Z,Y).
child(X,Y) :- child(X,Z), sibling(Z,Y).
child(X,Y) :- so(X,Z), child(Z,Y).
parent(X,Y) :- sibling(X,Z), parent(Z,Y).
parent(X,Y) :- child(X,Z), grandparent(Z,Y).
sibling(X,Y) :- child(X,Z), uncle(Z,Y).
sibling(X,Y) :- parent(X,Z), child(Z,Y).
sibling(X,Y) :- sibling(X,Z), sibling(Z,Y).
child_in_law(X,Y) :- child(X,Z),so(Z,Y).
parent_in_law(X,Y) :- so(X,Z), parent(Z,Y).
nephew(X,Y) :- sibling(X,Z), child(Z,Y).
uncle(X,Y) :- parent(X,Z), sibling(Z,Y).
```

Listing 1: The background knowledge used in **T11**

Results. Figure 15 shows the result for the systematic generalization experiment. The DeepProbLog model shows strong generalization capabilities for all three training datasets. This is in contrast with the results as shown in Sinha et al. [32], where most methods have a steady decline in accuracy beyond the dataset. This result shows the benefit of performing logical reasoning with background knowledge. If the relation extraction is correct, DeepProbLog has no problem generalizing to larger values of k.

Table 9 shows the result for the robust reasoning experiment. It shows that the DeepProbLog model is indeed robust. This is in line with its observed

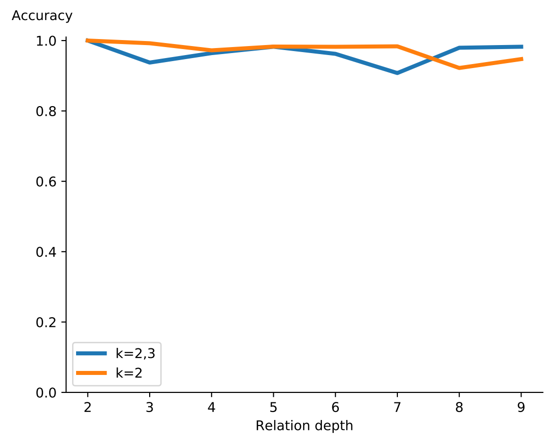


Figure 15: Systematic generalization on the CLUTRR task for different training sets. It shows the accuracy (y-axis) with respect to the number of relations needed to discover the query relation (x-axis).

Training	Testing	Accuracy
Clean	Clean	1.0
	Supporting	0.99
	Irrelevant	0.98
	Disconnected	0.99
Supporting	Supporting	1.0
Irrelevant	Irrelevant	1.0
Disconnected	Disconnected	0.94

Table 9: Robust reasoning on the CLUTRR task for different types of noise.

generalization capabilities. As long as the individual relations are predicted correctly, the reasoning is also correct.

6.5. Implementation details

For the implementation we integrate ProbLog2 [34] with PyTorch [35]. All programs are listed in Appendix A. In all experiments except **T9**, we optimize the cross-entropy loss between the predicted and target query probabilities, as we found that this works better to learn the probabilistic parameters. In experiment **T9** we optimize the MSE between the predicted and target query probabilities as this proved to converge more reliably. This setting was previously used in Gutmann et al. [25]. We use Adam [**?**] optimization for the neural networks, and SGD for the logic parameters. For **T9**, we add random rotations (max 10 degrees) and shift the colors in the HSV by up to 5% to the

```
%Neural network that predicts relation for the given embedding
nn(net1,[E],R,[child,child_in_law,...]) :: nn_rel(E,R).
%Neural network that predicts the gender for a given embedding
nn(net2,[E],G,[male,female]) :: nn_gender(E,G).

%Predicate that proves the relation R between X and Y
query_rel(T,X,R,Y):-
    %Determine the gender G for entity Y in text T
    neural_gender(T,Y,G),
    %Turn the gendered relation R into the un-gendered relation R2
    gender_rel(R2,G,R),
    %Prove relation R2 between entities X and Y
    call(R2,T,X,Y).

%Predicate that predicts the gender G for entity E
neural_gender(T,E,G) :-
    %E is an entity mentioned in the text T
    entity(T,E),
    %Only include relevant sentences (i.e. E is mentioned)
    include(mentioned(E), T, Relevant),
    %Map each relevant sentence onto its embedding
    maplist(embed([E]),Mentioned,Embedded),
    %Take the element-wise max of all embeddings
    max_tensor(Embedded,Max),
    %Classify the resulting embedding
    nn_gender(Max,G).

%Predicate that predicts the relation R between entities X and Y
neural_relation(T,X,R,Y) :-
    %X and Y are entities mentioned in the text T
    entity(Sentences,X),
    entity(Sentences,Y),
    X \= Y,
    %Only include relevant sentences (i.e. X and Y are mentioned)
    include((mentioned(X),mentioned(Y)),Sentences,Mentioned),
    %Map each relevant sentence onto its embedding
    maplist(embed([Y,X]),Mentioned,Embedded),
    %Take the element-wise max of all embeddings
    max_tensor(Embedded,Max),
     %Classify the resulting embedding
    nn_rel(Max,R).


child(T,X,Y) :- neural(T,X,child,Y).
child_in_law(T,X,Y) :- neural(T,X,child_in_law,Y).
...

grandchild(T,X,Y) :- X\=Y, child(T,X,Z), child(T,Z,Y).
grandchild(T,X,Y) :- X\=Y, so(T,X,Z), grandchild(T,Z,Y).
grandchild(T,X,Y) :- X\=Y, grandchild(T,X,Z), sibling(T,Z,Y).

grandparent(T,X,Y) :- X\=Y, parent(T,X,Z), parent(T,Z,Y).
grandparent(T,X,Y) :- X\=Y, sibling(T,X,Z), grandparent(T,Z,Y).
...
```

Listing 2: Part of the code for the CLUTRR experiment (**T11**)

Number length	Grounding (s)	Compilation (s)	Evaluation (s)
1	0.018 ± 0.00	0.004 ± 0.00	0.017 ± 0.01
2	1.39 ± 0.25	0.035 ± 0.02	0.158 ± 0.07
3	114 ± 5.23	0.140 ± 0.04	0.567 ± 0.08

Table 10: Mean and standard deviation of the duration in seconds of different inference steps for 100 queries of the MNIST addition example for increasing number lengths (k).

images of the cards.
In **T11** we use negative mining by proving the query for all possible relations and using the correct label as a positive example and all other labels as negative examples. This prevents the model to predict incorrect labels for irrelevant facts. We did not include the templates the authors collected using Amazon Mechanical Turk and have generated the same datasets as in the original paper, using the code provided by the authors.

The neural network architectures are summarized in Table 13. *Conv(o,k)* denotes a convolutional layer with o output channels and kernel size k. *Lin(n)* denotes a fully connected layer of size n. *(Bi)GRU(h)* denotes a single-layer (bi-directional) GRU with a hidden size h. *(layer)*×2 means that there are two identical layers in parallel that are concatenated. A layer in bold means it is followed by a ReLU activation function. All neural networks end with a Softmax layer, unless otherwise specified. The hyperparameters used in the experiments are shown in Table 11. The sizes of the datasets used are specified in Table 12.

The regularisation term used in **T3** is calculated per network and per batch on the average of the neural network output. It is calculated as

$$1.0 - H_n\left(\frac{1}{N}\sum_{i=1}^{n} P_i\right) \tag{18}$$

where P_i is the i-th output of the neural network and H_n is the n-ary entropy (i.e. entropy using the base-n logarithm).

6.6. Computation time

Due to the nature of the exact inference used in DeepProbLog, which it inherits from aProbLog [26], the grounding and compilation steps can become expensive as the problem size grows. Table 10 shows the mean time and standard deviation for the different steps in inference for the MNIST addition task for numbers of increasing length. As we can see, the time spent on grounding increases steeply and quickly becomes the dominant factor.

It is important to note that when we evaluate an example a second time, the structure of the AC, which is determined by the grounding and compilation, remains the same. Only the learned probabilities in the nAD change. We make use of this to improve the performance by caching the arithmetic circuits so that we only have to perform the (potentially expensive) grounding and compilation

Task	Batch size	Learning rate	Parameter learning rate	Regularisation
T1-T3	2	1e-3		
T4	2	1e-3	1e-3	2, 4, 8
T5	50	0.02		
T6	16	1.0		
T7	100	0.005		
T8	5	1e-4		0.25
T9	50	1e-4	1e-3	0.5
T10	4	1e-3		
T11	2	1e-3		

Table 11: Overview of the hyperparameters used in the experiments.

Task	Training set	Validation Set	Test set
T1	29 500, 3 000, 300	500	5 000
T2	14750, 1 500, 150	250	2 500
T3	16 000	2 000	3 000
T4	29 500	500	5 000
T5	512	256	1 024
T6	256	32	32
T7	300	100	200
T8	100	–	20
T9	500	–	25
T10	30 000	–	500
T11	1000	–	50

Table 12: Overview of the sizes of the datasets used in the experiments.

steps once. During evaluation, we only re-evaluate the neural networks and evaluate the AC with the updated probabilities.

Note that this optimization can also be applied to, for example, the queries `addition(` , `, 8)` and `addition(` , `, 8)`, as both have the same structure. To do this, we introduce placeholder constants and change both queries to the single query `addition(a, b, 8)`, which reduces all queries in **T1** to 19 unique queries, one for each different label. During the evaluation of the neural networks, we replace the constants with the correct input and use the resulting probabilities in the cached ACs. We apply these optimizations to all experiments.

Task	Network	Architecture
T1-T4	digit/2	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(10)
T1,T3	baseline	MNISTConv×2, **Lin(120)**, **Lin(84)**, Lin(19)
T2	baseline	(MNISTConv×2, **Lin(100)**)×2,**Lin(128)**, Lin(199)
T5	result/4	Lin(50), TanH, Lin(10)
	carry/4	Lin(10), TanH, Lin(2)
T6	swap/3	**Lin(20)**, Lin(10)
T7	RNN	Embedding(256), BiGRU(512), Dropout(0.5)*
	perm/2	Lin(6)
	op1/2	Lin(4)
	swap/2	Lin(2)
	op2/2	Lin(4)
T8	coin1/2	AlexNetConv, **Lin(100)**, Lin(2)
	coin2/2	AlexNetConv, **Lin(100)**, Lin(2)
T9	rank/2	AlexNetConv, **Lin(100)**, Lin(4)
T10	embed/2	Embed(2)
	cnn_embed/2	MNISTConv, **Lin(120)**, **Lin(84)**, Lin(2)*
T11	embed/2	Embed(16), GRU(16)
	nn_rel/2	Lin(11)
	nn_gender/2	Lin(2)

MNISTConv: Conv(6,5), **MP(2,2)**, Conv(16,5), **MP(2,2)***
AlexNetConv: **Conv(64, 11, 2,2)**, MP(3,2), **Conv(192, 5, 2)**, MP(3,2),
Conv(384, 3, 1), **Conv(256, 3, 1)**, **Conv(256, 3, 1)**, MP(3,2)*
* Does not end with a Softmax layer.

Table 13: Overview of the neural network architectures used in the experiments.

7. Related Work

The integration of logical reasoning and neural networks is a field with a long tradition that currently enjoys a lot of interest. Most of the work on combining neural networks and logical reasoning comes from the *neuro-symbolic reasoning* literature [7, 36]. These approaches typically focus on approximating logical reasoning with neural networks by encoding logical terms in Euclidean space. However, these approaches have to limit their expressivity to non-recursive and acyclic logic programs [37] and do neither support probabilistic reasoning nor perception.

The renewed interest in neuro-symbolic systems has introduced many new perspectives that make the individual approaches difficult to compare. The remainder of this section provides an overview of the main ideas present in the field. Before we do that, we identify four dimensions that distinguish DeepProbLog from other systems. Moreover, we argue that these dimensions are important for understanding the capabilities and limitations of neuro-symbolic systems in general. These dimensions (Table 14) are:

1. **Logical, neural and probabilistic components.** DeepProbLog has a probabilistic, a logical, and a neural component, whereas most other methods only have two. Though this may appear as a complication, our work shows that it greatly simplifies the integration of neural networks with logic. The probabilistic framework provides the semantics of DeepProbLog programs. It also provides a clear optimization criterion, namely the probability of the training example. Real-valued probabilistic quantities are also well-suited for gradient-based training procedures, as opposed to discrete logic quantities.

2. **Logic and neural networks as special cases.** DeepProbLog has both logic and neural networks as special cases. In our opinion, further elaborated in [38], the integration should preserve the original concepts it integrates. If not, some of the power of the original methods is lost in the integration.

3. **Semantics.** DeepProbLog has well defined semantics. This is important as this allows us to specify exactly what is calculated by the DeepProbLog framework and can be given a probabilistic interpretation. This is missing from many other works where soft scores are used instead, which do not have a clear, semantical meaning.

4. **Logical or neural origin of a system.** DeepProbLog originated from probabilistic logic programming technology rather than neural network technology. Due to its origins, DeepProbLog offers three benefits: (i) DeepProbLog does neither sacrifice the expressivity of logic nor neural networks; (ii) DeepProbLog incorporates probabilistic inference 'for free', alongside logical reasoning and neural predictions; (iii) DeepProbLog is a Turing-complete programming language.

The three most prominent neuro-symbolic research lines are (1) pushing the logic as regularisation, (2) templating neural networks, and (3) neural program induction. Before we outline these prominent lines , we cover the two approaches that relate the most to DeepProbLog: grounding-specific MLNs [50] and Deep Probabilistic Logic [51].

7.1. The most comparable approaches

Grounding-specific MLNs [50] extend Markov logic networks (MLN) [52], another popular StarAI framework, so that the weight of a formula can differ for different groundings of the formula. The weights can be determined by neural models, similar to how neural networks in DeepProbLog define probabilities. In contrast to ProbLog, which assigns probabilities to facts, MLNs represent uncertain knowledge with tuples of weighted formulas, $\{(w_i, F_i)\}$. In standard MLNs, every grounding of a formula F_i has the same associated weight w_i; in grounding-specific MLNs, the weight can change depending on the constants present in the grounding.

Method	Components	Special cases	Semantics	Origin
	(P)robabilistic (L)ogic (N)eural	(L)ogic (N)eural	(P)robabilistic (F)uzzy (O)thers	(L)ogic (N)eural
[39]	L + N	-	F	-
FSL[40]	L + N	-	O	-
ASR [41]	L + N	-	O	-
SBR[42]	L + N	-	F	-
LTN[43]	L + N	-	F	N
SL[44]	P+L+N	N	P	L
NTP [9]	L+N	-	O	L+N
NLProlog[45]	L+N	-	O	L+N
TensorLog[10]	L+N	L	O	L
NMN[46]	N	-	-	-
δ4 [8]	N	(L+)N	O	-
δILP[47]	L+N	L	O	L
LRNN[48]	L+N	-	F	
RelNN[49]	L+P	L	P	L

Table 14: An overview of related work across the four discussed concepts.

Deep Probabilistic Logic (DPL) [51] is a general framework for indirect supervision of deep learning models: it treats the predictions of deep models as latent variables that probabilistic logic models operate on, imposing the indirect supervision in terms of (soft) constraints. DeepProbLog is a more general framework, as the same setting that is shown in DPL, namely that of distant supervision of neural networks, is used here in most experiments. However, as we have shown in **T10** and **T11**, DeepProbLog has a wider interface that allows for bi-directional flow of information (i.e, the neural networks can influence the logic.

7.2. Logic as regularisation

The main idea behind this line of research is to *include the logic as a regularizer of the neural network*, during the optimization or the learning of the embeddings. This formalism encodes the logic into the weights of the network, so that even when the logic is not explicitly present, the model still follows the characteristics of the logic. [39, 40, 41, 42, 43, 44] all center around including logical background knowledge as a regularizer during training. Rocktäschel et al. [39] inject background knowledge into a matrix factorization model for relation extraction, by adding differentiable loss terms for propositionalized first-order rules. Demeester et al. [40] propose a more efficient alternative by inducing order relations in embedding space, effectively leading to a lifted application of the rules. This is further generalized by Minervini et al. [41], who investigate injecting rules by minimizing an inconsistency loss on adversarially-generated examples. Diligenti et al. [42] use first-order logic to specify constraints on the output of the neural network. They use fuzzy logic to create a differentiable way of measuring how much the output of the neural networks violates these constraints. This term is then added as an additional loss term that acts as a regularizer. More recent work by Xu et al. [44] introduces a similar method that uses probabilistic logic instead of fuzzy logic, and is thus more similar to DeepProbLog. They also compile the formulas to an SDD for efficiency.

The major difference between DeepProbLog and these regularization approaches is that DeepProbLog maintains explicit logical knowledge. In the other approaches, the explicit logical knowledge is lost in the regularization approaches as it is included as a penalty during training and not considered during prediction. Another major difference is in the logic employed by the systems: DeepProbLog is based on probabilistic logic programming, whereas the regularization methods rely on (fuzzy) first-order logic. This is reminiscent to the difference between ProbLog and Markov Logic [52] or PSL [53]. DeepProbLog can be used on a wider variety of tasks than these systems. Because it is also a programming language, it well suited to work with structured data, or data of varying length. For example, in the multi-digit addition example **T2**, DeepProbLog can deal with the varying input sizes naturally, while most neural methods cannot cope with this.

7.3. Templating neural networks

The second line of work uses logic as a template for constructing the architecture of neural networks. This is reminiscent of the knowledge base model construction approaches of statistical relational artificial intelligence [4].

Rocktäschel and Riedel [9] introduce a differentiable framework for theorem proving. They re-implement Prolog's theorem proving procedure (with bounded proof depth) in a differentiable manner and enhance it with learning a subsymbolic representation of the existing symbols, which are used to handle noise in data. Whereas Rocktäschel and Riedel use logic only to construct a neural network and focus on learning subsymbolic representations, DeepProbLog focuses on tight interactions between the two and parameter learning for both the neural and the logic components. In this way, DeepProbLog retains the best abilities of both worlds. Recently, Weber et al. [45] extend the notion of soft unification towards structured textual knowledge, i.e., unification can be performed between sentences, not only symbols. In contrast to Rocktäschel and Riedel [9], Weber et al. [45] retain the full ability of logical reasoning, and as such is closer to DeepProbLog, but it is specialised for NLP tasks.

Cohen et al. [10] introduce a framework to compile a tractable subset of logic programs into differentiable functions and to execute it with neural networks. It provides an alternative probabilistic logic but it has a different and less developed semantics. Furthermore, to the best of our knowledge it has not been applied to the kind of tasks tackled in the present paper. An idea similar in spirit to ours is that of Andreas et al. [46], who introduce a neural network for visual question answering composed out of smaller modules responsible for individual tasks, such as object detection. Whereas the composition of modules is determined by the linguistic structure of the questions, DeepProbLog uses probabilistic logic programs to connect the neural networks.

7.4. Neural program induction.

The third line of work focuses on learning programs from data by combining neural and symbolic approaches.

Neural execution. The first category captures a program behaviour with neural networks and therefore focuses on program execution. This includes the works such as $\partial 4$ by Bošnjak et al. [8], in which neural networks are used to fill in *holes* in a partially defined Forth program, and ∂ILP by Evans and Grefenstette [47], in which neural networks learn a program by selecting a subset of clauses to form the final program. In contrast to $\partial 4$ which uses a procedural host language, DeepProbLog uses ProbLog, and consequently Prolog, which results in native support for both logical and probabilistic reasoning. ∂ILP focuses on program induction with neural networks and not on the integration of the two approaches like DeepProbLog.

Neurally guided search. The second category includes the work on incorporating the neural components in search procedures for the symbolic program induction techniques. As these methods are not logic-based, the comparison made in Table 14 is not relevant, and is therefore omitted. The key principle these techniques employ is to perform the search over programs in a systematic symbolic way, but guide the search with a heuristic learned by a deep neural network. For instance, Kalyan et al. [54] train a neural network to predict the scores of branches during the branch-and-bound search procedure, Zhang et al. [55] train a neural network to choose which candidate program to expand next while exploiting the constraints on the input-output examples, while Ellis et al. [56] use a neural network to efficiently search over a well-designed DSL. DeepProbLog currently supports only program induction by sketching and, thus, none of these approaches are directly comparable to it.

Neural program construction. The final category involves techniques that decompose a problem into independent parts that can be individually solved by either neural or symbolic components and synchronize the individual components to solve the main problem. For instance, Yi et al. [57], Mao et al. [58] propose a neuro-symbolic approach towards visual question answering by using a neural network to generate a program computing the answer to the question and executing the program symbolically. Ellis et al. [59] generate $\LaTeX$ code from a hand-drawn sketch by using a neural network to recognize basic shapes within a sketch and symbolically inducing the program describing the sketch.

These approaches are close in spirit to DeepProbLog as they retain both the neural and the symbolic component explicitly; however they focus on their synchronization in which one components provides the input for the other, while DeepProbLog tightly integrates them so that both components inter-operate.

7.5. *Symbolic deep learning*

The success of neural deep learning has inspired several works introducing symbolic deep learning methods which, instead of representing the logical aspects in a vector space, retain the logical data representation in the latent representation. These include the symbolic versions of deep neural networks: Šourek et al. [48] treat symbolic rules expressed in first-order logic as a template for constructing a neural network, while Kazemi and Poole [49] compose a relational neural network by adding hidden layers to relational logistic regression [60]. Another research direction focuses on task-agnostic discovery of relational (symbolic) latent representations by exploiting approximate symmetries [61], a symbolic extension of the auto-encoding principle [62], or self-play [63].

8. Limitations and future work

One of the current limitations is that DeepProbLog currently only supports exact probabilistic inference. As a consequence, several steps in the inference can become prohibitively expensive as the size of the program grows. This issue is however not intrinsic to the proposed integration, but can be resolved in future work by extending DeepProbLog with approximate inference. This is similar to standard work in StarAI, where approximate inference is needed to scale up. There has already been research into approximate inference in ProbLog [64, 65] which could be adapted to the work presented here.

In the comparison with other methods, we discussed some features that were available in DeepProbLog that are not in other methods, and vice versa. One feature that stands out is the ability to use and explicitly manipulate embeddings and tensors [43, 40, 39]. Although we have shown preliminary result of the applicability of embeddings in experiments **T10-T11**, future work could look into this more extensively.

Another feature that is present in other methods is structure learning. Although we have shown some results in this (**T5-T7**), we have currently only explored the sketching approach. There are however other approaches that are worth investigating, such as inductive logic programming.

Finally, a problem that DeepProbLog faces is the problem where the desired interpretation of the learned facts does not overlap with what is actually learned. For example, in **T3**, DeepProbLog learns the trivial solution $0+0=0$, which, although logically correct, is not a desired solution. This was solved by using additional regularization. In **T8** we investigate whether the desired interpretation (i.e., that the coins get correctly classified as heads and tails) can be reliably achieved using additional supervision. Ideally, we would like to develop a more principled approach to solving these issues.

9. Conclusion

We introduced DeepProbLog, a framework where neural networks and probabilistic logic programming are integrated in a way that exploits the full expressiveness and strengths of both worlds and can be trained end-to-end based on examples. This was accomplished by extending an existing probabilistic logic programming language, ProbLog, with neural predicates. Learning is performed by using aProbLog to calculate the gradient of the loss which is then used in standard gradient based methods to optimize parameters in both the probabilistic logic program and the neural networks. We evaluated our framework on experiments that demonstrate its capabilities in combined symbolic and subsymbolic reasoning, program induction, probabilistic logic programming, embeddings and natural language processing. We showed that with the inclusion of logic, DeepProbLog models can outperform neural networks in accuracy, data efficiency, robustness and generalization. We also showed that embeddings can be used directly in DeepProbLog, which illustrates the power of the interface that DeepProbLog provides.

Acknowledgements

This research has been supported by the Research Foundation - Flanders, and the European Research Council Advanced Grant project SYNTH (ERC AdG-694980). It has also received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme.

[1] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. De Raedt, Deepproblog: Neural probabilistic logic programming, in: Advances in Neural Information Processing Systems, 2018, pp. 3749–3759.

[2] D. Kahneman, Thinking, fast and slow, Farrar, Straus and Giroux New York, 2011.

[3] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, T. Lillicrap, A simple neural network module for relational reasoning, in: Advances in Neural Information Processing Systems, volume 30, 2017, pp. 4974–4983.

[4] L. De Raedt, K. Kersting, S. Natarajan, D. Poole, Statistical relational artificial intelligence: Logic, probability, and computation, Synthesis Lectures on Artificial Intelligence and Machine Learning 10 (2016) 1–189.

[5] L. Getoor, B. Taskar, Introduction to statistical relational learning, MIT press, 2007.

[6] L. De Raedt, A. Kimmig, Probabilistic (logic) programming concepts, Machine Learning 100 (2015) 5–47.

[7] A. S. d. Garcez, K. B. Broda, D. M. Gabbay, Neural-symbolic learning systems: foundations and applications, Springer Science & Business Media, 2012.

[8] M. Bošnjak, T. Rocktäschel, S. Riedel, Programming with a differentiable forth interpreter, in: Proceedings of the 34th International Conference on Machine Learning, volume 70, 2017, pp. 547–556.

[9] T. Rocktäschel, S. Riedel, End-to-end differentiable proving, in: Advances in Neural Information Processing Systems, volume 30, 2017, pp. 3788–3800.

[10] W. W. Cohen, F. Yang, K. R. Mazaitis, Tensorlog: Deep learning meets probabilistic databases, Journal of Artificial Intelligence Research 1 (2018) 1–15.

[11] L. De Raedt, R. Manhaeve, S. Dumancic, T. Demeester, A. Kimmig, Neuro-symbolic= neural+ logical+ probabilistic, in: NeSy'19@ IJCAI, the 14th International Workshop on Neural-Symbolic Learning and Reasoning, 2019, pp. 1–4.

[12] L. De Raedt, A. Kimmig, H. Toivonen, ProbLog: A probabilistic Prolog and its application in link discovery, in: IJCAI, 2007, pp. 2462–2467.

[13] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (1998) 2278–2324.

[14] J. W. Lloyd, Foundations of Logic Programming, 2. ed., Springer, 1989.

[15] A. Van Gelder, K. A. Ross, J. S. Schlipf, The well-founded semantics for general logic programs, Journal of the ACM 38 (1991) 620–650.

[16] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] J. Pearl, Probabilistic reasoning in intelligent systems: networks of plausible inference, Morgan Kaufmann Publishers Inc., 1988.

[18] A. Kimmig, G. Van den Broeck, L. De Raedt, Algebraic model counting, Journal of Applied Logic 22 (2017) 46–62.

[19] A. Skryagin, K. Stelzner, A. Molina, F. Ventola, Z. Yu, K. Kersting, Sum-product logic: Integrating probabilistic circuits into deepproblog, in: Working Notes of the ICML 2020 Workshop on Bridge Between Perception and Reasoning: Graph Neural Networks and Beyond, 2020.

[20] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted Boolean formulas, Theory and Practice of Logic Programming 15 (2015) 358–401.

[21] A. Darwiche, P. Marquis, A knowledge compilation map, Journal of Artificial Intelligence Research 17 (2002) 229–264.

[22] A. Darwiche, SDD: A new canonical representation of propositional knowledge bases, in: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, IJCAI-11, 2011, pp. 819–826.

[23] P. Domingos, D. Lowd, Markov logic: An interface layer for artificial intelligence, Synthesis lectures on artificial intelligence and machine learning 3 (2009) 1–155.

[24] M. Frazier, L. Pitt, Learning from entailment: An application to propositional horn sentences, in: Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993, 1993, pp. 120–127.

[25] B. Gutmann, A. Kimmig, K. Kersting, L. De Raedt, Parameter learning in probabilistic databases: A least squares approach, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2008, pp. 473–488.

[26] A. Kimmig, G. Van den Broeck, L. De Raedt, An algebraic Prolog for reasoning about possible worlds., in: AAAI, 2011.

[27] J. Eisner, Parameter estimation for probabilistic finite-state transducers, in: Proceedings of the 40th annual meeting on Association for Computational Linguistics, Association for Computational Linguistics, 2002, pp. 1–8.

[28] A. Solar-Lezama, Program sketching, International Journal on Software Tools for Technology Transfer 15 (2013) 475–495.

[29] S. Reed, N. de Freitas, Neural programmer-interpreters, in: International Conference on Learning Representations (ICLR), 2016.

[30] S. Roy, D. Roth, Solving general arithmetic word problems, in: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, 2015, pp. 1743–1752.

[31] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, O. Yakhnenko, Translating embeddings for modeling multi-relational data, in: Advances in neural information processing systems, 2013, pp. 2787–2795.

[32] K. Sinha, S. Sodhani, J. Dong, J. Pineau, W. L. Hamilton, Clutrr: A diagnostic benchmark for inductive reasoning from text, Empirical Methods of Natural Language Processing (EMNLP) (2019).

[33] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using rnn encoder-decoder for statistical machine translation, arXiv preprint arXiv:1406.1078 (2014).

[34] A. Dries, A. Kimmig, W. Meert, J. Renkens, G. Van den Broeck, J. Vlasselaer, L. De Raedt, Problog2: Probabilistic logic programming, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2015, pp. 312–315.

[35] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, in: Proceedings of the Workshop on The future of gradient-based machine learning software and techniques, co-located with the 31st Annual Conference on Neural Information Processing Systems (NIPS 2017), 2017.

[36] B. Hammer, P. Hitzler, Perspectives of neural-symbolic integration, volume 8, Springer Heidelberg:, 2007.

[37] S. Hölldobler, Y. Kalinke, H.-P. Störr, Approximating the semantics of logic programs by recurrent neural networks, Applied Intelligence 11 (1999) 45–58.

[38] L. De Raedt, R. Manhaeve, S. Dumancic, T. Demeester, A. Kimmig, Neurosymbolic = neural + logical + probabilistic, in: NeSy'19@ IJCAI, the 14th International Workshop on Neural-Symbolic Learning and Reasoning, 2019.

[39] T. Rocktäschel, S. Singh, S. Riedel, Injecting logical background knowledge into embeddings for relation extraction, in: NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2015, pp. 1119–1129.

[40] T. Demeester, T. Rocktäschel, S. Riedel, Lifted rule injection for relation embeddings, in: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, 2016, pp. 1389–1399.

[41] P. Minervini, T. Demeester, T. Rocktäschel, S. Riedel, Adversarial sets for regularised neural link predictors, in: Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI), 2017.

[42] M. Diligenti, M. Gori, C. Sacca, Semantic-based regularization for learning and inference, Artificial Intelligence 244 (2017) 143–165.

[43] I. Donadello, L. Serafini, A. S. d'Avila Garcez, Logic tensor networks for semantic image interpretation, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, 2017, pp. 1596–1602.

[44] J. Xu, Z. Zhang, T. Friedman, Y. Liang, G. V. den Broeck, A semantic loss function for deep learning with symbolic knowledge, in: Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, 2018, pp. 5498–5507.

[45] L. Weber, P. Minervini, J. Münchmeyer, U. Leser, T. Rocktäschel, Nl-prolog: Reasoning with weak unification for question answering in natural language, in: Proceedings of ACL 2018, Tutorial Abstracts, 2019.

[46] J. Andreas, M. Rohrbach, T. Darrell, D. Klein, Neural module networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 39–48.

[47] R. Evans, E. Grefenstette, Learning explanatory rules from noisy data, Journal of Artificial Intelligence Research 61 (2018) 1–64.

[48] G. Šourek, V. Aschenbrenner, F. Železný, S. Schockaert, O. Kuželka, Lifted relational neural networks: Efficient learning of latent relational structures, Journal of Artificial Intelligence Research to appear (2018).

[49] S. M. Kazemi, D. Poole, RelNN: A deep neural model for relational learning, in: AAAI, 2018.

[50] M. Lippi, P. Frasconi, Prediction of protein β-residue contacts by Markov logic networks with grounding-specific weights, Bioinformatics 25 (2009) 2326–2333.

[51] H. Wang, H. Poon, Deep probabilistic logic: A unifying framework for indirect supervision, in: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018, 2018, pp. 1891–1902.

[52] M. Richardson, P. Domingos, Markov logic networks, Machine learning 62 (2006) 107–136.

[53] S. H. Bach, M. Broecheler, B. Huang, L. Getoor, Hinge-loss markov random fields and probabilistic soft logic, arXiv preprint arXiv:1505.04406 (2015).

[54] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, S. Gulwani, Neural-guided deductive search for real-time program synthesis from examples, in: ICLR, 2018.

[55] L. Zhang, G. Rosenblatt, E. Fetaya, R. Liao, W. E. Byrd, M. Might, R. Urtasun, R. Zemel, Neural guided constraint logic programming for program synthesis, in: NeurIPS, 2018.

[56] K. Ellis, L. Morales, M. Sablé-Meyer, A. Solar-Lezama, J. Tenenbaum, Learning libraries of subroutines for neurally–guided bayesian program induction, in: NeurIPS, 2018.

[57] K. Yi, J. Wu, C. Gan, A. Torralba, P. Kohli, J. B. Tenenbaum, Neural-symbolic vqa: Disentangling reasoning from vision and language understanding, in: NeurIPS, 2018.

[58] J. Mao, C. Gan, P. Kohli, J. B. Tenenbaum, J. Wu, The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision, in: ICLR, 2019.

[59] K. Ellis, D. Ritchie, A. Solar-Lezama, J. Tenenbaum, Learning to infer graphics programs from hand-drawn images, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems 31, Curran Associates, Inc., 2018, pp. 6059–6068.

[60] S. M. Kazemi, D. Buchman, K. Kersting, S. Natarajan, D. Poole, Relational logistic regression: The directed analog of markov logic networks, in: Proceedings of the 13th AAAI Conference on Statistical Relational AI, AAAIWS'14-13, AAAI Press, 2014, pp. 41–43.

[61] S. Dumančić, H. Blockeel, Clustering-based relational unsupervised representation learning with an explicit distributed representation, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, pp. 1631–1637.

[62] S. Dumančić, T. Guns, W. Meert, H. Blockeel, Learning relational representations with auto-encoding logic programs, in: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, 2019, p. To appear.

[63] A. Cropper, Playgol: Learning programs through play, in: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, 2019, p. To appear.

[64] A. Kimmig, V. S. Costa, R. Rocha, B. Demoen, L. De Raedt, On the efficient execution of problog programs, in: International Conference on Logic Programming, Springer, 2008, pp. 175–189.

[65] J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, L. De Raedt, Anytime inference in probabilistic logic programs with tp-compilation, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.

Appendix A. DeepProbLog Programs

```
nn(m_digit,[X],Y,[0,...,9]) :: digit(X,Y).

addition(X,Y,Z) :- digit(X,X2), digit(Y,Y2), Z is X2+Y2.
```

Listing 3: Single-digit MNIST addition (**T1**)

In Listing 3, `digit/2` is the neural predicate that classifies an MNIST image into the integers 0 to 9. The `addition/3` predicate's first two arguments are MNIST digits, and the last is the sum. It classifies both images using `digit/2` and calculates the sum of the two results.

```
nn(m_digit,[X],Y,[0,...,9]) :: digit(X,Y).

number([],Result,Result).
number([H|T],Acc,Result) :-
    digit(H,Nr),
    Acc2 is Nr+10*Acc,
    number(T,Acc2,Result).
number(X,Y) :- number(X,0,Y).

multi_addition(X,Y,Z) :- number(X,X2), number(Y,Y2), Z is X2+Y2.
```

Listing 4: Multi-digit MNIST addition (**T2**)

In Listing 4, the only difference with Listing 3 is that the `multi_addition/3` predicate now uses the `number/2` predicate instead of the `digit/2` predicate. The `number/3` predicate's first argument is a list of MNIST images. It uses the `digit/2` neural predicate on each image in the list, summing and multiplying by ten to calculate the number represented by the list of images (e.g. `number([` `,` `],38)`).

```
nn(m_digit,[X],Y,[0,...,9]) :: digit(X,Y).

addition(X,Y,Z) :- digit(X,X2), digit(Y,Y2), digit(Z,Z2), Z2 is X2+Y2.
```

Listing 5: All-digit MNIST addition (**T3**)

In Listing 5, the only difference with Listing 3 is that all 3 inputs `X,Y,Z` are images. As such, the `digit/2` predicate is also used on the third input. The sum is also redefined as `Z2 is X2+Y2`.

In Listing 6, an additional probabilistic fact (`noisy/2`) is added that encodes the chance of an example being noisy. The `addition/3` predicate is split into two cases: when the noisy is true or when noisy is false. The latter is the same as in Listing 3. If `noisy` is true, Z is considered to be drawn from the uniform distribution (`uniform/3`).

```
nn(classifier, [X], Y, [0 .. 9]) :: digit(X,Y).
t(0.2) :: noisy.

1/19 :: uniform(X,Y,0) ; ... ; 1/19 :: uniform(X,Y,18).

addition(X,Y,Z) :- noisy, uniform(X,Y,Z).
addition(X,Y,Z) :- \+noisy, digit(X,N1), digit(Y,N2), Z is N1+N2.
```

Listing 6: Noisy MNIST addition (**T4**)

```
nn(m_result,[D1,D2,Carry],Y,[0,...,9])::result(D1,D2,Carry,Y).

nn(m_carry,[D1,D2,Carry],Y,[0,1])::carry(D1,D2,Carry,Y).

hole(I1,I2,Carry,NewCarry,Result) :-
    result(I1,I2,Carry,Result),
    carry(I1,I2,Carry,NewCarry).

add([],[],[C],C,[]).

add([H1|T1],[H2|T2],C,Carry,[Digit|Res]) :-
    add(T1,T2,C,NewCarry,Res),
    hole(H1,H2,NewCarry,Carry,Digit).

forth_addition(L1,L2,C,[Carry|Res]) :- add(L1,L2,C,Carry,Res).
```

Listing 7: Forth addition sketch (**T5**)

In Listing 7, there are two neural predicates: `result/4` and `carry/4`. These are used in the `hole/4` predicate that corresponds to the hole in the Forth program. The first three arguments are the two digits and the previous carry to be summed. The next two arguments are the new carry and the new resulting digit. The `add/5` predicate's arguments are: the two list of input digits, the input carry, the resulting carry and the resulting sum. It recursively calls itself to loop over both lists, calling the `hole/5` predicate on each position, using the carry from the previous step.

In Listing 8, there's a single neural predicate: `swap/3`. Its first two arguments are the numbers that are compared, the last argument is an indicator whether to swap or not. The `bubble/3` predicate performs a single step of bubble sort on its first argument using the `hole/4` predicate. The second argument is the resulting list after the bubble step, but without its last element, which is the third argument. The `bubblesort/3` predicate uses the `bubble/3` predicate, and recursively calls itself on the remaining list, adding the last element on each step to the front of the sorted list.

In Listing 9, there are four neural predicates: `net1/2` to `net4/2`. Their first argument is the input question, and the second argument are indicator variables for the choice of respectively: one of six permutations, one of 4 operations, swapping and one of 4 operations. These are implemented in the `permute/7`, `swap/5` and `operator/4` predicates. The `wap/5` predicate then sequences these steps to

```
nn(m_swap, [X]) :: swap(X,Y,).

hole(X,Y,X,Y):-\+swap(X,Y).

hole(X,Y,Y,X):-swap(X,Y).

bubble([X],[],X).
bubble([H1,H2|T],[X1|T1],X):-
    hole(H1,H2,X1,X2),
    bubble([X2|T],T1,X).

bubblesort([],L,L).

bubblesort(L,L3,Sorted) :-
    bubble(L,L2,X),
    bubblesort(L2,[X|L3],Sorted).

forth_sort(L,L2) :- bubblesort(L,[],L2).
```

Listing 8: Forth sorting sketch (**T6**)

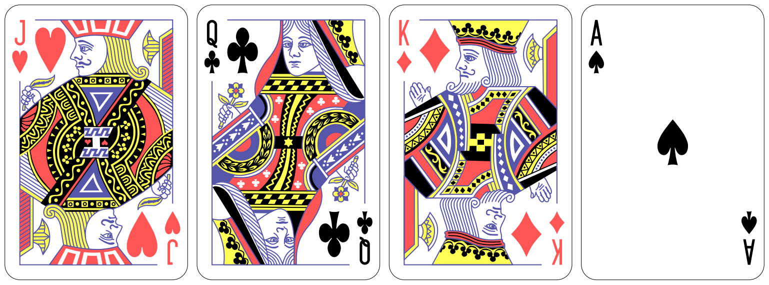


Figure A.16: Examples of cards used as input for the Poker without perturbations(**T9**) experiment.

calculate the result.

In Listing 10, there are two neural predicates: `coin1/2` and `coin2/2`. Their input is the image of the two coins (e.g. Figure 10). The output is heads or tails. The `coins/2` classifies both coins using these two predicates and then performs the comparison of the classes with the `compare/3` predicate.

In Listing 11, there's a single neural predicate `rank/2` that takes as input the image of a card and classifies it as either a jack, queen, king or ace. There's also an AD with learnable parameters that represents the distribution of the unseen community card (`house_rank/1`). The `hand/2` predicate's first argument is a list of 3 cards. It unifies the output with any of the valid hands that these cards contain. The valid hands are: high card, pair (two cards have the same rank), three of a kind (three cards have the same rank), low straight (jack, queen king) and high straight(queen, king, ace). Each hand is assigned a rank with the

```
permute(0,A,B,C,A,B,C).
permute(1,A,B,C,A,C,B).
permute(2,A,B,C,B,A,C).
permute(3,A,B,C,B,C,A).
permute(4,A,B,C,C,A,B).
permute(5,A,B,C,C,B,A).

swap(0,X,Y,X,Y).
swap(1,X,Y,Y,X).

operator(0,X,Y,Z) :- Z is X+Y.
operator(1,X,Y,Z) :- Z is X-Y.
operator(2,X,Y,Z) :- Z is X*Y.
operator(3,X,Y,Z) :- Y > 0, 0 =:= X mod Y,Z is X//Y.

nn(m_net1,[Repr],Y,[0,...,5])::net1(Repr,Y).
nn(m_net2,[Repr],Y,[0,...,3])::net2(Repr,Y).
nn(m_net3,[Repr],Y,[0,1])::net3(Repr,Y).
nn(m_net4,[Repr],Y,[0,...,3])::net4(Repr,Y).

wap(Text,X1,X2,X3,Out) :-
    net1(Text,Perm),
    permute(Perm,X1,X2,X3,N1,N2,N3),
    net2(Text,Op1),
    operator(Op1,N1,N2,Res1),
    net3(Text,Swap),
    swap(Swap,Res1,N3,X,Y),
    net4(Text,Op2),
    operator(Op2,X,Y,Out).
```

Listing 9: Forth WAP sketch (**T7**)

`hand_rank/2` predicate. The `best_hand_rank/2` predicate takes as input a list of cards, and unifies the second argument with the highest hand rank that is possible with the three given cards. The `outcome/3` predicate determines the outcome by comparing the two ranks of the best hand. The `game/3` predicate's first argument is a list of the 4 input images. Its second input is the labeled community card. It classifies the cards using the neural predicates, determines the best rank, and then unifies the last argument with the outcome. The `game/2` determines the community card from the learned distribution `house_rank/1`, and then determines the outcome using the `game/3` predicate. The `member/2` and `select/3` predicates are predicates from the *lists* library. `member/2` is true if its second argument is a list and the first argument appears in that list. `select/3` is true if its second argument is a list and the first argument appears in that list. It also unifies the last argument with the list that is the same as its second argument, but with the first argument removed.

In Listing 13, the main predicate is the `query_rel(T,X,R,Y)` predicate that will predict the relationship R that holds between entities X and Y in text T. To simplify reasoning, all relations are reduced to a non-gendered version

```
nn(net1, [X], Y, [heads, tails]) :: coin1(X,Y).
nn(net2, [X], Y, [heads, tails]) :: coin2(X,Y).

compare(X,X,same).
compare(X,Y,different) :- \+compare(X,Y,same).

coins(X,Comparison) :-
    coin1(X,C1),
    coin2(X,C2),
    compare(C1,C2,Comparison).
```

Listing 10: The coins experiment (**T8**)

(e.g., significant other instead of husband). The `gender_rel/3` predicate provides information about all the possible relations, and ties the gendered version to its non-gendered version (e.g., `gender_rel(child,female,daughter)`. The gender is determined separately using the `neural_gender/3` predicate.

After this, a meta-call happens to try to prove the actual relation using the `call` predicate. There are two ways to prove the relations: either directly using the neural predicates, or through the logic that encodes the family relations. Also note that we explicitly encode the depth `D` when proving recursively to avoid cycles. Although DeepProbLog can deal with cycles, this was done to improve performance. Note also that we pre-process the text by splitting it into individual sentences, replacing the entity names with numbers, and keeping track which entities are mentioned in each sentence. Although this could be done inside of the logic, we did this in a separate step to keep the logic more readable. In the `neural/4` predicate, we first filter out any sentences that do not mention the relevant entities we want to know the relation for. This is done using the `include/3` predicate. We then map every sentence onto its embedding by using the `maplist/3` predicate to loop over the list and the `embed/3` predicate to map the given sentence. The first argument is a list of entities that determine how the entities should be encoded. The first entity in this list will be encoded with the token ENT1, the second with ENT2, ... Any other entity will be tokenized as ENT. We take the element-wise max of all these embeddings using the `max_tensor/2` predicate. Then the actual prediction is made using the `nn_rel/2` neural predicate.

```
t(1/4)::house_rank(jack);t(1/4)::house_rank(queen);
    t(1/4)::house_rank(king);t(1/4)::house_rank(ace).
nn(net1,[X],Y,[jack,queen,king,ace]):: rank(X,Y).

hand(Cards,straight(low)) :-
    member(card(jack),Cards),
    member(card(queen),Cards),
    member(card(king),Cards).
hand(Cards,straight(high)) :-
    member(card(queen),Cards),
    member(card(king),Cards),
    member(card(ace),Cards).
hand([card(R), card(R), card(R)],threeofakind(R)).
hand(Cards,pair(R)) :-
    select(card(R),Cards,Cards2),
    member(card(R),Cards2).
hand(Cards,high(R)) :-
    member(card(R),Cards).

hand_rank(high(jack),0).
...
hand_rank(straight(high),13).

best_hand_rank(Cards,R) :-
    hand(Cards,H),
    hand_rank(H,R),
    \+(hand(Cards,H2),hand_rank(H2,R2),R2>R).

outcome(R1,R2,win) :- R1 > R2.
outcome(R1,R2,loss) :- R1 < R2.
outcome(R,R,draw).

cards(C1,C2,House,[card(R1), card(R2), House]) :-
    rank(C1,R1),
    rank(C2,R2).

game([C1,C2,C3,C4],House,Outcome) :-
    cards(C1,C2,House,Hand1),
    cards(C3,C4,House,Hand2),
    best_hand_rank(C1,R1),
    best_hand_rank(C2,R2),
    outcome(R1,R2,Outcome).

game(Cards,Outcome) :-
    house_rank(House),
    game(Cards,House,Outcome).
```

Listing 11: The Poker experiment (**T9**)

```
successor(I1,I2,N) :-
    %Encode images I1 and I2 into E1 and E2
    cnn_encode(I1,E1), cnn_encode(I2,E2),
    %Embed the successor relation into embedding S
    embed(successor, S),
    %E = E1 + N*S
    mul(S,N,S2), add(E1,S2,E),
    rbf(E,E2).
```

Listing 12: The MNIST successor task (**T10**)

```
:-use_module(library(apply)).

nn(net1,[T],R,[child,child_in_law,...]) :: nn_rel(T,R).
nn(net2,[T],G,[male,female]) :: nn_gender(T,G).

query_rel(T,X,R,Y):-gender_rel(R2,G,R),neural_gender(T,Y,G),call(R2,T,X,Y,3).

mentioned(X,s(E,_)) :- member(X,E).

neural_gender([Entities|Sentences], E, G) :-
    member(E,Entities),
    include(mentioned(E), Sentences, [H|Sentences2]),
    maplist(embed([E]),[H|Sentences2],Embedded),
    max_tensor(Embedded,Max),
    nn_gender(Max,G).

neural([Entities|Sentences], X,R,Y) :-
    member(X,Entities), member(Y,Entities), X \== Y,
    include(mentioned(X),Sentences,Sentences2),
    include(mentioned(Y),Sentences2,[H|Sentences3]),
    maplist(embed([Y,X]),[H|Sentences3],Embedded),
    max_tensor(Embedded,Max),
    nn_rel(Max,R).


child(T,X,Y,_) :- neural(T,X,child,Y).
child_in_law(T,X,Y,_) :- neural(T,X,child_in_law,Y).
grandchild(T,X,Y,_) :-  neural(T,X,grandchild,Y).
...

grandchild(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), child(T,Z,Y,D-1).
grandchild(T,X,Y,D) :- D>0,X\=Y, so(T,X,Z,D-1), grandchild(T,Z,Y,D-1).
grandchild(T,X,Y,D) :- D>0,X\=Y, grandchild(T,X,Z,D-1), sibling(T,Z,Y,D-1).
grandparent(T,X,Y,D) :- D>0,X\=Y, parent(T,X,Z,D-1), parent(T,Z,Y,D-1).
grandparent(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), grandparent(T,Z,Y,D-1).
child(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), sibling(T,Z,Y,D-1).
child(T,X,Y,D) :- D>0,X\=Y, so(T,X,Z,D-1), child(T,Z,Y,D-1).
parent(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), parent(T,Z,Y,D-1).
parent(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), grandparent(T,Z,Y,D-1).
sibling(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1), uncle(T,Z,Y,D-1).
sibling(T,X,Y,D) :- D>0,X\=Y, parent(T,X,Z,D-1), child(T,Z,Y,D-1).
sibling(T,X,Y,D) :- D>0,X\=Y, sibling(T,X,Z,D-1), sibling(T,Z,Y,D-1).
child_in_law(T,X,Y,D) :- D>0,X\=Y, child(T,X,Z,D-1),so(T,Z,Y,D-1).
parent_in_law(T,X,Y,D) :- D>0,X\=Y, so(T,X,Z,D-1), parent(T,Z,Y,D-1).
nephew(T,X,Y,D) :-D>0,X\=Y, sibling(T,X,Z,D-1), child(T,Z,Y,D-1).
uncle(T,X,Y,D) :- D>0,X\=Y, parent(T,X,Z,D-1), sibling(T,Z,Y,D-1)

gender_rel(child,male,son).
gender_rel(child,female,daughter).
gender_rel(parent,male,father).
gender_rel(parent,female,mother).
...
```

Listing 13: The CLUTRR experiment (**T11**)