Hypergraph Neural Networks with Logic Clauses

João Pedro Gandarela de Souza
COPPE
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
joaosouza@cos.ufrj.br

Gerson Zaverucha
COPPE
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
gerson@cos.ufrj.br

Artur S. d'Avila Garcez
Department of Computer Science
City University London
London, United Kingdom
a.garcez@city.ac.uk

***Abstract*—The analysis of structure in complex datasets has become essential to solving difficult Machine Learning problems. Relational aspects of data, capturing relationships between objects, play a crucial role in understanding the underlying data structure. While traditional graph algorithms have been widely used for binary relations, recent evidence suggests that hypergraphs can provide a more effective approach for modeling complex, non-binary relations. Hypergraph Neural Networks (HGNN) have been shown to offer a small improvement in performance when compared to Graph Neural Networks (GNN). In this paper, a new approach is proposed for inserting relational domain knowledge into HGNNs using a logic clause expressing non-binary relations. We evaluate the performance of this new hypergraph model, called Bottom-clause HGNN (BHGNN), in comparison with well-known approaches. Results show that BHGNN can achieve statistically significant improvement of performance, based on the Wilcoxon signed-ranks test, in comparison with HGNN and GNNs.**

***Index Terms*—Relational Learning, Neurosymbolic AI, Hypergraphs, Inductive Logic Programming, Graph Neural Networks.**

I. Introduction

In various fields of Machine Learning (ML), the analysis of relational knowledge has been important to help find solutions to intricate problems [1]. A relational database can be visualized as a graph, where the primary keys are represented as nodes, and connections between nodes represent foreign keys [2]. Then, Graph Neural Networks (GNNs) [3]–[5] can be utilized to manipulate this graph structure [6], [7].

GNN algorithms have been widely used to solve relational problems based on pairwise relationships. For example, a recent approach called BotGNN [7] generates a bipartite graph and uses a GNN to classify the graphs. However, pairwise relationships are limited in their ability to represent rich knowledge. Non-pairwise relationships are needed to represent complex data [8], [9].

Recently, studies have shown that hypergraphs, which allow hyper-edges to connect multiple vertices, can represent rich knowledge, offering an effective approach to modelling complex, non-pairwise relationships [10]. Additionally, there is a natural correspondence between relational databases and hypergraphs [11], and many applications already use hypergraphs because of the limitations of pairwise relationships [12]–[14].

João Souza thanks the Brazilian funding agency CAPES and Gerson Zaverucha thanks the Brazilian funding agency CNPq.

To handle hypergraphs, one can employ a Hypergraph Neural Network (HGNN) [15], [16] which is a variant of a neural network that extends the traditional Graph Neural Network to process data represented by a hypergraph. The main idea of a HGNN is to create an embedding from a hypergraph which is then used to learn a given task.

Unfortunately, creating a hypergraph from relational data relying solely on facts has its limitations. A fact in logic is a statement of the form $P(a)$ stating that property P holds true for object a, where P represents a predicate or property and a is an individual element or object within the domain of discourse. For example, let the predicate $P(X)$ denote that variable X is a prime number. A corresponding fact would be $P(5)$ indicating that the property of being a prime number holds true for the object 5. While utilizing facts provides a structured approach to representing relationships, it may not capture the dependencies and contextual intricacies present in real-world data. Additionally, the sheer volume of factual information required could become intractable when dealing with large datasets. The risk of information overload and the potential loss of relevant context poses a challenge to the effectiveness of the hypergraph representation and learning.

In this paper, instead of using facts only, a new method is proposed allowing the use of logic clauses, e.g. $P(a) \rightarrow Q(b)$, to construct the hypergraph. Furthermore, the clauses may contain any number of n-ary predicates, e.g. $R(a, b, c)$ with $n = 3$. By using logic clauses in the construction of the hypergraph we will infuse richer knowledge into the hypergraph neural network. Our hypothesis is that hypergraph neural networks can learn and represent intricate data relationships expressed by logic clauses.

The proposed approach is expected to facilitate the creation of elaborate hypergraphs. Changes made to the depth of a clause representing data relationships will be shown to impact the performance of the subsequent hypergraph learning. As an example, take the concept of a *lecture*, which can be defined as a ternary predicate $L(t, \mathbf{s}, r)$ with arguments representing a teacher (t), a list of students ($\mathbf{s}$), and a room (r). This ternary predicate defines a *lecture* as a teacher and a group of students in a room. Alternatively, using binary predicates only, one would require a predicate for each teacher, $L_t(\mathbf{s}, r)$ where L_t denotes lecture for teacher t, such that *lecture* is now defined as the list of binary predicates for all the teachers. This process is known as reification [17] in computational logic. When combined with learning, the representation choices matter and results may vary depending on the arity of predicates.

To explore the use of logic clauses with HGNNs, we introduce Bottom-clause Hypergraph Neural Networks (BHGNN). A bottom-clause is an artefact of Inductive Logic Programming (ILP) [18], a sub-field of symbolic ML. A clause is built from a single data point and it is used as the starting point (the bottom element of a lattice) for a search in the symbolic space of logic clauses. In [19], bottom clauses were first used with neural networks. A clause was created for each data point with the search being carried out in the vector space using a feedforward neural network. Here, a clause is created for each data point following [19], but the search uses a hypergraph neural networks.

Subsequently, leveraging the power of HGNNs, we generate an embedding derived from this hypergraph. This embedding forms the basis for classification and reasoning within the model. By integrating domain-specific knowledge into the generation of the hypergraph and subsequent embedding, we aim to allow the system to benefit from complex data relationships that may be known in advance or that may need to be imposed into the learning process. This follows the tradition of neurosymbolic systems whereby symbolic and subsymbolic components are combined [19]–[24].

Statistical analyses of the performance of BHGNN in comparison to traditional Graph Neural Networks (GNN) and Hypergraph Neural Networks (HGNN) indicate that BHGNN can consistently outperform both HGNN and GNN across multiple datasets. Notably, BHGNN achieves statistically significant improvements compared to HGNN and GNN.

The remainder of the paper is organized as follows: Section 2 formalizes the concepts from ILP and hypergraph neural networks used in the paper. Section 3 discusses related work. Section 4 describes the proposed method and algorithms. Section 5 contains the experimental results. Section 6 concludes the paper and discusses future work.

II. Background

A. Inductive Logic Programming

Inductive Logic Programming (ILP) [18], [25], [26] is a subfield of symbolic machine learning that aims to induce logical programs from examples. ILP learns logic theories consisting of Horn clauses in first-order logic from examples and background knowledge. It combines knowledge representation using the language of logic programming with inductive learning to search for logic rules that satisfy the examples.

In ILP, the input data is also represented in the form of logic programs containing positive and negative examples. The goal is to learn a logic clause that satisfies all positive examples but do not satisfy any of the negative examples. The background knowledge is crucial for the effectiveness of the search. It can take the form of facts such as *father(Lucas,Maria)* or rules such as *grandfather(X,Y) :- father(X,Z), father(Z,Y)*, read "X is a grandfather of Y *if* X is the father of Z *and* Z is the father of Y". This rule represents the clause $\forall X, Y, Z(father(X, Z) \wedge father(Z, Y) \rightarrow grandfather(X, Y))$. The conjunction of the literals in {*father(X,Z), father(Z,Y)*} is called the *body* of the rule, while *grandfather(X,Y)* is called the *head* of the rule.

Searching for a first-order logic clause in ILP is a combinatorial optimization problem. Because of that, the number of combinations of potential hypotheses forming a clause can become computationally intractable. To ameliorate this problem, ILP systems use *mode declarations* as language bias [27]. Modes provide a specification of the input and output of predicates that are allowed in the clauses. Mode declarations also provide information about the predicates that are required to solve a search problem. These declarations indicate whether a predicate can be used in the head (modeh) or body (modeb) of a rule, and they also specify the determination statements that declare the predicates that can be used to form a hypothesis. Additionally, the first argument of a mode declaration is the recall number, which establishes an upper limit on the number of instantiations of a predicate. Given a predicate $P(X)$, the predicate $P(5)$ is an instantiation of $P(X)$ by substituting the variable X with constant 5. As an example, consider a simple ILP scenario aiming to learn a rule for a binary classification task given a predicate *friend* with two arguments X and Y of type *person*, denoting that output Y is a friend of input X. The task is to learn a rule that predicts whether two persons are friends or not. A mode declaration for predicate *friend* could be: modeh(1,friend(+person, -person)), stating that with recall 1, the predicate that appears in the head of the clause is binary predicate $friend(X, Y)$, expected to have an object of type *person* as its first argument which is instantiated (denoted by the "+" symbol), and to have an object of type *person* as its second argument which is not instantiated (denoted by the "-" symbol). With recall number 1, the predicate will be instantiated only once, e.g. to denote $friend(Lucas, Y)$. An example of a rule to be learned based on the available data, in this case examples of friend and non-friend instances $\{friend(Mary, John), \neg friend(Mary, Lucas), ...\}$, might be transitivity of the relation: a friend of a friend is a friend [28].

The process of searching for a logic clause in ILP involves a saturation step to efficiently find all relationships implied by data instances given the background knowledge. This set of relationships is referred to as the most specific clause as it is obtained from the data. To narrow down the search space, the search is limited downward by the most specific clause, also known as the bottom clause, and upward by the most general clause. The head of the clause contains a single predicate representing the target concept, while the body starts empty. Consequently, the most general clause predicts any input as positive, e.g. $\forall X, Y friend(X, Y)$. As the search progresses, the clause is refined and instantiated by adding predicates from the Bottom Clause to the body based on observed positive and negative examples along with any background knowledge or constraints. The matching of modeh and modeb is structured to maintain variable chaining, ensuring that each variable used as an input in a body predicate is also an input variable in a head predicate or an output variable in another body predicate. In what follows, we will use the ILP process of constructing a bottom clause, which incorporates all the ground logical consequences from the background knowledge for a training example, to build a hypergraph neural network.

B. Graph Neural Networks

GNN is a type of neural network that is specifically designed to deal with graph-structured data [3]–[5]. Unlike traditional neural networks that operate on vectors or sequential data, GNNs seek to model relationships between entities in a graph.

The main idea in GNN algorithms is the message passing process, in which each node in the graph aggregates information from its neighboring nodes. A node is updated using information passed by its neighbors. Thus, the neighborhood function is crucial and in this paper it will be defined following the process for constructing a bottom clause, as discussed earlier. The typical steps involved in the use of a GNN are:

1) Initialization: The first step in a GNN is to initialize the node features. Each node in the graph is assigned an initial feature vector, which represents its initial state.
2) Message Passing: The core operation in a GNN is message passing, where information is exchanged between nodes in the graph. During message passing, each node aggregates information from its neighboring nodes.
3) Node Update: After aggregating information from neighboring nodes, each node updates its representation based on information gathered from the neighboring nodes.
4) Iteration: Steps 2 and 3 (message passing and node update) are performed repeatedly for a fixed number of iterations or until convergence. Each iteration allows the GNN to propagate information through the graph.
5) Readout/Pooling: After iteration, a readout or pooling operation is performed to aggregate information from all the nodes into a fixed-sized graph-level representation. This operation is expected to capture the overall characteristics or a summary of the graph.
6) Output: Finally, the graph-level representation obtained from the readout/pooling step is used to make predictions or perform a specific task. For instance, in node classification, the graph-level representation can be fed into a classifier to predict the label of each node. In graph classification, the graph-level representation is used to predict a label for an entire graph.

C. Hypergraph Neural Networks

While traditional GNNs operate on graphs consisting of nodes and edges, hypergraph neural networks (HGNNs) introduce hyper-edges that can connect multiple nodes. In hypergraphs [29], a hyper-edge is a higher-order structure that connects any number of nodes simultaneously. It permits the representation of n-ary predicate relationships compared to traditional graphs which encode binary predicates. HGNNs leverage this additional structure to capture complex and higher-order interactions in data.

The key idea behind HGNN is to generalize the concept of neighborhood aggregation used in GNNs. While node neighborhoods in graphs are just the immediate neighboring nodes, in a hypergraph the neighborhood of a node is defined by the nodes that share the same hyper-edges. This flexibility enables the propagation of information from a hyper-edge to all the nodes that it connects, incorporating higher-order dependencies which are especially relevant to model when multiple hyper-edges overlap partially, as discussed below.

One common approach in HGNN is to decompose hyper-edges into pairwise connections in order to adopt GNN methods. This decomposition allows the application of existing GNN architectures, such as Graph Convolutional Networks (GCNs), by treating the hypergraph as a bipartite graph. This method, known as hyper-edge decomposition, transforms the hypergraph into an auxiliary bipartite graph where hyper-edges are represented as edges between hyper-nodes and regular nodes. However, as our experiments indicate, this approach can lead to information loss and increased computational complexity due to the potential explosion in number of pairwise connections.

The algorithm for training HGNN is similar to that of GNN. The key difference is that with the data represented as a hypergraph, neighbors are not pair-wise anymore but are defined by the hyper-edges. The message passing follows the same idea as in GNNs, only now using the hyper-edges.

In [15], the GCN framework is extended to hypergraphs by introducing a hypergraph adjacency matrix and a hypergraph Laplacian matrix. The authors propose a hypergraph-based technique to adapt the traditional GCN operations to hypergraph data. [16] proposes a hypergraph convolutional neural network (HGCN) that generalizes the concept of graph convolutions to hypergraphs. The HGCN can employ a hypergraph attention mechanism to weigh the importance of hyper-edges during the convolution operation. In this paper, we choose to adopt a standard approach so as to simplify the comparative evaluation of results; we use the graph convolutional network of [30] and the HGNN defined in [16]. The adoption of other convolutional approaches is seen, however, as a natural extension of this work towards pursuing further increases in performance. The HGNN model used here is the following:

$$\mathbf{X}' = \mathbf{D}^{-1}\mathbf{H}\mathbf{W}\mathbf{B}^{-1}\mathbf{H}^{\top}\mathbf{X}\mathbf{\Theta}$$

where the incidence matrix $\mathbf{H}$, representing connections, is multiplied by the diagonal hyper-edge weight matrix $\mathbf{W}$.[1] The resulting product is then multiplied by $\mathbf{D}^{-1}$ and $\mathbf{B}^{-1}$, which represent the corresponding degree matrices. Finally, this multiplication is applied to the transpose of $\mathbf{H}$, the input matrix $\mathbf{X}$, and the parameter matrix $\mathbf{\Theta}$, yielding the updated matrix $\mathbf{X}'$ [16].

In summary, HGNNs extend the GNN framework to handle hypergraphs, allowing the modeling of higher-order dependencies among nodes. Moreover, by capturing complex interactions through hyper-edges, HGNN offers a richer representation of data. As the experimental results introduced in this

[1]https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.HypergraphConv.html

paper indicate, this richer representation can improve performance on various learning tasks that involve hypergraphs.

III. Related Work

BotGNN [7] is a method that creates a Bottom Graph from the Bottom Clause using bipartite graph structures. It works by representing predicates from the Bottom Clause on one side of the graph and their terms on the other side. This data can then be classified using a GNN. Our work is similar to the approach presented in [7], but instead of using graphs, we represent the logic programs as hypergraphs and process them using HGNN.

HetSAGE [6] is a GNN architecture designed to deal with heterogenous graphs. It applies a uniform sampling similar to that of GraphSAGE [31]. Additionally, it uses a node-centric sampling method to create sub-graphs from the entire graph. The method creates sub-graphs beginning from the target node that will be classified and retrieving n-hop neighbors. For example, with 1-hop it will create a sub-graph with the immediate neighbors, with 2-hops it will create a sub-graph with immediate neighbors and the neighbors of those neighbors. Then, instead of using the entire graph to classify the target node, it uses the sub-graph to classify the node.

As discussed before, ILP uses the power of First-order Logic (FOL) that enables the representation of complex relationships and quantification of variables. Nevertheless, the most efficient and effective machine learning algorithms of late have been based on neural networks which operate at a propositional level, not handling variable quantification. Propositionalization techniques have been used in ML to convert (finite) first-order logic into propositional logic. This process involves transforming a set of FOL clauses, typically represented by definite clauses, into propositional logic statements ready to be used by neural network learning algorithms.

A propositionalization technique called Bottom Clause Propositionalization (BCP) was introduced in [19] to allow the use of neural networks in the solution of ILP problems. BCP generates a Bottom Clause from each training example to generate features for each predicate from the Bottom Clause. Each predicate in the Bottom Clause is assigned a position in a vector. For example, a set with two clauses associated with target predicate *motherInLaw*: {motherInLaw(A, B) :− parent(A, C), wife(C, B); ¬motherInLaw(A, B) :− wife(A, C)}, will produce a 3-dimensional vector for parent(A, C), wife(C, B) and wife(A, C) in this order. The first clause maps to vector (1,1,0) with target output 1, the second clause maps to vector (0,0,1) with target output -1. A neural network classifier for the target predicate is then trained in the usual way. BCP will be used next with HGNNs.

IV. Hypergraph with Logic Clause Method

Neural-symbolic computation seeks to merge neural computation with symbolic reasoning. Our method involves integrating symbolic knowledge into hypergraphs extending the idea of Bottom Clause Propositionalization, and training a classifier from data and background knowledge using hypergraph nets.

We introduce the concept of a Bottom-HyperGraph, where from the predicates of the bottom clause, terms are mapped onto nodes and hyper-edges are used to connect the terms of any n-ary predicate. The process of creating the Bottom-HyperGraph follows a series of steps outlined below and detailed in the pseudo-code provided in the Appendix and the implementation available from GitHub:[2]

1) For each example in the training set:
    a) Generate a Bottom Clause;
    b) Delete the head of the Bottom Clause (the entire hypergraph will represent the target predicate);
    c) Generate a Bottom-HyperGraph from the body of the Bottom Clause:
        i) For each predicate of the form P(t_1,t_2,...,t_n), create a node in the hypergraph labelled t_i, $1 \leq i \leq n$, and create a hyper-edge connecting the terms t_i labelled P.
        ii) Create a feature vector for each node t_i encoding the arguments and data types of t_i;
        iii) Create a feature vector for each hyper-edge P encoding the terms in P.

The above process for creating a Bottom-HyperGraph produces a hypergraph in a format that a HGNN can handle. Each node has its respective feature vector, hyper-edges that contain the information of which nodes are in the relation, and the hyper-edge type encoded as a feature vector.

To exemplify the entire process, we use the trains dataset [32], a widely known dataset in the ILP community. The goal is to classify trains as being eastbound or otherwise based on characteristics of the cars of the trains such as car length (short or long), whether the car is open-top or not, the shape of the load, the type of content, number of wheels, etc. There are only ten train in the data, although the relational descriptions may become quite complicated. Examples of the logic clauses used in the solution of the problem are shown below and include predicates to state for example that train A has a car B which is closed-top with 3 circle-shaped loads and a car C with 1 triangle-shaped load. Crucially, the simple trains dataset includes ternary predicates such as $load(C, triangle, 1)$. The mode declarations are listed in the Appendix.

We work with four types of data in this mode of operation: '#int', '#shape', 'car' and 'train'. Additionally, there are eleven predicates, namely 'closed', 'double', 'eastbound', 'has_car', 'jagged', 'load', 'long', 'open_car', 'shape', 'short' and 'wheels'. While processing the data, we focus on the '#' symbol, which acts as a constant in the bottom clause. Seven constants are non-numeric, namely 'circle', 'ellipse', 'hexagon', 'nil', 'rectangle', 'triangle', and 'u_shape'.

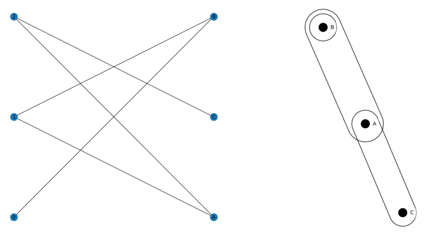
To induce a logical theory using ILP, positive and negative examples are necessary, as well as background knowledge and mode declarations. There are five positive examples of the form $eastbound(east_n)$, $1 \leq n \leq 5$, and five negative examples $eastbound(west_n)$, $6 \leq n \leq 10$.

[2]https://github.com/JoaoPedroD/Bottom-HyperGraph

With the background knowledge, mode declarations, and positive and negative examples, we generate a bottom clause. Below, we show the bottom clause for the first negative example, which is $eastbound(west6)$.

$eastbound(A) :-$
$closed(B), has\_car(A, B), has\_car(A, C),$
$load(B, circle, 3), load(C, triangle, 1), long(B),$
$open\_car(C), shape(B, rectangle), shape(C, rectangle),$
$short(C), wheels(B, 2), wheels(C, 2).$[3]

In the *trains* example, with hypergraphs not being restricted to pairwise relations between objects, 'load' is a hyper-edge connecting three nodes. Similarly, 'closed' can be represented by a unary relation, and therefore a hyper-edge for a single node. This changes the message passing: in a bipartite graph, it takes two hops for a predicate node to reach another predicate node or for a term node to reach another term node. By contrast, in a hypergraph, only one hop is required because each hyper-edge directly connects multiple nodes, allowing for more efficient communication and information transfer between nodes. This streamlined connectivity in hypergraphs reduces the complexity of traversing the graph compared to bipartite graphs. For instance, Figure 1a and Figure 1b represent the same set of relations, illustrating a shared underlying structure. Figure 1a displays a bipartite graph, while Figure 1b adopts a hypergraph representation. Despite the clear difference in representation, both figures are equivalent in terms of the relationships that they model. It can be seen that a single hop in the hypergraph may require multiple hops in the bipartite graph.



(a) bipartite graph (b) hypergraph

Fig. 1: Figures 1a and 1b depict equivalent sets of relations. While Figure 1a displays a bipartite graph, Figure 1b represents the same relationships using a hypergraph.

To process the hypergraph for HGNN training, feature vectors representing the hypergraph need to be created. Next, we illustrate this process using the *trains* example. We use the first eleven positions to encode the predicates: $[eastbound, closed, double, has\_car, jagged, load, long, open\_car, shape, short, wheels]$. We set the value to 1 for the predicates that hold true in the bottom clause, and 0 otherwise. Next, we encode the data types: $[\#int, \#shape, car, train]$, and reserve seven positions for constants: $[circle, ellipse, hexagon, nil, rectangle, triangle, u\_shape]$. Again, a 1 denotes true, and 0 denotes false. We allocate the last position to represent numeric values from the *load* and number of *wheels* predicates.

As an example, the feature vector for the predicate 'closed' would be a one-hot encoding with a 1 in the second position of a vector of size 23. Constant 2 is represented by feature vector [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2], with a 1 indicating the type $\#int$ and a 2 in the last position. For the variable 'A', the feature vector is again a one-hot encoding with a 1 in the fifteenth position, representing that 'A' is of type 'train'. For constant 'circle', the feature vector would be [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], indicating that 'circle' is of type '$\#shape$' and has a value of '$circle$'. The feature vectors for the hypergraph for $eastbound(west6)$ are shown in the Appendix.

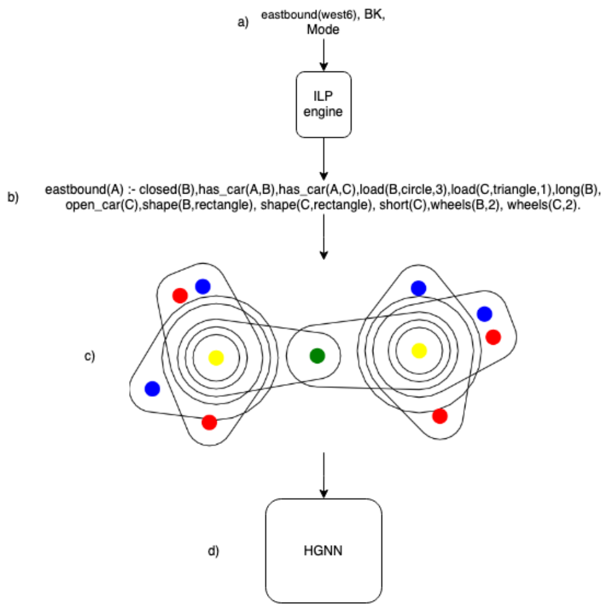

a)
eastbound(west6), BK, Mode
ILP engine
b)
eastbound(A) :- closed(B),has_car(A,B),has_car(A,C),load(B,circle,3),load(C,triangle,1),long(B), open_car(C),shape(B,rectangle), shape(C,rectangle), short(C),wheels(B,2), wheels(C,2).
c)
d)
HGNN

Fig. 2: Bottom Clause Hypergraph Neural Network. Background knowledge and mode declarations are shown here for the example $eastbound(east6)$ of the *trains* classification problem) (a). A bottom clause is generated for $eastbound(east6)$ (b). The corresponding hypergraph is created, disregarding the head of the bottom clause, with color green denoting type *train*, yellow denoting type *car*, red denoting *shape*, and blue denoting *int* (c). The hypergraph is converted into feature vectors for training the HGNN (d).

The algorithm in the Appendix is used to create the hypergraph from the Bottom Clause. We expect that domain knowledge will be incorporated into the HGNN using the Bottom Clause to improve performance. Next, we report the experimental results indicating that this is indeed the case.

To summarize, the process begins with the bottom clause of $eastbound(east6)$, as illustrated in Figure 2. This clause is then transformed into a hypergraph, creating the Bottom-

[3]This clause states that train A is eastbound if it has two cars, one closed (B) and one open (C). The closed car (B) carries a load with 3 circles, is long, has a rectangular shape, and two wheels. The open car (C) carries a load with one triangle, is short, has a rectangular shape, and two wheels.

HyperGraph. Finally, we utilize the Bottom-HyperGraph to train a HGNN, as detailed below.

V. Experimental Results

To evaluate the performance of the proposed method using a hypergraph neural network, we conducted a ten-fold cross-validation on multiple datasets. We implement the HGNN with Pythorch[4]. We conduct an analysis of the proposed approach (BGHNN) in direct comparison with its most closely related approaches, HGNN and GNN. We then evaluate BHGNN against BotGNN, CILP++, and other related work. We processed graphs using GNN with Convolutional ARMA Filters [30]. We optimized both HGNN and GNN using Adam [33]. The architectures of both models have three layers and were trained for 500 epochs with early stopping set to 50 and a learning rate of 0.0001. Additionally, we report the results obtained by Aleph [34], a purely-symbolic ILP system.

The datasets and background knowledge are written in Prolog and described in Aleph, including the mode declarations to create the bottom clause. We investigate diverse datasets such as Mutagenesis [35], Carcinogenesis [36] and four Alzheimer's datasets [37] - Choline, Scopolamine, Toxic, Amine - each with different characteristics and features. The Aleph system [34] was used to generate the bottom clauses with variable depth set to five.

The experiments were performed as follows:

1. Data preparation: we pre-processed each dataset to ensure a smooth transition from Prolog to Python libraries.

2. Cross-validation iterations: for each dataset, we produced 10 variations for cross-validation. In each, 1/10 of the data was held out as the validation set, while the remaining data was used for training.

3. Model training and evaluation: we created a Bottom-HyperGraph and trained a HGNN for each fold, with the graph embedding as initialization, recording training and validation set classification performance (accuracy) for each fold.

4. Result aggregation: after completing the ten-fold cross-validation for each dataset, we aggregated the results by calculating the average and standard deviation of accuracy across all folds for each dataset.

The results of the experiments are presented in Table II, displaying the average accuracy and standard deviation for each dataset. After applying the Wilcoxon signed-rank test [38] to compare the performance of BHGNN with HGNN and GNN on the same folds, we found that BHGNN outperforms both. Therefore, we found evidence to reject the null hypothesis, indicating significant differences in their performance.

Next, we compared our method (BHGNN) with BotGNN. We found that BHGNN exhibits in general a lower standard deviation than BotGNN and a better average accuracy in four out of seven data sets. However, applying the Wilcoxon statistical test revealed no substantial differences between the two models.

[4]https://pytorch-geometric.readthedocs.io/en/latest/

TABLE I: Average accuracy results of the proposed approach (BHGNN) in comparison with directly related approaches HGNN and GNN on seven data sets. BHGNN outperforms HGNN and GNN on all data sets with statistically significant results; HGNN does not use bottom clauses in the hypergraph and GNN uses a bipartite graph.

	BHGNN	**HGNN**	**GNN**
Muta42	$\mathbf{75 \pm 13.4}$	70 ± 9.3	68 ± 13.8
Muta188	$\mathbf{90 \pm 8.9}$	83 ± 6.1	79 ± 6.3
Carcinogenesis	$\mathbf{64 \pm 7.9}$	54 ± 6.3	54 ± 5.2
Scopolamine	$\mathbf{54 \pm 9.9}$	51 ± 6.6	49 ± 5.6
Amine	$\mathbf{53 \pm 8.2}$	52 ± 6.6	49 ± 4.4
Toxic	$\mathbf{55 \pm 8.6}$	53 ± 5.9	54 ± 4.9
Choline	$\mathbf{54 \pm 7.9}$	52 ± 5.6	50 ± 4.8

TABLE II: Average accuracy results of the proposed approach (BHGNN) in comparison with related work BotGNN, HetSAGE, CILP++ and Aleph. Among the graph-based approaches (BHGNN, BotGNN, HetSAGE), BHGNN achieves the best results in three out of seven data sets. In two cases, the use of a vector instead of a graph representation (CILP++) produces the best results. In three cases, a purely symbolic approach wins (Aleph). These data sets have been all studied extensively in ILP with the use of Aleph optimizations. The results indicate the need for further investigation and optimization of the graph-based approaches.

	BHGNN	**BotGNN**	**HetSAGE**	**CILP++**	**Aleph**
Muta42	75 ± 13.4	73 ± 16.4	70 ± 14.6	73 ± 18.3	$\mathbf{76 \pm 14.5}$
Muta188	$\mathbf{90 \pm 8.9}$	88 ± 9.9	86 ± 10.9	87 ± 7.8	85 ± 9.1
Carcinogenesis	$\mathbf{64 \pm 7.9}$	62 ± 8.9	62 ± 11.9	58 ± 8.9	61 ± 8.7
Scopolamine	54 ± 9.9	53 ± 10.4	52 ± 12.9	52 ± 5.5	$\mathbf{60 \pm 4.7}$
Amine	53 ± 8.2	55 ± 8.7	52 ± 8.6	$\mathbf{68 \pm 6.7}$	62 ± 7.9
Toxic	55 ± 8.6	62 ± 8.3	54 ± 9.6	$\mathbf{72 \pm 6.1}$	65 ± 8.2
Choline	54 ± 7.9	58 ± 9.2	53 ± 10.2	53 ± 4.3	$\mathbf{59 \pm 7.7}$

Finally, the results obtained by Aleph and CILP++ were competitive, showcasing their robustness and efficacy in addressing relational learning cases. The differences between BHGNN and Aleph and CILP++ were also found to be not statistically significant.

VI. Conclusion and Future Work

The main objective of this paper was to investigate different forms of graph representation and their impact on the effectiveness of graph-based learning. In particular, graph neural networks as a richer form of representation than standard neural networks are expected to handle relational data well. We introduced a method allowing graph neural networks to benefit from a symbolic relational learning setting via the use of hypergraphs. We evaluated learning results on datasets that are not restricted to binary relations, comparing performance with hypergraph neural networks, graph neural networks and other neurosymbolic and purely-symbolic approaches for relational learning. The paper defined a systematic method for integrating domain expertise into hypergraph networks using the concept of saturating examples to create a most-specific clause as an initial embedding for training. The approach is novel in that it is the first to instil relational knowledge of arbitrary arity into hypergraph networks. Experimental results have indicated the promise of the approach. The results obtained warrant further evaluations on extensive data sets to consider also variations and the latest optimizations of the graph learning methods and algorithms.

While symbolic Machine Learning induces a logic theory as a result of learning, graph neural networks do not. A logic theory is in principle interpretable and explainable, capable of providing a justification for the answers obtained in the form of a logic proof. Therefore, explainability of hypergraph networks is a natural area for further research. A hypergraph network to which prior knowledge has been added should be easier to interpret via querying than one that starts from a random initialization. The goal will be to post-process the hypergraph learned to revert to logic form. We plan to investigate variations of HGNN such as [39], expand the graph optimization efforts which were not the focus of this paper, and evaluate explainability of the decision-making process. Given the results obtained here, one should also consider adapting approaches such as [40] to work with hypergraphs.

References

[1] A. K. Wong, P.-Y. Zhou, and Z. A. Butt, "Pattern discovery and disentanglement on relational datasets," *Scientific Reports*, vol. 11, no. 1, p. 5688, 2021.

[2] M. Fey, W. Hu, K. Huang, J. E. Lenssen, R. Ranjan, J. Robinson, R. Ying, J. You, and J. Leskovec, "Relational deep learning: Graph representation learning on relational databases," *arXiv preprint arXiv:2312.04615*, 2023.

[3] W. L. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[4] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2. IEEE, 2005, pp. 729–734.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[6] V. Jankovics, M. G. Ortiz, and E. Alonso, "Hetsage: Heterogenous graph neural network for relational learning (student abstract)," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 18, 2021, pp. 15 803–15 804.

[7] T. Dash, A. Srinivasan, and A. Baskar, "Inclusion of domain-knowledge into gnns using mode-directed inverse entailment," *Machine Learning*, pp. 1–49, 2022.

[8] P. Bonacich, A. C. Holdren, and M. Johnston, "Hyper-edges and multidimensional centrality," *Social networks*, vol. 26, no. 3, pp. 189–203, 2004.

[9] S. Klamt, U.-U. Haus, and F. Theis, "Hypergraphs and cellular networks," *PLoS computational biology*, vol. 5, no. 5, p. e1000385, 2009.

[10] M. M. Wolf, A. M. Klinvex, and D. M. Dunlavy, "Advantages to modeling relational data using hypergraphs versus graphs," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.

[11] R. Fagin, "Degrees of acyclicity for hypergraphs and relational database schemes," *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 514–550, 1983.

[12] H. Shi, Y. Zhang, Z. Zhang, N. Ma, X. Zhao, Y. Gao, and J. Sun, "Hypergraph-induced convolutional networks for visual classification," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 10, pp. 2963–2972, 2019.

[13] S. Feng, E. Heath, B. Jefferson, C. Joslyn, H. Kvinge, H. D. Mitchell, B. Praggastis, A. J. Eisfeld, A. C. Sims, L. B. Thackray *et al.*, "Hypergraph models of biological networks to identify genes critical to pathogenic viral response," *BMC bioinformatics*, vol. 22, no. 1, pp. 1–21, 2021.

[14] M. Contisciani, F. Battiston, and C. De Bacco, "Inference of hyperedges and overlapping communities in hypergraphs," *Nature communications*, vol. 13, no. 1, p. 7229, 2022.

[15] N. Yadati, M. Nimishakavi, P. Yadav, V. Nitin, A. Louis, and P. Talukdar, "Hypergcn: A new method for training graph convolutional networks on hypergraphs," *Advances in neural information processing systems*, vol. 32, 2019.

[16] S. Bai, F. Zhang, and P. H. Torr, "Hypergraph convolution and hypergraph attention," *Pattern Recognition*, vol. 110, p. 107637, 2021.

[17] N. Grewe, "A generic reification strategy for n-ary relations in dl," in *OBML 2010 Workshop Proceedings*, 2010.

[18] S.-H. Nienhuys-Cheng and R. de Wolf, *What is inductive logic programming?* Springer, 1997.

[19] M. V. França, G. Zaverucha, and A. S. d'Avila Garcez, "Fast relational learning using bottom clause propositionalization with artificial neural networks," *Machine learning*, vol. 94, pp. 81–104, 2014.

[20] A. S. Avila Garcez and G. Zaverucha, "The connectionist inductive learning and logic programming system," *Applied Intelligence*, vol. 11, pp. 59–77, 1999.

[21] A. d'Avila Garcez and L. C. Lamb, "Neurosymbolic AI: the 3rd wave," *Artif. Intell. Rev.*, vol. 56, no. 11, pp. 12 387–12 406, 2023.

[22] L. C. Lamb, A. S. d'Avila Garcez, M. Gori, M. O. R. Prates, P. H. C. Avelar, and M. Y. Vardi, "Graph neural networks meet neural-symbolic computing: A survey and perspective," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 4877–4884.

[23] T. R. Besold, A. S. d'Avila Garcez, S. Bader, H. Bowman, P. M. Domingos, P. Hitzler, K. Kühnberger, L. C. Lamb, P. M. V. Lima, L. de Penning, G. Pinkas, H. Poon, and G. Zaverucha, "Neural-symbolic learning and reasoning: A survey and interpretation," in *Neuro-Symbolic Artificial Intelligence: The State of the Art*, ser. Frontiers in Artificial Intelligence and Applications, P. Hitzler and M. K. Sarker, Eds. IOS Press, 2021, vol. 342, pp. 1–51.

[24] A. S. d'Avila Garcez, K. Broda, and D. M. Gabbay, *Neural-symbolic learning systems - foundations and applications*, ser. Perspectives in neural computing. Springer, 2002.

[25] S. Muggleton, "Inductive logic programming," *New generation computing*, vol. 8, pp. 295–318, 1991.

[26] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19, pp. 629–679, 1994.

[27] S. Muggleton, "Inverse entailment and progol," *New generation computing*, vol. 13, pp. 245–286, 1995.

[28] A. L. d. C. L. Duboc, "Utilizando a cláusula mais específica e declaração de modos na revisão de teorias de primeira-ordem a partir de exemplos," Master's thesis, UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2008.

[29] A. Bretto, "Hypergraph theory," *An introduction. Mathematical Engineering. Cham: Springer*, vol. 1, 2013.

[30] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, "Graph neural networks with convolutional arma filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 1–1, 2021.

[31] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[32] D. Michie, S. Muggleton, D. Page, and A. Srinivasan, "To the international computing community: A new east-west challenge," *Distributed email document available from http://www. doc. ic. ac. uk/˜ shm/Papers/ml-chall. pdf*, 1994.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6628106

[34] A. Srinivasan, "The aleph manual," 2001.

[35] A. Srinivasan, S. Muggleton, R. D. King, and M. J. Sternberg, "Mutagenesis: Ilp experiments in a non-determinate biological domain," in *Proceedings of the 4th international workshop on inductive logic programming*, vol. 237. Citeseer, 1994, pp. 217–232.

[36] A. Srinivasan, R. D. King, S. H. Muggleton, and M. J. Sternberg, "Carcinogenesis predictions using ilp," in *International Conference on Inductive Logic Programming*. Springer, 1997, pp. 273–287.

[37] R. D. King, M. J. Sternberg, and A. Srinivasan, "Relating chemical activity to structure: an examination of ilp successes," *New Generation Computing*, vol. 13, pp. 411–433, 1995.

[38] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine learning research*, vol. 7, pp. 1–30, 2006.

[39] Y. Gao, Y. Feng, S. Ji, and R. Ji, "Hgnn+: General hypergraph neural networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 3, pp. 3181–3199, 2023.

[40] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

Appendix A
Trains dataset

Mode Declarations:

$: -modeh(1, eastbound(+train)).$
$: -modeb(1, short(+car)).$
$: -modeb(1, closed(+car)).$
$: -modeb(1, long(+car)).$
$: -modeb(1, open\_car(+car)).$
$: -modeb(1, double(+car)).$
$: -modeb(1, jagged(+car)).$
$: -modeb(1, shape(+car, \#shape)).$
$: -modeb(1, load(+car, \#shape, \#int)).$
$: -modeb(1, wheels(+car, \#int)).$
$: -modeb(*, has\_car(+train, -car)).$
$: -determination(eastbound/1, short/1).$
$: -determination(eastbound/1, closed/1).$
$: -determination(eastbound/1, long/1).$
$: -determination(eastbound/1, open\_car/1).$
$: -determination(eastbound/1, double/1).$
$: -determination(eastbound/1, jagged/1).$
$: -determination(eastbound/1, shape/2).$
$: -determination(eastbound/1, wheels/2).$
$: -determination(eastbound/1, has\_car/2).$
$: -determination(eastbound/1, load/3).$

Feature vectors:
$closed : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$has\_car : [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$load : [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$long : [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$open\_car : [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$shape : [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$short : [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$wheels : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$A : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0.]$
$B : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$C : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$circle : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0.]$
$triangle : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0.]$
$rectangle : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0.]$
$1 : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.]$
$2 : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2.]$
$3 : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3.]$

Algorithm for creating the Bottom hypergraph:

```
**Require:** generate_hypergraph($bottom\_clause$: a bottom clause from a single example)
  Remove head of the bottom clause
  $nodes \leftarrow \emptyset$, $hyperedges \leftarrow \emptyset$
  **for** each $atom$ in $bottom\_clause$ **do**
    $hyperedge \leftarrow \emptyset$, $hyperedge\_type \leftarrow \emptyset$
    extract terms into $terms$ and extract predicate into $predicate$
    **for** each $t$ in $terms$ **do**
      **if** $t$ not in $nodes$ **then**
        add $t$ into $nodes$
      **end if**
      $t\_index \leftarrow$ index of $t$ in $nodes$ # get the index of the node $t$ in the $nodes$
      add $t\_index$ to $hyperedge$
    **end for**
    Create node of type $predicate$ and add to $hyperedge$
    Add $hyperedge$ to $hyperedges$ and add $predicate$ to hyperedges_type
  **end for**
  **return** $nodes$, $hyperedges$, $hyperedge\_type$
**Require:** $bottom\_clauses$: Bottom clause for all examples; $preds$: list of all predicates; $modes$: modes; $consts$: constants that appear in the bottom clause; $node\_types$: type of each term of each predicate
  **for** each $bottom\_clause$ in $bottom\_clause$ **do**
    $hypergraph \leftarrow generate\_hypergraph(bottom\_clause)$
    **for** each $node$ in $hypergraph$ **do**
      $feat \leftarrow$ array of zeros of length $preds$
      **if** $node$ type is predicate **then**
        $feat \leftarrow oneHot(predicate, preds)$
      **end if**
      $feat$ concatenate into $feats$
      $feat$ = array of zeros of length $node\_types$
      **if** $node$ type is $node\_type$ **then**
        $feat = oneHot(node\_type, node\_types)$
      **end if**
      $feat$ concatenate into $feats$
      $feat$ = array of zeros of length $consts$
      **if** $node$ type is $constant$ **then**
        $feat = oneHot(constant, consts)$
      **end if**
      $feat$ concatenate into $feats$
      $feat$ = array of zero of length 1
      **if** $node$ type is $numeric$ **then**
        $feat = number$
      **end if**
      $feat$ concatenate into $feats$
      $hypergraph[node] = feats$
    **end for**
    add $hypergraph$ in $hypergraphs$
  **end for**
  **return** $hypergraphs$
```