NEURALLOG: A NEURAL LOGIC LANGUAGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Application domains that require considering relationships among objects which have real-valued attributes are becoming even more important. In this paper we propose NeuralLog, a first-order logic language that is compiled to a neural network. The main goal of NeuralLog is to bridge logic programming and deep learning, allowing advances in both fields to be combined in order to obtain better machine learning models. The main advantages of NeuralLog are: to allow neural networks to be defined as logic programs; and to be able to handle numeric attributes and functions. We compared NeuralLog with two distinct systems that use first-order logic to build neural networks. We have also shown that NeuralLog can learn link prediction and classification tasks, using the same theory as the compared systems, achieving better results for the area under the ROC curve in four datasets: Cora and UWCSE for link prediction; and Yelp and PAKDD15 for classification; and comparable results for link prediction in the WordNet dataset.

1 INTRODUCTION

Deep learning has been remarkably successful on a wide range of tasks (LeCun & Bengio, 1998). However, most of those task are based on propositional numeric inputs, assuming that there are no relation between the examples. Logic programming (Brachman & Levesque, 2004), on the other hand, uses logic programs to describe and to reason about structured and multi-relational data, but it struggles to deal with uncertainty and noise, two factors inherent to real world problems.

In order to overcome these limitations, we propose *NeuralLog*, a first-order logic language that is compiled to a neural network. The main goal of NeuralLog is to bridge logic programming and deep learning in order to exploit the advantages of these fields in both discrete and continuous data.

Another main contribution of NeuralLog is to allow one to abstract the design of deep neural networks by the use of a logic language, which would facilitate the use of deep learning models by people that are not familiarized with common programming languages, since logic languages have been used for a long time as modelling language for different domains.

In addition, NeuralLog allows to define neural networks to address relational tasks. It also supports numeric attributes and the use of numeric functions, thus providing the ability to create generic neural networks; in contrast with most of the works in this field, that are restricted at combining logic with neural networks on discrete domains composed by relations among entities. Furthermore, the logic program is mapped to a neural network in such a way that the logic of the program can be easily identified in the structure of the network and vice versa, making the model easier to interpret.

We compare NeuralLog with two state-of-the-art systems in two very different domains: link prediction with TensorLog (Cohen et al., 2017) and classification with RelNN (Kazemi & Poole, 2018). Our results show that NeuralLog performs as well as, and often better than, these two other systems on tasks they were designed for, and support our claim that NeuralLog is a significant contribution towards a flexible neural relational system.

The remaining of the paper is organised as follows: in Section 2, we show some related works; in Section 3, we give the background knowledge to follow this work; in Section 4 we present NeuralLog; and the performed experiments in Section 5; finally, we present our conclusion and directions of future work in Section 6.

2 RELATED WORK

There are many ways to combine first-order logic with neural networks, from using logic features as input to other machine learning models (Lodhi, 2013; Dash et al., 2018) to the use of the logic rules to define the structure of the neural network (França et al., 2014; Nguembang Fadja et al., 2017; Sourek et al., 2018; Dong et al., 2019; Kaur et al., 2019).

TensorLog (Cohen et al., 2017) is a system that performs inference of first-order logic by using numeric operations. It represents the logic facts as matrices and performs logic inference through mathematical operations on those matrices. Another well-known work that relies on matrix representations is Logic Tensor Network (Serafini & d'Avila Garcez, 2016), which uses a tensor network (Socher et al., 2013) to predict the confidence of the logic formulas.

NeuralLog uses the same underlying idea of representing logic facts as matrices and mathematical operations to compute logic rules. However, it differs from TensorLog in two main aspects: (1) in the way how the logic rules are compiled to a neural network; and (2) by treating numeric attributes as logic terms that can be flexibly manipulated and also given as input to numeric functions. This allows NeuralLog to be more flexible than TensorLog.

Similar to TensorLog, RelNN uses matrices and vectors operations to perform relational learning, using convolution neural networks (Kazemi & Poole, 2018). They use the number of logic proofs of rules as features, that can also be combined into other rules, multiplied by weights, added to bias, and passed as input to activation functions. Both TensorLog and RelNN can tune the weights of facts to better predict a set of examples.

We compared NeuralLog with TensorLog and RelNN, in particular, because in those systems the structure of the neural network clearly reflects the logic theory.

It is important to notice that the use of first-order logic is not the only way to allow neural networks to be applied to relational tasks. However, the other approaches, although promising, are beyond the scope of this paper.

3 BACKGROUND KNOWLEDGE

In this work we use first-order logic to represent and learn neural network models in multi-relational tasks. The knowledge is represented as a set of facts that is called knowledge base (KB). A fact is composed by a *predicate* (relation) and, possibly, a set of constants (entities). A fact can be represented as $p(s_1, \ldots, s_n)$., where p is the predicate, and s_i are the constants; and states that there is a relation of type p among the constants s_i. A fact may also have no constants, in this case, we say the fact is propositional; for instance $rain$ states that it rains.

To reason about this knowledge we use Horn Clauses (Horn, 1951). A Horn Clause, also referred as a rule, has the form: $p_0(.) \leftarrow p_1(.) \wedge \cdots \wedge p_n(.)$. Where p_i are predicates and $(.)$ are lists of *terms*, a term can be either a *constant* or a *variable*. A constant is represented by a string starting with a lower case letter, while variables start with an upper case letter. The predicate followed by its terms, in the clause, is called a literal. A literal is an atom or the negation of an atom. Although we do not consider negation in this work, we will use the term literal to refer to atoms in the body of a rule. The atom in the left-hand side of the rule is called the head, and the conjunction of literals in the right-hand side is called the body. The arity of a predicate is the number of terms a literal of this predicate accepts. A predicate p of arity n is also represented as p/n. In this work, we do not consider first-order logic functions, only numeric functions, as will be described in more detail in the next section. We also limit the facts to have arity of at most two.

An atom is proved (considered to be true) whenever it matches a fact in KB or it can be proved by a rule. A variable can be replaced by a constant in order to make the atom to match a fact. A rule proves the atom in its head whenever there is a substitution of its variables that makes **all** proves all the literals in its body, possible by using other rules. A negated literal (not $\mathbf{A}$) is considered to be true whenever we cannot prove $\mathbf{A}$. We base the NeuralLog syntax on DataLog (Abiteboul et al., 1995), adding the weighted fact syntax from ProbLog (De Raedt et al., 2007).

Generically, an artificial neural network is a directed graph of layers, where each layer receives as input the results of its income layers and computes its output based on its inputs and its internal

parameters. The learning process consists of finding set of the parameters of the layers that better describes the training data. This is usually done by a gradient descent algorithm (Haykin, 1994).

4 NEURALLOG

In this section, we present NeuralLog, a first-order logic language that describes neural networks to perform logic inference.

Since neural networks process numeric inputs, it is convenient to describe the KB in a numeric form. More precisely, we are given a logic program that can be seen as a triple (E, F, R), where E is the set of entities available in the domain; F is the set of *facts*; and R is the set of rules. Each fact may have at most two terms and always has a weight w (by default 1), that might be omitted.

Similarly to (Cohen et al., 2017), NeuralLog stores knowledge as (sparse) matrices, to be evaluated through algebraic operations on those matrices. First, we construct an index for E by assigning to each entity an integer value in $[0, n)$, where $n = |E|$. Next, for each binary predicate p, we create a matrix $P \in \mathbb{R}^{n \times n}$ where $P_{ij} = w$ if there is a fact $p(e_i, e_j)$ with weight w in F, where i and j are the indices of the entities e_i and e_j, respectively; otherwise, $P_{ij} = 0$. Predicates of arity 1 (unary) are mapped to vectors; and propositional predicates to scalars. An entity (or constant) is represented by an one-hot vector, having the value of 1 for the entry correspondent to the index of the entity and the value of 0 for every other entries.

Real-Valued Data One of our goals in the design of NeuralLog was to support numeric value data. Naively introducing real values as any other constant in E could lead to very large matrices. Instead, we assume that real values are bounded to entities in KB as attributes of the entities. Our implementation uses two vectors ($\mathbf{p_v}, \mathbf{p_w} \in \mathbb{R}^n$) for each attribute predicate p. Where $\mathbf{p_v}_i = a$ and $\mathbf{p_w}_i = w$ if there is a fact $p(e_i, a)$ with weight w in F, and i is the index of the entity e_i; otherwise, $\mathbf{p_v}_i = \mathbf{p_w}_i = 0$. As an example, the predicate $length$ will consist of facts such as $length(x, 1.5)$, informing that entity x has an attribute *length* of value 1.5.

4.1 INFERENCE

The key idea of NeuralLog is to transform a logic program into a neural network such that both rule inference and the integration of different rules can be performed through matrix operations.

The goal of NeuralLog is to find all possible entities that might replace the variable Y, associated with the entity $X = \mathbf{a}$, with relation p, for a given a query of the form $? - p(X, Y)$; with respect to the knowledge base and clauses.

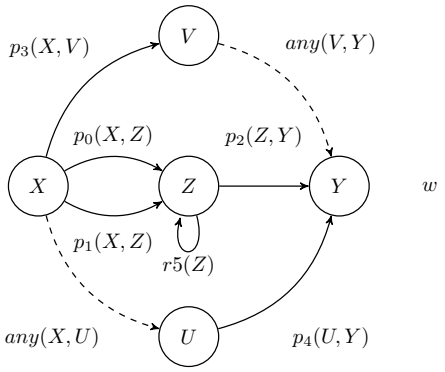
To briefly explain the translation, imagine a KB composed of facts for a single binary predicate $p/2$. The implementation is a matrix P, with an one-hot row vector $\mathbf{a}$ representing a given entity in KB, as described above. Given $X = \mathbf{a}$ and a query $? - p(X, Y)$, $Y = \mathbf{a}^\intercal P$ describes the values of Y. In the other direction, if we would like to compute the vector for the entities X related to a where $Y = \mathbf{a}$, we would do so by transposing the matrix P such as $X = a^\intercal P^\intercal = (Pa)^\intercal$. We extend this method to unary and attribute facts by using the element-wise multiplication.

Finally, the result of the query $? - p(X, Y)$, for $X = \mathbf{a}$ will be the set of facts, for each possible entities in KB, where the weight of each fact $p(a, e_i)$, where e_i is the entity assigned to position i, would be Y_i, from the resulting vector Y.

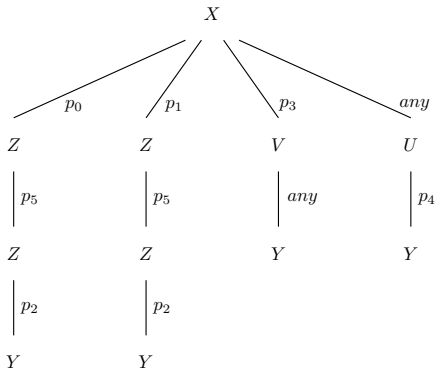
4.2 RULE GRAPH CONSTRUCTION

In this section, we extend the previous inference approach in order to compute the output value of arbitrary rules. In order to do so, we start by creating an undirected graph from the rule, where the nodes are the terms in the rule; and there is an edge between two nodes if they both appear together as terms in a literal of the body of the rule. Then, we define the last term in the head of the rule as the output (destination) node and the remaining terms in the head as inputs (sources) nodes.

Assuming that edges represent influence, we perform a breadth-first search in this graph, in order to find all the paths from the source node to the destination node. We constraint ourself with rules



(a) The DAG representation of the rule



(b) All paths from X to Y represented as a tree

Figure 1: Example of the DAG representation (on the left-hand side) and the found paths (on right-hand side) of the rule

whose arity of the head are greater than zero and we use a topological order to disallow back-edges and loops.

Figure 1 shows an example of the DAG and the found paths (as a tree) of the rule below:

$$target(X,Y) \leftarrow p_0(X,Z) \wedge p_1(X,Z) \wedge p_2(Z,Y) \wedge p_3(X,V) \wedge p_4(U,Y) \wedge p_5(Z) \wedge w.$$

In this case, our neural network (NN) will be a function $Y = f(X)$. A first step shows edges $X - Z$ and $X - V$. In a second step we obtain $Z - Y$; as we do not allow cycles, V has no path to the output, and U is disconnected from the network.

In order to improve the compiled network, by accounting by the influence of disconnected terms, we use a special $any/2$ predicate, inspired by TensorLog (Cohen et al., 2017), that is essentially true for any pair of terms. There are two possible cases where we use this predicate:

1. connected nodes that do not have a path to target can still influence the target (Y): this is implemented by adding an edge from the node to the output, as shown in $V - Y$;
2. unconnected nodes may also influence the output; we use $any/2$ to connect the input to the node, as shown in $X - U$, in order to consider the relation from U.

Notice that the second case introduces new partial paths and we need to restart the algorithm to propagate these paths. The process repeats until no new edges are needed to be added. The algorithm always terminates as the number of steps is bounded by the number of nodes. The algorithm to find those paths are described in Appendix A.

Given the found paths, we construct a directed acyclic graph (DAG) representation of the rule. Equal edges in different paths are collapsed into a single edge; and different incoming $any/2$ edges to the destination are represented as a single any/n literal, where n is the number of distinct terms appearing in all these edges and the destination is the last term. All unary predicates (which would represent a loop in the graph) is then added to their correspondent edges.

The inference of the rule, given its inputs, is represented by the result of the destination node in the DAG, multiplied by any disconnected grounded literal in the body of the rule. Those are grounded literals that do not appear in any path, which include the propositional literals. Since they do not have variables, they are represented as scalars. It is important to notice that, in the example, w is a propositional predicate, represented as a scalar, and not a term.

In order to compute the result of a node we combine the values of its incoming edges by an element-wise multiplication (representing a logic **AND**); then we combine the values of the unary predicates of the node, if any, in the same order they appear in the rule.

In order to compute the value of an edge (which represents a literal) we multiply the vector representing the input term of the edge by the matrix representation of the edge's predicate. If the

predicate is represented as a vector (in the case of unary predicate), we perform an element-wise multiplication, instead. If it is an attribute predicate, we perform the element-wise multiplication between the input term, the weight and the attribute vectors. Since we need the values of the incoming nodes to compute the value of the current node, we recursively transverse the DAG until we reach a source node, whose value in given from the input.

The any/n predicate has a special treatment. First, it is important to notice that the any/n predicate can only appear in two specific situations, as enumerated above: (1) connecting the end of a path to the destination; or (2) connecting an input term to the beginning of a path that will lead to the destination (which was computed from the destination to the input and then reversed).

In order to compute the first case of the any/n predicate, for each term of the predicate, except the last one, we compute the vector resulting from the term and sum it to get a scalar. For the last term (which is always the destination term of the rule), we compute the vector resulting from this term, based on its unary predicates, by passing a 1 vector as input value for the predicates; then, we multiply it by the scalar representation of the previous terms.

Intuitively, the result represents the multiplication of the sum of the results of each (any) term by the results of each (any) entities of the last term. The final result is combined with other possible edges arriving at the destination term. In this case, the unary predicate is not combined with the final result, since it has been already combined by the any/n predicate.

The second case of the any/n occurs when we have edges connected to the destination that do not lead to any input. As such, the first $n - 1$ terms are input terms and the last term is the beginning of the path that leads to the destination. In this case, we pass a 1 vector, representing the input of the term. Then, we apply the existent unary predicates, if any, to it. In numeric terms, the 1 vector represents the union of all (any) logic constants. In logic terms, the node represents a free variable in the rule. Thus, the result of the rule does not depend on it, meaning that it is valid for any constant in this variable.

In addition, NeuralLog allows the use of numeric functions. The functions are applied, in order of appearance in the rule, to the computed vector of its inputs. There is no explicit differentiation between functions and predicates in NeuralLog language. The difference is that predicates have facts defined in the KB and the computation is the multiplication of the input terms with the matrix (or vector) representation of the facts of the predicates, while the result of function predicates is the application of the function to the input terms. These functions can be any differentiable function, including other network layers or even whole neural network models.

If a relation involves a constant, we multiply the input (output) variable with the vector representation of the input (output) constant. It represents a logic **AND** of the input (output) variable with the input (output) constant. We do the same for the any/n predicate.

4.3 Network Construction

We have explained how we represent the knowledge data and how to compute an individual rule represented by a Horn clause. In this section, we show how to construct a neural network from a first-order logic program (a set of logic rules and facts).

In order to build the network we start by creating a *Literal Layer* for each target predicate. This layer receives as input all the rules and facts that have the target predicate in their heads. The output of a literal layer is the sum of the results of the *Fact Layer*, representing the matrix of facts in the KB and the *Rule Layers* of the target literal, similar to (Sourek et al., 2018); given the input terms of the literal. The sum of the results represents a logic **OR**.

The output of the *Fact Layer* is the input vector multiplied by the matrix (or vector) of the weights of the facts in the KB. For attribute facts, their weight and value vectors are combined by an element-wise multiplication.

Recursively, for each rule connected to the target predicate node, we create a *Rule Layer*, and connect, as its inputs, the corresponding *Literal Layers* from the rule's body. We represent function predicates in the body of the rule as *Function Layers*, the output of a function layer is the application of the function to the vector representation of its inputs terms. The computation of the rule layer is performed as explained in the subsection above, using the computation performed by the literal

layers, instead of multiplying the matrices of the facts. We repeat this process until no more layers are created.

In this way, we unfold the network from the structure of the logic rules, similar to (Nguembang Fadja et al., 2017). However, our language allows the use of attribute predicates and functions. Whenever a recursion is found in the rules, it is unfolded until a pre-defined depth. Since the rule inference operation is differentiable, the unfolding of the rules, and thus the whole network, is differentiable.

The *Literal Layers* of the target predicates receive as input the vector corresponding to their input terms, and represent the inputs of the neural network; while the result of these layers are the output of the network. Note that if we want to compute an inverted literal, by going from the last term of the literal to the first, we can simply use the transpose matrix of the relation, for the fact layer; and compute the reversed paths, for the rule layers. The depth of the neural network is defined by the logic program, and a hierarchy of abstract concepts can be created by the use of predication invention.

In order to learn a task from data, we allow the user to specify predicates whose weights will be adjusted from examples. In this way, the weights of those predicates will become parameters to be learned. After training, we can extract the learned weights to update the logic program that generated the network. We show some examples and the algorithms to find the paths in the rules in appendix.

NeuralLog is implemented in Python (Van Rossum & Drake, 2009) using TensorFlow (Abadi et al., 2015) as backend and is available at `<placeholder>`. Therefore NeuralLog provides a very flexible approach, it has some limitations, for instance, it is unable of handling propositional predicates in the head of the rules. We also limited the use of facts to predicates with arity up to two, since TensorFlow, in its used version (2.0), cannot handle sparse tensors with more than two dimensions and the use of dense tensors to represent facts with arity bigger than two would easily consume too much memory. On the other hand, predicates of arity bigger than two are useful when used as target predicates, since we can provide more information for each example; and they are also useful in the rules, to combine different information, facilitating the definition of siamese network (Bromley et al., 1993), for instance.

We opt to use multiplication to represent the logic **AND** and sum to represent the logic **OR** because they have a similar meaning to what we would expect from logic programs and they are already provided by TensorFlow. Furthermore, we allow those functions to be changed in the logic program, given the implementation is provided.

It is important to point out that the first-order logic expressiveness cannot be exactly represented by means of differentiable function operations. In this work, we present a logic language to abstract the structure of neural networks and we try to make the semantics of the neural network as close as possible as the one of the logic program. However, may not be exactly the same.

5 EXPERIMENTS

In order to demonstrate the flexibility of NeuralLog, we compared it with two distinct relational systems that uses first-order logic to create relational neural networks. In these experiments, we would like to answer two main research questions, in order to address the flexibility of NeuralLog: (**Q1**) does NeuralLog perform well on *link prediction* tasks? And (**Q2**) does NeuralLog perform well on *classification* tasks?

In order to answer each question, we compared NeuralLog with other similar systems proposed specifically for each type of task. The first one is TensorLog (Cohen et al., 2017), which uses belief propagation to compile logic theories into neural networks. The second one is RelNN (Kazemi & Poole, 2018), which learns relational neural networks that use the number of instantiation of logic rules as features to predict the value of a predicate for a given entity.

TensorLog and RelNN are two distinct systems that use logic to build neural networks. On one hand, TensorLog focus at *link prediction*, by retrieving queries of the form $q(a, X)$, where the goal is to find all the entities that are related (as second term) to the entity a (as first term) through the relation q. On the other hand, RelNN focus at *classification*, by predicting the value of relations of the form $q(X)$, for a given entity a, based on the relations of a, defined by logic rules.

Table 1: Size of the datasets

Dataset	Facts	Train		Test	
		Pos	Neg	Pos	Neg
Cora	44,711	28,180	24,104	7,318	3,660
WordNet	106,088	35,354	53,732	5,000	9,098
UWCSE	3,146	90	49,622	23	3,320
Yelp	45,541	2,160	1,368	540	342
PAKDD15	127,420	5,301	18,702	1,325	4,676

First, we compared NeuralLog with TensorLog. Since TensorLog predicts the values of the entities related to a given entity, we ran link prediction experiments in three datasets: the Cora dataset (Poon & Domingos, 2007), and the UWCSE dataset (Richardson & Domingos, 2006), two very popular datasets among ILP systems; and the WordNet dataset (Miller, 1995), a frequently used benchmark dataset.

The link prediction task is the task of predicting the entities that relate to a given entity, through a target predicate. To perform this task, we have a set of examples in the form $q(a, x)$, where we give the first entity to the system a, and wants to predict the last entity x.

Finally, we compared NeuralLog with RelNN in classification tasks in two different datasets: the Yelp dataset and the PAKDD15 dataset, both datasets were available with RelNN at (Kazemi & Poole, 2018). For the Yelp dataset, the task is to predict whether a restaurant serves mexican food through the relation $mexican(X)$; while in the PAKDD15, the task is to predict the gender of a person, based on the movies the person watches, through the relation $male(X)$.

5.1 METHODOLOGY

We use the same logic theory available with the compared systems for the respective datasets. In our system, for Cora, WordNet and UWCSE, we applied a tanh function to the output of the rules, and also added a bias to the target predicate; then, applying another tanh to the weighted sum of the rules and the bias for the target predicate. Those functions act as activation and output function, respectively. The theories for the Yelp and PAKDD15 already had activation functions and biases, the only difference is that RelNN used the sigmoid function, while we use the tanh function.

In the case of the UWCSE, we have used a manually written theory provided by Alchemy[1]. Since Alchemy support a more complex logic set, we removed rules that uses logic features not support by TensorLog and NeuralLog. The UWCSE dataset also contains facts of arity three, which are not supported by either the systems, we converted them by concatenating two terms that always appears together in the theory. We also added two additional predicates to extract either term, given the concatenated terms.

A bias rule has the form $target(X, Y) \leftarrow b.$, where b is a fact to be learned. Since the variables of the head do not depend on its body, the rule is always true. Then, the rule is summed to other rules with the same head, acting as a bias.

We use adagrad to optimize the mean square error, running for 10 epochs for each dataset. We ran each experiment 10 times and reported the average area under the ROC curve for each dataset. We set the recursion depth to 1, meaning that the recursive rule is applied only once. For TensorLog and RelNN, we used the parameters reported at their papers, (Cohen et al., 2017) and (Kazemi & Poole, 2018), respectively. We use holdout with Cora and WordNet; and 5-folds cross-validation with UWCSE, Yelp and PAKDD15, since the datasets were divided that way.

In the WordNet dataset, we used 25% of the facts in the train set and the remaining as KB. Since the WordNet has no negative examples, we artificially generated approximately 2 negative examples for each entity appearing in the first position of a target relation. We did this by replacing the second entity of the example by another entity appearing in same relation, which does not appear as positive example, following the Local Closed World Assumption (Galárraga et al., 2013).

[1] `http://alchemy.cs.washington.edu/`

Table 2: Area under ROC curve for Cora dataset

Relation	Individual		Together	
	NeuralLog	TensorLog	NeuralLog	TensorLog
S. Author	$\mathbf{1.0000 \pm 0.0000}$	0.9276 ± 0.0000	$\mathbf{0.9273 \pm 0.0867}$	0.8853 ± 0.0019
S. Bib	0.9174 ± 0.0073	$\mathbf{0.9339 \pm 0.0000}$	$\mathbf{0.9299 \pm 0.0085}$	0.8112 ± 0.0487
S. Title	$\mathbf{0.8710 \pm 0.0003}$	0.5000 ± 0.0000	$\mathbf{0.8728 \pm 0.0046}$	0.8442 ± 0.0274
S. Venue	$\mathbf{0.7810 \pm 0.0077}$	0.5000 ± 0.0000	$\mathbf{0.7561 \pm 0.0184}$	0.7000 ± 0.0000

The size of the datasets can be seen in Table 1. The UWCSE, Yelp and PAKDD15 lines show the average size of the cross-validation folds.

5.2 Results

We applied both NeuralLog and TensorLog to each relation of the Cora dataset. When learning each relation individually, NeuralLog achieved better results than TensorLog for the *Same Author* relation, while it is slightly worst for the *Same Bib* relation. Strangely, TensorLog was not able to learn the *Same Title* and *Same Venue* relations in any of the 10 runs. We repeated the Cora experiments learning all the relations together. In this setup, NeuralLog outperformed TensorLog in all relations. We reported these results in Table 2. It is worth pointing out that TensorLog was able to learn only in 6 and 1 out of 10 runs, for the *Same Title* and *Same Venue* relations, respectively. We reported the average of the successful runs.

In the WordNet dataset, NeuralLog achieved slightly worst results than TensorLog. NeuralLog achieved $\mathbf{0.6593}$ and $\mathbf{0.6679}$ for the weighted and arithmetic average, respectively; of the area under ROC curve of the 18 relations. Against $\mathbf{0.6604}$ and $\mathbf{0.6766}$ from TensorLog.

In the UWCSE dataset, NeuralLog achieved $\mathbf{0.9509}$ for the area under the ROC curve, while TensorLog was only able to achieve $\mathbf{0.7107}$. It is important to note that we were not able to run TensorLog with the same theory as NeuralLog, since it raised an error when rules containing free variables or rules whose the output variable had no inputs connected to it, for more details, see Appendix B. Using the same limited theory as TensorLog, NeuralLog achieved $\mathbf{0.7111}$ on area under the ROC curve.

Finally, in comparisons with RelNN, NeuralLog achieved significantly better results in both Yelp and PAKDD15 datasets. NeuralLog achieved $\mathbf{0.7652}$ and $\mathbf{0.8052}$, against $\mathbf{0.7454}$ and $\mathbf{0.7562}$ from RelNN, for the Yelp and PAKDD15, respectively.

In most cases, NeuralLog achieved better results than the other systems, in tasks they were designed for, even using the same logic theory. This might be due to better optimisation algorithm, and, possibly, better initial parameters. Nevertheless, our experiments show that NeuralLog is flexible enough to successfully represent other systems based on logic theories and neural networks. Thus, we can affirmatively answer both questions **Q1** and **Q2**.

6 Conclusion

We present NeuralLog, a first-order logic language that is compiled into a neural network. Experiments show that NeuralLog can represent both TensorLog (Cohen et al., 2017) and RelNN (Kazemi & Poole, 2018) programs, achieving even better results in the Cora, UWCSE, Yelp and PAKDD15 datasets; and comparable results in the WordNet dataset. NeuralLog is also capable of representing MLPs and other more complex neural network structures (such as cosine similarity) due to its capability of handling numeric attributes and functions; as well as rules with free variables, which improved the results in the UWCSE dataset.

The main goal of NeuralLog is to bridge advances in both deep learning and logic programming, two distinct fields of machine learning that can complement each other. As future work, we would like to explore ILP algorithms in order to learn the logic programs (and, consequently, the structure of the neural networks) from examples.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, and Eugene Brevdo *et al.* TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015. URL http://download.tensorflow.org/paper/whitepaper2015.pdf.

Serge Abiteboul, Richard Hull, and Victor Vianu (eds.). *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition, 2004. ISBN 1558609326.

Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, pp. 737–744, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

William W. Cohen, Fan Yang, and Kathryn Rivard Mazaitis. Tensorlog: Deep learning meets probabilistic dbs, 2017.

Tirtharaj Dash, Ashwin Srinivasan, Lovekesh Vig, Oghenejokpeme I. Orhobor, and Ross D. King. Large-scale assessment of deep relational machines. In Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese (eds.), *Inductive Logic Programming*, pp. 22–37, Cham, 2018. Springer International Publishing.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pp. 2468–2473, Hyderabad, India, January 2007. Morgan Kaufmann Publishers Inc.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *International Conference on Learning Representations*, 2019.

Manoel V. M. França, Gerson Zaverucha, and Artur S. d'Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning*, 94 (1):81–104, Jan 2014. ISSN 1573-0565. doi: 10.1007/s10994-013-5392-1. URL https://doi.org/10.1007/s10994-013-5392-1.

Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: Association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pp. 413–422, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2035-1. doi: 10.1145/2488388.2488425. URL http://doi.acm.org/10.1145/2488388.2488425.

Simon Haykin. *Neural networks*, volume 2. Prentice hall New York, 1994.

Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

Navdeep Kaur, Gautam Kunapuli, Saket Joshi, Kristian Kersting, and Sriraam Natarajan. Neural networks for relational data, 2019.

Seyed Mehran Kazemi and David Poole. RelNN: A deep neural model for relational learning. In *AAAI-18*, pp. 6367–6375, 2018.

Yann LeCun and Yoshua Bengio. The handbook of brain theory and neural networks. chapter Convolutional Networks for Images, Speech, and Time Series, pp. 255–258. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-51102-9. URL http://dl.acm.org/citation.cfm?id=303568.303704.

Huma Lodhi. Deep relational machines. In Minho Lee, Akira Hirose, Zeng-Guang Hou, and Rhee Man Kil (eds.), *Neural Information Processing*, pp. 212–219, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-42042-9.

George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995. ISSN 0001-0782.

Arnaud Nguembang Fadja, Evelina Lamma, and Fabrizio Riguzzi. Deep probabilistic logic programming. *4th International Workshop on Probabilistic logic programming, PLP 2017*, 11 2017.

Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *Proceedings of the 22Nd National Conference on Artificial Intelligence*, volume 1 of *AAAI'07*, pp. 913–918, Vancouver, British Columbia, Canada, July 2007. AAAI Press.

Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1): 107–136, 2006.

Luciano Serafini and Artur S. d'Avila Garcez. Learning and reasoning with logic tensor networks. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea (eds.), *AI*IA 2016 Advances in Artificial Intelligence*, pp. 334–348, Cham, 2016. Springer International Publishing.

Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 26*, pp. 926–934. Curran Associates, Inc., 2013.

Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezný, Steven Schockaert, and Ondrej Kuzelka. Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62:69–100, 2018.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

A Rule Paths Algorithms and Examples

A.1 Rule Examples

The following examples show two examples of the expressiveness of NeuralLog, each of which with a single rule: (1) a rule where the prediction is weighted by an attribute of the entity; and (2) a rule to aggregate a value, based on the relations of an entity.

Example 1 *consider the following rule:*

$$influence(X, Y) \leftarrow friends(X, Y) \wedge age(X, A) \wedge weight.$$

There will be two possible paths between X and Y: (1) $X \rightarrow friends \rightarrow Y$; and (2) $X \rightarrow age \rightarrow A \rightarrow any \rightarrow Y$.

From the first path, we would get a resulting vector Y' with all non-zero values representing the entities that are friends of X. From the second path, we would have a vector Y'' with the age of X in every position. The final vector, Y would then be given by the element-wise multiplication $Y' \odot Y''$ and then by the scalar $weight$.

Thus, this rule means that a person X influences their friends Y proportionally to his/her. Additionally, we can learn the weight $weight$ for the rule, independent of any entity. Similar kind of rules could be used to retrieve numeric attributes from entities and combine them to create a Multilayer Perceptron *(MLP).*

Example 2 *consider the following rule:*

$$mean\_age\_of\_friends(X) \leftarrow friends(X, Y) \wedge age(Y, A) \wedge mean(A).$$

This rule have a single path, starting from X: $X \rightarrow friends \rightarrow Y \rightarrow age \rightarrow A \rightarrow mean \rightarrow A$.

This path starts by computing vector Y, which represents the friends of X by multiplying X by $friends$. Then, it obtains the vector A, containing the ages of the friends of Y, by multiplying Y by age. Finally, it applies the function $mean$ to the vector A, which computes the mean of all non-zero entries for A, resulting in the average of the age of the friends of X.

Algorithm 1 Find clause paths: Algorithm to find the paths between the sources and the destination terms of a clause and the disconnected literals

Input: The clause ($clause$); and the destination term index ($dest\_index$);
Output: The paths from the source terms to destination; and the disconnected ground literals;

function find_clause_paths($clause, dest\_index$)
    $sources \leftarrow clause.head.terms$
    $compute\_reverse \leftarrow True$
    $destination \leftarrow source[dest\_index]$
    if $|sources| > 1$ **then**
        $sources.remove[dest\_index]$
    else
        $compute\_reverse \leftarrow False$
    end if
    $all\_paths \leftarrow \{\}$
    $all\_visited\_nodes \leftarrow \{\}$
    $sources\_set \leftarrow$ set($sources$)

    for each $source \in sources$ **do**
        $visited\_nodes \leftarrow \{\}$
        $paths, visited\_nodes \leftarrow$ find_paths($clause, source, destination, visited\_nodes, sources\_set$)
        $all\_paths \leftarrow all\_paths \cup paths$
        $all\_visited\_nodes \leftarrow all\_visited\_nodes \cup visited\_nodes$
    end for

    if $compute\_reverse$ **and** $clause.body \nsubseteq all\_visited\_nodes$ **then**
        $destination \leftarrow source$
        for each $source \in sources$ **do**
            $sources\_set \leftarrow$ set($clause.head.terms$)$-\{destination\}$
            $backwards\_paths, all\_visited\_nodes \leftarrow$
                find_paths($clause, source, destination, all\_visited\_nodes, sources\_set$)
            for each $backwards\_path \in backwards\_paths$ **do**
                $path \leftarrow$ reverse($backwards\_path$)
                $all\_paths \leftarrow all\_paths \cup \{path\}$
            end for
        end for
    end if

    $disconnected\_literals \leftarrow$ get_disconnected_literals($clause, all\_visited\_nodes$)
        ▷ gets the ground literals that does not bellong to any path
return $all\_paths, disconnected\_literals$
end function

Algorithm 2 Find paths: algorithm to find the paths between the source and destination terms

Input: The clause ($clause$); the source term ($source$); the destination term ($destination$); the visited nodes ($visited\_nodes$); and all source terms ($sources\_set$);
Output: The paths from the source terms to destination; and the visited nodes

function find_paths($clause, source, destination, visited\_nodes, sources\_set$)
    $completed\_paths \leftarrow []$
    $partial\_paths \leftarrow [[source]]$
    $edge\_literals \leftarrow$ get_non_loop_literals($clause$)
    while $|partial\_paths| > 0$ **do**
        $size \leftarrow |partial\_paths|$
        for $i \leftarrow 0; i < size; i++$ **do**
            $path \leftarrow$ pop_left($partial\_paths$)
            if end($path$) $= destination$ **then**
                $completed\_paths \leftarrow completed\_paths + [path]$
                continue to next for iteration
            end if
            $not\_added\_path \leftarrow True$
            for each $literal \in$ get_literal_with_term($edge\_literals$, end($path$)) **do**
                $new\_end \leftarrow$ compute_end_term($path, literal$)
                if $new\_end \in path$ **or** $new\_end \in sources\_set$ **then**
                    continue to next for iteration ▷ path comes back to itself or to another input
                end if
                $new\_path \leftarrow path + [literal, new\_end]$
                if $new\_end = destination$ **then**
                    $completed\_paths \leftarrow completed\_paths + [new\_path]$
                else
                    $partial\_paths \leftarrow partial\_paths + [new\_path]$
                end if
                $visited\_nodes \leftarrow visited\_nodes \cup literal$
                $not\_added\_path \leftarrow False$
            end for
            if not_added_path **then**
                $completed\_paths \leftarrow completed\_paths + [path + [ANY, destination]]$
            end if
        end for
    end while
    $completed\_paths \leftarrow$ append_loops_to_paths($clause, completed\_paths$)
return $completed\_paths, visited\_nodes$
end function

Table 3: Cosine similarity example

$h1(X,Y) \leftarrow l1(X) \wedge l1(Y).$	$square\_sum(X) \leftarrow l1(X) \wedge l1(X).$
$\vdots$	$\vdots$
$h10(X,Y) \leftarrow l10(X) \wedge l10(Y).$	$square\_sum(X) \leftarrow l10(X) \wedge l10(X).$
	$norm(X) \leftarrow square\_sum(X) \wedge square\_root(X).$
$num(X,Y) \leftarrow h1(X,Y).$	$den(X,Y) \leftarrow norm(X) \wedge norm(Y).$
$\vdots$	$inv\_den(X,Y) \leftarrow inv\_den(X,Y) \wedge inverse(Y).$
$num(X,Y) \leftarrow h10(X,Y).$	$similarity(X,Y) \leftarrow num(X,Y) \wedge inv\_den(X,Y).$

A.2 COSINE EXAMPLE

Table 3 shows an example of a program to learn latent features to the entities in the knowledge base and compute the similarity of those entities, based on the cosine of the angle between their latent feature vectors.

In this program: $square\_root$ and $inverse$ are numeric functions to compute the element-wise square root and inverse value of the elements in a vector, respectively; the predicates $l1$ to $l10$ are predicates to be learned; and $similarity$ is the target predicate that computes the cosine similarity between the vectors of X and Y, where the components of the vectors are stored in the predicates $l1$ to $l10$, for each entity in the KB. The cosine similarity between two vectors is shown in Equation 1. From a set of examples of similar entities, we can train the network to learn the weights of the predicates $l1$ to $l10$, representing 10 latent features for each entity.

$$similarity(\mathbf{x}, \mathbf{y}) = \frac{< \mathbf{x}, \mathbf{y} >}{\|\mathbf{x}\| \, \|\mathbf{y}\|} \tag{1}$$

The rules $h_i(X,Y) \leftarrow \ldots$ calculates, for a given X, a vector with the value of the latent feature i of X times the latent feature i of Y, for each possible entity Y. This is due the syntax of the any/n predicate. The rules $num(X,Y) \leftarrow \ldots$ sums all the results from the rules $h_i(X,Y) \leftarrow \ldots$, computing the numerator part of the cosine similarity. In a similar way, the rule $den(X,Y) \leftarrow \ldots$ calculates the denominator, by multiplying the norm of the latent features of the input entity X for each possible output entity Y. Finally, $similarity(X,Y) \leftarrow \ldots$ computes the cosine similarity, by multiplying the numerator by the inverse of the denominator, given by the rule $inv\_den(X,Y) \leftarrow inv\_den(X,Y) \wedge inverse(Y)$.

B UWCSE THEORY

We also experimented with the UWCSE dataset (Richardson & Domingos, 2006), a well-known dataset in ILP. We applied a simplified version of the theory provided by Alchemy [2]. Since the original UWCSE dataset contains two predicates of arity 3, we have to convert them to binary predicates. The original predicates were $taughtby(course, person, season)$ and $ta(course, person, season)$. We transformed these predicates to binary by creating a single constant k, by concatenating the $course$ and $season$, since they appears together in the theory. We also added a new predicate $hascourse(k, course)$, to extract the course constant from the concatenated keys k.

Table 4 shows the theory used in the experiments with the UWCSE dataset. The underlined part represents the subset of the theory used by TensorLog, since it was unable to deal with rules containing free variables (variables that appears only once in the rule), either in the head or in the body. It also does not accept rules whose output variable has no inputs connected to it, for example $advisedby(X1, X2) \leftarrow student(X1) \wedge professor(X2)$. Using the full theory, NeuralLog achieved 0.9509 for the area under the ROC curve. Since TensorLog has to use a simpler theory, it was only able to achieve 0.7107 on the area under ROC curve. Using the same limited theory as TensorLog, NeuralLog achieved 0.7111 on the same metric.

[2] http://alchemy.cs.washington.edu/

Table 4: UWCSE Theory

$advisedby(X1, X2) \leftarrow taughtby2(K, X2) \wedge ta2(K, X1) \wedge hascourse(K, X3)$
$\wedge courselevel(X3, X5) \wedge tanh(X2) \wedge w(w1).$

$advisedby(X1, X2) \leftarrow taughtby2(K, X2) \wedge ta2(K, X1)$
$\wedge inphase(X1, X5) \wedge tanh(X2) \wedge w(w2).$

$advisedby(X1, X2) \leftarrow publication(X3, X1) \wedge publication(X3, X2) \wedge student(X1)$
$\wedge professor(X2) \wedge tanh(X2) \wedge w(w3).$

$advisedby(X1, X2) \leftarrow publication(X3, X1) \wedge publication(X3, X2) \wedge tanh(X2) \wedge w(w4).$

$advisedby(X1, X2) \leftarrow publication(X3, X1) \wedge publication(X3, X2) \wedge student(X1)$
$\wedge tanh(X2) \wedge w(w5).$

$advisedby(X1, X2) \leftarrow student(X1) \wedge tanh(X2) \wedge w(w6).$
$advisedby(X1, X2) \leftarrow professor(X2) \wedge hasposition(X2, X3) \wedge tanh(X2) \wedge w(w7).$
$advisedby(X1, X2) \leftarrow professor(X2) \wedge tanh(X2) \wedge w(w8).$
$advisedby(X1, X2) \leftarrow b(advisedby).$