ALMA MATER STUDIORUM

UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Deep Learning

DEEP GENERATIVE MODELS WITH PROBABILISTIC LOGIC PRIORS

CANDIDATE

Eleonora Misino

SUPERVISOR

Prof. Claudio Sartori

CO-SUPERVISORS

Prof. Luc De Raedt

Giuseppe Marra, PhD.

Emanuele Sansone, PhD.

Academic year 2020/21

Session 2nd

Abstract

Many different extensions of the VAE framework have been introduced in the past. However, the vast majority of them focused on pure sub-symbolic approaches that are not sufficient for solving generative tasks that require a form of reasoning. In this thesis, we propose the *probabilistic logic* VAE (PLVAE), a neuro-symbolic deep generative model that combines the representational power of VAEs with the reasoning ability of probabilistic-logic programming. The strength of PLVAE resides in its probabilistic-logic prior, which provides an interpretable structure to the latent space that can be easily changed in order to apply the model to different scenarios. We provide empirical results of our approach by training PLVAE on a base task and then using the same model to generalize to novel tasks that involve reasoning with the same set of symbols.

Contents

Introduction

Deep generative modeling aims at learning an unknown density function from a set of i.i.d samples. Computing the mapping function may be really challenging, and even infeasible for high-dimensional input spaces. Thus, this task is usually solved by relying on deep neural networks (DNNs), that have the ability to learn and model non-linear mappings with high-dimensional domains. Although DNNs excel at solving many different tasks, they are not yet sufficient for solving reasoning tasks. Moreover, due to their pure sub-symbolic nature, DNNs lack of interpretability, which is a key feature in all those settings that require to understand and control the output of the model. Consequently, deep generative models inherit the weaknesses of DNNs.

We can identify five main classes of deep generative models: Generative Adversarial Network (GAN) [1], Variational Autoencoder (VAE) [2], Autoregressive models [3] [4] [5], Flow-based models [6] [7] [8] and Energy-based models like Boltzmann Machines [9].

Although there exist several deep generative models, in this thesis we exploit the representational power of VAEs, which are Bayesian latent variable models based on DNNs to perform inference in high-dimensional domains. In particular, VAEs assume a generative process in which the observations are generated from unobserved latent variables.

The majority of the research efforts on improving the VAE framework is limited on exploring pure sub-symbolic approaches. Several works focused on reducing the gap between approximate and true posterior distributions [10–13], others on defining more flexible prior distributions [14–17] or on structuring the latent representation [18].

Although these works represent remarkable advances and achieve state-of-the-art results, they are still not sufficient for solving generative tasks that require a form of reasoning. For example, let's suppose to train a VAE model on a supervised dataset composed of pairs of digits labeled with their sum (Figure 1). In this setting, generating two numbers that sum up to 3 can be easily solved by the existing models [19]. However, none of them would be able to answer questions like “Generate two numbers that multiply to 6.” or “Generate two numbers that differ by 1”.

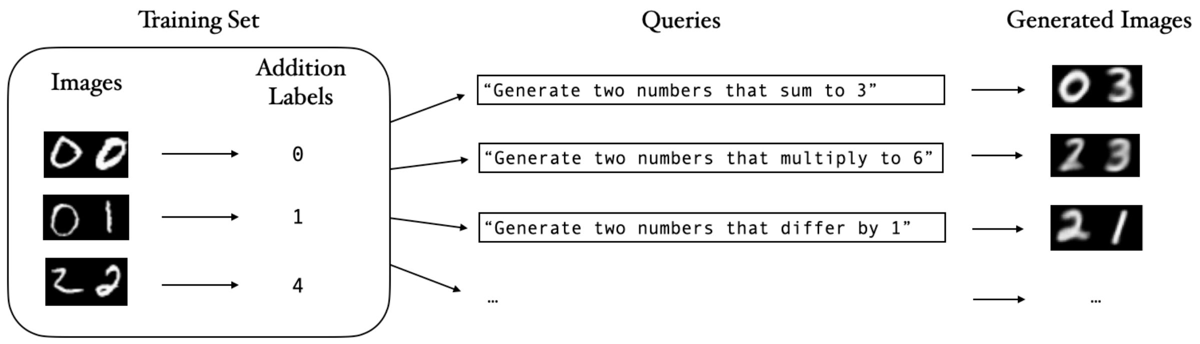

Training Set
Images
Addition Labels
0
1
4
Queries
“Generate two numbers that sum to 3”
“Generate two numbers that multiply to 6”
“Generate two numbers that differ by 1”
...
Generated Images
...

Figure 1: Example of task generalization in the generative framework.

To achieve this kind of task generalization, we need to go beyond the pure sub-symbolic representation of a scene and introduce a form of reasoning to define the desired relationship between generated images and labels. The two most prominent frameworks for reasoning are logic and probability theory, whose integration is an open research direction with remarkable results in the areas of statistical relational artificial intelligence [20] and probabilistic-logic programming [21].

In this thesis, we propose the *probabilistic logic* VAE (PLVAE), a neuro-symbolic deep generative model that combines the representational power of VAEs with the reasoning ability of ProbLog [22], a probabilistic-logic programming language. By virtue of the integration between the two approaches, PLVAE is able to generate new images starting from a logic formula, while preserving the predictive ability of VAEs. Moreover, since the probabilistic framework provides real-valued probabilistic quantities instead of discrete logic quantities, we are able to define an end-to-end trainable framework based on gradient training procedures to train PLVAE on examples. In contrast to other works on neuro-symbolic generative models [23, 24], we fully exploit the expressiveness of both neural and symbolic methods without limiting the symbolic engine to a structuring tool. PLVAE uses the reasoning capabilities of the probabilistic-logic framework to solve previously unseen generative tasks, such as the generalization to different arithmetic operations from a training task focused only on the addition. Moreover, the results show that the use of a logic-based prior helps the learning in contexts characterized by data scarcity.

Chapter 1

Background

1.1 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) are graph-based representations of complex distributions. They provide a simple way to visualize the structure of a probabilistic model and offer insights into model properties, such as conditional independencies among variables.
In a PGM, nodes correspond to variables, and edges correspond to direct probabilistic interactions between them. There is a dual perspective from which to interpret the structure of a PGM. On one hand, the graph is a compact representation of a set of independencies that hold in the distribution. On the other hand, the graph defines a skeleton for factorizing the full joint distribution as the product of local conditional distributions.
These two perspectives are equivalent: the independence properties of the distribution allow it to be represented compactly in a factorized form. Conversely, a particular factorization of the distribution guarantees that certain independencies hold.
There exist two main families of PGMs: the *Bayesian networks*, that use directed graphs, where each edge has a source and a target, and the *Markov networks*, that rely on undirected graphs. Both these graphical representations provide the duality of independencies and factorization, but they differ in the set of independencies they can encode and in the factorization of the distribution that they induce.
For our formulation, we are interested in Bayesian networks, where each node is conditionally independent of its non-descendants given its parents, and the joint distribution can be factorized as

$$P(X_1, X_2, ...X_n) = \prod_{i=i}^{n} P(X_i|parents(X_i)) \tag{1.1}$$

To better understand this class of PGMs, let's consider the well-known burglary alarm example, represented in Figure 1.1. By looking at the graph structure, we see that the independencies encoded by the graph structure are

$$B \perp E, \quad B \perp M|A, \quad E \perp M|A$$
$$B \perp J|A, \quad E \perp J|A, \quad M \perp J|A$$

and by applying equation 1.1, we can factorize the overall joint distribution as

$$P(B, E, A, M, J) = P(J|A)P(M|A)P(A|B, E)P(B)P(E)$$

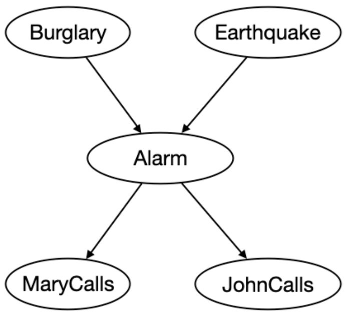


Figure 1.1: PGM of the burglary alarm example (due to J. Pearl).

1.2 Variational Inference

Variational inference [25] provides an analytical approximation of intractable probability distributions. If we consider a probability distribution p that we cannot infer using sampling-based methods, the variational approach transforms the inference into an optimization problem over a class of tractable distributions Q. The goal is to find $q \in Q$ that is the most similar to p, in order to query q, rather than p, and get an approximate solution to the inference problem.
In order to estimate the information lost in approximating p with the tractable distribution q, variational inference usually relies on the *Kullback-Leibler (KL) divergence* [26], that measures differences in information contained within the two distributions. The KL divergence over a

continuous domain X is defined as follows

$$D_{KL}(q\|p) = \int_X q(x) \log \frac{q(x)}{p(x)} dx \tag{1.2}$$

and it is asymmetric ($D_{KL}(q\|p) \neq D_{KL}(p\|q)$), non-negative, and null when $q(\,\cdot\,) \equiv p(\,\cdot\,)$.

The Variational Lower Bound

Let's now consider a *latent variable model* $p(x, z) = p(x|z)p(z)$, where $x \in X$ is the observed variable and $z \in Z$ is the *latent* (never observed) variable (Figure 1.2).

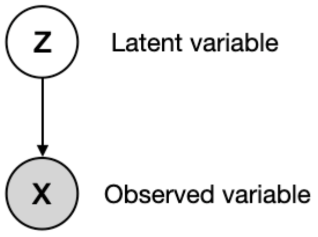


Figure 1.2: PGM of the latent variable model.

Let's suppose we are interested in inferring the posterior distribution of the latent variable given the observation, that is

$$p(z|x) = \frac{p(x, z)}{p(x)} = \frac{p(x|z)p(z)}{\int_Z p(x, z)dz} \tag{1.3}$$

For many models, computing the marginal likelihood of an observation $p(x) = \int_Z p(x, z)dz$ is infeasible. Thus, it is common to rely on variational inference to approximate $p(z|x)$ with a tractable distribution $q(z|x)$.
Given this formulation, the inference now amounts to solving the following optimization problem

$$\begin{aligned} q^* &= \arg\min_{q \in Q} \{D_{KL}(q(z|x) \| p(z|x)\} \\ &= \arg\min_{q \in Q} \{\mathbb{E}_q[\log(q(z|x)] - \mathbb{E}_q[\log(p(z|x))]\} \\ &= \arg\min_{q \in Q} \{\mathbb{E}_q[\log(q(z|x)] - \mathbb{E}_q[\log(p(z, x))] + \log(p(x))\} \end{aligned} \tag{1.4}$$

However, directly computing $D_{KL}(q(z|x) \| p(z|x)$ is not possible, because it needs to evaluate p.

Thus, let's consider an alternative objective that has the same form as the KL divergence, but does not involve the intractable $p(x)$

$$J(q) = \mathbb{E}_q[\log q(z|x)] - \mathbb{E}_q[\log p(x, z)] \tag{1.5}$$

By looking at 1.4, we can see that J is not only tractable, but is also equivalent to D_{KL} up to an added constant w.r.t. q, that is

$$J(q) = D_{KL}(q(z|x)||p(z|x)) - \log p(x). \tag{1.6}$$

Since D_{KL} is non-negative, then

$$\log p(x) = D_{KL}(q(z|x)||p(z|x)) - J(q) \geq -J(q). \tag{1.7}$$

Because of this property, $-J(q)$ is usually referred to as the evidence lower bound (a.k.a. *ELBO*) to emphasize that it is a lower bound on the evidence of the observations.

1.3 Deep Generative Models

Generative modeling aims at learning a representation of an intractable probability distribution $\mathcal{X}$ over $\mathbb{R}^n$, where n is typically very large. To this end, generative models learn a function g that maps samples from a tractable distribution $\mathcal{Z}$ supported in $\mathbb{R}^q$,with $q < n$, to points in $\mathbb{R}^n$ that resemble the original data.
In other words, in the generative approach we assume that for each vector $x \sim \mathcal{X}$, there is at least one *latent vector* $z \sim \mathcal{Z}$ such that $g(z) \approx x$. Once learned, the mapping function g can be used to generate samples from the intractable distribution $\mathcal{X}$ by sampling from the latent distribution $\mathcal{Z}$.
The task of deriving the mapping function g may be really challenging, and even infeasible for very high-dimensional input spaces. Therefore, since the early 2000s, deep neural networks have been largely used in generative approaches to effectively approximate the mapping function.
There exist four main classes of deep generative models: Generative Adversarial Network (GAN) [1], Variational Autoencoder (VAE) [2], Autoregressive models [3] [4] [5] and Flow-

based models [6] [7] [8], and Enegry-based models like Boltzmann Machines [9].
GAN is inspired by game theory: two models, a *generator* G and a *discriminator* D, play antagonistic roles and are trained simultaneously through a *Minimax* strategy. The discriminative model estimates the probability that a given sample comes from the real dataset. It works as a critic and is optimized to discriminate between real and synthetic samples. The generative model captures the real data distribution and is trained to generate samples as more realistic as possible to fool the discriminator.
On the other hand, VAE models the distribution of observations x using a stochastic latent vector $z \sim p(z)$ along with a likelihood $p(x|z)$ that connects z with the observation. In this framework, learning a representation of data consists of learning the posterior distribution $p(z|x)$ that constructs the distribution of latent values. VAE relies on variational inference (Section 1.2) to approximate the intractable posterior $p(z|x)$, and jointly trains an inference network (*encoder*) and a generative network (*decoder*) to model $p(x|z)$ and $q(z|x)$ respectively.
Neither GAN nor VAE explicitly learn the probability density function of real data $p(x)$, since it is usually infeasible. However, flow-based models are able to learn a good estimation of it by relying on *Normalizing Flow* methods [27], which transform simple distributions into complex ones by applying a sequence of invertible transformation functions. Flow-based generative models consist of a sequence of invertible transformations, and are trained to directly minimize the negative log-likelihood over the training set.
Finally, Autoregressive models are based on the chain rule of probability $p(x_1, ..., x_n) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1})$ to generate new data samples. The term *autoregressive* originates from the literature on time-series models where observations from the previous time-steps are used to predict the value at the current time step. Autoregressive generative models fix a precise order in the variable decomposition $x = x_1, ..., x_n$, and directly maximise the likelihood of the data $p(x)$ by training a recurrent neural network to model $p(x_i|x_1, ..., x_{i-1})$.

1.3.1 Variational Autoencoders

VAEs are Bayesian latent variable models that assume a generative process in which the observations x are generated from unobserved latent variables z through a likelihood $p_\theta(x|z)$ (Figure 1.3).
The target likelihood $p_\theta(x|z)$ is parameterized by a deep neural network (the *encoder*) trained on the observed data. Since the exact inference on the posterior $p_\theta(z|x)$ is intractable, as de-

scribed in Section 1.2, VAE approximates it with a tractable posterior $q_\phi(z|x)$, which is also parameterized by a deep neural network (the *decoder*).

Typically, the likelihood, the prior and the approximate posterior are chosen to be Gaussian

$$p_\theta(x|z) = \mathcal{N}(x;\theta)$$
$$p(z) = \mathcal{N}(z;0,I)$$
$$q_\phi(z|x) = \mathcal{N}(z;\phi).$$

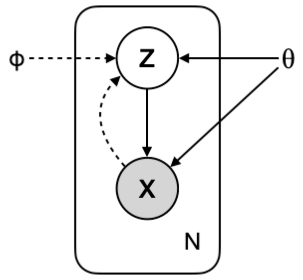


Figure 1.3: Graphical model of the VAE latent variable model.
Solid lines denote the generative model $p_\theta(x|z)p(z)$, dashed lines denote the inference process where the intractable posterior $p(z|x)$ is approximate through $q_\phi(z|x)$.

The encoder and the decoder are jointly trained by optimizing the ELBO

$$\mathcal{L}(x;\theta,\phi) = \mathbb{E}q_\phi(z|x)[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p(z)). \tag{1.8}$$

Thus, the training process involves a back-propagation w.r.t the generative and the variational parameters θ and ϕ.

However, estimating 1.8, requires to sample z from the posterior $q_\theta(z|x)$, and this sampling operation prevents back-propagation – and thus training. To solve this problem, Kingma et al. [28] proposed the so called *Reparametrization Trick*. This method consists in decomposing the latent variable z into a deterministic and a stochastic part to safely back-propagate through it (Figure 1.4). More precisely, z is decomposed as

$$z = \mu + \sigma \odot \epsilon \tag{1.9}$$

where $\epsilon \sim \mathcal{N}(0,I)$, and μ and σ are the mean and the standard deviation of $q_\phi(z|x) = \mathcal{N}(z;\phi)$.

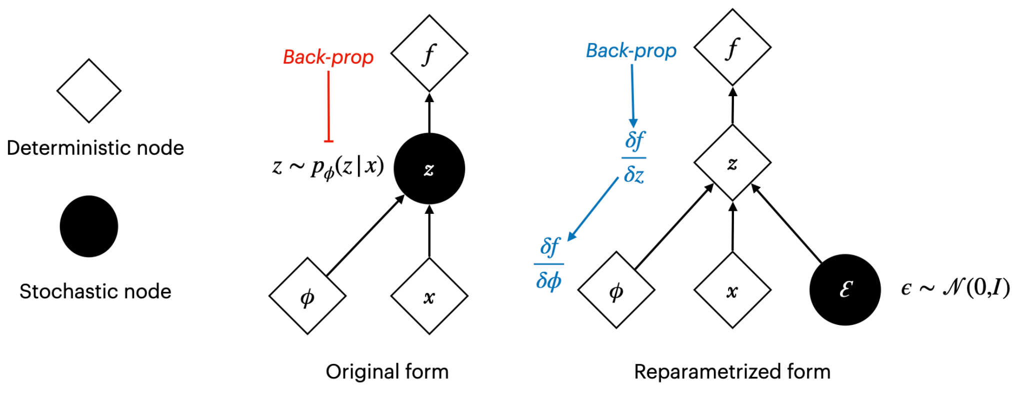


Figure 1.4: The reparametrization trick allows to back-propagate through the normally distributed variable z.

The reparametrization trick described above deals with normally distributed variables only, but our model involves also categorical latent variables. Therefore, let's briefly introduce the *Categorical Reparametrization with Gumbel-Softmax* [29], which allows to efficiently train a generative model with discrete latent variables by using the *softmax* function to provide a continuous, differentiable approximation of the *Gumbel-Max trick* [30].
Let's consider a categorical variable z with class probabilities $\pi_1, \pi_2, ...\pi_J$, and let's assume that each categorical sample to be encoded as a k-dimensional one-hot vector lying on one corner of the $(k-1)$-dimensional simplex Δ^{k-1}. By sampling from Gumbel$(0,1)$ distribution, we can generate k-dimensional sample vectors $\boldsymbol{y} \in \Delta^{k-1}$ defined as

$$\hat{y}_i = \tau_{y_i}(\hat{g}_i, \hat{\boldsymbol{\epsilon}}; \phi) := \frac{\exp((\log(\pi_i) + \hat{g}_i)/\tau)}{\sum_{j=1}^{J} \exp((\log(\pi_j) + \hat{g}_j)/\tau)}, \text{ with } \hat{g}_i \sim Gumbel(0,1) \tag{1.10}$$

These samples are identical to samples from a categorical distribution as $\tau \to 0$, and are differentiable for $\tau > 0$. Thus, by replacing categorical samples with Gumbel-Softmax samples, we can safely back-propagate to compute gradients and train the model.

1.4 Logic Programming

Logic Programming refers to a style of programming paradigm which is largely based on formal logic. One of the most widely used logic programming languages is *Prolog*. In this section, we briefly summarize some basic concepts of Prolog, necessary to understand its probabilistic extension: *ProbLog* [22].
In Prolog:

- *atoms* are expressions of the form $q(t_1, ..., t_n)$ where q is a predicate (of arity n) and t_i are terms;
- a *term* t can be either a constant c, a variable V, or a structured term of the form $f(u_1, ..., u_k)$ where f is a functor. According to Prolog convention, constants start with a lower case character and variables with an upper case;
- *literals* are atoms or the negation ($\neg$) of atoms;
- a *rule* is an expression of the form $q : -l_1, ..., l_n$ where q is an atom, l_i are literals, $: -$ represents logical implication ($\Longleftarrow$), and commas (,) represents logical conjunctions ($\wedge$). Therefore, the meaning of $q : -l_1, ..., l_n$ is that its head (q) holds whenever its tail (i.e. the conjunction of the l_i) holds;
- *facts* are rules with an empty body;
- a *clause* can be either a fact or a rule;
- a *substitution* $\theta = \{V_1 = t_1, ..., V_n = t_n\}$ is an assignment of terms t_i to variables V_i. When applying a substitution θ to an expression e, we simultaneously replace any occurrence of V_i by t_i and denote the resulting expression as $e\theta$.

The execution of a Prolog program is initiated by the definition of a goal, called *query*. Prolog engine tries to logically find a resolution refutation of the negated query by applying the *Selective Linear Definite clause resolution* (*SLD resolution*). The SLD resolution implicitly defines a search tree of alternative computations, in which the initial goal clause is associated with the root of the tree. Every path from the query root to the empty clause corresponds to a proof tree (i.e. a successful refutation proof). Thus, if the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program.

Here is a toy example of a Prolog program that models the addition of two digits:

```
digit(img1, 0). % Fact 1
digit(img1, 1). % Fact 2
digit(img2, 0). % Fact 3

% Rule 1
add(img1, img2, N):- digit(img1,N1), digit(img2,N2), N is N1 + N2.
```

and three different queries:

```
?-add(img1, img2, 1). % Query A
Yes

?-add(img1, img2, 3). % Query B
No

?-add(img1, img2, X). % Query C
Yes
X = 0
```

Let's have a look at the SLD tree of *Query A* (Figure 1.5) to better understand Prolog resolution. Since there are two facts that unifies with the clause `digit(img1,N1)`, namely *Fact 1* and *Fact 2*, the tree has 2 branches. The righ sub-tree is a successful refutation proof, i.e. it leads to an empty clause. Conversely, the left sub-tree is a failed branch since $1 \neq 0$. However, since there exists at least one successful refutation proof, *Query A* is a logical consequence of the program.

The SLD resolution tree of *Query B* (Figure 1.6) has the same structure of the one of *Query A*, but both the sub-trees lead to a failure since 3 is neither equal to 0 nor to 1. Thus, *Query B* is not a logical consequence of the program.

Regarding *Query C*, both the sub-trees lead to an empty clause: in the left one X is unified with the numerical constant 0, while in the right sub-tree it is unified with 1. Since Prolog traverse the SLD tree depth-first left-most, the answer to *Query C* is $X = 0$.

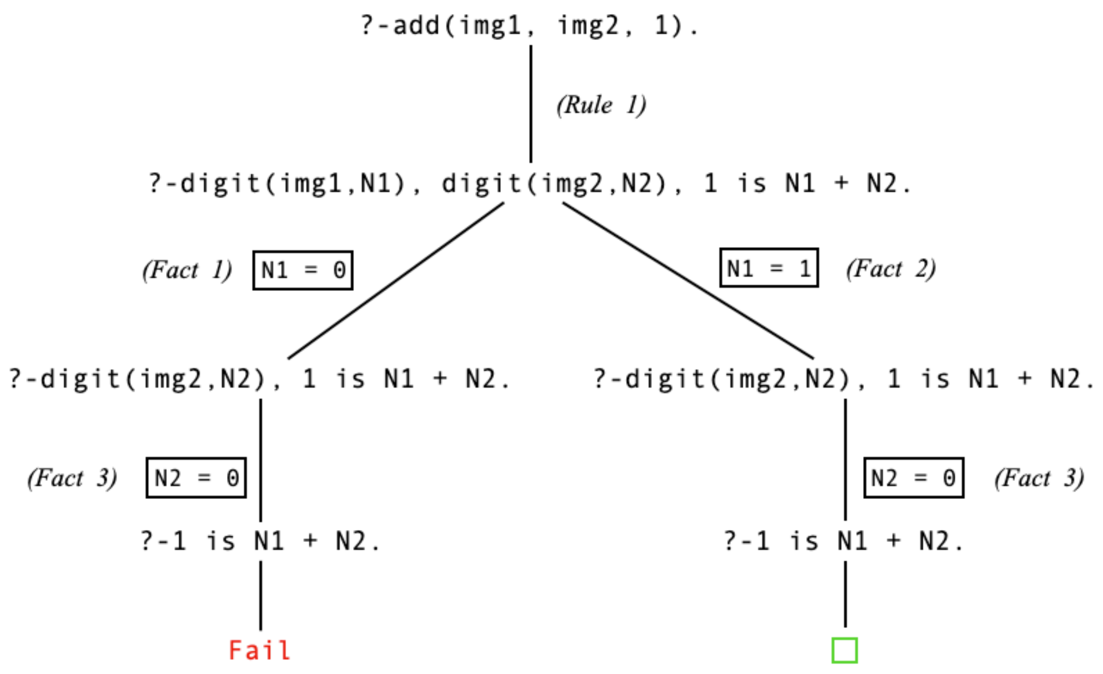


Figure 1.5: SLD resolution tree of *Query A*.

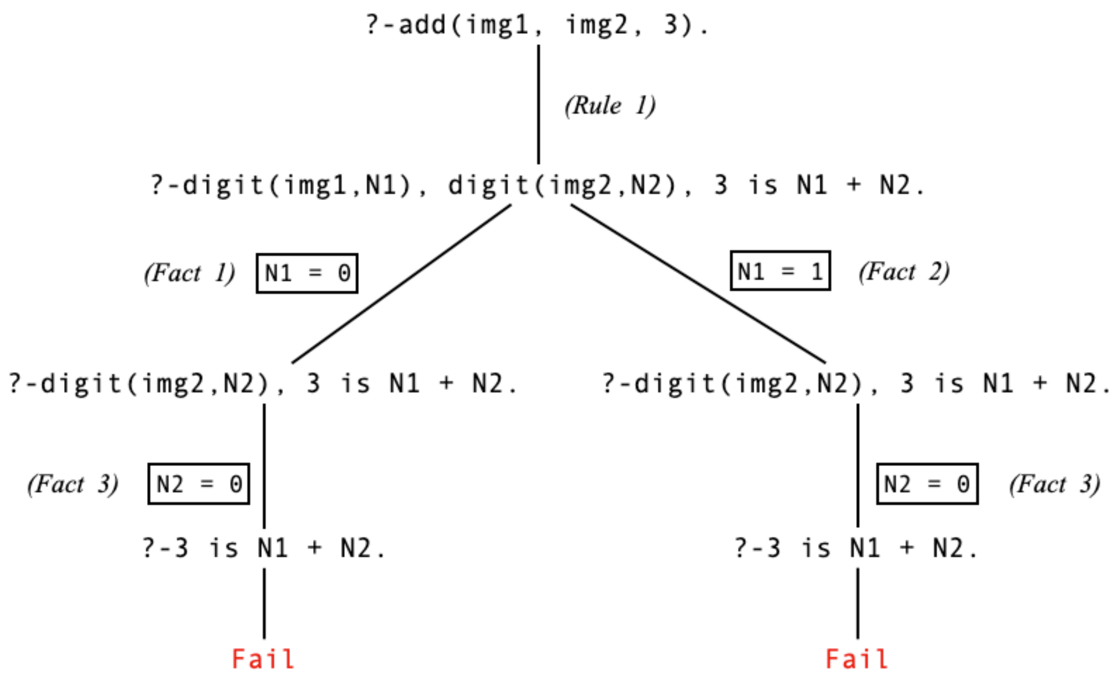


Figure 1.6: SLD resolution tree of *Query B*.

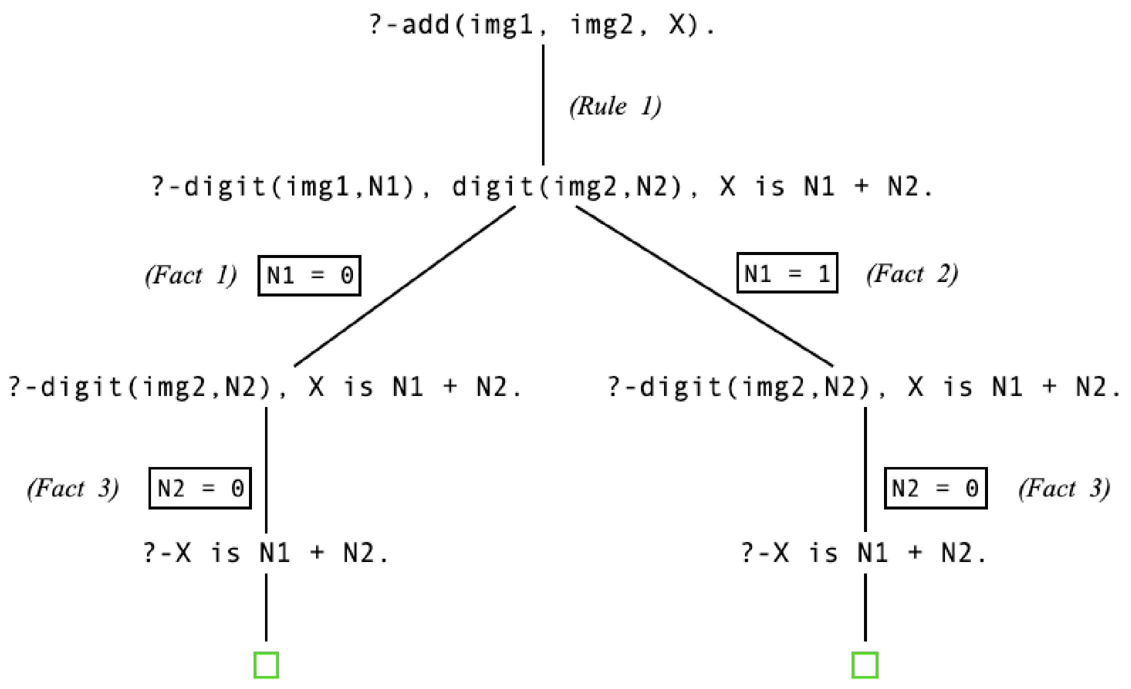


Figure 1.7: SLD resolution tree of *Query C*.

1.5 ProbLog

ProbLog [22] is a probabilistic extension of Prolog that allows to express complex, probabilistic models by assigning a probability to each clause.
A ProbLog program consists of

- a set of *ground probabilistic facts* $\mathcal{F}$ of the form $p :: f$, where p is a probability and f a ground atom. Each probabilistic facts represents an independent Boolean random variable with probability p of being true and $(1 - p)$ of being false;
- a set of rules $\mathcal{R}$.

Every subset $F \subseteq \mathcal{F}$ allows to define a possible world defined as the union between F and the set of ground atoms that are logically entailed by F and by the set of rules $\mathcal{R}$, namely

$$w_F := F \cup \{f\theta | \mathcal{R} \cup F \models f\theta, \text{ and } f\theta \text{ is ground}\}.$$

The probability $P(w_F)$ of a possible world is given by the product of the probabilities of the truth values of the probabilistic facts, that is

$$P(w_F) = \prod_{f_i \in F} p_i \prod_{f_i \in \mathcal{F} \setminus F} (1 - p_i). \tag{1.11}$$

The probability of a ground fact q, (i.e. the *success probability of* q), is defined as the sum of the probabilities of all worlds containing q

$$P(q) = \sum_{F: w_F \models q} P(w_F). \tag{1.12}$$

The entity q for which we want to compute the probability is called the *query*, and in a ProbLog program queries are specified by adding a fact `query(Query)`.
ProbLog also allows us to specify *evidences*, i.e. any observations on which we want to condition the probability of the query. The syntax to specify an evidence is the fact `evidence(Literal)`, which conditions parts of the program to be true or false.

Let's now briefly describe the four steps of ProbLog inference [31]:

1. *Grounding step.* The logic program is grounded with respect to the query via backward reasoning that allows to determine which ground rules are relevant to derive the truth value of the query, and may perform additional logical simplifications that do not affect the query's probability.

2. In the second step the ground logic program is transformed into a propositional logic formula that defines the truth value of the query in terms of the truth values of probabilistic facts. We can calculate the query success probability by performing *weighted model counting* (WMC, [31]) on this logic formula.

3. *Knowledge Compilation* [32]. Since performing WMC directly on the logical formula defined in the previous step is not efficient, ProbLog rewrites it into a form that allows for efficient WMC. ProbLog system uses *Sentential Decision Diagrams* (SDDs, [33]), which are a subset of deterministic *decomposable negational normal forms* (d-DNNFs, [34]) and allow for polytime model counting [32].

4. The final step transforms the SDD into an *arithmetic circuit* (AC, [35]), that is a repre-

sentation of a Bayesian network capable of answering arbitrary marginal and conditional queries, with the property that the cost of inference is linear in the size of the circuit. Starting from the SDD, the corresponding AC is built by putting the probabilities of the probabilistic facts or their negations on the leaves, replacing the OR and AND nodes with addition and multiplication respectively. Then, the weighted model counting is calculated with an evaluation of the AC.

If we add an evidence to a ProbLog program, the knowledge compilation is performed on the conjuction between the propositional logic formula of the ground program and the evidence itself; then, the inference proceeds as usual with the final step.
Therefore, a ProbLog program with an evidence $\boldsymbol{e}$ can be defined as the distribution on the possible worlds ω given $\boldsymbol{e}$, with the probability of the probabilistic facts $p :: f$ as parameters, namely

$$P(\omega \mid \boldsymbol{e}; p).$$

Let's now consider the example introduced in Section 1.4, where we define the addition of two digits. ProbLog allows us to specify the probability of each digit: for example, we can say that the second digit is a 0 or a 1 with probability 0.8 and 0.1 respectively. Such a disjunction of probabilistic facts can be expressed via an *annotated disjunction* (AD), that is nothing else than syntactic sugar. An AD is an expression of the form $p_1 :: f_1; ...; p_n :: f_n : -b_1, ..., b_m$, where the p_i are probabilities that sum to at most 1, the f_i are atoms, and the b_j are literals. The meaning of an AD is that whenever all b_j hold, one of the heads f_i will be true with probability p_i, or none of them with probability $1 - \sum_i p_i$. Note that several of the f_i may be true at the same time if they also appear as heads of other rules or ADs.
Thus, the addition program can be written in its probabilistic form as

```
% Set of ground probabilistic facts
0.2::digit(img1, 0); 0.4::digit(img1, 1). % AD 1
0.8::digit(img2, 0); 0.1::digit(img2, 1). % AD 2


% Rule
add(img1, img2, N):- digit(img1,N1), digit(img2,N2), N is N1 + N2.


% Query
```

```
    query(add(img1, img2, 1)).
```

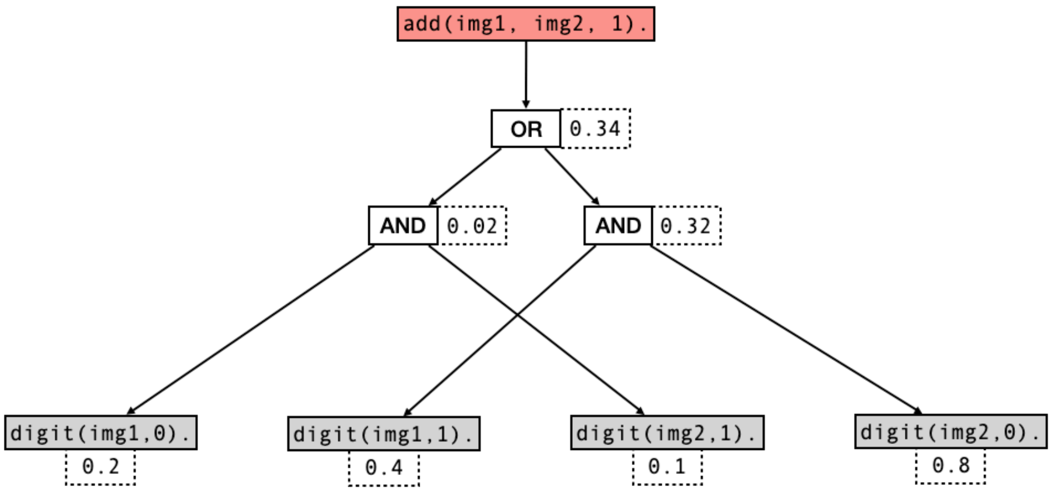
The SDD of this program and the corresponding AC are shown in Figure 1.8, where the query root is coloured in red and the probabilistic facts in grey. The white rectangles correspond to logical operators applied to their children, and the dotted squares next to the nodes show the intermediate results.

Recalling 1.11 and 1.12, the probability of the query `add(img1, img2, 1)` is given by

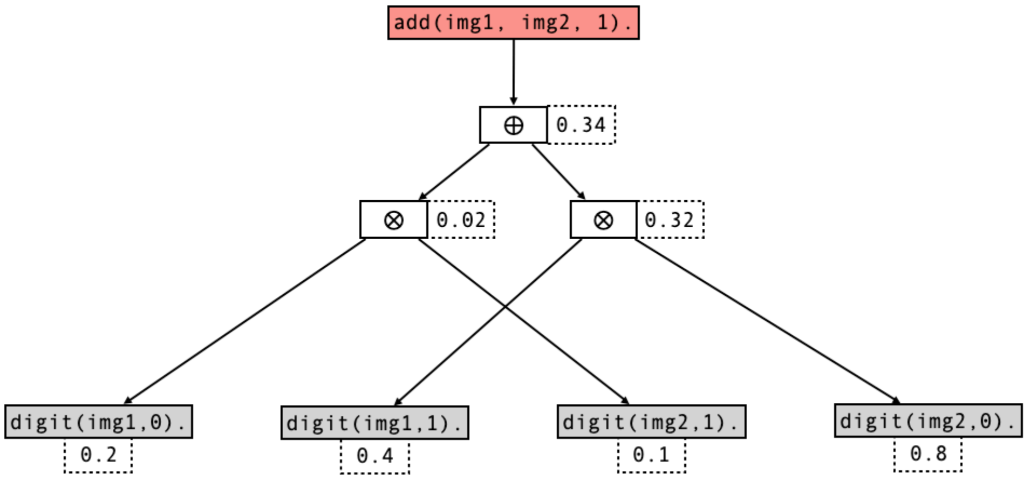
$$\begin{aligned} P(\texttt{add(img1, img2, 1)}) &= P(w_{\texttt{\{digit(img1,0),digit(img2,1)\}}}) + P(w_{\texttt{\{digit(img1,1),digit(img2,0)\}}}) \\ &= 0.2 \times 0.1 + 0.4 \times 0.8 = 0.34. \end{aligned}$$

If we now add the evidence `digit(img1,1)` to the program, the probability of the query becomes

$$\begin{aligned} P(\texttt{add(img1, img2, 1)} \mid \texttt{digit(img1,1)}) &= P(w_{\texttt{\{digit(img1,1),digit(img2,0)\}}}) \\ &= 1 \times 0.8 = 0.8 \end{aligned}$$



(a) Sentential Decision Diagram.



(b) Arithmetic circuit.

Figure 1.8: The SDD and the corresponding AC for query `add(img1,img2,1)`.

1.5.1 DeepProbLog

DeepProbLog [36] is a neural probabilistic logic programming language that incorporates deep learning by means of neural predicates.
Let's consider the addition example of the previous section, and let `img1` and `img2` to be two images of handwritten digits from the MNIST dataset [37], and `N` the natural number corresponding to the sum of these digits. Once trained, DeepProbLog allows us to make a probabilistic estimate on the validity of predicates like `add(` `,` `,1)`. Although such a predicate can be learned directly by a standard neural classifier, this approach cannot incorporate background knowledge like the definition of the addition of two natural numbers. By combining probabilistic-logical programming and neural network, DeepProbLog is able to encode background knowledge in rules, like the one of our example:

```
add(img1, img2, N):- digit(img1,N1), digit(img2,N2), N is N1 + N2.
```

Therefore, the MNIST addition program can be written as

```
% Set of ground probabilistic facts
p_10::digit(img1, 0); p_11::digit(img1, 1). % AD 1
p_20::digit(img2, 0); p_21::digit(img2, 1). % AD 2

% Rule
add(img1, img2, N):- digit(img1,N1), digit(img2,N2), N is N1 + N2.
```

where the probabilities `p_ij` are going to be grounded with the output of the neural network evaluated on the images specified in the query.
Therefore, by adding `query(add(` `,` `,1))` to the program, the evaluation of it leads to the AC shown in Figure 1.9, where the query root is coloured in red and the probabilistic facts in grey. The probability `p_ij` of each probabilistic fact in the leaves is grounded with the corresponding output of the neural network with input `img1` ≡ or `img2` ≡ , and the inference proceeds as in ProbLog.

To jointly train both the neural networks and the Problog model, DeepProbLog relies on the *learning from entailment* [38] and *aProbLog* [39] to compute the gradients and optimize the parameters in order to have an end-to-end trainable framework based on examples.

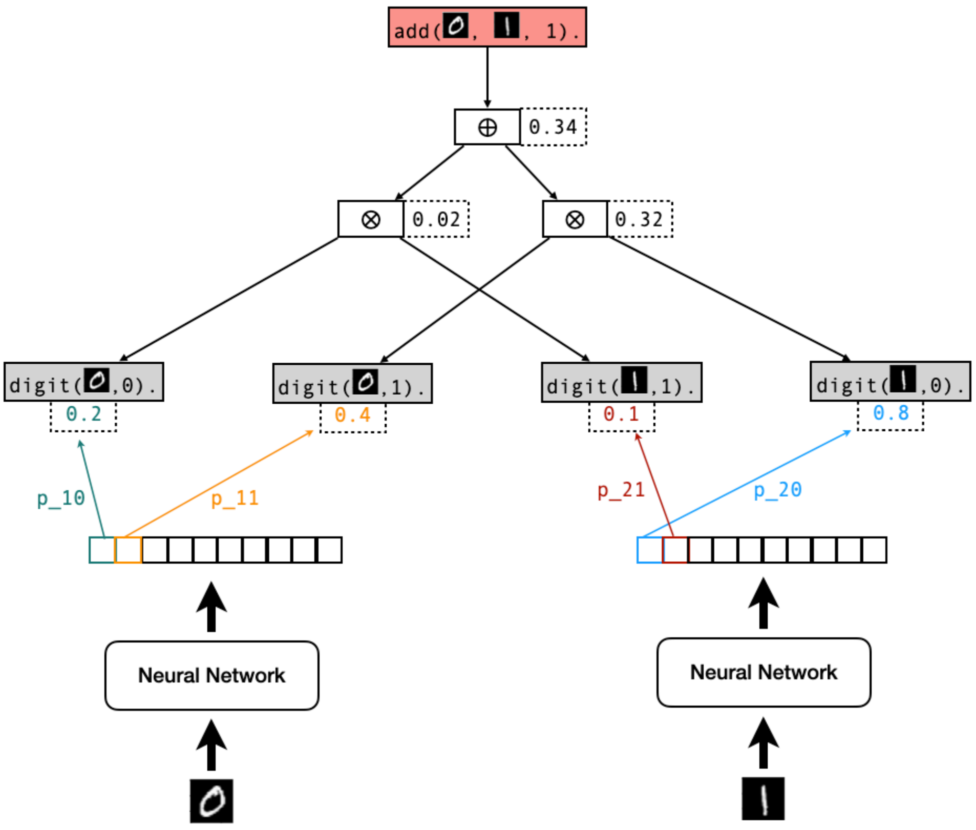


Figure 1.9: The arithmetic circuit for query `add(`, , `,1)`

Learning from entailment is a discriminative training setting in which the examples correspond to facts for a specific target predicate with the evidence residing in the background theory. In particular, given a DeepProbLog program with parameters Θ, a set Q of tuples (q, X, p) with q a query, X the neural input for this query and p its desired success probability, and a loss function $\mathcal{L}$, learning from entailment consists of computing

$$\arg\min_{\Theta} \frac{1}{Q} \sum_{(q,X,p)\in Q} \mathcal{L}(P(q|X,\Theta),p). \tag{1.13}$$

To get a seamless integration between ProbLog and neural network training, DeepProbLog relies gradient-based learning, since the same AC that ProbLog uses for inference can be used for gradient computations as well. In fact, an AC is a differentiable structure, as it is composed of addition and multiplication operations.
The generalization of the ProbLog language and inference called *Algebraic ProbLog* (aProbLog, [39])) provides an extension to arbitrary commutative semirings, including the *gradient semiring* [40]. Whereas ProbLog is confined to only calculating probabilities, the use of this gradient semiring allows aProbLog to calculate the gradient alongside the probabilities, and thus to perform gradient-based learning by deriving the gradient with respect to ProbLog parameters.

In the next section we briefly describe aProbLog and the gradient semiring to introduce the gradient-based learning in ProbLog.

1.5.2 Gradient-based learning in ProbLog

As mentioned in section 1.5, ProbLog annotates each probabilistic fact f with the probability p that f is true (i.e. $p :: f$), which implicitly also defines the probability $1-p$ that the negated fact $\neg f$ is true. Thus, the labeling function L used by ProbLog is defined as

$$\begin{aligned} L(f) &= p && \text{for } p :: f \\ L(\neg f) &= 1 - p && \text{with } L(f) = p \end{aligned}$$

To compute the probability of a query, ProbLog uses the probability semiring with regular addition and multiplication as $\oplus$ and $\otimes$ operators on the corresponding AC. ProbLog probability semiring is given by

$$
\begin{aligned}
&a \oplus b = a + b \\
&a \otimes b = a \times b \\
&e^{\oplus} = 0 \\
&e^{\otimes} = 1
\end{aligned}
$$

where $e^{\oplus}$ and $e^{\otimes}$ refer to the identity element of the binary operation $\oplus$ and $\otimes$ respectively.

aProbLog generalized this idea to any arbitrary commutative semirings. Therefore, instead of probability labels on facts, aProbLog uses a labeling function that explicitly associates values from the chosen semiring with both facts f and their negations $\neg f$, and combines these using semiring addition and multiplication on the AC.
If we consider the gradient semiring, its elements can be described as tuples $(p, \frac{\delta p}{\delta \theta})$, where p is a probability, as in ProbLog, and $\frac{\delta p}{\delta \theta}$ is the partial derivative of that probability with respect to a parameter θ. More precisely, the parameter θ is the learnable probability p_i of a probabilistic fact written as $t(p_i) :: f_i$. This framework is easily extended to a vector of parameters $\boldsymbol{\theta} = [\theta_1, ..., \theta_N]^T$, that can be the concatenation of all N probabilistic parameters in the ground program.
Thus, gradient semiring is defined as follows:

$$
\begin{aligned}
&(a_1, \boldsymbol{a_2}) \oplus (b_1, \boldsymbol{b_2}) = (a_1 + b_1, \boldsymbol{a_2} + \boldsymbol{b_2}) \\
&(a_1, \boldsymbol{a_2}) \otimes (b_1, \boldsymbol{b_2}) = (a_1 b_1, b_1 \boldsymbol{a_2} + a_1 \boldsymbol{b_2}) \\
&e^{\oplus} = (0, \vec{\mathbf{0}}) \\
&e^{\otimes} = (1, \vec{\mathbf{1}})
\end{aligned}
$$

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

To use the gradient semiring for gradient-based learning in ProbLog, we need to transform the ProbLog program into an aProbLog program by extending the label of each probabilistic fact $p :: f$ with the gradient vector of p with respect to the probabilities of all probabilistic facts and ADs present in the program, that is

$$\begin{aligned} L(f) &= (p, \vec{\mathbf{0}}) && \text{for } p :: f \text{ with } p \text{ fixed,} \\ L(f_i) &= (p_i, \boldsymbol{e}_i) && \text{for } t(p_i) :: f_i \text{ with } p_i \text{ learnable,} \\ L(\neg f) &= (1 - p, -\nabla p) && \text{with } L(f) = (p, \nabla p). \end{aligned}$$

where the N-dimensional vector $\boldsymbol{e}_i$ has a 1 in the i-th position and 0 otherwise.
For fixed probabilities, the gradient does not depend on any parameters and thus is $\vec{\mathbf{0}}$, while for the other cases, we use the semiring labels as introduced above.

As we are going to see more in detail in the next chapter, PLVAE is similar to DeepProbLog in the way it connects the AC with the differentiable structure of the neural networks: both of the models uses the outputs of the neural networks as probabilistic facts in the logic program. Thus, their AC is connected with the neural network at the leaves, creating a single differentiable structure through which the gradients can flow.
However, whereas DeepProbLog has been used on MNIST images with one single digit only for classification tasks (Figure 1.9), PLVAE solves conditional generative tasks by exploiting the object detection ability of neural networks to identify multiple digits inside a unique input image. Moreover, differently from DeepProbLog, PLVAE does not apply the neural predicates directly on the input images, but on the latent vector sampled by the inference model (Figure 1.10).

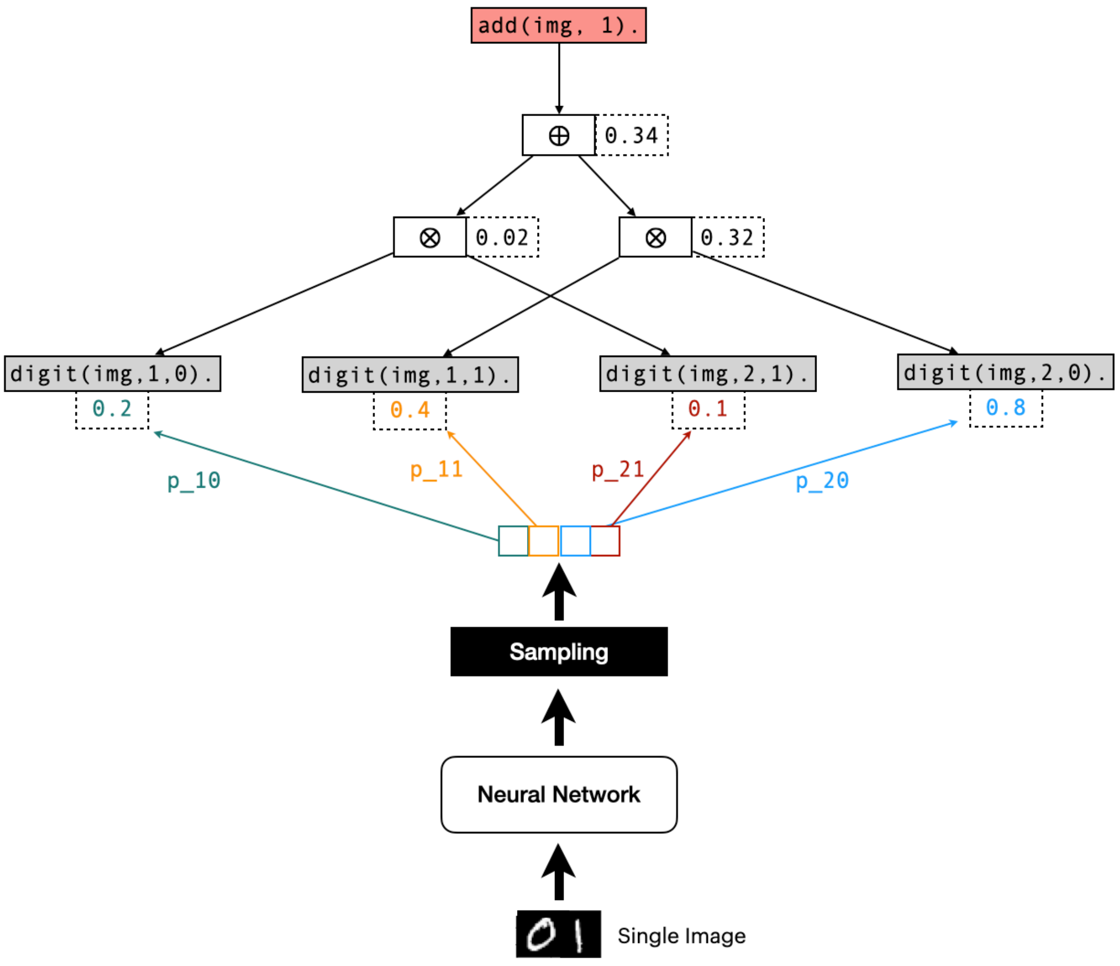


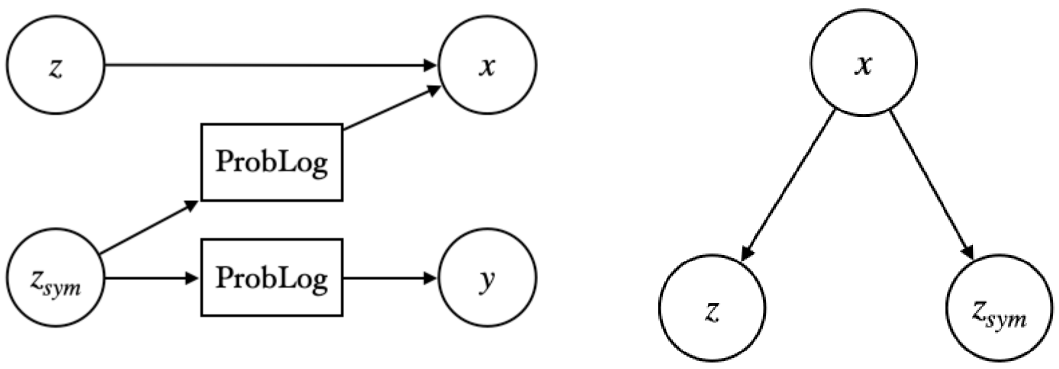
Figure 1.10: The AC of PLVAE for query `add(img,1)` with 2 binary digits.

Chapter 2

Model

2.1 Graphical Model

The core of PLVAE is in its graphical model (Figure 2.1), which allows us to combine the symbolic and sub-symbolic approaches.



(a) Generative model.

(b) Inference Model.

Figure 2.1: Graphical model of PLVAE.

The **generative process** of PLVAE (Figure 2.1a) is based on two latent variables $z_{sym} \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$ that are used to generate new images x as well as to predict their labels y. Both z_{sym} and z have a Gaussian distribution, but they differ in their purpose: whereas z_{sym} is designed to capture information on the probabilities of ProbLog facts, z is intended to capture any other information in the scene. For example, in the MNIST addition program, z_{sym} contains information on the probability of a digit to have a certain value, while z is the sub-symbolical representation of features like the pen width, the spatial orientation of the digit and so on.
In the generative process, z and z_{sym} are unconditionally independent , while x and y are

conditionally independent given z_{sym}, namely

$$z \perp\!\!\!\perp z_{sym},$$
$$x \perp\!\!\!\perp y \mid z_{sym}.$$

This set of independences allows us to factorize the joint distribution of PLVAE generative model as:

$$p_\theta(x, y|\mathbf{z}) = p_{\theta_1}(x|z, z_{sym})p_{\theta_2}(y|z_{sym})p(z)p(z_{sym}) \tag{2.1}$$

where $\mathbf{z} = \{z_{sym}, z\}$, and the posterior distributions of x and y are parameterized by neural networks with parameters $\theta = \{\theta_1, \theta_2\}$.

The posterior distribution of the latent variable $\mathbf{z}$ given the observation x is given by

$$p(\mathbf{z}|x) = \frac{p(x, \mathbf{z})}{p(x)} = \frac{p(x|\mathbf{z})p(\mathbf{z})}{p(x)} \tag{2.2}$$

Since $p(x)$ is not known, the equation (2.2) cannot be solved, thus we cannot directly compute the posterior $p(\mathbf{z}|x)$, but we have to infer it by relying on an inference model.

The **inference model** of PLVAE (Figure 2.1b) is very close to the one of a VAE [41]: both the latent variables z_{sym} and z are inferred from the observations x. Since the exact posterior distribution $p_\theta(\mathbf{z}|x)$ is intractable, we define a variational distribution $q_\phi(\mathbf{z}|x)$ to approximate the true posterior distribution as closely as possible (see Section 1.3.1). Since z and z_{sym} are conditionally independent given x, we can write the inference network as

$$q_\phi(\mathbf{z}|x) = q_{\phi_1}(z|x)q_{\phi_2}(z_{sym}|x). \tag{2.3}$$

The prior of the latent variables $\mathbf{z}$ are chosen to be a standard Gaussian, while its variational posterior is a Gaussian distribution with mean and diagonal covariance parameterized by neural networks $\phi = \{\phi_1, \phi_2\}$.

$$p(\mathbf{z}) = \mathcal{N}(0, 1)$$
$$p(\mathbf{z}|x) = \mathcal{N}(\boldsymbol{\mu}(x), \boldsymbol{\sigma}^2(x))$$

2.2 The prior: ProbLog

A ProbLog program with an evidence $\boldsymbol{e}$ can be defined as a distribution on the possible worlds ω given $\boldsymbol{e}$. This probability distribution is parametrized by the probability p on the probabilistic facts in the program, namely

$$P(\omega \mid \boldsymbol{e}; p)$$

As described in the previous section, the latent space of PLVAE is split into two components, $\mathbf{z} = \{z, z_{sym}\}$, so that z_{sym} captures information on the probabilities of ProbLog facts, and z any other information in the scene. The Gaussian variable z_{sym} is mapped to the multinomial probability distribution p, so that

$$p = f(z_{sym})$$

where the mapping function f in parametrized by the decoder network. Thus, differently from DeepProbLog [36], the neural predicates are not directly applied on the input images, but on the latent vector z_{sym} sampled by the inference model.

PLVAE relies on ProbLog inference for two different tasks, represented as ProbLog blocks in PLVAE generative model (Figure 2.1a): (i) the computation of the query probability $P(q \mid \boldsymbol{e})$ through marginalization; (ii) the sampling of one possible world from the worlds distribution $P(\omega \mid \boldsymbol{e})$ defined by the model.
To allow for gradient-based learning, both the inference tasks must be differentiable. Whereas the first task can be easily made differentiable by relying on *aProbLog* extension (see Section 2.4), the literature does not offer any method to perform differentiable sampling in ProbLog. In this work, we developed a simple solution that replaces DNF sampling [42] with multiple marginalization operations. We define a query for each world and we compute $P(\omega \mid \boldsymbol{e})$ by relying on ProbLog marginalization. Then, we sample $\omega \sim P(\omega \mid \boldsymbol{e})$ method by relying on *Gumbel-Softmax Reparametrization Trick* ([29], Section 1.3.1).
Unfortunately, the proposed solution strongly limits the model scalability. Thus, one of the future research direction should be the development of new methods to effectively exploit knowledge compilation principles to define differentiable sampling for ProbLog inference.

Let's see more in detail how PLVAE integrates ProbLog reasoning power into the generative model by considering the MNIST addition example introduced in section 1.5.1. Instead of using raw MNIST images, we now consider input images with more than one digit to simulate a scene with multiple objects (e.g.). The high-level diagram of PLVAE working on this example is shown in Figure 2.2.
As described in the previous section, the encoder of PLVAE provides the latent representation $\mathbf{z} = \{z, z_{sym}\}$ of the input image, and the decoder uses the information contained in z_{sym} as prior for the probabilistic facts. More precisely, z_{sym} is mapped to the probability p_{ij} of ProbLog facts, that in our example corresponds to the probability that the i-th digit in the image has value j.
By doing so, we force the model to provide a disentangled representation of the scene, where z_{sym} contains the information regarding objects entity, and z represents any other feature that is not encoded in the probabilitic-logic program.
Once the probabilities have been grounded, PLVAE predicts the labels y of the input and the probability distribution over all the possible worlds by relying on ProbLog inference, as previously described. In the MNIST addition example, y corresponds to the sum of the two digits in the image and the possible worlds are all the possible pairs of values taken by the digits. Finally, PLVAE samples one possible world according to the worlds distribution, and gives it as input to the decoder, along with the latent variable z, to reconstruct the original image.

To effectively exploits PLVAE in the MNIST addition example, we need to slightly modify the ProbLog program to consider one single image with multiple digits and retrieve the probability distribution of all the possible pairs of digits. Thus, the ProbLog program used by PLVAE becomes

```
p_10::digit(img, 1, 0); p_11::digit(img, 1, 1). % AD 1
p_20::digit(img, 2, 0); p_21::digit(img, 2, 1). % AD 2

add(img, N):- digit(img, 1, N1), digit(img, 2, N2), N is N1 + N2. % R1
digits(X1,X2):- digit(img, 1, X1), digit(img, 2, X2). % R2

query(add(img, 1)). % Query A
```

```
    query(digits(X1,X2)). % Query B
```

Once evaluated, *Query A* gives us the probability that the digits in the images sum up to 1, while with *Query B* we retrieve the categorical distribution of the 4 possible pairs of binary digits.

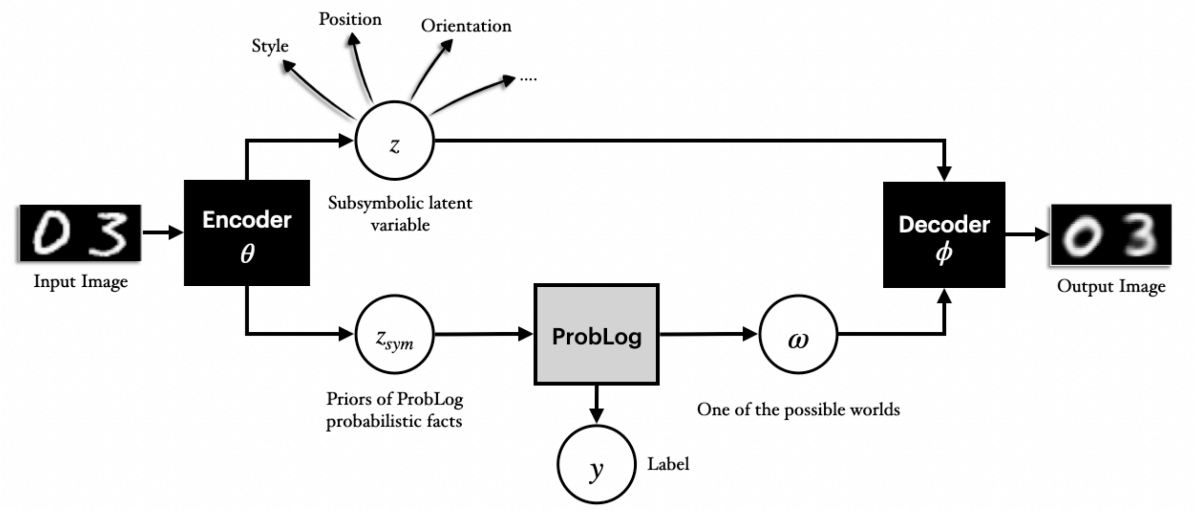

Position
Orientation
Style
....
z
Subsymbolic latent variable
Encoder θ
Input Image
Decoder ϕ
Output Image
z_sym
ProbLog
ω
Priors of ProbLog probabilistic facts
One of the possible worlds
y
Label

Figure 2.2: High-level diagram of PLVAE working on MNIST addition example. The encoder network θ represents the input image with the low-dimensional vector $\mathbf{z} = \{z_{sym}, z\}$. Then, by using the information contained in z_{sym}, ProbLog provides the label y of the input image (i.e. the sum of the two digits) and the probability distribution over the possible worlds (i.e. all the possible the pairs of digits). According to this distribution, the model extracts one single world ω (e.g. the first digit is a 0 and the second one a 3), and passes it as input to the decoder ϕ along with z. Finally, the decoder network reconstructs the input image by combining the information contained in z with the world ω.

2.3 Objective Function

The objective function of PLVAE is given by the following evidence lower bound

$$\mathcal{L}(\theta,\phi) = \mathcal{L}_{REC}(\theta,\phi) + \mathcal{L}_{Q}(\theta,\phi) - \mathcal{D}_{\mathcal{KL}}[q_\phi(\mathbf{z}|x)||p(\mathbf{z})]] \tag{2.4}$$

where

$$\mathcal{L}_{REC}(\theta,\phi) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|x)}[log(p_\theta(x|\mathbf{z})], \tag{2.5}$$

$$\mathcal{L}_{Q}(\theta,\phi) = \mathbb{E}_{z_{sym}\sim q_\phi(z_{sym}|x)}[log(p_\theta(y|z_{sym})]]. \tag{2.6}$$

Derivation. To derive the ELBO defined in (4.1) we start from the maximization of the log-likelihood of the input image x and the class y, namely

$$\log(p(x,y)) = \log\left(\int p(x,y|\mathbf{z})d\mathbf{z}\right). \tag{2.7}$$

Recalling the generative network factorization (2.1), we can write

$$\log(p(x,y)) = \log\left(\int p_\theta(x|z,z_{sym})p_\theta(y|z_{sym})p(z)p(z_{sym})dzdz_{sym}\right) \tag{2.8}$$

Then, by introducing the variational approximation $q_\phi(\mathbf{z}|x)$ to the intractable posterior $p_\theta(\mathbf{z}|x)$ and applying the factorization (2.3), we get

$$\log(p(x,y)) = \log\left(\int \frac{q_\phi(z|x)q_\phi(z_{sym}|x)}{q_\phi(z|x)q_\phi(z_{sym}|x)} p_\theta(x|z,z_{sym})p_\theta(y|z_{sym})p(z)p(z_{sym})dzdz_{sym}\right). \tag{2.9}$$

We now apply the *Jensen's inequality* [43] to equation (2.9) and we obtain the lower bound for the log-likelihood of x and y given by

$$\int q_\phi(z|x)q_\phi(z_{sym}|x)\log\left(p_\theta(x|z,z_{sym})p_\theta(y|z_{sym})\frac{p(z)p(z_{sym})}{q_\phi(z|x)q_\phi(z_{sym}|x)}dzdz_{sym}\right). \tag{2.10}$$

Finally, by relying on the linearity of expectation and on logarithm properties, we can rewrite equation (2.10) as

$$\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|x)}\left[\log(p_\theta(x|\mathbf{z}))\right] + \mathbb{E}_{z_{sym}\sim q_\phi(z_{sym}|x)}\left[\log(p_\theta(y|z_{sym}))\right] + \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|x)}\left[\log\left(\frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|x)}\right)\right].$$

The last term is the negative Kullback-Leibler divergence between the variational approximation $q_\phi(\mathbf{z}|x)$ and the prior $p(\mathbf{z})$. This leads us to the ELBO of equation (4.1), that is

$$\begin{aligned}\log(p(x,y)) &\geq \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|x)}\left[\log(p_\theta(x|\mathbf{z}))\right] + \mathbb{E}_{z_{sym}\sim q_\phi(z_{sym}|x)}\left[\log(p_\theta(y|z_{sym}))\right] - \mathcal{D}_{\mathcal{KL}}[q_\phi(\mathbf{z}|x)||p(\mathbf{z})]\\ &:= \mathcal{L}(\theta,\phi).\end{aligned}$$

Estimation. To estimate the ELBO and its gradients w.r.t. the model parameters, we rely on the Monte Carlo estimates of expectations [41]. Since both $q_\phi(\mathbf{z}|x)$ and $p(\mathbf{z})$ are chosen to be Gaussian distributions, the Kullback-Leibler divergence in (4.1) can be integrated analytically by relying on its closed form. Thus, only the expected reconstruction and query errors $\mathcal{L}_{REC}(\theta,\phi)$ and $\mathcal{L}_Q(\theta,\phi)$ require estimation by sampling.
We can therefore define the ELBO estimator as

$$\mathcal{L}(\theta,\phi) \approx \tilde{\mathcal{L}}(\theta,\phi;\epsilon) = \tilde{\mathcal{L}}_{REC}(\theta,\phi;\boldsymbol{\epsilon}) + \tilde{\mathcal{L}}_Q(\theta,\phi;\boldsymbol{\epsilon}) - \mathcal{D}_{\mathcal{KL}}[q_\phi(\mathbf{z}|x)||p(\mathbf{z})]. \tag{2.11}$$

The estimators of $\mathcal{L}_{REC}$ and $\mathcal{L}_Q$ can be write as

$$\tilde{\mathcal{L}}_{REC}(\theta,\phi;\boldsymbol{\epsilon}) = \frac{1}{N}\sum_{n=1}^{N}(\log(p_\theta(x|\hat{\mathbf{z}}^{(n)}))) \tag{2.12}$$

$$\tilde{\mathcal{L}}_Q(\theta,\phi;\boldsymbol{\epsilon}) = \frac{1}{N}\sum_{n=1}^{N}(\log(p_\theta(y|\hat{z}^{(n)}_{sym}))) \tag{2.13}$$

where

$$\begin{aligned}&\hat{\mathbf{z}}^{(n)} = \{\hat{z}^{(n)}, \hat{z}^{(n)}_{sym}\} := \boldsymbol{\mu}(x) + \boldsymbol{\sigma}(x)\boldsymbol{\epsilon}^{(n)},\\ &\boldsymbol{\epsilon}^{(n)} \sim \mathcal{N}(0,1).\end{aligned}$$

In Algorithm 1 we summarize the training steps of PLVAE.

Algorithm 1: PLVAE Training.

```
Data: Set of images $\mathcal{X}$
$\theta, \phi \leftarrow$ Initialization of paramters
repeat
    Forward Phase
    $\mathcal{X}^N \leftarrow$ Random minibatch of $N$ images
    $\boldsymbol{\epsilon} \leftarrow$ Random samples from $\mathcal{N}(0, 1)$
    $\omega \leftarrow$ ProbLog Inference (possible world)
    $y \leftarrow$ ProbLog Inference (image label)
    $\tilde{\mathcal{X}}^N \leftarrow$ Reconstruction of $\mathcal{X}^N$
    Backward Phase
    $\mathbf{g} \leftarrow \nabla_{\theta,\phi}\tilde{\mathcal{L}}(\theta, \phi; \boldsymbol{\epsilon})$ (gradients estimator (2.11))
    $\theta, \phi \leftarrow$ Update parameters using gradients $\mathbf{g}$
until convergence of parameters $(\theta, \phi)$;
```

2.4 Learning

During the training, we aim at maximizing $\mathcal{L}(\theta, \phi)$ with respect to both the encoder and the decoder parameters, we therefore need to compute the gradient w.r.t. θ and ϕ. Since any sampling operation prevents back-propagation, we need to reparametrize the two sampled variables $\mathbf{z}$ and ω. Due to their nature, we use the well-known *Reparametrization Trick* [28] for the Gaussian $\mathbf{z}$, while we exploit the *Categorical Reparametrization with Gumbel-Softmax* [29] for the discrete variable ω corresponding to the sampled possible world.
In particular, by defining ω as the one-hot encoding of the possible worlds, we have

$$\hat{\omega}_i = \frac{\exp((\log \pi_i + \hat{g}_i)/\lambda}{\sum_{j=1}^{J} \exp((\log \pi_j + \hat{g}_j)/\lambda)}, \text{ with } \hat{g}_i \sim \mathit{Gumbel}(0, 1) \tag{2.14}$$

where J is the number of possible worlds (e.g. all the possible pairs of digits), and π_i depends on z^i_{sym}, which is reparametrized with the Gaussian Reparametrization Trick.

Training PLVAE consists of jointly learning not only the neural networks parameters, but also the probabilistic ones. We therefore rely on the *learning from entailment* [38] and *aProbLog* [39] to have an end-to-end trainable framework based on examples, as in DeepProbLog ([36], Sections 1.5.1).
As mentioned in Section 1.5.2, the learnable probability p_i of a probabilistic fact can be optimized by using the gradient semiring. This semiring allows us to calculate the gradient of the

probability of the query w.r.t. p_i, namely $\frac{\delta P(q)}{\delta p_i}$. Then, we can directly update p_i by performing gradient-based learning performed along the arithmetic circuit of the program.

Since the outputs of the neural networks are used as probabilistic facts in the logic program, the AC is connected with the differentiable structure of the neural network at the leaves. Thus, we have a single differentiable structure through which the gradients can flow.

In Figure 2.3 we represent the AC for the addition example evaluated using the gradient semiring. The gradient vectors with respect to the probabilities of all the probabilistic facts in the program are annotated in green, and the query root in red. Differently from DeepProbLog (Figure 1.9), we have one single image with multiple digits and the neural predicates are not applied directly on the images, but on the latent vector sampled through the reparametrization trick.

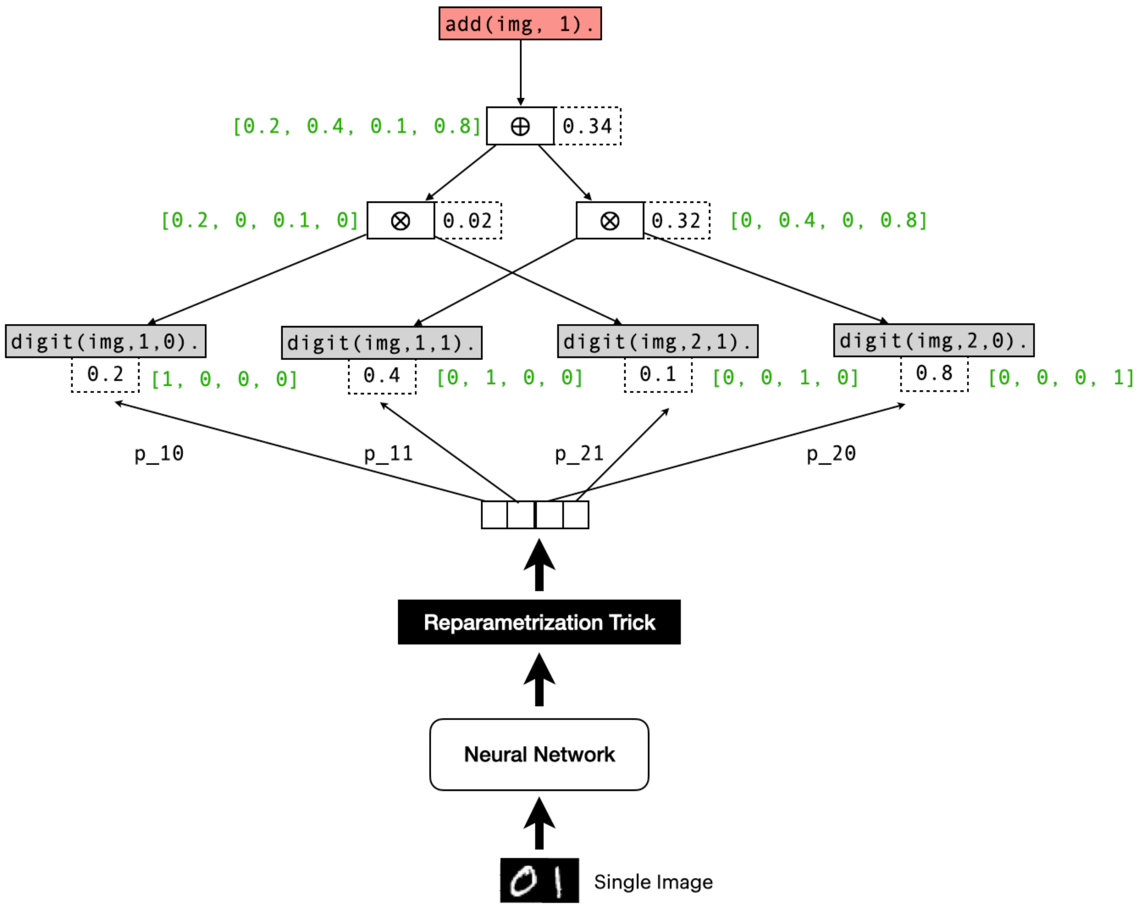


Figure 2.3: The arithmetic circuit of PLVAE for query `add(img,1)` with two binary digits evaluated using the gradient semiring.

Chapter 3

Related Work

Kingma et al. [19] introduced a supervision in the VAE framework by defining the *M2 model*, where data is generated by a latent class variable y in addition to a continuous latent variable z. This model allows for the conditional generation of new samples with a clear separation between the supervised content and the remaining elements, by fixing the class label y, and then sampling different values of the latent variables z.

Maaløe et al. [10] extended the basic M2 model with an auxiliary latent variable that increases the flexibility of the variational posterior and allows to model more complex latent distributions.

Another extension of the M2 model is represented by the works of Sønderby et al. [11] and Maaløe et al. [12], in which they introduce hierarchical structures in the prior to achieve better generative performance. A recent work [13] focused on designing neural architectures for hierarchical VAEs achieves state-of-the-art results on large high-quality images by exploiting novel network architecture modules and parameterization of approximate posteriors.

Recent works extend the VAE framework with new types of prior, such as a mixture distribution with components given by variational posteriors [15] or a learnable hierarchical prior [14]; others use rejection sampling with a learned acceptance function [16] or combine VAEs with autoregressive models [17]. Differently from PLVAE, where we have the fixed structure given by the logic probabilistic program, these models introduce a more flexible prior definition to learn complex functions. However, the structure of PLVAE is interpretable and can be easily changed in order to apply the model to different scenarios. Moreover, the logic probabilistic program allows to directly interpret the encoder output without the need of a manual supervision, that is often demanding.

Another research direction is represented by the work of Tom Joy et al. [18], who proposed a model to capture label characteristics in VAEs avoiding any direct correspondence with the label itself. Their idea is to learn representations by capturing label characteristics explicitly. Although they perform variational inference by treating the label y as a partially observed auxiliary variable, *CCVAE* is not a hierarchical VAE.

All of these models represent remarkable advances in the VAE framework achieving state-of-the-art results, but they are not able to generalize to novel tasks. For example, let's suppose to train a VAE model on a supervised dataset composed by pairs of digits labeled with their sum. In this setting, generating two numbers that sum up to 3 is an easy task even for the original M2 model, but none of the previously mentioned models would be able to answer questions like "Generate two numbers that multiply to 5." or "Generate two numbers that differ by 1.". The reason is that VAEs rely only on sub-symbolic representations of the scene, without including symbolic reasoning on its content. Conversely, PLVAE exploits the structuring and reasoning power of probabilistic-logic programming to learn symbolic relationships in the latent space which allow for a high degree of generalization.

Recently, Jiang at al. [23] proposed a neuro-symbolic generative model, called *GNM*, that combines the benefits of sub-symbolic and symbolic representations. Their model is based on a two-layer latent hierarchy, where the top layer provides a global sub-symbolic latent variable for flexible density modeling and the bottom layer a symbolic latent map for structured representation. This dual representation allows to generate novel scenes by controlling the structured representation of an object, such as its position, colour, etc. However, although GNM generates a symbolic representation of the scene, it is still not able to manipulate this knowledge in order to answer new generative questions. This is because GNM uses the symbolic engine only to give a structure to the latent representation, and does not fully exploit its power as a reasoning tool, as in PLVAE.
In their work [24], R. Feinman and B. M. Lake combine neural networks and symbolic probabilistic programs to learn a generative model. They propose a generative model of novel handwritten characters, called *Full NS*, that represents a character as a sequence of strokes, with each stroke decomposed into a starting location and a stroke trajectory. However, Full NS does not use logic and relies on a probabilistic program only to render each stroke to an image

canvas and not as a reasoning tool, as in PLVAE. Moreover, the partitioning of characters in strokes prevents to tackle any generative tasks that involve reasoning upon the semantics of a character.

As described in Section 1.5.1, PLVAE is similar to *DeepProblog* [36] in the use of *aProbLog* [39] to compute the gradients and optimize the parameters. However, differently from DeepProbLog, PLVAE does not apply the neural predicates directly on the input images, but on the latent vector sampled by the inference model. Furthermore, whereas DeepProbLog has been used only for discriminative tasks, PLVAE solves conditional generative tasks by exploiting the object detection ability of neural networks to identify multiple objects inside a scene.

In Table 3.1 we compare the generative models mentioned above in terms of their prior. On one hand, we have VAE frameworks with structured priors that model statistical relationships in the latent space (*Structured Sub-symbolic*). However, despite their flexibility in learning complex priors, these models are not able to generalize to novel tasks, since their latent structure is still pure sub-symbolical and does not allow any form of reasoning.
On the other hand, we have generative models with priors that allow for a symbolic structure in the latent space, but this structure cannot be used to solve reasoning tasks (*Structured Symbolic*). Furthermore, the dependencies among the variables are not interpretable and we cannot extract rules as in PLVAE.
Conversely, PLVAE prior gives a structure to the latent space that can be easily used by a probabilistic-logic programming to generalize to previously unseen tasks that require to reason upon the content of a scene (*Logic-based Symbolic*).

	Structured Sub-symbolic	*Structured Symbolic*	*Logic-based Symbolic*
M2[19]	**Yes**	No	No
AVAE [10]	**Yes**	No	No
LVAE [11]	**Yes**	No	No
BIVA [12]	**Yes**	No	No
NVAE [13]	**Yes**	No	No
VampVAE [15]	**Yes**	No	No
VHP [14]	**Yes**	No	No
LARS [16]	**Yes**	No	No
VLAE [17]	**Yes**	No	No
CCVAE [18]	No	**Yes**	No
GNM [23]	No	**Yes**	No
FullNS [24]	No	**Yes**	No
PLVAE	No	No	**Yes**

Table 3.1: Models comparison.

Chapter 4

Results

By virtue of the integration of probabilistic-logic programming into the VAE framework, PLVAE is able to generate new images starting from a logic formula. For example, we can ask the model to *generate two numbers that sum up to 3*. Moreover, once trained on a base task, PLVAE can generalize to any other task involving reasoning with the same symbols, since the logic program allows us to define the relationship between images and labels. Furthermore, thanks to the structure of its graphical model, PLVAE also preserves the predictive ability of VAEs in the supervised setting.
In the following sections, we describe the experiments we performed to evaluate our approach. Whenever possible, we compare PLVAE against CCVAE [18] (see Appendix A.2 for the implementation details).

4.1 Base Task: Two Digits Addition

To validate our approach, we created a supervised dataset of $28,000$ images of two digits taken from the MNIST dataset [37]. Each image has dimension 28×56 and is labelled with the sum of its digits (Figure 4.1). Since our purpose is to give a proof-of-concept, we focused on a simplified, but still interesting task by considering digits with values from 0 to 2. Whereas the number of digits in each image is known, their spatial position is not specified and needs to be learnt by the model. By doing so, we can easily extend the framework to images with several objects in different positions.

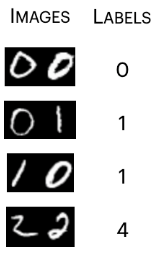

IMAGES
LABELS
0
1
1
4

Figure 4.1: Example of data from the two MNIST digits addition dataset.

The ProbLog program used to train PLVAE for the base tasks is the MNIST addition program introduced in Section 2.2, namely

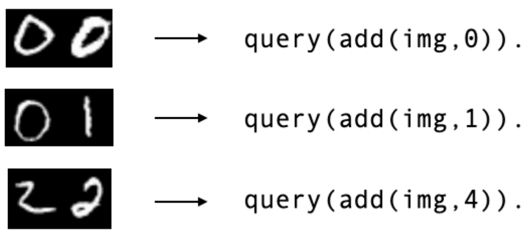
```
% ADs
p_10::digit(img, 1, 0); p_11::digit(img, 1, 1); p_12::digit(img, 1, 2).
p_20::digit(img, 2, 0); p_21::digit(img, 2, 1); p_22::digit(img, 2, 2).

add(img, N):- digit(img, 1, N1), digit(img, 2, N2), N is N1 + N2. % R1
digits(X1,X2):- digit(img, 1, X1), digit(img, 2, X2). % R2
```

According to the specific task, we define the corresponding queries and evidences. For example, in the two digits addition example we have:

```
query(add(img, 3)). % Query A
query(digits(X1,X2)). % Query B
```

where the second term of *Query A* changes according to the label of the input image in order to use only positive examples for training (Figure 4.2).



```
query(add(img,0)).
query(add(img,1)).
query(add(img,4)).
```

Figure 4.2: During the training of PLVAE we only use positive examples (i.e. query with desired success probability $p = 1$), and we change *Query A* accordingly.

4.2 Evaluation

We evaluate our approach on three aspects: (i) reconstruction ability, (ii) predictive ability and (iii) generative ability.
We use binary cross entropy for the reconstruction ability (m_{REC}), while for the predictive ability we rely on classification accuracy on the true labels (m_{CLASS}). To measure the generative ability (m_{GEN}), we trained an independent classifier on MNIST dataset (see Appendix A.3 for the implementation details), and we used it to evaluate conditionally generated samples.
The evaluation process for generative ability (Figure 4.3) can be summarized as: (i) generate an image with conditioning label y; (ii) split the image in two sub-images[1]; (iii) then, sum together the outputs of the pre-trained classifier evaluated on the sub-images; (iv) compare the resulting label $\tilde{y}$ with y.

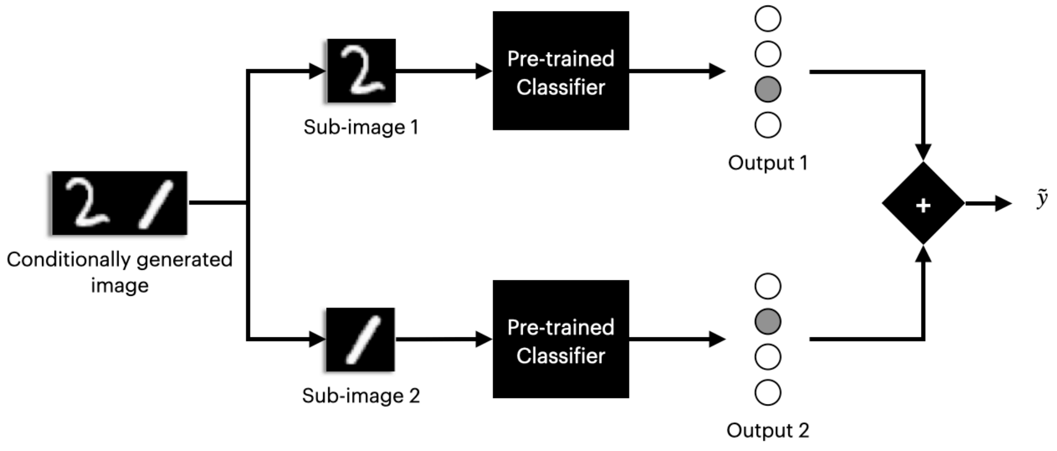


Figure 4.3: Process to assess generative accuracy.

[1]We want to stress that the split of the image is only used for evaluation purposes and it is never used during training.

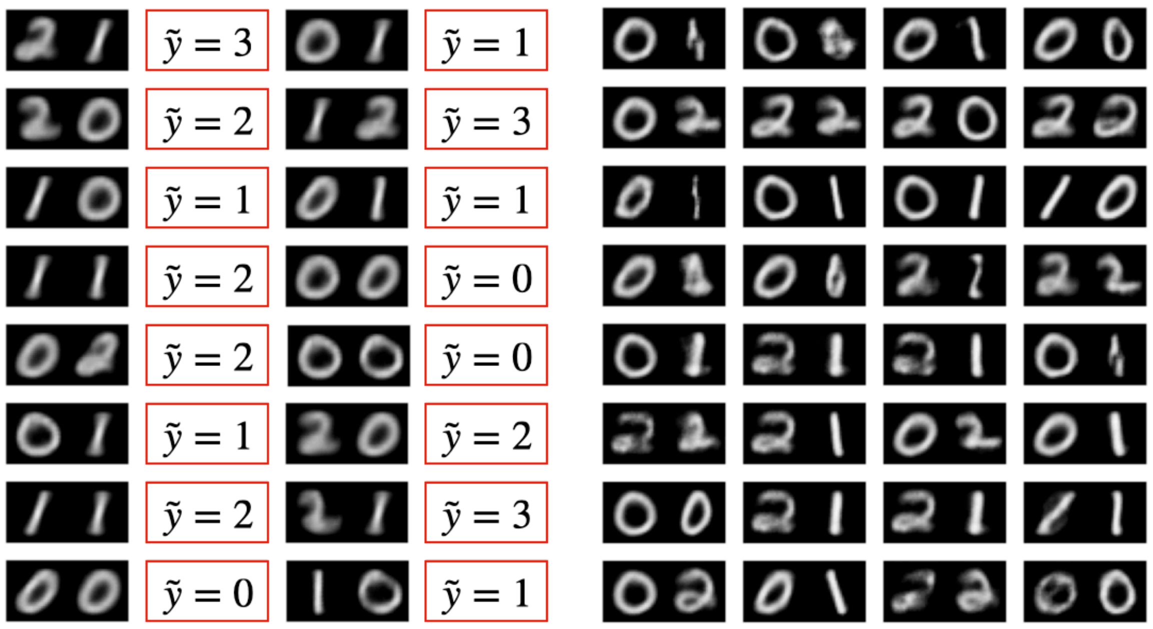
4.3 Experiment 1: Generation

In the generative model of PLVAE (Figure 2.1a) both the image x and the label y are leaf nodes. Thus, differently from other VAE frameworks [10, 18, 19], where the label y is at the top of the generative process, PLVAE jointly generates a new image and the corresponding label directly from the latent vector $\mathbf{z} \sim p(\mathbf{z})$.
We can summarize the generative process of PLVAE as follows:

1. sampling $\widehat{\mathbf{z}} = \{\hat{z}_{sym}, \hat{z}\}$ from the prior distribution $\mathcal{N}(0, 1)$;
2. given $\hat{z}_{sym}$, inferring the distribution over the possible worlds through ProbLog;
3. sampling a world ω according to the distribution over the possible worlds;
4. given the world ω, inferring the distribution $p(y|\omega)$ over the labels through ProbLog;
5. sampling the label y according to $p(y|\omega)$;
6. generating a new image from ω and $\hat{z}$.

The process described above is different from the one performed during the training, where y is obtained from z_{sym} through an expectation. The reason is that currently there are no methods to perform differentiable sampling in ProbLog, and therefore we replace the sampling with multiple marginalization operations to allow for an efficient gradient-based learning (see Section 2.2). However, by doing so the generation of ω and y are made independent, and thus they may not be consistent with each other.

In Figure 4.4 we provide an example of generation with PLVAE and CCVAE. PLVAE generates both the image and the label $\tilde{y}$ starting from the sampled latent vector $\mathbf{z} \sim \mathcal{N}(0, 1)$. Conversely, CCVAE generative process starts by sampling the label y from its prior, then proceeds by sampling the latent vector from $p(\mathbf{z}|y)$, and finally generates the new image.



(a) PLVAE. (b) CCVAE.

Figure 4.4: Generation for PLVAE and CCVAE. Differently from CCVAE, PLVAE jointly generates both the image and the corresponding label $\tilde{y}$ (red box) from the latent vector.

4.4 Experiment 2: Conditional Generation

As described in Section 1.5, ProbLog allows us to specify *evidences* in a program, that are observations on which we want to condition the probability of the query. Thus, to generate a new pair of digits that sum up to a desired number, we simply replace *Query A* with the corresponding evidence $\mathbf{e}$. For example, by replacing *Query A* with `evidence(add(img,2))`, the evaluation of *Query B* results in the probability distribution of the possible worlds given this evidence, namely $P($`digits(X, Y)` $\mid$ `add(img,2)`$)$. The resulting distribution will assign zero probability to those pairs of digits that do not sum to 2, and therefore the sampled worlds ω is guaranteed to be compatible with the desired evidence.
The process of generating a new image according to a given evidence $\mathbf{e}$ can be summarized as follows: (1) sampling $\widehat{\mathbf{z}} = \{\hat{z}_{sym}, \hat{z}\}$ from the prior distribution $\mathcal{N}(0, 1)$; (2) given $\hat{z}_{sym}$ and the evidence $\mathbf{e}$, inferring the distribution over the possible worlds through ProbLog; (3) sampling a world ω according to the distribution over the possible worlds; (4) passing ω and $\hat{z}$ as input to the decoder to generate the image.

We provide qualitative results of conditional generation for PLVAE in Figure 4.5, in which we

can see that the model is able to generate pairs consistent with the evidence and with a variety of styles and combinations of digits.
On the contrary, although CCVAE achieves higher classification accuracy (Table 4.1), it is not completely able to conditionally generate new images (Figure 4.6), since part of the information on the image content is also contained in $\mathbf{z}_{\setminus c}$.

Intervention. The clear separation between the symbolic and sub-symbolic engines allows us to perform interventions by changing the value of one or more digits in an image. To this end, instead of sampling $\widehat{\mathbf{z}}$ from its prior, we use the latent representation of a target image, and we proceed with steps (2) to (4) as for the conditional generation.
The examples shown in Figure 4.7 confirm that PLVAE is able to keep the style of an image while changing its content according to the provided evidence. Moreover, since the latent representation $\mathbf{z}$ of the original image is held constant along each row, the new pairs of digits reflect the information on the probabilistic facts contained in z_{sym}. For example, by looking at the first row, we can notice that, whenever it is consistent with the evidence, PLVAE generates pairs of digits with a 1 in the first position, as in the original image (e.g. columns corresponding to evidences 1 and 2).
Conversely, we cannot safely perform interventions with CCVAE, since part of the information on the image content is also contained in $\mathbf{z}_{\setminus c}$, preventing the model from correctly generating new pairs of numbers starting from the style of another image (Figure 4.8).

Classification. Along with conditional generation and intervention, one of the most prevalent tasks that VAEs are required to solve in a supervised setting is the prediction of image labels. The graphical model of PLVAE allows us to preserve the predictive ability of VAEs by relying on ProbLog inference. During the training, the term $\mathcal{L}_Q(\theta, \phi)$ of the loss function (1.8) forces PLVAE to adjust its weights to maximize the probability of *Query A* for all training examples. The classification accuracy is reported in Table 4.1.

	Model	$m_{REC}(\downarrow)$	$m_{CLASS}(\uparrow)$	$m_{GEN}(\uparrow)$
Base Task	PLVAE	0.1289 ± 0.0808	0.8773 ± 0.0718	0.6867 ± 0.2024
(addition)	CCVAE	0.1001 ± 0.0668	0.9510 ± 0.0153	0.6139 ± 0.0748

Table 4.1: Reconstruction, predictive and generative ability of PLVAE and the modified CCVAE trained on the base task.

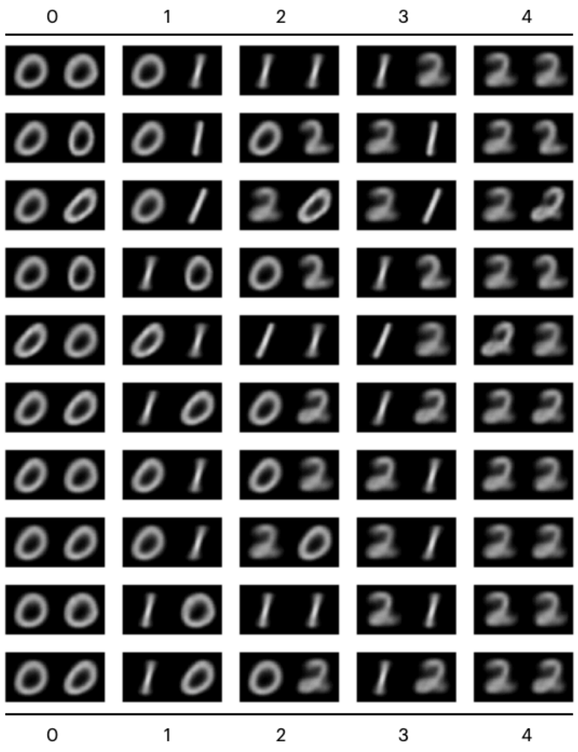


Figure 4.5: Conditional generation for PLVAE. Each column refers to a different evidence, from `add(img,0)` to `add(img,4)`; the latent vector $\mathbf{z}$ is held constant along each row.

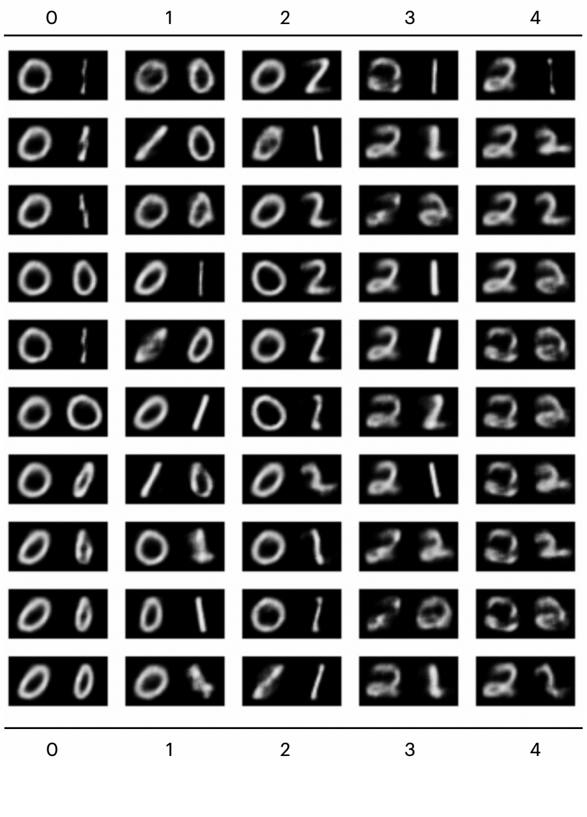


Figure 4.6: Conditional generation for CCVAE. Each column refers to a different sum, from 0 to 4; the latent vector $\mathbf{z}_c$ is held constant along each row.

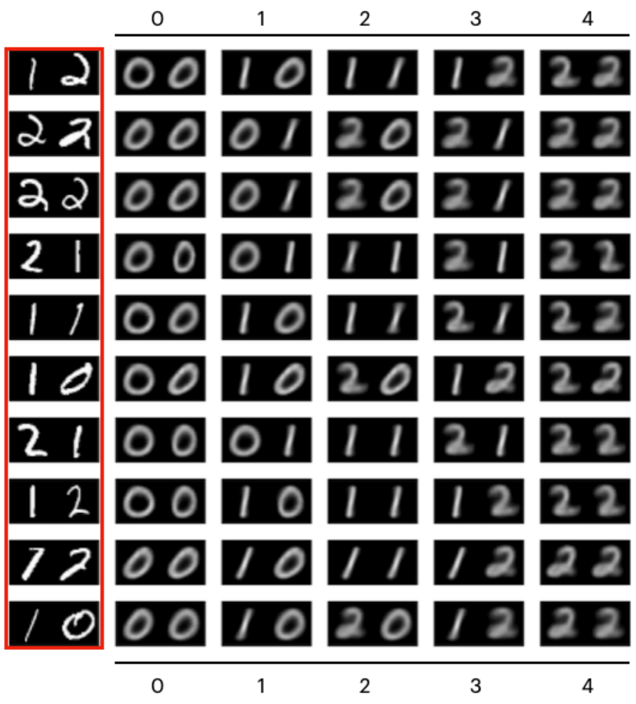


Figure 4.7: Intervention for PLVAE. The original images are displayed in the first column (red), and their latent representation $\mathbf{z}$ is held constant in the other columns obtained with evidences from `add(img,0)` to `add(img,4)`.

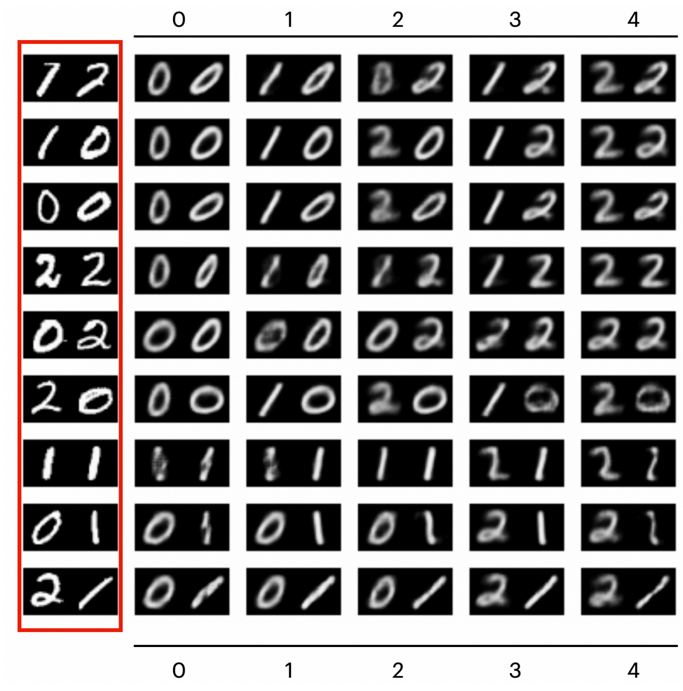


Figure 4.8: Intervention for CCVAE. The original images are displayed in the first column (red), and their latent representation $\mathbf{z}_{\setminus c}$ is held constant in the other columns obtained by conditioning on different values of sum evidences from 0 to 4.

4.5 Experiment 3: Task Generalization

Dici il perche': questo e' dovuto al fatto che il prior permette di esprire un task generativo a partire da due immagini mentre il link con la label e' gestito da un programma logico. Fintantoche' puoi definire un programma logico che mappa il tuo task iniziale come la generazione di una coppia di immagini allora generalizza.

As we have seen, PLVAE is able to generate new images starting from a logic formula, like the addition of two numbers in the previous example. By virtue of its probabilistic-logic prior, PLVAE can also generalize to previously unseen tasks. In particular, once trained on a specific symbolic task, PLVAE is able to generalize to any novel task that involves reasoning with the same set of symbols. For example, by training PLVAE on the addition task, we can easily use the same model to generate pairs of digits that multiply to a given value. The reason is that PLVAE factorizes the generation task into two steps: (i) generation of the digits labels; (ii) generation of the image given the labels. Whereas the second step requires to be parameterized by a black-box model (e.g. a convolutional neural network), which is hard to explicitly define, the labels generation can be easily handled by an explicit generative process. The symbolic nature of the labels generation allows its substitution with any arbitrary probabilistic logic program that maps to the same set of integer labels.
Therefore, to generate two numbers that multiply to a given value, we need to slightly modify the ProbLog program by substituting the addition rule (*R1*) with the multiplication rule, and using the evidence accordingly:

```
% New Rule
mul(img, N):- digit(img, 1, N1), digit(img, 2, N2), N is N1 * N2.


% New Evidence
evidence(mul(img,0)).
```

The same process can be extended to any task involving two integer numbers that can be expressed in the logic program, regarding its complexity.
As best of our knowledge, such a level of task generalization cannot be achieved by any other VAE frameworks. Moreover, since PLVAE allows us to jointly generate new images and the corresponding labels (Section 4.3), we can use a model trained on a base task to generate new supervised datasets in which the desired relationships among objects in the scene are expressed

by the ProbLog program.

We provide examples of the generalization ability of PLVAE by working with tasks like the multiplication (Figures 4.9), the subtraction (Figure 4.10) or the power (Figure 4.11) between two digits. In all the three tasks, PLVAE generates pairs of numbers consistent with the evidence, and it also shows a variety of combinations by relying on the probabilistic engine of ProbLog.
As we can see in Table 4.2, the generative accuracy of PLVAE decreases in those tasks where the digits order is relevant (i.e. *subtraction* and *power*). The reason is that the model may learn a mapping between symbol and meaning that is logically correct, but different from the desired one. For example, during the training on the base task, PLVAE may switch the pairs $(2, 1)$ and $(1, 2)$, since they both sum up to 3. This would prevent PLVAE from generalizing to tasks involving non-commutative operations.
A possible solution to solve this issue is to introduce an additional supervision on very few digits to guide the model toward the desired symbols interpretation. As described in Section 2.2, by virtue of ProbLog inference we can easily retrieve the predicted label of the digits in an image by relying on the query over the digits values, namely `query(digits(X1,X2))` (*Query B*). Thus, to effectively introduce the additional supervision in our training procedure, we simply add a regularizer term to the objective function $\mathcal{L}(\theta, \phi)$ defined in (4.1),

$$\mathcal{L}_{\mathcal{SUP}}(\theta, \phi) := \mathcal{L}(\theta, \phi) + \mathcal{L}_{digits}(\theta, \phi) \tag{4.1}$$

where

$$\mathcal{L}_{digits}(\theta, \phi) = \mathbb{E}_{z_{sym} \sim q_{\phi}(z_{sym}|x)}[log(p_{\theta}(y_{digits}|z_{sym}))]. \tag{4.2}$$

In equation (4.2), y_{digits} refers to the digits label (e.g. for we have $y_{digits} = [0, 1]$).

In Table 4.2 we report the generative accuracy of PLVAE trained on a dataset with 10 out of 28800 fully supervised images.

	Model	$m_{GEN}(\uparrow)$
Task Generalization *(multiplication)*	PLVAE	0.7280 ± 0.1567
	PLVAE *(with supervision)*	0.7314 ± 0.1952
	CCVAE	*Not possible*
Task Generalization *(subtraction)*	PLVAE	0.3797 ± 0.1920
	PLVAE *(with supervision)*	0.6201 ± 0.2755
	CCVAE	*Not possible*
Task Generalization *(power)*	PLVAE	0.4908 ± 0.2261
	PLVAE *(with supervision)*	0.6546 ± 0.2675
	CCVAE	*Not possible*

Table 4.2: Generative accuracies for task generalization experiments.

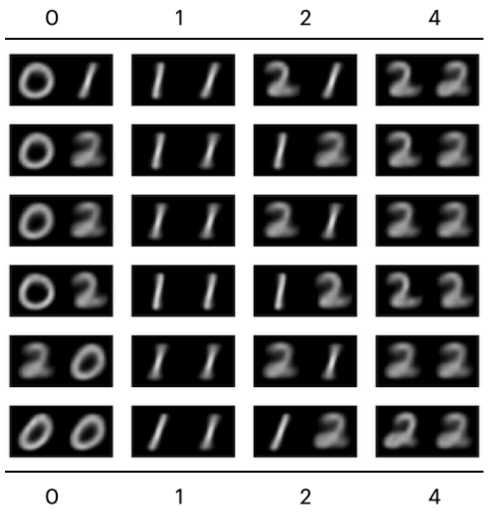


Figure 4.9: Novel task: multiplication between two numbers.
Each column refers to a different evidence, from `add(img,0)` to `add(img,4)`; the latent vector $\mathbf{z}$ is held constant along each row.

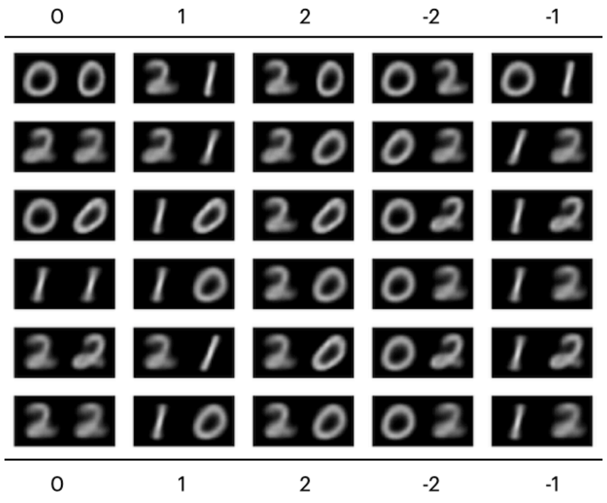


Figure 4.10: Novel task: subtraction between the first and the second digits.
Each column refers to a different evidence, from `add(img,0)` to `add(img,-1)`; the latent vector **z** is held constant along each row.

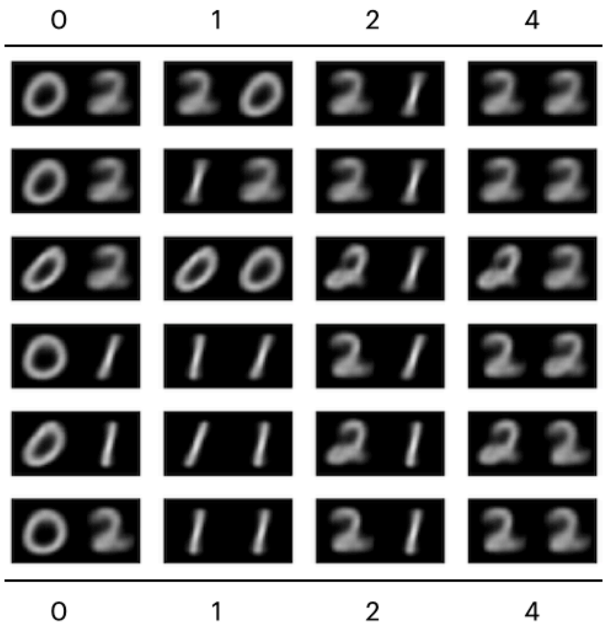


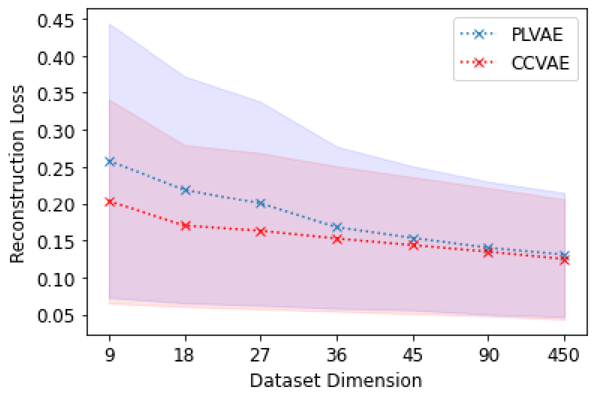
Figure 4.11: Novel task: first digit to the power of the second one.
Each column refers to a different evidence, from `add(img,0)` to `add(img,4)`; the latent vector **z** is held constant along each row.

4.6 Experiment 4: Data Efficiency

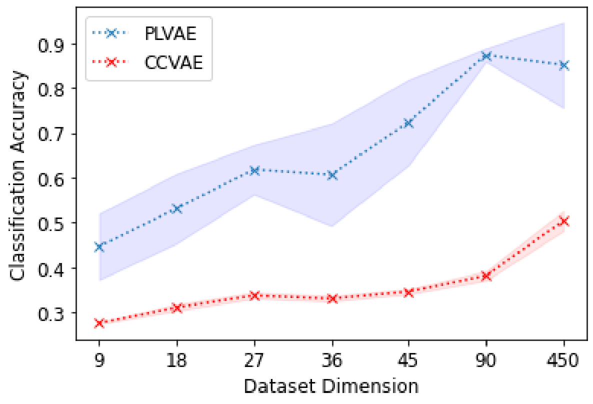
To verify whether the use of a logic-based prior helps the learning in contexts characterized by data scarcity, we trained both PLVAE and the baseline with training sets of increasing size. In particular, we defined 7 levels of data availability for the base task:

- 50 images per digits pair (i.e. 450 examples);
- 10 images per digits pair (i.e. 90 examples);
- 5 images per digits pair (i.e. 45 examples);
- 4 images per digits pair (i.e. 36 examples);
- 3 images per digits pair (i.e. 27 examples);
- 2 images per digits pair (i.e. 18 examples);
- 1 image per digits pair (i.e. 9 examples).

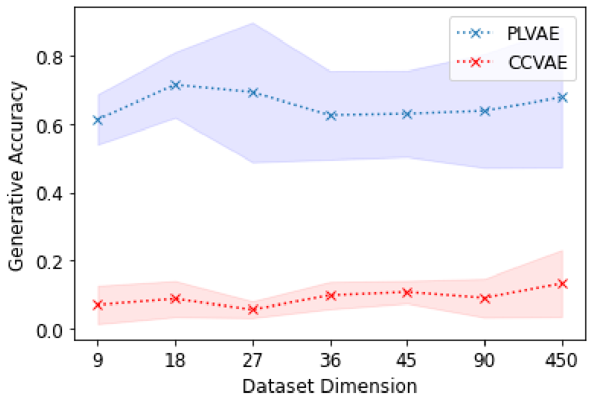
In Figure 4.12 we compare PLVAE with the baseline in terms of reconstructive, predictive and generative ability in the 7 levels of data availability. Whereas the difference in the reconstruction loss of the two models is not significant, PLVAE outperforms the baseline in terms of predictive and generative ability. In particular, PLVAE achieves a generative accuracy greater than 60% even when trained on the smallest dataset.
In Figure 4.13 we report some examples of conditional generation for PLVAE and CCVAE trained with 90 examples. As we can see, whereas the baseline cannot generate sample-like images, our model is able to generate pairs of digits consistent with the condition. Moreover, the generated images shows a variety of digits combinations thanks to the probabilistic engine of ProbLog. The reason behind this disparity is that the logic-based prior helps the neural model in properly structuring the latent representation, so that one part can easily focus on recognizing individual digits and the other on capturing the remaining information in the scene. Conversely, the baseline needs to learn how to correctly model very different pairs that sum up to the same value.



(a) Reconstruction loss.

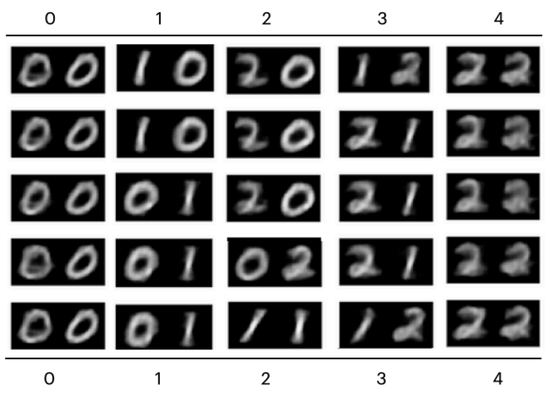


(b) Classification accuracy.

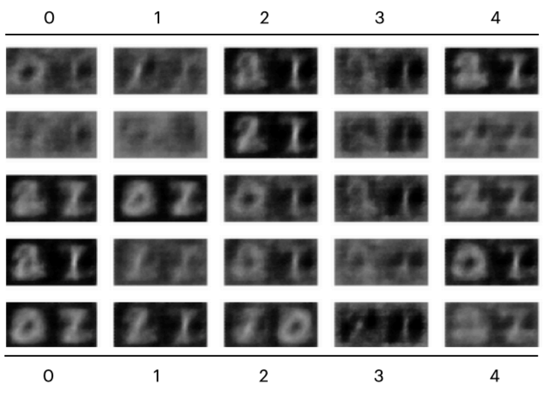


(c) Generative accuracy.

Figure 4.12: The graphs compare the performance of PLVAE and CCVAE on the test set for different training set sizes.



(a) PLVAE.



(b) CCVAE.

Figure 4.13: Conditional generation for PLVAE and CCVAE trained with 10 images per pair of digits.

Chapter 5

Conclusions

In this thesis, we propose the *probabilistic logic* VAE (PLVAE), a neuro-symbolic deep generative model that combines the representational power of VAEs with the reasoning ability of probabilistic-logic programming. The strength of PLVAE resides in its probabilistic-logic prior which provides an interpretable structure to the latent space that can be easily changed in order to apply the model to different scenarios. Thus, PLVAE fully exploits the expressiveness of both neural and symbolic method to solve previously unseen generative tasks in an end-to-end trainable framework.

As described in Chapter 4, we validated our approach by training PLVAE on the two digits addition task. Then, we used the same model to generalize to new generative tasks involving different arithmetic operations. In all the experiments, PLVAE was able to generate pairs of numbers that are logically consistent with the condition and that show a variety of combinations thanks to the probabilistic engine of ProbLog.
Moreover, we defined different levels of data availability to verify whether the use of a logic-based prior helps the learning in contexts characterized by data scarcity. The results show that our model outperformed the baseline in all the datasets. PLVAE was able to generate pairs of digits consistent with the condition even if trained with very few examples.

One of the current limitations of PLVAE is related to model scalability. As described in Section 2.2, PLVAE relies on ProbLog inference for two different tasks that involve marginalization and sampling. Whereas the marginalization can be easily made differentiable by relying on *aProbLog*, there are currently no methods to perform differentiable sampling in ProbLog.

In this thesis, we solve the problem by replacing sampling with multiple marginalization operations. However, the proposed solution may become prohibitively expensive as the size of the program grows. Thus, one of the future research directions is the development of new methods to effectively exploit knowledge compilation principles to define differentiable sampling for ProbLog inference.

Another issue of PLVAE is that the model may learn a mapping between symbol and meaning that is logically correct, but different from the desired one. For example, in the two digits addition problem, PLVAE may switch the pairs $(2, 1)$ and $(1, 2)$, since they both sum up to 3. This would prevent from generalizing to tasks where the digits order is relevant (e.g. non-commutative operations). In the two digits addition example, we solve this issue by introducing an additional supervision on very few digits to guide the model toward the desired symbols interpretation. Although we have shown preliminary result of the effectiveness of this solution, future work could look into this more extensively.

Finally, as other models, PLVAE suffers from instability due to the high number of model hyper-parameters. Thus, another research direction would be the exploration of different definitions of the reconstruction loss, to achieve both model stability and higher image quality.

In the future, we would also like to extend PLVAE to more complex settings, using datasets like *CLEVR* [44] that allow for more composite tasks.

Bibliography

[1] Ian Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: `https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf`.

[2] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: `1312.6114 [stat.ML]`.

[3] Yoshua Bengio et al. “A Neural Probabilistic Language Model”. In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[4] Benigno Uria et al. *Neural Autoregressive Distribution Estimation*. 2016. arXiv: `1605.02226 [cs.LG]`.

[5] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. *Pixel Recurrent Neural Networks*. 2016. arXiv: `1601.06759 [cs.CV]`.

[6] Laurent Dinh, David Krueger, and Yoshua Bengio. *NICE: Non-linear Independent Components Estimation*. 2015. arXiv: `1410.8516 [cs.LG]`.

[7] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. *Density estimation using Real NVP*. 2017. arXiv: `1605.08803 [cs.LG]`.

[8] Diederik P. Kingma and Prafulla Dhariwal. *Glow: Generative Flow with Invertible 1x1 Convolutions*. 2018. arXiv: `1807.03039 [stat.ML]`.

[9] Geoffrey Hinton. “Boltzmann Machines”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 132–136. ISBN: 978-0-387-30164-8. DOI: `10.1007/978-0-387-30164-8_83`. URL: `https://doi.org/10.1007/978-0-387-30164-8_83`.

[10] Lars Maaløe et al. *Auxiliary Deep Generative Models*. 2016. arXiv: `1602.05473 [stat.ML]`.

[11] Casper Kaae Sønderby et al. *Ladder Variational Autoencoders*. 2016. arXiv: `1602.02282 [stat.ML]`.

[12] Lars Maaløe et al. *BIVA: A Very Deep Hierarchy of Latent Variables for Generative Modeling*. 2019. arXiv: `1902.02102 [stat.ML]`.

[13] Arash Vahdat and Jan Kautz. *NVAE: A Deep Hierarchical Variational Autoencoder*. 2021. arXiv: `2007.03898 [stat.ML]`.

[14] Alexej Klushyn et al. *Learning Hierarchical Priors in VAEs*. 2019. arXiv: `1905.04982 [stat.ML]`.

[15] Jakub M. Tomczak and Max Welling. *VAE with a VampPrior*. 2018. arXiv: `1705.07120 [cs.LG]`.

[16] Matthias Bauer and Andriy Mnih. *Resampled Priors for Variational Autoencoders*. 2019. arXiv: `1810.11428 [stat.ML]`.

[17] Xi Chen et al. *Variational Lossy Autoencoder*. 2017. arXiv: `1611.02731 [cs.LG]`.

[18] Tom Joy et al. *Capturing Label Characteristics in VAEs*. 2021. arXiv: `2006.10102 [cs.LG]`.

[19] Diederik P. Kingma et al. *Semi-Supervised Learning with Deep Generative Models*. 2014. arXiv: `1406.5298 [cs.LG]`.

[20] Luc De Raedt et al. 2016.

[21] Luc De Raedt and Angelika Kimmig. "Probabilistic (Logic) Programming Concepts". In: *Mach. Learn.* 100.1 (July 2015), pp. 5–47. ISSN: 0885-6125. DOI: `10.1007/s10994-015-5494-z`. URL: `https://doi.org/10.1007/s10994-015-5494-z`.

[22] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. "ProbLog: A probabilistic Prolog and its application in link discovery". eng. In: Veloso, M. IJCAI-INT JOINT CONF ARTIF INTELL, 2007, pp. 2462–2467.

[23] Jindong Jiang and Sungjin Ahn. *Generative Neurosymbolic Machines*. 2021. arXiv: `2010.12152 [cs.LG]`.

[24] Reuben Feinman and Brenden M. Lake. *Generating new concepts with hybrid neuro-symbolic models*. 2020. arXiv: `2003.08978 [cs.LG]`.

[25] M.I. Jordan et al. “An Introduction to Variational Methods for Graphical Models”. In: *Machine Learning* 37 (Nov. 1999), pp. 183–233. URL: `https://doi.org/10.1023/A:1007665907178`.

[26] S. Kullback and R. A. Leibler. “On Information and Sufficiency”. In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: `10.1214/aoms/1177729694`. URL: `https://doi.org/10.1214/aoms/1177729694`.

[27] Danilo Jimenez Rezende and Shakir Mohamed. *Variational Inference with Normalizing Flows*. 2016. arXiv: `1505.05770 [stat.ML]`.

[28] Diederik P. Kingma, Tim Salimans, and Max Welling. *Variational Dropout and the Local Reparameterization Trick*. 2015. arXiv: `1506.02557 [stat.ML]`.

[29] Eric Jang, Shixiang Gu, and Ben Poole. *Categorical Reparameterization with Gumbel-Softmax*. 2017. arXiv: `1611.01144 [stat.ML]`.

[30] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. *The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables*. 2017. arXiv: `1611.00712 [cs.LG]`.

[31] Daan Fierens et al. “Inference and learning in probabilistic logic programs using weighted Boolean formulas”. In: *Theory and Practice of Logic Programming* 15.3 (Apr. 2014), pp. 358–401. ISSN: 1475-3081. DOI: `10.1017/s1471068414000076`. URL: `http://dx.doi.org/10.1017/S1471068414000076`.

[32] A. Darwiche and P. Marquis. “A Knowledge Compilation Map”. In: *Journal of Artificial Intelligence Research* 17 (Sept. 2002), pp. 229–264. ISSN: 1076-9757. DOI: `10.1613/jair.989`. URL: `http://dx.doi.org/10.1613/jair.989`.

[33] Adnan Darwiche. “SDD: A New Canonical Representation of Propositional Knowledge Bases”. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*. IJCAI’11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 819–826. ISBN: 9781577355144.

[34] Adnan Darwiche. “On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision”. In: *Journal of Applied Non-Classical Logics* 11.1-2 (2001), pp. 11–34. DOI: `10.3166/jancl.11.11-34`. eprint: `https://doi.org/10.3166/jancl.11.11-34`. URL: `https://doi.org/10.3166/jancl.11.11-34`.

[35] Adnan Darwiche. *A Differential Approach to Inference in Bayesian Networks*. 2013. arXiv: 1301.3847 [cs.AI].

[36] Robin Manhaeve et al. *DeepProbLog: Neural Probabilistic Logic Programming*. 2018. arXiv: 1805.10872 [cs.AI].

[37] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[38] Michael Frazier and Leonard Pitt. "Learning From Entailment: An Application to Propositional Horn Sentences". In: *Machine Learning Proceedings 1993*. San Francisco (CA): Morgan Kaufmann, 1993, pp. 120–127. ISBN: 978-1-55860-307-3. DOI: https://doi.org/10.1016/B978-1-55860-307-3.50022-8. URL: https://www.sciencedirect.com/science/article/pii/B9781558603073500228.

[39] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. "An Algebraic Prolog for Reasoning about Possible Worlds". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI'11. San Francisco, California: AAAI Press, 2011, pp. 209–214.

[40] Jason Eisner. "Parameter Estimation for Probabilistic Finite-State Transducers". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 1–8. DOI: 10.3115/1073083.1073085. URL: https://aclanthology.org/P02-1001.

[41] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. arXiv: 1312.6114 [stat.ML].

[42] Dimitar Sht. Shterionov et al. *DNF Sampling for ProbLog Inference*. 2010. arXiv: 1009.3798 [cs.LO].

[43] J. L. W. V. Jensen. *Sur les fonctions convexes et les inégualités entre les valeurs Moyennes*. Nov. 1906. DOI: 10.1007/bf02418571. URL: https://doi.org/10.1007/bf02418571.

[44] Justin Johnson et al. *CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning*. 2016. arXiv: 1612.06890 [cs.CV].

[45] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`.

Appendix A

A.1 PLVAE Architecture

In Table A.1 and A.2 we report the architecture of PLVAE. In all the experiments we trained the model with with Adam [45] with a learning rate of 1×10^{-3}.

PLVAE Encoder
Input $28 \times 56 \times 1$
Conv layer $4 \times 4 \times 32$ & ReLU
Conv layer $4 \times 4 \times 64$ & ReLU
Conv layer $4 \times 4 \times 128$ & ReLU
Linear layer 1792×14

Table A.1

PLVAE Decoder
Linear layer 5×10 & ReLU
Linear layer 5×6
ProbLog (IN dim: 6, OUT dim: 9)
Linear layer 1792×7
Conv layer $5 \times 4 \times 128$ & ReLU
Conv layer $4 \times 4 \times 64$ & ReLU
Conv layer $4 \times 4 \times 32$ & Sigmoid

Table A.2

A.2 Baseline: Modified CCVAE

In the original paper [18], there was a direct supervision on each single element of the latent space. To preserve the same type of supervision in our two digits addition task, where the supervision is on the sum and not directly on the single digits, we slightly modify the encoder and decoder mapping functions of CCVAE. By doing so, we ensure the correctness of the approach without changing the graphical model.
The original encoder function learns from the input both the mean μ and the variance σ of the latent space distribution, while the decoder gets in input the latent representation $\mathbf{z} = \{\mathbf{z}_c, \mathbf{z}_{\setminus c}\}$ (please refer to the orginal paper for more details [18]).

In our modified version, the encoder only learns the variance, while the mean is set to be equal to the image label $\mu = y$, and the decoder gets in input the label directly $\mathbf{z}^* := \{y, \mathbf{z}_{\setminus c}\}$. The encoder and decoder architecture are reported in Table A.3 and A.4.

CCVAE Encoder
Input $28 \times 56 \times 1$
Conv layer $4 \times 4 \times 32$ & ReLU
Conv layer $4 \times 4 \times 64$ & ReLU
Conv layer $4 \times 4 \times 128$ & ReLU
Linear layer 1792×16

Table A.3

CCVAE Decoder
Linear layer 1792×8
Conv layer $5 \times 4 \times 128$ & ReLU
Conv layer $4 \times 4 \times 64$ & ReLU
Conv layer $4 \times 4 \times 32$ & Sigmoid

Table A.4

A.3 MNIST Classifier

In Table A.5 we report the architecture of the MNIST classifier used to measure the generative ability of PLVAE and the baseline. We trained the classifier on $60,000$ MNIST images [37] for 15 epochs with SGD with a learning rate of 1×10^{-2} and a momentum of 0.5, achieving 0.97 accuracy on the test set.

MNIST classifier
Input $28 \times 28 \times 1$ channel image
784×128 Linear layer & ReLU
128×64 Linear layer & ReLU
64×10 Linear layer & LogSoftmax

Table A.5