Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Extending Graph Neural Networks with Relational Logic

Bc. Lukáš Zahradník

Supervisor: Ing. Gustav Šír
Field of study: Open Informatics
Subfield: Artificial Intelligence
August 2021

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Zahradník Lukáš** Personal ID number: **492968**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master’s thesis details

Master’s thesis title in English:

Extending Graph Neural Networks with Relational Logic

Master’s thesis title in Czech:

Rozšíření grafových neuronových sítí o relační logiku

Guidelines:

1) Get acquainted with the graph neural network (GNN) principles and relational logic
2) Perform an extensive review of existing related SW frameworks
3) Propose an integrative solution enabling researchers to use the relational logic expressiveness (graphs, hypergraphs, databases, rules) for specifying GNN learning problems in a user friendly manner
4) Integrate your library with some of the existing frameworks
5) Test your solution on different benchmarks, discuss pros/cons

Bibliography / sources:

1. Zhou, Jie, et al.: Graph neural networks: A review of methods and applications. preprint arXiv:1812.08434 (2018).
2. Kimmig, Angelika, Lilyana Mihalkova, and Lise Getoor: Lifted graphical models: a survey. Machine Learning 99.1 (2015): 1-45.
3. Sourek, Gustav, Filip Zelezny, and Ondrej Kuzelka: Beyond Graph Neural Networks with Lifted Relational Neural Networks. arXiv preprint arXiv:2007.06286 (2020).
4. Sourek, Gustav, and Filip Zelezny: Lossless Compression of Structured Convolutional Models via Lifting. arXiv preprint arXiv:2007.06567 (2020).

Name and workplace of master’s thesis supervisor:

Ing. Gustav Šír, Department of Computer Science, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **21.02.2021** Deadline for master's thesis submission: **13.08.2021**

Assignment valid until: **19.02.2023**

Ing. Gustav Šír Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt — Student’s signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank my supervisor Ing. Gustav Šír, for his invaluable expertise, guidance, and feedback, making this thesis possible.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, 13. August 2021

Abstract

The field of Graph Neural Networks extends the capabilities of Deep Learning methods from grid-like tensor data to structured representations outside the Euclidean domain. This allows to utilize the power of Neural Networks upon irregular graph-like structures, which are common descriptors of learning problems in various domains. This thesis builds upon the principles of Graph Neural Networks and extends them further by coupling with concepts from relational logic to address learning problems beyond simple graph propagation. This is done via introduction of a new declarative modelling language building upon the logic programming paradigm, in the spirit of a concept previously introduced as “Lifted Relational Neural Networks”. The proposed extension significantly adds to the expressiveness of existing Graph Neural Networks, allowing to encode complex deep relational learning models in a transparent, interpretable and user-friendly manner within the Python language.

Keywords: Graph Neural Networks, Relational Logic, Lifted Relational Neural Networks

Supervisor: Ing. Gustav Šír

Abstrakt

Oblast grafových neuronových sítí rozšiřuje možnosti metod hlubokého učení z reprezentace dat ve formě tenzorů do strukturované reprezentace mimo Euklidovský obor. To umožňuje využití výkonu neurálních sítí na nepravidelných grafových strukturách, které jsou běžné pro popis učících problémů v mnoha doménách. Práce se opírá o principy grafových neuronových sítí a rozšiřuje je zapojením konceptů relační logiky, díky kterým je možné vyjádření učících problémů nad rámec jednoduché propagace grafu. Tohoto je docíleno pomocí nového deklarativního modelovacího jazyka, který je založen na logickém programovacím paradigmatu v duchu konceptu již dříve představeným pod označením “Lifted Relational Neural Networks”. Navržené rozšíření výrazně přidává na expresivitě existujících grafových neuronových sítí a umožňuje vyjádření komplexních učících modelů z oblasti hlubokého relačního učení transparentním, interpretovatelným a uživatelsky přívětivým způsobem pomocí Python programovacího jazyka.

Klíčová slova: Grafové Neuronové Sítě, Relační Logika, Lifted Relational Neural Networks

Překlad názvu: Rozšíření grafových neuronových sítí o relační logiku

Contents

1 Introduction 1

2 Theoretical Background 3

2.1 Graph Neural Networks 3

2.1.1 Existing Python GNN Frameworks 5

2.2 Relational Logic 7

2.3 Lifted Relational Neural Networks 9

2.4 Related Work 9

3 Internals of PyNeuraLogic 11

3.1 The anatomy of a rule 11

3.2 Template, query, and example . . 12

3.3 Dynamic computation graphs . . 14

3.4 Model formats 16

3.5 Data formats 17

3.6 Miscellaneous 18

4 Introduction into PyNeuraLogic 19

5 Expressing Graph Neural Networks 23

5.1 Expressing GNN models 23

5.2 Learning problems 25

6 Extending Graph Neural Networks 27

6.1 Relational Logic 27

6.1.1 Feature embedding and aggregation 28

6.1.2 Ousting aggregation with embedding 30

6.1.3 Utilizing Hypergraphs 31

6.2 Learning on knowledge graphs . . 33

6.3 Beyond GNNs 34

6.3.1 Matching Patterns 35

6.3.2 Expressing Features 38

6.3.3 Extending Graph Outputs . . 40

6.3.4 Heterogenous Graphs 40

6.3.5 Heterophily Setting 42

7 Conclusions **45**

A Bibliography **47**

B Contents of attached CD **51**

Figures

2.1 This figure shows how the new representation h_u^i of the blue vertex in the left graph is computed via the message passing mechanism, given the current representation of each node h_j^{i-1} and the *aggregate* and *combine* functions. 5

3.1 H_2O and H_2 molecules samples . 15

3.2 H_2O and H_2 molecular computation graphs, grounded and neuralized with the Template 3.8. . 15

6.1 The Length feature embedding . 29

6.2 The template graph representation 31

6.3 The template graph with embedding of different carriages . . 32

6.4 3-regular not isomorphic graphs 36

6.5 Not regular not isomorphic graphs 37

6.6 Bicyclopentyl and Decalin 38

Tables

4.1 The XOR truth table 19

Chapter 1

Introduction

Deep Learning methods span a wide range of application areas ranging from Natural Language Processing to Computer Vision, where they have proven to be powerful tools. The input data for these methods have been usually represented within the Euclidean domain in grid-like formats of numeric tensors, which are suitable for numerous tasks, however, there are also domains where these grid-like formats are not a proper representation for capturing information about a given problem.

The introduction of Graph Neural Networks (GNNs) approached this issue by bringing the power of Neural Networks into the non-Euclidean domains - particularly to graph representations. Graphs can efficiently capture entities and their relations, making them natural and fitting descriptors for a diverse range of real-world problems, such as drug discovery, social network analysis, or fraud detection.

In this thesis, we build upon the Graph Neural Network concepts by extending them with Relational Logic. This yields a highly expressive custom modelling language inspired by the declarative logic programming paradigm, in the spirit of so-called "Lifted Relational Neural Networks", which have been introduced with a similar aim of extending standard feed-forward neural networks. By marrying these two disciplines, we try to reach beyond the possibilities of regular Graph Neural Networks by enhancing their expressiveness, while also improving the interpretability of the underlying models. We showcase various possible extensions and scenarios, which cannot be performed using regular Graph Neural Networks or require some specifically crafted models which are usually presented as black boxes that cannot be

easily customized or reasoned about. In contrast to these, the proposed modelling paradigm makes it easy to modify, combine, and extend various modelling concepts in a very direct and interpretable manner.

The main contribution of this thesis is a custom Python Machine Learning library, which can be used in conjunction with the extensive existing Python ecosystem, making this work usable for a broad audience of machine learning practitioners. The main contribution of the library itself is implementation of the aforementioned GNN modelling constructs and their relational extensions, while utilizing features of the Python language itself. To reflect this, examples of various learning scenarios are presented throughout the thesis in the form of modelling concepts in the library's custom modelling language stemming from the relational logic programming paradigm. Note that all the presented examples are an actual Python code that can be run within the library, which is available at `https://github.com/LukasZahradnik/PyNeuraLogic`.

The thesis is structured as follows. Chapter 2 introduces readers to Graph Neural Networks and their existing frameworks, Relational Logic and Lifted Relational Neural Networks. In Chapter 3, we present the basic syntax, formats, and other internals of the implemented library, which are critical for understanding the models, examples, and GNN enhancements. Chapter 4 presents the workflow of defining the model, dataset, and learning on a simple example. Chapter 5 examines the representation of a selected set of Graph Neural Network models in the implemented library. In Chapter 6, we demonstrate use cases that extend the possibilities of Graph Neural Networks via Relational Logic, and conclude in Chapter 7.

Chapter 2

Theoretical Background

2.1 Graph Neural Networks

Graph Neural Networks (GNN) provide frameworks for learning from data represented in the form of graph data structures. Such graph representation can be beneficial and natural for several problems in various areas, such as chemistry, biology, and social networks, expressing relations (edges) between objects (vertices) [1].

Graph Classes

We define a simple homogenous graph as a tuple $G = (V, E)$, where V is a set of vertices and $E = \{(u_1, v_1), ..., (u_n, v_n)\}$ is a set of pairs of vertices $u_j, v_j \in V$ representing edges. Additionally, we define a function $x_v : V \rightarrow \mathbb{R}^{n \times m}$ and a function $x_e : E \rightarrow \mathbb{R}^{n \times m}$, which map nodes and edges to feature tensors. Graph G is called a directed graph if the pairs in the set of edges E are ordered.

A heterogeneous graph is then a graph $G = (V, E)$, with additional mapping functions $f(v) : V \rightarrow Y$ and $g(e) : E \rightarrow Z$, which are mapping vertices and edges to different types.

A hypergraph is a generalization of a simple graph $G = (V, E)$, that does not bound only two vertices to form an edge, but allows an edge to be formed from an arbitrary number of vertices; thus, $(v_0, ..., v_n) \in E, v_{0...n} \in V$.

GNN Classes

GNNs, according to [2], can be categorized into four categories:

- RecGNNs
- ConvGNNs
- GAE
- STGNNs

For the purposes of this work, we will focus on the ConvGNN class, but the proposed library is not strictly limited to the said class, as the main principles are similar amongst categories. We have made a choice to focus on ConvGNN because it is the most popular category. The ConvGNN class can be further divided into two subcategories - Spectral methods and Spatial methods. We will further discuss only spatial-based methods, as they are significantly more used than the spectral-based methods, and will use the term GNN interchangeably with spatial based ConvGNNs.

ConvGNNs, during updating node representation, utilize multiple layers with learnable parameters unique to every layer. ConvGNNs take the currently updated node and convolve its representation with nodes based on their spatial properties - convolve with direct neighbors' representations of the updated node. Such mechanism of propagation which depends on the existence of edges is termed message passing.

Message Passing

The computation of the update at layer i of node u via message passing-based GNNs can be characterized by two steps. The first one being the aggregation step, where we take the central node's neighbors' $N(u)$ hidden representations $h_{j \in N(u)}^{(i-1)}$ as inputs to the *aggregation* function. In the next step - the combination or the update step, we take the aggregated value and combine it via the *combine* function with the hidden representation $h_u^{(i-1)}$ of the currently updated node u (the central node). Both *aggregation*

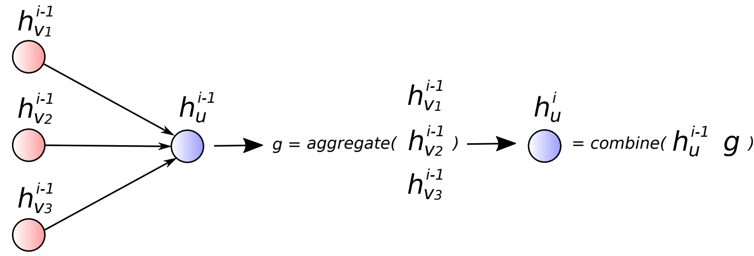

Figure 2.1: This figure shows how the new representation h_u^i of the blue vertex in the left graph is computed via the message passing mechanism, given the current representation of each node h_j^{i-1} and the *aggregate* and *combine* functions.

and *combine* functions can be parameterized and do not share parameters throughout layers.

$$h_v^{(0)} = X(v)$$
$$aggregate^{(i)}(u) = aggregation^{(i)}(\{h_j^{(i-1)} : j \in N(u)\})$$
$$h_u^{(i)} = combine^{(i)}(h_u^{(i-1)}, aggregate^{(i)}(u))$$

To learn the graph embedding h_G for graph classification and graph regression tasks, GNNs utilize yet another function *output* with learnable parameters. The hidden representations from the last layer k of the GNN are passed as inputs for the function *output*.

$$h_G = output(\{h_v^{(k)} : v \in V\})$$

2.1.1 Existing Python GNN Frameworks

The area of Graph Neural Networks gained momentum recently, resulting in a higher frequency of introductions of new libraries. The following review of GNN Python libraries considers only a selected sample of libraries. We included the most popular ones at the present time.

PyTorch Geometric

PyTorch Geometric (PyG) [3] is a popular GNN library[1] from the PyTorch ecosystem. It implements numerous models for learning on irregular structures such as graphs, a framework for implementing custom message-passing models, and last but not least, data helpers and transformers.

The representation of graphs differs depending on the used model, but most common is a list of tuples representing the id of the source and the destination node of an edge. The representation of features is a fixed numeric tensor of dimensions $N \times M$, where N is a number of nodes and M is a number of features. The number of features is constant for every node; we cannot represent features as jagged arrays.

Currently, PyTorch Geometric has only one model which supports heterogeneous graphs. Only one implemented model also supports hypergraphs.

DGL

The Deep Graph Library (DGL) [4] is another popular Graph Neural Networks library[2]. The library itself is framework agnostic - can be run on top of different frameworks/backends such as PyTorch, TensorFlow, or Apache MXNet.

DGL offers a number of implemented models mainly focused on homogenous graphs with one heterogeneous model. Currently, DGL does not support working on hypergraphs.

DGL represents the graph as a list of two lists with indices of the source and destination node. The features are represented as one fixed tensor which can be multidimensional.

[1] `https://github.com/rusty1s/pytorch_geometric` - At the time, it has over 11.5k stars on GitHub

[2] `https://github.com/dmlc/dgl` - At the time, it has over 7.8k stars on GitHub

Spektral

Spektral [5] is a GNN library built on top of Keras and TensorFlow 2 and follows the Keras principles to make usage simpler. The implemented models focus on homogenous graphs and do not consider heterogeneous graphs. Spektral also does not implement models operating on hypergraphs.

The graph is usually represented as an adjacency matrix - $N \times N$, where N is the number of nodes, or as a list of $E \times 3$ where E is the number of edges and each row contains the source node of the edge, destination node of the edge and weight. The node features are in the form of one fixed numeric tensor of dimension $N \times M$, where N is a number of nodes and M is a number of features. The edge features can be represented as a tensor of dimension $E \times M$, where E is the number of edges and M is the number of features.

Stellar Graph

Stellar Graph [6] is another GNN library built on top of TensorFlow 2 and Keras. It offers various models, and many of them support heterogeneous graphs to some extent. Hypergraphs are not currently supported.

The graph is represented similarly to other GNN frameworks in the form of a list of tuples with indices of the source and destination nodes. The features are then in the form of tensor $N \times M$, where N is the number of nodes and M is the number of features.

During our research, we have come across more libraries and frameworks for GNNs; their features and possibilities were mainly identical to the ones listed above. Many of them had poor documentation, were no longer supported, or seemed to be abandoned; thus, we decided to omit those.

2.2 Relational Logic

Relational logic [7] introduces an expressive language to describe entities and relations amongst them. We define relational logic following the definition from [8]; that is, a relational logic theory is a set of formulae constructed

from constants representing objects, variables representing placeholders for objects, predicates representing properties of objects and their relations, and propositional connectives ($\wedge, \vee, \neg$) and quantifiers ($\exists, \forall$). For this work, we assume the formulae to be function-free, and all variables are considered to be universally quantified.

Constants are represented either in numeric form or as strings with a first lower letter (e.g., *jerry*). On the other hand, the variables are represented as strings starting with an upper letter (e.g., *Dog*, *Molecule*, X). Variables and constants are called terms. An atom is a predicate applied to a tuple of $n \in \mathbb{N}$ terms and is used to describe relations between objects or their properties, e.g., *haveBond(X, hydrogen)*. A ground atom is an atom with no variables in its terms, e.g., *haveBond(oxygen, hydrogen)*. A literal is then an atom or its negation, e.g., a, $\neg a$. A disjunction of universally quantified literals forms a clause. A clause is then called a definite clause iff the clause has precisely one positive literal. A fact is a definite clause with only one atom. A definite clause $h \vee \neg b_1 \vee ... \vee \neg b_n$ can be expressed in the form of implication as $h \impliedby b_1 \wedge ... \wedge b_n$, where the conjunction of b_j atoms forms a body and h atom forms a head.

The Herbrand base of a set of definite clauses is the set of all ground atoms formed via constants and predicates in the set of definite clauses. A Herbrand interpretation of a set of definite clauses P is a mapping assigning a truth value to all elements from the Herbrand base of P. A Herbrand interpretation I satisfies a ground atom F if $F \in I$. A set of ground clauses is satisfiable if there is at least one Herbrand interpretation I that assigns a true value to all clauses. This Herbrand interpretation is called a Herbrand model. Every set of definite clauses has a unique Herbrand model that is minimal w.r.t. the subset relation - the least Herbrand model.

The grounding of a clause a from a set of non-ground definite clauses P is the set of ground clauses $G(a) = a\theta_0, ..., a\theta_n$, where $\theta_0, ..., \theta_1$ is the set of all possible substitutions (mappings of variables from non-ground definite clause to constants in P). Grounding of P is then $G(P) = \bigcup_{a \in P} G(a)$, and P's least Herbrand model is the least Herbrand model of $G(P)$. The restricted grounding is grounding limited to only those clauses which have satisfied body in the least Herbrand model.

2.3 Lifted Relational Neural Networks

Lifted (templated) models utilize highly expressive representations to capture symmetries in relational learning problems [9]. A Lifted Relational Neural Networks (LRNN) [8] language encodes various neural architectures and problems as parametrized logic programs. In this work, the implemented library builds upon the LRNN language, which in turn is based on the language of Datalog [10]. LRNNs are inspired by lifted graphical models based on probability theory and logic programming [11]. The internals and formats of the library language are discussed in Chapter 3.

The mapping from the logic representation into the neural form is called neuralization. Such neuralization of the logic program (set of clauses) follows the definition from [8] and is done by taking a set of rules and facts N from an example E and template T while omitting weights. We then construct the least Herbrand model of N denoted as $\bar{N}$. To be able to construct a computation graph for learning, the current query has to be present in $\bar{N}$. The mapping is then done as:

- All weighted ground facts in the example are mapped to fact nodes.
- All ground atoms in $\bar{N} \setminus E$ are mapped to atom nodes.
- All ground rules present in $\bar{N}$ are mapped to rule nodes.
- All ground rules with the same head atom are mapped to respective aggregation nodes.

All nodes are later connected to follow the derivation of the logical facts by the immediate consequence operator, starting from the fact nodes, ultimately forming a computational graph, such as that of a neural network.

2.4 Related Work

The library implemented in this thesis (PyNeuraLogic[3]) utilizes features, such as the ground resolution and neuralization, of the NeuraLogic framework [8].

[3] https://github.com/LukasZahradnik/PyNeuraLogic

NeuraLogic implements its own custom declarative language that PyNeuraLogic extends, simplifies, and introduces into the Python ecosystem, thus adding expressiveness while simplifying neural networks representation. This is similar to the historical relationship between Torch [12] and PyTorch [13].

The NeuraLogic Toolbox [14] is remotely related work to the PyNeuraLogic library. This toolbox also utilizes the NeuraLogic framework and serves as a helper library wrapper and analyzer of problems for the NeuraLogic framework. The PyNeuraLogic's scope overlaps in marginal areas with the NeuraLogic Toolbox, but its primary goals, use cases and features differ significantly.

The ProbLog [15] language is also similar to the implemented library in extending logic programs with numerical parameters. It offers a probabilistic interpretation that is not present in Lifted Relational Neural Networks, which PyNeuraLogic is based on. DeepProbLog [16] is then an extension of ProbLog that incorporates deep learning employing neural predicates. The shortcoming of DeepProbLog is not being able to model complex convolutional patterns [8].

Chapter 3

Internals of PyNeuraLogic

The library implemented in this work, PyNeuraLogic[1], allows users to encode machine learning problems via parameterized, rule-based constructs. Said constructs are based on a custom declarative language that follows a logic programming paradigm. This, in conjunction with the Python language, results in a highly expressive language which can describe an extensive range of problems.

3.1 The anatomy of a rule

In PyNeuraLogic, rules are primitives used for building models and datasets. The rule $R = ((W_0, h) \longleftarrow (W_1, b_1), ..., (W_n, b_n))$, where W_j's are real-valued tensors and h and b_j's are atoms, consists of two main parts - the head and the body. Additionally, the rule can have metadata attached, which can be used to specify aggregation functions, activation functions for the rule, and more. Example 3.1 shows the syntax of rules, specifically one rule with two atoms (`b_1` and `b_n`) in its body. Weights assigned to each atom (i.e., `W_0`, `W_1`, and `W_n`) are either scalars or numeric tensors.

Listing 3.1: Example of rule syntax

```
Atom.h[W_0] <= (Atom.b_1[W_1], Atom.b_n[W_n])
```

[1] `https://github.com/LukasZahradnik/PyNeuraLogic`

The head can contain only one atom called the head atom, and the body can contain an arbitrary number of atoms. Each atom consists of the predicate name, an optional arbitrary number of terms, optional weight or value, and optional predicate modifier.

3.2 Template, query, and example

To be able to learn a problem via the PyNeuraLogic library, it is necessary to encode the problem in the form of a template, queries, and examples. A set of queries, together with a learning example or examples, forms a learning dataset. When only one example is provided to the dataset, the example is streamed to all the queries present in the dataset.

Template

The template $T = \{R_1, ..., R_n\}$ is a set of rules that encode the problem's architecture - i.e. the model, which is essentially equivalent to composing modules in popular frameworks, but more versatile. The versatility is achieved by utilizing lower-level primitives within the rules that can be freely modified.

Listing 3.2: Simple example template

```
[
    Atom.predict <= (
        Atom.first_feature[1,], Atom.second_feature[1,]
    ),
]
```

The template 3.2 defines one rule consisting of three atoms where atoms `first_feature` and `second_feature` have a learnable one-dimensional (scalar) weights associated, and these atoms imply the atom `predict`. Every atom can have an arbitrary number of terms consisting of variables and constants, as shown in Example 3.3.

Listing 3.3: Simple example template with terms

```
[
    Atom.predict(Var.X) <= (
        Atom.relation(Var.X, Var.Y),
        Atom.feature(Var.Y, Term.some_constant)[1,],
    ),
]
```

The logic variables (e.g., `Var.X`) are the primary source of the said versatility. They allow us to operate with entities (nodes, edges) and reference them arbitrarily, enabling to express and capture various relations amongst entities and other features discussed in Chapter 6.

Example

The learning example is a set of facts which can also be used to express node features, edge features, or any other facet or knowledge of the current instance of the world.

Listing 3.4: Simple example

```
[Atom.first_feature[1.0], Atom.second_feature[0.0]]
```

In the Example 3.4 we have defined one example set of two facts that assign values 1.0 and 0.0 to atoms `first_feature` and `second_feature`, respectively. Values are not restricted only to scalars; it is also possible to assign vectors and matrices as values, as shown in 3.5.

Listing 3.5: Vector and matrix values

```
[
    Atom.vector_feature[[1.0, 2.0, 3.0]],
    Atom.matrix_feature[[
        [0.0, 1.0],
        [2.0, 3.0],
    ]],
]
```

The predicate names are chosen arbitrarily by the user and serve as a descriptor of relations and data in our models. There also exist special built-in predicates that entail additional functionality. Such predicates are prepended with the *special* keyword, such as `Atom.special.alldiff`, `Atom.special.true`, and `Atom.special.false`.

Query

The query is a valued fact used to describe the target regression or classification target for any atom of the template. The query is not bound to only the last layer of the defined template. In contrast to regular GNN frameworks, where

we usually have only one output layer, in PyNeuraLogic, the output changes depending on the query. Example 3.6 shows the encoding of a simple query that might be used for regression or classification on the graph, as it does not target a specific node. On the other hand, Example 3.7 utilize terms to target specific node for querying. Queries in Example 3.7 will yield different computation graphs depending on the example set.

Listing 3.6: Simple query of the atom predict with the target value 1.0

```
Atom.predict[1.0]
```

Listing 3.7: Two simple queries with different terms - 0 and 1, which result in different computation graphs

```
Atom.predict(1)[0.0],
Atom.predict(0)[1.0],
```

3.3 Dynamic computation graphs

For each data instance (i.e. example and query), a unique computational graph is constructed based on the grounding (Sec. 2.2) and neuralization (Sec. 2.3) of the defined template. We present the visualization of the behavior of neuralization and the construction of dynamic computation graphs on a learning problem from the chemistry field, which is a model for learning on molecules based on a model presented in [8]. Example 3.8 shows a template of the model that defines two layers (rules). For the purpose of this section, we do not specify concrete weights; we specify aliases (Wa, Wb, $Wh1$, $Wh2$) instead to make them distinguishable in Figure 3.2.

Listing 3.8: Template for learning on molecules

```
Atom.h(Var.X)[Wh1] <= (
    Atom.atom(Var.Y)[Wa],
    Atom.bond(Var.X, Var.Y)[Wb],
),
Atom.q[Wq] <= Atom.h(Var.X)[Wh2],
```

The template defines one layer (a rule) for aggregating and combining the central node's neighbors to compute the central node's embedding h_x, similarly to the message passing mechanism (Sec. 2.1). The second rule does the global pooling - it takes computed embeddings of all nodes and calculates the graph output h_G (`Atom.q`).

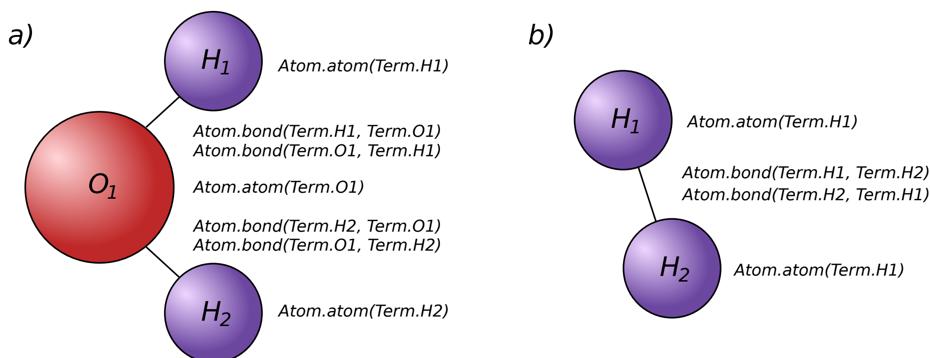

Figure 3.1: H_2O and H_2 molecules samples

As the learning examples, we define two molecules - H_2O and H_2 visualized in Figure 3.1 together with their analogous encodings. We then ground those examples with the template and retrieve two unique computation graphs visualized in Figure 3.2.

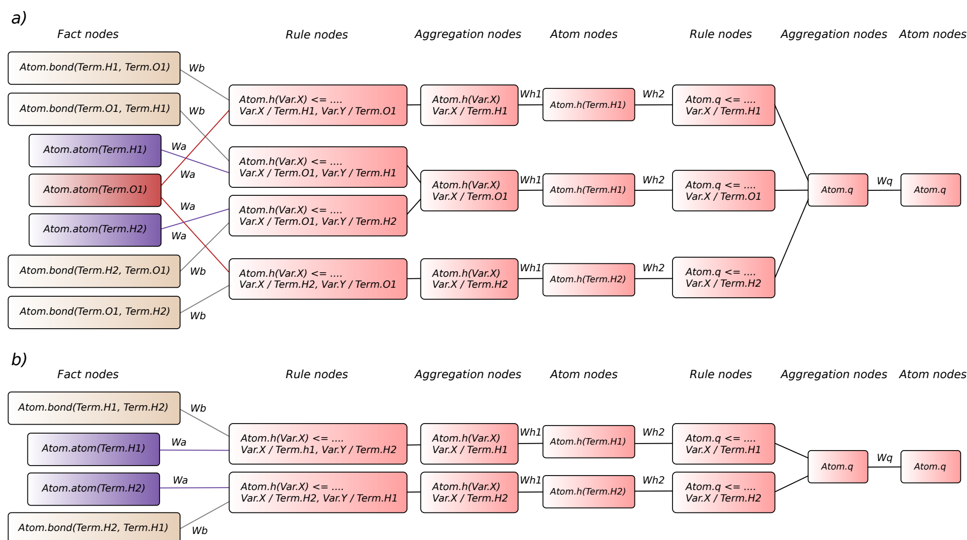

Figure 3.2: H_2O and H_2 molecular computation graphs, grounded and neuralized with the Template 3.8.

We note that the resulting computation is equivalent to the GNN computation scheme from Section 2.1. The graphs are evaluated from the fact nodes containing input values to the last atom node on the right side (`Atom.q`), which serves as the computation graph output.

3.4 Model formats

The PyNeuraLogic library offers a set of formats that can be used to describe the model.

Plaintext LRNN

Firstly, the template can be encoded in an external plaintext file using a Datalog-like language. This format offers all implemented language-level features but cannot be freely manipulated via code.

Listing 3.9: GCNConv encoded using plaintext file format

```
{1, 8} predict(X) :- edge(X, Y), feature(Y).
```

Template using Python objects

Templating using Python objects brings the apparent advantage of manipulating the template rules via Python language's features and libraries, which can simplify the creation of the template.

Listing 3.10: GCNConv encoded using Python objects

```
Atom.predict(Var.X)[1, 8] <= (
    Atom.edge(Var.X, Var.Y), Atom.feature(Var.Y)
)
```

Predefined modules

For convenience, the library also contains some predefined popular modules such as GCNConv [17], SAGEConv [18], or GINConv [19]. Those modules can be either translated in the background into template rules (Python objects) or modules from Python Geometric, depending on the chosen backend.

Listing 3.11: GCNConv via predefined modules

```
GCNConv(8, 1, name="predict")
```

3.5 Data formats

The data in datasets can also be represented and stored in multiple formats.

Logic

Logic format divides the dataset into two parts - examples and queries and uses facts and rules to describe the data. This approach has the advantage of being more expressive but, in some instances, can be considered too verbose. An instance of logic format encoding can be seen in Example 3.12, which describes the edges of a graph.

Listing 3.12: Encoding example of edges using a logic

```
examples = [
    Atom.edge(0, 1),
    Atom.edge(1, 0),
    Atom.edge(1, 2),
    Atom.edge(2, 1),
    Atom.edge(2, 0),
    Atom.edge(0, 2),
]
```

Matrices/Vectors

This format is taken over from the PyTorch Geometric library and allows users to describe nodes features, edge index, and target values as numeric tensors. This format reduces verbosity but reduces readability due to features having only numeric values without any text descriptors. Example 3.13 shows encoded edges of the graph from Example 3.12 via matrix format.

Listing 3.13: Encoding example of edges using an array

```
edge_index = [
    [0, 1, 1, 2, 2, 0],
    [1, 0, 2, 1, 0, 2],
]
```

This format also enables users to use the dataset for the template built via predefined modules, such as GCNConv, in a scenario with PyTorch Geometric modules, in which the equivalent encoded in logic cannot be used.

The format is not strictly the same as the PyTorch Geometric format and requires some additional transformation to be compatible with the template built on top of rules. Thus the PyNeuraLogic library contains a helper PyTorch Geometric data converter.

3.6 Miscellaneous

PyNeuraLogic allows retrieving the embedding of hooked atoms while querying different atoms. This can be achieved by adding a callback function, a regular Python function that can wrap anything (e.g., visualization), as a hook to specific predicate and terms. Usage of hooks can be seen in Example 3.14, where the hook is attached to the first rule's head. Every time the hooked atom's (`layer_1(3)`) value is being calculated, the callback function `my_hook` is called with the updated value, which can be utilized for helpful insight while querying other atoms.

Listing 3.14: Hooking an atom with predicate *somePredicate* and a term *3*

```
template.add_rules([
    Atom.layer_1(Var.X)[1,] <= (
        Atom.feature(Var.Y), Atom.edge(Var.X, Var.Y)
    ),
    Atom.layer_2(Var.X)[1,] <= (
        Atom.layer_1(Var.Y), Atom.edge(Var.X, Var.Y)
    ),
    Atom.predict(Var.X)[1,] <= Atom.layer_2(Var.X),
])

def my_hook(value):
    print("Value of the hooked atom layer_1(3) is:", value)

template.add_hook(Atom.layer_1(3), my_hook)
```

Chapter 4

Introduction into PyNeuraLogic

Learning the XOR operation is a relatively elementary task, but it serves as a good example to showcase the basics of problem encoding and library usage.[1] Note that the problem is used for simple library introduction and is, in fact, a propositional rather than a relational problem.[2]

The XOR operation has two inputs - $I_1 \in \{0, 1\}$ and $I_2 \in \{0, 1\}$, and one output $O \in \{0, 1\}$. The whole operation can be summarized by Table 4.1.

X	Y	O
0	0	0
1	0	1
0	1	1
1	1	0

Table 4.1: The XOR truth table

[1]The fully working example in the form of Jupyter Notebook can be found at `https://github.com/LukasZahradnik/PyNeuraLogic/blob/master/examples/IntroductionIntoPyNeuraLogic.ipynb`

[2]i.e. the template here does not contain any variables, causing it to correspond to a standard neural network rather than a GNN.

Template

The model for learning the XOR operation can be expressed in multiple ways; the following model (Example 4.1) reduces the architecture into one rule, representing one layer. The rule can be read as: *"Atom xor is implied by atom xy."*

Listing 4.1: The XOR Template definition

```
with Template().context() as template:
    template.add_rule(Atom.xor[1, 8] <= Atom.xy[8, 2])
```

We also declared weight with given dimensions for each atom - W_{xor} for atom `xor` and W_{xy} for atom `xy`. Since we did not specify concrete values for weights, those learnable parameters will be sampled randomly from, by default, the uniform distribution.[3]

This rule subsequently represents the following equation, where the output of $f(x)$ is the output of the `xor` atom and x is the value of the `xy` atom. Functions ϕ_{rule} and ϕ_{xor} are activation functions of our rule and the atom `xor`, respectively. In our case, we did not explicitly specify activation functions via metadata; thus, they will be set to default activation functions depending on the node type. This results in ϕ_{rule} being equal to the tanh function, and ϕ_{xor} being the identity function.

$$W_{xor} \in \mathbb{R}^{1,8}, W_{xy} \in \mathbb{R}^{8,2}, x \in \{0,1\}^2$$
$$f(x) = \phi_{xor}(W_{xor} \cdot \phi_{rule}(W_{xy} \cdot x))$$

Defining a Dataset

To be able to learn our parameters W_{xor} and W_{xy}, we need to create a training dataset that contains examples. In our case, the dataset examples are straightforward and mimic the truth table (Table 4.1). The representation of the dataset is in Example 4.2.

[3] The distribution can be changed via settings.

Listing 4.2: The XOR Dataset definition

```
with template.context():
    dataset.add_examples(
        [
            Atom.xor[0] <= Atom.xy[[0, 0]],
            Atom.xor[1] <= Atom.xy[[0, 1]],
            Atom.xor[1] <= Atom.xy[[1, 0]],
            Atom.xor[0] <= Atom.xy[[1, 1]],
        ]
    )
```

Each example in the dataset corresponds to one row in the truth table. In the scope of datasets, the value of each atom is its actual value and not (a learnable) weight.

For example, Example 4.3 can be read as: *"Given the atom xy's value is equal to the vector* $(0, 1)$*, we are expecting the atom xor to have a value equal to scalar* 1*."*

Listing 4.3: Single example

```
Atom.xor[1] <= Atom.xy[[0, 1]]
```

Training

We can do the training manually by writing a training loop, similarly to popular frameworks, or using a predefined training loop implemented inside evaluators, which are suitable for quick prototyping and switching between different backends, such as DyNet [20] or Java. Such evaluators can be conveniently customized via settings to specify optimizer, learning rate, error function, and more. In Example 4.4, we have chosen the DyNet backend with a stochastic gradient descent optimizer for training and evaluated the training using the `evaluator.train` function.

Listing 4.4: Training the model

```
settings = Settings(optimizer=Optimizer.SGD, epochs=100)

evaluator = get_evaluator(
    Backend.DYNET, template, settings=settings
)

for epoch, (total_loss, seen_instances) in \
    enumerate(evaluator.train(dataset)):
    pass
```

Before the training is evaluated, our dataset is grounded with our template. The grounding then yields one computation graph for each query from the dataset. In our case this will produce, for each query, a computation network with the same structure but with different input and target values.

Testing

Evaluators also encapsulate testing with a user-friendly interface that is analogous to training, which is presented in Example 4.5.

Listing 4.5: Testing the model

```
for label, predicted in evaluator.test(dataset):
    print(f"Label: {label}, predicted: {predicted}")
```

Chapter 5

Expressing Graph Neural Networks

5.1 Expressing GNN models

The area of Graph Neural Networks encloses a broad number of different models. This section will focus on a few prominent ones and show their possible implementations in the PyNeuraLogic library. In the proposed implementations, we can see similarities of the showcased models and inspect their internals. These properties and similarities are commonly hidden in the popular GNN frameworks which treat them as black boxes hidden behind many abstraction layers, making additional customization and tuning relatively unfeasible.

GCNConv

The Graph Convolutional Network [17], shortly GCN, is one of the most popular proposed architectures for learning on graphs. Its architecture can be essentially encoded as a rule in Example 5.1. The notable disadvantage of GCN is not considering the central node while computing its new embedding.

Listing 5.1: GCNConv representation via rules

```
Atom.h(Var.V)[1,] <= (
    Atom.previous_h(Var.U),
    Atom.edge(Var.V, Var.U),
)
```

We encapsulate the GCN model in the `GCNConv` class (Example 5.2), which is equivalent to the rule representation with the additional possibility to be evaluated directly in different GNN backend library modules.

Listing 5.2: The `GCNConv` class prototype

```
GCNConv(in_channels, out_channels, *, activation, aggregation)
```

SAGEConv

The GraphSAGE [18] architecture can be looked at as an extension of the GCN architecture. The GraphSAGE extends the GCN to incorporate the central node itself in the computation of the node's new embedding. The architecture can be encoded similarly to GCNConv with an additional rule for the task, as shown in Example 5.3. The model is also encapsulated as SAGEConv class (Example 5.4).

Listing 5.3: SAGEConv representation via rules

```
Atom.h(Var.V)[1,] <= (
    Atom.previous_h(Var.U),
    Atom.edge(Var.V, Var.U),
)
Atom.h(Var.V)[1,] <= Atom.previous_h(Var.V),
```

Listing 5.4: The `SAGEConv` class prototype

```
SAGEConv(in_channels, out_channels, *, activation, aggregation)
```

GINConv

The Graph Isomorphism Network (GIN) [19] is a powerful architecture based on Weisfeiler-Lehman graph isomorphism test (WL test) [21] and is ranked upon the maximally powerful GNNs. GIN adds on top of aggregation an MLP with two layers. Such architecture can be encoded as in Example 5.5, where one of the GIN variations, GIN-0 [19], is implemented. Example 5.6 then shows the encapsulated model of GIN-0 in class.

Listing 5.5: GINConv representation via rules

```
Atom.mlp(Var.V) <= (Atom.h(Var.U), Atom.edge(Var.V, Var.U)),
Atom.mlp(Var.V) <= Atom.h(Var.V),

Atom.h(Var.V)[1,] <= Atom.mlp(Var.V)[1,]
```

Listing 5.6: The `GINConv` class prototype

```
GINConv(in_channels, out_channels, *, activation, aggregation)
```

5.2 Learning problems

PyNeuraLogic can be used similarly to other frameworks. This section showcases the usage of pre-defined modules on a well-known example - the citation network dataset Cora [22]. To load the dataset, we first load it using PyTorch Geometric dataset loader, then we use one of PyNeuraLogic's helpers to get the train and test data out of the PyTorch Geometric dataset, which will we then use to build PyNeuraLogic's datasets, as shown in Example 5.7.

Listing 5.7: The definition of a template with pre-defined components

```
dataset = Planetoid(
    path, "Cora", transform=T.NormalizeFeatures()
)

[train_data, test_data, _] = Data.from_pyg(dataset[0])

train_dataset = Dataset(data=[train_data])
test_dataset = Dataset(data=[test_data])
```

Our datasets are now in the form of tensors and not in logic form. Dataset can be implicitly transformed to the logical form, specifically to the plaintext LRNN format.

The definition of the model (template) is similar to popular frameworks. We can utilize a template list with pre-defined models, which will be sequentially connected depending on their order in the list. Those modules can be again implicitly translated into the logic format. In Example 5.8, we define a template with two GCNConv layers with different activations - ReLu and Sigmoid.

Listing 5.8: The definition of a template with pre-defined components

```
template_list = TemplateList([
    GCNConv(in_channels=dataset.num_features, out_channels=16,
        activation=Activation.RELU),
    GCNConv(in_channels=16, out_channels=dataset.num_classes,
        activation=Activation.SIGMOID),
])

template = Template(module_list=template_list)
```

The training and testing then follow the previously shown workflow. We can define our custom loop, which gives us more control, or we can use implemented evaluator customizable via settings.

Listing 5.9: The training with evaluators

```
evaluator = get_evaluator(
    Backend.PYG, template, settings, native_backend_models=True
)

for epoch, (total_loss, seen_instances) \
        in enumerate(evaluator.train(train_dataset)):
    pass
```

Example 5.9 shows using the PyTorch Geometric backend to train the defined model. In the background, the template is being mapped from PyNeuraLogic models to PyTorch Geometric models. The learning loop is abstracted from the user, but we can still do some operations with metrics in the loop (e.g., plotting loss). The testing is then the same as shown in the previous chapter.

Chapter 6

Extending Graph Neural Networks

6.1 Relational Logic

We utilize Relational Logic to add expressiveness and features to the area of Graph Neural Networks. Via logic variables, we are able to query different nodes and entities in the graph, depending on some of their characteristics, which, together with other aspects of relational logic, opens doors to various use-cases. This chapter presents some ways to utilize the expressive power of the constructs implemented in PyNeuraLogic, which are based on the Relational Logic language.

The Trains problem [23], also known as East-West trains or East-West challenge, is a toy problem from the field of inductive logic programming (ILP). The problem consists of a set of trains $T = \{\vec{c_0}, ..., \vec{c_n}\}$, and a set of ordered carriages - feature vectors $\vec{c_i}$. The goal of the problem is to classify the direction of each train, which can be either east or west, based on the features of the carriages $\vec{c_i}$. Provided properties consist of information about *the shape, the length, the sides, the roof, the number of wheels, the load shape*, and *the load number* of each carriage.

This chapter is about showcasing how we can interpret one problem in different ways with only a few minor changes.[1] The proposed approaches can be improved, for example, by tuning weights' dimensions or modifying rules.

[1]The aim is not to come up with the best approach

6.1.1 Feature embedding and aggregation

In the proposed model in this section, we purposely do not consider each carriage's order. The accuracy of this model converges slower than the other approaches to 1.0.

The Dataset

For each train, we create one example containing features of only the specific train and one query related to the train. We can directly encode the features into examples as facts, where the atom predicate name is equal to the feature's name, and the feature value is present as a term together with the carriage identification (in our case, the carriage's position).

For example, we can encode the feature vector $\vec{c_i}$ of one carriage in the form of $(position, shape, length, sides, roof, wheels, load, loadnum)$ and with the following values:

$$\vec{c_i} = (1, "rectangle", "short", "not_double", "none", 2, "circle", 1)$$

into an example set encoded as in Example 6.1.

Listing 6.1: Encoding of one carriage

```
[
    Atom.shape(1, "rectangle"),
    Atom.length(1, "short"),
    Atom.sides(1, "not_double"),
    Atom.roof(1, "none"),
    Atom.wheels(1, 2),
    Atom.loadshape(1, "circle"),
    Atom.loadnum(1, 1),
]
```

The query is again a valued fact, with an assigned value of either −1.0 (east) or 1.0 (west), depending on the train's class. For the predicate's name, we have chosen the name `direction`. For example, the query for the train which is supposed to go to the east will be defined as in Example 6.2.

Listing 6.2: Train direction query

```
Atom.direction[-1.0]
```

The Template

To define the template, we first declare lists of all possible values of each feature (shown in Example 6.3), which we will utilize for the feature embedding. For convenience, we also declare a list of all atoms related to the features of a carriage.

Listing 6.3: All Features' possible values

```
shapes = [Term.ellipse, Term.rectangle, Term.bucket,
          Term.hexagon, Term.u_shaped]

roofs = [Term.jagged, Term.arc, Term.none,
         Term.flat, Term.peaked]

loadshapes = [Term.hexagon, Term.triangle,
              Term.diamond, Term.rectangle, Term.circle]

sides = [Term.not_double, Term.double]

lengths = [Term.short, Term.long]

loadnums = [0, 1, 2, 3]

wheels = [2, 3]

carriage_features = [Atom.shape, Atom.length, Atom.sides,
                     Atom.wheels, Atom.loadnum,
                     Atom.loadshape, Atom.roofcarriage]
```

The proposed template embeds every possible value of all features into scalars unique for each value. This embedding results in all values having their own learnable parameter - weights are not being shared here.

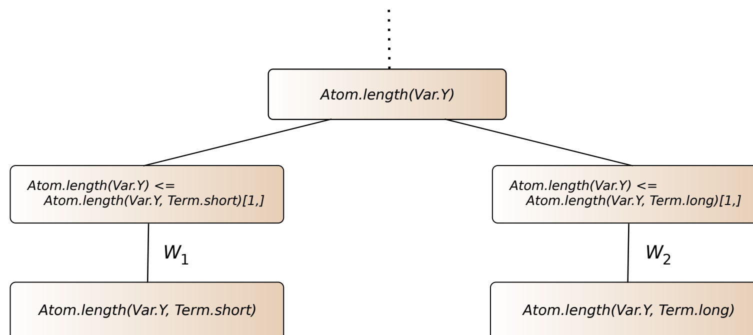

Figure 6.1: The Length feature embedding

The embedding of features (Example 6.4) can be visualized as a graph. Figure 6.1 shows the subgraph of our template, which describes the embedding of one feature - the `length` feature with two possible values - `short` and `long`,

having their unique weights $W_1 \in \mathbb{R}$ and $W_2 \in \mathbb{R}$, respectively.

Listing 6.4: Train feature embedding

```
Y = Var.Y

template.add_rules([
    *[Atom.shape(Y) <= Atom.shape(Y, shape)[1,]
      for shape in shapes],
    *[Atom.length(Y) <= Atom.length(Y, length)[1,]
      for length in lengths],
    *[Atom.sides(Y) <= Atom.sides(Y, side)[1,]
      for side in sides],
    *[Atom.roof(Y) <= Atom.roof(Y, roofs)[1,]
      for roofs in roofs],
    *[Atom.wheels(Y) <= Atom.wheels(Y, wheels)[1,]
      for wheels in wheels],
    *[Atom.loadnum(Y) <= Atom.loadnum(Y, load)[1,]
      for load in loadnums],
    *[Atom.loadshape(Y) <= Atom.loadshape(Y, shape)[1,]
      for shape in loadshapes],
])
```

The rest of the template is shown in Example 6.5, where we define one layer that embeds, aggregates, and combines all features to calculate `carriage`'s embedding. The next layer is aggregating all possible positions of the carriage present in an example set and embeds their associated `carriage`'s values into one embedding to calculate the `train` value. Finally, the `train` atom is again embedded to compute the `direction` value.

Listing 6.5: Train template

```
template.add_rules([
    Atom.carriage(Y)[1,] <= (feature(Y)[1,]
                             for feature in carriage_features),
    Atom.train <= Atom.carriage(Y)[1,],
    Atom.direction <= Atom.train[1,],
])
```

The part of the template defined in Example 6.5 can be visualized as a graph with weights as in Figure 6.2. At the bottom are omitted feature embeddings, which are in the form of a graph in Figure 6.1.

6.1.2 Ousting aggregation with embedding

This proposed template introduces additional information about the order of each train's carriage. The dataset and the template are the same as in the previous section; the only difference is in one rule. By changing the rule

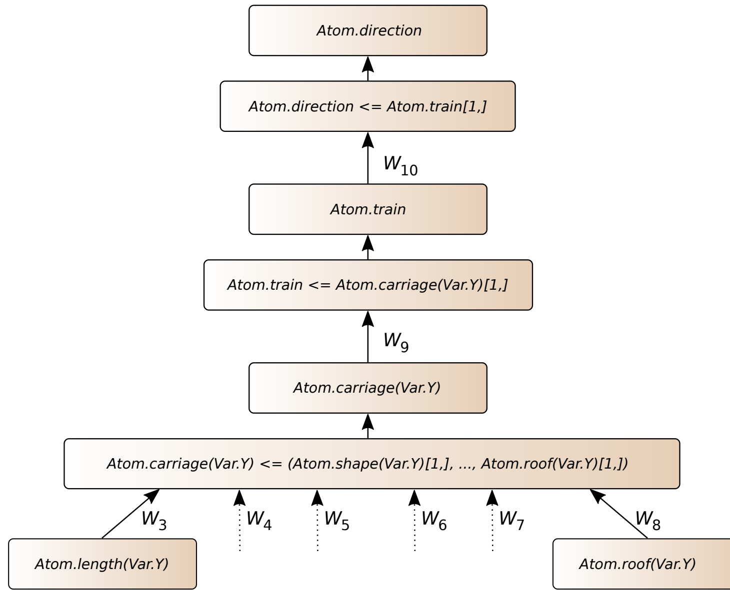

Figure 6.2: The template graph representation

defining the `train` to four rules as in Example 6.6, we can add weights unique to every possible position of carriages. From the observation of the dataset, we assume that there is a maximum of four carriages per train.

Listing 6.6: Rules with information about the carriage's order

```
*[Atom.train <= Atom.carriage(i)[1,] for i in [1, 2, 3, 4]]
```

This considerably small change yields a graph presented in Figure 6.3 that noticeably differs from the previous proposal. This model enables learning of individual embeddings W_{9-12} for each carriage position, compared to having only one embedding for all carriages.

6.1.3 Utilizing Hypergraphs

The last, third proposed approach utilizes PyNeuraLogic's ability to encode hyper-graphs. Such ability is not common in popular GNN frameworks and is usually limited. The PyNeuraLogic library expresses edges as atoms[2], with terms being the connected vertices. The number of terms is not limited; thus, we can express a generalized version of edges - hyperedges by simply putting an arbitrary number of vertices connected by a hyperedge as terms within a predicate.

[2] i.e. there is no actual difference between representing a node or an edge - they are both just logical facts about the world.

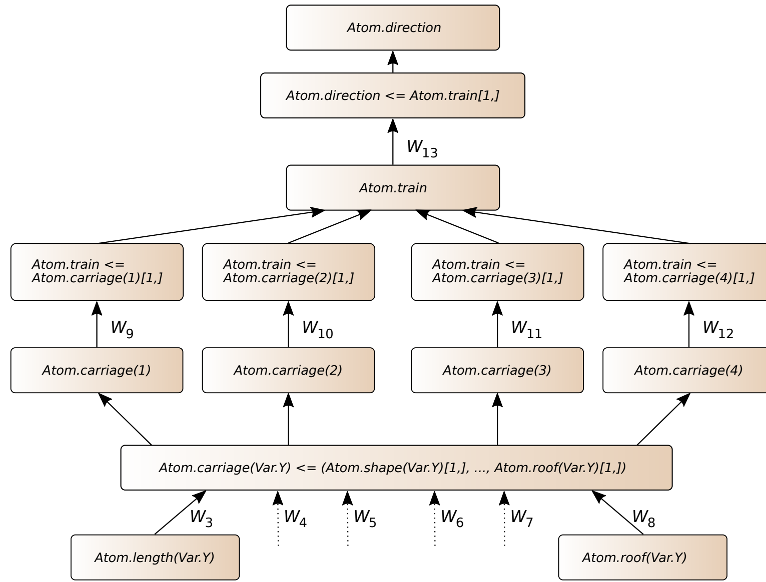

Figure 6.3: The template graph with embedding of different carriages

The Dataset

The dataset is similar to one from the previously proposed approaches, but we use only one example set holding information about the whole world - all train instances. The example set is shared between all queries; thus, we must establish additional terms to contain unique train ids. The introduction of the additional term for features essentially produces edges with a cardinality of three - connecting the train id, the carriage position, and the feature value, , as is shown in Example 6.7.

Listing 6.7: Encoding of features utilizing hypergraph

```
[
    Atom.shape(train_id, 1, "rectangle"),
    Atom.length(train_id, 1, "short"),
    Atom.sides(train_id, 1, "not_double"),
    Atom.roof(train_id, "none"),
    Atom.wheels(train_id, 2),
    Atom.loadshape(train_id, "circle"),
    Atom.loadnum(train_id, 1),
]
```

The query for this approach has to be extended with one term containing the train id as well, as in Example 6.8. This is due to sharing one example for all queries and the resulting need to distinguish which train we are targeting.

Listing 6.8: Parametrized direction query

```
Atom.direction(train_id)[-1.0]
```

The Template

Template 6.9 is inferred from the previous approach. The only change is an additional term for the train id.

Listing 6.9: Template of model utilizing hypergraph

```
*[Atom.shape(X, Y) <= Atom.shape(X, Y, s)[1,] for s in shapes],
...
Atom.carriage(X, Y) <= (atom(X, Y)[1,]
                    for atom in carriage_atoms),
*[Atom.train(X) <= Atom.carriage(X, i)[1,]
                    for i in [1, 2, 3, 4]],
Atom.direction(X) <= Atom.train(X)[1,],
```

The graph of the template has the same structure as the previously proposed model (Figure 6.3), the only difference being in the naming of nodes - nodes have an additional term.

6.2 Learning on knowledge graphs

Via the universal language of PyNeuraLogic, we are also able to encode tasks from the area of Knowledge Base Completion (KBC) [24]. The purpose of such settings is to deduce new information (e.g., property of entity or relations between entities) from the provided knowledge graph containing information about the instance of the world. The knowledge is commonly represented as tuple $(entity, relation, entity)$.

We might, for example, for the given background knowledge of different nations and their relationships, e.g., diplomatic associations, infer the probability of Brazil having treaties with the UK or the probability of properties such as the USA holding protests.

The representation of a GNN-based model for such a KBC setting can be seen in Example 6.10. The atom *predict/2* with arity two is used to compute the probability of properties of the entity (e.g., `Atom.predict(Term.usa, Term.protests)`, and the atom *predict/3* with arity three to compute the probability of a relationship between entities (e.g., `Atom.predict(Term.brazil, Term.treaties, Term.uk)`).

Listing 6.10: Knowledge base learning

```
Atom.embed_nationOne(Var.Nation)[3, 3] <= (
    Atom.embed_relation(Var.Relation)[3, 3],
    Atom.embed_nation(Var.NationTwo)[3, 3],
    Atom.r(Var.Nation, Var.Relation, Var.NationTwo)[3,],
),

Atom.embed_nationTwo(Var.Nation)[3, 3] <= (
    Atom.embed_relation(Var.Relation)[3, 3],
    Atom.embed_nation(Var.NationTwo)[3, 3],
    Atom.r(Var.Nation, Var.Relation, Var.NationTwo)[3,],
),

Atom.predict(Var.Nation, Var.Relation, Var.NationTwo)[1, 3] <= (
    Atom.embed_nationOne(Var.Nation)[3, 3],
    Atom.embed_nationTwo(Var.NationTwo)[3, 3],
    Atom.embed_relation(Var.Relation)[3, 3],
),

Atom.predict(Var.Nation. Var.Property)[1, 3] <= (
    Atom.embed_nation(Var.Nation)[3, 3],
    Atom.embed_property(Var.Property)[3, 3],
),
```

The knowledge $(entity, relation, entity)$ is represented in Example 6.10 as `r(Term.entity, Term.relation, Term.entity)`, and we omit the relation term when representing an entity having a property. We also assume that the template contains embeddings for all entities (nations, properties) and all relations, ensuring all relations and entities have their own learnable parameters. Embeddings, which we omit for the simplicity in the model, are then in the form of rules following the Example 6.11.

Listing 6.11: Embedding of knowledge

```
Atom.embed_relation(Term.relation)[3,] <= Atom.rel(Term.relation)
```

6.3 Beyond GNNs

This section introduces a few of many possible scenarios and concepts that can be expressed in the PyNeuraLogic library and go beyond the possibilities of regular Graph Neural Networks.

6.3.1 Matching Patterns

One of the substantial extensions of vanilla Graph Neural Networks introduced by the PyNeuraLogic library[3] is capturing underlying graph patterns. We can, for instance, recognize nodes that are parts of cycles, such as cycles of the length of three - triangles, presented in Example 6.12.

Listing 6.12: Capturing the pattern of triangles

```
Atom.triangle(Var.X)[1,] <= (
    Atom.edge(Var.X, Var.Y), Atom.feature(Var.Y)[1,],
    Atom.edge(Var.Y, Var.Z), Atom.feature(Var.Z)[1,],
    Atom.edge(Var.Z, Var.X), Atom.feature(Var.X)[1,],
),
```

Another slightly more complex example might be capturing cliques in a graph - i.e. subgraphs that are complete. We present capturing cliques of the size of four in Example 6.13. In the clique example[4], we utilize an atom with a special predicate `alldiff`, which guarantees the specified variables in its terms to have unique values (all different). We use `...` in place of terms, which PyNeuraLogic later substitutes for all variables found in the current rule.

Listing 6.13: Capturing the pattern of a clique of the size of four nodes

```
Atom.clique(Var.X)[1,] <= (
    Atom.feature(Var.X)[1,],
    Atom.edge(Var.X, Var.Y), Atom.feature(Var.Y)[1,],
    Atom.edge(Var.X, Var.Z), Atom.feature(Var.Z)[1,],
    Atom.edge(Var.X, Var.R), Atom.feature(Var.R)[1,],
    Atom.edge(Var.Y, Var.Z), Atom.edge(Var.Y, Var.R),
    Atom.edge(Var.Z, Var.R),
    Atom.special.alldiff(...),
),
```

Distinguishing *k*-regular graphs

Multiple graph structures are not distinguishable by the standard message passing Graph Neural Networks [25]. For instance, the Graph Neural Networks are not able to distinguish between k-regular graphs of the same size, such as the two graphs shown in Figure 6.4. Those graphs are not isomorphic and are both 3-regular, meaning all nodes have precisely three neighbors.

[3] `https://github.com/LukasZahradnik/PyNeuraLogic`

[4] The fully working example of capturing cliques and triangles in the form of Jupyter Notebook can be found at `https://github.com/LukasZahradnik/PyNeuraLogic/blob/master/examples/PatternMatching.ipynb`

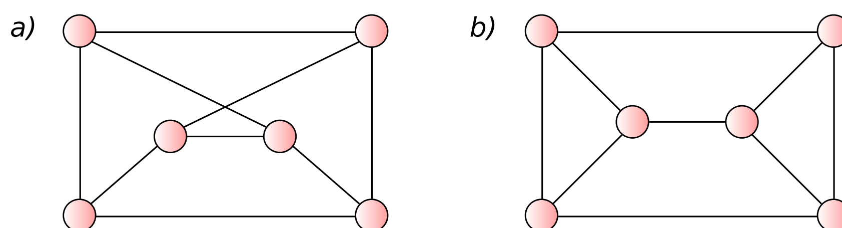

Figure 6.4: 3-regular not isomorphic graphs

When we assign the same features to all nodes, the messages during the update step of message passing GNNs will be identical, resulting in the same features and eventually classifying both graphs as the same class. Such misclassification can be problematic in multiple domains, e.g., chemistry, where two indistinguishable graphs represent two different molecules.

Via the PyNeuraLogic library, we are able to embed the pattern of both graphs or their parts. There are many alternative approaches to distinguish between those two graphs; our presented Example 6.14 utilizes previously shown encoding of triangles (Example 6.12) to capture triangles of graph b, with additional rules aggregating the general graph structure[5]. This results in two different computation graphs where the computation graph for graph a considers only the general rule, but the computation graph for graph b reflects both the general and the triangle rules.

Listing 6.14: Distinguishing between graph a and graph b

```
Atom.triangle(Var.X)[1,] <= (
    Atom.edge(Var.X, Var.Y), Atom.feature(Var.Y)[1,],
    Atom.edge(Var.Y, Var.Z), Atom.feature(Var.Z)[1,],
    Atom.edge(Var.Z, Var.X), Atom.feature(Var.X)[1,],
),

Atom.general(Var.X)[1,] <= Atom.feature(Var.X)[1,],
Atom.general(Var.X)[1,] <= (Atom.edge(Var.X, Var.Y),
                            Atom.feature(Var.Y)[1,]),

Atom.predict <= Atom.general(Var.X)[1,],
Atom.predict <= Atom.triangle(Var.X)[1,],
```

[5]The fully working example of distinguishing k-regular graphs in the form of Jupyter Notebook can be found at `https://github.com/LukasZahradnik/PyNeuraLogic/blob/master/examples/DistinguishingKRegularGraphs.ipynb`

Distinguishing non-regular graphs

There are instances of graphs that are not k-regular nor isomorphic and yet are not distinguishable via the message passing GNNs when their nodes have identical features [25]. An example of such graphs is shown in Figure 6.5. In PyNeuraLogic, we are capable of distinguishing those graphs, for example, via the previously proposed model (Example 6.14) which captures triangles of graph a to distinguish between graphs[6].

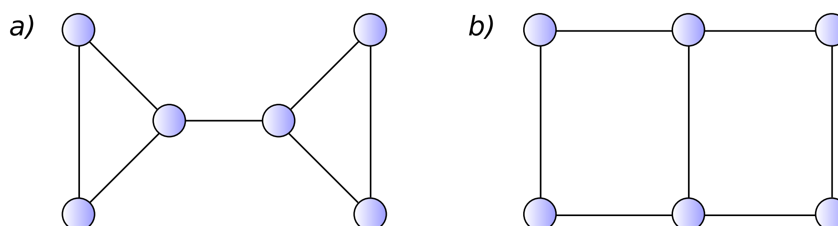

Figure 6.5: Not regular not isomorphic graphs

Another interesting approach of a slightly different extension of vanilla GNNs might be capturing based on the structure and the cardinality of nodes. We can add additional information about the cardinality of each node into examples, for instance, as atoms with predicate's name *cardinality* with two terms - the node id and its cardinality. We can then choose which atom will be aggregated based on its cardinality to distinguish graph a and graph b, as shown in Example 6.15, where we capture only sub-graphs of graphs.

The `a_graph` captures a triangle (`Var.X`, `Var.Y`, `Var.Z`) connected to one node (`Var.T`) with a cardinality of three. In contrast, the `b_graph` captures a cycle of length of four (`Var.X`, `Var.Y`, `Var.Z`, `Var.T`) which has to satisfy required cardinalities.

Listing 6.15: Distinguishing between graphs based on their cardinality

```
Atom.a_graph(Var.X) <= (
    Atom.edge(Var.X, Var.Y), Atom.cardinality(Var.Y, 2)[1,],
    Atom.edge(Var.Y, Var.Z), Atom.cardinality(Var.Z, 2)[1,],
    Atom.edge(Var.Z, Var.X), Atom.cardinality(Var.X, 3)[1,],
    Atom.edge(Var.X, Var.T), Atom.cardinality(Var.T, 3)[1,],
    Atom.special.alldiff(...),
),
Atom.b_graph(Var.X) <= (
    Atom.edge(Var.X, Var.Y), Atom.cardinality(Var.Y, 2)[1,],
    Atom.edge(Var.Y, Var.Z), Atom.cardinality(Var.Z, 2)[1,],
    Atom.edge(Var.Z, Var.T), Atom.cardinality(Var.T, 3)[1,],
```

[6] All fully working examples of distinguishing non-regular graphs from this section in the form of Jupyter Notebook can be found at `https://github.com/LukasZahradnik/PyNeuraLogic/blob/master/examples/DistinguishingNonRegularGraphs.ipynb`

```
    Atom.edge(Var.T, Var.X), Atom.cardinality(Var.X, 3)[1,],
    Atom.special.alldiff(...),
),
Atom.predict <= Atom.a_graph(Var.X)[1,],
Atom.predict <= Atom.b_graph(Var.X)[1,],
```

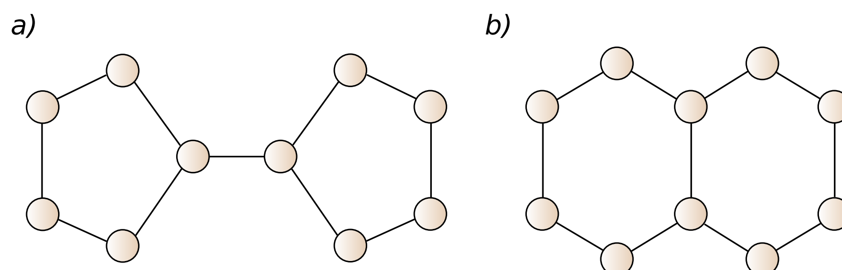

Figure 6.6: Bicyclopentyl and Decalin

Figure 6.6. shows two graphs, *a* and *b*, representing a real-world structure of two molecules *Bicyclopentyl* and *Decalin*, respectively. The message passing GNN cannot again distinguish between graphs under the condition of identical features for all nodes [25]. In PyNeuraLogic, we can embed, for example, the cycle of length five present in graph *a* and thus distinguish those instances, such as is shown in Example 6.16.

Listing 6.16: Capturing the cycle of the length of five

```
Atom.cycle_of_the_length_of_five(Var.X)[1,] <= (
    Atom.edge(Var.X, Var.Y), Atom.feature(Var.Y)[1,],
    Atom.edge(Var.Y, Var.Z), Atom.feature(Var.Z)[1,],
    Atom.edge(Var.Z, Var.R), Atom.feature(Var.R)[1,],
    Atom.edge(Var.R, Var.S), Atom.feature(Var.S)[1,],
    Atom.edge(Var.S, Var.X), Atom.feature(Var.X)[1,],
    Atom.special.alldiff(...),
),
```

6.3.2 Expressing Features

Features in Graph Neural Network frameworks and libraries are usually represented in the form of numerical tensors. Such representation loses the information about the meaning of each feature element, and the users are generally forced to use the exact dimension of the feature tensor for each node feature or/and edge feature.

PyNeuraLogic feature values are not limited to be numerical nor to be contained in a fixed-size tensor. We have already shown in Sec. 6.1 that we can encode a feature's as a string inside of fact's terms and that we can assign

a name to a feature via predicate's name, e.g., `shape`, `roof`, which simplifies the readability and the understanding of models compared to the numeric tensor representations.

Nodes in PyNeuraLogic can also have an arbitrary number of features of one type, i.e. values of the same feature. To proceed with our trains example, we might, for instance, propose a new carriage, which is constructed from two different shapes and not only one, as is shown in Example 6.17. This enables users to add features with ease and without the need to expand the feature vectors of different nodes. If we were to replicate this feature in regular Graph Neural Network frameworks, we might soon end up with high dimensional tensors leading to problems with memory.

Listing 6.17: One node having two features of the same type

```
Atom.shape(0, Term.ellipse), Atom.shape(0, Term.rectangle), ...
```

Another property of PyNeuraLogic is having features' values not limited to being one fixed tensor. We can mix different types and dimensions of inputs in one model, e.g., scalar and vector feature, as in Example 6.18, where we encoded a scalar value 1.0 and a vector value of the length of three with values 1.0, 2.0, and 3.0 for node with id 0.

Listing 6.18: Example of scalar and vector features

```
Atom.scalar_feature(0)[1.0],
Atom.vector_feature(0)[[1.0, 2.0, 3.0]]
```

Both features can then be embedded into one value, such as a scalar, as presented in Example 6.19. It is however important to make sure that aggregated embeddings have the exact dimensions; for that reason, we are embedding the vector value feature into a scalar first to be able to combine it with the scalar value feature later. We might also embed the scalar value feature into a vector instead since we are naturally not limited to scalar embedding.

Listing 6.19: Template with scalar and vector features

```
Atom.feature_embed(Var.X)[1,] <= (
    Atom.scalar_feature(Var.X)[1,],
    Atom.vector_feature(Var.X)[1, 3]
)
```

6.3.3 Extending Graph Outputs

We have already previously mentioned that via the proposed library, we are able to get the output from an arbitrary rule and even an arbitrary atom. For example, we can have multiple layers which we can query to obtain information about different properties of the queried entity, as shown to Example 6.20.

Listing 6.20: Layers embedding different properties

```
Atom.color(Var.X)[1,] <= Atom.layer_1(Var.X),
Atom.shape(Var.X)[2,] <= Atom.layer_1(Var.X),
```

The output is also not restricted to be of one fixed dimension, as is usual in regular GNN frameworks. In PyNeuralogic, different atoms can output tensors of different dimensions, and we can achieve such behavior even on atoms with the same predicate's name. In Example 6.21, we propose the atom `predict`. Its output depends on helper atoms arbitrarily named `scalar` and `vector`, which serve for conditional computation to determine if the value of atom `embed_layer` will be embedded via scalar or vector, respectively; thus yielding tensors of different dimensions.

Listing 6.21: Output of different dimensions

```
Atom.predict[1,] <= (Atom.embed_layer, Atom.scalar),
Atom.predict[2,] <= (Atom.embed_layer, Atom.vector),
```

6.3.4 Heterogenous Graphs

Most GNN models do not consider graphs being heterogeneous, and GNN frameworks support heterogenous only to some degree. Via PyNeuraLogic, we can easily encode heterogeneous graphs with an arbitrary number of node (e.g., Example 6.22) and edge (e.g., Example 6.23) classes.

Listing 6.22: GCN Layer aggregating nodes of the same type as the central node's type

```
Atom.layer(Var.X) <= (
    Atom.feature(Var.Y),
    Atom.node_type(Var.Y, Var.Type),
    Atom.node_type(Var.X, Var.Type),
    Atom.edge(Var.X, Var.Y),
),
```

Listing 6.23: GCN Layer with capturing edge type in Var.Type

```
Atom.layer(Var.X) <= (
    Atom.feature(Var.Y),
    Atom.edge(Var.X, Var.Y, Var.Type),
),
```

We might also encode heterogeneous graphs less universally by utilizing predicate as the name of types, as is showcased in Example 6.24.

Listing 6.24: Node types via predicates

```
Atom.layer(Var.X) <= (
    Atom.feature(Var.Y),
    Atom.my_type(Var.Y),
    Atom.my_type(Var.X),
    Atom.edge(Var.X, Var.Y),
),
```

We are even able to create a new type from merging types and create hierarchies of types. In Example 6.25, the `car` class captures nodes with types of types `bus` and `pickup`. The `vehicle` class is a superset of the `car` class with additional capturing of nodes of a `bicycle` type.

Listing 6.25: Creating hierarchies of types

```
Atom.type(Var.X, Term.car) <= Atom.type(Var.X, Term.bus),
Atom.type(Var.X, Term.car) <= Atom.type(Var.X, Term.pickup),

Atom.type(Var.X, Term.vehicle) <= (
    Atom.type(Var.X, Term.car)
),
Atom.type(Var.X, Term.vehicle) <= (
    Atom.type(Var.X, Term.bicycle)
),
```

We can attach features to all proposed constructs above. Edge and node types can have features, as is shown in Example 6.26.

Listing 6.26: Types' features

```
Atom.layer(Var.X) <= (
    Atom.feature(Var.Y),
    Atom.type_feature(Var.NodeType),
    Atom.edge_feature(Var.EdgeType),
    Atom.type(Var.Y, Var.NodeType),
    Atom.edge(Var.X, Var.Y, Var.EdgeType),
),
```

Nodes and edges in PyNeuraLogic are not limited to only one class. Such property can result in utilizing all features of classes linked to the node while

computing embeddings. We also can define features globally for types and the specific node of the class. This way, we can have features specific to the node but dependant on its type, as shown in Example 6.27.

Listing 6.27: Types' features

```
Atom.layer(Var.X) <= (
    Atom.type(Var.X, Var.Type),
    Atom.type_feature(Var.Type),
    Atom.type_node_feature(Var.X, Var.Type),
),
```

6.3.5 Heterophily Setting

GNN models usually consider homophily in the graph - frequently, nodes of similar classes are connected with each other [26]. This setting does not capture multiple problems adequately, where there is a heterophily amongst connected nodes - mainly nodes of different classes are connected, resulting in low accuracies of classifications. An example of such a problem might be the *arxiv* paper citation network [27], where each paper represents a node, citation an edge, and each node is labeled by the year it was written.

Recently, there has been a development of models specifically dealing with heterophily graphs, such as the H2GCN model [28]. The H2GCN model combines three key designs - embedding the central node separately, aggregating higher-order neighborhoods instead of the direct neighbors only, and combining intermediate representations of nodes at the final layer.

All of the listed vital ideas can be effortlessly encoded via versatile constructs of PyNeuraLogic, without the need for another specifically crafted black box. The separate embedding can be encoded as in Example 6.28, the aggregation of the higher-order (the second-order) neighborhood in Example 6.29 and combining previous layers in the last layer in Example 6.30. Note that we do not specify the weights here for readibility purposes.

Listing 6.28: Separate embedding of the central node

```
Atom.layer(Var.X) <= Atom.feature(Var.X)
```

Listing 6.29: Higher-order neighborhoods embedding

```
Atom.layer(Var.X) <= (
    Atom.feature(Var.Z),
    Atom.edge(Var.X, Var.Y),
    Atom.edge(Var.Y, Var.Z),
    Atom.special.alldiff(...),
),
```

Listing 6.30: The last layer combines the representations of previous layers

```
Atom.last_layer(Var.X) <= (
    Atom.layer(Var.X),
    Atom.layer_2(Var.X),
    Atom.layer_n(Var.X),
),
```

Chapter 7

Conclusions

In this work, we proposed and implemented a machine learning library with a custom language based on relational logic to enhance graph neural networks beyond their current state. We showed how some GNN models can be encoded in our proposed language via weighted clauses, essentially uncovering the underlying models' architectures, giving us the ability to adjust and extend the said architectures with ease.

Comparison with other libraries/frameworks

To quickly summarize, our solution supports both homogenous and heterogeneous graphs, whereas popular GNN libraries consider mainly only homogeneous graphs (some include one heterogeneous model). We also support expressing hypergraphs, which are not frequently supported in other GNN frameworks (usually only one specific model). Additionally, the library allows for introduction of completely new embedding propagation schemes upon these structures.

As a result, the common GNNs have problems with distinguishing some graph classes, which our solution can solve by utilizing higher-order pattern matching. While there exist some specialized GNN models explicitly targeting this issue, our solution is more general and can encode and propagate messages through arbitrary relational patterns. Via pattern matching, we are able to match various complex subgraphs and embed them, which covers numerous use-cases in domains such as chemistry or social networks. We also add more

expressiveness in encoding node's and edge's features. Our solution enables us to query any (logical) atom in the model, which leads to the possibility of retrieving different underlying representations from different parts of the model.

The shortcomings of our solution are mainly related to computational efficiency, where popular GNN libraries and frameworks utilize lower-level, well-optimized computing libraries. Our solution is noticeably slower for problems that require operating on large input tensors due to the lack of optimized multiplications and other tensor operations.

Future Work

Future work might be focused on more efficient computational operations, which would solve the trade-off between speed and expressiveness in some instances. Such improvement might be achieved, for example, by utilizing Torch for heavy bulk computations.

Appendix A

Bibliography

[1] Zhou, Jie, Cui, Ganqu, Hu, Shengding, et al. Graph Neural Networks: A Review of Methods and Applications[EB/OL]. 2021.

[2] Wu, Zonghan, Pan, Shirui, Chen, Fengwen, et al. A Comprehensive Survey on Graph Neural Networks[J]. IEEE Transactions on Neural Networks and Learning Systems, 2021, 32(1):4–24.

[3] Fey, Matthias and Lenssen, Jan E. Fast Graph Representation Learning with PyTorch Geometric[C]. In: ICLR Workshop on Representation Learning on Graphs and Manifolds. 2019.

[4] Wang, Minjie, Zheng, Da, Ye, Zihao, et al. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks[EB/OL]. 2020.

[5] Grattarola, Daniele and Alippi, Cesare. Graph Neural Networks in TensorFlow and Keras with Spektral[EB/OL]. 2020.

[6] Data61, CSIRO's. StellarGraph Machine Learning Library[EB/OL]. 2018.

[7] Smullyan, R.M. First-order Logic[M]. Dover, 1995.

[8] Sourek, Gustav, Zelezny, Filip, and Kuzelka, Ondrej. Beyond Graph Neural Networks with Lifted Relational Neural Networks[EB/OL]. 2020.

[9] Sourek, Gustav, Zelezny, Filip, and Kuzelka, Ondrej. Lossless Compression of Structured Convolutional Models via Lifting[EB/OL]. 2021.

[10] Ullman, Jeffrey D. Principles of database and knowledge-base systems[M]. Computer Science Press, 1989.

[11] Mihalkova, Lilyana and Getoor, Lise. Lifted Graphical Models: A Survey[EB/OL]. 2011.

[12] Collobert, R., Kavukcuoglu, K., and Farabet, C. Torch7: A Matlab-like Environment for Machine Learning[C]. In: BigLearn, NIPS Workshop. 2011.

[13] Paszke, Adam, Gross, Sam, Massa, Francisco, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library[EB/OL]. 2019.

[14] Studeny, Jan. Learning Relevant Reasoning Patterns with Neuro-Logic Programming[EB/OL]. 2017.

[15] De Raedt, Luc, Kimmig, Angelika, and Toivonen, Hannu. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery[C]. 2007. 2462–2467.

[16] Manhaeve, Robin, Dumancic, Sebastijan, Kimmig, Angelika, et al. DeepProbLog: Neural Probabilistic Logic Programming[EB/OL]. 2019-07.

[17] Kipf, Thomas N. and Welling, Max. Semi-Supervised Classification with Graph Convolutional Networks[EB/OL]. 2017.

[18] Hamilton, William L., Ying, Rex, and Leskovec, Jure. Inductive Representation Learning on Large Graphs[EB/OL]. 2018.

[19] Xu, Keyulu, Hu, Weihua, Leskovec, Jure, et al. How Powerful are Graph Neural Networks?[EB/OL]. 2019.

[20] Neubig, Graham, Dyer, Chris, Goldberg, Yoav, et al. DyNet: The Dynamic Neural Network Toolkit[J]. arXiv preprint arXiv:1701.03980, 2017.

[21] Weisfeiler, B. On construction and identification of graphs, vol 558[M]. Springer, 2006.

[22] Yang, Zhilin, Cohen, William W., and Salakhutdinov, Ruslan. Revisiting Semi-Supervised Learning with Graph Embeddings[EB/OL]. 2016.

[23] Michie, Donald, Muggleton, Stephen, Page, David, et al. To the international computing community: A new east-west challenge[EB/OL]. 1994.

[24] Kadlec, Rudolf, Bajgar, Ondrej, and Kleindienst, Jan. Knowledge Base Completion: Baselines Strike Back[EB/OL]. 2017.

[25] Sato, Ryoma. A Survey on The Expressive Power of Graph Neural Networks[EB/OL]. 2020.

[26] Zhu, Jiong, Rossi, Ryan A., Rao, Anup, et al. Graph Neural Networks with Heterophily[EB/OL]. 2021.

[27] Hu, Weihua, Fey, Matthias, Zitnik, Marinka, et al. Open Graph Benchmark: Datasets for Machine Learning on Graphs[EB/OL]. 2021.

[28] Zhu, Jiong, Yan, Yujun, Zhao, Lingxiao, et al. Beyond Homophily in Graph Neural Networks: Current Limitations and Effective Designs[EB/OL]. 2020.

Appendix B

Contents of attached CD

```
pyneuralogic.......The source code of the implemented library
thesis
  source.....................The source code of this thesis
  thesis.pdf.......................The thesis in PDF file
```