

**Bachelor Thesis**



**Czech  
Technical  
University  
in Prague**

**F3**

**Faculty of Electrical Engineering  
Department of Cybernetics**

# **Developing a Domain-Specific Differentiable Functional Language**

**Jakub Kraus**

**Supervisor: Ing. Gustav Šír, Ph.D.**

**Study program: Open Informatics**

**Specialisation: Artificial Intelligence and Computer Science**

**May 2024**



## I. Personal and study details

Student's name: **Kraus Jakub** Personal ID number: **499211**  
Faculty / Institute: **Faculty of Electrical Engineering**  
Department / Institute: **Department of Cybernetics**  
Study program: **Open Informatics**  
Specialisation: **Artificial Intelligence and Computer Science**

## II. Bachelor's thesis details

Bachelor's thesis title in English:

**Developing a Domain-Specific Differentiable Functional Language**

Bachelor's thesis title in Czech:

**Vývoj doménov specifického diferencovatelného funkcionálního jazyka**

Guidelines:

The subject of this bachelor thesis is to develop a small, domain-specific, differentiable programming language building upon simple functional programming principles. The language should support basic algebraic expressions and functions commonly used in deep learning, and should be embedded within Python via suitable syntax and operator overloading. Ultimately, it is to be used as a part of a wider framework for differentiable logic programming [1], with the aim to allow for a more elegant and efficient encoding of advanced deep learning architectures [2]. The student is expected to:

1. Study up the core principles underlying modern deep learning architectures, and briefly review existing related frameworks (e.g. [3]).
2. Get acquainted with the "NeuraLogic" framework [1] for differentiable logic programming, both the Java backend [4] and Python frontend [5].
3. Propose suitable syntax and semantics of your new language, following the functional paradigm.
4. Implement the respective frontend and backend capabilities for parsing and interpreting expressions written in your language.
5. Focus on computational efficiency of inference and learning, and propose solutions for improvements.

Bibliography / sources:

- [1] Sourek, Gustav, et al. "Lifted relational neural networks: Efficient learning of latent relational structures." *Journal of Artificial Intelligence Research* 62 (2018): 69-100.  
[2] Šourek, Gustav, Filip Železný, and Ondřej Kuželka. "Beyond graph neural networks with lifted relational neural networks." *Machine Learning* 110.7 (2021): 1695-1738.  
[3] HaskTorch library introduction: <https://github.com/hasktorch/>  
[4] NeuraLogic framework backend: <https://github.com/GustikS/NeuraLogic>  
[5] PyNeuraLogic frontend: <https://github.com/LukasZahradnik/PyNeuraLogic>

Name and workplace of bachelor's thesis supervisor:

**Ing. Gustav Šír, Ph.D. Intelligent Data Analysis FEE**

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **10.01.2023** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **22.09.2024**

Ing. Gustav Šír, Ph.D.  
Supervisor's signature

prof. Ing. Tomáš Svoboda, Ph.D.  
Head of department's signature

prof. Mgr. Petr Páta, Ph.D.  
Dean's signature

### III. Assignment receipt

The student acknowledges that the bachelor'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 bachelor's thesis, the author must state the names of consultants and include a list of references.

\_\_\_\_\_  
Date of assignment receipt

\_\_\_\_\_  
Student's signature

## Acknowledgements

I would like to thank my supervisor Ing. Gustav Šír, Ph.D. for his never ending patience, valuable guidance and understanding, that made this thesis possible.

I declare the use of ChatGPT 4.0, <https://chatgpt.com/>, for formatting the text, formulating my thoughts, and brainstorming ideas throughout this thesis.

## Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 24, 2024

.....  
signed Jakub Kraus

## Abstract

Deep Neural Networks have revolutionized many fields but often struggle with capturing complex, irregular structures found in real-world data. To address this, relational graph neural network frameworks were created, enabling processing and learning even from such patterns. This work aims to develop a new domain-specific differentiable functional language to support logic programming within one such framework.

The proposed language simplifies and enhances problem declaration, making it more intuitive for users. It also expands the expressive capabilities of relational models, allowing for a richer and more flexible data representation. Through multiple case-specific examples, the effectiveness of the new syntax is demonstrated, showing that it retains the computational efficiency of the original framework while significantly improving user experience and reducing development time. Result of this work provides a robust foundation for integrating logical reasoning with neural network learning, facilitating more accurate and interpretable models in various applications such as natural language processing, knowledge graph completion, and bioinformatics.

**Keywords:** Lifted Relation Neural Networks, Relational Logic

**Supervisor:** Ing. Gustav Šír, Ph.D.

## Abstrakt

Hluboké neuronové sítě způsobily revoluci v mnoha oborech, ale často mají problémy se zachycením složitých, nepravidelných struktur, které se vyskytují v reálných datech. Pro řešení tohoto problému byly vytvořeny frameworky relačních grafových neuronových sítí, které umožňují zpracovávat a učit se i z takových vzorů. Cílem této práce je vyvinout nový doménově specifický diferencovatelný funkcionální jazyk pro podporu logického programování v rámci jednoho takového frameworku.

Navržený jazyk zjednodušuje a vylepšuje deklarování problémů a činí jej pro uživatele intuitivnějším. Rozšiřuje také vyjadřovací schopnosti relačních modelů, což umožňuje bohatší a flexibilnější reprezentaci dat. Na několika konkrétních příkladech je demonstrována efektivita nové syntaxe, která zároveň zachovává výpočetní efektivitu původního frameworku a zároveň výrazně zlepšuje uživatelský komfort a zkracuje dobu vývoje. Výsledek této práce poskytuje robustní základ pro integraci logického uvažování s učením neuronových sítí, což vede k přesnějším a lépe interpretovatelným modelům aplikací, jako je zpracování přirozeného jazyka, doplňování znalostních grafů a bioinformatika.

**Klíčová slova:** Lifted Relation Neural Networks, Relační Logika

**Překlad názvu:** Vývoj doménově specifického diferencovatelného funkcionálního jazyka

# Contents

<b>1 Introduction</b>	<b>1</b>	5.5.2 Example 2: Graph . . . . .	27
<b>2 Deep Learning</b>	<b>3</b>	<b>6 Conclusions</b>	<b>31</b>
2.1 Neural Networks . . . . .	3	<b>Bibliography</b>	<b>33</b>
2.1.1 Basic Concepts . . . . .	3	<b>Appendices</b>	<b>35</b>
2.1.2 Structure of Perceptron . . . . .	3	<b>Contents of Attached CD</b>	<b>37</b>
2.1.3 Structure of Neural Networks . . . . .	4		
2.1.4 Training of Neural Networks . . . . .	4		
2.2 Differentiable Programming . . . . .	6		
2.2.1 Automatic Differentiation . . . . .	6		
2.2.2 Applications . . . . .	6		
2.3 Dynamic Computation Graphs . . . . .	6		
2.4 Existing Frameworks . . . . .	7		
2.4.1 PyTorch . . . . .	7		
2.4.2 TensorFlow . . . . .	7		
2.4.3 Haskell . . . . .	8		
<b>3 Deep Relational Learning</b>	<b>9</b>		
3.1 Graph Neural Networks . . . . .	9		
3.1.1 Types of GNNs . . . . .	10		
3.2 Relational Logic . . . . .	10		
3.2.1 Uses in Machine Learning . . . . .	10		
3.3 Lifted Relational Neural Networks . . . . .	11		
<b>4 PyNeuraLogic</b>	<b>13</b>		
4.1 Introduction . . . . .	13		
4.1.1 Structure of Framework . . . . .	14		
4.2 Python Front End . . . . .	14		
4.2.1 Rule . . . . .	14		
4.2.2 Dataset . . . . .	15		
4.2.3 Template . . . . .	15		
4.2.4 Query . . . . .	16		
4.2.5 Example . . . . .	17		
4.3 Java Back End . . . . .	18		
4.3.1 Internal Structure . . . . .	18		
<b>5 Functional Language Proposition</b>	<b>21</b>		
5.1 Need of New Syntax . . . . .	21		
5.2 Approach Analysis . . . . .	22		
5.2.1 Editing Python . . . . .	22		
5.2.2 File Encoding . . . . .	23		
5.2.3 Enhancing Python Syntax . . . . .	24		
5.3 Storing and managing input data . . . . .	24		
5.3.1 Functional Tree . . . . .	25		
5.3.2 Functional Container . . . . .	25		
5.4 Data Processing and Analysis . . . . .	25		
5.4.1 Functional Call Simulator . . . . .	26		
5.5 Examples . . . . .	26		
5.5.1 Example 1: Four Nodes . . . . .	26		

## Figures

## Tables

2.1 Perceptron . . . . .	4
2.2 Activation functions . . . . .	5
3.1 Node embedding update . . . . .	10
5.1 Example 1: Template . . . . .	27
5.2 Example 1: Dataset . . . . .	28
5.3 Example 2: Template . . . . .	29
5.4 Example 2: Dataset . . . . .	29





# Chapter 1

## Introduction

Artificial Intelligence (AI) has seen a remarkable boom in recent years, transforming from a niche field to a crucial driving force in technology and society. This surge in AI has been fueled by advances in machine learning, particularly through the development and application of neural networks. Inspired by the human brain, neural networks have enabled machines to recognize patterns, learn from data, and make decisions in ways that were previously impossible.

The rise of neural networks began with the concept of artificial neurons, which mimic the way biological neurons process information. These artificial neurons are organized into layers, forming what is known as a neural network. When these networks are deep, meaning they have many layers, they are referred to as deep learning models. Deep learning has been at the heart of many recent AI breakthroughs, powering applications such as image and speech recognition, natural language processing, and even autonomous vehicle development.

Several frameworks have been developed to facilitate the creation and deployment of neural networks. Popular ones include TensorFlow<sup>1</sup>, PyTorch<sup>2</sup>, and Keras<sup>3</sup>. These frameworks have made it accessible for developers and researchers to leverage the power of neural networks, contributing to the rapid growth and application of AI technologies across various domains. However, as the complexity and demand for AI systems grow, there is a need to incorporate more sophisticated reasoning and decision-making capabilities. This is where the concept of neural-symbolic integration comes into play, blending neural networks with symbolic logic.

PyNeuraLogic<sup>4</sup> is a framework that stands out in this context by integrating symbolic logic with neural networks. Traditional neural networks excel at pattern recognition and learning from data, but often struggle with tasks requiring structured reasoning and understanding complex relationships.

---

<sup>1</sup><https://www.tensorflow.org/>

<sup>2</sup><https://pytorch.org/>

<sup>3</sup><https://keras.io/>

<sup>4</sup><https://github.com/LukasZahradnik/PyNeuraLogic>

Symbolic logic, on the other hand, is adept at handling such tasks due to its rule-based nature and ability to represent knowledge explicitly.

By combining these two paradigms, PyNeuraLogic allows for the creation of models that benefit from the strengths of both neural networks and symbolic logic. This integration enables more robust AI systems capable of handling tasks that require both learning from data and logical reasoning. For example, in areas like knowledge graph completion or relational learning, where understanding the relationships between different entities is crucial, PyNeuraLogic can provide significant advantages.

The purpose of this thesis is to develop a new domain-specific differentiable functional language to support the PyNeuraLogic framework. The new language aims to make problem declaration more intuitive and expressive, while also preserving computational efficiency. To achieve this, the thesis will review existing deep learning frameworks, provide a detailed description of PyNeuraLogic, propose a new syntax and semantics and implement the syntax.

The following chapters will delve into the various aspects of deep learning and deep relational learning, setting the stage for understanding the advancements proposed in this thesis. Chapter 2 will provide an in-depth overview of deep learning, covering the fundamental concepts, structures, and training processes of neural networks. It will also discuss differentiable programming, dynamic computation graphs, and existing frameworks.

# Chapter 2

## Deep Learning

Deep learning is a type of machine learning that uses artificial neural networks to model complex patterns in data. It has become increasingly popular in recent years due to its success in many fields such as image generation<sup>1</sup>, natural language processing<sup>2</sup>, and game playing<sup>3</sup>.

Deep learning models, like convolutional neural networks (CNNs) and recurrent neural networks (RNNs), have achieved remarkable results in previously difficult tasks for computers. This chapter will explore the basics of neural networks, the concept of differentiable programming, and dynamic computational graphs, and will review some popular frameworks used in deep learning, including Haskell and PyTorch.

### 2.1 Neural Networks

#### 2.1.1 Basic Concepts

Neural networks are computational models inspired by the structure of the human brain. They consist of layers of interconnected nodes or neurons. Each neuron takes some inputs, processes them, and passes the output to the next layer of neurons. The simplest type of neural network is the feed-forward neural network, where connections do not form cycles.[10]

#### 2.1.2 Structure of Perceptron

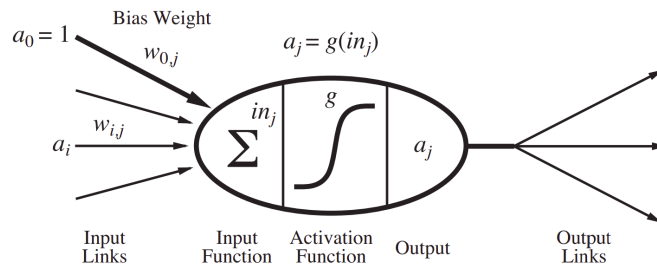
The building block of a neural network is the perceptron. A perceptron takes several numbers as inputs and produces a single number as the output. It does this by calculating a weighted sum of the inputs and bias (typically notated as 0th or  $(n+1)$ th input), and then applying an activation function.[10]

---

<sup>1</sup>E.g.: <https://www.midjourney.com/>

<sup>2</sup>E.g. <https://chatgpt.com/>

<sup>3</sup>E.g. <https://deepmind.google/discover/blog/agent57-outperforming-the-human-atari-benchmark>



**Figure 2.1:** Perceptron design from [10]

$$a_j = g\left(\sum_{i=0}^n w_{i,j} \cdot a_i\right) \quad (2.1)$$

A mathematical definition of perceptron can be shown, see Eq. 2.1.  $a_j$  is the output of perceptron.  $g$  is the activation function.  $w_{i,j}$  is the weight of input  $i$ . And  $a_i$  is the input. This notation of the perceptron is designed to clearly illustrate how to connect the output of this perceptron (or more precisely, a layer of perceptrons) to the next layer of perceptrons.

### 2.1.3 Structure of Neural Networks

Feed-forward neural networks are organized into layers. The input layer receives the initial data, the hidden layers process the data, and the output layer produces the final result. Hidden layers apply non-linear transformations to the data using activation functions. Common activation functions include the sigmoid function, hyperbolic tangent (tanh), rectified linear unit (ReLU), and others, see Fig. 2.2.

### Common Types of Neural Networks

**Multilayer Perceptrons (MLPs):** These are simple feed-forward neural networks with one or more hidden layers. They are good for basic tasks but struggle with complex data like images and sequences.

**Convolutional Neural Networks (CNNs):** CNNs are designed for processing grid-like data such as images. They use convolutional layers to detect features like edges, textures, and objects.

**Recurrent Neural Networks (RNNs):** RNNs are designed for sequence data. They use feedback connections to process data that comes in sequences, making them ideal for tasks like language modeling and time series prediction.

### 2.1.4 Training of Neural Networks

The process has 2 key steps: forward pass and backpropagation. Forward pass calculates the output given simply by layer formulas. Backpropagation



**Figure 2.2:** Activation functions from <https://www.v7labs.com/blog/neural-networks-activation-functions>

takes the difference (Error) between the output and the desired output. Using differentiation and chain rule the error is backpropagated to the weights in each layer. These weights can be updated in the direction to improve the output. [1]

1. Initialize the weights and biases of the network to small random values
2. Present a training example to the network's input layer
3. Propagate the input forward through the network to compute the output of each layer
4. Compare the network's output to the desired output and calculate an error value for each output neuron
5. Propagate the errors backward through the network, from the output layer to the hidden layers
6. Use the backward propagated errors to update the weights and biases of the network
7. Repeat steps 2-6 for each example in the training set, then repeat the entire process for multiple epochs until the network converges

This process can be improved by batching, connection drop-outs, and several methods to enhance gradient descent (for example using momentum).[1]

## 2.2 Differentiable Programming

Differentiable programming is defined as "a programming paradigm in which complex computer programs (including those with control flows and data structures) can be differentiated end-to-end automatically, enabling gradient-based optimization of parameters in the program." [2]

In other words, the parameters (e.g. weights in Neural Network) are found automatically using differentiation. This is essential in deep learning because it enables the use of backpropagation to train neural networks.

But differentiable programming generalizes the idea of neural networks to any differentiable program. This means that any program, not just neural networks, can be trained using gradient descent if it is differentiable. [2]

### 2.2.1 Automatic Differentiation

Automatic differentiation (AD, also auto-diff [2]) is a key component of differentiable programming. It calculates derivatives of functions efficiently and accurately. There are two main modes of auto-diff: forward mode and reverse mode. Reverse mode is particularly useful for deep learning because it computes the gradient of the output with respect to all inputs in a single pass, which is essential for backpropagation. [18]

Both forward-mode and reverse-mode auto-diff rely on the fact that all numerical computations can be decomposed into a sequence of elementary operations, for which the derivatives are known. By applying the chain rule, the derivatives of the overall function can be computed efficiently. [2]

Before automatic differentiation (auto-diff), researchers had to manually compute gradients for functions they wanted to optimize, which was tedious and demanded repeating with every function change. Auto-diff revolutionized this process by allowing users to quickly and creatively experiment with functions without deriving gradients manually.[2]

### 2.2.2 Applications

Differentiable programming allows the integration of machine learning models into larger systems. For example, it can be used to optimize control systems, solve differential equations, and even train neural networks within physical simulations.[5]

## 2.3 Dynamic Computation Graphs

Dynamic computational graphs (DCGs) are a type of computational graph (Neural network architecture) where the structure can change during runtime.

This contrasts with static computational graphs, where the structure is fixed before execution. [7]

DCGs provide flexibility and allow for more complex models that can adapt to different inputs. They are particularly useful for tasks where the input size or structure can vary, such as natural language processing with variable-length sentences or dynamic batch sizes or where the graph structure changes over time, such as in social networks where friendships evolve or in financial networks where trading relationships change [8]

Frameworks like PyTorch and TensorFlow (in eager execution mode) support dynamic computational graphs. This means that the graph is constructed on-the-fly as operations are called, which makes debugging and model development easier and more intuitive.<sup>4</sup>

## 2.4 Existing Frameworks

### 2.4.1 PyTorch

PyTorch is an open-source deep learning framework developed by Meta AI Research lab. It is known for its flexibility, ease of use, and support for dynamic computational graphs. PyTorch builds the computational graph on-the-fly, which makes the programming intuitive and easy to debug.

It also has a rich ecosystem of libraries for various tasks like computer vision (torchvision), natural language processing (torchtex), and reinforcement learning (stable-baselines3). PyTorch has a large and active community, with many tutorials, forums, and resources available.<sup>5</sup>

### 2.4.2 TensorFlow

TensorFlow, developed by Google Brain, is another popular open-source deep learning framework. It originally used static computational graphs but now supports eager execution, which allows for creating dynamic computational graphs similar to PyTorch.

TensorFlow is designed to be highly scalable and to have multi-platform availability, from mobile devices to large-scale distributed systems. It includes TensorFlow Extended (TFX), a production-ready platform for deploying machine learning models, and TensorFlow Lite, a lightweight version for mobile and embedded devices.<sup>6</sup>

---

<sup>4</sup><https://www.geeksforgeeks.org/dynamic-vs-static-computational-graphs-pytorch-and-tensorflow/>

<sup>5</sup><https://pytorch.org>

<sup>6</sup><https://www.tensorflow.org>

### ■ 2.4.3 Haskell

Haskell is a purely functional programming language known for its strong type system and expressive syntax. While not as widely used as Python for deep learning, Haskell has frameworks like Grenade for building neural networks.<sup>[4]</sup>

Haskell's functional nature ensures immutability and referential transparency, which allows reasoning about programs easier. Haskell's type system can catch many errors at compile-time, improving reliability. Its lazy evaluation allows for efficient computation and the handling of infinite data structures.<sup>7</sup>

---

<sup>7</sup><https://www.haskell.org>



## Chapter 3

# Deep Relational Learning

Existing relations between elements in a given set can be used as powerful additional information in machine learning. Enhancing effective deep learning techniques with relational data is an innovative approach called Deep Relational Learning.

Unlike traditional deep learning, which often works with unstructured data like images and text, deep relational learning deals with data that has explicit relationships and structures, such as social networks, knowledge graphs, and databases. By understanding and leveraging these relationships, models can make better predictions and provide deeper insights. [16]

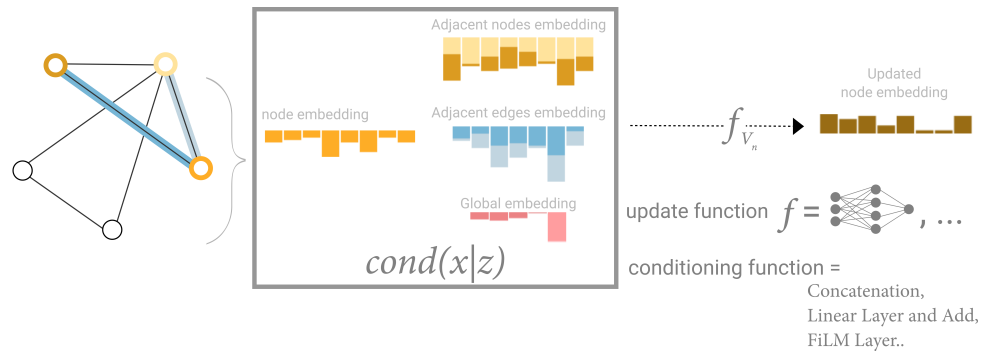
This chapter will cover Graph Neural Networks, Relational Logic, and Lifted Relational Neural Networks, providing an overview of how these techniques are used in deep Relational Learning.

### 3.1 Graph Neural Networks

Graph Neural Network (GNN) is a type of neural network designed to work directly with graph data structures. GNNs process data represented as graphs. That means sets of nodes (or vertices) and edges (connections between nodes). GNNs are powerful because they can capture the dependencies and relationships between different nodes in the graph, making them ideal for tasks where data is naturally structured as a graph. [19]

In a GNN, each node in the graph can be represented by a feature vector, which contains information about the node. The edges hold information about the relationship between nodes and can be represented by feature vectors as well. The goal of a GNN is to learn a representation (embedding) for each node that captures both its features and the features of its neighbors. [19]

GNNs typically work in layers, where each layer updates the node embeddings by aggregating information from neighboring nodes. This process is known as message passing or neighborhood aggregation. After several layers, the final node embeddings possess complex information including the relational data. This network is basically an optimizable transformation of



**Figure 3.1:** Node embedding update [12]

all graph properties. [12]

The resulting set has the same graph structure but additionally contains comprehensive data that can be used for various tasks like node classification, link prediction, and graph classification. [12]

### ■ 3.1.1 Types of GNNs

- Graph Convolutional Networks (GCNs): These networks apply a convolution-like operation to the graph. Each node updates its embedding based on its neighbors' embeddings and its own features. GCNs are popular for semi-supervised learning tasks. [17]
- Graph Attention Networks (GATs): GATs use attention mechanisms to weigh the importance of neighboring nodes differently. This allows the model to focus on more relevant nodes during the aggregation process. [17]
- Graph Recurrent Networks (GRNs or GRNNs): These networks use recurrent neural network architectures to handle dynamic and temporal graphs, where the graph structure or node features change over time. GRNs combine the advantage of RNNs on temporal data and GNNs on graph data. [6, 9]

## ■ 3.2 Relational Logic

Relational Logic is an extension of Propositional Logic that incorporates linguistic features like constants, variables, and quantifiers. Unlike Propositional Logic, simple sentences in Relational Logic have more structure, allowing for expressions about multiple objects without listing them and the existence of objects meeting certain conditions without specifying them. [3]

### ■ 3.2.1 Uses in Machine Learning

Relational Logic is applied in machine learning to incorporate prior knowledge and constraints into models, particularly in domains where the relationships

between entities are crucial, such as recommendation systems, natural language understanding, and the semantic web.

By using logic-based formalism, Relational Logic allows machine learning models to represent and reason about relationships and structured data, capturing complex relational structures and dependencies present in the data. This capability enhances the model's ability to understand and process interconnected information effectively. [13]

### ■ 3.3 Lifted Relational Neural Networks

Lifted Relational Neural Networks (LRNNs) combine relational logic with deep learning, providing a framework for learning weights of latent relational structures. This method bridges the gap between symbolic reasoning and neural computation. LRNNs allow for the learning of complex patterns in relational data by lifting the reasoning process to a higher level of abstraction. LRNNs are designed to handle data with rich relational structures, making them suitable for tasks that traditional neural networks struggle with. [14]

In LRNNs, the neural network components are used to process the features of entities and relationships, while the relational logic components handle the structure and dependencies. This combination allows LRNNs to learn both the features and the relational patterns in the data.

For example, in a social network, an LRNN can learn to predict new friendships based on both the features of individuals (like interests and activities) and the existing friendship patterns. These physical network can be transformed into relationship graphs to be processed by neural network. [15]



# Chapter 4

## PyNeuraLogic

The main subject of this thesis is enhancing the already existing framework called PyNeuraLogic<sup>1</sup>. At the moment, the newly implemented differentiable functional language has not been yet integrated into the official PyNeuraLogic version for possible further development. Instead, a new GitHub project has been created, as a forked version of PyNeuraLogic framework, and can be found at <https://github.com/krausjakub/PyNeuraLogicFork>

The chapter aims to comprehensively explain the structure, workflow, features, and usage of the original untouched framework since the enhancement directly builds upon this concept later in Chapter 5.

### 4.1 Introduction

PyNeuraLogic is a framework, that builds upon the concept of Deep Learning with Relational Logic Representations[13]. It combines the strengths of neural networks and logical reasoning to tackle complex problem-solving tasks. It allows for the seamless integration of logic programming with deep learning, making it possible to build models capable of handling both structured and unstructured data. This unique approach is particularly useful in fields such as natural language processing, knowledge graph completion, and bioinformatics, where there is a need to incorporate domain-specific knowledge along with data-driven learning.

The primary motivation behind the development of PyNeuraLogic was to create a framework that bridges the gap between symbolic AI and machine learning. Traditional deep learning frameworks like TensorFlow<sup>2</sup> and PyTorch<sup>3</sup> are excellent for processing numerical data and learning from large datasets. However, they often fall short when it comes to incorporating logical rules and domain-specific knowledge. PyNeuraLogic addresses this limitation by allowing users to define logical rules that guide the learning

---

<sup>1</sup><https://github.com/LukasZahradnik/PyNeuraLogic>

<sup>2</sup><https://www.tensorflow.org/>

<sup>3</sup><https://pytorch.org/>

process, thereby enhancing the expressiveness and flexibility of the models. This positions PyNeuraLogic uniquely among deep learning frameworks, offering a more holistic approach to solving complex problems that require both reasoning and learning capabilities.

### ■ 4.1.1 Structure of Framework

The framework itself can be in a way divided into two main parts. The PyNeuraLogic front end (Python) and NeuraLogic back end (java). NeuraLogic was created first, but since most of AI development took place in Python, the idea of creating Python front end, which would further enhance the original framework, rose up. In following section, both parts will be discussed separately.

## ■ 4.2 Python Front End

In this section, we will discuss the most important structures of PyNeuraLogic. Those will play crucial role later in Chapter 5.

### ■ 4.2.1 Rule

Rules are fundamental in the PyNeuraLogic framework, serving as the primary method for defining relationships and transformations within the data. In the context of neural-symbolic learning, rules are used to encode logical relationships that govern how different entities in the dataset interact. These rules are typically expressed in a declarative form, specifying what needs to be done without detailing the procedural steps to accomplish it.

In PyNeuraLogic, rules can be used to capture both direct and indirect relationships between entities. For instance, a simple rule might specify that if entity A is related to entity B, and entity B is related to entity C, then entity A should have some inferred relationship with entity C. This kind of rule allows the framework to infer new relationships based on existing ones, leveraging the power of logical reasoning to enhance the learning process.

The integration of rules into the neural network training process allows PyNeuraLogic to combine symbolic reasoning with deep learning. This combination is particularly powerful for tasks that involve structured data, such as social network analysis or molecular property prediction, where the relationships between entities are as important as the entities themselves.

Rules also play a significant role in defining the model architecture in PyNeuraLogic. By specifying how different parts of the model should interact, rules help to guide the learning process and ensure that the learned representations respect the underlying logical structure of the data. This leads

to models that are not only more accurate but also more interpretable, as the rules provide a clear explanation of how the model arrives at its predictions.

### ■ 4.2.2 Dataset

The dataset component in PyNeuraLogic is designed to handle learning samples formatted in a logic-based manner, allowing users to fully leverage the expressive power of logical constructs. The framework supports datasets that include relational data, which can be represented using logical facts and rules. This capability is particularly useful for domains where the data naturally forms complex graphs or networks, such as social networks, biological networks, and knowledge graphs.

A PyNeuraLogic dataset is typically composed of samples that encapsulate the relationships between entities. Each sample can include a set of facts that describe the properties of individual entities and the relationships between them. These facts are often represented as tuples or logical statements, providing a flexible and structured way to represent the data.

For instance, in a social network dataset, a sample might include facts about individuals (nodes) and their friendships (edges). Each individual could be described by a set of attributes, such as age, gender, and interests, while friendships could be represented as binary relationships between nodes. This structured representation allows the dataset to capture both the attributes of entities and the complex web of relationships that connect them.

The PyNeuraLogic framework provides tools for managing and manipulating datasets, including functions for adding, removing, and querying samples. These tools make it easy to preprocess the data, construct training and test sets, and perform other common data management tasks.

One of the key advantages of using a logic-based dataset in PyNeuraLogic is the ability to perform logical inference. By defining rules that govern the relationships between entities, the framework can infer new facts from existing ones, enriching the dataset and improving the performance of the learning algorithms. This capability is particularly useful for tasks that require reasoning about the data, such as predicting missing links in a knowledge graph or identifying hidden patterns in a social network.

### ■ 4.2.3 Template

The template component in PyNeuraLogic is used to define the overall structure and architecture of the neural network model. A template specifies how the different parts of the model should interact, providing a blueprint for constructing the neural network. This allows users to design complex models

that leverage both logical reasoning and neural network learning.

Templates in PyNeuraLogic are designed to be flexible and modular, allowing users to define custom architectures that suit their specific needs. Each template can include a combination of neural network layers and logical rules, providing a powerful way to integrate symbolic reasoning with deep learning. By specifying the structure of the model in a template, users can ensure that the neural network respects the logical relationships in the data and performs the desired computations.

For example, a template might include a series of convolutional layers to process the input features, followed by a set of rules to combine the features and infer new relationships. This approach allows the model to capture both the local patterns in the data and the global structure of the relationships, providing a comprehensive representation of the input.

The framework provides tools for creating and managing templates, making it easy to design and modify the model architecture. Users can define templates using a simple and intuitive syntax, specifying the neural network layers, logical rules, and other components that make up the model. This allows for a clear and structured representation of the model architecture, ensuring that the neural network is correctly configured and optimized for the learning task.

Templates also play a crucial role in the training process, guiding the learning process and ensuring that the model learns the desired patterns and relationships. By defining the structure of the model in a template, users can control how the different parts of the neural network interact and ensure that the learned representations respect the logical relationships in the data.

#### ■ 4.2.4 Query

Queries in PyNeuraLogic are used to specify the learning targets or outputs that the model is expected to predict. A query typically includes a logical statement that defines the target relationship or property that the model should learn to predict. By specifying the queries, users can guide the learning process and ensure that the model focuses on the relevant aspects of the data.

In the context of neural-symbolic learning, queries are essential for defining the learning objectives and evaluating the model's performance. They allow the framework to connect the input data (examples) with the expected outputs, providing a clear and structured way to represent the learning task.

For instance, in a social network analysis task, a query might specify that the model should predict whether two individuals are friends based on their attributes and connections. This query would be represented as a logical



statement that relates the input facts (attributes and connections) to the target relationship (friendship). By defining the queries in this way, PyNeuraLogic can use logical reasoning to infer the target relationships and guide the learning process.

The framework provides tools for defining and managing queries, making it easy to specify the learning targets and evaluate the model's performance. Users can define queries using a simple and intuitive syntax, specifying the logical relationships and properties that the model should learn to predict. This allows for a clear and structured representation of the learning objectives, ensuring that the model focuses on the relevant aspects of the data.

Queries are also used during the evaluation process to measure the model's performance. By comparing the predicted outputs with the true targets specified by the queries, users can assess the accuracy and effectiveness of the model. This provides a clear and objective way to evaluate the model's performance and identify areas for improvement.

### ■ 4.2.5 Example

In the PyNeuraLogic framework, examples are used to represent the input data that the model will learn from. An example typically includes a set of logical facts that describe the properties and relationships of the entities in the dataset. These facts are used to construct the input to the neural network, providing the information needed to perform the learning task.

Examples in PyNeuraLogic are designed to be flexible and expressive, allowing users to represent a wide range of data types and structures. Each example can include facts about individual entities, such as their attributes and properties, as well as relationships between entities. This allows the framework to capture both the local features of entities and the global structure of the data.

For instance, in a knowledge graph, an example might include facts about different entities (nodes) and their relationships (edges). These facts could describe the types of entities, their attributes, and the connections between them. By representing the data in this way, PyNeuraLogic can leverage the power of logical reasoning to infer new relationships and improve the accuracy of the model.

The framework provides tools for creating and managing examples, making it easy to preprocess the data and construct the input for the neural network. Users can define examples using a simple and intuitive syntax, specifying the facts and relationships that make up each example. This makes it easy to represent complex data structures and ensure that the input to the model is

accurate and consistent.

Examples are also used in the training process to provide the input data for the neural network. During training, the model uses the facts in each example to learn the patterns and relationships in the data. This allows the model to make accurate predictions and perform well on the learning task.

## ■ 4.3 Java Back End

The NeuraLogic back end is the core computational engine of the PyNeuraLogic framework. Written in Java, the back end is optimized for handling the intensive processing tasks required for neural-symbolic learning. It is responsible for the execution of logical inferences, the training of neural networks, and the efficient management of large and complex datasets.

NeuraLogic's back end processes include the evaluation and optimization of logical rules, the execution of graph-based computations, and the application of machine learning algorithms to extract patterns from data. The Java implementation ensures that these tasks are performed efficiently, leveraging Java's robust performance capabilities and its well-established libraries for data processing and parallel computation.

In addition to handling the core computational tasks, the NeuraLogic back end also manages the integration of neural network components with logical inference mechanisms. This involves combining symbolic reasoning with sub-symbolic learning, allowing the framework to handle complex relational data and perform sophisticated inference tasks. The back end's design ensures that the computational workload is distributed effectively, providing scalability and robustness for large-scale neural-symbolic learning applications.

By dividing the responsibilities between the front end and the back end, PyNeuraLogic allows users to benefit from the ease of use and flexibility of Python, while relying on the performance and computational power of Java for the heavy lifting. This separation ensures that the framework can handle both the complexity of logical reasoning and the demands of deep learning, making it a versatile tool for advanced AI research and development.

### ■ 4.3.1 Internal Structure

The NeuraLogic backend is organized into several modular packages, each responsible for different aspects of the framework's functionality, facilitating efficient development, testing, and maintenance. The primary packages include Algebra, CLI, Drawing, Learning, Logging, Logic, Neural, Neuralization, Parsing, Pipelines, Settings, Utilities, and Workflow. The Algebra package

handles essential mathematical operations and data structures, forming the foundation for numerical computations. The Logic package provides core logic programming capabilities, including a subsumption engine for relational logic grounding and logical reasoning. The Neural package focuses on neural network computations and deep learning tasks, while the Neuralization package converts logical structures into neural networks, integrating symbolic reasoning with sub-symbolic learning. The Parsing package uses ANTLR to interpret the NeuraLogic language, ensuring correct integration of user-defined logical rules. The Pipelines package offers tools for creating machine learning workflows, and the Workflow package provides components for typical NeuraLogic tasks such as data preprocessing, model training, and evaluation. Additional packages include CLI for command-line interaction, Drawing for visualizations, Learning for supervised machine learning, Logging for utilities, Settings for configuration, and Utilities for generic operations like serialization and benchmarking.

### ■ Combination, Transformation and Aggregation

In the NeuraLogic framework, the concepts of Combination, Transformation, and Aggregation play crucial roles in the processing and manipulation of data within neural-symbolic learning models. These operations are fundamental for defining how data is combined, transformed, and aggregated throughout the learning process, enabling the integration of logical reasoning with neural network computations.

Combination refers to the process of integrating multiple inputs or features into a single cohesive representation. In NeuraLogic, this is often done using logical rules that specify how different pieces of data should be combined. For example, combining features from multiple nodes in a graph to form a single node representation. This operation is essential for creating richer and more informative representations from the input data, allowing the model to capture complex relationships and dependencies.

Transformation involves applying a function to data to change its representation or structure. In NeuraLogic, transformations are used to map input features to new spaces, often through neural network layers or other mathematical functions. For instance, a transformation might involve applying a linear transformation followed by a non-linear activation function to a feature vector. This allows the model to learn and represent complex patterns in the data. Transformations are key to the flexibility and power of neural networks, enabling them to learn intricate mappings from inputs to outputs.

Aggregation is the process of combining multiple data points into a single summary statistic or representation. In the context of NeuraLogic, aggregation might involve summing, averaging, or applying other statistical operations to features from multiple nodes or edges in a graph. This is particularly important in graph neural networks, where the information from neighboring

nodes needs to be aggregated to update the representation of a given node. Aggregation allows the model to distill relevant information from a potentially large and complex set of inputs, making it manageable and useful for further processing.

## Chapter 5

# Functional Language Proposition

### 5.1 Need of New Syntax

Although PyNeuraLogic<sup>1</sup> and its predecessor, NeuraLogic<sup>2</sup>, are highly effective frameworks, there is room for further enhancement. The existing syntax is quite capable and supports essential features for integrating neural networks with logical reasoning. However, improving it to be more user-friendly and intuitive would greatly benefit users. The current syntax, while functional, can sometimes be challenging for users to write and debug, especially for those new to the framework.

To address these challenges, there was a need to develop a new syntax that simplifies and enhances the user experience. This newly proposed syntax aims to be more straightforward, making it easier to write and understand code. It will introduce additional ways to evaluate expressions, offering greater flexibility in model definition. Moreover, it will include better error correction capabilities, helping users to quickly spot and fix simple input mistakes. These improvements are expected to make the development process smoother and aid in creating more reliable models.

Even though the primary goal is to reuse as much of the original source code as possible, the proposed syntax will bring some changes to the internal evaluation processes of the framework, not just its appearance. By optimizing how expressions are evaluated, the new syntax will prepare the framework for future enhancements and scalability. The well-organized structure of this new syntax will ensure that PyNeuraLogic can easily adapt to new features and improvements, making it more versatile for tackling various relational logic learning problems.

Initially, there were multiple approaches considered for this new syntax. In the following sections, we will look into these different approaches and discuss their pros and cons, providing a comprehensive understanding of why the

---

<sup>1</sup><https://github.com/LukasZahradnik/PyNeuraLogic>

<sup>2</sup><https://github.com/GustikS/NeuraLogic>

chosen path is expected to be the most beneficial.

## ■ 5.2 Approach Analysis

When developing a new syntax for a library, one of the core principles is ensuring simple and efficient usage. This is particularly crucial for open-source libraries, as they often begin with limited functionality and expand gradually based on their popularity and user feedback. However, if the syntax is made too simple, it might later pose challenges when adding new functionalities, which could be difficult to address. Creating a syntax that balances simplicity and future expandability is vital. For open-source projects, the initial design choices can significantly impact how easily the library can grow and adapt to new demands. Too much simplicity might limit the library's capabilities in the long run, making it harder to implement advanced features without significant restructuring. On the other hand, a more complex initial design might deter new users due to its steep learning curve.

### ■ 5.2.1 Editing Python

One straightforward approach considered was to introduce completely new operators (and possibly operands), that have currently no usage in Python. For instance, let's say, we would like to express, that certain state  $Y$  can be computed from any state  $X$ , that already has an existing relation with the state  $Y$ .

$$(State(X)R.relation(X, Y) \rightarrow State(Y))$$

Then, such information could be encoded very intuitively in Python with respect to the rest of the framework as follows.

$$((R.state(V.X), R.relation(V.X, V.Y)) \rightarrow R.state(V.Y))$$

Apart from the right arrow, there are several other symbols that can be easily typed on an English keyboard but currently have no specific use, making them potential candidates for new operators.

The most significant advantage of this approach is its clarity within Python code. Such operations would be unmistakable, and once user understands their purpose, he would likely remember them. Since functional programming syntax tends to be straightforward, only a few new operators would be needed, preventing users from feeling overwhelmed.

While this option initially appears promising, implementation challenges quickly become evident. To interpret code, Python first creates a parse tree

using its parser PEG. The instructions for PEG are contained in a grammar specification file, where new operators would have to be declared. Following this, Python converts the parse tree into an Abstract Syntax Tree (AST) for easier manipulation. This step would require modifying the AST structure and defining new processes, which would involve using C, the language of Python's core.[11]

Then, the real troubles would begin with optimization. Python compiles the AST into Bytecode for efficient execution. Creating a custom operator necessitates custom Bytecode, which is a complex task requiring a deep understanding of Python's compilation process. Additionally, modifications to the Symbol Table would be necessary for the compilation process to recognize and handle new operators properly.[11]

For those reasons, such an approach would not be suitable for developing new Python syntax. Even if all the steps described above were implemented successfully, the portability of the code would be ruined. Thus, we must consider alternative approaches.

### ■ 5.2.2 File Encoding

Another promising approach is to write the syntax into a non-executable file and then create a custom parser to interpret and implement the desired functionality. This method opens up many possibilities, as it removes the limitations imposed by Python's native syntax and allows us to define our own. Additionally, all the code could be written in Python, requiring only basic knowledge of the language internals, or we could use an existing parser to streamline the process.

This approach, while advantageous, also presents its own set of negatives. One major downside is that it can make using the library more cumbersome. Modern Integrated Development Environments (IDEs) offer numerous advanced features, such as auto-complete, error detection, and hover-hints, which enhance the coding experience and efficiency. If we rely on custom syntax in a non-executable file, we lose the immediate benefits of these features. To address this, we would need to develop a new editor capable of supporting our custom syntax, which would be a substantial undertaking.

Finally, we would have to ensure, that the custom syntax is integrated seamlessly with the rest of the library. This involves thorough testing and validation to ensure that the new syntax performs as expected and does not introduce unexpected bugs or inefficiencies. While this approach requires careful planning and execution, it offers a flexible and powerful way to extend the capabilities of PyNeuraLogic without being constrained by Python's existing syntax.

### ■ 5.2.3 Enhancing Python Syntax

The final approach considered for developing the new syntax for PyNeuraLogic draws inspiration from other well-known deep learning libraries, partly described in Chapter 5. This approach involves enhancing the existing Python syntax by editing current operators and adding new functionality through magic functions, also known as dunder methods. Magic functions are invoked automatically by Python when specific operators are used, and they can be customized for our own classes.

This method offers several advantages. Firstly, it maintains the user-friendly nature of Python. Users are already familiar with this approach and expect such functionality in advanced libraries. By leveraging magic functions, we ensure that users can utilize the full range of IDE features, such as auto-complete, error detection, and code hints, which enhance the development experience. Moreover, this approach does not require any modifications to Python's internal functionalities, ensuring that the code remains easily portable and compatible across different devices and environments.

A significant benefit of using magic functions is the preservation of the correct order of operations. Python naturally handles the precedence of common operators such as addition and multiplication, so developers do not need to manually enforce these rules. This ensures that operations are performed in the correct sequence without additional intervention.

However, this approach also has potential drawbacks. One notable disadvantage is that overloading operators can sometimes confuse users. The same operators can behave fairly differently depending on the types on which they are invoked. For example, an addition operator might perform numerical addition on integers but could concatenate strings. This variability can lead to misunderstandings and errors if users are not fully aware of how operators are overloaded in specific contexts.

For all the reasons mentioned above, this last approach has been chosen as most suitable for the needs of this thesis, and later in this chapter, we are going to delve deeper into how this approach has been implemented while exploring specific examples.

## ■ 5.3 Storing and managing input data

One of the objectives of this thesis was to allow for more complex and general data encoding. Originally, those data would be passed into list-like structures and attached to specific data class (typically Template). Those would be served into appropriate NeuraLogic<sup>3</sup> core functions, where they would serve

---

<sup>3</sup><https://github.com/GustikS/NeuraLogic>



as a template for creating the required neural network structure. A similar approach has been preserved, but this time, entities would be first of all folded into a tree structure, and then they would undertake further processing. The main reason for that was the necessity of encapsulation of more complex information.

### ■ 5.3.1 Functional Tree

A tree structure is a widely-used data organization method that resembles a hierarchical tree, with nodes representing elements connected by edges. The core properties of a tree include a single root node, which acts as the starting point, and child nodes that branch out from parent nodes, forming a connected, acyclic graph. Since the tree structure was needed for elegant encapsulation of data functional properties, it had been named Functional Tree and in this thesis, we will refer to it as such. Its specific implementation can be found in the framework in the file `tree.py`<sup>4</sup>.

Functional Tree has been implemented as a class, where each instance is a node. Each node includes information, about the operation that needs to be performed on its child nodes, as well as a reference to the left and right ones. A new tree is created either by adding Rules, other trees, or by calling a function from the Function Container.

### ■ 5.3.2 Functional Container

Functional Container is another class, that can be found in the same file as Functional Tree, and as its name suggests, it serves as a container for all different mathematical operations applicable in the framework. Actually, there are over thirty of them, even though this number contains some of those more than once since they can be applicable to different stages of evaluation.

The container (in examples frequently referred to simply as  $F$ ) also serves as an indicator for the user, that functions from within the Java back end are being used. That is important since one of the next steps in developing the framework will be the addition of the possibility to use other evaluating modules or packages, such as Torch<sup>5</sup>.

## ■ 5.4 Data Processing and Analysis

The role of a Functional Tree is not only storing information but also analyzing and handling it. Before anything else, a tree instance is inspected to assure the correctly specified problem. If the user doesn't follow syntax rules, an

---

<sup>4</sup><https://github.com/krausjakub/PyNeuralLogicFork/blob/master/neurallogic/core/constructs/function/tree.py>

<sup>5</sup><https://github.com/pytorch/pytorch>

appropriate exception is raised with an additional explanation. During this operation, some of the functions are automatically edited to match their role in processed problems. And by that, it is meant the association of function with Combination, Transformation, or Aggregation. The concept of those three and their role in the framework was previously explained in Chapter 4.3.1. This would not be necessary if all functions had only one association possible, but that is unfortunately not true. Just softmax by itself can be present in any of those forms, or perhaps all of them at once. The right association is determined by the recursive analysis of a given tree from root to leaves.

### ■ 5.4.1 Functional Call Simulator

When developing PyNeuraLogic, we aimed to provide the flexibility of calling functions using both square brackets and parentheses, such as `F.avg[]` and `F.avg()`, where `F` is an instance of Function Container. To achieve this, it was necessary to create the Function Call Simulator class. Reasoning of this claim would not be very beneficial for this thesis and thus is not described here, but just mentioning the fact itself could be handy for anyone who tries to understand the framework better. Finding out this information from the code itself is quite challenging because the way the class is implemented represents a certain trick in Python, that the average user rarely encounters.

## ■ 5.5 Examples

There is a rich variety of beautiful examples described in detail in PyNeuraLogic<sup>6</sup> and NeuraLogic<sup>7</sup> GitHub as well as in its official documentation<sup>8</sup>. Together, they cover a set of diverse complex problems and show where the framework's greatest strength lies. Due to the main purpose of the thesis, the creation of a new language, in this chapter we will discuss two new smaller examples because on those the syntax is easier to understand. Only snippets of code will be displayed, but in the end of each example, there will be a reference to the executable jupyter notebook with the rest of the code. All figures in this chapter were created using PyNeuraLogic drawing functions.

### ■ 5.5.1 Example 1: Four Nodes

Let's say we have four nodes: A, B, C and D. We know the value of the first three of those, and we want to know the value of the fourth. Example and query must be added to the dataset.

```
dataset.add_example([R.a[3], R.b[5], R.c[2]])
dataset.add_queries([R.d[1]])
```

**Listing 5.1:** Example 1: Assignment

<sup>6</sup><https://github.com/LukasZahradnik/PyNeuraLogic/tree/master/examples>

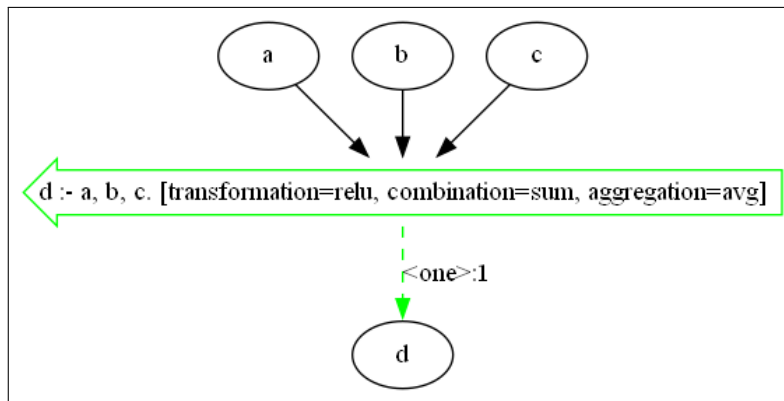
<sup>7</sup><https://github.com/GustikS/NeuraLogic/tree/master/Resources/datasets>

<sup>8</sup><https://pyneuralogic.readthedocs.io/en/latest/index.html>

Following the logic paradigm, we declare, that value of D can be computed using first three nodes. We can now decide the way, we want them to be combined, if we want to transform the combined output somehow and how are going to be all examples that match the demanded definition aggregated. There is only one such occurrence in this example, so defining aggregation is useless. The rule that contains all the information is handed to the template, drawn in Figure 5.1.

```
template += (R.d <= (x:=R.a, y:= R.b, z:=R.c)) >> F.avg[F
.relu(x + y + z)]
```

**Listing 5.2:** Example 1: Definition



**Figure 5.1:** Example 1: Template

That being done, the problem is correctly defined and the dataset can be built. The neural logic evaluator will then call on it its train method, so the neural network can hopefully learn appropriate weights. The structure of neurons, their values and operations can be displayed using the dataset's ability to draw itself. In the following Figure 5.2 we can indeed see three Fact neurons being combined into one Rule neuron using sum. On Rule, neuron is applied ReLu function, which does not affect its value, and finally non-linearity is applied to the final Atom neuron in the presence of hyperbolic tangent. The final value of Neuron is as expected 1. v

### ■ 5.5.2 Example 2: Graph

In the first example, there was no space to use the concept of Aggregation. In the problem declaration, we specified exactly which elements should be used to compute output. Example 2 will be declared truly in logic programming paradigm, and relations between entities will be described by general variables. The encoded structure is a directed graph with three nodes: A, B and C. Some of the nodes are connected via edges.

```
train_dataset.add_examples(
[
```

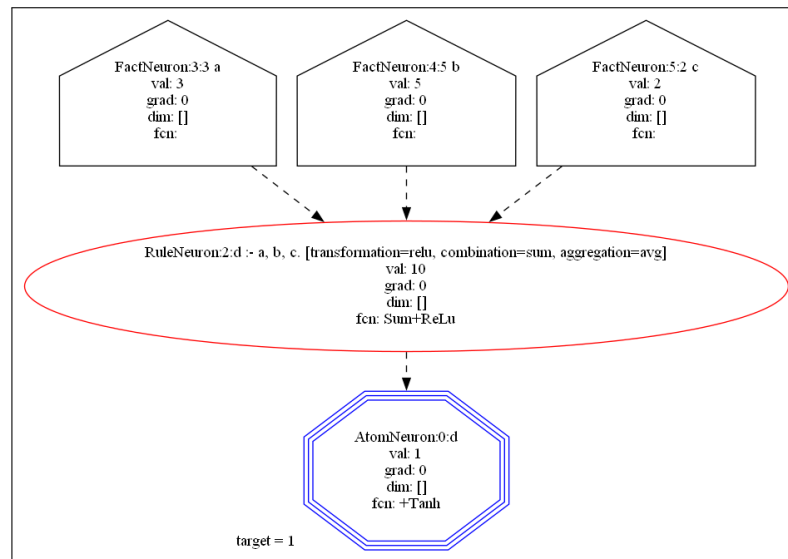


Figure 5.2: Example 1: Dataset

```
[
  R.node(C.A)[1], R.node(C.B)[2], R.node(C.C)
  [3],
  R.edge(C.A, C.B)[5], R.edge(C.B, C.A)[5], R.
  edge(C.A, C.C)[5], R.edge(C.C, C.A)[5]
]
)

train_dataset.add_queries([
  R.predict(C.A)["w3": 1]
])
```

Listing 5.3: Example 2: Assignment

What also changed is the query. The value that we are trying to compute is not node A, but some abstract representation prediction, which takes node A as an argument (in a way). Let's add evaluation rules to the template.

```
template += (R.predict(V.X) <= F.sum[ F.identity( F.avg(
  [R.node(V.Y)["w1": 1], R.edge(V.Y, V.X)["w2": 1] ] ) )
  ])
```

Listing 5.4: Example 2: Definition

The Listing 5.4 is really just a logical rule, that can be in this example applied to two neurons. Those are supposed to be aggregated by a sum. And their individual values are computed by combining the right edges with the right nodes using the function average. The relations and values got suddenly much more complicated, but luckily, the framework drawing functions are again able to capture all relations in a very clear way.

Now let's try to calculate the value of representation  $predict(A)$ .

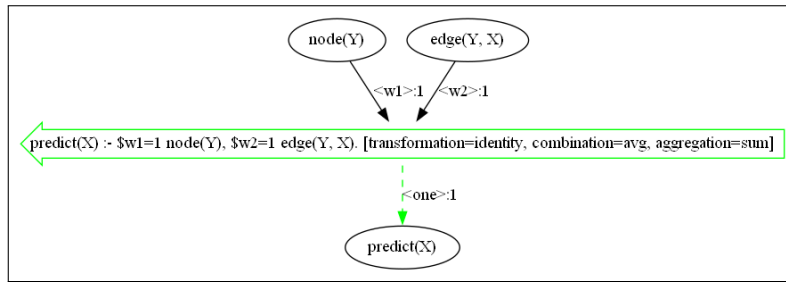


Figure 5.3: Example 2: Template

$$\begin{aligned} \text{predict}(A) &= \text{avg}[(\text{node}(B) \cdot w_1), (\text{edge}(A, B) \cdot w_2)] \\ &\quad + \text{avg}[(\text{node}(C) \cdot w_1), (\text{edge}(A, C) \cdot w_2)] \\ \text{predict}(A) &= \text{avg}[2, 5] + \text{avg}[3, 5] \\ \text{predict}(A) &= 3.5 + 4 = 7.5 \end{aligned}$$

As we can see in Figure 5.4, the value of Aggregation neuron truly matches the predicted output.

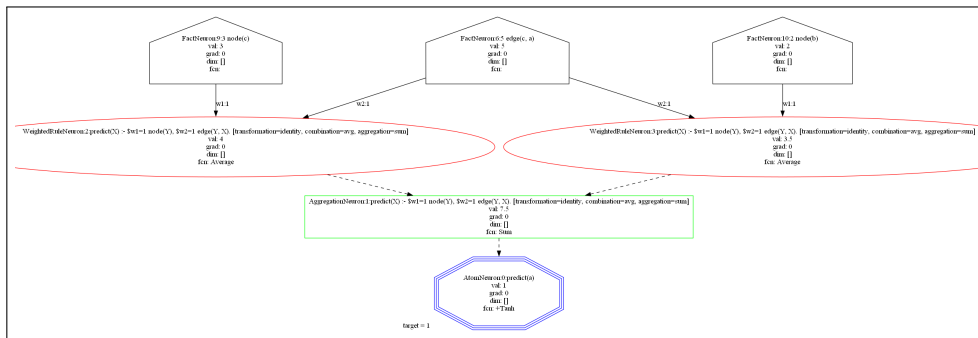



Figure 5.4: Example 2: Dataset

Full example can be found at <https://github.com/krausjakub/PyNeuraLogicFork/blob/master/examples/MyExamples/VariableExample.ipynb>.





## Chapter 6

### Conclusions

In this work, we proposed new language for a machine learning library, that further enhances it's Lifted Relational Neural Network's properties. Several new approaches of declaring computational problems were introduced, while the old ones were kept preserved. Part of the unnecessary process of function specification was left on automatic background analysis, yet again, the old way stayed available. Instead of following strict declaration rules, users can now use functions more dynamically, since the support for interaction between different data types was hugely improved.

Altogether, the code is now clearer, more intuitive, super versatile, dynamic, syntax forgiving and transparent. It's computational efficiency stayed unharmed, and it's tree structure is much more open to any future development, including possible use of different, better optimized functions, which could lead to computation speed up.

Smaller open source libraries usually require constant development, ideally from multiple people, so they can keep up with their commercial rivals. For that reason, it's ability to quickly pick up stable audience is crucial. We kept this idea in mind during the language development, and now we believe, that any new user with basic deep learning knowledge, could figure out the syntax at first glance.

Development of the framework turned out to be beneficial not only for potential new audience, but especially for myself, since it's cross platform nature, deep roots in difficult scientific field, volume and versatility, led me into number of great challenges, which only had been overcome with great effort, using everything I have been taught in previous years. And for that opportunity I am most thankful.







## Bibliography

- [1] Machine Learning For Artists. *How neural networks are trained* — *ml4a.github.io*. [https://ml4a.github.io/ml4a/how\\_neural\\_networks\\_are\\_trained/](https://ml4a.github.io/ml4a/how_neural_networks_are_trained/). [Accessed 24-05-2024].
- [2] Mathieu Blondel and Vincent Roulet. *The Elements of Differentiable Programming*. 2024. arXiv: 2403.14606 [cs.LG].
- [3] Michael Genesereth. *Introduction to Logic*. 3rd ed. Synthesis Lectures on Computer Science. San Rafael, CA: Morgan and Claypool Life Sciences, Nov. 2016.
- [4] *Haskell, Grenade, and Deep Learning* — *Monday Morning Haskell* — *mmhaskell.com*. <https://mmhaskell.com/machine-learning/deep-learning>. [Accessed 24-05-2024].
- [5] Adrián Hernández and José Amigó. “Differentiable programming and its applications to dynamical systems”. In: (Dec. 2019).
- [6] Zhiyuan Liu and Jie Zhou. “Graph Recurrent Networks”. In: *Introduction to Graph Neural Networks*. Cham: Springer International Publishing, 2020, pp. 33–37. ISBN: 978-3-031-01587-8. DOI: 10.1007/978-3-031-01587-8\_6. URL: [https://doi.org/10.1007/978-3-031-01587-8\\_6](https://doi.org/10.1007/978-3-031-01587-8_6).
- [7] Franco Manessi, Alessandro Rozza, and Mario Manzo. “Dynamic Graph Convolutional Networks”. In: *CoRR* abs/1704.06199 (2017). arXiv: 1704.06199. URL: <http://arxiv.org/abs/1704.06199>.
- [8] Franco Manessi, Alessandro Rozza, and Mario Manzo. “Dynamic graph convolutional networks”. In: *Pattern Recognition* 97 (2020), p. 107000. ISSN: 0031-3203. DOI: <https://doi.org/10.1016/j.patcog.2019.107000>. URL: <https://www.sciencedirect.com/science/article/pii/S0031320319303036>.
- [9] Luana Ruiz, Fernando Gama, and Alejandro Ribeiro. “Gated Graph Recurrent Neural Networks”. In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 6303–6318. ISSN: 1941-0476. DOI: 10.1109/tsp.2020.3033962. URL: <http://dx.doi.org/10.1109/TSP.2020.3033962>.
- [10] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

- [11] Pablo Galindo Salgado. *Python Developer's Guide*. <https://devguide.python.org/internals/parser/>. Accessed: 2024-05-23. 2023.
- [12] Benjamin Sanchez-Lengeling et al. "A Gentle Introduction to Graph Neural Networks". In: *Distill* (2021). <https://distill.pub/2021/gnn-intro>. DOI: 10.23915/distill.00033.
- [13] Gustav Šír. "Deep Learning with Relational Logic Representations". In: (2022). arXiv: 1706.03762 [cs.CL].
- [14] Gustav Sourek et al. *Lifted Relational Neural Networks*. 2015. arXiv: 1508.05128 [cs.AI].
- [15] Gustav Šourek, Filip Železný, and Ondřej Kuželka. "Beyond graph neural networks with lifted relational neural networks". In: *Machine Learning* 110.7 (June 2021), pp. 1695–1738. ISSN: 1573-0565. DOI: 10.1007/s10994-021-06017-3. URL: <http://dx.doi.org/10.1007/s10994-021-06017-3>.
- [16] Werner Uwents et al. "Neural networks for relational learning. an experimental comparison". In: *Machine Learning* 82.3 (2011), pp. 315–349. ISSN: 0885-6125. DOI: 10.1007/s10994-010-5196-5. URL: <http://link.springer.com/10.1007/s10994-010-5196-5>.
- [17] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations* (2018). URL: <https://openreview.net/forum?id=rJXMpikCZ>.
- [18] Fei Wang et al. "Demystifying differentiable programming: shift/reset the penultimate backpropagator". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341700. URL: <https://doi.org/10.1145/3341700>.
- [19] Lingfei Wu et al. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Singapore: Springer Singapore, 2022, p. 725.

# Appendices





## 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.