Neural Probilistic Logic Programming in Discrete-Continuous Domains: Supplementary Material

Lennert De Smet[1] **Pedro Zuidberg Dos Martires**[2] **Robin Manhaeve**[1] **Giuseppe Marra**[1] **Angelika Kimmig**[1]
Luc De Raedt[1,2]

[1]Department of Computer Science, KU Leuven, Belgium
[2]Center for Applied Autonomous Systems, Örebro, Sweden

A SPECIAL CASES OF DEEPSEAPROBLOG

The syntax and semantics of DeepSeaProbLog generalise a number of probabilistic logic programming dialects. For instance, if we assume no dependency of the distributional facts on input data or external neural functions, we obtain a language equivalent to Gutmann et al.'s *Distributional Clauses* (DC) [Gutmann et al., 2011] when restricted to distributional facts. Finally, if we allow for data dependent neural functions in the NDFs but restrict them to Bernoulli and categorical distributions, we obtain Manhaeve et al.'s DeepProbLog [Manhaeve et al., 2021a] as a special case.

Proposition A.1 (DeepSeaProbLog strictly generalises DeepProbLog)**.** DeepProbLog is a strict subset of DeepSeaProbLog where the set of comparison predicates is restricted to $\{\texttt{=:=}\}$, comparisons involve exactly one random variable and the measure $\mathrm{d}P_{\mathcal{F}_D}$ factorizes as a product of independent Bernoulli measures $\prod_{i:\mathtt{x}_i\sim\mathtt{b}_i\in\mathcal{F}_D}\mathrm{d}P_{b_i}$. The subscript on $\mathrm{d}P_{b_i}$ explicitly identifies the measure as the i^{th} Bernoulli measure and the indices of the product go over all the (Bernoulli) random variables defined in the set of distributional facts $\mathcal{F}_D$.

Proof. We prove Proposition A.1 by showing that applying the restrictions on the constraints and measure in a DeepSeaProbLog program leads to possible worlds that have the same probability of being true as in DeepProbLog. First we write down the definition of the probability $P(\omega_{C_M})$ of a possible world in a DeepSeaProbLog program

$$\int \left[\left(\prod_{c_i\in C_M}\mathbb{1}(c_i)\right)\left(\prod_{c_i\in\mathcal{C}_M\setminus C_M}\mathbb{1}(\bar{c}_i)\right)\right]\mathrm{d}P_{\mathcal{F}_D}. \tag{1}$$

Now observe that, since there are only Bernoulli distributions, we only need to consider two possible outcomes of a random variable $\mathtt{x}_i$, either zero or one. Therefore, only two kinds of comparisons are present in the program, $\mathtt{x}_i\texttt{=:=0}$ or $\mathtt{x}_i\texttt{=:=1}$ (remember that we restrict ourselves to univariate comparisons). Now note that the following equivalence $\mathtt{x}_i\texttt{=:=1} \leftrightarrow \neg(\mathtt{x}_i\texttt{=:=0})$ holds, which means that we can arbitrarily limit comparisons to one of the two possible outcomes of a random variable, e.g., $\mathtt{x}_i\texttt{=:=0}$.

This equivalence can be used to replace the constraints c_i in Equation 1 by equality constraints involving comparisons to the zero outcome, i.e., $P(\omega_{C_M})$ is equal to

$$\int\left(\prod_{i:c_i\in C_M}\mathbb{1}(x_i{=}0)\right)\cdot\left(\prod_{i:c_i\in\mathcal{C}_M\setminus C_M}\mathbb{1}(x_i{\neq}0)\right)\prod_{i:\mathtt{x}_i\sim\mathtt{b}_i\in\mathcal{F}_D}\mathrm{d}P_{b_i}, \tag{2}$$

where the factorisation of the measure was also applied. Next, we introduce the following notation for the random variables present in the set of constraints C_M and $\mathcal{C}_M\setminus C_M$:

$$\boldsymbol{x}^+ := x_i : c_i\in C_M \tag{3}$$

$$\boldsymbol{x}^- := x_i : c_i\in\mathcal{C}_M\setminus C_M \tag{4}$$

Accepted for the 39th Conference on Uncertainty in Artificial Intelligence (UAI 2023).

Note that we only need to consider the case where $\boldsymbol{x}^+ \cap \boldsymbol{x}^- = \emptyset$, as otherwise the probability of the possible world would simply be zero and would not contribute to the overall probability of the query atom. Because of this, we can further factorize the measure as

$$\prod_{i:\mathrm{x}_i \sim \mathrm{b}_i \in \mathcal{F}_D} \mathrm{d}P_{b_i} = \underbrace{\left(\prod_{i:x_i \in \boldsymbol{x}^+} \mathrm{d}P_{b_i} \right)}_{=:\mathrm{d}P^+} \underbrace{\left(\prod_{i:x_i \in \boldsymbol{x}^-} \mathrm{d}P_{b_i} \right)}_{=:\mathrm{d}P^-}, \tag{5}$$

so the integral of a product in Equation 2 can be rewritten as the product of integrals

$$P(\omega_{C_M}) = \left[\int \left(\prod_{i:c_i \in C_M} \mathbb{1}(x_i{=}0)\, \mathrm{d}P^+ \right) \right] \cdot \left[\int \left(\prod_{i:c_i \in \mathcal{C}_M \setminus C_M} \mathbb{1}(x_i{\neq}0)\, \mathrm{d}P^- \right) \right]. \tag{6}$$

We have two integrals with integrands that are a product of univariate comparisons. In other words, the factors are all independent. Furthermore, we have a Bernoulli product measure, which means that we can again push the integral inside the product to yield

$$P(\omega_{C_M}) = \left[\prod_{i:c_i \in C_M} \left(\int \mathbb{1}(x_i{=}0)\, \mathrm{d}P^+ \right) \right] \cdot \left[\prod_{i:c_i \in \mathcal{C}_M \setminus C_M} \left(\int \mathbb{1}(x_i{\neq}0)\, \mathrm{d}P^- \right) \right]. \tag{7}$$

At this point we can simply perform the integrations and obtain

$$P(\omega_{C_M}) = \prod_{i:c_i \in C_M} p_i \prod_{i:c_i \in \mathcal{C}_M \setminus C_M} (1 - p_i), \tag{8}$$

which coincides with the probability of a possible world in DeepProbLog [Manhaeve et al., 2021a, Section 3]. □

Proposition A.1 can easily be extended to also allow for measures of finite categorical distributions, which then translates to (neural) annotated disjunctions. Consequently, as DeepProbLog is a strict superset of ProbLog [Fierens et al., 2015], DeepSeaProbLog also strictly generalises ProbLog.

B PROOF OF PROPOSITION B.1

Proposition B.1 (Measureability of query atom)**.** Let $\mathbb{P}$ be a DeepSeaProbLog program, then $\mathbb{P}$ defines, for an arbitrary query atom q, the probability that q is true.

Proof. DeepSeaProbLog is in essence a subset of the probabilistic logic programming language defined by Gutmann et al. [2011] – the only difference being that the parameters on the right-hand side of a neural distributional fact are not limited to numerical constants any more but can be arbitrary numeric terms. Under the condition that all NDFs and PCFs are valid, this does, however, not violate any of the assumptions made in [Gutmann et al., 2011, Proposition 1] (proving the measurability of a program). We can, hence, conclude that a valid DeepSeaProbLog program induces a probability measure for q. □

Note that, similar to ProbLog and DeepProbLog, the semantics for DeepSeaProbLog are only defined for so-called sound programs [Riguzzi and Swift, 2013], which means that all programs become ground eventually when queried.

C PROOF OF PROPOSITION C.1

Proposition C.1 (Inference as WMI)**.** Assume that the measure $\mathrm{d}P_{\mathcal{F}_D}$ decomposes into a joint probability density function $w(\boldsymbol{x})$ and a differential $\mathrm{d}\boldsymbol{x}$, then the probability $P(q)$ of a query atom q can be expressed as the weighted model integration problem

$$\int \left[\sum_{C_M \subseteq \mathcal{C}_M : q \in \omega_{C_M}} \prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(\boldsymbol{x})) \right] w(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}, \tag{9}$$

where $\overline{C}_M \coloneqq \{ \bar{c}_i \mid c_i \in \mathcal{C}_M \setminus C_M \}$.

Proof. First, let us consider the indices of the two product expressions in

$$P(\omega_{C_M}) = \int \left[\Big(\prod_{c_i \in C_M} \mathbb{1}(c_i) \Big) \Big(\prod_{c_i \in \mathcal{C}_M \setminus C_M} \mathbb{1}(\bar{c}_i) \Big) \right] \mathrm{d}P_{\mathcal{F}_D}. \tag{10}$$

We define

$$\overline{C}_M \coloneqq \{ \bar{c}_i \mid c_i \in \mathcal{C}_M \setminus C_M \}$$

such that Equation 10 can be rewritten as

$$P(\omega_{C_M}) = \int \left(\prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(\boldsymbol{x}) \right) \mathrm{d}P_{\mathcal{F}_D} \tag{11}$$

Furthermore, decomposing the measure into a probability density function $w(\boldsymbol{x})$ and a differential $\mathrm{d}\boldsymbol{x}$ of the integration variables yields

$$\int \left(\prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(\boldsymbol{x})) \right) \cdot w(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}. \tag{12}$$

We can now plug this last expression into

$$P(q) = \sum_{C_M \subseteq \mathcal{C}_M : q \in \omega_{\mathcal{C}_M}} P(\omega_{C_M}), \tag{13}$$

resulting in

$$P(q) = \int \sum_{\substack{C_M \subseteq \mathcal{C}_M : \\ q \in \omega_{\mathcal{C}_M}}} \left(\prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(\boldsymbol{x})) \right) \cdot w(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}. \tag{14}$$

Note that we changed the order of the integration and summation. This operation was shown to be valid in Zuidberg Dos Martires et al. [2019] using de Finetti's theorem. Zuidberg Dos Martires et al. [2019] also showed that the expression in Equation 14 is indeed a weighted model integral as defined by Belle et al. [2015]. Specifically, line P2 in the proof of Theorem 2 in Zuidberg Dos Martires et al. [2019] corresponds to C.3, which is shown to be equal to an instance of WMI. □

D DETAILS ON DERIVATIVE ESTIMATE

To give further details on estimating the derivative we will write the expression $\partial_\lambda P_{\boldsymbol{\Lambda}}(q)$ in terms of indicator functions

$$\partial_\lambda P_{\boldsymbol{\Lambda}}(q) = \partial_\lambda \int \mathrm{SP}(\boldsymbol{x}) \cdot w_{\boldsymbol{\Lambda}}(\boldsymbol{x}) \, \partial \boldsymbol{x} \tag{15}$$

$$= \partial_\lambda \int \sum_{\substack{C_M \subseteq \mathcal{C}_M : \\ q \in \omega_{\mathcal{C}_M}}} \left(\prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(\boldsymbol{x})) \right) \cdot w_{\boldsymbol{\Lambda}}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}, \tag{16}$$

where the dependency of the probability on the neural parameters $\boldsymbol{\Lambda}$ is again made explicit. Reparametrising the distribution $w_{\boldsymbol{\Lambda}}(\boldsymbol{x})$ yields

$$\partial_\lambda P_{\boldsymbol{\Lambda}}(q) = \partial_\lambda \int \sum_{\substack{C_M \subseteq \mathcal{C}_M : \\ q \in \omega_{\mathcal{C}_M}}} \left(\prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \right) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}. \tag{17}$$

Explicitly writing out the indicators clearly illustrates the non-differentiability of $\text{SP}(\boldsymbol{x})$, which prevents us from applying Leibniz' integral rule [Flanders, 1973] to swap the order of integration and differentiation. To obtain the necessary differentiability of the integrand, the continuous relaxations introduced by Petersen et al. [2021] are utilised. These relaxations allow for comparison formulae of the form

$$(g(\boldsymbol{x}) \bowtie 0), \quad \text{with} \bowtie \in \{<, \leq, >, \geq, =, \neq\} \tag{18}$$

to be relaxed. We write the continuous relaxation of an indicator function $\mathbb{1}(c_i(\boldsymbol{x})) = \mathbb{1}(g_i(\boldsymbol{x}) \bowtie 0)$ as $s_i(\boldsymbol{x})$. Four specific cases of relaxations arise, depending on the comparison operator used. Specifically, we define

$$s_i(\boldsymbol{x}) = \begin{cases} \sigma(\beta_i \cdot g_i(\boldsymbol{x})) & \text{if} \bowtie \in \{>, \geq\}, \\ \sigma(-\beta_i \cdot g_i(\boldsymbol{x})) & \text{if} \bowtie \in \{<, \leq\}, \\ \prod \sigma(\beta_i \cdot g_i(\boldsymbol{x})) \cdot \sigma(-\beta_i' \cdot g_i(\boldsymbol{x})) & \text{if} \bowtie \in \{=\}, \\ 1 - \sigma(\beta_i \cdot g_i(\boldsymbol{x})) \cdot \sigma(-\beta_i' \cdot g_i(\boldsymbol{x})) & \text{if} \bowtie \in \{\neq\}, \end{cases} \tag{19}$$

where β_i and β_i' are the coolness parameters of the continuous relaxations and σ denotes the sigmoid function. Note that all four cases originate from the root choice of approximating the step function as a sigmoid function. Additionally, this choice is sound as we have that

$$\lim_{\beta_i \to +\infty} \sigma(\beta_i \cdot g_i(\boldsymbol{x})) = \mathbb{1}(g_i(\boldsymbol{x}) \geq 0). \tag{20}$$

Continuously relaxing indicator functions using the definition of Equation 19 renders the integrand differentiable, allowing the application of Leibniz' integral rule and yielding

$$\partial_\lambda P_{\boldsymbol{\Lambda}}(q) \approx \int \partial_\lambda \sum_{\substack{C_M \subseteq \mathcal{C}_M: \\ q \in \omega_{C_M}}} \left(\prod_{i: c_i \in C_M \cup \overline{C}_M} s_i(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \right) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}.$$

The derivative $\partial_\lambda P_{\boldsymbol{\Lambda}}(q)$ can now be computed using off-the-shelf automatic differentiation software such as PyTorch [Paszke et al., 2019] or TensorFlow [Abadi, 2016], which entails that estimating the gradient $\nabla_{\boldsymbol{\Lambda}} P(q) = (\partial_\lambda P(q))_{\lambda \in \boldsymbol{\Lambda}}$ is computationally as expensive as computing the probability itself, up to a constant factor [Griewank and Walther, 2008].

E PROOF OF PROPOSITION E.1

Proposition E.1 (Unbiased in the infinite coolness limit)**.** Let $\mathbb{P}$ be a DeepSeaProbLog program with PCFs $(g_i(\boldsymbol{x}) \bowtie 0)$ and corresponding coolness parameters β_i.
If all $\partial_\lambda (g_i \circ r)$ are locally integrable over $\mathbb{R}^k$ and every $\beta_i \to +\infty$, then we have, for any query atom q, that

$$\partial_\lambda P(q) = \int \partial_\lambda \text{SP}_s(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}. \tag{21}$$

Proof. First we express $P(q)$ using Equation 14, which we then rewrite without loss of generalisation using only Heaviside distributions[1].

$$\begin{aligned} P(q) &= \int \sum_{\substack{C_M \subseteq \mathcal{C}_M: \\ q \in \omega_{C_M}}} \left(\prod_{c_i \in C_M \cup \overline{C}_M} \mathbb{1}(c_i(\boldsymbol{x})) \right) \cdot w(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} & (22) \\ &= \int \sum_{\substack{C_M \subseteq \mathcal{C}_M: \\ q \in \omega_{C_M}}} \left(\prod_{g_i \in \Sigma_{C_M \cup \overline{C}_M}} H(g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \right) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}. & (23) \end{aligned}$$

[1]Here we use the term *distribution* in the sense of a generalised function [Schwartz, 1957] and not in the sense of a probability distribution.

In the Equation above, $H(x)$ denotes the Heaviside distribution and $\Sigma_{C_M \cup \overline{C}_M}$ is the set of all sigmoid functions involved in the continuous relaxations of the set $C_M \cup \overline{C}_M$.

This rewrite is possible as the indicator function of any PCF $c(\boldsymbol{x})$ is either a step function or decomposes into a product of step functions. Indeed, if $c(\boldsymbol{x})$ is of the form $g(\boldsymbol{x}) \geq 0$, then $\mathbb{1}(c(\boldsymbol{x})) = H(g(\boldsymbol{x}))$. If it is of the form $g(\boldsymbol{x}) = 0$, then $\mathbb{1}(c(\boldsymbol{x})) = H(g(\boldsymbol{x})) \cdot H(-g(\boldsymbol{x}))$. The other cases with different comparison operators follow from these two.

Differentiating in a distributional sense and applying Leibniz' integral rule [Flanders, 1973] then yields

$$\sum_{\substack{C_M \subseteq \mathcal{C}_M: \\ q \in \omega_{\mathcal{C}_M}}} \sum_{g_j \in \Sigma_{C_M \cup \overline{C}_M}} \int \partial_\lambda H(g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot \prod_{i \neq j} H(g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}. \tag{24}$$

We can reduce the discussion by considering each term in this equation separately, because of the linearity of the integral. In other words, to prove our statement, it suffices to show that

$$\int \partial_\lambda H(g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot \prod_{i \neq j} H(g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}, \tag{25}$$

is equal to

$$\lim_{\beta_1, \ldots, \beta_n \to +\infty} \int \partial_\lambda \sigma(\beta_j \cdot g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot \prod_{i \neq j} \sigma(\beta_i \cdot g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}. \tag{26}$$

For brevity's sake, we will write the products

$$\prod_{i \neq j} H(g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \qquad \text{and} \qquad \prod_{i \neq j} \sigma(g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))), \tag{27}$$

as $\pi_j(\boldsymbol{u})$ and $\pi_j^\sigma(\boldsymbol{u})$, respectively. Next, using distributional notation, Equation 25 can be further simplified as

$$\langle \partial_\lambda (H \circ g_j \circ r), \, \pi_j \cdot p \rangle = \langle \delta \circ g_j \circ r, \, \partial_\lambda \, (g \circ r) \cdot \pi_j \cdot p \rangle \, . \tag{28}$$

Note that this expression utilises the assumption that $\partial_\lambda (g_j \circ r) \in L^1_{loc}(\mathbb{R}^k)$, i.e., $\partial_\lambda (g_j \circ r)$ is locally integrable over $\mathbb{R}^k$. This asssumption is not very demanding, since distributions (generalised functions) are only well-defined when acting on functions that are at least locally integrable. Equation 26 can similarly be rewritten and simplified to obtain the equality

$$\lim_{\beta_1, \ldots, \beta_n \to +\infty} \left\langle \partial_\lambda (\sigma \circ g_j \circ r), \, \pi_j^\sigma \cdot p \right\rangle = \lim_{\beta_1, \ldots, \beta_{j-1}, \beta_{j+1}, \ldots, \beta_n \to +\infty} \left\langle \delta \circ g_j \circ r, \, \partial_\lambda (g \circ r) \cdot \pi_j^\sigma \cdot p \right\rangle . \tag{29}$$

More explicitly,

$$\begin{aligned}
(26) &= \lim_{\beta_1, \ldots, \beta_n \to +\infty} \int \partial_\lambda \sigma(\beta_j \cdot g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot \pi_j^\sigma(\boldsymbol{u}) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u} & (30) \\
&= \lim_{\beta_1, \ldots, \beta_n \to +\infty} \int \frac{l \cdot e^{-g(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot \beta_j}}{(1 + e^{-g(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot \beta_j})^2} \cdot \int \partial_\lambda g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot \pi_j^\sigma(\boldsymbol{u}) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u} & (31) \\
&= \lim_{\beta_1, \ldots, \beta_{j-1}, \beta_{j+1}, \ldots, \beta_n \to +\infty} \int \delta(g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) \cdot \int \partial_\lambda g_j(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot \pi_j^\sigma(\boldsymbol{u}) \cdot p(\boldsymbol{u}) \, \mathrm{d}\boldsymbol{u}. & (32)
\end{aligned}$$

The last transition uses the fact that

$$\lim_{\beta_j \to +\infty} \frac{\beta_j \cdot e^{-g(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot \beta_j}}{(1 + e^{-g(r(\boldsymbol{u}, \boldsymbol{\Lambda})) \cdot \beta_j})^2} = \delta(g(r(\boldsymbol{u}, \boldsymbol{\Lambda}))), \tag{33}$$

in the distributional sense. In addition, we also have (again in distributional sense) that

$$\lim_{\beta_i \to +\infty} \sigma(\beta_i \cdot g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))) = H(g_i(r(\boldsymbol{u}, \boldsymbol{\Lambda}))). \tag{34}$$

This final equation allows us to replace $\pi_j^\sigma(\boldsymbol{u})$ in the final line of Equation 26 with $\pi_j(\boldsymbol{u})$ by repeating the above steps for each index i separately. Hence, we can conclude that our relaxation of $\partial_\lambda P(q)$ is indeed unbiased in the infinite coolness limit. $\square$

F EXPERIMENTAL DETAILS

This section will give detailed DeepSeaProbLog programs, neural network architectures and elaborated figures for each of the experiments present in the main body of the paper. All experiments were run on an RTX 3080 Ti coupled with a Intel Xeon Gold 6230R CPU @ 2.10GHz and 256 GB of RAM, except the LTN results. Note that the optimisation of any hyperparameters, such as learning rate or number of training epochs, was done via a grid search on a separate validation set.

F.1 NESY ATTENTION

Setup details and DeepSeaProbLog program. The full DeepSeaProbLog program for the detection of handwritten years is given in Listing 1. The query `year` is optimised for a different number of samples depending on the experiment. For **(E1)**, we have 28 000 training samples while there are 4000 validation and 8000 test samples. The set of years in the training, validation and test set are disjoint. For **(E2)**, the size of validation and test set is the same as in the case of **(E1)**, but with a training set of 40 000 samples. Here, the set of years in validation and test set are disjoint, but both are a subset of the set of years of the training set.

```
box(Params, B) ~ generalisednormal(Params).
digit(Im, Loc, D) ~ categorical(classifier([Im, Loc]), [0, ..., 9]).

year(Im, Year1, Year2, Year3, Year4) :-
    region(Im, [Y1, Y2, Y3, Y4]), ordered_output([Y1, Y2, Y3, Y4]),
    box(Y1, B1), box(Y4, B4),
    x_diff(0.0, B1, B1diff), B1diff < 0,
    x_diff(1.0, B4, B4diff), 0 < B4diff,
    digit(Im, Y1, D1), digit(Im, Y2, D2), digit(Im, Y3, D3), digit(Im, Y4, D4),
    Year1 =:= D1, Year2 =:= D2, Year3 =:= D3, Year4 =:= D4.

ordered_output([]).
ordered_output([[Mu, Sigma]]).
ordered_output([[Mu, Sigma], H2 | T]) :-
    box([Mu, Sigma], B1), box(H2, B2), x_diff(B1, B2, Bdiff),
    Bdiff < 0, ordered_output([H2 | T]).
```

Listing 1: There is one continuous NDF, `box`, which represents a bounding box as a generalised normal distribution with mean and scale being the center and width of the box, respectively. `digit` is a discrete NDF that denotes the categorical distribution of the digit classifications made by the network `classifier`. `region` is the detection network that predicts the 4 bounding boxes, i.e., the parameters of four instances of `box`. Given these parameters, the predicate `ordered_output` will enforce the spatial constraints that `region` predicts its boxes in order from left to right on the image. It does so by taking the difference of the x coordinate of each subsequent bounding box, which is a 2-dimensional random variable, and employing a '$<$' PCF. Finally, the supervision on the digits of the year is given to the correct bounding box.

Parameters and neural architectures. A schematic overview of the neural architecture used for all different methods can be seen in Figure F.1. The neural baseline simply outputs the four predictions of the classification network and optimises them by minimising the categorical cross-entropy on each digit of the year. In the case of the neural-symbolic methods, the output of both the regression and classification components are used in the logic. DeepSeaProbLog optimises a binary cross-entropy on the probability of `year`, while LTN optimises the MAX-sat objective function. As optimiser, we utilised Adamax [Kingma and Ba, 2015] with its default learning rate of 10^{-3}. DeepSeaProbLog and LTNs were run for 10 epochs, while the neural baseline was given 20 epochs, all with a batch size of 10. This number of epochs proved sufficient for all methods to converge. Interestingly, no special annealing scheme was necessary for this experiment as constant value of 50 for the coolness parameters lead to satisfactory results. All these hyperparameters were determined through a grid search on the validation set.

Additional results and interpretations. Roughly speaking, every 100 iterations took about 25 seconds for DeepSeaProbLog while the neural baseline took around 15 seconds. Given the results and DeepSeaProbLog's satisfactory solution to the problem, the additional computational cost of adding probabilistic logic is worthwhile in this case.

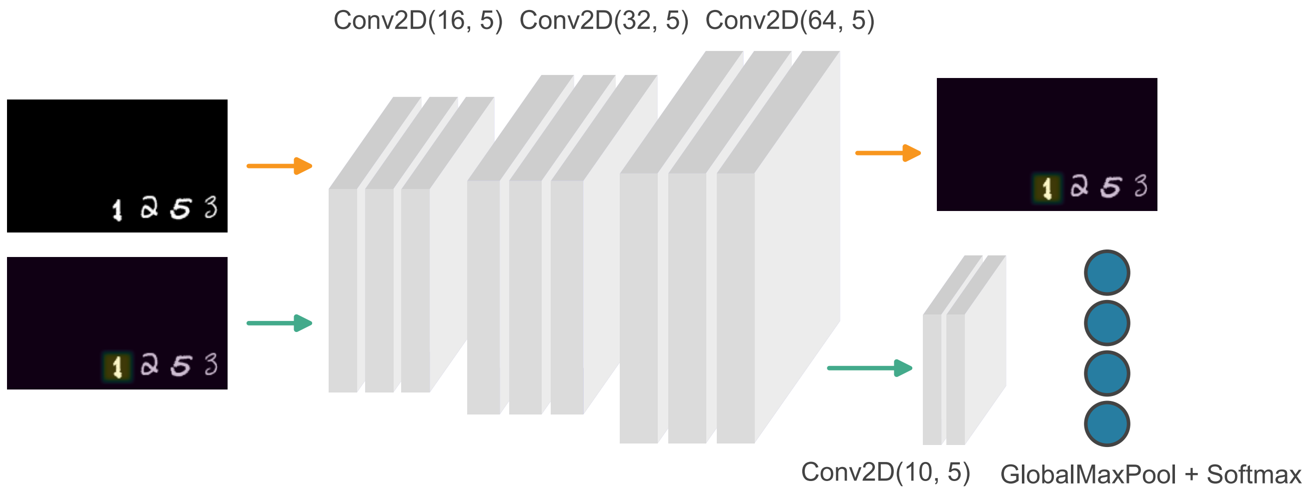

Figure F.1: Overall neural architecture for the dates experiment. Following the orange arrows first, the parameters of 4 generalised normal distributions are predicted for each image. Then, following the green arrows, the images are attenuated separately by each of the 4 distributions and then classified as a digit between 0 and 9 to give the total overall year prediction as an ordered tuple of 4 digits.

F.2 NEURAL HYBRID BAYESIAN NETWORK

Setup details and DeepSeaProbLog program. Our encoding of the neural hybrid Bayesian network is given in Listing 2. The goal is to optimise the neural networks responsible for the classification of `humid` and `cloudy` conditions, as well as the network that predicts the temperature value. Additionally, we explicitly model the noise present on the true temperature labels as a learnable parameter. To achieve this, a set of 1200 triples (`Im1`, `Im2`, `X`) are used as training set, where `Im1` is a CIFAR-10 image belonging to one of the first three classes, while `Im2` belongs to the last two classes. In other words, we use CIFAR-10 images as proxies for real imagery data. `X` is a set of 25 numerical meteorological features sampled from a publicly available Kaggle dataset [Cho et al., 2020]. The label of each triple is the probability that the weather, as described by the correct labels of `humid`, `cloudy` and `temperature`, is good. Computing this probability label is non-trivial in itself. We utilised a large set of 1000 samples to approximate the correct underlying distributions and to obtain an approximate probability label.

```
humid(Im, H) ~ bernoulli(humid_detector(Im)).
cloudy(Im, C) ~ categorical(cloud_detector(Im), [0, 1, 2]).

temperature(X, T) ~ normal(temperature_detector(X), t(_)).
snowy_pleasant ~ beta(11, 7).
rainy_pleasant ~ beta(1, 9)
cold_sunny_pleasant ~ beta(1, 1).
warm_sunny_pleasant ~ beta(9, 2).

rainy(I1, I2) :-
    cloudy(I1, C), C =\= 0, humid(I2, H), H =:= 1.

good_weather(I1, I2, X) :-
    rainy(I1, I2), temp(X, T), T < 0,
    snowy_pleasant > 0.5.
good_weather(I1, I2, X) :-
    rainy(I1, I2), temp(X, T), T >= 0, rainy_pleasant > 0.5.
good_weather(I1, I2, X) :-
    \+rainy(I1, I2), temp(X, T), T > 15, warm_sunny_pleasant > 0.5.
good_weather(I1, I2, X) :-
    \+rainy(I1, I2), temp(X, T), T <= 15, cold_sunny_pleasant > 0.5.

P :: depressed(I1) :-
    cloudy(I1, C), C =:= N, P is N * 0.2.
```

```
enjoy_weather(I1, I2, X) :-
    \+depressed(I1), good_weather(I1, I2, X).
```

Listing 2: The NDFs `humid` and `cloudy` classify a given image as describing humid and cloudy conditions, respectively. `temp` takes a set of 25 numerical features and predicts a mean temperature from those. Note that `t(_)` is ProbLog notation for a single optimisable parameter. Depending on the value of the temperature, 4 different cases of weather and their degree of pleasantness are described by beta distributions. We define `good_weather` as being true if the degree of pleasantness of any case is larger than 0.5. Finally, a person can be `depressed` with probability 0.2 or 0.4 depending on the degree of `cloudy`. Both then determine whether a person can enjoy the weather, if they are not `depressed` and `good_weather` is the case.

Parameters and neural architectures. We utilise simple classifiers (Figure G.3) in the NDFs `cloudy` and `humid`, while the network in the neural predicate `temperature` has three dense layers of size 35, 35 with ReLU activations and 1 with linear activation. Both classifiers share a common set of convolutional layers, requiring the learning of features that generalise to both classification problems. Additionally, the noise on the temperature prediction is modelled explicitly as a learnable TensorFlow variable with an initial value of 10. This choice is not arbitrary, as the initial neural parameter estimate will hover around the middle of the possible temperature values and a choice of 10 as initial standard deviation allows covering the entire range of temperature values with a non-insignificant probability mass. In this way, gradient information across the entire temperature domain can be accumulated during learning. Finally, DeepSeaProbLog was trained for 10 epochs using Adamax with learning rate 10^{-3} and batch size of 10.

Complications. Ideally, simple 0-1 labels of `enjoy_weather` would be more intuitive, as we often do not observe the probability of an event but single cases where it is either true or false. However, our experiments have showed that our small dataset is insufficient to find an optimal solution using such labels in conjunction with the very distant supervision. To show that DeepSeaProbLog is still able to find solutions in cases where the supervision is slightly less distant using only 0-1 labels, we added a different neural hybrid Bayesian network experiment in Section G based on the well-known burglary-alarm example of probabilistic logic.

Additional results and interpretations. We want to stress that learning to predict the right mean temperature from the distant supervision is not straightforward. The only learning signal for the temperature has to pass through PCFs with a very wide range, meaning they do not specify the exact temperature value. Additionally, these PDFs still do not directly influence the supervision of `enjoy_weather`, only `good_weather`. The Gaussian noise that renders the temperature into a continuous random variable only further convolutes the task. We conclude that DeepSeaProbLog can extract meaningful learning signals from reasonably distant supervision.

F.3 NEURAL-SYMBOLIC VARIATIONAL AUTOENCODER

Setup details and DeepSeaProbLog programs. Each data sample consists of 2 regular MNIST digits and the result of their subtraction. The first digit takes the place of the minuend while the second one is interpreted as the subtrahend. The training, validation and test sets had 30 000, 1 000 and 1 000 samples of this form, respectively. Encoding a VAE without additional logic in DeepSeaProbLog is straightforward (Listing 3), while adding logic involves more engineering freedom (Listing 4). We opted for the simplest use of a conditional variational auto-encoder by only using the classified digit as additional input to the decoder. Note that during optimisation, both the VAE and digit classifier are trained jointly.

```
prior(P) ~ normal(0, 1).
latent(Im, L) ~ normal(encoder_net(Im)).

good_image(Image) :-
    prior(P), latent(Im, L), P =:= L,
    decoder_net(L, G), soft_unification(G, Image).
```

Listing 3: Prototypical implementation of a Gaussian VAE in DeepSeaProbLog. A normal prior `prior` is used to regularise a Gaussian latent space modelled by the second NDF by expressing that they should be equal. The decoder component of the VAE is given by `decoder_net` and returns a generated image `G` by sampling the latent space. This generation is self-supervised by soft unifying it with the given image. Note that we do not define the decoder component of the VAE using a delta-distribution `g ~ delta(decoder_net(L))`. While such a definition would strictly comply with our defined syntax, we introduce the easier predicate notation `decoder_net(L, G)` as a form of syntactic sugar.

```
prior(ID, P) ~ normal(0, 1).
digit(Emb, D) ~ categorical(digit_classifier(Emb), [0, ..., 9]).
latent(Im, L) ~ normal(encoder_net(Im)).

good_subtraction(Im1, Im2, Diff) :-
    prior(1, P1), prior(2, P2), latent(Im1, L1), latent(Im2, L2),
    L1 =:= P1, L2 =:= P2, embedding(Im1, E1), embedding(Im2, E2),
    digit(E1, D1), digit(E2, D2), Diff =:= D1 - D2,
    concat(L1, D1, ConditionalL1), concat(L2, D2, ConditionalL2),
    decoder_net(ConditionalL1, G1), decoder_net(ConditionalL2, G2),
    soft_unification(G1, Image1), soft_unification(G1, Image1).
```

Listing 4: Combining subtraction logic with a VAE in DeepSeaProbLog. Each image is encoded into a Gaussian latent space and embedded into a lower-dimensional real space. The latent space is regularised by the standard normal prior while the embedding forms the input to a digit classifier to find which digit is on the image. The two classified digits, which follow a categorical distribution, should subtract to the given value of `Diff`. Finally, the Gaussian latent space and the categorical digits are concatenated into the conditional latent space of the CVAE. The decoder network again samples from this space to construct a generation for both images, which should softly unify with the original images.

Parameters and neural architectures. The NeSy VAE has two main neural components (Figure F.2), one for the VAE itself and another that handles the digit classification used in the subtraction logic. A small set of 256 samples with direct supervision on the digit labels is used to pre-train the classification portion of the overall network to avoid degenerate solutions. All training utilised Adam as optimiser with a learning rate of $\cdot 10^{-3}$ and took 20 epochs using a batch size of 10. The pre-training was given 1 epoch with a batch size of 4.

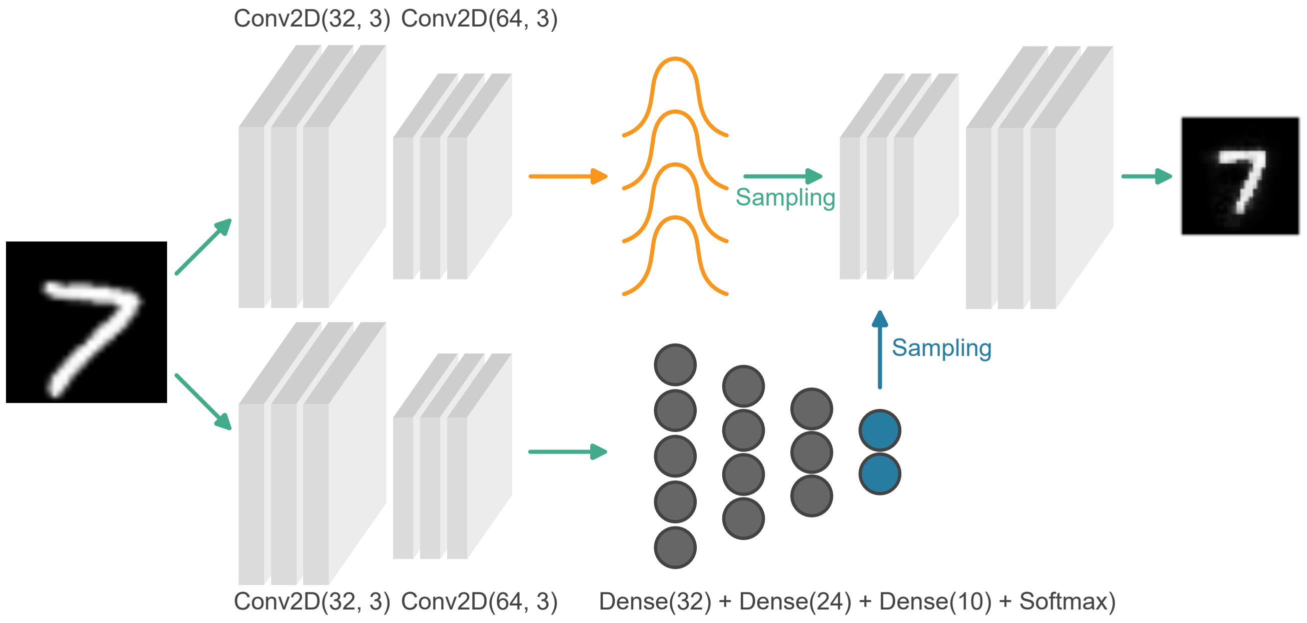
Conv2D(32, 3) Conv2D(64, 3)
Sampling
Sampling
Conv2D(32, 3) Conv2D(64, 3)
Dense(32) + Dense(24) + Dense(10) + Softmax)

Figure F.2: VAE encoder-decoder architecture. The decoder is equivalent to the transpose of the encoder. All layers use ReLU activation functions, except the final convolutional one, which applies a hyperbolic tangent.

Complications. Regular Gaussian VAE optimisation has two components: a Kullback-Leibler (KL) divergence term and a reconstruction loss term. Since DeepSeaProbLog requires probabilistic values, i.e., between 0 and 1, a probabilistic translation of these terms is necessary for optimisation in DeepSeaProbLog. The KL divergence term compares the latent distribution of the VAE to a standard normal prior and can as such be replaced by a `=:=` comparison in the logic. The reconstruction loss is chosen to be the exponentiation of a negated average L_1 loss function, as it yields a value between 0 and 1 that can be interpreted as the probability that two images match. Specifically, the loss between two such images $\boldsymbol{I_1}, \boldsymbol{I_2} \in \mathbb{R}^{768}$ is given by

$$\exp\left(-\frac{1}{768}\sum_{i=1}^{768}|I_{1i} - I_{2i}|\right). \tag{35}$$

The latter can be interpreted as a form of soft unification [Rocktäschel and Riedel, 2017], which is why we denote it by the predicate `soft_unification`.

Additional results and interpretations. Emphasis has to be put on the flexibility of generation in DeepSeaProbLog, as the generation of digits can be carried out in a range of different contexts without further optimisation. One only needs to write a query describing that logical context. The query that yields an image of both a left and right digit that subtract to a given value is given in Listing 5. The conditional query that generates an image of a right digit given an image of the left digit and their difference value is given in Listing 6.

```
generate_subtraction(G1, G2, Diff) :-
    member(D1, [0, ..., 9]), member(D2, [0, ..., 9]),
    prior(1, P1), prior(2, P2), Diff =:= D1 - D2,
    concat(P1, D1, ConditionalL1), concat(P2, D2, ConditionalL2),
    decoder_net(ConditionalL1, G1), decoder_net(ConditionalL2, G2).
```

Listing 5: The logic finds all possible combinations for `D1` and `D2` that meet the subtraction evidence `Diff` and concatenates these to a standard normal prior component into the conditional latent space. The decoder then generates images from a sample of this space.

```
generate_left(RightIm, Diff, LeftG) :-
    member(D1, [0, ..., 9]), embedding(RightIm, RightE), digit(RightE, RightD),
    Diff =:= LeftD - RightD, latent(RightIm, RightL),
    concat(RightL, LeftD, LeftCondL), decoder_net(LeftCondL1, LeftG).
```

Listing 6: Given an image of the right digit and a difference value, we generate an image of the left digit. The right's image is classified such that the logic can find the value of `LeftD` that meets the given difference. By attaching that value to the Gaussian latent space of the right digit, the VAE can generate an image of the correct left digit in the 'style' of the right one.

G ADDITIONAL EXPERIMENT

An additional experiment was performed to show the promise of discrete-continuous neural probabilistic logic programming. It is similar to the neural hybrid Bayesian network, but with more practical 1-0 query supervision.

G.1 NEURAL-CONTINUOUS BURGLARY ALARM

Setup details and DeepSeaProbLog program. The neural-continuous burglary alarm (Listing 8) extends the classic example from Bayesian network literature (Listing 7).

```
0.1 :: earthquake.
0.3 :: burglary.
0.9 :: hears.

0.7 :: alarm :- earthquake.
0.9 :: alarm :- burglary.
```

```
calls :- alarm, hears.
```

Listing 7: Classical burglary-alarm ProbLog program. Three probabilistic facts `earthquake`, `burglary` and `hears` are given with their probabilities. A neighbour calls when hearing an alarm, while an alarm can go off because of an earthquake or a burglary.

Each data sample is a triple (E, B, L), where E can be an MNIST digit 0, 1 or 2 while B can be an MNIST 8 or 9. Values for E of 0, 1 and 2 correspond to no earthquake, a mild earthquake or a heavy earthquake respectively. If B is an MNIST 8, then there is no burglary. If it is 9, then there is a burglary. L can have either the value 0 or 1, indicating whether the neighbour called or not. Our dataset contains 12 000 such triples for training, while having 1 000 for validation and 2000 for testing purposes. Obtaining the weak supervision L is done by sampling according to the true probability of calling given the input To compute this true probability, a single sample is taken from the neighbour's true distribution. This true distribution has respective means of 6 and 3 for the horizontal and vertical Gaussian while both directions have a standard deviation of 3. Additionally, there are two possible ways to express that the distance of the neighbour should be smaller than 10 distance steps before hearing the alarm. One can use either the squared distance or the true distance in the rule `hears`. A separation is often maintained in the weighted model integration literature [Zuidberg Dos Martires et al., 2019] between comparison formulae that are polynomial and those that are generally non-polynomial. To illustrate that DeepSeaProbLog can deal with both classes of formulae, we will perform experiments for both the squared distance (polynomial, Listing 8) and the true distance (non-polynomial, Listing 9). Both these functions are implemented in Python and DeepSeaProbLog allows them to be easily imported as built-in predicates.

```
earthquake(Im, E) ~ categorical(earthquake_net(Im), [0, 1, 2]).
burglary(Im, B) ~ categorical(burglary_net(Im), [8, 9]).

neighbour(N) ~ normal([t(μ_x), t(μ_y)], [t(σ_x), t(σ_y)]).

hears :-
    neighbour(N), squared_distance(0, N, D), D < 100.

P :: alarm(Im1, _) :-
    earthquake(Im1, E), E =:= N, P is N * 0.35.
0.9 :: alarm(_, Im2) :-
    burglary(Im2, B), B =:= 9.

calls(Im1, Im2) :-
    alarm(Im1, Im2), hears.
```

Listing 8: Our extension of the burglary alarm example has two categorical NDFs that model the chance of an earthquake and a burglary given an image. Additionally, whether the neighbour can hear the alarm if it goes off depends on their spatial distribution, which is modelled as a two-dimensional Gaussian distribution. This distribution is randomly initialised and its parameters need to be optimised.

```
hears :-
    neighbour(N), distance(0, N, D), D < 10.
```

Listing 9: Using the true distance in the `hears` predicate as a case of a non-polynomial comparison formula.

Parameters and neural architectures. The complete neural architecture of both the earthquake and burglary classifiers is given in Figure G.3. In addition to the neural parameters in these networks, four independent parameters are present in the program. These are used as the means and standard deviations for the neighbour's spatial distribution and are randomly initialised. Specifically, the means are sampled uniformly from the interval $[0, 10]$ while the standard deviations were sampled from $[2, 10]$. All optimisation was performed using regular stochastic gradient descent with a learning rate of $8 \cdot 10^{-2}$ for two epochs using a batch size of 10.

Complications. Because of the difference in nature between the parameters in the neural networks and the four independent parameters in the Gaussian distribution, the latter required a boosted learning rate to provide consistent convergence.

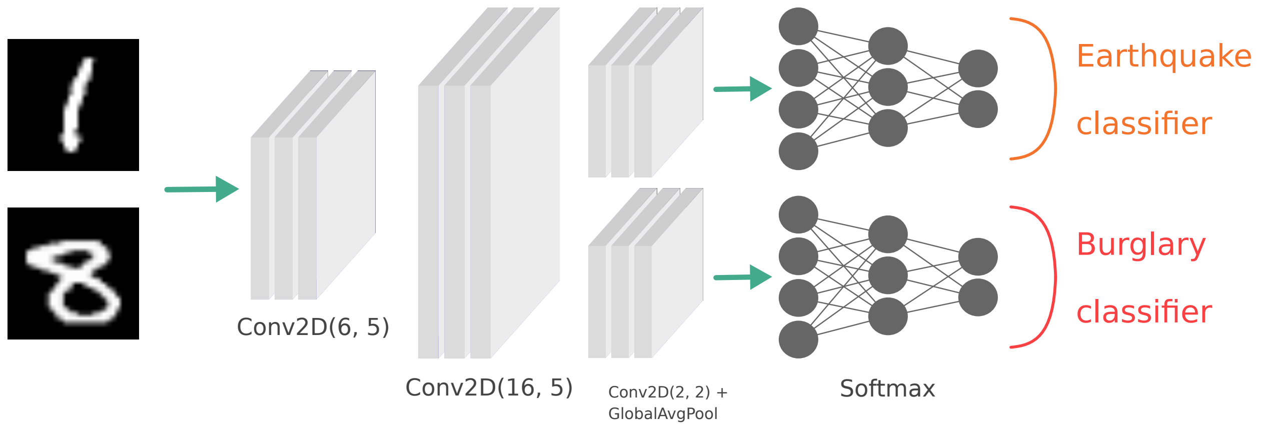

Figure G.3: Overview of the architecture of the earthquake and burglary networks. Both share two convolutional layers, but each specific network applies its own final convolutional layer followed by a global average-pooling operation with softmax activation. All other activation functions are ReLUs.

Specifically, the gradients for these four parameters were multiplied by a value of 20, which was found by a hyperparameter optimisation on the validation set.

Results and interpretation. Initial learning progress of the neural networks seems volatile (Figure G.4), which is likely due to the unoptimised state of the neighbour's spatial distribution. Two epochs of training proves to be sufficient to optimise both the neural detectors and the neighbour's distribution. The 4 parameters of the neighbour's distribution do not converge to the true values, which is to be expected as their supervision is underspecified. However, they do converge to values that result in PCF probabilities that are close to those of the true underlying distribution. All in all, three conclusions can be drawn. First, this experiment indicates that DeepSeaProbLog is capable of jointly optimising neural parameters and independent, distributional parameters. Second, DeepSeaProbLog seems to be able to fully exploit both polynomial and more general non-polynomial comparison formulae. It shows the strength of our approximate approach, as exact methods often fail to efficiently deal with non-polynomial formulae [Zuidberg Dos Martires et al., 2019]. Third, DeepSeaProbLog can deduce meaningful probabilistic information from weak labels. Indeed, in order to optimise the neural detectors and the neighbour's distribution, DeepSeaProbLog has to aggregate meaningful update signals from the 0-1 labels across the given training data set to approximate the underlying probability of `calls`.

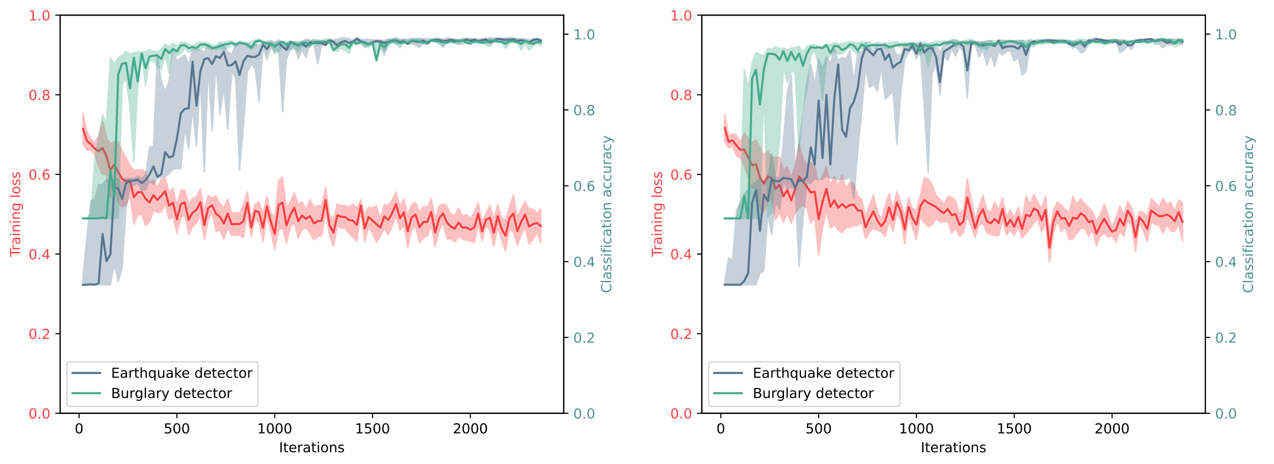

Figure G.4: Evolution of the training loss and validation accuracy of the neural 'earthquake' and 'burglary' detectors. For both squared (left) and true distance (right), the discrete supervision seems to be sufficient to facilitate meaningful learning.

H LIMITATIONS

The main limitation of DeepSeaProbLog is one that it inherits from probabilistic logic in general, computational tractability. Efficiently representing a probabilistic logic program is done via knowledge compilation, which is $\#P$-hard. Once the probabilistic program is knowledge compiled, evaluating the compiled structure is linear in the size of this structure. Inference remains linear in the size of the compiled structure after the addition of continuous random variables as all samples can be run in parallel with the current inference algorithm.

Although our sampling strategy is efficient in the sense that it is linear in the number of samples, uses the advanced inference techniques of Tensorflow Probability to effectively sample higher dimensional distributions, and it can be executed in parallel for each sample, it remains ignorant of the comparison formulae that are approximated. More intricate inference strategies exist within the field of weighted model integration [Morettin et al., 2021], yet they currently lack the differentiability property to be integrated in DeepSeaProbLog's gradient-based optimisation. Conversely, our examples illustrate that our rather naive strategy is sufficient to solve basic tasks. It is still an open question how to perform successful joint inference and gradient-based learning under general comparisons.

Orthogonal to the estimation of the integral during inference, exact knowledge compilation also prevents the scaling of DeepSeaProbLog to larger problem instances. Approximate knowledge compilation is the field of research that deals with tackling this issue. While it contains interesting recent work [Fierens et al., 2015, Huang et al., 2021, Manhaeve et al., 2021b], it was highlighted by Manhaeve et al. that the introduction of the neural paradigm does lead to further complications. As such, we opted for exact knowledge compilation, but it has to be noted that we will be able to benefit from any future advances in the field of approximate inference. Alternatively, different semantics [Winters et al., 2022] can simplify inference, but they lead to a degradation of expressivity of the language.

A potential future avenue for scaling up DeepSeaProbLog inference would be the use of further continuous relaxation schemes. More specifically, replacing discrete random variables with relaxed categorical variables [Maddison et al., 2017, Jang et al., 2017] might allow us, for instance, to forego the knowledge compilation step while still being able to pass around training signals

References

Martín Abadi. Tensorflow: learning functions at scale. In *International Conference on Functional Programming*, 2016.

Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *IJCAI*, 2015.

Dongjin Cho, Cheolhee Yoo, Jungho Im, and Dong-Hyun Cha. Comparative assessment of various machine learning-based bias correction methods for numerical weather prediction model forecasts of extreme air temperatures in urban areas. *Earth and Space Science*, 2020.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 2015.

Harley Flanders. Differentiation under the integral sign. *The American Mathematical Monthly*, 1973.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming*, 2011.

Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. *NeurIPS*, 2021.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparametrization with gumble-softmax. In *ICLR*, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017.

Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 2021a.

Robin Manhaeve, Giuseppe Marra, and Luc De Raedt. Approximate inference for neural probabilistic logic programming. In *KR*, 2021b.

Paolo Morettin, Pedro Zuidberg Dos Martires, Samuel Kolb, and Andrea Passerini. Hybrid probabilistic inference with logical and algebraic constraints: a survey. In *IJCAI*, 2021.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS*, 2019.

Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Learning with algorithmic supervision via continuous relaxations. *NeurIPS*, 2021.

Fabrizio Riguzzi and Terrance Swift. Well–definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and practice of logic programming*, 2013.

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. *NeurIPS*, 2017.

Laurent Schwartz. Théorie des distributions à valeurs vectorielles. i. In *Annales de l'institut Fourier*, 1957.

Thomas Winters, Giuseppe Marra, Robin Manhaeve, and Luc De Raedt. Deepstochlog: Neural stochastic logic programming. In *AAAI*, 2022.

Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Exact and approximate weighted model integration with probability density functions using knowledge compilation. In *AAAI*, 2019.