dPASP: A Comprehensive Differentiable Probabilistic Answer Set Programming Environment For Neurosymbolic Learning and Reasoning

Renato Lui Geh [1] **Jonas Gonçalves** [1] **Igor Cataneo Silveira** [1] **Denis Deratani Mauá** [1] **Fabio Gagliardi Cozman** [2]

Abstract

We present dPASP, a novel declarative probabilistic logic programming framework for differentiable neuro-symbolic reasoning. The framework allows for the specification of discrete probabilistic models with neural predicates, logic constraints and interval-valued probabilistic choices, thus supporting models that combine low-level perception (images, texts, etc), common-sense reasoning, and (vague) statistical knowledge. To support all such features, we discuss the several semantics for probabilistic logic programs that can express nondeterministic, contradictory, incomplete and/or statistical knowledge. We also discuss how gradient-based learning can be performed with neural predicates and probabilistic choices under selected semantics. We then describe an implemented package that supports inference and learning in the language, along with several example programs. The package requires minimal user knowledge of deep learning system's inner workings, while allowing end-to-end training of rather sophisticated models and loss functions.

1. Introduction

Answer Set Programming (ASP) (Gebser et al., 2012; Lifschitz, 2019) is a powerful declarative paradigm for specifying domain knowledge by means of logic programming. For example, the following program very intuitively describes the causal relationships between stress, smoking and peer pressure (Fierens et al., 2015).

```
smokes(X) :- stress(X).
smokes(X) :- influences(Y,X), smokes(Y).
```

The program can be extended with a database of known facts

such as `influences(anna,bill)` and `stress(anna)`, and used to conclude that `smokes(bill)` must be true (according to some selected semantics).

While powerful, ASP cannot cope with uncertainty, that abounds in data-driven situations. For instance, suppose that we only known (or are only willing to ascertain) that Anna influences Bill with probability 0.8:

```
0.8::influences(anna,bill).
```

Such probabilistic facts appear quite naturally when they are the output of perception models. In particular, when differentiable models such as deep neural networks are used, as is the case of image, text and speech recognition.

Once probabilistic facts are allowed, we enter the realm of Probabilistic Answer Set Programming (PASP) (Cozman & Mauá, 2020); and when the probabilistic facts are linked to the output of neural probabilistic classifiers, we name the resulting framework Neural-Probabilistic Logic Programming (NPLP). Essentially, NPLP provides a tight coupling of low-level reasoning models (e.g., image recognition) and high-level reasoning (e.g., planning). By exploiting existing domain knowledge, NPLP allows, among other things, end-to-end gradient-based weakly supervised learning of neural models, thus providing an effective implementation for neurosymbolic reasoning (Yang et al., 2020; Manhaeve et al., 2021).

An important and mostly overlooked component of NPLP systems is the interface connecting real-world objects, neural network models and logic programming languages. Existing implementations of NPLP such as DEEPPROBLOG (Manhaeve et al., 2021) and NEURASP (Yang et al., 2020) require the user to write the "glue" between the neural-probabilistic logic language and the deep learning framework themselves, a task which is not always straightforward. These frameworks are also limited: DEEPPROBLOG forbids negative cycles and nondeterministic knowledge; NEURASP disallows probabilistic facts; and neither of them deals with contradictions or vague uncertain knowledge.

In order to fill those gaps, we introduce dPASP, a new NPLP framework that features a large set of ASP constructs, can learn from both probabilistic facts and neural atoms, and

---
[1]Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, Brazil. [2]Escola Politécnica, Universidade de São Paulo, São Paulo, Brazil. Correspondence to: Renato Lui Geh <renatogeh@gmail.com>, Denis Deratani Mauá <ddm@ime.usp.br>, Fabio Gagliardi Cozman <fgcozman@usp.br>.

supports several semantics, from standard maximum entropy probability measures to richer semantics that can accommodate contradictions and partial specification of probabilistic knowledge.

2. Background

We review some background knowledge on logic and probabilistic logic programming. We assume the reader has some familiarity with the terminology and basics of logic programming (Gebser et al., 2012), and focus on less common concepts such as L-stable semantics, probability models and the credal L-stable semantics (Rocha & Cozman, 2022).

2.1. Logic Programming

A logic program is a finite set of disjunctive rules of the form:

$$h_1; \ldots; h_k \text{ :- } b_1, \ldots, b_n, \text{ not } b_{n+1}, \ldots, \text{ not } b_{n+m}.$$

where each h_i and b_j is an atom and not denotes default negation. We say that $\text{head}(r) = \{h_i\}_{i=1}^{k}$ is the *head* of the rule, while the atoms right of :- are the *body*, denoted $\text{body}(r)$; sets $\text{body}^-(r)$ and $\text{body}^+(r)$ denote, resp., the sets of negated (i.e., preceded by not) and non-negated atoms in $\text{body}(r)$. The rule is disjunctive if the head has two or more atoms. It is called an integrity constraint if the head is empty, and normal if the head is a singleton. And it is a fact if it is normal and the body is empty.

The *Herbrand base* of a program is the set formed by all ground atoms that can be built using predicate names and constants in the program. The *grounding* of a program is the propositional program obtained by grounding each rule, that is, replacing variables with constants from the Herbrand base in every consistent way. The semantics of a program with variables is the semantics of its grounding.

Let t, f and u denote ground atoms which do *not* occur in a program L. A three-valued interpretation I, also called a partial interpretation, is a function from the atoms in the Herbrand base to $\{0, 0.5, 1\}$ such that $I(t) = 1$, $I(f) = 0$, $I(u) = 0.5$ We define $I^t = \{A \mid I(A) = 1\}$, $I^f = \{A \mid I(A) = 0\}$ and $I^u = \{A \mid I(A) = 0.5\}$ as the sets of true, false and undefined atoms, respectively, according to I. An interpretation is *total* if it assigns every atom (other than u) to either true or false. The interpretation of the positive body of a rule r is the minimum over the value of its atoms, $I(\text{body}^+(r)) = \min\{I(A) \mid A \in \text{body}^+(r)\}$. The value of the negative body is the minimum of the complements of the atom's values, $I(\text{body}^-(r)) = \min\{1 - I(A) \mid A \in \text{body}^-(r)\}$. The interpretation of the body is $I(\text{body}(p)) = \min\{I(\text{body}^+(r)), I(\text{body}^-(r))\}$. Last, the interpretation of the rule's head is the maximum of its atoms, $I(\text{head}(r)) = \max\{I(A) \mid A \in \text{head}(r)\}$.

An interpretation *satisfies* a rule r if and only if $I(\text{head}(r)) \geq I(\text{body}(r))$. I is a *model* of a program if it satisfies all of its rules. We define a partial order $\leq$ (reflexive, antisymmetric and transitive) of interpretations as: $I_0 \leq I_1$ if and only if $I_0(A) \leq I_1(A)$ for all A. A model I is minimal if there is no model $I' \leq I$ such that $I \neq I'$. If I is total then it is minimal if and only if I^t is $\subseteq$-minimal. Note that by (3) an interpretation that assigns undefined to all atoms is always a model as long as the program can be rewritten so it does not contain disjunctive rules nor integrity constraints. Thus, since $\leq$ is a partial order and the Herbrand base is finite, every normal program admits one or more minimal models (Przymusinski, 1991). This is in contrast to complete (i.e., true/false) semantics, for which a normal program might have none, one or multiple minimal models.

The stability of a model I is connected to the notion of the program's *reduct* w.r.t. I, written P/I, obtained by the modified Gelfond–Lifschitz transformation (Przymusinski, 1991). The transformation operates on each atom $A \in \text{body}^-(r)$ in the negative body of a rule $r \in L$ and replaces it by the atom t, f or u corresponding to its semantics in I. Formally, it replaces A with: (i) t if $A \in I^f$; or (ii) f if $A \in I^t$; or (iii) u if $A \in I^u$. We say that I is a *partial stable model* of L if I is a minimal model of L/I. This is the partial stable model semantics (P-stable) for logic programs with disjunctions and default negation (Przymusinski, 1991). A stable model I_0 is *least undefined* if there is no other stable model I_1 with $I_0^u \subset I_1^u$. That is, I is least undefined if there is no other stable model that defines (as true or false) more atoms than it. Then we say that I is a least undefined stable model of L, or L-stable model, for short. This is finally the L-stable model semantics of disjunctive logic programs (Saccà & Zaniolo, 1997).

2.2. Probabilistic Logic Programming

Probabilistic logic programming extend logic programs with *annotated disjunctive rules* (ADR) of the form

$$p_1 :: a_1; \ldots; p_k :: a_k \text{ :- } b_1, \ldots, b_n, \text{not } b_{n+1}, \ldots, \text{not } b_{n+m}.$$

where $p_1, \ldots, p_k$ are nonnegative real values whose sum is equal to one. We also allow ADRs where $\sum_i p_i < 1$. Semantically, this is a syntactic sugar where the head is extended by atom f (which is never true) with probability $1 - \sum_i p_i$. For example, a probabilistic fact is an ADR with a singleton head and empty body, written as:

$$\theta :: a.$$

It is equivalent to the ADR

$$\theta :: a; \ 1 - \theta :: f.$$

A probabilistic logic program is a finite set of ADRs and (non-probabilistic) rules. The semantics of probabilistic logic programs extends that of Sato's distribution semantics (Sato, 1995), in which independent random choices induce logic programs.

A *total choice* independently selects one atom of the head of each ADR in a probabilistic logic program. Note that a total choice can select different atoms for rules with the same head (and different bodies). Each total choice θ induces a logic program where each ADR r is transformed into a normal rules

```
h :- b_1, ..., not b_m.
```

where h is the atom selected by the total choice for rule r, and $b_1, \ldots, b_m$ are the body.

An interpretation of a probabilistic logic program is simply an interpretation of the logic program obtained by dropping probabilities (i.e., turning ADRs into dijsunctive rules). Let θ denote both a total choice and its induced logic program, and $\Gamma(\theta)$ denote the selected models of θ, according to some semantics. A *probability model* is a probability measure $\mathbb{P}$ over the interpretations I of the program such that (i) $\mathbb{P}(I) > 0$ if and only if I is in $\Gamma(\theta)$, (ii) $\mathbb{P}(\Gamma(\theta)) = \prod_r p_r$, where the product runs over all ADRs in the probabilistic logic program and p_r is the probability annotating $\theta(r)$ in rule r. The *maximum entropy* probability model, or max-ent model for short, is the probability model that splits $\mathbb{P}(\Gamma(\theta))$ evenly over each stable model, that is, $\mathbb{P}(I) = \sum_{\theta : I \in \Gamma(\theta)} (1/|\Gamma(\theta)|)$ for any stable model I.

The *credal semantics* assigns to each atom a a pair of lower and upper probabilities such that $\underline{\mathbb{P}}(a) = \min_{\mathbb{P}} \mathbb{P}(a)$ and $\overline{\mathbb{P}}(a) = \max_{\mathbb{P}} \mathbb{P}(a)$, where the optimizations are over the set of probability models (which is closed and convex). The max-ent semantics assigns to a the probability of the corresponding max-ent probability model. Note that by definition we have that $\mathbb{P}_{\text{max-ent}}(a) \in [\underline{\mathbb{P}}(a), \overline{\mathbb{P}}(a)]$.

Note that the previous probabilistic semantics are agnostic with respect to the logic semantics used to produce $\Gamma(\theta)$, requiring only that it is non-empty for each θ. In the literature, such a set is more commonly selected as the set of stable models (Cozman & Mauá, 2020) or well-founded models (Fierens et al., 2015).

As we will discuss later, dPASP supports the combination of either the stable, partial, L-stable or SMPROBLOG semantics, for the logic part, and the credal or max-ent semantics, for the probabilistic part. The result is thus one of eight different semantics, to be selected by the user and according to the task at hand. Not every feature is however available for each selected semantics: for instance, learning under credal or L-stable semantics remains an open problem.

3. The dPASP Framework

We now describe the language as well as inference and learning routines of the dPASP framework.

3.1. Language

dPASP extends CLINGO's syntax (Gebser et al., 2017) to neural-probabilistic logic programs by including annotated disjunctive rules and *neural annotated disjunctive rule* (NADR). The latter are ADR whose probabilities are set by a(n externally defined) neural network.

The neural networks that parametrize NADRs are specified using standard deep learning frameworks such as PyTorch. To facilitate the integration between the probabilistic logic program and the deep learning framework, dPASP allows the embedding of Python code within the program via the `#python` guard. Code within this guard is executed and all functions declared are available for use within special predicates in the program. As an example, consider the following `#python` code prototype for constructing a neural network in PyTorch for MNIST.

```python
#python
def net(): return ... # Neural network.
# (i=1)st or (i=2)nd half of train set as tensor.
def mnist_tr(i): return ...
# (i=1)st or (i=2)nd half of test set as tensor.
def mnist_te(i): return ...
#end.
```

The specification of the interface between raw data (that is fed to the neural network in the python code part) and program constants is managed by a special rule of the form

```
atom(x) ~ test(@arg1), train(@arg2).
```

In that rule, `atom` is a user-defined one-place predicate name used to represent an object fed into the neural network, `x` is a constant identifying a particular object (since the same network can be used several times in the same program) and `test` and `train` are reserved predicates, whose arguments are either paths to CSV files or Python functions as defined in `#python`. As an example, consider the example of adding two digit images classified by neural networks, as described in Manhaeve et al. (2021); here, the network's inputs are images, with each grounding of the NADR representing one of the two digits in the sum, and different images in the train and test sets.

```
input(0) ~ test(@mnist_tr(0)), train(@mnist_te(0)).
input(1) ~ test(@mnist_tr(1)), train(@mnist_te(1)).
```

Note that `0` and `1` are arbitrary constants used to identify the two distinct inputs of the same neural network. These

constants are associated with pairs of images taken either from the test or the train set, depending on the task being solved.

A NADR's head must contain a single (possibly non-ground) atom of the form `l::f(X, {v₁,…,vₖ})`, where `l` is either a `?`, indicating the neural ADR is learnable, or a `!`, in which case its parameters are fixed during learning. Variable `X` is used to ground the NADR according to its body, while $v_1, \ldots, v_k$ are the possible values the annotated disjunctive rule may take. Optionally, an interval may be passed as a shorthand; for instance, `0..9` is equivalent to writing out all digits from 0 to 9. The network in the NADR is embedded by either passing a function declared in `#python` or a GitHub repository in PyTorch Hub format. If the NADR is learnable, then we may optionally pass parameters to the PyTorch optimizer via the `with` operator. Finally, the NADR must then be declared with the data predicate as one of the subgoals in its body. In our running MNIST addition example, the neural predicate `digit` must cover all digits from 0 to 9.

```
?::digit(X, {0..9}) as @net :- input(X).
```

Data embedded into special data predicates (e.g. `input(0)` and `input(1)` in the previous example) are passed as PyTorch tensors to the neural networks, allowing for efficient parallelization in the CPU or GPU.

Regular ADRs are declared in a similar fashion to NADRs in dPASP, except they allow setting an initial probability value for the optimization when the rule is learnable. For example:

```
0.5?::h₁; ?::h₂; ?::h₃ :- b₁, b₂, ..., b₄.
```

When no initialization value is given, like in the case of h_2 and h_3, dPASP uniformly distributes the remaining mass to the rest of the atoms. dPASP also supports credal facts in order to specify a credal interval for imprecise inference, where

```
[0.2, 0.7]::f.
```

indicates `f` may take a probability as low as `0.2` and as high as `0.7`.

Purely logic rules may contain arithmetic operations and comparisons over variables of annotated disjunctive rules as long as they are safe and appear in the body.

```
sum(Z) :- digit(0, X), digit(1, Y), Z=X+Y.
both_even :- digit(0, X), digit(1, Y), X\2=0, Y\2=0.
```

Querying a partial interpretation (possibly conditioned on another interpretation) is done through the special directive `#query`.

```
#query digit(0, 4).          % P(digit(1, 4))
#query sum(8)|not both_even. % P(sum(4)|¬both_even)
```

The `undef` keyword may be used within `#query` for querying the probability of an atom being undefined under the L-stable or partial semantics. Each query returns either a precise probability if under the max-entropy semantics, or a pair of probabilities encoding lower and upper bounds if under the credal semantics.

In order to define the semantics of the program, a special directive `#semantics` may be used. It must receive at least one of the supported logic or probabilistic semantics, i.e. stable model, partial, L-stable, max-entropy or credal semantics. If none is given, dPASP defaults to the credal and stable models semantics. If only one is given, the missing one is set to its respective default semantics.

```
#semantics lstable, maxent. % L-stable; max-entropy.
#semantics credal, partial. % Credal; partial.
#semantics stable.          % Stable; credal.
#semantics maxent.          % Max-entropy, stable.
```

The `#learn` directive specifies the learning procedure to take place in the program, and receives as parameters the training dataset with the observed atoms, as well as the learning parameters such as learning rate, number of epochs and batch size.

```
#learn @observed_atoms, lr = 0.001, niters = 5.
```

3.2. Inference

The most typical inference one draws with neuro-probabilistic logic programming models is to compute the probability of a query atom, possibly conditional on some evidence. That is, if $\mathbf{q} = \{q_1, \ldots, q_m\}$ and $\mathbf{e} = \{e_1, \ldots, e_n\}$ are disjoint set of literals, then we are usually interested in computing

$$\mathbb{P}(\mathbf{q}|\mathbf{e}) = \frac{\sum_{I \models \mathbf{q}, \mathbf{e}} \mathbb{P}(\mathbf{q}, \mathbf{e})}{\sum_{I \models \mathbf{e}} \mathbb{P}(\mathbf{e})} \, ,$$

for some or all probability models $\mathbb{P}$.

dPASP provides exact inference by enumerating total choices and using CLINGO's solver to enumerate all models for each induced ASP program. Performing exact inference by that exhaustive approach, be it under the max-ent or credal semantics, limits the scalability of inference to programs with few NADRs and ADRs.

More scalable approximate inference based on knowledge compilation (Totis et al., 2021), sampling (Tuckey et al., 2021; Azzolini et al., 2023) and variational methods are planned features for future versions of dPASP, which is currently in early stage development.

3.2.1. Partial, L-stable and SMProbLog semantics

Internally, dPASP only accepts the stable model semantics when performing inference or learning. To enable support to different semantics, we implement translation procedures to the stable model semantics.

The partial semantics in dPASP is implemented via the translation described in (Janhunen et al., 2006). In a nutshell, dPASP creates an auxiliary atom and rule for each non-probabilistic atom in the program and duplicates logic rules in order to allow undefined values for non-probabilistic atoms. The L-stable semantics is implemented by checking, at each total choice, if there exists a stable model for the program: in the positive case, dPASP performs inference over the stable models of such a program, otherwise it queries from the translated program's partial stable models.

We also implement the SMProbLog semantics, introduced in Totis et al. (2021). The main difference between the L-stable and SMProbLog's semantics is how to deal with undefined atoms. While in L-stable a model may contain undefined, true and false atoms, in SMProbLog, if an atom is set to undefined in a model, then all atoms in this model must also be undefined.

3.2.2. Maximum entropy semantics

If the max-ent semantics is selected, it is sufficient to simply add up the (uniform) probabilities of each model that is consistent with the query; this is the same procedure done in Yang et al. (2020). More formally, the probability of some observation O under the max-ent semantics is given by

$$\mathbb{P}(O) = \sum_{\theta \in \Theta} \mathbb{P}(\theta) \cdot \frac{N(I_\theta \models O)}{N(\theta)}, \qquad (1)$$

where Θ is the set of all total choices, and $N(I_\theta \models O)$ and $N(\theta)$ return respectively the number of stable models that are consistent with both θ and O, and the number of stable models consistent with θ.

dPASP computes $N(I_\theta \models O)$ and $N(\theta)$ by calling CLINGO's solver and counting, for each total choice θ, how many models are consistent with observation O and how many models are in total. The probability $\mathbb{P}(\theta)$ is easily computable by simply multiplying the probabilities of each probabilistic and neural component (i.e. probabilistic and neural facts, rules and annotated disjunctions), as we assume them to be marginally independent from each other.

3.2.3. Credal semantics

For the credal semantics, one is interested in the interval of all probabilities $\mathbb{P}(\mathbf{q}|\mathbf{e})$ obtained by some probability model. That interval can be described by its lower and upper values, which in dPASP are obtained by the exact algorithm

described in Cozman & Mauá (2020). In short, given query $\mathbf{q} = \{q_1, \ldots, q_m\}$ and evidence $\mathbf{e} = \{e_1, \ldots, e_n\}$ literals, we compute the lower $\underline{\mathbb{P}}(\mathbf{q}|\mathbf{e})$ and upper $\overline{\mathbb{P}}(\mathbf{q}|\mathbf{e})$ probabilities by iterating over each total choice θ and counting the models where (a) every model satisfies both $\mathbf{q}$ and $\mathbf{e}$, (b) some model satisfies both $\mathbf{q}$ and $\mathbf{e}$, (c) every model satisfies $\mathbf{e}$ but does not satisfy some value in $\mathbf{q}$, and (d) some model satisfies $\mathbf{e}$ but does not satisfy some value in $\mathbf{q}$. The credal interval is then $[0, 0]$ if $b + c = 0$ and $d > 0$, $[1, 1]$ if $a + d = 0$ and $b > 0$, and $[a/(a + d), b/(b + c)]$ otherwise.

Credal facts in dPASP are only available when the credal semantics is selected. To perform inference with credal facts, dPASP constructs four multilinear polynomials corresponding to a, b, c and d; each term is a total choice θ, each coefficient is the probability of θ, and variables in the polynomial are x if $X = 1$ in θ or $1 - x$ otherwise. The domain of the polynomial is the cartesian product of all pairs of lower and upper probabilities in credal facts. The functions $a(\mathbf{x})/(a(\mathbf{x}) + d(\mathbf{x}))$ and $b(\mathbf{x})/(b(\mathbf{x}) + c(\mathbf{x}))$ are then optimized in order to find the two global minimum and maximum respectively, with the first amounting to the lower and the second the upper probabilities of the queries.

3.3. Parameter learning

dPASP currently implements three parameter learning rules for the max-ent stable model semantics: (i) a fixed-point learning procedure for non-neural programs, (ii) a Lagrange multiplier derivation for gradient ascent, and (iii) an implementation of NeurASP's learning procedure (Yang et al., 2020). How to learn the parameters of programs in partial or least undefined stable model semantics either under the max-ent or credal semantics is an open problem.

We now describe the first two learning rules, which as far as we are aware, are novel in the literature. Both rules provide rules for maximizing the log-likelihood $\mathcal{L}(\mathbf{O}) = \sum_{O \in \mathbf{O}} \log \mathbb{P}(O)$ of a set of observations $\mathbf{O}$ with respect to the parameters $\mathbf{p}$ of the program, which are the probabilities that annotate ADRs.

3.3.1. Fixed-point parameter learning

We start with the fixed-point learning procedure, which can be used when we have only ADRs (but no NADR).

Proposition 3.1. *Let $\mathbb{P}(X = x)$ be the probability of a specific probabilistic component X we wish to learn from the set of observations $\mathbf{O}$. If the iterated application of the rule*

$$\mathbb{P}(X = x) = \frac{1}{|\mathbf{O}|} \cdot \sum_{O \in \mathbf{O}} \frac{\mathbb{P}(X = x, O)}{\mathbb{P}(O)}. \qquad (2)$$

converges, then it does so to a critical point of the log-likelihood function.

The marginal $\mathbb{P}(X = x, O)$ can be computed by counting

the models consistent with both the observation O and $X = x$ and weighting over the probability of the total choices.

3.3.2. LAGRANGIAN PARAMETER LEARNING

We now derive an update rule that applies also in the presence of NADRs, and is an alternative to NEURASP's learning rule (Yang et al., 2020). To better understand the need of our alternative parameter learning method, we must first understand the shortcomings of the NEURASP's learning rule. The rule updates parameters by $\mathbf{p} \leftarrow \mathbf{p} - \eta \nabla_\rho \mathcal{L}(O)$, where η is a learning rate and the gradient components are:

$$\frac{\partial}{\partial p_x} \mathcal{L}(O) =$$
$$\frac{1}{\mathbb{P}(O)} \sum_{\theta_x} \frac{\mathbb{P}(\theta_x)}{\mathbb{P}(X=x)} \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)} \qquad (3)$$
$$- \sum_{\overline{x}, \overline{x} \neq x} \frac{1}{\mathbb{P}(O)} \sum_{\theta_{\overline{x}}} \frac{\mathbb{P}(\theta_{\overline{x}})}{\mathbb{P}(X=\overline{x})} \cdot \frac{N(I_{\theta_{\overline{x}}} \models O)}{N(\theta_{\overline{x}})}.$$

The intuition is that, interpretations that are consistent with O increase the value of the derivative, while interpretations that are not decrease it.

Note, however, that the sum of the updates over all the parameters p_x of a probabilistic component X is only zero when X is binary, which means that rule (3) can produce estimates that are outside the feasible set of valid parameters (i.e., they are not probability distributions). This issue can be mitigated by either projecting the parameters back to the feasible set or ensuring that parameter updates lie within the feasible set, for instance by using a softmax layer. To avoid this issue, we instead constrain parameters to remain within the feasible set by employing Lagrange multipliers.

Proposition 3.2. *The constrained derivative of the log-likelihood function with respect to the probability $\mathbb{P}(X = x) = p_x$ of a probabilistic component X is*

$$\frac{\partial}{\partial p_x} \mathcal{L}(O) =$$
$$\left(1 - \frac{1}{m}\right) \frac{1}{\mathbb{P}(O)} \sum_{\theta_x} \frac{\mathbb{P}(\theta_x)}{\mathbb{P}(X=x)} \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)} \qquad (4)$$
$$- \frac{1}{m} \sum_{\overline{x}, \overline{x} \neq x} \frac{1}{\mathbb{P}(O)} \sum_{\theta_{\overline{x}}} \frac{\mathbb{P}(\theta_{\overline{x}})}{\mathbb{P}(X=\overline{x})} \cdot \frac{N(I_{\theta_{\overline{x}}} \models O)}{N(\theta_{\overline{x}})}.$$

where m is the number of possible values X can take.

Interestingly, (4) yields a similar expression to (3), with the only distinction being the factors $1 - \frac{1}{m}$ and $\frac{1}{m}$. Thus, when $m = 2$, the Lagrangian rule is equivalent to halving the learning rate of NEURASP's rule. For $m > 2$, rule (4) assigns more weight to the probability of interpretations consistent with the observation, and less weight to its complement. Note that this is more sensible, since the latter sums over more terms than the former.

The extension of (4) to the neural case is trivial; by applying the chain rule on the derivative of the log-likelihood with respect to the output p_x of the neural component X, we easily find that the resulting gradient is (4) multiplied by the derivative of the neural network with respect to network weights $\mathbf{w}$

$$\frac{\partial}{\partial \mathbf{w}} \mathcal{L}(O) = \frac{\partial \mathcal{L}(O)}{\partial p_x} \frac{\partial p_x(\mathbf{x})}{\partial \mathbf{w}}, \qquad (5)$$

where $\frac{\partial p_x(\mathbf{x})}{\partial \mathbf{w}}$ is the standard backward pass in a neural network.

4. Experiments

In this section, we present two preliminary experiments showcasing the dPASP system. The first experiment compares the performance of our system against two competitors on the task of parameter learning in image classification. The second showcases a possible use of the credal semantics in cautious ensemble classification.

4.1. MNIST Addition

We compare the performance of dPASP to NEURASP, DEEPPROBLOG and a purely data-driven convolutional neural network (CNN) on the task of learning addition of MNIST image digits, a common distant supervision benchmark for NPLP (Manhaeve et al., 2021). This is a preliminary experiment, as dPASP is still in early development. Given two unlabelled images (e.g. 4 and 5) of digits, and the corresponding atom (e.g. sum(9)) as a distant label, the program must learn to identify the sum of digits.

The dPASP program to perform this task is quite simple and short if we do not account for the Python code needed for processing the MNIST data. The program in its entirety can be found in Appendix B.

```
#python
def net(): ...          # neural network
def mnist_tr(i): ...     # train images for i-th digit
def mnist_te(i): ...     # test images for i-th digit
def labels(): ...        # sum(x) labels
#end.

input(0) ~ test(@mnist_te(0)), train(@mnist_tr(0)).
input(1) ~ test(@mnist_te(1)), train(@mnist_tr(1)).

?::digit(X, {0..9}) as @net :- input(X).
sum(Z) :- digit(0, X), digit(1, Y), Z = X+Y.

#semantics maxent.
#learn @labels, lr = 0.001, niters = 5, batch = 1000.
```

We briefly highlight the fact that the prior know-how needed to write a program in NEURASP or DEEPPROBLOG can be

a significant barrier to the widespread use of NPLPs. Not only is the user required to have a good grasp of Python, but they must also have a significant understanding of the deep learning system used by the NPLP. For instance, one might compare the equivalent programs in NEURASP[1] and DEEPPROBLOG[2] to ours in order to understand the steep learning curve of current NPLPs.

Going back to our preliminary experiment, we follow a similar methodology used in NEURASP (Yang et al., 2020) and DEEPPROBLOG (Manhaeve et al., 2021) for the MNIST digit addition task. Just like the aforementioned works, we split the original MNIST dataset in half, taking the first (resp. second) half as the images of the first (resp. second) digit; the labels observed by the program are the atoms corresponding to the sum of the labels of the two halves. We use the same learning parameters for dPASP, NEURASP, DEEPPROBLOG and CNN: a learning rate of 0.001, batch size of 1000, and the ADAM optimizer for the neural components/networks (Kingma & Ba, 2017).

Figure 1 shows a comparison of both the performance in terms of classification accuracy, as well as training time. The plot on the left compares the accuracy of programs in classifying the correct sum of digits, while the plot on the right shows the digit classification accuracy of the embedded neural networks in the programs while they learn to classify the sum of digits. Curve CNN SUM corresponds to the performance of evaluating a CNN whose input is a single image consisting of concatenating the two digits and whose output are the probabilities of the 19 possible two-digit sum values; CNN DIGIT is the accuracy of a single digit classification network under the same parameter conditions as dPASP, NEURASP and DEEPPROBLOG.

Interestingly, both purely data-driven CNN approaches performed poorly compared to NPLPs. In particular, CNN SUM struggled to even break 50% accuracy, while CNN DIGIT quickly converged to the 80% mark, below that of NPLPs. We again stress the fact that these results were obtained by subjecting all systems to the same learning parameters.

Comparing dPASP against DEEPPROBLOG and NEURASP, we find that NEURASP achieves better accuracy faster, although dPASP eventually catches up. This difference between dPASP and NEURASP might be explained by the correction factor discussed in Section 3.3.2; the factors involved in the Lagrangian optimization slow down learning, as the gradient is diminished due to the correction.

Of note is the significant difference in training time for the four methods, with a surprising gap between dPASP and CNN. We conjecture that the main factor that explains this discrepancy is implementation overhead: dPASP is mostly written in C, with only the grammar parsing part of the language implemented in Python; in contrast, even though most of the computation in a pure Python script using PyTorch or any other deep learning framework is also done in C, a lot of the boilerplate code runs in Python. Thus, the careful implementation of highly optimized C code might make up for the use of logical inference routines in small problems like MNIST addition.

Given that dPASP is quite faster compared to the other approaches, we argue that this speed-up is a key advantage of dPASP (when the number of probabilistic and neural components is low) and should be exploited. With this in mind, we lower the batch size of dPASP in order to show how, by taking advantage of the careful optimized implementation in dPASP, we might achieve better performance by slightly increasing training time. The dPASP batch 500 curve in Figure 1 shows the impact of performance in terms of accuracy when halving the batch size, which causes only a slight increase (6 seconds) in training time.

4.2. Ensemble Classification

The following program considers pooling probabilistic predictions of different forecasters without information about accuracy of each forecaster. The neural predicates f and g are pre-trained MNIST digit classifiers. Our goal is to perform a digit classification from a two-classifier ensemble and compare a precise strategy that gives equal weight to each of the two classifiers, against a credal strategy that employs a more cautious approach.

```
data(x) ~ test(@mnist_test), train(@mnist_train).
% we have two classifiers/forecasters
class(f). class(g).
!::f(X,{0,...,9}) as @net1 :- data(X).
!::g(X,{0,...,9}) as @net2 :- data(X).
pred(f,Y) :- f(x,Y).  pred(g,Y) :- g(x,Y).
% at least one of them is correct
hit(f); hit(g).
% their correctness is consistent
hit(N2)  :- pred(N1,Y), pred(N2,Y), N1 != N2, hit(N1).
miss(N2) :- pred(N1,Y), pred(N2,Y), N1 != N2, miss(N1).
miss(N1); miss(N2) :- pred(N1,Y), pred(N2,Z), Y != Z.
miss(N) :- not hit(N), class(N).
% classification agrees with correct prediction
digit(Y) :- pred(N,Y), hit(N).
% We must pick one of the below semantics.
#semantics credal. % for credal semantics.
#semantics maxent. % for max-ent semantics.
% Query for the digit probabilities.
#query digit(0). ... #query digit(9).
```

The program encodes the assumption that some of the predictions made by the forecaster must be correct (hit), and
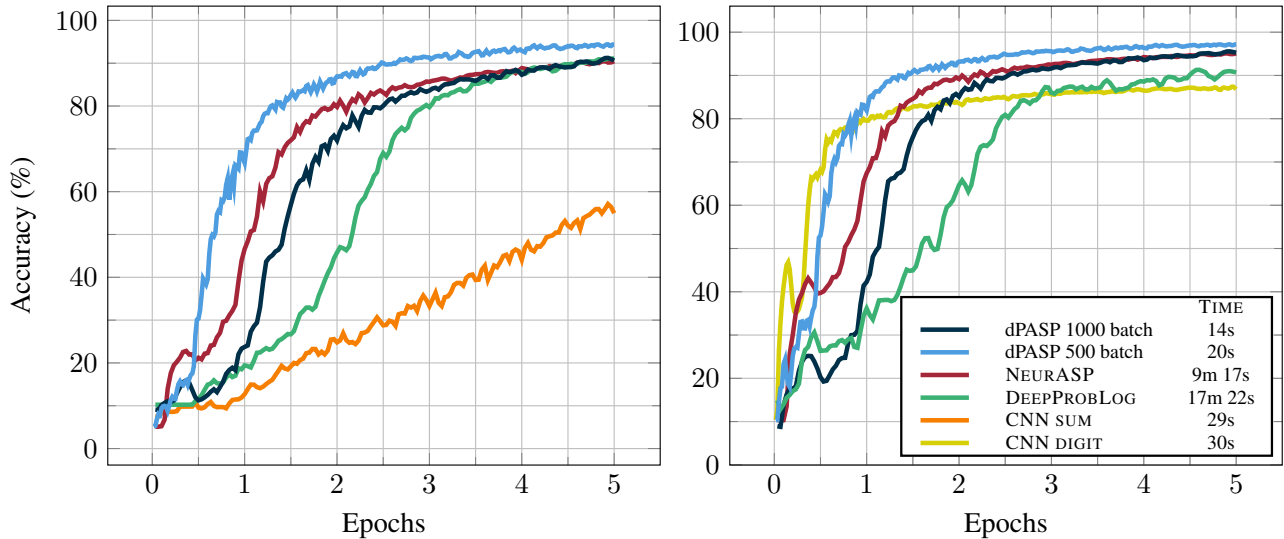

Figure 1. Sum and digit classification accuracy and training time for dPASP, NEURASP, DEEPPROBLOG and CNN. On the left, accuracy per iteration of classifying sums; on the right, accuracy of learned networks on classifying digits.

	Max-Ent	Credal	# examples
0	**98.36**	97.95	980
1	98.14	**98.23**	1135
2	92.24	**92.92**	1032
3	**93.56**	92.57	1010
4	**95.92**	95.72	982
5	92.48	**93.83**	892
6	96.13	**96.55**	958
7	94.06	**94.35**	1028
8	89.42	**89.93**	974
9	46.77	**47.77**	1009
Total	89.73	**89.99**	10000

Table 1. Accuracy when classifying each MNIST digit under the max-ent or credal semantics. Best accuracy in bold.

rules outs inconsistent assessments (e.g., two different predictions being considered both correct).

Using our framework and the previous logic program, we can perform either precise or credal inference. The precise inference amounts to simply computing the probability of each class under the maximum entropy semantics and taking the most likely one. For the credal semantics, we employ a max-max decision strategy to classify digits; for each digit's lower and an upper probabilities, we select the upper (max) as the probability of being that digit. Once all upper probabilities have been selected, we then select the class with the highest (max) probability.

The accuracy for detecting each digit using the two strategies is shown in Table 1. We can see that a more cautious

ensemble strategy based on the credal semantics can achieve slightly better performance in terms of accuracy compared to a simple uniform weighting of components.

5. Conclusion

We have presented dPASP, a new and flexible framework for neurosymbolic reasoning based on probabilistic logic programming. The framework extends the answer set programming declarative language with probabilistic and neural facts that allows the specification of uncertain knowledge and the tight integration of deep perception models (image classifiers, named entity recognizers, etc) with logic reasoning and constraint solving.

Unlike other similar systems such as NEURASP and DEEPPROBLOG, the framework implementation provides several different semantics for the logic and probabilistic parts, and a more friendly interface between machine learning components (e.g., PyTorch models) and the logic specification.

The system is also relatively fast for learning certain class of models, as illustrated by preliminary empirical results. This is due to a careful software implementation, which we release as free and open source at https://kamel.ime.usp.br/dpasp.

There is still much to achieve to make the system more broadly applicable. In particular, more efficient learning and inference routines need to be devised to scale to larger domains.

A. Proofs

Proposition 3.1. *Let $\mathbb{P}(X = x)$ be the probability of a specific probabilistic component X we wish to learn from the set of observations $\mathbf{O}$. If the iterated application of the rule*

$$\mathbb{P}(X = x) = \frac{1}{|\mathbf{O}|} \cdot \sum_{O \in \mathbf{O}} \frac{\mathbb{P}(X = x, O)}{\mathbb{P}(O)}. \qquad (2)$$

converges, then it does so to a critical point of the log-likelihood function.

Proof. The log-likelihood function of the program is given by

$$\mathcal{L}(\mathbf{O}) = \sum_{O \in \mathbf{O}} \log \sum_{\theta \in \Theta} \mathbb{P}(\theta) \cdot \frac{N(I_\theta \models O)}{N(\theta)}. \qquad (6)$$

We wish to constrain the probabilities of a probabilistic component X to $\sum_{x \in \mathcal{X}} \mathbb{P}(X = x) = 1$ and $\mathbb{P}(X = x) > 0$, where $\mathcal{X}$ is the set of all possible values X can take. To do this, instead of directly computing the derivative of $\mathcal{L}$ with respect to $\mathbb{P}(X = x)$, we instead compute $\frac{\partial}{\partial w_x} \mathcal{L}$, where $w_x \in \mathbb{R}$ is unconstrained and define

$$\mathbb{P}(X = x) = \frac{e^{w_x}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}}, \qquad (7)$$

i.e. we optimize with respect to a softmax function instead of directly working with the probabilities. With this in mind, the derivative of the log-likelihood function is given by

$$\frac{\partial}{\partial w_x} \mathcal{L}(\mathbf{O}) =$$
$$\sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \sum_{\theta \in \Theta} \frac{e^{w_x}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \mathbb{P}(\theta_{-x}) \cdot \frac{N(I_\theta \models O)}{N(\theta)}, \qquad (8)$$

where $\mathbb{P}(\theta_{-x})$ is the probability of the total choice θ excluding the probability of X, or more formally

$$\mathbb{P}(\theta_{-x}) := \prod_{\substack{(Y=y) \in \theta \\ Y \neq X}} \mathbb{P}(Y = y) = \frac{\mathbb{P}(\theta)}{\mathbb{P}(X = x)}. \qquad (9)$$

We may then split (6) into two terms: one where the total choices Θ_x agree with the weight w_x to be derived, and another where total choices $\Theta_{\overline{x}}$ choose other values $X = \overline{x}$,

$\overline{x} \neq x$ for X

$$\frac{\partial}{\partial w_x} \mathcal{L}(\mathbf{O}) = \sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \cdot$$

$$\overbrace{\left(\underbrace{\frac{\partial}{\partial w_x} \sum_{\theta_x \in \Theta_x} \frac{e^{w_x}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \mathbb{P}(\theta_{-x}) \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)}}_{(11)} \right.}^{}$$
$$\left. + \underbrace{\frac{\partial}{\partial w_x} \sum_{\theta_{\overline{x}} \in \Theta_{\overline{x}}} \frac{e^{w_{\overline{x}}}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \mathbb{P}(\theta_{-\overline{x}}) \cdot \frac{N(I_{\theta_{\overline{x}}} \models O)}{N(\theta_{\overline{x}})}}_{(12)} \right). \qquad (10)$$

For the sake of clarity, let us call $N_\theta^O = \frac{N(I_\theta \models O)}{N(\theta)}$. We may simplify the first term in (10) to

$$\sum_{\theta_x \in \Theta_x} \frac{e^{w_x} \cdot \sum_{x' \in \mathcal{X}} e^{w_{x'}} - e^{w_x} \cdot e^{w_x}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}} \cdot \sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \mathbb{P}(\theta_{-x}) \cdot N_{\theta_x}^O =$$

$$\sum_{\theta_x \in \Theta_x} \left(\frac{e^{w_x}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} - \left(\frac{e^{w_{x'}}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \right)^2 \right) \cdot \mathbb{P}(\theta_{-x}) \cdot N_{\theta_x}^O =$$

$$\sum_{\theta_x \in \Theta_x} \mathbb{P}(\theta_x) \cdot N_{\theta_x}^O - \sum_{\theta_x \in \Theta_x} \frac{e^{w_{x'}}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \mathbb{P}(\theta_x) \cdot N_{\theta_x}^O =$$

$$\mathbb{P}(X = x, O) - \mathbb{P}(X = x) \cdot \mathbb{P}(X = x, O) \qquad (11)$$

and the second term to

$$-\sum_{\theta_{\overline{x}} \in \Theta_{\overline{x}}} \frac{e^{w_{\overline{x}}}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \frac{e^{w_x}}{\sum_{x' \in \mathcal{X}} e^{w_{x'}}} \cdot \mathbb{P}(\theta_{-\overline{x}}) \cdot N_{\theta_{\overline{x}}}^O =$$

$$-\sum_{\theta_{\overline{x}} \in \Theta_{\overline{x}}} \mathbb{P}(X = x) \cdot \mathbb{P}(\theta_{\overline{x}}) \cdot N_{\theta_{\overline{x}}}^O = \qquad (12)$$

$$\mathbb{P}(X = x) \cdot \mathbb{P}(X = \overline{x}, O). $$

Putting (11) and (12) together, we get

$$\frac{\partial}{\partial w_x} \mathcal{L}(\mathbf{O}) = \sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \big[\mathbb{P}(X = x, O) -$$

$$\mathbb{P}(X = x) \cdot \mathbb{P}(X = x, O) - \mathbb{P}(X = x) \cdot \mathbb{P}(X = \overline{x}, O) \big] =$$

$$\sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \big[\mathbb{P}(X = x, O) - \mathbb{P}(X = x) \cdot \mathbb{P}(O) \big]. \qquad (13)$$

By setting the derivative of the objective function to zero to

find the critical point, we finally find that

$$\sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \big[\mathbb{P}(X = x, O) - \mathbb{P}(X = x) \cdot \mathbb{P}(O) \big] =$$
$$\sum_{O \in \mathbf{O}} \left(\frac{\mathbb{P}(X = x, O)}{\mathbb{P}(O)} - \mathbb{P}(X = x) \right) =$$
$$\sum_{O \in \mathbf{O}} \frac{\mathbb{P}(X = x, O)}{\mathbb{P}(O)} - |\mathbf{O}| \cdot \mathbb{P}(X = x) = 0$$
$$\implies \mathbb{P}(X = x) = \frac{1}{|\mathbf{O}|} \sum_{O \in \mathbf{O}} \frac{\mathbb{P}(X = x, O)}{\mathbb{P}(O)}. \qquad (14)$$

$\square$

Proposition 3.2. *The constrained derivative of the log-likelihood function with respect to the probability* $\mathbb{P}(X = x) = p_x$ *of a probabilistic component* X *is*

$$\frac{\partial}{\partial p_x} \mathcal{L}(O) =$$
$$\left(1 - \frac{1}{m} \right) \frac{1}{\mathbb{P}(O)} \sum_{\theta_x} \frac{\mathbb{P}(\theta_x)}{\mathbb{P}(X = x)} \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)} \qquad (4)$$
$$- \frac{1}{m} \sum_{\overline{x}, \overline{x} \neq x} \frac{1}{\mathbb{P}(O)} \sum_{\theta_{\overline{x}}} \frac{\mathbb{P}(\theta_{\overline{x}})}{\mathbb{P}(X = \overline{x})} \cdot \frac{N(I_{\theta_{\overline{x}}} \models O)}{N(\theta_{\overline{x}})}.$$

where m *is the number of possible values* X *can take.*

Proof. Recall that the log-likelihood function is given by

$$\mathcal{L}(\mathbf{O}) = \sum_{O \in \mathbf{O}} \log \sum_{\theta \in \Theta} \mathbb{P}(\theta) \cdot \frac{N(I_\theta \models O)}{N(\theta)}, \qquad (15)$$

and that the derivative of the log-likelihood function with respect to a probabilistic component $p_x = \mathbb{P}(X = x)$ is given by the expression

$$\frac{\partial}{\partial p_x} \mathcal{L}(\mathbf{O}) = \sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \frac{\partial}{\partial p_x} \sum_{\theta \in \Theta} \mathbb{P}(\theta) \cdot \frac{N(I_\theta \models O)}{N(\theta)}. \qquad (16)$$

We define the objective function as the log-likelihood subject to the restriction that all probabilities over X sum to one, that is, let $\mathcal{X}$ be the set of all possible values taken by X, then the new objective function is the log-likelihood with the added Lagrange multiplier λ

$$\hat{\mathcal{L}}(\{p_x\}_{x \in \mathcal{X}}, \lambda, \mathbf{O}) = \mathcal{L}(\mathbf{O}) - \lambda \left(\sum_{x \in \mathcal{X}} p_x - 1 \right). \qquad (17)$$

Optimization must now take place within the new constrained log-likelihood $\hat{\mathcal{L}}$. To find the complete expression of (17), we must find the value of the Lagrange multiplier λ,

which is achievable by noting that the sum of all derivatives with respect to X must sum to zero

$$\sum_{x \in \mathcal{X}} \frac{\partial}{\partial p_x} \hat{\mathcal{L}}(\{p_x\}_{x \in \mathcal{X}}, \lambda, \mathbf{O}) = \sum_{x \in \mathcal{X}} \left(\frac{\partial}{\partial p_x} \mathcal{L}(\mathbf{O}) - \lambda \right) =$$
$$\sum_{x \in \mathcal{X}} \frac{\partial}{\partial p_x} \mathcal{L}(\mathbf{O}) - |\mathcal{X}| \cdot \lambda \implies \lambda = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \frac{\partial}{\partial p_x} \mathcal{L}(\mathbf{O}). \qquad (18)$$

Let us first revisit (16), as it appears as a term of the gradient of our new objective function. We may split (16) into two terms, one where total choices Θ_x agree with the assignment $X = x$, and the other where total choices $\Theta_{\overline{x}}$ choose other values $X = \overline{x}, \overline{x} \neq x$

$$(16) =$$
$$\sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \left(\frac{\partial}{\partial p_x} \sum_{\theta_x \in \Theta_x} \mathbb{P}(\theta_x) \frac{N(I_{\theta_x} \models O)}{N(\theta_x)} + \frac{\partial}{\partial p_x} \sum_{\theta_{\overline{x}} \in \Theta_{\overline{x}}} \mathbb{P}(\theta_{\overline{x}}) \frac{N(I_{\theta_{\overline{x}}} \models O)}{N(\theta_{\overline{x}})} \right). \qquad (19)$$

We now derive each coordinate of the gradient of (17)

$$\frac{\partial}{\partial p_x} \hat{\mathcal{L}}(\{p_x\}_{x \in \mathcal{X}}, \lambda, \mathbf{O}) = \frac{\partial}{\partial p_x} \mathcal{L}(\mathbf{O}) - \lambda =$$
$$(19) - \frac{1}{|\mathcal{X}|} \sum_{x' \in \mathcal{X}} \frac{\partial}{\partial p_{x'}} \mathcal{L}(\mathbf{O}). \qquad (20)$$

To further simplify the above expression, we must further develop the formula in (19). Within the constraints set by the Lagrange multiplier, we may then simplify (19) as

$$(19) = \sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \sum_{\theta_x \in \Theta_x} \frac{\mathbb{P}(\theta_x)}{\mathbb{P}(X = x)} \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)}, \qquad (21)$$

since the second term in (19) cancels out to zero. Note that we do not need to write $p_{\overline{x}}$ as a function of p_x (which would mean that the term would not be equal to zero), as the Lagrange multiplier is already taking into account this relationship. Finally,

$$(18) = \sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \sum_{\theta_x \in \Theta_x} \frac{\mathbb{P}(\theta_x)}{\mathbb{P}(X = x)} \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)} -$$
$$\frac{1}{|\mathcal{X}|} \sum_{x' \in \mathcal{X}} \sum_{O \in \mathbf{O}} \frac{1}{\mathbb{P}(O)} \sum_{\theta_{x'} \in \Theta_x} \frac{\mathbb{P}(\theta_{x'})}{\mathbb{P}(X = x')} \cdot \frac{N(I_{\theta_{x'}} \models O)}{N(\theta_{x'})} \qquad (22)$$

can be rearranged so that when $x' = x$, the first and second

terms are subtracted, yielding the final form

$$
\frac{\partial}{\partial p_x} \hat{\mathcal{L}}(O) =
$$

$$
\left(1 - \frac{1}{|\mathcal{X}|}\right) \frac{1}{\mathbb{P}(O)} \sum_{\theta_x \in \Theta_x} \frac{\mathbb{P}(\theta_x)}{\mathbb{P}(X = x)} \cdot \frac{N(I_{\theta_x} \models O)}{N(\theta_x)}
$$

$$
- \frac{1}{|\mathcal{X}|} \sum_{\overline{x}, \overline{x} \neq x} \frac{1}{\mathbb{P}(O)} \sum_{\theta_{\overline{x}} \in \Theta_{\overline{x}}} \frac{\mathbb{P}(\theta_{\overline{x}})}{\mathbb{P}(X = \overline{x})} \cdot \frac{N(I_{\theta_{\overline{x}}} \models O)}{N(\theta_{\overline{x}})};
$$

$$
\tag{23}
$$

call $m = |\mathcal{X}|$ and we get the expression in our claim. $\qquad\square$

B. dPASP MNIST Addition Program

References

Azzolini, D., Bellodi, E., and Riguzzi, F. Approximate inference in probabilistic answer set programming for statistical probabilities. In Dovier, A., Montanari, A., and Orlandini, A. (eds.), *AIxIA 2022 – Advances in Artificial Intelligence*, pp. 33–46, Cham, 2023.

Cozman, F. G. and Mauá, D. D. The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference. *International Journal of Approximate Reasoning*, 2020.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and de Raedt, L. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15:358–401, 2015.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Answer set solving in practice. In *Answer Set Solving in Practice*, 2012.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.

Janhunen, T., Niemelä, I., Seipel, D., Simons, P., and You, J.-H. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Logic*, 7, 2006.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Lifschitz, V. *Answer Set Programming*. Springer Cham, 2019.

Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., and De Raedt, L. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 298, 2021.

Przymusinski, T. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

Rocha, V. H. N. and Cozman, F. G. A Credal Least Undefined Stable Semantics for Probabilistic Logic Programs and Probabilistic Argumentation. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, 2022.

Saccà, D. and Zaniolo, C. Deterministic and non-deterministic stable models. *Journal of Logic and Computation*, 7(5):555–579, 1997.

Sato, T. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, 1995.

Totis, P., Kimmig, A., and Raedt, L. D. Smproblog: Stable model semantics in problog and its applications in argumentation. *ArXiv*, abs/2110.01990, 2021.

Tuckey, D., Russo, A., and Broda, K. PASOCS: A parallel approximate solver for probabilistic logic programs under the credal semantics. *CoRR*, abs/2105.10908, 2021.

Yang, Z., Ishay, A., and Lee, J. Neurasp: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 2020.

```
% Addition of MNIST digits.

#python
import torch
import torchvision

class Net(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.encoder = torch.nn.Sequential(
      torch.nn.Conv2d(1, 6, 5),
      torch.nn.MaxPool2d(2, 2),
      torch.nn.ReLU(True),
      torch.nn.Conv2d(6, 16, 5),
      torch.nn.MaxPool2d(2, 2),
      torch.nn.ReLU(True)
    )
    self.classifier = torch.nn.Sequential(
      torch.nn.Linear(16 * 4 * 4, 120),
      torch.nn.ReLU(),
      torch.nn.Linear(120, 84),
      torch.nn.ReLU(),
      torch.nn.Linear(84, 10),
      torch.nn.Softmax(1)
    )

  def forward(self, x):
    x = self.encoder(x)
    x = x.view(-1, 16 * 4 * 4)
    x = self.classifier(x)
    return x

def digit_net(): return Net()

def mnist_data():
  train = torchvision.datasets.MNIST(root = "/tmp", train = True, download = True)
  test  = torchvision.datasets.MNIST(root = "/tmp", train = False, download = True)
  return train.data.float().reshape(len(train), 1, 28, 28)/255., train.targets, \
         test.data.float().reshape(len(test), 1, 28, 28)/255., test.targets

def normalize(X_R, Y_R, X_T, Y_T, mu, sigma):
  return (X_R-mu)/sigma, Y_R, (X_T-mu)/sigma, Y_T

train_X, train_Y, test_X, test_Y = normalize(*mnist_data(), 0.1307, 0.3081)
def pick_slice(data, which):
  h = len(data)//2
  return slice(h, len(data)) if which else slice(0, h)
def mnist_images_train(which): return train_X[pick_slice(train_X, which)]
def mnist_images_test(which): return test_X[pick_slice(test_X, which)]
def mnist_labels_train():
  labels = torch.concatenate((train_Y[:(h := len(train_Y)//2)].reshape(-1, 1),
                              train_Y[h:].reshape(-1, 1)), axis=1)
  return [[f"sum({x.item() + y.item()})"] for x, y in labels]
#end.

input(0) ~ test(@mnist_images_test(0)), train(@mnist_images_train(0)).
input(1) ~ test(@mnist_images_test(1)), train(@mnist_images_train(1)).

?::digit(X, {0..9}) as @digit_net with optim = "Adam", lr = 0.001 :- input(X).
sum(Z) :- digit(0, X), digit(1, Y), Z = X + Y.

#semantics maxent.
#learn @mnist_labels_train, lr = 1., niters = 5, alg = "lagrange", batch = 1000.
```