

# Learning Relational Representations with Auto-encoding Logic Programs

Sebastijan Dumančić<sup>1\*</sup>, Tias Guns<sup>2</sup>, Wannes Meert<sup>1</sup> and Hendrik Blockeel<sup>1</sup>

<sup>1</sup>KU Leuven, Belgium

<sup>2</sup>VUB, Belgium

{sebastijan.dumancic, wannes.meert, hendrik.blockeel}@cs.kuleuven.be, tias.guns@vub.be

## Abstract

Deep learning methods capable of handling relational data have proliferated over the last years. In contrast to traditional relational learning methods that leverage first-order logic for representing such data, these deep learning methods aim at re-representing symbolic relational data in Euclidean spaces. They offer better scalability, but can only numerically approximate relational structures and are less flexible in terms of reasoning tasks supported. This paper introduces a novel framework for relational representation learning that combines the best of both worlds. This framework, inspired by the auto-encoding principle, uses first-order logic as a data representation language, and the mapping between the original and latent representation is done by means of logic programs instead of neural networks. We show how learning can be cast as a constraint optimisation problem for which existing solvers can be used. The use of logic as a representation language makes the proposed framework more accurate (as the representation is exact, rather than approximate), more flexible, and more interpretable than deep learning methods. We experimentally show that these latent representations are indeed beneficial in relational learning tasks.

## 1 Introduction

Learning models from ubiquitous relational data – with examples in large knowledge graphs and protein interaction networks – has typically been tackled by logic-based approaches to Artificial Intelligence such as *Statistical relational learning* (SRL) [Getoor and Taskar, 2007]. These models combine the knowledge representation capabilities of first-order logic with probability theory, and hence express both complex relational structures and uncertainty in data. Their unique feature are the *reasoning capabilities* inherited from first-order logic: they are capable of performing complex chains of reasoning and answering questions about any part of a domain (instead of one pre-defined concept). Most machine

learning methods lack this component. Unfortunately, the benefits of SRL are countered with a high computational cost of learning which conforms to the *learning by search paradigm* [Mitchell, 1997].

An effective way to reduce the complexity of learning is to modify the data such that it *simplifies the learning task*; this is achieved with a series of data transformation steps that define a new feature space (so called *latent representation*) that makes regularities in data more explicit. This is known as deep representation learning (DL) [Goodfellow *et al.*, 2016]. Recent years have yielded various adaptations of standard DL models towards relational data, namely *Knowledge graph embeddings* [Nickel *et al.*, 2016] and *Graph neural networks* [Kipf and Welling, 2017; Hamilton *et al.*, 2017]. These approaches aim to re-represent relational data in vectorised Euclidean spaces, on top of which feature-based machine learning methods can be used. Though this offers good learning capabilities, it sacrifices the flexibility of reasoning [Trouillon *et al.*, 2019] and can only *approximate* relational data, but not capture it in its entirety.

This work proposes a framework that unites the benefits of both the SRL and the DL research directions. We start with the question:

*Is it possible to learn latent representations of relational data that improve the performance of SRL models, such that the reasoning capabilities are preserved?*

We revisit the basic principles of relational representation learning and introduce a novel framework to learn latent representations based on *symbolic*, rather than gradient-based computation. The proposed framework implements the *auto-encoder principle* [Goodfellow *et al.*, 2016; Hinton and Salakhutdinov, 2006] – one of the most versatile deep learning components – but uses *logic programs* as a computation engine instead of (deep) neural networks. For this reason, we name our approach *Auto-encoding logic programs* (Alps).

Retaining the logic as a representation language offers several benefits, adding to its expressivity and the reasoning capabilities. Logic is easy to understand and interpret (while DL is *black-box*), which is important for trust in AI systems. Furthermore, SRL methods allow for incorporation of *expert knowledge* and thus can easily build on previously gathered knowledge. Finally, SRL systems are capable of learning

\*Contact Author

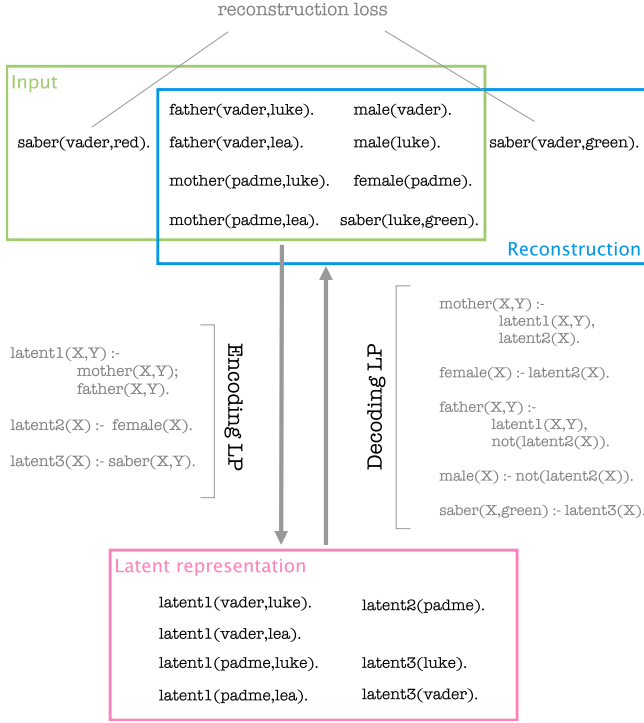


Figure 1: An auto-encoding logic program maps the input data, given in a form of a set of facts, to its latent representation through an *encoding logic program*. A *decoding logic program* maps the latent representation of data back to the original data space. The facts missing from the reconstruction (e.g., `saber(vader,red)`) and the wrongly reconstructed facts (e.g., `saber(vader,green)`) constitute the *reconstruction loss*.

from a few examples only, which is in sharp contrast to DL methods that are typically *data-hungry*.

Alongside the formalism of Alps, we contribute a generic procedure to learn Alps from data. The procedure translates the learning task to a constraint optimisation problem for which existing efficient solvers can be used. In contrast to neural approaches, where the user has to provide the architecture beforehand (e.g., a number of neurons per layer) and tune many hyper-parameters, the architecture of Alps is the *outcome* of learning. Notably, we show that the learned latent representations help with learning SRL models afterwards: SRL models learned on the latent representation outperform the models learned on the original data representation.

## 2 Auto-encoding Logic Programs

Auto-encoders learn new representations through the *reconstruction principle*: the goal is to learn an *encoder*, mapping the input data to its latent representation, and a *decoder*, mapping the latent representation back to the original space, so that the input data can be faithfully reconstructed. For a latent representation to be useful, it is important to prevent it from learning an identity mapping – often done by limiting the dimensionality and/or enforcing sparsity.

In neural auto-encoders, data is represented with vectors and mapping functions are matrices. Our goal is, intuitively, to lift the framework of auto-encoders to use *first-order logic*

as a data representation language, and logic programs as mapping functions of encoder and decoder (Figure 1). In the following paragraphs we describe the basic components of Alps.

**Data.** To handle arbitrary relational data, Alps represent data as a set of logical statements, such as `father(vader,luke)` (Figure 1, Input). These statements consist of *constants* representing the entities in a domain (e.g., `vader`, `luke`) and *predicates* indicating the relationships between entities (e.g., `father`). A *ground atom* is a predicate symbol applied to constants (e.g., `father(vader,luke)`); if an atom evaluates to *true*, it represents a *fact*. Given a set of predicates  $\mathcal{P}$  and a set of constants  $\mathcal{C}$  (briefly, a vocabulary  $(\mathcal{P}, \mathcal{C})$ ), the Herbrand base  $\mathcal{HB}(\mathcal{P}, \mathcal{C})$  is the set of all atoms that can be constructed using  $\mathcal{P}$  and  $\mathcal{C}$ . A knowledge base is a subset of the Herbrand base; it contains all the atoms that evaluate to *true*.

**Mapping functions.** The mapping functions of both encoder and decoder are realised as *logic programs*. A logic program is a set of *clauses* – logical formulas of the form  $h :- b_1, \dots, b_n$ , where  $h$  is called the *head literal* and  $b_i$  are *body literals* (comma denotes conjunction). A literal is an atom or its negation. Literals can contain variables as arguments; these are by definition universally quantified. Given a vocabulary  $(\mathcal{P}, \mathcal{C})$ , we call a literal a  $(\mathcal{P}, \mathcal{C})$ -literal if its predicate is in  $\mathcal{P}$  and its argument are constants in  $\mathcal{C}$  or variables. Clauses are read as logical implications; e.g., the clause `mother(X,Y)`  $:-$  `parent(X,Y), female(X)` states that for all  $X$  and  $Y$ ,  $X$  is a mother of  $Y$  if  $X$  is a parent of  $Y$  and  $X$  is female.

**Encoding / decoding logic programs.** Given an input vocabulary  $(\mathcal{P}, \mathcal{C})$ , an *encoding logic program*  $\mathcal{E}$  (Fig. 1 middle left) is a set of clauses with  $(\mathcal{P}, \mathcal{C})$ -literals in the body and a positive  $(\mathcal{L}, \mathcal{C})$ -literal in the head, where  $\mathcal{L}$  is a set of predicates that is disjoint with  $\mathcal{P}$  and is extended by the learner as needed.

$\mathcal{E}$  takes an input a knowledge base  $\mathcal{KB} \subseteq \mathcal{HB}(\mathcal{P}, \mathcal{C})$  and produces as output a latent representation  $\mathcal{KB}' \subseteq \mathcal{HB}(\mathcal{L}, \mathcal{C})$ , more specifically the set of all facts that are implied by  $\mathcal{E}$  and  $\mathcal{KB}$ .

A *decoding logic program*  $\mathcal{D}$  similarly maps a subset of  $\mathcal{HB}(\mathcal{L}, \mathcal{C})$  back to a subset of  $\mathcal{HB}(\mathcal{P}, \mathcal{C})$ . Its clauses are termed *decoder clauses*; they contain  $(\mathcal{L}, \mathcal{C})$  literals in the body and a positive  $(\mathcal{P}, \mathcal{C})$ -literal in the head.

**Alps.** Given encoding and decoding logic programs  $\mathcal{E}$  and  $\mathcal{D}$ , their composition  $\mathcal{D} \circ \mathcal{E}$  is called an *auto-encoding logic program* (Alp). An Alp is lossless if for any  $\mathcal{KB}$ ,  $\mathcal{D}(\mathcal{E}(\mathcal{KB})) = \mathcal{KB}$ . In this paper, we measure the quality of Alps using the following loss function:

**Definition 1 Knowledge base reconstruction loss.** The knowledge base reconstruction loss (the disagreement between the input and the reconstruction),  $\text{loss}(\mathcal{E}, \mathcal{D}, \mathcal{KB})$ , is defined as

$$\text{loss}(\mathcal{E}, \mathcal{D}, \mathcal{KB}) = |\mathcal{D}(\mathcal{E}(\mathcal{KB})) \Delta \mathcal{KB}| \quad (1)$$

where  $\Delta$  is the symmetric difference between two sets.

## 3 Learning as constraint optimisation

With the main components of Alps defined in the previous section, we define the learning task as follows:

**Definition 2** Given a knowledge base  $\mathcal{KB}$  and constraints on the latent representation, **find**  $\mathcal{E}$  and  $\mathcal{D}$  that minimise  $\text{loss}(\mathcal{E}, \mathcal{D}, \mathcal{KB})$  and  $\mathcal{E}(\mathcal{KB})$  fulfils the constraints.

The constraints on the latent representation prevent it from learning an identity mapping. For example, enforcing sparsity by requiring that the  $\mathcal{E}(\mathcal{KB})$  has at most  $N$  facts. We formally define these constraints later.

Intuitively, learning Alps corresponds to *searching for a well-performing combination of encoder and decoder clauses*. That is, out of a set of possible encoder and decoder clauses, *select* a subset that minimises the reconstruction loss. To find a well-performing subset of clauses, we introduce a learning method inspired by the *enumerative* and *constraint solving* techniques from program induction [Gulwani *et al.*, 2017] (illustrated in Figure 2). Given a  $\mathcal{KB}$  and predicates  $\mathcal{P}$ , we first enumerate possible encoder clauses. These clauses define a set of candidate *latent predicates*  $\mathcal{L}$  which are subsequently used to generate candidate decoder clauses. The obtained sets, which define the space of candidate clauses to choose from, are then pruned and used to formulate the learning task as a generic *constraint optimisation problem* (COP) [Rossi *et al.*, 2006]. Such a COP formulation allows us to tackle problems with an extremely large search space and leverage existing efficient solvers. The COP is solved using the Oscar solver [Oscar Team, 2012]. The resulting solution is a subset of the candidate encoder and decoder clauses that constitute an Alp.

A COP consists of three components: **decision variables** whose values have to be assigned, **constraints** on decision variables, and an **objective function** over the decision variables that expresses the quality of the assignment. A *solution* consists of a value assignment to the decision variables such that all constraints are satisfied. In the following sections we describe each of these components for learning Alps.

### 3.1 Decision variables: candidate clauses

The COP will have one Boolean decision variable  $ec_i$  for each generated candidate encoder clause, and a Boolean decision variable  $dc_i$  for each generated candidate decoder clause, indicating whether a clause is selected (having the value 1) or not (having value 0).

To generate the candidate encoder clauses, we start from the predicates in the input data, and generate all possible bodies (conjunctions or disjunctions of input predicates with logic variables as entities) up to a given maximum length  $l$ . Furthermore, we enforce that the predicates share at least one logic variable, e.g.  $p_1(X, Y), p_2(Y, Z)$  is allowed while  $p_1(X, Y), p_2(Z, W)$  is not. For each possible body, we then define a new *latent predicate* that will form the head of the clause. This requires deciding which variables from the body to use in the head. We generate all heads that use a subset of variables, with the maximal size of the subset equal to the maximum number of arguments of predicates  $\mathcal{P}$ . Candidate decoder clauses are generated in the same way, but starting from the predicates  $\mathcal{L}$ .

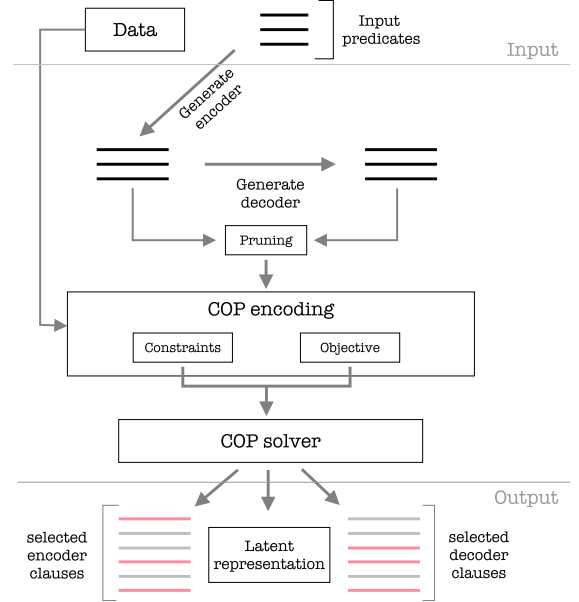


Figure 2: **Learning Alps.** Given the data and a set of predicates as an input, we first enumerate possible encoder clauses and subsequently the decoder clauses. These are used to generate the COP encoding, including the constraints and the objective, which is pruned and passed to the COP solver. The solver returns the selected encoder/decoder clauses and the latent representation.

### 3.2 Constraints

#### Capacity constraint

The primary role of constraints in Alps is to impose a bottleneck on the capacity of the latent representation; this is the key ingredient in preventing the auto-encoder from learning the *identity mapping* as  $\mathcal{E}$  and  $\mathcal{D}$ . This is often done by enforcing compression in the latent representation, sparsity or both.

The straightforward way of imposing compression in Alps is to limit the number of facts in the latent representation. Preliminary experiments showed this to be a very restrictive setting. In Alps we impose the bottleneck by limiting the *average number of facts per latent predicate* through the following constraint

$$\frac{\sum_{i=1}^N w_i ec_i}{\sum_{i=1}^N ec_i} \leq \gamma G$$

where  $ec_i$  are decision variables corresponding to the encoder clauses,  $w_i$  is the number of latent facts the encoder clause  $ec_i$  entails,  $G$  is the average number of facts per predicate in the original data representation and  $\gamma$  is the *compression parameter* specified by the user. For example, in Figure 1,  $G = 9/5$  and  $w = 4$  for  $\text{latent1}(X, Y) :- \text{mother}(X, Y); \text{father}(X, Y)$ .

#### Semantic constraints

The secondary role of constraints is to impose additional structure to the search space, which can substantially speed up the search. The following set of constraints reduces the

search space by removing undesirable and redundant solutions<sup>1</sup>. These constraints are automatically generated and do not require input from the user.

**Connecting encoder and decoder.** A large part of the search space can be cut out by noticing that the encoder clauses deterministically depend on the decoder clauses. For instance, if a decoder clause  $mother(X, Y) :- latent1(X, Y), latent2(X)$  is selected in the solution, then the encoder clauses defining the latent predicates  $latent1$  and  $latent2$  have to be selected as well. Consequently, encoder clauses are implied by decoded clauses and search only has to happen over candidate decoder clauses. The implication is modelled with a constraint ensuring that *the final solution must contain an encoder clause defining a predicate  $\perp$  if the solution contains at least one of the decoder clauses that uses  $\perp$  in the body*.

**Generality.** Given the limited capacity of the latent representation, it is desirable to prevent the solver from ever exploring regions where clauses are too similar and thus yielding a marginal gain. One way to establish the similarity of clauses is to analyse the ground atoms the clauses cover: a clause  $c_1$  is said to be more *general* than a clause  $c_2$  if all examples entailed by  $c_2$  are also entailed by  $c_1$ . As  $c_2$  cannot bring new information if  $c_1$  is already a part of the solution, we introduce constraints ensuring that *if a clause  $c_1$  is more general than a clause  $c_2$ , at most one of them can be selected*.

**Reconstruct one of each input predicate.** If  $\mathcal{KB}$  contains a predicate with a substantially larger number of facts than the other predicates in  $\mathcal{KB}$ , a trivial but undesirable solution is one that focuses on reconstructing the predicate and its facts while ignoring the predicates with a smaller number of facts. To prevent this, we impose the constraints ensuring that *among all decoder clauses with the same input predicate in the head, at least one has to be a part of the solution*. This of course does not mean all facts of each input predicate will be reconstructed. We did notice that it allows the solver to find a good solution substantially faster.

### 3.3 Objective function: the reconstruction loss

We wish to formulate the objective over all *missing* (in  $\mathcal{KB}$  but not being reconstructed) and *false reconstructions* (produced by the decoder, but not in  $\mathcal{KB}$ ). To do so, we first obtain a union of *latent facts* generated by each of the candidate encoder clauses; these are a subset of  $\mathcal{HB}(\mathcal{L}, \mathcal{C})$ . These latent facts allow us to obtain a union of all ground atoms generated by the candidate decoder clauses; these form a *reconstruction* and are a subset of  $\mathcal{HB}(\mathcal{P}, \mathcal{C})$ . Additionally, for each ground atom in the reconstruction, we remember which candidate decoder clause reconstructed it.

We hence use the above correspondence between the candidate decoder clauses and the reconstructions to create an auxiliary Boolean decision variable  $rf_i$  for each possible ground atom in  $\mathcal{HB}(\mathcal{P}, \mathcal{C})$  that can be reconstructed. Whether it is reconstructed or not depends on the decoder clauses that are in the solution.

For example, assume that  $mother(padme, lea)$  can be reconstructed with either of the following decoder clauses:

$mother(X, Y) :- latent1(X, Y), latent2(X).$   
 $mother(X, Y) :- latent3(X, Y).$

Let the two decoder clauses correspond to the decision variables  $dc_1$  and  $dc_2$ . We introduce  $rf_i$  to represent the reconstruction of fact  $mother(padme, lea)$  and add a constraint

$$rf_i \Leftrightarrow dc_1 \vee dc_2$$

Associating such boolean variable  $rf_e$  with every  $e \in \mathcal{HB}(\mathcal{P}, \mathcal{C})$ , we can formulate the objective as

$$\text{minimize } \underbrace{\sum_{i \in \mathcal{KB}} \text{not}(rf_i)}_{\text{missing reconstruction}} + \underbrace{\sum_{j \in \mathcal{HB}(\mathcal{P}, \mathcal{C}) \setminus \mathcal{KB}} rf_j}_{\text{false reconstruction}}. \quad (2)$$

### 3.4 Search

Given the combinatorial nature of Alps, finding the optimal solution is impossible in all but the smallest problem instances. Therefore, we resort to the more scalable technique of *large neighbourhood search* (LNS) [Ahuja *et al.*, 2002]. LNS is an iterative search procedure that, in each iteration, performs the search over a subset of decision variables. This subset of variables is called the *neighbourhood* and it is constructed around the best solution found in the previous iterations.

A key design choice in LNS is the construction of the neighbourhood. The key insight of our strategy is that the solution is necessarily sparse – only a tiny proportion of candidate decoder clauses will constitute the solution at any time. Therefore, it is important to preserve at least some of the selected decoder clauses between the iterations. Let a variable be *active* if it is part of the best solution found so far, and *inactive* otherwise. We construct the neighbourhood by remembering the value assignment of  $\alpha$  % active variables (corresponding to decoder clauses), and  $\beta$  % inactive variables corresponding to encoder clauses. For the individual search runs, we use *last conflict search* [Gay *et al.*, 2015] and the *max degree* ordering of decision variables.

### 3.5 Pruning the candidates

As the candidate clauses are generated naively, many candidates will be uninformative and introduce mostly false reconstructions. It is therefore important to help the search by pruning the set of candidates in an insightful and non-trivial way. We introduce the following three strategies that leverage the specific properties of the problem at hand.

**Naming variants.** Two encoder clauses are *naming variants* if and only if they reconstructed the same set of ground atoms, apart from the name of the predicate of these ground atoms. As such clauses contain the same information w.r.t. the constants they contain, we detect all naming variants and keep only one instance as a candidate.

**Signature variants.** Two decoder clauses are *signature variants* if and only if they reconstructed the same set of ground atoms and their bodies contain the same predicates.

<sup>1</sup>Exact constraint formulations are in the supplementary material available on this link: <http://bit.ly/AlpsSupplement>

As signature variants are redundant w.r.t. the optimisation problem, we keep only one of the clauses detected to be signature variants and remove the rest.

**Corruption level.** We define the *corruption level* of a decoder clause as a *proportion of the false reconstructions in the ground atoms reconstructed by the decoder clause*. This turns out to be an important notion: if the corruption level of a decoder clause is greater than 0.5 then the decoder clause cannot improve the objective function as it introduces more *false* than *true reconstructions*. We remove the candidate clauses that have a corruption level  $\geq 0.5$ .

These strategies are very effective: applying all three of them during the experiments has cut out more than 50 % of candidate clauses.

## 4 Experiments and results

The experiments aim at answering the following question:

**Q:** *Does learning from latent representations created by Alps improve the performance of an SRL model?*

We focus on learning generative SRL models, specifically generative *Markov Logic Networks* (MLN) [Richardson and Domingos, 2006]. The task of generative learning consists of learning a single model capable of answering queries about any part of a domain (i.e., any predicate). Learning an SRL model consists of searching for a set of logical formulas that will be used to answer the queries. Therefore, we are interested in whether learning the structure of a generative model in *latent space*, and decoding it back to the original space, is more effective than learning the model in the original data space.

We focus on this task primarily because no other representation learning method can address this task. For instance, embeddings vectorise the relational data and thus cannot capture the generative process behind it, nor do they support conditioning on evidence.

The deterministic logical mapping of Alps might seem in contrast with the probabilistic relational approaches of SRL. However, that is not the case as the majority of SRL approaches consider data to be deterministic and express the uncertainty through the probabilistic model.

**Procedure.** We divide the data in training, validation and test sets respecting the originally provided splits. The models are learned on the training set, their hyper-parameters tuned on the validation set (in the case of Alps) and tested on the test set. This evaluation procedure is standard in DL, as full cross-validation is infeasible. We report both AUC-PR and AUC-ROC results for completeness; note, however, that the AUC-PR is the more relevant measure as it is less sensitive to class imbalance [Davis and Goadrich, 2006], which is the case with the datasets we use in the experiments. We evaluate the MLNs in a standard way: we query facts regarding one specific predicate given everything else as evidence, and repeat it for each predicate in the test interpretation.

**Models.** We are interested in whether we can obtain better SRL models by learning from the latent data representation. Therefore, we compare the performance of an MLN learned on the original representation (the **baseline** MLN) and an

MLN learned on the latent representation (the **latent** MLN) resulting from Alps. To allow the comparison between the latent and the baseline MLNs, once the latent MLN is learned we add the corresponding decoder clauses as deterministic rules. This ensures that the baseline and latent MLNs operate in the same space when being evaluated.

Both the baseline and the latent MLNs are obtained by the BUSL learner [Mihalkova and Mooney, 2007]. We have experimented with more recent MLN learner LSM [Kok and Domingos, 2010], but tuning its hyper-parameters proved challenging and we could not get reliable results. Note that our main contribution is a method for learning Alps and subsequently the latent representation of data, not the structure of an MLN; MLNs are learned on the latent representation created by Alps. Therefore, the exact choice of an MLN learner is not important, but whether latent representation enables the learner to learn a better model is.

**Practical considerations.** We limit the expressivity of MLN models to formulas of length 3 with at most 3 variables (also known as a *lifiable* class of MLNs). This does not sacrifice the predictive performance of MLNs, as shown by Van Haaren *et al.* [2016]. Imposing this restriction allows us to better quantify the contribution of latent representations: *given a restricted language of the same complexity, if the latent MLN performs better that is clear evidence of the benefit of latent representations*. The important difference when performing inference with a latent MLN is that each latent predicate that could have been affected by the removal of the test predicate (i.e., the test predicate is present in the body of the encoder clause defining the specific latent predicate). Hence it has to be declared *open world*, otherwise MLNs will assume that all atoms not present in the database are *false*.

**Alps hyper-parameters.** As with standard auto-encoders, the hyper-parameters of Alps allow a user to tune the latent representation to its needs. To this end, the hyper-parameters pose a trade-off between the expressivity and efficiency. When learning latent representations, we vary the length of the encoder and decoder clauses separately in  $\{2, 3\}$  and the compression level (the  $\alpha$  parameter) in  $\{0.3, 0.5, 0.7\}$ .

**Data** We use standard SRL benchmark datasets often used with MLN learners: Cora-ER, WebKB, UWCSE and IMDB. The descriptions of the datasets are available in [Mihalkova and Mooney, 2007; Kok and Domingos, 2010], while the datasets are available on the Alchemy website<sup>2</sup>.

### 4.1 Results

The results (Figure 3) indicate that BUSL is able to learn better models from the latent representations. We observe an improved performance, in terms of the AUC-PR score, of the latent MLN on all datasets. The biggest improvement is observed on the Cora-ER dataset: the latent MLN achieves a score of 0.68, whereas the baseline MLN achieves a score of 0.18. The IMDB and WebKB datasets experience smaller but still considerable improvements: the latent MLNs improve the AUC-PR scores by approximately 0.18 points. Finally, a more moderate improvement is observed on the UWCSE

<sup>2</sup><http://alchemy.cs.washington.edu/>

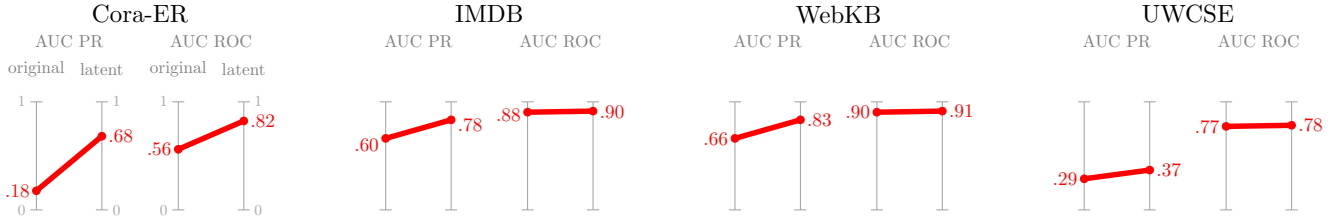


Figure 3: The MLN models learned on the latent data representations created by Alps outperform the MLN models learned on the original data representation, in terms of the AUC-PR scores (red line indicate the increase in the performance), on all dataset. The AUC-ROC scores, which are less reliable due to the sensitivity to class imbalance, remain unchanged.

dataset: the latent MLN improves the performance for 0.09 points.

These results indicate that latent representations are a useful tool for relational learning. The latent predicates capture the data dependencies more explicitly than the original data representation and thus can, potentially largely, improve the performance. This is most evident on the Cora-ER dataset. To successfully solve the task, a learner has to identify complex dependencies such as *two publications that have a similar title, the same authors and are published at the same venue are identical*. Such complex clauses are impossible to express with only three predicates; consequently, the baseline MLN achieves a score of 0.18. However, the latent representation makes these pattern more explicit and the latent MLN performs much better, achieving the score of 0.68.

Neural representation learning methods are sensitive to the hyper-parameter setup, which tend to be domain dependent. We have noticed the same behaviour with Alps by inspecting the performance on the validation set (details in the supplement). The optimal parameters can be selected, as we have shown, on a validation set with a rather small grid as Alps have only three hyper-parameters.

**Runtime** Figure 4 summarises the time needed for learning a latent representation. These timings show that, despite their combinatorial nature, Alps are quite efficient: the majority of latent representations is learned within an hour, and a very few taking more than 10 hours (this excludes the time needed for encoding the problem to COP, as we did not optimise that step). In contrast, inference with MLN takes substantially longer time and was the most time-consuming part of the experiments. Moreover, the best result on each dataset (Figure 3) is rarely achieved with the latent representation with the most expressive Alp, which are the runs that take the longest.

## 5 Related work

The most prominent paradigm in merging SRL and DL are (knowledge) graph embeddings [Nickel *et al.*, 2016; Hamilton *et al.*, 2017]. In contrast to Alps, these methods do not retain full relational data representation but approximate it by vectorisation. Several works [Minervini *et al.*, 2017; Demeester *et al.*, 2016] impose logical constraints on embeddings but do not retain the relational representation.

Kazemi and Poole [2017] and Sourek *et al.* [2016] introduce symbolic variants of neural networks for relational data. Evans and Grefenstette [2018] introduce a differentiable way to learn predictive logic programs. In contrast to Alps they focus on predictive learning, often with specified architecture.

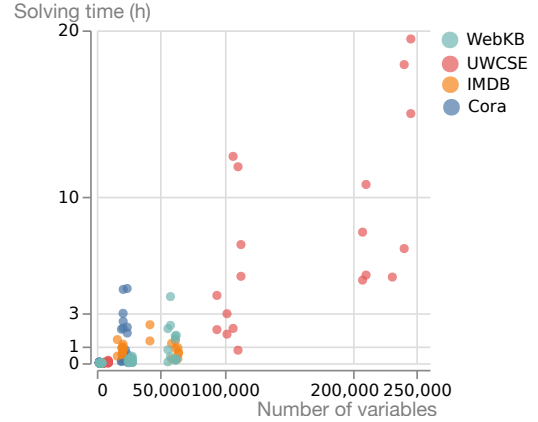


Figure 4: Relationship between runtimes and the number of variables.

Several works integrate neural and symbolic components, but do not explore learning new symbolic representation. Rocktäschel and Riedel [2017] introduce a differentiable version of Prolog’s theorem proving procedure. Manhaeve *et al.* [2018] combine symbolic and neural reasoning into a joint framework, but only consider the problem of parameter learning not the (generative) structure learning.

Inventing a new relational vocabulary defined in terms of the provided one is known as *predicate invention* in SRL [Kramer, 1995; Cropper and Muggleton, 2018]. In contrast to Alps, these methods create latent concepts in a weakly supervised manner – there is no direct supervision for the latent predicate, but there is indirect supervision provided by the accuracy of the predictions. An exception to this is the work by Kok and Domingos [2007]; however, it does not provide novel language constructs to an SRL model, but only compresses the existing data by identifying entities that are *identical*.

We draw inspiration from program induction and synthesis [Gulwani *et al.*, 2017], in particular, unsupervised methods for program induction [Ellis *et al.*, 2015; Lake *et al.*, 2015]. However, these methods do not create new latent concepts.

## 6 Conclusion

This work introduces *Auto-encoding Logic Programs* (Alps) – a novel logic-based representation learning framework for relational data. The novelty of the proposed framework is that it learns a latent representation in a symbolic, instead of a



gradient-based way. It achieves that by relying on first-order logic as a data representation language, which has a benefit of exactly representing the rich relational data without the need to approximate it in the embeddings spaces like many of the related works. We further show that learning Alps can be cast as a constraint optimisation problem, which can be solved efficiently in many cases. We experimentally evaluate our approach and show that learning generative models from the relational latent representations created by Alps results in substantially improved AUC-PR scores compared to learning from the original data representation.

This work shows the potential of latent representations for the SRL community and opens challenges for bringing these ideas to their maturity; in particular, the understanding of the desirable properties of relational representations and the development of scalable methods to create them.

## References

- [Ahuja *et al.*, 2002] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Appl. Math.*, 123(1-3):75–102, 2002.
- [Cropper and Muggleton, 2018] Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. *Machine Learning*, 2018.
- [Davis and Goadrich, 2006] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *ICML, ICML*, pages 233–240. ACM, 2006.
- [Demeester *et al.*, 2016] Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Lifted rule injection for relation embeddings. In *EMNLP*, pages 1389–1399, 2016.
- [Ellis *et al.*, 2015] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In *NIPS*, pages 973–981, 2015.
- [Evans and Grefenstette, 2018] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.
- [Gay *et al.*, 2015] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming*, pages 140–148. Springer, 2015.
- [Getoor and Taskar, 2007] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.
- [Goodfellow *et al.*, 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [Gulwani *et al.*, 2017] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.
- [Hamilton *et al.*, 2017] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.
- [Hinton and Salakhutdinov, 2006] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
- [Kazemi and Poole, 2017] Seyed Mehran Kazemi and David Poole. Relnn: A deep neural model for relational learning. In *AAAI*, 2017.
- [Kipf and Welling, 2017] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.
- [Kok and Domingos, 2007] Stanley Kok and Pedro Domingos. Statistical predicate invention. In *ICML, ICML*, pages 433–440. ACM, 2007.
- [Kok and Domingos, 2010] Stanley Kok and Pedro Domingos. Learning markov logic networks using structural motifs. In *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML*, pages 551–558. Omnipress, 2010.
- [Kramer, 1995] Stefan Kramer. Predicate Invention: A Comprehensive View. *Technical report*, 1995.
- [Lake *et al.*, 2015] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350:1332–1338, 2015.
- [Manhaeve *et al.*, 2018] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3749–3759. Curran Associates, Inc., 2018.
- [Mihalkova and Mooney, 2007] Lilyana Mihalkova and Raymond J. Mooney. Bottom-up learning of markov logic network structure. In *ICML*, 2007.
- [Minervini *et al.*, 2017] Pasquale Minervini, Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Adversarial sets for regularising neural link predictors. In *UAI*, 2017.
- [Mitchell, 1997] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.
- [Nickel *et al.*, 2016] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.
- [OscaR Team, 2012] OscaR Team. OscaR: Scala in OR, 2012. Available from <https://bitbucket.org/oscarlib/oscar>.
- [Richardson and Domingos, 2006] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.
- [Rocktäschel and Riedel, 2017] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *NIPS*, pages 3788–3800. Curran Associates, Inc., 2017.
- [Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*

(*Foundations of Artificial Intelligence*). Elsevier Science Inc., 2006.

[Sourek *et al.*, 2016] Gustav Sourek, Suresh Manandhar, Filip Zelezny, Steven Schockaert, and Ondrej Kuzelka. *Learning Predictive Categories Using Lifted Relational Neural Networks*, volume 10326 of *LNAI*. Springer International Publishing, 2016.

[Trouillon *et al.*, 2019] Théo Trouillon, Eric Gaussier, Christopher R. Dance, and Guillaume Bouchard. On inductive abilities of latent factor models for relational learning. *Journal of Artificial Intelligence Research*, 64, 2019.

[Van Haaren *et al.*, 2016] Jan Van Haaren, Guy Van den Broeck, Wannes Meert, and Jesse Davis. Lifted generative learning of markov logic networks. *Machine Learning*, 103(1):27–55, 2016.