HYPERGRAPH NEURAL NETWORKS WITH LOGIC CLAUSES

João Pedro Gandarela de Souza

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Gerson Zaverucha

Rio de Janeiro
Outubro de 2024

HYPERGRAPH NEURAL NETWORKS WITH LOGIC CLAUSES

João Pedro Gandarela de Souza

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientador: Gerson Zaverucha

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2024

Gandarela de Souza, João Pedro

Hypergraph Neural Networks with Logic Clauses/João Pedro Gandarela de Souza. – Rio de Janeiro: UFRJ/COPPE, 2024.

XII, 82 p.: il.; $29,7$cm.

Orientador: Gerson Zaverucha

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2024.

Referências Bibliográficas: p. 61 – 67.

1. Relational Learning. 2. Neurosymbolic AI. 3. Hypergraphs. 4. Inductive Logic Programming. 5. Graph Neural Networks. I. Zaverucha, Gerson. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A minha família, principalmente
meus pais e avós.

Agradecimentos

Gostaria de expressar minha gratidão a todos que, de alguma forma, contribuíram para a realização deste trabalho.

Em primeiro lugar, agradeço ao meu orientador, Professor Gerson Zaverucha, por sua orientação, paciência, parceria e incentivo durante todo o processo. Sua experiência foi fundamental para o desenvolvimento desta dissertação.

Aos meus pais e familiares, meu imenso agradecimento pelo apoio em todos os momentos. Sem eles, esta jornada não teria sido possível.

Agradeço aos amigos e colegas de curso por todas as conversas e por terem compartilhado comigo esta jornada acadêmica.

Agradeço também à UFRJ e aos professores que forneceram os recursos necessários para que este trabalho fosse realizado.

Sou grato à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro, que foi essencial para a dedicação e realização desta pesquisa.

Gostaria também de agradecer à banca pela presença e pelo papel na avaliação desta dissertação.

Por fim, sou grato a todos aqueles que, direta ou indiretamente, colaboraram para a conclusão desta etapa tão importante em minha vida.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

HYPERGRAPH NEURAL NETWORKS WITH LOGIC CLAUSES

João Pedro Gandarela de Souza

Outubro/2024

Orientador: Gerson Zaverucha

Programa: Engenharia de Sistemas e Computação

A análise da estrutura em conjuntos de dados complexos tornou-se essencial para resolver problemas difíceis de aprendizagem automática. Os aspectos relacionais dos dados, que captam as relações entre objectos, desempenham um papel crucial na compreensão da estrutura subjacente dos dados. Embora os algoritmos tradicionais de grafos tenham sido amplamente utilizados para relações binárias, provas recentes sugerem que os hipergrafos podem proporcionar uma abordagem mais eficaz para modelar relações complexas e não binárias. As redes neurais de hipergrafos (HGNN) demonstraram oferecer uma pequena melhoria no desempenho quando comparadas com as redes neurais de grafos (GNN). Neste trabalho, é proposta uma nova abordagem para inserir conhecimento de domínio relacional em HGNNs usando uma cláusula lógica que expressa relações não binárias. Avaliamos o desempenho deste novo modelo hipergrafos, designado por BHGNN (Bottom-clause HGNN), em comparação com abordagens bem conhecidas. Os resultados mostram que o BHGNN pode alcançar uma melhoria estatisticamente significativa do desempenho, com base no teste Wilcoxon signed-ranks, em comparação com HGNN e GNNs.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

HYPERGRAPH NEURAL NETWORKS WITH LOGIC CLAUSES

João Pedro Gandarela de Souza

October/2024

Advisor: Gerson Zaverucha

Department: Systems Engineering and Computer Science

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 work, 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.

Contents

List of Figures

List of Tables

Chapter 1

Introdution

1.1 Introduction

In several areas of machine learning (ML) and data science, the analysis of relational knowledge contributes to problem solving [3]. Relational data can be represented in different ways that reflect different relationships. For example, a relational database can be interpreted as a graph where primary keys serve as nodes and links between them are represented by foreign keys [4, 5]. In this context, Graph Neural Networks (GNNs) [6–8] facilitate the manipulation of this graph structure [2, 9].

GNNs model relational structures in domains such as social networks and biological systems. They effectively capture interactions between entities. For example, BotGNN [9] constructs a bipartite graph from relational data and uses a GNN for classification. However, pairwise relations do not always capture knowledge comprehensively, leading to the need for non-binary relations to address complex data structures [10, 11].

Research suggests that hypergraphs, which allow hyperedges to connect multiple vertices, provide a framework for modelling non-paired relations [12]. Additionally, a relational database can also correspond to hypergraphs [13], with existing applications using hypergraphs because of the limitations of pairwise relationships [14–16]. For example, [15] models systems such as protein complexes and metabolic reactions using hypergraphs, while [17] applies hypergraphs to social recommendation.

GNNs work with graphs, while hypergraph neural networks (HGNNs) [18, 19] extend this approach to hypergraphs. An HGNN generates embeddings from hypergraphs, facilitating several tasks.

Creating a hypergraph from relational data based solely on facts has limitations. A fact in logic is a statement such as $P(a)$, where P is a predicate and a is an object. For example, $P(X)$ can represent the property of being a prime number, while $P(5)$ indicates that 5 is a prime number. Facts provide structure, but they may not fully capture the complexity of real-world data. Relying on facts alone can lead to information overload, which reduces the effectiveness of hypergraph representations.

To improve data representation, this work explores the use of logical clauses to construct hypergraphs, moving beyond traditional fact-based approaches. By incorporating n-ary predicates within logical clauses, the method captures relationships and dependencies within the data. This approach addresses the limitations of using facts alone and allows for a more detailed understanding of the relational structure.

In addition, the integration of logical clauses aims to evaluate the impact on hypergraph neural networks by providing a more informative embedding if it leads to accurate classifications and inferences. It is hypothesised that the use of these logical constructs can improve the ability of hypergraph neural networks to model relationships in relational datasets.

To address this limitation, a method is proposed that uses logical clauses such as $P(a) \rightarrow Q(b)$ to construct the hypergraph. These clauses can contain n-ary predicates, such as $R(a, b, c)$. This method aims to enrich the hypergraph neural network with additional knowledge. The hypothesis is that hypergraph neural networks can effectively represent data relationships defined by logical clauses.

For example, the concept of a *lecture* can be defined as a ternary predicate $L(t, s, r)$, with arguments for a teacher (t), students (s), and a room (r). This ternary predicate identifies a *lecture* involving a teacher and students in a room. Alternatively, using binary predicates would require a separate predicate for each teacher, $L_t(s, r)$, where L_t represents a lecture for teacher t. This conversion of a predicate into a term is known as reification [20]. The choice of representation affects the results, as they can vary with the arity of the predicate.

There are many combinations of clauses. To explore these, a bottom-clause and a mode-declaration are used. A bottom-clause, derived from Inductive Logic Programming (ILP) [21], is constructed from a single data point and serves as the basis for searching the space of logical clauses. The introduction of Bottom-clause Hypergraph Neural Networks (BHGNN) [22] allows the exploration of logical clauses with HGNNs. In [23], bottom clauses were first used with neural networks, creating a clause for each data point, next using a feed-forward neural network to create a classifier. Here a clause is created for each data point, following [23], but the clause is represented as a hypergraph, therefore we use a hypergraph neural network.

The proposed method facilitates hypergraph construction, with the depth of clauses representing data relationships influencing the subsequent hypergraph learning.

HGNNs generate embeddings from the hypergraph, which are used for classification and inference within the model.

By integrating domain-specific knowledge into hypergraph generation and embedding, this approach enables the system to exploit complex data relationships and infuse symbolic knowledge into HGNNs. This approach is consistent with neurosymbolic systems, which combine symbolic and sub-symbolic components [23–29].

The thesis is structured as follows: Chapter 2 formalises the concepts from ILP, Neural Networks, Graph Neural Networks and Hypergraph Neural Networks used in the thesis; Chapter 3 describes the proposed methodology and algorithms; Chapter 4 discusses related work; Chapter 5 presents the experimental results; Chapter 6 concludes the thesis and discusses future directions.

Chapter 2

Background

Understanding the structures and models used to represent data is central to computational methods. Five concepts are introduced in this chapter: First-Order Logic (FOL), Inductive Logic Programming (ILP), Artificial Neural Networks (ANNs), Graphs and Hypergraphs, along with their neural network counterparts, Graph Neural Networks (GNNs) and Hypergraph Neural Networks (HGNNs).

Artificial Neural Networks (ANNs) are a core component of machine learning and artificial intelligence. These models, inspired by the structure of the human brain, consist of interconnected nodes or neurons arranged in layers. We also describe the basic components of ANNs, including weights, biases and activation functions, and discuss the forward and backpropagation algorithms that enable these networks to learn from data.

First-Order Logic (FOL) is a formal system for expressing statements about objects and their relationships. FOL extends propositional logic to include quantifiers and predicates, allowing for more expressive representations of knowledge. It is used in artificial intelligence for knowledge representation, automated reasoning and natural language understanding.

ILP combines machine learning with logic programming. ILP uses FOL to represent hypotheses and background knowledge, and allows logical rules to be learned from examples. This approach is useful for tasks that require symbolic representations of knowledge, such as bioinformatics, natural language processing, and software engineering.

Graphs and hypergraphs represent relationships in data. A graph is a set of vertices (or nodes) connected by edges, which can be directed or undirected, weighted or unweighted. Graphs are used in areas such as social networks, communication systems and biological networks to model pairwise relationships. Hypergraphs extend this idea by allowing edges, called hyperedges, to connect multiple vertices simultaneously, capturing more complex interactions.

Graph Neural Networks (GNNs) and Hypergraph Neural Networks (HGNNs) are neural networks designed to work with graph and hypergraph data. GNNs use the structure of graphs to propagate information between nodes, allowing the modelling of relationships and processes within the graph. Similarly, HGNNs extend this to hypergraphs, allowing the modelling of higher order relationships and interactions.

This chapter covers the theoretical underpinnings of these concepts. By the end, you will have a clear understanding of the neural networks, graphs, hypergraphs, FOL and ILP used in this thesis.

2.1 Propositional and First Order Logic

2.1.1 Propositional Logic

Propositional logic [30–32], often called Boolean logic, is the most basic form of logic. It works with propositions, which are declarative statements that can be either true or false. The simplicity of propositional logic derives from its use of binary truth values, without regard to the internal structure of propositions.

The main elements of propositional logic are propositions and logical connectors. Propositions are statements such as "It is raining" or "The light is on". Logical connectors such as AND ($\wedge$), OR ($\vee$), NOT ($\neg$), IMPLIES ($\rightarrow$) and BICONDITIONAL ($\leftrightarrow$) are used to combine these statements into more complex expressions. For example, "It is raining AND it is cold" combines two statements using the AND connective.

Truth tables are used to determine the truth value of compound statements. These tables list the possible truth values of individual propositions and calculate the resulting truth value of the compound statement. This structured approach makes propositional logic a useful tool for reasoning about simple statements and their combinations. However, propositional logic is limited in its ability to represent statements that involve objects or their relationships.

To illustrate propositional logic, consider a scenario involving weather conditions and an outdoor event. Propositional logic allows us to combine these statements using logical connectors to form more complex expressions.

Let us define the following propositions:

- P: It is raining.
- Q: The event is outside.
- R: The event is cancelled.

Each statement can be either true or false. Logical connectors can now be used to combine these statements and analyse their relationships.

- **AND (Conjunction)**
  - The conjunction of two propositions P and Q is represented as $P \wedge Q$ and is true if and only if both P and Q are true.
  - Example: "It is raining and the event is held outdoors" can be represented as $P \wedge Q$.
- **OR (Disjunction)**
  - The disjunction of two propositions P and Q is represented as $P \vee Q$ and is true if at least one of P or Q is true.
  - Example: "It is raining or the event is held outdoors" can be represented as $P \vee Q$.
- **NOT (Negation)**
  - The negation of a proposition P is represented as $\neg P$ and is true if P is false.
  - Example: "It is not raining" can be represented as $\neg P$.
- **IMPLIES (Implication)**
  - The implication $P \to R$ is true if either P is false or R is true (i.e., P implies R).
  - Example: "If it is raining, then the event will be canceled" can be represented as $P \to R$.
- **BICONDITIONAL (Equivalence)**
  - The biconditional $P \leftrightarrow Q$ is true if P and Q are both true or both false.
  - Example: "The event will be held outdoors if and only if it is not raining" can be represented as $Q \leftrightarrow \neg P$.

Let's consider a more complex statement and construct its truth table. We will examine the statement: "If it is raining and the event is held outdoors, then the event will be canceled" ($(P \wedge Q) \to R$).

Here is the truth table for this expression:

P	Q	R	$P \land Q$	$(P \land Q) \to R$
T	T	T	T	T
T	T	F	T	F
T	F	T	F	T
T	F	F	F	T
F	T	T	F	T
F	T	F	F	T
F	F	T	F	T
F	F	F	F	T

In the truth table:

- The columns for P, Q and R represent the possible truth values of each proposition.
- The column for $P \land Q$ shows the truth value of the conjunction of P and Q.
- The last column shows the truth value of the whole expression $(P \land Q) \to R$.

The truth table shows how the truth value of the compound proposition $(P \land Q) \to R$ depends on the truth values of P, Q and R. In particular, $(P \land Q) \to R$ is false only if $P \land Q$ is true and R is false. This corresponds to the understanding "If it is raining and the event is being held outdoors, then the event will be cancelled".

This example shows how propositional logic can be used to systematically analyse relationships between simple statements, providing a structured approach to reasoning about different scenarios.

2.1.2 First Order Logic

First-order logic (FOL) [33–36], also known as predicate logic or first-order predicate logic, is a formal system for defining and working with propositions and predicates, providing a framework for formalising statements about objects and their relations. It is widely used in fields such as mathematics, philosophy, linguistics and computer science. Unlike propositional logic, which deals only with the true or false values of whole propositions, FOL allows for more detailed and complex statements by introducing elements such as quantifiers and predicates.

FOL uses symbols to represent objects, relationships and operations within a domain. The syntax of FOL consists of several key elements: constants, variables, predicates, functions, logical connectives and quantifiers.

Constants are symbols that denote specific objects in the domain of discourse. These objects are usually represented by lowercase letters such as a, b, and c. For example, in a domain about people, constants might refer to individuals such as josé, maria and joão.

Variables are symbols that represent arbitrary objects in the domain, typically identified by letters such as X, Y, and Z. Variables allow general statements to be made about objects, allowing universal and existential claims to be formulated.

Predicates are symbols used to express properties of objects or relationships between multiple objects. For example, $P(X)$ can express "X is a prime number", and $Q(X, Y)$ can express "X is greater than Y".

Functions map objects to objects within the domain. They are similar to mathematical functions and are often represented by symbols such as $f(X)$ or $g(X, Y)$. For example, if f represents the function "father of," then $f(X)$ would denote the father of X.

Logical connectives are symbols that combine or modify statements to form more complex expressions. The main logical connectives in FOL include:

- $\land$ (and): A conjunction that combines two statements, both of which must be true.
- $\lor$ (or): A disjunction that combines two statements, at least one of which must be true.
- $\neg$ (not): A negation that inverts the truth value of a statement.
- $\rightarrow$ (implies): A conditional that asserts the second statement must be true if the first one is true.
- $\leftrightarrow$ (if and only if): A biconditional that asserts both statements must be simultaneously true or false.

Quantifiers are symbols used to indicate the scope of a statement over the objects in the domain. There are two primary quantifiers in FOL:

- $\forall$ (for all): A universal quantifier indicating that a statement applies to all objects in the domain. For example, $\forall x P(x)$ asserts that $P(x)$ is true for every object x.
- $\exists$ (there exists): An existential quantifier indicating that there is at least one object in the domain for which the statement is true. For instance, $\exists y Q(y)$ asserts that there is at least one y for which $Q(y)$ holds true.

These symbols together form the basic vocabulary of first-order logic, allowing for the construction of precise and complex logical statements about objects and their interrelations within a specified domain.

Terms in first-order logic represent objects and can be constants, variables, or function applications. For instance, if f is a function and x is a variable, then $f(x)$ constitutes a term. Terms serve as the building blocks for constructing formulas, which can be either atomic or complex.

Atomic formulas are the simplest type of formulas and consist of predicates applied to terms. Examples of atomic formulas include $P(x)$, where P is a predicate and x is a term, or $Q(x, f(y))$, where Q is a predicate applied to the terms x and $f(y)$, with f being a function and y a variable.

Complex formulas are formed by combining atomic formulas using logical connectives and quantifiers. Logical connectives such as $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), and $\leftrightarrow$ (if and only if) enable the construction of more intricate statements. For example, $\neg P(x)$ is a complex formula where the truth value of $P(x)$ is negated. Similarly, $P(x) \wedge Q(y)$ combines two atomic formulas into a conjunction that is true only if both $P(x)$ and $Q(y)$ are true.

Quantifiers, $\forall$ (for all) and $\exists$ (there exists), further enrich the expressive power of first-order logic. For instance, $\forall x P(x)$ asserts that the predicate P holds for all objects x in the domain of discourse, while $\exists y Q(y)$ asserts that there exists at least one object y for which the predicate Q is true. These constructs allow for the creation of detailed and nuanced logical statements about objects and their relationships within a specified domain.

The semantics of first-order logic (FOL) provides the meaning of its syntactical constructs by interpreting symbols within a specific domain of discourse. This process involves several key components: the domain of discourse, interpretation, and truth.

The Domain of Discourse is the set of all objects under consideration. It defines the scope within which the symbols of FOL are interpreted.

The Interpretation assigns meaning to the various symbols used in FOL:

- The constants are mapped to specific objects in the domain.
- The variables represent elements of the domain.
- The predicates are interpreted as relations on the domain.
- The functions are interpreted as mappings from tuples of domain elements to domain elements.

An interpretation also determines the truth value of statements within the domain. For example, the statement $\forall x P(x)$ is true if $P(x)$ holds for every x in the domain. This allows FOL to express and evaluate the truth of logical statements based on the relationships and properties of objects within the domain.

For instance, the formula $\exists x(\text{Human}(x) \wedge \text{Loves}(x, \text{IceCream}))$ means "There exists some x such that x is a human and x loves ice cream." Here, the existential quantifier $\exists$ indicates that at least one element x satisfies both conditions: being human ($\text{Human}(x)$) and loving ice cream ($\text{Loves}(x, \text{IceCream})$). This concise expression efficiently captures the existence of an object with specific properties within a domain.

2.2 Inductive Logic Programming

ILP [21, 37, 38] is a branch of symbolic machine learning that attempts to induce logical programs from examples. ILP is dedicated to learning logical theories composed of Horn clauses in first-order logic, using both examples and prior knowledge. Horn clauses are a specific subset of first-order logic, known for their simplicity and effectiveness in representing logical relationships and constraints. This approach combines knowledge representation through logical programming languages with inductive learning techniques to derive logical rules that satisfy the given examples.

It predominantly uses logic programming, in particular the Prolog language, to represent its constructs. The two main pillars of its theoretical foundation are logic programming and induction.

First, logic programming forms the backbone of ILP, allowing knowledge and hypotheses to be represented by logic programs. These programs consist of a set of rules and facts, typically expressed as Horn clauses.

Second, induction is a fundamental process in ILP. It involves the task of generalising from specific instances to broader, more comprehensive rules. The primary goal here is to induce a hypothesis that not only accounts for the given observations, but also has predictive power for new, unseen instances. By abstracting from the details of the given examples, ILP aims to discover underlying patterns and relationships that can comprehensively explain the data.

The task of ILP can be defined within a formal framework. The essence of ILP lies in its goal of deriving a logical hypothesis from given data consisting of background knowledge and examples. In ILP, input data is expressed in the form of logical programs that include both positive and negative examples. The objective is to learn a logical clause that satisfies all positive examples while not satisfying any of the negative examples. This process can be broken down into specific components and goals.

2.2.1 Given

- Background Knowledge (B): This encompasses a set of facts and rules that are known beforehand. These form the initial knowledge upon which further reasoning and learning are built.
- Positive Examples (E^+): These are instances where the target concept holds true. They serve as the basis for identifying patterns and constructing the desired hypothesis.
- Negative Examples (E^-): These are instances where the target concept does not hold true. They are crucial for ensuring that the induced hypothesis does not overgeneralize and incorrectly cover instances that should be excluded.

A hypothesis H that satisfies the following conditions:

1. $B \cup H \models E^+$: This means that the combination of background knowledge B and the hypothesis H must entail all the positive examples. In other words, the hypothesis should correctly explain all instances where the target concept is true.
2. $B \cup H \not\models E^-$: This condition ensures that the background knowledge B and the hypothesis H together do not entail any of the negative examples. This means that the hypothesis should exclude all instances where the target concept is false, thus maintaining its accuracy and specificity.

The presence of prior knowledge is crucial for the effectiveness of this approach, manifesting through facts, such as *father(Lucas, Maria)*, or rules, such as *grandfather(X,Y) :- father(X,Z), father(Z,Y)*, where the interpretation would be "X is the grandfather of Y *if* X is the father of Z *and* Z is the father of Y". These rules represent logical clauses in the form $\forall X, Y, Z, \mathit{father}(X, Z) \wedge \mathit{father}(Z, Y) \rightarrow \mathit{grandfather}(X, Y)$. The set *father(X,Z), father(Z,Y)* is referred to as the "body" of the rule, while *grandfather(X,Y)* is termed the "head" of the rule.

The investigation of a logical clause in ILP involves a saturation phase, aiming to efficiently identify all implicit relationships in the data instances. ILP uses a range of methods and search strategies to generate and refine hypotheses. These methods include inverse entailment, top-down and bottom-up approaches, refinement operators and various search strategies.

- Top-Down: This approach starts with the most general hypothesis and specializes it to fit the data. An example of this methodology is the FOIL algorithm [39].

- Bottom-Up: This approach begins with specific facts (examples) and generalizes them. The GOLEM system [40] exemplifies this methodology.

Figure 2.1 shows the difference between the top-down and bottom-up approaches in ILP. On the left is the top-down approach, starting with a general hypothesis at the top. This general hypothesis is gradually specialised through successive steps, leading to more specific hypotheses, until the data (examples) are fully explained. This method aims to refine broad concepts into detailed rules that fit the observed data.

Conversely, the right side of the figure represents the bottom-up approach. This method starts with the specific data (examples) at the bottom. From these specific instances, the approach gradually generalises the hypotheses, moving upwards towards a more general hypothesis. Each step involves abstracting the specific facts into broader generalisations, culminating in a broad general hypothesis that encapsulates the data.

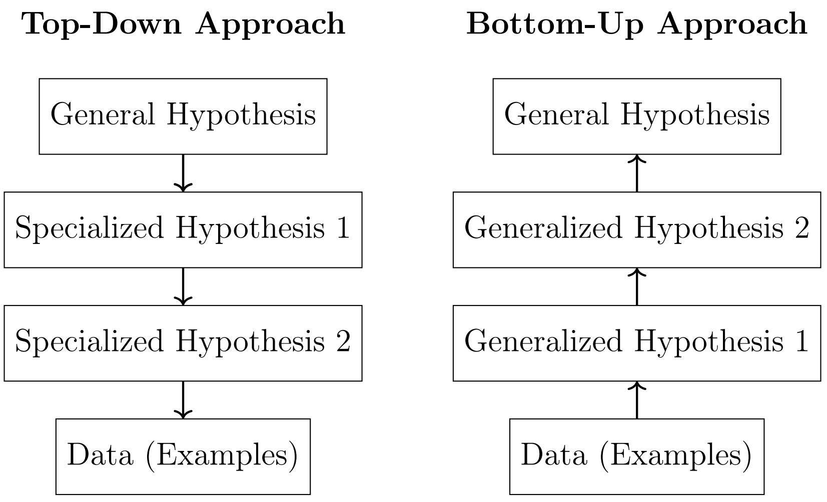


Figure 2.1: Illustration of Top-Down and Bottom-Up Approaches in ILP

Introduced in the Progol system [41], inverse entailment involves finding a hypothesis H such that $B \cup \neg E \models \neg H$. This approach combines bottom-up with inverse entailment, and transforms the problem into a deductive one, making it easier to manage the search space.

ILP faces several significant challenges that affect its efficiency and applicability. One of the main challenges is computational complexity. The hypothesis space in ILP is often large, resulting in high computational demands for both search and inference. As the system attempts to explore and evaluate numerous potential hypotheses, the computational resources required can become significant. This complexity can hinder the practical application of ILP, particularly in scenarios involving large data sets or complex background knowledge. Effective strategies to manage and reduce this computational burden are essential for the further development and wider use of ILP.

2.2.2 Mode Declarations

Mode declarations are an important aspect of ILP. They serve to define the structure and constraints of the hypotheses that the ILP system can generate, effectively guiding the search through the hypothesis space. By specifying which predicates and terms can be used and how they can be instantiated, mode declarations ensure that hypotheses remain meaningful and computationally feasible.

They are syntactic constructs that specify how predicates can be used in hypotheses. They indicate which arguments of a predicate should be input (already known values) and which should be output (values to be determined). This distinction helps in the systematic construction and refinement of hypotheses during the learning process.

Its advantages are, firstly, that it restricts the hypothesis space by defining the allowable forms of predicates and their arguments. This narrows down the vast space of possible hypotheses.

Secondly, mode declarations ensure that the hypotheses generated are meaningful and relevant to the domain of interest. By specifying the structure and constraints of predicates, they guide the system in generating hypotheses that are semantically valid and applicable to the given problem.

Finally, they improve the computational efficiency of the ILP system. By reducing the number of potential hypotheses that need to be considered, they significantly reduce the computational resources required for hypothesis generation and evaluation. This efficiency is particularly important in complex domains with large datasets, where the computational cost of exploring all possible hypotheses can be prohibitive.

A mode declaration typically consists of the following components:

1. The name of the predicate and its arity (number of arguments).

2. Symbols indicating whether an argument is an input (+), or an output (-) The first argument is the maximum number of occurrences of this predicate, if the argument is (*) it means that the predicate can be used as many times as needed. Input arguments are known values, while output arguments are variables that the ILP system has to find.

For example, in the mode declaration `:- modeb(*, parent(+person, #person)).`, the predicate `parent/2` has two arguments. The first argument is an input (denoted by +), and the second argument is a constant (denoted by #). Meanwhile, the mode declaration is `:- modeb(*, grandparent(+person, -person)).` The first argument is an input (denoted by +) and the second argument is an output (denoted by -). This declaration indicates that for any clause using the `parent` predicate, the first argument must be instantiated and the second argument is to be determined by the ILP system.

In these declarations, the `parent` and `grandparent` predicates are followed by a number after a slash, such as `parent/2` and `grandparent/2`. This number represents the *arity* of the predicate, which indicates the number of arguments it takes. For example, `parent/2` means that the predicate `parent` takes two arguments, while `grandparent/2` also takes two arguments.

The symbols +, # and - represent different patterns for the arguments:

- + indicates that the argument is an input, i.e. it must already be instantiated when the predicate is used.
- # indicates that the argument is a constant, which cannot be changed or further instantiated.
- - marks the argument as an output, which is not instantiated and is inferred by the ILP system.

Thus, the `predicate/arity` notation specifies the name of the predicate and the number of arguments it takes, while the mode declarations describe how those arguments are used during the ILP process.

Now let's consider a simple ILP problem in the area of family relationships, with the following mode declarations

```
:- modeh(*, grandparent(+person, -person)).
:- modeb(*, parent(+person, #person)).
:- modeb(*, parent(#person, +person)).
:- modeb(*, male(+person)).
:- modeb(*, female(+person)).
```

- `:- modeh(*, grandparent(+person, -person)).`: This declaration specifies that the head of a hypothesis can use the `grandparent` predicate, where the first argument is an input and the second is an output.
- `:- modeb(*, parent(+person, #person)).`: This declaration indicates that the `parent` predicate can be used in the body of a hypothesis, with the first argument as input and the second as output.

- `:- modeb(*, parent(#person, +person)).`: This allows the `parent` predicate to be used with the first argument as output and the second as input.
- `:- modeb(*, male(+person)).` and `:- modeb(*, female(+person)).`: These declarations allow the `male` and `female` predicates to be used with a single input argument.

In summary, mode declarations are important in ILP for several reasons: they guide the construction of hypotheses by specifying valid forms and combinations of predicates, enhance search efficiency by reducing the hypothesis space and computational complexity, and provide flexibility in the use of predicates, allowing various ways to construct and test hypotheses.

2.2.3 Bottom Clause

The bottom clause plays an important role in constraining the hypothesis space, making the search for an appropriate hypothesis more efficient. The bottom clause, typically denoted as $\perp$, is the most specific clause that can be constructed from a given example using the background knowledge and language constraints. It serves as a lower bound in the hypothesis space from which more general hypotheses can be derived. The bottom clause contains all the literals relevant to the example according to the background knowledge, ensuring that any hypothesis generalised from it will cover the example.

The head of the clause initially contains a single predicate representing the target concept, while the body starts empty. Thus, the most general clause considers any input as positive, for example $\forall X, Y grandparent(X, Y)$. Next, the system chooses a specific example e (a positive instance) that the ILP system aims to explain. Next, use mode declarations to specify which predicates and terms can be used in the clauses. This helps to restrict the literals that can appear in the lower clause. Finally, it saturates the example e with all possible literals that can be derived using the background knowledge B. This means adding every possible literal that relates to the example based on the predicates and terms allowed by the mode declarations. As the search progresses, the clause is refined and instantiated by adding predicates to the body of the lower clause, based on the observed positive and negative examples, along with any prior knowledge or constraints. The correspondence between *modeh* and *modeb* is structured to preserve variable chaining, ensuring that any variable used as an input in a body predicate is also either an input variable in the head of the rule, or an output variable in another body predicate.

The Algorithm 1 is for generating bottom clauses. It is the modified version of Progol's Bottom Clause created by [23]. The main differences between this algorithm

```
1: $E_\perp = \emptyset$
2: **for** each example $e$ of $E$ **do**
3:   Add $e$ to background knowledge and remove any previously inserted examples
4:   $inTerms = \emptyset$, $\perp_e = \emptyset$, $currentDepth = 0$
5:   Find the first *mode declaration* with head $h$ which $\theta$-subsumes $e$
6:   **for all** $v/t \in \theta$ **do**
7:     If $v$ is of type #, replace $v$ in $h$ to $t$
8:     If $v$ is of one of $\{+, -\}$, replace $v$ in $h$ to $v_k$, where $k = hash(t)$
9:     If $v$ is of type $+$, add $t$ to $inTerms$
10:  **end for**
11:  Add $h$ to $\perp_e$
12:  **for** each body mode declaration $b$ with recall value $recall$ **do**
13:    **for all** substitutions $\theta$ of arguments $+$ of $b$ to elements of $inTerms$ **do**
14:      **repeat**
15:        **if** querying $b\theta$ against the background knowledge succeeds **then**
16:          **for** each $v/t$ in $\theta$ **do**
17:            If $v$ is of type #, replace $v$ in $b$ to $t$
18:            Else, replace $v$ in $b$ to $v_k$, where $k = hash(t)$
19:            If $v$ is of type $-$, add $t$ to $inTerms$
20:          **end for**
21:          Add $b\theta$ to $\perp_e$, if it has not been added already
22:        **end if**
23:      **until** $recall$ number of iterations has been reached
24:    **end for**
25:  **end for**
26:  Increment $currentDepth$; if it is less than $depth$, go back to line 12
27:  If $e$ is a negative example, add an explicit negation symbol "$\sim$" to the head of $\perp_e$
28:  Add $\perp_e$ to $E_\perp$
29: **end for**
30: **return** $E_\perp$
```

Algorithm 1

and Progol's original are two: First, it introduces the depth parameter to allow the user to specify the depth of the clause; second, it allows to create a bottom clause on positive and negative examples, which the original algorithm did not allow the user to do.

Now, consider an example in the domain of family relationships, where the background knowledge includes predicates such as `parent/2`, `male/1` and `female/1`. Suppose we have a positive example `grandparent(maria, jose)`.

1. **Example**: `grandparents(maria, jose)`.

2. **Mode declarations**:

```
:- modeh(*, grandparent(+person, -person)).
:- modeb(*, parent(+person, -person)).
:- modeb(*, parent(-person, +person)).
:- modeb(*, male(+person)).
:- modeb(*, female(+person)).
```

3. **Background Knowledge**:

```
parent(maria, carlos).
parent(maria, ana).
parent(carlos, jose).
parent(ana, luis).
parent(ana, sofia).

female(maria).
female(ana).
female(sofia).
male(jose).
male(carlos).
male(luis).
```

4. **Saturation**: Using the background knowledge and mode declarations, the bottom clause for `grandparent(maria, jose)` might contain

```
grandparent(A, B) :-
    parent(A, C),
    parent(C, B),
    male(B).
```

The grandparent rule states that A is the grandparent of B if A is the parent of C and C is the parent of B. In addition, B must be male for A to be recognised as a grandparent in this context. In summary, A is the grandparent of B if A is the parent of B's parent and B is male. However, it's important to note that the bottom clause generates the most specific clause for each example up to a certain depth, based on the BK and the mode declaration. In this clause, specifying that B must be male is not strictly necessary for a general grandparent rule, but because the bottom clause systematically constructs the most specific clause for the given examples, it includes both the parental relationships and other conditions, such as gender, where relevant.

This example illustrates how mode declarations can guide the ILP system in the construction of hypotheses. The mode declarations specify the valid forms of the predicates `grandparent`, `parent`, `male` and `female`. The saturation process then uses these declarations together with the background knowledge to form a bottom clause, which is a specific hypothesis explaining the given example. In this case, the hypothesis suggests that for `jose` to be a grandparent of `maria`, there must be an intermediate `parent` relationship through `maria`, and `jose` must be male.

2.3 Graphs and Hypergraphs

Graphs [42–48] and hypergraphs [49–53] are structures in discrete mathematics, with wide-ranging applications in computer science, network theory, and various scientific disciplines. A graph G is an ordered pair (V, E), where:

- V is a set of vertices.
- $E \subseteq \{(u, v) \mid u, v \in V\}$ is a set of edges.

In other words, a graph is a collection of vertices (or nodes) and edges connecting pairs of vertices. Graphs can be directed, where edges have a direction, or undirected, where edges have no direction. They can also be weighted, with numbers assigned to edges representing quantities like distance or cost, or unweighted.

For a directed graph G, the edges E are ordered pairs of vertices:

- $E \subseteq \{(u, v) \mid u, v \in V\}$.

For a weighted graph G:

- Each edge $e \in E$ is assigned a weight $w(e) \in \mathbb{R}$.

Graphs are used to model relationships and processes in numerous fields, such as social networks, communication networks, and biological systems.

In a graph, a path is a sequence of edges connecting a sequence of vertices, and a cycle is a path that starts and ends at the same vertex without traversing any edge more than once.

For instance, given $V = \{A, B, C, D, E\}$ and $E = \{\{A, B\}, \{A, C\}\{B, D\}, \{D, E\}\}$, we have:

- $G = (V, E)$ represents a graph.
- $V = \{A, B, C, D, E\}$ represents the nodes.
- $E = \{(A, B), (A, C), (B, D), (D, E)\}$ represents the edges.

Hypergraphs extend the concept of graphs by allowing edges, known as hyperedges, to connect more than two vertices. This makes hypergraphs suitable for modeling more complex relationships. For example, in a hypergraph representing a collaboration network, a hyperedge might connect all researchers who have co-authored a paper. This allows for the representation of multi-way relationships that cannot be captured by simple graphs.

In a hypergraph, the incidence structure is often of interest. This structure describes which vertices are contained in which hyperedges. Hypergraphs can be uniform, meaning all hyperedges have the same number of vertices, or non-uniform. An example of a uniform hypergraph is a 3-uniform hypergraph, where each hyperedge connects exactly three vertices. The study of hypergraphs includes examining properties such as hyperedge connectivity, vertex degrees (number of hyperedges a vertex belongs to), and coloring problems.

Both graphs and hypergraphs are represented mathematically using matrices and adjacency lists. For graphs, the adjacency matrix is a square matrix used to represent a finite graph, with elements indicating whether pairs of vertices are adjacent or not. In hypergraphs, the incidence matrix is used, with rows representing vertices and columns representing hyperedges, where an entry indicates the presence of a vertex in a hyperedge. These representations are crucial for computational algorithms that process graphs and hypergraphs.

Figure 2.2 illustrate the key differences between a graph and a hypergraph.

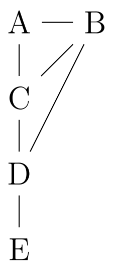
Figure 2.2a, we have a simple graph. A graph consists of vertices (or nodes) connected by edges. Each edge connects exactly two vertices. The vertices are labeled A, B, C, D, and E, and there are edges between certain pairs of vertices, forming a network of pairwise connections.

Figure 2.2b, we depict a hypergraph. Unlike a simple graph, a hypergraph allows for hyperedges, which can connect more than two vertices at a time. In this example, the vertices are the same (A, B, C, D, and E), but the connections are represented by ellipses that enclose groups of vertices. These ellipses (hyperedges) illustrate multi-way relationships, showing how hypergraphs can capture more complex interactions.

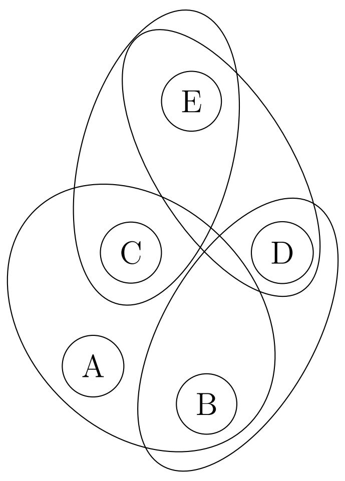
In summary, graphs and hypergraphs are powerful tools for modeling and analyzing complex relationships in various domains. Graphs are suitable for pairwise relationships, while hypergraphs are needed for multi-way interactions.

2.3.1 From Relational to Graph Representation

In Prolog, data is structured using predicates applied to terms, where each predicate defines a relationship. A relational dataset consists of facts and target examples, divided into positive and negative cases. This dataset can be represented as a bi-



(a) A simple graph with vertices A, B, C, D, and E connected by edges.



(b) A hypergraph with the same vertices, but hyperedges (ellipses) connecting multiple vertices at once.

Figure 2.2: Comparison between a graph and a hypergraph.

partite graph, where terms and predicates are placed in different sets and connected based on the relationships specified by the facts.

To create a bipartite graph from the dataset, the process begins by selecting one of the target examples. This example, either positive or negative, will guide the construction of the graph.

1. **Choose a target example:** Begin by selecting a target example from the set of positive or negative cases.

2. **Identify related facts:** Gather all the facts that relate to the chosen example. These facts are those that have the target as a term.

3. **Create Two Sets:** Separate the terms and predicates involved in the facts into two different sets. One set will contain the terms and the other the predicates.

4. **Connect Sets:** Establish links between terms and predicates based on the relationships defined in the selected facts.

This process is repeated for each target example, resulting in a bipartite graph for each individual example.

For example, consider the target is west6 and the Prolog facts are as follows:

$$\texttt{has\_car(west6,car\_61)} \tag{2.1}$$

$$\texttt{has\_car(west6, car\_62)} \tag{2.2}$$

The relations `has_car(west6, car_61)` and `has_car(west6, car_62)` describe that the entity "west6" has two different cars: "car_61" and "car_62". In other words, "west6" owns or is associated with these two cars.

In a bipartite graph, the data is represented as two sets of nodes:

- **Predicates**:`has_car`.
- **Terms**: `west6`, `car_61`, and `car_62`.

The edges are drawn between predicates and their corresponding terms:

- `has_car` is connected to `west6` and `car_61`.
- `has_car` is linked to `west6` and `car_62`.

Figure 2.3 is the bipartite graph where the predicates are connected to the terms based on the facts provided in Prolog.

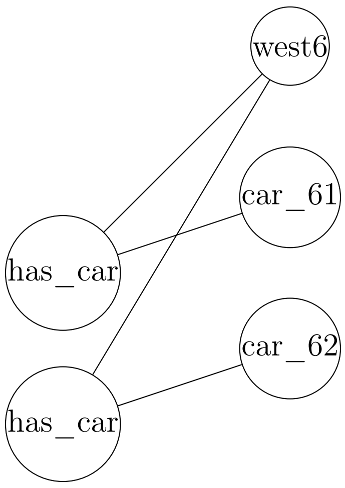


Figure 2.3: Bipartite graph representation of relational data, illustrating the connections between predicates and terms. The predicates `has_car` are connected to various terms, such as `west6` and `car_61`.

In order to use methods like GNN, we need to represent this data as feature vectors. One-hot encoding can be used to create the features for the predicates and terms.

The feature vector assigns a specific position in the feature vector to represent each constant and predicate. In our example with one predicate and 3 constants, the feature vector will look like this

$$\begin{aligned}
\texttt{has\_car} &: [1, 0, 0, 0] \\
\texttt{has\_car} &: [1, 0, 0, 0] \\
\texttt{west6} &: [0, 1, 0, 0] \\
\texttt{car\_61} &: [0, 0, 1, 0] \\
\texttt{car\_62} &: [0, 0, 0, 1]
\end{aligned}$$

However, this method may have limitations when trying to retrieve data related to a target example by identifying relevant facts from related terms - such as the fact load(car_61, circle, 3) which is not retrieved because it does not have the term west6, even though it provides information about car_61.

The process of representing the hypergraph is similar to that of the bipartite graph. The main difference is in the structure: instead of using predicates as nodes, the representation involves hyperedges connecting multiple nodes.

Alternative works also converts relational data into graph structure [2, 4, 5].

2.4 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks specifically designed to process structured data in the form of graphs [6–8, 54–56]. Unlike traditional neural networks, which operate on fixed structures such as grids or sequences, GNNs are built to capture the dependencies between entities, which are naturally represented as nodes in a graph.

The main idea of GNNs is message passing, where each node in the graph updates its state by aggregating information from its neighbours. This iterative process allows nodes to refine their representations based on the structure of the graph and the characteristics of neighbouring nodes.

Initially, the hidden state of each node i, denoted by $\mathbf{h}_i^{(0)}$, is set using the associated feature vector $\mathbf{x}_i$ such that $\mathbf{h}_i^{(0)} = \mathbf{x}_i$. As GNN processes the graph layer by layer, the hidden state $\mathbf{h}_i^{(t)}$ is updated at each step t by message passing, where each node aggregates information from its neighbours and combines it with its own current hidden state. This process allows the hidden states to capture increasingly rich representations of both the local neighbourhood and the global structure of the graph. Over multiple layers, the hidden states evolve to reflect not only the individual node features, but also the broader context of the graph, making them suitable for tasks such as node classification, graph classification, or link prediction.

The general steps for applying a GNN are outlined below:

1. Initialisation:

    - We begin by assigning a feature vector to each node in the graph. The feature vector, denoted by $\mathbf{x}_i$, represents the initial information or attributes of node i.
    - Initialise the hidden state of each node i, denoted by $\mathbf{h}_i^{(0)}$, using the associated feature vector $\mathbf{x}_i$. In this step, $\mathbf{h}_i^{(0)} = \mathbf{x}_i$.
    - The initial hidden states serve as input to the subsequent message passing and update phases.

2. Message Passing:

    - At each step, nodes exchange information with their neighbours.
    - Node i sends a message to its neighbour j, based on its current state $\mathbf{h}_i^{(t)}$ at step t.
    - Node j collects messages from all its neighbours, allowing it to learn about its local neighbourhood.

3. Node Update:

    - After receiving messages from its neighbours, node j aggregates the incoming messages. The aggregation function may involve summing, averaging, or applying more complex operations to the received messages.
    - Once the messages have been aggregated, the node j updates its hidden state. This update is achieved by applying a node update function that transforms the aggregated messages and the previous hidden state of the node. The updated hidden state for node j at iteration $t+1$ is denoted by $\mathbf{h}_j^{(t+1)}$.
    - The updated hidden state reflects the refined representation of node j, which now includes information from its immediate neighbours.

4. Iteration:

    - The message passing and node update steps are repeated for a predetermined number of iterations or until a predetermined stopping criterion is met.
    - With each iteration, nodes incorporate information from nodes further away in the graph, gradually increasing the amount of information they can access.

5. Readout/Pooling:

    - At the end of the iterations, each node has a final hidden state containing information from its neighbourhood.
    - To generate a graph-level representation, the final hidden states of all nodes are combined using a readout or pooling function. This function may involve summing, averaging, or applying more sophisticated operations to the node representations.
    - The result is a single vector representing the entire graph. This vector contains information about both the nodes and the structure of the graph.

6. Output:

    - The graph-level representation obtained from the readout or pooling step is used to perform the desired task.
    - Depending on the application, the task may involve predicting a label for the whole graph (e.g. classification) or predicting a continuous value (e.g. regression).

The message passing process can be described as follows [57]:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}\left(\left\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\right\}\right)\right) && (2.3)\\ &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right), && (2.4)\end{aligned}$$

In this formulation, $h_i^{(l)}$ denotes the representation of node i at level l, while $m_{\mathcal{N}(u)}^{(k)}$ refers to the message from node's neighbours at the same level. $N(i)$ represents the set of neighbours of node i, and the Aggregation and Update are responsible for constructing, aggregating and updating messages.

The following example employs numerical data.

Consider the graph 2.4. It is a graph with 4 nodes labelled from A to D and a feature vector next to it.

We start by generating the message, aggregating it using the mean, and updating the corresponding nodes. Starting with the node A, which is highlighted in orange, and its neighbours, which are highlighted in green, are then considered in the process (see Figure 2.5).

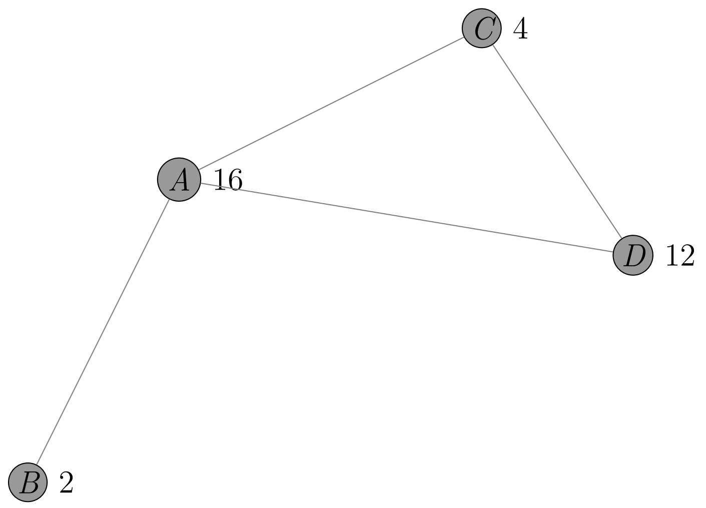


Figure 2.4: The initial graph comprises four nodes, labelled A, B, C and D. Node A is connected to nodes B, C and D, and nodes C and D are also connected to each other.

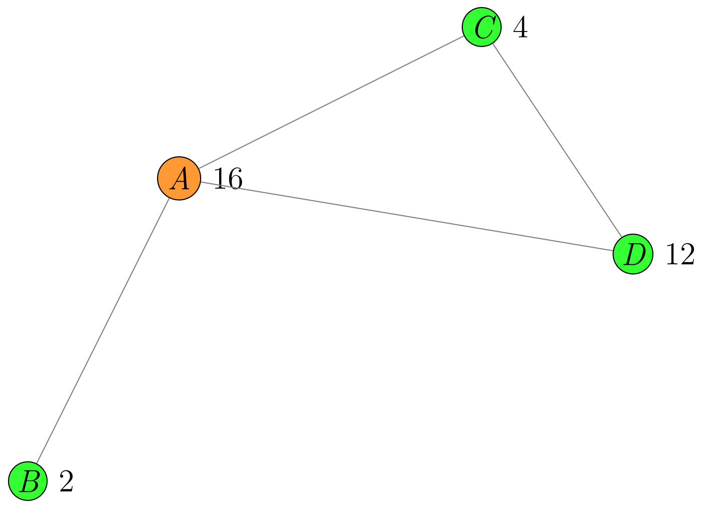


Figure 2.5: Graph illustrating node A (orange) connected to its neighbours, the green nodes B, C and D, by grey edges.

The weight W is set to 1 and the bias B is set to 1. The function f is a non-linear function such as ReLU (Rectified Linear Unit), defined as

$$f(x) = \max(0, x)$$

Beginning with node A:

$$h_A^{(1)} = f\left(W^{(1)} \times \frac{h_B^{(0)} + h_C^{(0)} + h_D^{(0)}}{3} + B^{(1)} \times h_A^{(0)}\right)$$

$$h_A^{(1)} = f\left(1 \times \frac{2+4+12}{3} + 1 \times 16\right)$$

$$h_A^{(1)} = f\left(6 + 16\right)$$

$$h_A^{(1)} = f\left(22\right)$$

In the following, node B.

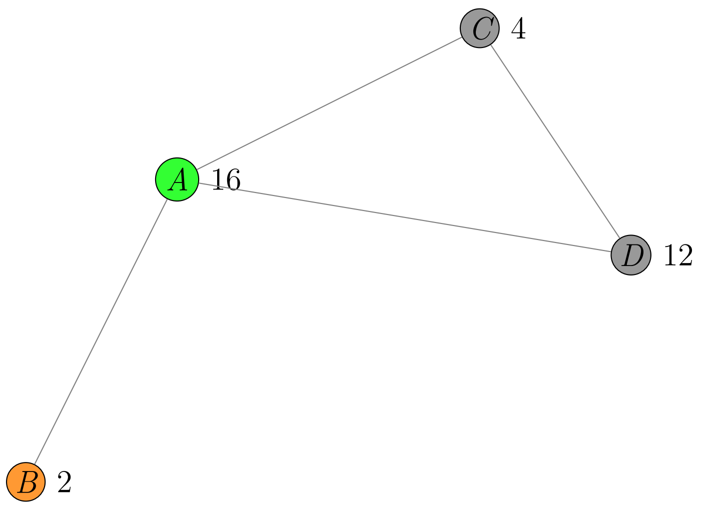


Figure 2.6: Graph with four nodes, where node A (green) is connected to nodes B (orange), C (grey) and D (grey). Node A, shown in green, is a neighbour of node B, shown in orange.

$$h_B^{(1)} = f\left(W^{(1)} \times \frac{h_A^{(0)}}{1} + B^{(1)} \times h_B^{(0)}\right)$$

$$h_B^{(1)} = f\left(1 \times \frac{16}{1} + 1 \times 2\right)$$

$$h_B^{(1)} = f\left(16 + 2\right)$$

$$h_B^{(1)} = f\left(18\right)$$

Next, Node C:

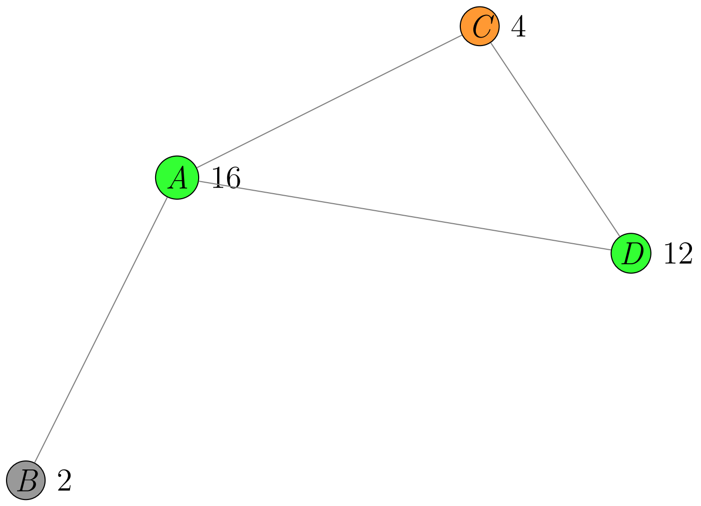


Figure 2.7: Graph with four nodes, where nodes A (green) and D (green) are connected to nodes B (grey) and C (orange). The green nodes A and D are neighbours of the orange node C.

$$h_C^{(1)} = f\left(W^{(1)} \times \frac{h_A^{(0)} + h_D^{(0)}}{2} + B^{(1)} \times h_C^{(0)}\right)$$

$$h_C^{(1)} = f\left(1 \times \frac{16+12}{2} + 1 \times 4\right)$$

$$h_C^{(1)} = f\left(14 + 4\right)$$

$$h_C^{(1)} = f\left(18\right)$$

Finally, Node D:

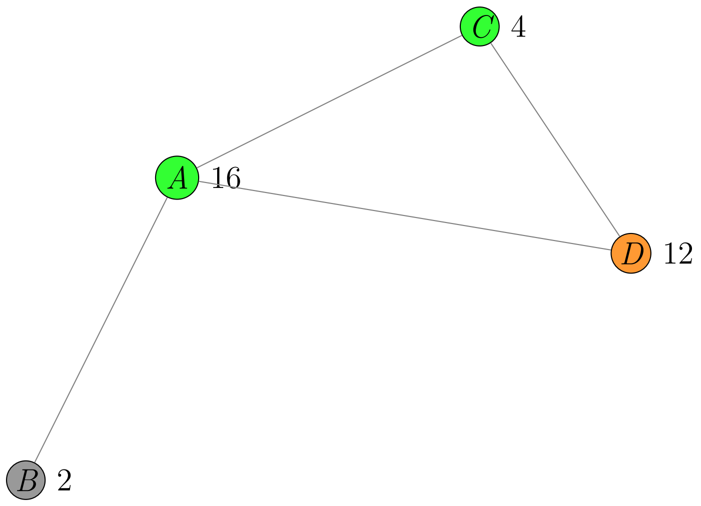


Figure 2.8: Graph with four nodes, where node A (green) and node C (green) are connected to node D (orange) and node B (grey). The green nodes A and C are the neighbours of the orange node D.

$$h_D^{(1)} = f\left(W^{(1)} \times \frac{h_A^{(0)} + h_C^{(0)}}{2} + B^{(1)} \times h_D^{(0)}\right)$$

$$h_D^{(1)} = f\left(1 \times \frac{16+4}{2} + 1 \times 12\right)$$

$$h_D^{(1)} = f\,(10 + 12)$$

$$h_D^{(1)} = f\,(22)$$

This process will occur for all nodes. Figure 2.9 is the updated graph after updating all nodes.

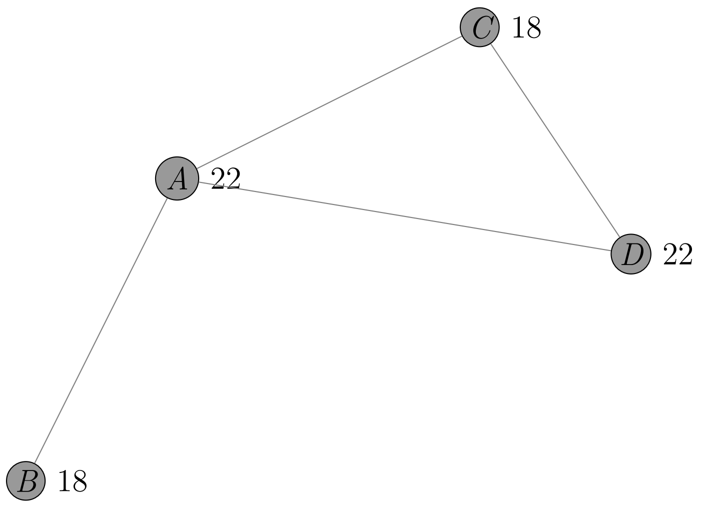


Figure 2.9: Graph with four grey nodes. After the update, node A now has a value of 22, node B has a value of 18, node C has a value of 18 and node D has a value of 22. Each node now has information about its neighbours.

Note that in the first iteration, node B only receives information from node A. In the second iteration, node B receives information not only from node A, but also from A's neighbours, nodes C and D.

In the example, the aggregation function used is the mean. However, alternative aggregation functions such as summation, maximum or minimum can be used depending on the task at hand.

Figure 2.10 summarise the process. The image represents a message-passing layer in a GNN. In this particular GNN layer:

1. **Message Passing**: Nodes in the graph (such as node A) receive information from neighboring nodes (in this case, B, C, and D). Each node sends a message to its neighbors, and these messages contain features or embeddings that are learned during the training of the model.

2. **Aggregation**: The incoming messages from neighbors are aggregated (typically by summation, averaging, or more complex functions) to capture the information from neighboring nodes.

3. **Node Update**: Once aggregation is complete, node A updates its internal state or feature vector based on the aggregated information. The updated state of node A is propagated to other layers or used for prediction.

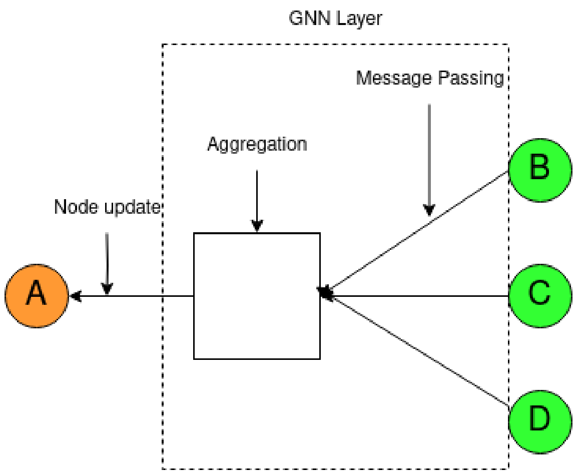


Figure 2.10: Illustration of a single layer in a GNN showing the message-passing mechanism. Node A receives messages from its neighbors (B, C, and D), aggregates them, and updates its state accordingly. This process allows node A to incorporate information from its neighborhood in the graph.

The output of a GNN is a learned representation, often referred to as a node embedding or graph embedding, depending on the task. After several layers of message passing and hidden state updates, the final hidden state $\mathbf{h}_i^{(T)}$ of each node encapsulates information about the node itself as well as the structure and features of its local neighbourhood. This final hidden state serves as a high-level representation that can be used for various downstream tasks. For node-level tasks, such as node classification or regression, $\mathbf{h}_i^{(T)}$ can be used directly as the output for each node. For graph level tasks, such as graph classification, a separate "readout" function is used.

The readout function aggregates the node-level representations into a single graph-level embedding. After GNN has generated node embeddings by aggregating information from neighbouring nodes, the readout function combines these embeddings into a unified graph representation. Common methods for this include simple techniques such as sum, mean or max pooling, which summarise the information across all nodes. For example, applying sum as a readout function to the node embeddings of figure 2.9 (with node A = 22, node B = 18, node C = 18, and node D =

22) results in a graph-level embedding of $22+18+18+22 = 80$. This representation is crucial for downstream tasks such as graph classification.

The output of the readout function is typically fed into a final task-specific layer, such as a softmax for classification or a regression layer to generate the final prediction. In this way, the readout function condenses the node-level information into a comprehensive, informative representation of the whole graph.

In many GNN applications, including our work, the final learned representations are passed to an MLP for classification or regression tasks.

After the GNN has updated the hidden states of the nodes through message passing, the output, whether at the node or graph level, serves as input to an MLP.

For node classification tasks, the final hidden state $\mathbf{h}_i^{(T)}$ of each node is fed into an MLP, which acts as a classifier. The MLP processes these node embeddings and outputs a class prediction for each node. In graph-level tasks, after the readout function aggregates the node representations into a single graph representation, this graph-level embedding is fed into the MLP to predict a label for the entire graph.

The MLP serves as the task-specific component of the pipeline, translating the learned GNN representations into the desired output. Its role is to map the high-dimensional node or graph embeddings into a more interpretable space, such as class probabilities in classification problems.

2.5 Hypergraph NNs

While traditional GNNs operate on graphs composed of nodes and edges, Hypergraph Neural Networks (HGNNs) introduce hyperedges capable of connecting multiple nodes simultaneously. In hypergraphs [58], a hyperedge is a higher-order structure that establishes connections among any number of nodes simultaneously. This enables the representation of n-ary predicate relationships, unlike traditional graphs that encode binary predicates. HGNN leverages this additional structure to capture complex and higher-order interactions in the data.

The main concept behind HGNN is the generalization of the neighborhood aggregation concept employed in GNNs. Unlike node neighborhoods in graphs, which consist only of immediately adjacent nodes, in a hypergraph, the neighborhood of a node is defined by the nodes that share the same hyperedges. This flexibility allows the dissemination of information from a hyperedge to all nodes it connects, incorporating higher-order dependencies.

In typical approaches of HGNNs, a common practice involves decomposing hyperedges into binary connections to employ methods based on GNNs. This enables the use of established GNN architectures, such as Graph Convolutional Networks (GCNs), treating the hypergraph as a bipartite graph. Termed as "hyperedge decomposition," this method transforms the hypergraph into an auxiliary bipartite graph, where hyperedges are represented as edges between hypernodes and conventional nodes. However, as our experiments suggest, this approach may result in information loss and an increase in computational complexity due to the potential explosion in the number of connections between pairs.

The algorithm employed in HGNNs resembles that used in GNNs. The crucial distinction lies in the fact that, when representing data as a hypergraph, neighbors are no longer conventional pairs but are determined by hyperedges. Message passing follows the same logic adopted by GNNs; however, in this context, it is carried out through hyperedges.

In the work of [18], the GCN structure is expanded to hypergraphs by introducing a hypergraph adjacency matrix and a hypergraph Laplacian matrix. The authors present a hypergraph-based approach to adapt conventional GCN operations to hypergraph data. Additionally, in [19], a Hypergraph Convolutional Neural Network (HGCN) is proposed, which generalizes the concept of graph convolutions to hypergraphs. The HGCN can incorporate a hypergraph attention mechanism to assess the importance of hyperedges during convolutional operation. The model is as follows:

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

In the presented equation, the incidence matrix $\mathbf{H}$, which represents the connections, is multiplied by the diagonal matrix of hyperedge weights $\mathbf{W}$.[1] The resulting product is then multiplied by $\mathbf{D}^{-1}$ and $\mathbf{B}^{-1}$, which represent the respective degree matrices. Finally, this multiplication is applied to the transpose of $\mathbf{H}$, the input matrix $\mathbf{X}$, and the parameter matrix $\boldsymbol{\Theta}$, resulting in the updated matrix $\mathbf{X}'$ [19].

In summary, Hypergraph Neural Networks extend the framework of graph neural networks to handle hypergraphs, enabling the modeling of higher-order dependencies among nodes. Moreover, by capturing complex interactions through hyperedges, HGNN provides a richer representation of the data. As indicated by the experimental results presented in this work, this comprehensive representation can enhance performance in various learning tasks involving hypergraph structures.

The process begins with the initialization of a linear transformation using weights initialized to ones. This transformation is applied to an input vector $\mathbf{x}$ to produce an output vector $\mathbf{y}$, which can be expressed mathematically as:

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}$$

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

where W is the weight matrix, and $\mathbf{b}$ is the bias vector.

Step 1: Scatter Operation (Node Degree)

Next, a scatter operation is performed to compute the sum of the weights of the hyperedges connected to node i. The incidence matrix $H(i,e)$ encodes the relationship between nodes and hyperedges (the degree). The inverse degree is computed as:

$$D_i = \frac{1}{\sum_e H(i,e)w_e}$$

where D_i is the degree of node i, and $H(i,e)$ reflects the connections between nodes and hyperedges.

Step 2: Scatter Operation (Hyperedge Degree)

Following this, another scatter operation is conducted to count how many nodes are associated with each hyperedge. This can be written as:

$$B = \sum_i H(i,e)$$

To normalize, the inverse of each element in B is computed, resulting in:

$$B = \frac{1}{B}$$

Step 3.1: Message Passing Step

Next is the propagation step, where the feature vector of each neighboring node x_j is scaled by the corresponding normalization factor B. This operation is represented as:

$$\mathbf{out} = \mathbf{y} \cdot B$$

Step 3.2: Message Passing Step

Following propagation, another message passing step occurs. During this step, the features of neighboring nodes are scaled by the normalization factor D, represented as:

$$\mathbf{out}_i = D_i \cdot \mathbf{out}_i$$

where $\mathbf{out}_i$ is the resulting output for node i.

Step 4: Mean Aggregation

Once propagation is complete, the next step is to compute the mean of the outputs for each node, ensuring that the information is aggregated across the different nodes.

Step 5: Bias Addition

Finally, a bias term is added to the output. This bias is a trainable parameter that is added element-wise to the output vector. The operation can be written as:

$$\mathbf{out}_i = \mathbf{out}_i + \mathbf{b}$$

where $\mathbf{b}$ is the bias vector.

Summary of the Process

The overall process involves:

1. Initializing and applying a linear transformation.
2. Performing scatter operations to count node and hyperedge connections.
3. Computing the inverse of these counts.
4. Propagating information using normalization factors.
5. Aggregating the outputs via mean.
6. Adding a bias term to the final output.

Example Application

Let's apply the above steps with all parameters set to 1 ($W = 1, \mathbf{b} = 1, w_e = 1$) on the following input:

$$x = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 10 \end{bmatrix}$$

Node degree D (before inversion):

$$D = [1, 2, 2, 1]$$

After inverting:

$$D = [1, 0.5, 0.5, 1]$$

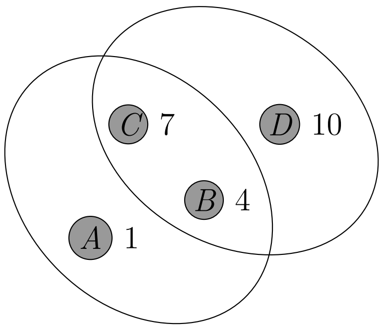


Figure 2.11: Initial hypergraph structure with four nodes labeled A, B, C, and D, where each node is annotated with its corresponding feature values (1, 4, 7, and 10). The ellipses represent two hyperedges connecting subsets of the nodes.

Hyperedge degree B (before inversion):

$$B = [3, 3]$$

After inverting:

$$B = [0.3333, 0.3333]$$

Propagation step:

$$\mathbf{out} = \mathbf{y} \cdot B = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

Message passing step:

$$\mathbf{out} = D \cdot \mathbf{out} = \begin{bmatrix} 4 \\ 5.5 \\ 5.5 \\ 7 \end{bmatrix}$$

Mean aggregation:

$$mean(\mathbf{out}) = \begin{bmatrix} 4 \\ 5.5 \\ 5.5 \\ 7 \end{bmatrix}$$

Bias addition:

$$\mathbf{out}_i = \mathbf{out}_i + \mathbf{b} = \begin{bmatrix} 5 \\ 6.5 \\ 6.5 \\ 8 \end{bmatrix}$$

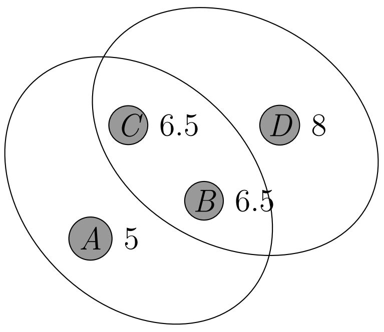


Figure 2.12: Updated hypergraph structure after the transformation process. Nodes A, B, C, and D are shown with their new feature values (5, 6.5, 6.5, and 8), reflecting the results of the propagation and bias addition steps. The ellipses represent the same hyperedges as in the initial structure.

Chapter 3

Proposed Method

In artificial intelligence, the integration of neural computation with symbolic reasoning combines the strengths of both approaches. Neural networks are effective at learning from data and dealing with noisy inputs, but are challenged by tasks involving logical reasoning and the use of prior knowledge. Symbolic reasoning, on the other hand, works well with abstract concepts and logical rules, but struggles with large data sets. Neural-symbolic computing seeks to combine the learning capabilities of neural networks with the expressive power of symbolic logic to tackle tasks that require both data-driven learning and reasoning. This chapter examines one method within this area: the use of hypergraphs to encode symbolic knowledge that is then used by HGNNs to train classifiers.

Hypergraphs extend traditional graphs by providing a representation of complex relationships between entities. Unlike standard graphs, where edges connect pairs of nodes, hypergraphs allow hyperedges to connect multiple nodes, making them suitable for representing n-ary relationships found in symbolic logic. By encoding symbolic knowledge in hypergraphs, we obtain a structured representation.

Our approach focuses on the Bottom HyperGraph [22], a structure that extends the propositionalisation of the bottom clause, an essential element of the ILP search process. The transformation of symbolic knowledge into a hypergraph format allows the integration of data and prior knowledge into the training process. This hybrid method exploits both neural computation and symbolic reasoning.

The construction of the Bottom hypergraph involves several steps. This representation is then processed by a hypergraph neural network, which processes the connected data for classification tasks. To demonstrate our method, we apply it to the trains dataset from the ILP community. Although the dataset is small in terms of examples, it contains rich relational descriptions. We classify trains based on their direction - whether they are heading east or not - using features such as car length, door opening, cargo shape, and number of wheels.

This chapter explores the theoretical underpinnings, construction process and implementation of bottom hypergraphs, and provides details of the proposed method.

3.1 Bottom-HyperGraph

The approach of neural-symbolic computing aims to synergistically merge neural computation with symbolic reasoning. At the core of our method, we integrate symbolic knowledge into hypergraphs, extending the propositionalization of the bottom clause. With this, we can train a classifier using both data and prior knowledge, employing hypergraph neural networks.

We introduce the concept of Bottom-HyperGraph, where, based on the predicates of the bottom clause, we transform terms into nodes and use hyperedges to interconnect terms of any n-ary predicate. The construction process of the Bottom-HyperGraph follows a delineated sequence of steps outlined and presented below.

1. For each example in the training set:
   - (a) Generate a Bottom Clause;
   - (b) Delete the head of the Bottom Clause;
   - (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, i.e., each variable become one node, and constants turn into many nodes. For example: the clause P1(A,circle,1),P2(A,1) will have 1 node for variable A, 1 for circle, and 2 nodes for constant 1; A hyperedge labelled P1 is created, connecting the nodes A and circle with the node 1. Similarly, a new hyperedge labelled P2 should be created, connecting the nodes A and 1. These hyperedges represent the relationship between the terms through the predicate $P1$ and $P2$.
     - 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.

Figure 3.1 illustrates the process flow to create a Bottom-HyperGraph.

With Bottom-HyperGraph we have a hypergraph represented in a format that can be used by a HGNN. Each node individually has its own feature vectors, while hyperedges contain pertinent information about which nodes they are connected to. In addition, the type of hyperedge is encoded as a feature vector. We then

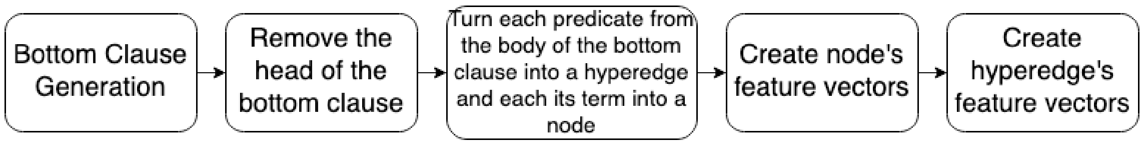

Bottom Clause Generation
Remove the head of the bottom clause
Turn each predicate from the body of the bottom clause into a hyperedge and each its term into a node
Create node's feature vectors
Create hyperedge's feature vectors

Figure 3.1: Construction do Bottom-HyperGraph

use a HGNN to process this structure, where the node and hyperedge features are propagated across the hypergraph to capture complex relationships. After applying the HGNN layers, a readout function is used to aggregate the node and hyperedge representations into a unified feature vector. Finally, a MLP is applied to this aggregated representation to perform classification, allowing us to predict the desired output based on the learned hypergraph embeddings.

To illustrate the entire procedure, we employ the dataset called trains [59], which is related to trains. This dataset is widely recognized in the community of ILP. The objective is to classify trains according to their direction, specifically whether they are heading eastward or not. This classification is based on the characteristics of the train cars, such as length (short or long), door opening, cargo shape, content type, and number of wheels, among others.

The dataset in question contains only ten trains, although the relational descriptions can become notably complex. The solution to the problem utilizes logical clauses, some of which are exemplified below. These clauses incorporate predicates that assert, for example, that train A has a car B with a closed lid containing three circular-shaped cargos, and a car C with a triangular-shaped cargo. Crucially, the simple train dataset includes ternary predicates, as exemplified by $load(C, triangle, 1)$. The mode declarations are detailed in Appendix B, providing an in-depth view of the rule set used in solving the problem.

We operate with four distinct data categories in this mode: #int, #shape, car, and train. Additionally, we have eleven predicates, namely: closed, double, eastbound, has_car, jagged, load, long, open_car, shape, short, and wheels. During data processing, our attention is especially drawn to the symbol #, which functions as a constant in the bottom clause. It is noteworthy that seven of these constants do not have numerical types, namely: circle, ellipse, hexagon, nil, rectangle, triangle, and u_shape.

For the induction of a logical theory using ILP, it is necessary to have both positive and negative examples, along with prior knowledge and mode declarations. In this specific context, we present five positive examples expressed in the form $eastbound(east_n)$, where $1 \leq n \leq 5$. Simultaneously, five negative examples are provided, modeled as $eastbound(west_n)$, where $6 \leq n \leq 10$.

The data consists of several categories of entities and their definitions:

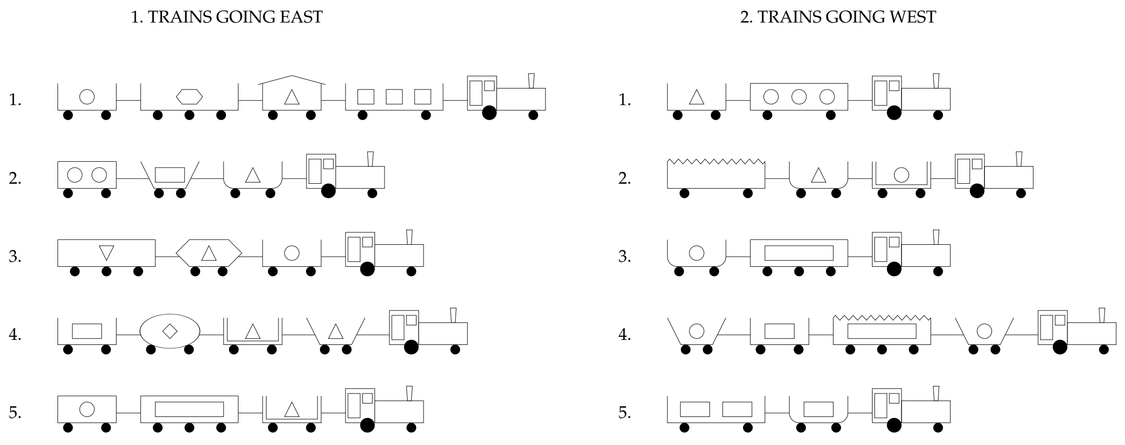


Figure 3.2: Michalski's set of trains [1]

Positive	**Negative**
$eastbound(east1)$.	$eastbound(west6)$.
$eastbound(east2)$.	$eastbound(west7)$.
$eastbound(east3)$.	$eastbound(west8)$.
$eastbound(east4)$.	$eastbound(west9)$.
$eastbound(east5)$.	$eastbound(west10)$.

- **Car Definitions**: There are two car identifiers, namely `car_61` and `car_62`, which are labelled as cars.
- **Shape Definitions**: Seven different shapes are defined: `elipse`, `hexagon`, `rectangle`, `u_shaped`, `triangle`, `circle`, and `nil`.
- **Train Definitions**: Ten trains are listed, five travelling east (`east1` to `east5`) and five travelling west (`west6` to `west10`).

Each category defines entities with unique labels within its type. No specific relationships or properties between these entities are provided in this data.

```
% Type definitions
% Car Definitions
car(car_61).  car(car_62).

% Shape Definitions
shape(elipse).  shape(hexagon).  shape(rectangle).  shape(u_shaped).
shape(triangle).  shape(circle).  shape(nil).

train(east1).  train(east2).  train(east3).  train(east4).  train(east5).
train(west6).  train(west7).  train(west8).  train(west9).  train(west10).
```

The following data provides details about the westbound train `west6` and its associated carriages:

- **Train:** `west6`
  - Has two cars: `car_61` and `car_62`.
- **Car Definitions:**
  - `car_61`
    - Length: Long
    - Shape: Rectangle
    - Status: Closed
    - Load: 3 circles
    - Wheels: 2
  - `car_62`
    - Length: Short
    - Shape: Rectangle
    - Status: Open
    - Load: 1 triangle
    - Wheels: 2

```
% westbound train 6
has_car(west6,car_61).
has_car(west6,car_62).
long(car_61).
short(car_62).
shape(car_61,rectangle).
shape(car_62,rectangle).
closed(car_61).
open_car(car_62).
load(car_61,circle,3).
load(car_62,triangle,1).
wheels(car_61,2).
wheels(car_62,2).
```

With the background knowledge, mode declarations, and presentation of both positive and negative examples, we can generate the bottom clause. Utilizing the first negative example, *eastbound*(*west*6), we create the bottom clause 3.1.

$$\begin{gathered} eastbound(A):- \\ closed(B), has\_car(A,B), \\ has\_car(A,C), load(B, circle, 3), \\ load(C, triangle, 1), long(B), \\ open\_car(C), shape(B, rectangle), \\ shape(C, rectangle), short(C), \\ wheels(B,2), wheels(C,2). \end{gathered} \tag{3.1}$$

The clause 3.1 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.

Having created the bottom clause, the next step is to remove its head. The head refers to the objective of the clause. Once removed, we are left with the body of the clause, which contains the predicates and terms relevant to this example and is encoded as a hypergraph. This process ensures that the feature vector captures only the body information following the chaining process of the bottom clause, without the goal that the classifier will learn. The example 3.2 corresponds to the bottom clause 3.1 without the head.

$$\begin{gathered} closed(B), has\_car(A,B), \\ has\_car(A,C), load(B, circle, 3), \\ load(C, triangle, 1), long(B), \\ open\_car(C), shape(B, rectangle), \\ shape(C, rectangle), short(C), \\ wheels(B,2), wheels(C,2). \end{gathered} \tag{3.2}$$

In the "trains" example, where hypergraphs go beyond pairwise relations between nodes, the load relation is set up as a hyperedge connecting three nodes. Similarly, the closed property can be represented by a unary relation forming a hyperedge with a single node. This setup affects the message-passing process: in a bipartite graph, two hops are required for a predicate node to reach another predicate node, or for a term node to reach another term node. In a hypergraph, only one hop is required because each hyperedge connects multiple nodes directly, allowing faster communication between them. This direct connection in hypergraphs reduces the complexity of graph traversal compared to bipartite graphs.

To illustrate this difference, figures 3.3a and 3.3b show the same relations: $closed(B), has\_car(A, B), load(B, circle, 3)$, but in distinct representations.

Figure 3.3a shows a bipartite graph with two different sets of vertices. Edges are only drawn between vertices from different sets. Bipartite graphs are often used to model relationships between two types of entities. Here, terms are on one side and relationships are on the other, with edges indicating when a term is part of a relationship.

In contrast, Figure 3.3b shows a hypergraph. A hypergraph generalises a standard graph by allowing a hyperedge to connect any number of vertices. This makes hypergraphs useful for representing complex relationships involving multiple entities at once. In this case, concepts are nodes and relationships are hyperedges.

Although the representations are different, both diagrams show the same relationships. The bipartite graph and the hypergraph express the same underlying data structure, but they do so in different ways.

By comparing the two figures, we can see how different graph forms provide different views and ways of managing the same set of relationships.

In order to process the hypergraph for HGNN training, feature vectors must be created to represent the hypergraph. Using the mode file, we can identify all the predicates and term types that will appear in the bottom clauses. In the case of the trains dataset, the feature vector has 23 positions that encode the necessary information, the predicates and term types, for HGNN training.

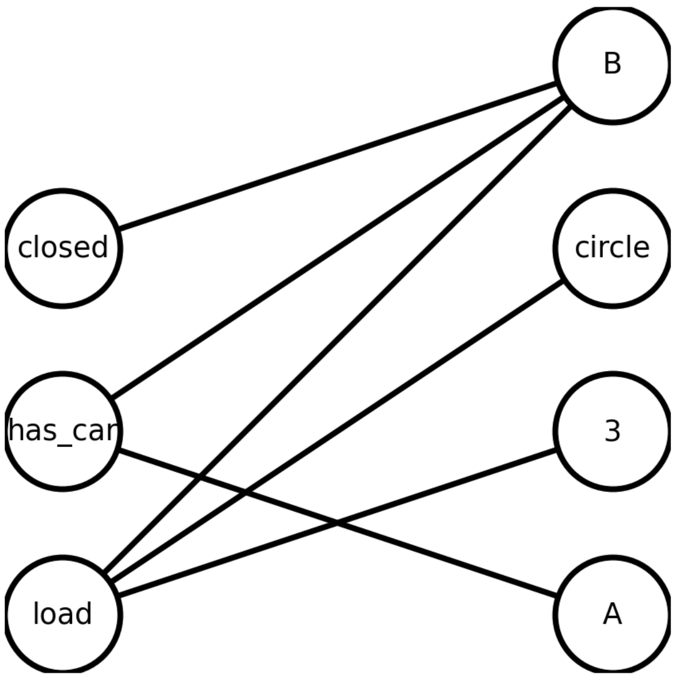
The first 11 positions are used to encode predicates: $[eastbound, closed, double, has\_car, jagged, load, long, open\_car, shape, short, wheels]$. Each predicate is assigned a unique position, with a truth value represented by either 1 or 0.

Next, four positions are assigned to data types: $[\#int, \#shape, car, train]$ and their presence or absence is represented by 1 or 0.

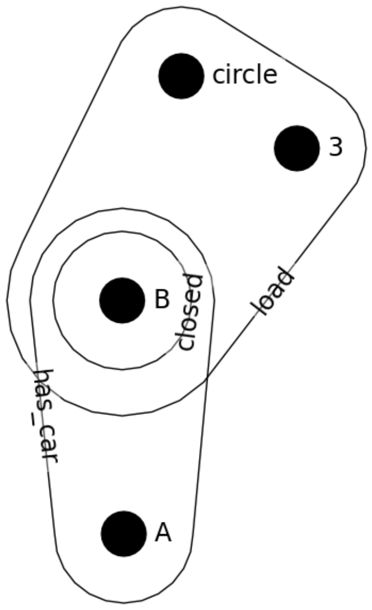
Then seven more positions represent constants such as $[circle, ellipse, hexagon, null, rectangle, triangle, u\_shape]$ values using the same binary coding.

The last position is reserved for numerical values, such as those in predicates like "load" and "wheels".

These three components - predicates, types and constants - serve as templates for structuring the final feature vectors, with the overall size determined by the need



(a) bipartite graph



(b) hypergraph

Figure 3.3: Figures 3.3a and 3.3b show equivalent sets of relations: $closed(B), has\_car(A, B)$, and $load(B, circle, 3)$. Figure 3.3a shows these relations as a bipartite graph, while Figure 3.3b shows the same relations using a hypergraph.

to represent all elements.

For example, the predicate "closed" is represented by a one-hot encoding, where the second position of a 23-dimensional vector contains a 1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0, 0.0]. The 1 in the second position reflects the assignment of "closed" to that position in the vector.

Similarly, the constant 2 is represented by the vector [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2.0], where the 12th position represents the data type "#int" (with a 1) and the last position contains the value 2.

For the variable A, which is of type "train", the 15th position in its vector is set to 1, following the same logic. The first 11 positions are reserved for predicates, the next 4 for types, and the 15th position marks the type "train".

For the constant "circle" the vector looks like [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0.0]. The 13th position marks the "#shape" type, while the first position after the first 15 slots represents "circle". This shows that "circle" is of type "#shape" and has its own value.

The feature vectors for the hypergraph associated with $eastbound(west6)$ are presented in 3.1. Appendix C provides an example of the feature vector corresponding to each term and predicate found in the trains dataset.
$closed : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$

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 : $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0.]$
$circle$: $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0.]$
$rectangle$: $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0.]$
$triangle$: $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0.]$
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 : $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3.]$

In summary, the procedure starts with the bottom clause of $eastbound(west6)$, as shown in figure 3.4. Using this example as input, we generate the bottom clause shown in b). This clause is then transformed into a hypergraph, along with its feature vectors, resulting in the Bottom HyperGraph. Finally, we use this Bottom-HyperGraph to train a HGNN. After processing the hypergraph structure with the HGNN, a readout function is applied to aggregate the learned representations from the nodes and hyperedges. Finally, an MLP is used to classify the output using the rich embeddings learned from the hypergraph during the training process.

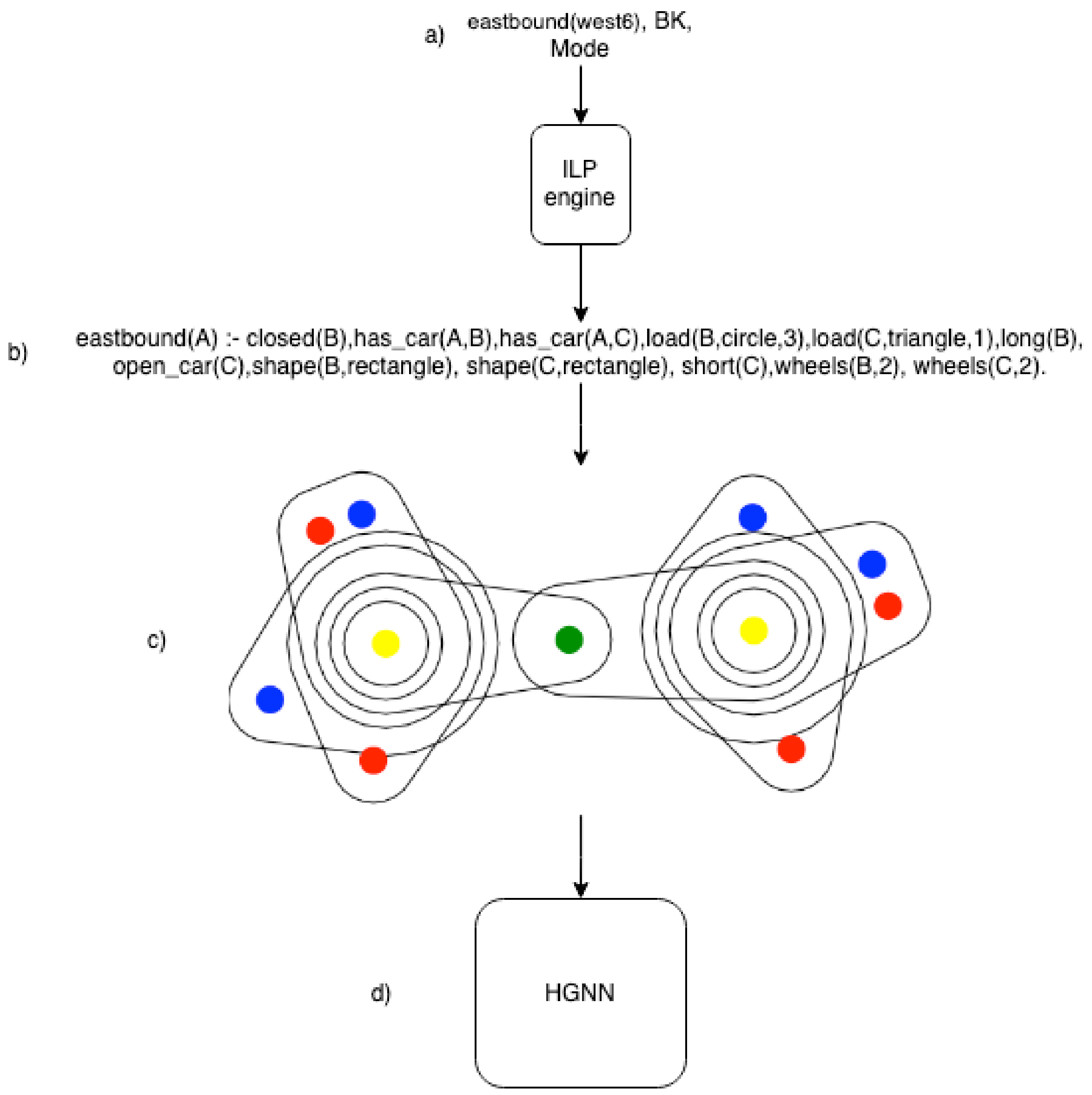

a)
eastbound(west6), BK, Mode
ILP engine
b)
eastbound(A) :- closed(B),has_car(A,B),has_car(A,C),load(B,circle,3),load(C,triangle,1),long(B), open_car(C),shape(B,rectangle), shape(C,rectangle), short(C),wheels(B,2), wheels(C,2).
c)
d)
HGNN

Figure 3.4: Bottom Clause Hypergraph Neural Network: a) the process starts with the background knowledge, and mode declarations (shown here for example $eastbound(west6)$ of the *trains* classification problem; b) the generation of the bottom clause for the example $eastbound(west6)$; c) the corresponding hypergraph (after removing the head of the bottom clause). The colors of the nodes indicate their type: green for *train*, yellow for *car*, red for *shape*, and blue for *int*; d) the hypergraph is converted into feature vectors for training with HGNN.

Chapter 4

Bibliographical review

HetSAGE [2] and BotGNN [9] are graph-based methods designed for heterogeneous and bipartite graph structures respectively. HetSAGE addresses the challenge of different node and edge types by applying a unified sampling strategy, similar to GraphSAGE [60], to generate subgraphs that support node classification. BotGNN incorporates symbolic domain knowledge into GNNs by converting ground clauses into ground graphs, which improves the network's ability to learn from relational data.

Non-graph-based methods such as CILP++ [23] focus on transforming first-order logic into propositional logic to allow neural networks to process relational data. The CILP++ system, derived from C-IL2P, uses Bottom Clause Propositionalisation (BCP) to simplify feature extraction and vector mapping, thereby improving relational learning.

This chapter explores these methods, presenting their techniques and contributions to neuro-symbolic learning. The following sections examine each method, detailing its mechanisms and implications for machine learning.

4.1 Related work

This chapter examines related work, including graph-based methods such as HetSage and BotGNN, and non-graph-based methods such as CILP++ and BCP.

4.1.1 BotGNN

BotGNN [9] is a novel approach that integrates GNNs with domain knowledge using the framework of Mode-Directed Inverse Entailment (MDIE) [41], derived from ILP. This method allows multi-relational background knowledge to be incorporated into GNNs, improving performance on complex graph-based tasks such as molecular graph classification.

MDIE is a technique from ILP that constructs logical rules by inverting entailment. Given a data instance e and background knowledge B, MDIE identifies a specific logical formula $\perp_B(e)$, which captures all the relevant information from B related to e. In BotGNN, this logical formula is represented as a bottom-graph. This graph includes labeled vertices and edges that encode the domain knowledge.

The transformation of knowledge into a graph form suitable for GNNs is an important step. Each graph $G = (V, E, \sigma, \psi, \epsilon)$ consists of

- V: a set of vertices (nodes),
- E: a set of edges connecting the vertices,
- σ: a neighbourhood function (often derived from the edges),
- ψ: a vertex labelling function that assigns feature vectors to vertices,
- ϵ: an edge labelling function.

After several iterations, a final vector embedding of the graph, $\mathbf{X}_G \in \mathbb{R}^d$, is generated for downstream tasks such as classification.

The bottom graph is constructed by transforming the logical formula $\perp_B(e)$ into a graph. This transformation allows the GNN to use both the structural information of the graph and the multi-relational domain knowledge. By incorporating this bottom-graph, BotGNN enhances the GNN's ability to reason over structured data, thus improving its predictive performance.

BotGNN was evaluated on several datasets. The results showed that BotGNN outperformed:

- **Standard GNNs:** BotGNN performed better than GNNs that did not use background knowledge.
- **Simplified Knowledge Incorporation Methods:** A recently proposed method for incorporating knowledge into GNNs, VEGNN [61], showed lower accuracy compared to BotGNN [9].
- **Multi-Layer Perceptrons (MLPs):** BotGNN also outperformed MLPs using propositionalised features of background knowledge [9].

Key strengths of BotGNN are:

- **Combination of GNNs and ILP:** BotGNN uses the computational power of GNNs together with the symbolic reasoning capabilities of ILP to efficiently handle complex, relational data.

- **General applicability:** BotGNN's method of incorporating domain knowledge is applicable to other fields such as social network analysis, bioinformatics, and more.

BotGNN is a step forward in the integration of domain knowledge into GNNs. By embedding structured background information into the graph architecture via MDIE, it allows GNNs to achieve higher performance on complex tasks involving relational reasoning. This combination of the computational efficiency of GNNs with the expressiveness of ILP makes BotGNN a powerful tool for a wide range of applications in machine learning and knowledge representation.

4.1.2 HetSAGE

HetSAGE [2] is a GNN architecture designed to handle the complexity of heterogeneous graphs, where nodes and edges can represent different types of entities and relationships. Such graphs are commonly found in real-world datasets such as social networks, knowledge graphs, and recommendation systems. By integrating concepts from both neuro-symbolic learning and GNNs, HetSAGE enhances the model's ability to process and learn from multi-relational data, allowing for more nuanced representation and inference in complex relational structures.

Traditional GNN architectures, such as GraphSAGE, have been successful in homogeneous graph settings where all nodes and edges are of the same type. However, they often face challenges when applied to heterogeneous graphs, which involve a variety of node and edge types, each representing different types of relationships. This complexity makes it difficult for traditional models to effectively capture and represent the multi-relational information that is critical for tasks such as classification and link prediction.

To overcome these limitations, HetSAGE uses the logical reasoning provided by neuro-symbolic learning. This hybrid approach combines the symbolic reasoning strengths of logic-based systems with the statistical learning capabilities of GNNs. In doing so, HetSAGE enhances the representation and learning processes, enabling the model to better handle the complexity of heterogeneous graphs.

The HetSAGE architecture builds on the sampling and aggregation techniques introduced in GraphSAGE [60], but adapts them to heterogeneous graph structures. The following key components of the HetSAGE framework allow it to efficiently process such graphs:

To represent knowledge within these heterogeneous graphs, HetSAGE uses a grounded graph construction approach. Knowledge is first represented through logic programming and then transformed into a graph structure:

- Terms with arity 0 are mapped to graph nodes.

- Relations with arity 1 and 2 are mapped to node attributes and edges respectively.
- For relations with higher arity, a technique known as reification is used, where complex relations are represented as additional nodes and edges.

This transformation ensures that the complex relationships within the graph are effectively captured, providing the basis for further processing by HetSAGE.

Similar to GraphSAGE, HetSAGE uses a sample and aggregate approach. However, in heterogeneous graphs, nodes may be connected by multiple types of edges, each representing different relationships. HetSAGE addresses this by performing a two-step aggregation process:

- **Neighbour-Level Aggregation:** For each target node, HetSAGE samples a fixed number of neighbours of the same edge type, capturing the specific relational structure. For example, in a recommendation system, different edge types may represent product purchases versus views, and these are treated separately. The model applies a neighbour aggregation function, such as mean or max-pooling, to compute aggregated features from these sampled neighbours.
- **Edge-Level Aggregation:** After obtaining the aggregated features for each edge type, HetSAGE combines them using an edge-level aggregation function. This step incorporates information from multiple edge types, allowing the model to integrate the various relationships that a node may have. The result is a richer node embedding that captures the heterogeneity of the graph structure.

By combining logic-based graph construction with this two-step aggregation strategy, HetSAGE efficiently captures and processes the diverse relationships inherent in heterogeneous graphs.

To efficiently handle the size and complexity of large heterogeneous graphs, HetSAGE uses a node-centric sampling strategy. Instead of processing the entire graph, the model focuses on subgraphs centred around the target node. These subgraphs are constructed by n-hop neighbourhoods, where neighbours within a certain distance (e.g. one, two or n hops) from the target node are included in the subgraph. This localised approach allows HetSAGE to efficiently perform classification and other tasks without the computational burden of processing the entire graph. However, a limitation of HetSAGE is that it requires binary relationships, such as those found in graphs, and cannot handle higher order relationships, which may limit its application in scenarios where multi-way interactions or more complex relationships are required. Figure 4.1 illustrates the sampling method used in the HetSAGE

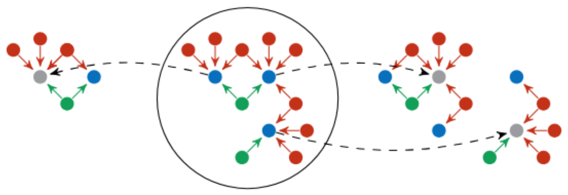


Figure 4.1: Illustration of the two-hop sampling method utilized in the HetSAGE [2].

model. This two-hop sampling technique effectively collects relevant nodes in the neighborhood, providing essential context for making predictions.

- Sampling Strategy: The graph is structured with multiple types of nodes and directed edges (relationships), demonstrating how HetSAGE samples a fixed number of neighbors of each type to maintain computational efficiency.
- Neighbor Representation: Different colored nodes represent different product categories, while arrows indicate directed relationships, focusing on how products influence each other.

In summary, HetSAGE combines the representational learning capabilities of GNNs with symbolic reasoning to enable efficient processing of complex, multi-relational data. Its two-level aggregation framework and node-centric subgraph sampling strategy allow it to capture rich, heterogeneous relationships, making it a powerful tool for a wide range of real-world applications. However, a limitation of HetSAGE is that it requires binary relationships, such as those found in graphs, and cannot handle higher-order relationships, limiting its ability to model more complex multi-way interactions.

4.1.3 CILP++

First-Order Logic (FOL) is a powerful formalism for expressing complex relationships and quantifying variables. However, while FOL excels in knowledge representation and reasoning, traditional neural networks, which are effective in pattern recognition tasks, typically operate at a propositional level. This means that they deal with concrete instances rather than quantifying variables, which limits their ability to generalise across logical structures involving variables and relations. To bridge this gap, propositionalisation techniques have been developed in machine learning. These techniques convert FOL into propositional logic, which is easier for neural networks to handle. By transforming sets of first-order logic rules or clauses into propositional logic, these techniques allow the structure and relationships inherent in FOL to be processed by neural learning algorithms, which are inherently propositional.

The CILP++ [23] system represents an advanced step in the field of neural-symbolic integration by combining neural networks with inductive logic programming (ILP), allowing a seamless blend of learning and reasoning from symbolic knowledge. The fundamental goal of CILP++ is to enable efficient relational learning by integrating first-order logic into neural networks. This allows the system to learn relational knowledge and first-order logic rules from examples, with applications ranging from graph mining to link analysis in social networks.

CILP++ builds on the earlier C-IL2P [25] system, which used background knowledge in propositional logic programs to initialise a neural network. A notable innovation in CILP++ is the use of Bottom Clause Propositionalisation (BCP). BCP transforms first-order logic rules into propositional representations by focusing on the bottom clause, a logical structure used in ILP to define a lower bound on the search space for hypotheses. This bottom clause serves as the basis for extracting relevant features that can be fed into a neural network. BCP simplifies the feature extraction process by mapping bottom clauses to numerical vectors, allowing the seamless integration of logical reasoning into neural models.

To illustrate, consider the logical relationship between individuals in a family environment. For example, in determining whether A is the mother-in-law of B, CILP++ would generate a bottom clause that encodes relationships such as "A is the parent of C, and C is married to B". Similarly, negative relationships are encoded, such as "A cannot be B's mother-in-law if A is already married to someone else". These lower clauses are then propositionalised, which means that each predicate is assigned a position in a vector. This transformation allows the neural network to process logical rules numerically, facilitating tasks such as classification.

The first rule states that A is considered the mother-in-law of B if A is the parent of C, and C is the wife of B. In other words, A becomes the mother-in-law when her child, C, is married to B as the wife. The second rule states that A cannot be the mother-in-law of B if A is a wife herself, meaning that if A is married to someone (referred to as C), she cannot simultaneously be considered the mother-in-law of B. The following formula will produce a three-dimensional vector for the predicates "motherInLaw(A,B)" and "wife(C,B)" in this order:

```
{
    motherInLaw(A,B) :- parent(A,C), wife(C,B);
    ¬motherInLaw(A,B) :- wife(A,C)
}
```

The first clause maps to the vector (1,1,0), with the target output being 1. In contrast, the second clause maps to the vector (0,0,1), with the target output being -1. A neural network classifier for the target predicate is then trained in the conventional way.

One of the key challenges in neural symbolic learning is the extraction of knowledge from neural networks. Although neural networks are powerful in learning complex patterns, they are often considered "black box" models due to their lack of transparency. CILP++ addresses this challenge by incorporating BCP, which allows the systematic transformation of logical structures into neural representations. As a result, CILP++ can perform relational learning tasks with accuracy comparable to traditional ILP systems such as Aleph, while often offering better runtime performance.

The propositionalisation approach adopted by CILP++ involves converting relational learning tasks into simpler attribute-value learning tasks, making it easier for neural networks to process and classify complex logical relationships. Empirical results from a variety of datasets show that CILP++ achieves competitive accuracy with existing ILP systems such as Aleph, and outperforms traditional propositionalisation methods such as RSD when combined with neural networks.

A major enhancement to CILP++ is the incorporation of Minimum Redundancy Maximum Relevance (mRMR), a statistical feature selection method that reduces the number of features while maintaining model performance. By eliminating redundant features, mRMR helps CILP++ handle large relational datasets more efficiently. This extension allows the system to perform feature selection that reduces the feature space by over 90% without significant loss of accuracy, making it more scalable for real-world applications.

In summary, CILP++ exemplifies the growing potential of neural symbolic learning systems. By effectively combining inductive logic programming with the powerful learning capabilities of neural networks, it provides a flexible and efficient method for relational learning in complex domains. Through innovative use of propositionalisation and feature selection, CILP++ pushes the boundaries of what can be achieved with neural-symbolic systems, enabling them to learn and reason with logical knowledge in ways that were previously infeasible. This makes CILP++ a highly valuable tool in areas such as knowledge extraction, social network analysis and graph mining.

Chapter 5

Results and Discussions

This chapter evaluates the performance of our proposed method, BHGNN, which uses an HGNN to represent logical programs as hypergraphs. We evaluate BHGNN through tenfold cross-validation on different datasets and compare its performance with several related approaches, including HGNN, GNN, BotGNN [9], CILP++ [23], and Aleph [62].

Our implementation uses PyTorch Geometric, a library for geometric deep learning. The evaluation aims to determine how effectively BHGNN captures relationships within datasets, including biological data such as mutagenesis [63], carcinogenesis [64] , and Alzheimer's datasets [65].

The experimental setup includes convolutional ARMA filters [66] for GNN and optimisation with the Adam optimiser [67]. Bottom clauses are generated using Aleph, a symbolic ILP system.

To classify graphs and hypergraphs, the aggregation method combines mean, sum and max operations to capture different aspects of graph structure.

Our analysis uses statistical methods, including the Wilcoxon signed-rank test [68], to compare the performance of BHGNN with other models. The results show that BHGNN outperforms traditional HGNN and GNN approaches.

This chapter provides a detailed examination of the capabilities of BHGNN, and the comparative performance of different methods.

5.1 Results

To assess the performance of our method using a Bottom-HyperGraph Neural Network (BHGNN), we performed a tenfold cross-validation on several datasets. In each iteration, the dataset was divided into ten parts: eight parts were used for training, one part for validation and one part for testing. This process was repeated ten times, with each part used once as a test set. The average performance across all iterations provides an estimate of the generalisation ability of the model. The HGNN was implemented using PyTorch Geometric[1].

We compared our approach, BHGNN, with pure neural methods, including HGNN and GNN.

We also evaluated BHGNN against neural symbolic and purely symbolic approaches such as BotGNN, CILP++ and other related work.

Graphs were processed using GNN with Convolutional ARMA Filters [66], and both HGNN and GNN were optimised with the Adam Optimiser [67]. The architectures for both HGNN and GNN comprised three layers with 128 hidden units per layer and were trained for 500 epochs with a early stopping criterion triggered after 50 epochs if the validation loss did not improve. The learning rate was set to 0.0001 for all models. We followed the same procedure for CILP++, using a similar architecture and training setup.

The datasets and background knowledge were expressed in Prolog and preprocessed using Aleph [62], an ILP system, to generate bottom clauses with a maximum clause depth of five. All datasets were binary classification datasets, i.e. they contained two possible class labels. For GNN we used the mutagenesis dataset from [69]. In cases where relational data was used, we converted the relational structure into a bipartite graph following the procedure described in section 2.3.1.

Lower clause depths generate smaller and less specific rules, while increasing the depth makes the rules highly specific to the selected examples. This trade-off was evaluated empirically, and a clause depth of five was found to give good results.

Each data set was designed with a specific objective in mind:

- The Mutagenesis dataset classifies molecules as mutagenic or non-mutagenic.
- The Carcinogenesis dataset classifies the carcinogenic potential of a drug molecule.
- The Alzheimer's datasets target properties critical for the treatment of Alzheimer's disease:
  - The Choline dataset aims to maximise acetylcholinesterase inhibition.
  - The Scopolamine dataset focuses on scopolamine reversal.
  - The Toxic dataset compares two drugs to classify which is less toxic.
  - The Amine dataset aims to maximise amine reuptake inhibition.

More detailed descriptions of these datasets can be found in the appendix A.

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

5.2 Experimental procedure

The experiments were carried out as follows:

1. Data Preparation: Each dataset was pre-processed using Aleph to generate bottom clauses for each example with a maximum clause depth of 5. The clauses were encoded as features for each example instance, forming the basis of the hypergraph structure in BHGNN.

2. Cross-Validation Iterations: We ran a tenfold cross-validation on each dataset. In each iteration, the dataset was randomly divided into ten equal parts. For each fold, one part was used as the test set, one as the validation set, and the remaining eight parts were used for training. We ensured that each example appeared exactly once in the test set.

3. Model Training and Evaluation: We built a Bottom-HyperGraph for each training fold and trained a HGNN with the pre-processed hypergraph data. The models were trained using the Adam optimiser with a learning rate of 0.0001, a batch size of 32 and a total of 500 epochs per fold. Early stop was applied with a patience of 50 epochs if the validation loss did not improve. For evaluation, we recorded the accuracy on the test set for each iteration. Hyperparameters such as number of layers (3), hidden units (128 per layer), and ARMA filter for GNN were kept consistent across experiments.

4. Result Aggregation: After completing the tenfold cross-validation for each dataset, we calculated the mean accuracy and standard deviation across all folds. The final performance metrics for each model were based on these aggregated results.

5.3 Statistical Analysis

To determine the statistical significance of the results, we performed the Wilcoxon signed-rank test [68] to compare the relative performance of BHGNN, HGNN and GNN across identical folds. The null hypothesis was that there was no significant difference in performance between the models. Based on the analysis, BHGNN outperformed both HGNN and GNN on all datasets, providing sufficient evidence to reject the null hypothesis and demonstrate a statistically significant improvement in performance.

The average accuracy and standard deviation for each dataset are reported in Table 5.1. Notably, HGNN and GNN does not use bottom clauses in the hypergraph structure, while BHGNN use bottom clauses, therefore symbolic knowledge.

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

Table 5.1: 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	**BotGNN**	**HetSAGE**	**CILP++**	**Aleph**
Muta42	75 ± 13.4	73 ± 16.4	70 ± 14.6	73 ± 18.3	**76 ± 14.5**
Muta188	**90 ± 8.9**	88 ± 9.9	86 ± 10.9	87 ± 7.8	85 ± 9.1
Carcinogenesis	**64 ± 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	**60 ± 4.7**
Amine	53 ± 8.2	55 ± 8.7	52 ± 8.6	**68 ± 6.7**	62 ± 7.9
Toxic	55 ± 8.6	62 ± 8.3	54 ± 9.6	**72 ± 6.1**	65 ± 8.2
Choline	54 ± 7.9	58 ± 9.2	53 ± 10.2	53 ± 4.3	**59 ± 7.7**

Table 5.2: 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.

The results of the Wilcoxon statistical test on BHGNN and BotGNN showed no significant differences between the two models, suggesting that their performance is statistically comparable. The results of Aleph and CILP++ were competitive, demonstrating their effectiveness in handling relational learning tasks. The differences in performance between BHGNN and Aleph and between BHGNN and CILP++ were also not statistically significant, confirming the competitiveness of these methods. The detailed results of the statistical comparisons are shown in Table 5.2.

A potential limitation of our approach is the reliance on the ILP engine to generate the bottom clause. Any problems or inefficiencies with the ILP engine can have a direct impact on the effectiveness of the method.

In BHGNN each constant in the bottom clause is represented as a separate node in the hypergraph. In this section, we explain this design choice and propose a hypothesis for its impact on performance.

In hypergraphs, the node representation determines how relationships between entities are modelled. In clauses, constants represent entities or values, and assigning one node per constant maps these entities directly into the hypergraph structure.

By representing each constant as a separate node, BHGNN preserves the structure of the logical relationships within the bottom clause, ensuring that the interactions between constants are clearly represented in the hypergraph.

Another hypothesis for the performance of BHGNN with one node per constant is that this approach may increase the representational power of the model. Specifically, more nodes may lead to varied node representations during the message-passing process in the HGNN layers.

In graph-based models, the representation of a node depends on its neighbours. If the same constant is represented as multiple nodes in bottom clauses, each instance of the constant is connected to different neighbours. This variability in neighbouring nodes leads to context-dependent node representations during the message-passing process.

- **Context-dependent representations:** With more nodes for each constant, BHGNN allows these nodes to develop different embeddings based on their neighbourhood. For example, in the mutagenesis dataset, the same constant representing an atom may have different neighbours in different bottom clauses or within the same clause. As the message-passing algorithm aggregates information from neighbours, each instance of the atom develops a representation that reflects its specific context.

- **Enhanced Expressiveness:** With separate nodes for constants, BHGNN captures more detailed patterns in the data. If constants were merged into fewer nodes, the model might miss these context-dependent distinctions, reducing its ability to model relationships.

This hypothesis is supported by the experimental results, where BHGNN outperformed other models that did not use this node representation.

Additionally, one possible reason why our neuro-symbolic model may have outperformed purely neural models could be the way our data is constructed, in particular through the use of the bottom clause. The construction of the bottom clause likely infuses symbolic knowledge directly into the data, providing a structured foundation that supports symbolic reasoning. This inherent structure may act as an inductive bias, guiding the model to capture underlying relationships more effectively, leading to improved performance over purely neural approaches.
In summary, the representation of each constant in the lower clause as a separate node was motivated by the need to capture the relational structure of the data. Empirical results showed that this approach improved performance, possibly due to the increased representational power of context-sensitive node embeddings during the message-passing process. The different node representations likely contributed to the accuracy of the model.

Chapter 6

Conclusions

The main goal of this work was to investigate different forms of graph representation and their effect on graph-based learning. In particular, graph neural networks are expected to handle relational data better than standard neural networks due to their representational structure. We introduced a method that allows HGNNs to benefit from a symbolic relational learning environment by using hypergraphs. Our evaluation compared the performance of hypergraph neural networks, HGNNs, and other neurosymbolic and symbolic approaches to relational learning on datasets containing more than binary relations.

This work outlines a method for integrating domain expertise into hypergraph networks, using saturation examples to create the most specific clause as the initial embedding for training. Our approach is novel in that it incorporates relational knowledge of arbitrary arity into hypergraph networks. Experimental results show the potential of this approach and suggest the need for further testing on larger datasets, considering variations and optimisations in graph learning methods and algorithms.

In our future research, we intend to go beyond the use of feature vectors representing only the binary values 0 (absence) and 1 (presence). Specifically, we will investigate the incorporation of hypergraph information, similar to what TDIDF does with text, and what node2vec [70] and struc2vec [71] do with graphs.

While symbolic machine learning produces a logical theory as a learning result, GNNs do not. A logical theory is interpretable and provides explanations for results through logical proofs. The explicability of hypergraph networks remains an area for further investigation. A hypergraph network initialised with prior knowledge should be easier to interpret through queries than one initialised at random. Our goal is to post-process the learned hypergraph into a logical format.

We plan to explore variations of HGNNs, as mentioned in [72], to extend graph optimisation efforts and to assess the explainability of decision processes in hypergraph classification. Based on our findings, we also aim to adapt techniques such as those in [73] for application to hypergraphs. In addition, as part of our future work, we will analyse the sparsity of feature vectors within hypergraph networks and investigate how this sparsity affects learning outcomes and model efficiency. Understanding these dynamics could lead to improvements in both performance and scalability, especially for large or high-dimensional datasets.

References

[1] MICHALSKI, R. S. “Pattern recognition as rule-guided inductive inference”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, , n. 4, pp. 349–361, 1980.

[2] JANKOVICS, V., ORTIZ, M. G., ALONSO, E. “HetSAGE: Heterogenous Graph Neural Network for Relational Learning (Student Abstract)”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 35, pp. 15803–15804, 2021.

[3] WONG, A. K., ZHOU, P.-Y., BUTT, Z. A. “Pattern discovery and disentanglement on relational datasets”, *Scientific Reports*, v. 11, n. 1, pp. 5688, 2021.

[4] FEY, M., HU, W., HUANG, K., et al. “Relational Deep Learning: Graph Representation Learning on Relational Databases”, *arXiv preprint arXiv:2312.04615*, 2023.

[5] ROBINSON, J., RANJAN, R., HU, W., et al. “RelBench: A Benchmark for Deep Learning on Relational Databases”. 2024. Disponível em: <`https://arxiv.org/abs/2407.20060`>.

[6] HAMILTON, W. L. *Graph representation learning.* Morgan & Claypool Publishers, 2020.

[7] GORI, M., MONFARDINI, G., SCARSELLI, F. “A new model for learning in graph domains”. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, v. 2, pp. 729–734. IEEE, 2005.

[8] KIPF, T. N., WELLING, M. “Semi-supervised classification with graph convolutional networks”, *arXiv preprint arXiv:1609.02907*, 2016.

[9] DASH, T., SRINIVASAN, A., BASKAR, A. “Inclusion of domain-knowledge into gnns using mode-directed inverse entailment”, *Machine Learning*, pp. 1–49, 2022.

[10] BONACICH, P., HOLDREN, A. C., JOHNSTON, M. "Hyper-edges and multidimensional centrality", *Social networks*, v. 26, n. 3, pp. 189–203, 2004.

[11] KLAMT, S., HAUS, U.-U., THEIS, F. "Hypergraphs and cellular networks", *PLoS computational biology*, v. 5, n. 5, pp. e1000385, 2009.

[12] WOLF, M. M., KLINVEX, A. M., DUNLAVY, D. M. "Advantages to modeling relational data using hypergraphs versus graphs". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7. IEEE, 2016.

[13] FAGIN, R. "Degrees of acyclicity for hypergraphs and relational database schemes", *Journal of the ACM (JACM)*, v. 30, n. 3, pp. 514–550, 1983.

[14] SHI, H., ZHANG, Y., ZHANG, Z., et al. "Hypergraph-Induced Convolutional Networks for Visual Classification", *IEEE Transactions on Neural Networks and Learning Systems*, v. 30, n. 10, pp. 2963–2972, 2019. doi: 10.1109/TNNLS.2018.2869747.

[15] FENG, S., HEATH, E., JEFFERSON, B., et al. "Hypergraph models of biological networks to identify genes critical to pathogenic viral response", *BMC bioinformatics*, v. 22, n. 1, pp. 1–21, 2021.

[16] CONTISCIANI, M., BATTISTON, F., DE BACCO, C. "Inference of hyperedges and overlapping communities in hypergraphs", *Nature communications*, v. 13, n. 1, pp. 7229, 2022.

[17] WANG, H., ZHOU, W., WEN, J., et al. "Multiple hypergraph convolutional network social recommendation using dual contrastive learning", *Data Mining and Knowledge Discovery*, pp. 1–29, 2024.

[18] YADATI, N., NIMISHAKAVI, M., YADAV, P., et al. "Hypergcn: A new method for training graph convolutional networks on hypergraphs", *Advances in neural information processing systems*, v. 32, 2019.

[19] BAI, S., ZHANG, F., TORR, P. H. "Hypergraph convolution and hypergraph attention", *Pattern Recognition*, v. 110, pp. 107637, 2021.

[20] GREWE, N. "A generic reification strategy for n-ary relations in DL". In: *OBML 2010 Workshop Proceedings*, 2010.

[21] "What is inductive logic programming?" In: *Foundations of Inductive Logic Programming*, pp. 162–177, Berlin, Heidelberg, Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-69049-8. doi: 10.1007/3-540-62927-0_9. Disponível em: <`https://doi.org/10.1007/3-540-62927-0_9`>.

[22] GANDARELA DE SOUZA, J. P., ZAVERUCHA, G., D'AVILA GARCEZ, A. S. "Hypergraph Neural Networks with Logic Clauses". In: *IEEE World Congress on Computational Intelligence (IEEE WCCI IJCNN 2024), Yokohama, Japan, 30 June - 5 July 2024*. IEEE, 2024.

[23] FRANÇA, M. V., ZAVERUCHA, G., D'AVILA GARCEZ, A. S. "Fast relational learning using bottom clause propositionalization with artificial neural networks", *Machine learning*, v. 94, pp. 81–104, 2014.

[24] FRANÇA, M. V. M., ZAVERUCHA, G., D'AVILA GARCEZ, A. S. "Neural Relational Learning Through Semi-Propositionalization of Bottom Clauses". In: *AAAI Spring Symposia*, 2015. Disponível em: <`https://api.semanticscholar.org/CorpusID:51735042`>.

[25] AVILA GARCEZ, A. S., ZAVERUCHA, G. "The connectionist inductive learning and logic programming system", *Applied Intelligence*, v. 11, pp. 59–77, 1999.

[26] D'AVILA GARCEZ, A., LAMB, L. C. "Neurosymbolic AI: the 3rd wave", *Artif. Intell. Rev.*, v. 56, n. 11, pp. 12387–12406, 2023. doi: 10.1007/S10462-023-10448-W.

[27] LAMB, L. C., D'AVILA GARCEZ, A. S., GORI, M., et al. "Graph Neural Networks Meet Neural-Symbolic Computing: A Survey and Perspective". In: Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 4877–4884. ijcai.org, 2020. doi: 10.24963/IJCAI.2020/679.

[28] BESOLD, T. R., D'AVILA GARCEZ, A. S., BADER, S., et al. "Neural-Symbolic Learning and Reasoning: A Survey and Interpretation". In: Hitzler, P., Sarker, M. K. (Eds.), *Neuro-Symbolic Artificial Intelligence: The State of the Art*, v. 342, *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 1–51, 2021. doi: 10.3233/FAIA210348.

[29] D'AVILA GARCEZ, A. S., BRODA, K., GABBAY, D. M. *Neural-symbolic learning systems - foundations and applications*. Perspectives in neural computing. Springer, 2002. ISBN: 978-1-85233-512-0. doi: 10.1007/978-1-4471-0211-3.

[30] POSPESEL, H., LYCAN, W. G. *Propositional Logic: Introduction to Logic*. Prentice Hall, 1998.

[31] GALLO, C. *An Introduction to Propositional Logic and Set Theory*. Understanding Calculus. Amazon Digital Services LLC - Kdp, 2021. ISBN: 9798593189264. Disponível em: <`https://books.google.ch/books?id=ZZUpzgEACAAJ`>.

[32] MAGNUS, P., BCCAMPUS. *Forall X: An Introduction to Formal Logic*. BC Campus open textbooks. BCcampus, 2012. Disponível em: <`https://books.google.ch/books?id=naF-0AEACAAJ`>.

[33] DUBOC, A. L. D. C. L. *UTILIZANDO A CLÁUSULA MAIS ESPECÍFICA E DECLARAÇÃO DE MODOS NA REVISÃO DE TEORIAS DE PRIMEIRA-ORDEM A PARTIR DE EXEMPLOS*. Tese de Mestrado, UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2008.

[34] HURLEY, P. *A concise introduction to logic*. Nelson Education, 2011.

[35] BARWISE, J., ETCHEMENDY, J. "The language of first-order logic". 1991. Disponível em: <`https://api.semanticscholar.org/CorpusID:61094033`>.

[36] BARWISE, J., ETCHEMENDY, J., ALLWEIN, G., et al. *Language, Proof and Logic*. CSLI Publications, 2002. ISBN: 9781575863740. Disponível em: <`https://books.google.ch/books?id=WZj5nQEACAAJ`>.

[37] MUGGLETON, S. "Inductive logic programming", *New generation computing*, v. 8, pp. 295–318, 1991.

[38] MUGGLETON, S., DE RAEDT, L. "Inductive logic programming: Theory and methods", *The Journal of Logic Programming*, v. 19, pp. 629–679, 1994.

[39] QUINLAN, J. R. "Learning logical definitions from relations", *Machine learning*, v. 5, pp. 239–266, 1990.

[40] MUGGLETON, S. H., FENG, C., OTHERS. *Efficient induction of logic programs*. Turing Institute London, UK, 1990.

[41] MUGGLETON, S. "Inverse entailment and Progol", *New generation computing*, v. 13, pp. 245–286, 1995.

[42] BONDY, J. A., MURTY, U. S. R. *Graph theory*. Springer Publishing Company, Incorporated, 2008.

[43] DIESTEL, R. *Graph theory*. Springer (print edition); Reinhard Diestel (eBooks), 2024.

[44] BONDY, J. A., MURTY, U. S. R., OTHERS. *Graph theory with applications*, v. 290. Macmillan London, 1976.

[45] WEST, D. B., OTHERS. *Introduction to graph theory*, v. 2. Prentice hall Upper Saddle River, 2001.

[46] TRUDEAU, R. J. *Introduction to graph theory.* Courier Corporation, 2013.

[47] HARTSFIELD, N., RINGEL, G. *Pearls in graph theory: a comprehensive introduction.* Courier Corporation, 2003.

[48] GROSS, J. L., YELLEN, J., ANDERSON, M. *Graph theory and its applications.* Chapman and Hall/CRC, 2018.

[49] BRETTO, A. "Hypergraph Theory: An Introduction". 2013. Disponível em: <`https://api.semanticscholar.org/CorpusID:60518907`>.

[50] OUVRARD BRUNET, X. "Hypergraphs: an introduction and review". 02 2020.

[51] DAI, Q., GAO, Y. "Mathematical Foundations of Hypergraph". In: *Hypergraph Computation*, Springer, pp. 19–40, 2023.

[52] DAI, Q., GAO, Y. *Hypergraph Computation.* Springer Nature, 2023.

[53] VOLOSHIN, V. I. *Introduction to graph and hypergraph theory.* Nova Science Publishers, 2013.

[54] WU, L., CUI, P., PEI, J., et al. "Graph neural networks: foundation, frontiers and applications". In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4840–4841, 2022.

[55] DAIGAVANE, A., RAVINDRAN, B., AGGARWAL, G. "Understanding Convolutions on Graphs", *Distill*, 2021. doi: 10.23915/distill.00032. https://distill.pub/2021/understanding-gnns.

[56] SANCHEZ-LENGELING, B., REIF, E., PEARCE, A., et al. "A Gentle Introduction to Graph Neural Networks", *Distill*, 2021. doi: 10.23915/distill. 00033. https://distill.pub/2021/gnn-intro.

[57] HAMILTON, W. L. "Graph Representation Learning", *Synthesis Lectures on Artificial Intelligence and Machine Learning*, v. 14, n. 3, pp. 1–159.

[58] BRETTO, A. "Hypergraph theory", *An introduction. Mathematical Engineering. Cham: Springer*, v. 1, 2013.

[59] MICHIE, D., MUGGLETON, S., PAGE, D., et al. "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.

[60] HAMILTON, W., YING, Z., LESKOVEC, J. "Inductive representation learning on large graphs", *Advances in neural information processing systems*, v. 30, 2017.

[61] DASH, T., SRINIVASAN, A., VIG, L. "Incorporating Symbolic Domain Knowledge into Graph Neural Networks", *CoRR*, v. abs/2010.13900, 2020. Disponível em: <`https://arxiv.org/abs/2010.13900`>.

[62] SRINIVASAN, A. "The aleph manual". 2001.

[63] SRINIVASAN, A., MUGGLETON, S., KING, R. D., et al. "Mutagenesis: ILP experiments in a non-determinate biological domain". In: *Proceedings of the 4th international workshop on inductive logic programming*, v. 237, pp. 217–232. Citeseer, 1994.

[64] SRINIVASAN, A., KING, R. D., MUGGLETON, S. H., et al. "Carcinogenesis predictions using ILP". In: *International Conference on Inductive Logic Programming*, pp. 273–287. Springer, 1997.

[65] KING, R. D., STERNBERG, M. J., SRINIVASAN, A. "Relating chemical activity to structure: an examination of ILP successes", *New Generation Computing*, v. 13, pp. 411–433, 1995.

[66] BIANCHI, F. M., GRATTAROLA, D., LIVI, L., et al. "Graph Neural Networks with Convolutional ARMA Filters", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 1–1, 2021. ISSN: 1939-3539. doi: 10.1109/tpami.2021.3054830.

[67] KINGMA, D. P., BA, J. "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

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

[69] MORRIS, C., KRIEGE, N. M., BAUSE, F., et al. "TUDataset: A collection of benchmark datasets for learning with graphs", *CoRR*, v. abs/2007.08663, 2020. Disponível em: <`https://arxiv.org/abs/2007.08663`>.

[70] GROVER, A., LESKOVEC, J. "node2vec: Scalable Feature Learning for Networks". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[71] RIBEIRO, L. F., SAVERESE, P. H., FIGUEIREDO, D. R. "struc2vec: Learning Node Representations from Structural Identity". In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, p. 385–394, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450348874. doi: 10.1145/3097983.3098061. Disponível em: <`https://doi.org/10.1145/3097983.3098061`>.

[72] GAO, Y., FENG, Y., JI, S., et al. "HGNN+: General Hypergraph Neural Networks", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 45, n. 3, pp. 3181–3199, 2023. doi: 10.1109/TPAMI.2022.3182052.

[73] YING, Z., BOURGEOIS, D., YOU, J., et al. "Gnnexplainer: Generating explanations for graph neural networks", *Advances in neural information processing systems*, v. 32, 2019.

[74] BISHOP, C. M., NASRABADI, N. M. *Pattern recognition and machine learning*, v. 4. Springer, 2006.

[75] RUSSELL, S., NORVIG, P. "Artificial Intelligence: A Modern Approach, 4th, Global ed". 2022.

[76] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep learning*. MIT press, 2016.

Appendix A

Dataset

Table A.1: Dataset Statistics

Dataset	**Positive Examples**	**Negative Examples**	**Total**
Muta188	125	63	188
Muta42	13	29	42
Carcinogenesis	162	136	298
Scopolamine	321	321	642
Amine	343	343	686
Toxic	443	443	886
Choline	663	663	1326

Appendix B

TRAINS DATASET - MODE DECLARATIONS

$$:-modeh(1, eastbound(+train)).$$

$$:-modeb(1, short(+car)).$$

$$:-modeb(1, closed(+car)).$$

$$:-modeb(1, long(+car)).$$

$$:-modeb(1, open\_car(+car)).$$

$$:-modeb(1, double(+car)).$$

$$:-modeb(1, jagged(+car)).$$

$$:-modeb(1, shape(+car, \#shape)).$$

$$:-modeb(1, load(+car, \#shape, \#int)).$$

$$:-modeb(1, wheels(+car, \#int)).$$

$$:-modeb(*, has\_car(+train, -car)).$$

$$:-determination(eastbound/1, short/1).$$

$$:-determination(eastbound/1, closed/1).$$

$$:-determination(eastbound/1, long/1).$$

$$:-determination(eastbound/1, open\_car/1).$$

$$:-determination(eastbound/1, double/1).$$

$$:-determination(eastbound/1, jagged/1).$$

$$:-determination(eastbound/1, shape/2).$$

$$:-determination(eastbound/1, wheels/2).$$

$$:-determination(eastbound/1, has\_car/2).$$

$$:-determination(eastbound/1, load/3).$$

Appendix C

Feature vectors

eastbound : [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]
closed : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.]
double : [0, 0, 1, 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.]
jagged : [0, 0, 0, 0, 1, 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.]
ellipse : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0.]
hexagon : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0.]
nil : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 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.]
triangle : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0.]
u_shape : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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.]

Appendix D

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** $hypergraphs$
```

Appendix E

Artificial Neural Network

Artificial Neural Networks [74–76] are computational graphs of interconnected nodes, which are arranged in layers. In the case of a feed-forward network, connections happen only in one direction, which means, it ends up being a directed acyclic graph with an input layer, hidden layer(s), and output layer. Figure E.1 displays an example of a neural network.

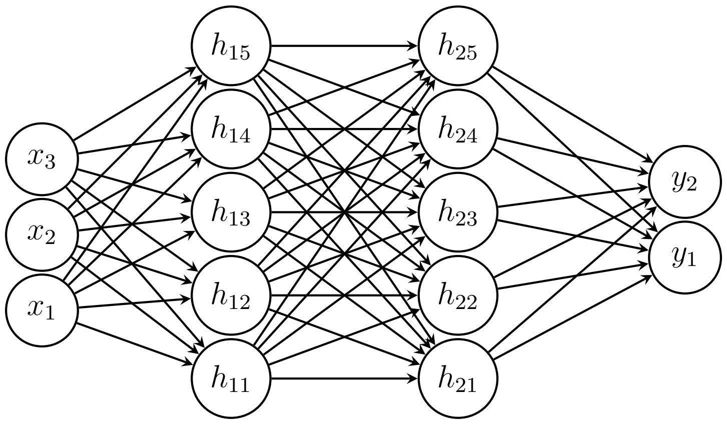


Figure E.1: Architecture of a Feed-forward Neural Network with two hidden layers. The network consists of an input layer with three nodes (x_1, x_2, x_3), two hidden layers with five nodes each, the first hidden layer with nodes (h_{11}, h_{12},h_{13},h_{14},h_{15}), and the second hidden layer with nodes (h_{21}, h_{22},h_{23},h_{24},h_{25}), and an output layer with two nodes (y_1, y_2). Each arrow represents a weighted connection between nodes, where the weights are learned during the training process.

Neural networks can be broken down into a few parts. Foremost, we have the weights and bias, which are the learnable parameters. Subsequently, the activation function introduces non-linearity into the network. Finally, we combine operations as follows:

1. For each layer $l = 1, 2, \ldots, L-1$, calculate the output:

$$Z^l = W^l \cdot A^{l-1} + b^l$$

$$A^l = f(Z^l)$$

2. For the output layer $l = L$, calculate the output:

$$Z^L = W^L \cdot A^{L-1} + b^L$$

$$A^L = f(Z^L)$$

For example, the forward propagation in the neural network represented in Figure E.1 can be expressed mathematically as:

$$\begin{aligned} Z^{(1)} &= W^{(1)}X + b^{(1)} \\ A^{(1)} &= f(Z^{(1)}) \\ Z^{(2)} &= W^{(2)}A^{(1)} + b^{(2)} \\ A^{(2)} &= f(Z^{(2)}) \\ Z^{(3)} &= W^{(3)}A^{(2)} + b^{(3)} \\ Y &= f(Z^{(3)}) \end{aligned}$$

where X is the input vector, $W^{(1)}, W^{(2)}, W^{(3)}$ are the weight matrices, $b^{(1)}, b^{(2)}, b^{(3)}$ are the bias vectors, Z^1, Z^2, Z^3 are the output of layers $1, 2, 3$ before activation, f is the activation function, and Y is the output vector.

To train a neural network, we use the backpropagation algorithm. After the forward pass, we end with the neural network's prediction. We use this prediction to compare with the expected output. Next, from the difference between the predictions and the expectations, it computes the loss and updates the weights and biases of all layers using the backpropagation algorithm.

Let δ^l represent the error term for layer l, $\odot$ the element-wise multiplication, and σ' the derivative of the activation function.

1. Compute the error term for the output layer:

$$\delta^L = \nabla_a Loss \odot f'(Z^L)$$

2. Backpropagate the error to previous layers:

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot f'(Z^l)$$

for $l = L - 1, L - 2, \ldots, 2$

3. Compute the gradients of the weights and biases:

$$\frac{\partial Loss}{\partial W^l} = \delta^l (a^{l-1})^T$$

$$\frac{\partial Loss}{\partial b^l} = \delta^l$$

for $l = L, L-1, \ldots, 2$

4. Update the weights and biases using an optimization algorithm:

$$W^l \to W^l - \eta \frac{\partial Loss}{\partial W^l}$$

$$b^l \to b^l - \eta \frac{\partial Loss}{\partial b^l}$$

where η is the learning rate.

For instance, the backpropagation algorithm in the neural network represented in Figure E.1 can be expressed :

1. Compute the error term for the output layer:

$$\delta^{(3)} = \nabla_{A^{(3)}} Loss \odot f'(Z^{(3)})$$

2. Backpropagate the error to previous layers:

$$\delta^{(2)} = (W^{(3)})^T \delta^{(3)} \odot f'(Z^{(2)})$$

$$\delta^{(1)} = (W^{(2)})^T \delta^{(2)} \odot f'(Z^{(1)})$$

3. Compute the gradients of the weights and biases:

$$\frac{\partial Loss}{\partial W^{(3)}} = \delta^{(3)} (A^{(2)})^T$$

$$\frac{\partial Loss}{\partial b^{(3)}} = \delta^{(3)}$$

$$\frac{\partial Loss}{\partial W^{(2)}} = \delta^{(2)} (A^{(1)})^T$$

$$\frac{\partial Loss}{\partial b^{(2)}} = \delta^{(2)}$$

$$\frac{\partial Loss}{\partial W^{(1)}} = \delta^{(1)} X^T$$

$$\frac{\partial Loss}{\partial b^{(1)}} = \delta^{(1)}$$

4. Update the weights and biases using an optimization algorithm:

$$W^{(1)} \leftarrow W^{(1)} - \eta \frac{\partial Loss}{\partial W^{(1)}}$$

$$b^{(1)} \leftarrow b^{(1)} - \eta \frac{\partial Loss}{\partial b^{(1)}}$$

$$W^{(2)} \leftarrow W^{(2)} - \eta \frac{\partial Loss}{\partial W^{(2)}}$$

$$b^{(2)} \leftarrow b^{(2)} - \eta \frac{\partial Loss}{\partial b^{(2)}}$$

$$W^{(3)} \leftarrow W^{(3)} - \eta \frac{\partial Loss}{\partial W^{(3)}}$$

$$b^{(3)} \leftarrow b^{(3)} - \eta \frac{\partial Loss}{\partial b^{(3)}}$$

   Where η is the learning rate. Repeat these steps for each training example to train the neural network.

This completes one iteration of the backpropagation algorithm. Repeat these steps for multiple iterations or epochs to train the neural network.

Appendix F

Trains dataset: Background knowledge

```
% Type definitions
% Car Definitions
car(car_11).  car(car_12).  car(car_13).  car(car_14).
car(car_21).  car(car_22).  car(car_23).
car(car_31).  car(car_32).  car(car_33).
car(car_41).  car(car_42).  car(car_43).  car(car_44).
car(car_51).  car(car_52).  car(car_53).
car(car_61).  car(car_62).
car(car_71).  car(car_72).  car(car_73).
car(car_81).  car(car_82).
car(car_91).  car(car_92).  car(car_93).  car(car_94).
car(car_101).  car(car_102).

% Shape Definitions
shape(elipse).  shape(hexagon).  shape(rectangle).  shape(u_shaped).
shape(triangle).  shape(circle).  shape(nil).

% Train Definitions
train(east1).  train(east2).  train(east3).  train(east4).  train(east5).
train(west6).  train(west7).  train(west8).  train(west9).  train(west10).

% Eastbound train 1
short(car_12). % 0
closed(car_12). % 1
long(car_11). % 2
long(car_13).
```

```
short(car_14).
open_car(car_11). % 3
open_car(car_13).
open_car(car_14).
shape(car_11,rectangle). % 4,5
shape(car_12,rectangle).
shape(car_13,rectangle).
shape(car_14,rectangle).
load(car_11,rectangle,3). % 6,7,8
load(car_12,triangle,1).
load(car_13,hexagon,1).
load(car_14,circle,1).
wheels(car_11,2).   % 9,10
wheels(car_12,2).
wheels(car_13,3).
wheels(car_14,2).
has_car(east1,car_11). % 11,12
has_car(east1,car_12).
has_car(east1,car_13).
has_car(east1,car_14).

% eastbound train 2
has_car(east2,car_21).
has_car(east2,car_22).
has_car(east2,car_23).
short(car_21).
short(car_22).
short(car_23).
shape(car_21,u_shaped).
shape(car_22,u_shaped).
shape(car_23,rectangle).
open_car(car_21).
open_car(car_22).
closed(car_23).
load(car_21,triangle,1).
load(car_22,rectangle,1).
load(car_23,circle,2).
wheels(car_21,2).
wheels(car_22,2).
wheels(car_23,2).
```

```
% eastbound train 3
has_car(east3,car_31).
has_car(east3,car_32).
has_car(east3,car_33).
short(car_31).
short(car_32).
long(car_33).
shape(car_31,rectangle).
shape(car_32,hexagon).
shape(car_33,rectangle).
open_car(car_31).
closed(car_32).
closed(car_33).
load(car_31,circle,1).
load(car_32,triangle,1).
load(car_33,triangle,1).
wheels(car_31,2).
wheels(car_32,2).
wheels(car_33,3).

% eastbound train 4
has_car(east4,car_41).
has_car(east4,car_42).
has_car(east4,car_43).
has_car(east4,car_44).
short(car_41).
short(car_42).
short(car_43).
short(car_44).
shape(car_41,u_shaped).
shape(car_42,rectangle).
shape(car_43,elipse).
shape(car_44,rectangle).
double(car_42).
open_car(car_41).
open_car(car_42).
closed(car_43).
open_car(car_44).
load(car_41,triangle,1).
load(car_42,triangle,1).
load(car_43,rectangle,1).
load(car_44,rectangle,1).
wheels(car_41,2).
wheels(car_42,2).
wheels(car_43,2).
wheels(car_44,2).

% eastbound train 5
has_car(east5,car_51).
has_car(east5,car_52).
has_car(east5,car_53).
short(car_51).
short(car_52).
short(car_53).
shape(car_51,rectangle).
shape(car_52,rectangle).
shape(car_53,rectangle).
double(car_51).
open_car(car_51).
closed(car_52).
closed(car_53).
load(car_51,triangle,1).
load(car_52,rectangle,1).
load(car_53,circle,1).
wheels(car_51,2).
wheels(car_52,3).
wheels(car_53,2).

% westbound train 6
has_car(west6,car_61).
has_car(west6,car_62).
long(car_61).
short(car_62).
shape(car_61,rectangle).
shape(car_62,rectangle).
closed(car_61).
open_car(car_62).
load(car_61,circle,3).
load(car_62,triangle,1).
wheels(car_61,2).
wheels(car_62,2).

% westbound train 7
has_car(west7,car_71).
has_car(west7,car_72).
has_car(west7,car_73).
short(car_71).
short(car_72).
long(car_73).
shape(car_71,rectangle).
shape(car_72,u_shaped).
shape(car_73,rectangle).
double(car_71).
open_car(car_71).
open_car(car_72).
jagged(car_73).
load(car_71,circle,1).
load(car_72,triangle,1).
load(car_73,nil,0).
wheels(car_71,2).
wheels(car_72,2).
wheels(car_73,2).

% westbound train 8
has_car(west8,car_81).
has_car(west8,car_82).
long(car_81).
short(car_82).
shape(car_81,rectangle).
shape(car_82,u_shaped).
closed(car_81).
open_car(car_82).
load(car_81,rectangle,1).
load(car_82,circle,1).
wheels(car_81,3).
wheels(car_82,2).

% westbound train 9
has_car(west9,car_91).
has_car(west9,car_92).
has_car(west9,car_93).
has_car(west9,car_94).
short(car_91).
long(car_92).
short(car_93).
short(car_94).
shape(car_91,u_shaped).
shape(car_92,rectangle).
shape(car_93,rectangle).
shape(car_94,u_shaped).
open_car(car_91).
jagged(car_92).
open_car(car_93).
open_car(car_94).
load(car_91,circle,1).
load(car_92,rectangle,1).
load(car_93,rectangle,1).
load(car_93,circle,1).
wheels(car_91,2).
wheels(car_92,2).
wheels(car_93,2).
wheels(car_94,2).

% westbound train 10
has_car(west10,car_101).
has_car(west10,car_102).
short(car_101).
long(car_102).
shape(car_101,u_shaped).
shape(car_102,rectangle).
open_car(car_101).
open_car(car_102).
load(car_101,rectangle,1).
load(car_102,rectangle,2).
wheels(car_101,2).
wheels(car_102,2).
```