PyNeuraLogic Documentation

Release latest

Sep 23, 2021

CONTENTS

1 Installation **1**
- 1.1 Using PIP . . . 1
- 1.2 Using Conda . . . 1

2 Quick Start **3**
- 2.1 Graph Representation . . . 3
- 2.2 Model Definition . . . 5
- 2.3 Evaluating Model . . . 5

3 PyNeuraLogic Language **7**
- 3.1 The anatomy of a rule . . . 7

4 Heterogeneous Graphs **9**

5 Hypergraph Neural Networks **11**
- 5.1 Representation of hyperedges . . . 11
- 5.2 Propagation on hyperedges . . . 11

6 neuralogic.core **13**
- 6.1 Subpackages . . . 13
- 6.2 Submodules . . . 13
- 6.3 neuralogic.core.builder module . . . 13
- 6.4 neuralogic.core.settings module . . . 13
- 6.5 neuralogic.core.sources module . . . 13
- 6.6 neuralogic.core.template module . . . 13
- 6.7 Module contents . . . 13

7 neuralogic.nn **15**
- 7.1 Subpackages . . . 15
- 7.2 Submodules . . . 15
- 7.3 neuralogic.nn.base module . . . 15
- 7.4 neuralogic.nn.dynet module . . . 15
- 7.5 neuralogic.nn.java module . . . 15
- 7.6 Module contents . . . 15

8 neuralogic.utils **17**
- 8.1 Subpackages . . . 17
- 8.2 Module contents . . . 18

9 Supported backends **19**

10 Prerequisites 21

11 Installation 23

12 Examples 25

Bibliography 27

CHAPTER ONE

INSTALLATION

The PyNeuraLogic library requires `Python >= 3.7` and `Java 1.8` to be installed. Additionally, if you plan to use one of the supported backends (e.g., DyNet), you have to install it manually.

1.1 Using PIP

PyNeuraLogic can be easily installed from PyPI repository via `pip install` command.

```
pip install neuralogic
```

1.2 Using Conda

TODO

CHAPTER

TWO

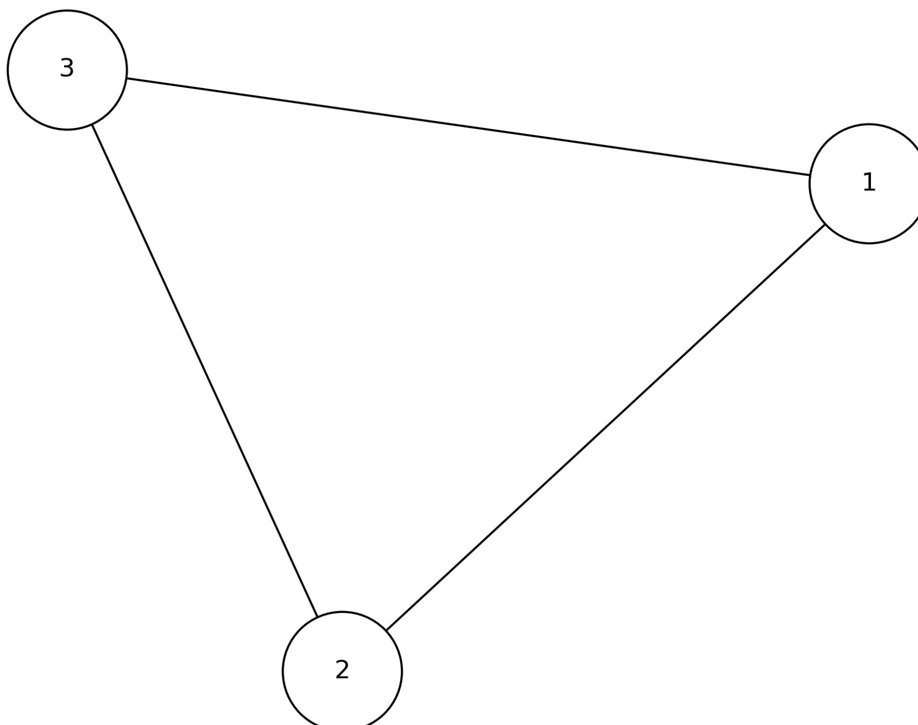
QUICK START

The PyNeuraLogic library serves for learning on structured data. For the purpose of the introduction to the library and its syntax, we will further discuss use cases on graph structures.

2.1 Graph Representation

Graphs can describe entities (vertices) and their relations (edges) which can be useful for various tasks. Graphs are used as inputs for models and are contained in the `Dataset` class.

The `Dataset` class containing information about graphs can be used in different ways depending on the data format. The next section will showcase how to represent the following graph (triangle) in two formats - tensor and logic.



2.1.1 Tensor Representation

The tensor format is a familiar format used in many other GNN focused frameworks and libraries. The input graph is represented in a graph connectivity format, i.e., tensor of shape `[2, num_of_edges]`.

```
from neuralogic.utils.data.dataset import Dataset
from neuralogic.utils.data.dataset import Data


data = Data(
    edge_index=[[1, 2], [2, 1], [1, 3], [3, 1], [2, 3], [3, 2]]
)

dataset = Dataset(data=[data])
```

In this example, we are encoding the simple graph (triangle) in the tensor format. The structure of the graph is encoded in `edge_index` property of the Data class instance. Each `Data` class instance holds information about exactly one graph. The `Dataset` instance then holds a list of data instances and serves as the input.

Note: We omitted a few `Data` class attributes, such as `x` for the nodes' features encoding, `edge_attr` for the edges' features encoding, and `y` and `y_mask` for the target labels encoding.

The tensor representation offers less verbose graph representation but it is more limited in its expressiveness than the logic format introduced in the next section.

2.1.2 Logic Representation

The logic format utilizes constructs based on relational logic to encode input data - graphs. The input data are represented in the form of ground atoms (facts), which can be expressed as `Atom.predicate_name(terms)[value]`.

```
from neuralogic.utils.data.dataset import Dataset
from neuralogic.core import Atom


dataset = Dataset()

dataset.add_example([
    Atom.edge(1, 2), Atom.edge(2, 1), Atom.edge(1, 3),
    Atom.edge(3, 1), Atom.edge(2, 3), Atom.edge(3, 2)
])
```

In this example, we represent the same simple graph (triangle) but in the logic format.

Note: We used the edge as the predicate name (`Atom.edge`) to represent the graph edges. This naming is arbitrary - edges and any other input data can have any predicate name. In this documentation, we will stick to *edge* predicate name for representing edges and *feature* predicate name for representing features.

Note: In the example, we encode the graph structure using an *example* (`add_example`), which does not handle target labels - those are handled by *queries* (`add_query`).

2.2 Model Definition

The model architecture is encoded in the instance of the `Template` class via rules or a list of predefined modules.

```
from neuralogic.core import Template
from neuralogic.utils.templates import GCNConv, TemplateList


template = Template(module_list=TemplateList([
    GCNConv(in_channels=5, out_channels=5),
    GCNConv(in_channels=5, out_channels=1),
]))
```

2.3 Evaluating Model

The PyNeuraLogic library allows users to evaluate and train models on different backends. Those backends (except for the Java backend) have to be installed separately. To get a model that can be evaluated/trained, you have to build its template first.

```
from neuralogic.core import Backend


model = template.build(Backend.JAVA)
```

The input dataset that we are trying to evaluate/train has to be also built. When we have the built dataset and model, performing the forward and backward propagation is straightforward.

```
built_dataset = template.build_dataset(dataset, Backend.JAVA)

model.train()  # or model.test() to change the mode
loss = model(built_dataset)

loss.backward()
```

2.3.1 Evaluators

For faster prototyping, we have prepared evaluators, which encapsulate helpers such as training loop and evaluation. Evaluators can be customized via various settings encapsulated in the `Settings` class.

```
from neuralogic.nn import get_evaluator
from neuralogic.core import Settings, Optimizer


settings = Settings(learning_rate=0.01, optimizer=Optimizer.SGD, epochs=100)
evaluator = get_evaluator(Backend.JAVA, template, settings)

evaluator.train(dataset, generator=False)
```

Note: In the example for the evaluator, we pass the `Dataset` instance (not built dataset) to the `train` method. The evaluator handles the building, but it can be more efficient to pass in an already built dataset (evaluator does not store

built dataset instances).

CHAPTER

THREE

PYNEURALOGIC LANGUAGE

Additionally to predefined modules, PyNeuraLogic allows users to encode machine learning problems via parameterized, rule-based constructs. Said constructs are based on a custom declarative language [Neuralogic] that follows a logic programming paradigm.

3.1 The anatomy of a rule

In PyNeuraLogic, rules are primitives used for building models and datasets.

```
from neuralogic.core import Atom, Var


Atom.h(Var.X)[W_0] <= (Atom.feature(Var.Y)[W_1], Atom.edge(Var.X, Var.Y))
```

The rule consists of a head (`Atom.h`) and a body (`Atom.feature`, `Atom.edge`). Our example can be then read as:

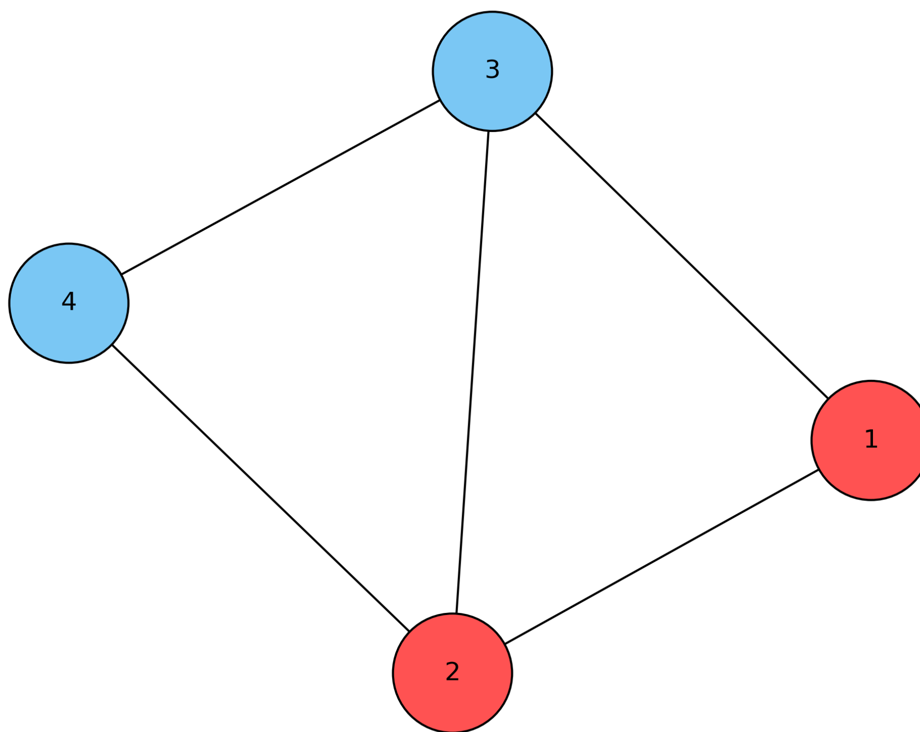
> "Atom h is implied by atom feature and atom edge"

CHAPTER

FOUR

HETEROGENEOUS GRAPHS

Most GNN models do not consider graphs being heterogeneous. Via PyNeuraLogic, we can easily encode heterogeneous graphs with an arbitrary number of node and edge classes.




```
Atom.type(1, Term.RED),
Atom.type(2, Term.RED),
Atom.type(3, Term.BLUE),
Atom.type(4, Term.BLUE),
```

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

```
Atom.type_feature(Term.RED)[[1, 2, 3]]
```

```
Atom.h(Var.X) <= (
    Atom.feature(Var.Y),
    Atom.type_feature(Var.Y),
    Atom.type(Var.Y, Var.Type),
    Atom.edge(Var.X, Var.Y),
)
```

CHAPTER

FIVE

HYPERGRAPH NEURAL NETWORKS

A hypergraph is a generalization of a simple graph $G = (V, E)$, where V is a set of vertices and E is a set of edges (hyperedges) connecting an arbitrary number of vertices.

5.1 Representation of hyperedges

When we encode input data (graph) in the form of logic data format (i.e., ground atoms), we can represent regular edges, for example, as `Atom.edge(1, 2)`.

This form of representation can be simply extended to express hyperedges by adding terms for each connected vertex by the hyperedge. For example, graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{3, 4, 5\}, \{1, 2, 3, 4\}\}$ can be represented as:

```
Atom.edge(1, 2),
Atom.edge(3, 4, 5),
Atom.edge(1, 2, 3, 4),
```

5.2 Propagation on hyperedges

The propagation through standard edges can be similarly extended to support propagation through hyperedges.

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

The propagation through standard edges above, where `Atom.feature` might represent vertex features, and `Atom.edge` represents an edge, might be extended to support hyperedges (for hyperedge connecting three vertices) as follows:

```
Atom.h(Var.X) <= (
    Atom.feature(Var.Y),
    Atom.feature(Var.Z),
    Atom.edge(Var.X, Var.Y, Var.Z),
)
```

CHAPTER

SIX

NEURALOGIC.CORE

6.1 Subpackages

6.1.1 neuralogic.core.constructs

Submodules

neuralogic.core.constructs.atom module

neuralogic.core.constructs.factories module

neuralogic.core.constructs.java_objects module

neuralogic.core.constructs.metadata module

neuralogic.core.constructs.predicate module

neuralogic.core.constructs.rule module

Module contents

6.2 Submodules

6.3 neuralogic.core.builder module

6.4 neuralogic.core.settings module

6.5 neuralogic.core.sources module

6.6 neuralogic.core.template module

6.7 Module contents

CHAPTER

SEVEN

NEURALOGIC.NN

7.1 Subpackages

7.1.1 neuralogic.nn.evaluators

Submodules

neuralogic.nn.evaluators.dynet module

neuralogic.nn.evaluators.java module

neuralogic.nn.evaluators.torch module

Module contents

7.1.2 neuralogic.nn.native

Submodules

neuralogic.nn.native.torch module

Module contents

7.2 Submodules

7.3 neuralogic.nn.base module

7.4 neuralogic.nn.dynet module

7.5 neuralogic.nn.java module

7.6 Module contents

CHAPTER

EIGHT

NEURALOGIC.UTILS

8.1 Subpackages

8.1.1 neuralogic.utils.data

Submodules

neuralogic.utils.data.dataset module

Module contents

8.1.2 neuralogic.utils.templates

Subpackages

neuralogic.utils.templates.modules

Submodules

neuralogic.utils.templates.modules.embedding module

neuralogic.utils.templates.modules.gcn module

neuralogic.utils.templates.modules.gin module

neuralogic.utils.templates.modules.gsage module

neuralogic.utils.templates.modules.pooling module

Module contents

Submodules

neuralogic.utils.templates.translator module

Module contents

8.1.3 neuralogic.utils.visualize

Module contents

8.2 Module contents

PyNeuraLogic is a framework built on top of NeuraLogic which combines relational and deep learning.

PyNeuraLogic 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.

CHAPTER

NINE

SUPPORTED BACKENDS

PyNeuraLogic currently supports following backends (to some extent), which have to be installed separately:

- DyNet
- Java
- PyTorch Geometric

CHAPTER

TEN

PREREQUISITES

To use PyNeuraLogic, you need to install the following prerequisites:

```
Python >= 3.7
Java 1.8
```

CHAPTER ELEVEN

INSTALLATION

To install PyNeuraLogic's latest release from the PyPI repository, use the following command:

```
pip install neuralogic
```

CHAPTER TWELVE

EXAMPLES

- XOR Example
- Pattern Matching
- Distinguishing k-regular graphs
- Distinguishing non-regular graphs

BIBLIOGRAPHY

[Neuralogic] https://github.com/GustikS/NeuraLogic