PyNeuraLogic

Lukáš Zahradník

Jul 15, 2022

CONTENTS

CHAPTER

ONE

INSTALLATION

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

Tip:

```
pip install neuralogic
```

1.1 Requirements

The PyNeuraLogic library requires `Python >= 3.7` and `Java >= 1.8` to be installed.

Additionally, if you plan to use one of the other supported backends, you have to install it manually.

In case you want to use visualization provided in the library, it is required to have Graphviz installed.

CHAPTER

TWO

QUICK START

The PyNeuraLogic library serves for learning on structured data. This quick start guide will showcase one of its uses on graph structures. Nevertheless, the library is directly applicable to more complex structures, such as relational databases.

Tip: Check out one of the runnable *Examples* in Google Colab!

2.1 Graph Representation

Graphs are structures describing entities (vertices) and relations (edges) between them. In this guide, we will look into how to encode graphs as inputs in different formats and how to learn on graphs.

2.1.1 Tensor Representation

In PyNeuraLogic, you can encode input graphs in various formats depending on your preferences. One such format is a tensor format that you might already know from 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]`. The features are encoded via tensor of shape `[num_of_nodes, num_of_features]`.

Let's consider a simple undirected graph shown above. We can simply encode the structure of the graph (edges) via the `edge_index` property and nodes' features via the `x` property of class *Data*, which encapsulates graphs' data. We can also assign a label to each node via the `y` property. The *TensorDataset* instance then holds a list of such graphs tensor representations (*Data* instances) and can be fed into models

```
from neuralogic.dataset import Data, TensorDataset


data = Data(
    edge_index=[
        [0, 1, 1, 2, 2, 0],
        [1, 0, 2, 1, 0, 2],
    ],
```

(continues on next page)

(continued from previous page)

```
    x=[[0], [1], [-1]],
    y=[[1], [0], [1]],
    y_mask=[0, 1, 2],
)

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

2.1.2 Logic Representation

The tensor representation works well for elementary use cases, but it can be quite limiting for more complex inputs. Not everything can be easily aligned and fitted into a few tensors, and working with tensors can get quickly cumbersome. That's where the logic representation comes in with its high expressiveness.

The logic format is based on relational logic constructs to encode the input data, such as graphs. Those constructs are mainly so-called facts, which are represented in PyNeuraLogic as `Relation.predicate_name(...terms)[value]`.

The *Dataset* class contains a set of fact lists representing input graphs. The encoding of the previously shown simple graph can look like the following:

```
from neuralogic.core import Relation
from neuralogic.dataset import Dataset


dataset = Dataset()

dataset.add_example([
    Relation.edge(0, 1), Relation.edge(1, 2), Relation.edge(2, 0),
    Relation.edge(1, 0), Relation.edge(2, 1), Relation.edge(0, 2),

    Relation.node_feature(0)[0],
    Relation.node_feature(1)[1],
    Relation.node_feature(2)[-1],
])
```

As you can see, this encoding can be pretty lengthy, but at the same time, it gives us multiple benefits over the tensor representation. For example, nothing stops you from adding edge features, such as `Relation.edge(0, 1)[1.0]`, or even introducing hypergraphs, such as `Relation.edge(0, 1, 2)` (read more about *Hypergraph Neural Networks*).

Note: We used the *edge* as the predicate name (`Relation.edge`) to represent the graph edges and the *feature* (`Relation.node_feature`) to represent nodes' features. 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.

To assign labels, we use queries. Labels can be assigned to basically anything - nodes, graphs, sub-graphs, etc. In this example, we will label nodes, just like in the case of tensor format representation.

```
dataset.add_queries([
    Relation.predict(0)[1],
    Relation.predict(1)[0],
    Relation.predict(2)[1],
])
```

Note: The name `Relation.predict` refers to the output layer of our model, which we will define in the next section.

2.2 Model Definition

Models in PyNeuraLogic are not just particular computational graphs, as common in classic deep learning, but can be viewed more generally as *templates* for (differentiable) computation. The template structure is encoded in the instance of the *Template* class via relational *rules* or, for convenience, pre-defined modules (which are also expanded into said rules, check out the *Module Zoo* for a list of modules).

```
from neuralogic.core import Template, Settings
from neuralogic.nn.module import GCNConv


template = Template()
template.add_module(
    GCNConv(in_channels=1, out_channels=5, output_name="h0", feature_name="node_feature",
↪ edge_name="edge")
)
template.add_module(
    GCNConv(in_channels=5, out_channels=1, output_name="predict", feature_name="h0",␣
↪edge_name="edge")
)
```

Here we defined two `GCNConv` layers via pre-defined modules. We further discuss template definition via the rule format, which forms the core advantage of this framework, in the section of the documentation.

2.3 Evaluating Model

Now when we have our template defined, we have to get (build) the model from the template to be able to run training and inference on it. We do that by calling the `build` method.

```
from neuralogic.core import Settings, Optimizer

settings = Settings(learning_rate=0.01, optimizer=Optimizer.SGD, epochs=100)
model = template.build(Settings())
```

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 = model.build_dataset(dataset)

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

2.3.1 Evaluators

For faster prototyping, we have prepared *evaluators* which encapsulate helpers, such as training loop and evaluation. Evaluators can then be customized via various settings wrapped 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(template, settings)

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

CHAPTER

THREE

PYNEURALOGIC LANGUAGE

The main feature of the PyNeuraLogic library is its custom declarative language (based on NeuraLogic) for describing the structure of the learning problems, data and models. In PyNeuraLogic, the language is fully embedded in Python, enabling users to utilize Python's convenient modules and features.

The idea of using a custom language (following the logic programming paradigm) instead of the predefined modules, as common in popular frameworks, is to achieve higher expressiveness, reduce the complexity of writing novel model architectures, and reveal the underlying relational principles of the models.

This section introduces users to the language's basic syntax, which is essential for understanding concepts presented in other sections and using the library to its full potential.

3.1 Relations

Relations are fundamental building blocks of the PyNeuraLogic language. Each instance of a relation consists of four parts - *predicate* name, an arbitrary number of *terms*, optional *weight* (or *value*), and optional modifier. Predicate name, together with the "arity" (number of terms) the relation forms its unique signature.

Relations are created via object `Relation` that can be imported from `neuralogic.core`.

Tip: You can also create relations via `R` object, which is a shortcut of `Relation`.

3.1.1 Predicate name

Predicates serve as a descriptive name for the relations. Predicate names are *case-sensitive* and have to start with a *lowercase* letter. Usually, relations with specific predicate names are created directly via `Relation` object (e.g., `Relation.my_rel` creates a relation with the predicate name `my_rel`). For convenience, we can also use the `Relation.get` method (e.g., `Relation.get("my_rel")`), which can be useful for generating relations.

```
from neuralogic.core import Relation

Relation.my_rel  # Relation with a predicate name "my_rel"
```

(continues on next page)

(continued from previous page)

```
for i in range(5):
    # Relations with predicate names "my_rel_0", ..., "my_rel_4"
    Relation.get(f"my_rel_{i}")
```

Note: Prepending the predicate name with an underscore (_) will make the relation "hidden" (e.g., `Relation.hidden.my_rel` is equal to `Relation._my_rel`). You can read more about modifiers, such as "hidden", in the *Modifiers* section.

3.1.2 Terms

Terms are an optional list of *constants* and/or logic *variables*.

- **Constants** are either numeric values (floats, integers) or string values with a lower-cased first letter. We can also define a constant via `neuralogic.core.Term`, which converts the provided value into a valid constant (string) for us.

```
from neuralogic.core import Term, Relation

Relation.my_rel  # A relation with NO terms, also called a "proposition" in logic
Relation.my_rel(1.0)  # A relation with one constant term 1.0
Relation.my_rel(Term.my_term, "string_term")  # A relation with two constant terms "my_
↪term" and "string_term"
Relation.my_rel(1.0, Term.My_Term)   # A relation with two constant terms 1.0 and "my_
↪term"
```

- **Variables** are *capitalized* string values. We can, similarly to constants, utilize helper `neuralogic.core.Var`, which converts the provided value into a valid variable (string) for us.

```
from neuralogic.core import Var, Relation

Relation.my_rel(Var.X)  # A relation with one variable "X"
Relation.my_rel(Var.x, "Y")  # A relation with two variable terms "X" and "Y"
```

Relations with logical variables express general *patterns*, which is essential for encoding deep *relational* models, such as GNNs.

Note: We call relation "ground" if all of its terms are constants (no variables). These are essentially specific (logical) statements, or *facts*, commonly used to encode the data and particular observations.

3.1.3 Weights

On top of classic relational logic programming, in PyNeuraLogic, the relations can be additionally associated with *weights*. A relation's weight is optional and servers as a learnable parameter. The weight itself can be defined in the following ways:

- Scalar value defining a learnable scalar parameter initialized to a specific value.

```
Relation.my_rel[0.5]  # Scalar weight initialized to 0.5
```

- Vector value defining a learnable vector parameter initialized to a specific value.

```
Relation.my_rel[[1.0, 0.0, 1.0]]  # Vector weight initialized to [1.0, 0.0, 1.0]
```

- Matrix value defining a learnable matrix parameter initialized to a specific value.

```
Relation.my_rel[[[1, 0], [0, 1]]]  # Matrix weight initialized to [[1, 0], [0, 1]]
```

Tip: Matrix and vector values can also be in the form of NumPy arrays.

Instead of defining particular values for the parameters, we can also choose to specify merely the dimensionality of it instead. Here, each element of the parameter represents the size of the corresponding dimension. The initialization of the values in this case is sampled from a distribution determined by the `Settings` object.

```
Relation.my_rel[2,]  # Specification of a randomly initialized weight vector of length 2
Relation.my_rel[3, 3]  # Specification of a randomly initialized 3x3 weight matrix
```

Warning: Notice the difference between `Relation.my_rel[2]` and `Relation.my_rel[2,]` where the first one represents a particular scalar weight with *value* "2", while the latter represents a randomly initialized weight vector of *length* 2.

Named Weights

Weight sharing is at the heart of modelling with PyNeuraLogic, where all the (ground) instances of a relation will share its associated parameters. However, you can also choose to share a single weight across multiple relations. This can be achieved by labeling the weight with some name, such as:

```
# Sharing a weight (2x2 matrix weight)
Relation.my_rel["shared_weight": 2, 2]
Relation.another_rel["shared_weight": 2, 2]

# Sharing a weight (vector weight)
Relation.my_rel["my_weight": 2,]
Relation.another_rel["my_weight": 2]
```

3.1.4 Modifiers

Predicate names are generally arbitrary, with no particular meaning other than the user-defined one. However, by including a modifier in the definition of a relation, we may utilize some of the extra pre-defined predicates with special built-in functionality.

More about individual modifiers can be read in *Modifiers*.

3.2 Rules

```
Relation.h <= (Relation.b_one, Relation.b_n)
```

Rules are the core concept in PyNeuraLogic for describing the architectures of the models by defining *templates* for their computational graphs. Each rule consists of two parts - the *head* and the *body*. The head is an arbitrary relation followed by an implication (<=) and subsequently the body formed from a tuple of `n` relations.

When there is only one relation in the body, we can omit the tuple and insert the relation directly.

```
Relation.h <= Relation.b
```

Such a rule can be then read as *"The relation (proposition) 'h' is implied by the relation (proposition) 'b'"*

3.2.1 Metadata

The rules have some (default) properties that influence their translation into the computational graphs (models), such as activation and aggregation functions. These properties can be modified, per rule, by attaching a `Metadata` instance to the rule.

```
from neuralogic.core import Metadata, Activation, Aggregation


(Relation.h <= (Relation.b_one, Relation.b_n)) | Metadata(activation=Activation.RELU,␣
↪aggregation=Aggregation.AVG)

# or, for short, just
(Relation.h <= (Relation.b_one, Relation.b_n)) | [Activation.RELU, Aggregation.AVG]
```

For example, with the construct above, we created a new rule with a specified activation function (relu) and aggregation function (avg).

CHAPTER

FOUR

PROBLEM DEFINITION

To approach relational machine learning problems with the PyNeuraLogic library in its full potential, we generally divide each learning scenario into (i) learning examples, (ii) queries, and (iii) a learning template. A set of learning examples together with the queries form a learning dataset. The learning template then constitues a "lifted" model architecture, i.e. a prescription for unfolding (differentiable) computational graphs.

4.1 Dataset

The dataset object holds factual information about the problem and is divided into two parts - (i) examples and (ii) queries.

> **Attention:** In the context of examples and queries, the "weights" of the relations are, in fact, not learnable parameters but concrete *values* that serve as inputs (example features) or target outputs (query labels).
>
> This means that it is not possible to use the dimensionality definition for the weight (value) in this case, as it does not represent a concrete value.

4.1.1 Examples

An example describes a specific learning instance, such as a graph, generally encoded through the language of *ground* relations/facts and rules. Intuitively, a learning example can be seen as the input to the model defined by a template.

Examples can be loaded from files in various formats, or encoded directly in Python in the NeuraLogic language. For instance, a complete graph with three nodes and some features can be encoded as:

```
from neuralogic.core import Relation, Dataset

dataset = Dataset()

dataset.add_example([
    Relation.edge(1, 2), Relation.edge(2, 1), Relation.edge(1, 3),
    Relation.edge(3, 1), Relation.edge(2, 3), Relation.edge(3, 2),

    Relation.feature(1)[0],
    Relation.feature(2)[1],
    Relation.feature(3)[-1],
])
```

4.1.2 Queries

Queries are relations (facts) corresponding to the desired outputs of the learning model/template. These are commonly associated with (non-learnable) weights determining the expected values of the target (relation) labels, given some input example(s).

We might, for example, want to learn the output values of the unary relation (property) `Relation.h` of the entity `anna` to be `0`, and for the entity `elsa` to be `1`. This might be expressed like this:

```
dataset.add_queries([
    Relation.h('anna')[0],
    Relation.h('elsa')[1],
])
```

Note that, in constrast to classic machine learning labels, queries are not restricted to a single target "output" in the template, such as the "output layer" in classic neural models. We can thus ask different completely arbitray queries at the same time:

```
dataset.add_queries([
    Relation.h('anna')[0],
    Relation.h('elsa')[1],
    Relation.friend('anna','elsa')[1],
])
```

Also, the associated labels can be of arbitrary shapes. We can thus, for example, combine a query `Relation.a[0]` with a scalar label with a query `Relation.b[[1, 0, 1]]` with a vector label, each associated with a different part of the learning template.

Note: Queries are valued ground relations, but we don't have to define the value explicitly. If the value is not present, the default value (`1.0`) is used as the label. This is useful, e.g., for queries outside the learning phase, where the labels are not needed/known.

A single learning example may then be associated with a single query, as common in classic supervised machine learning, or with multiple queries, as common e.g. in knowledge-base completion or collective classification tasks.

Tip: If the learning example does not change and is the same for every query, we can simly define only one example, and it will be reused for each query.

4.2 Template

The template (*`Template`*) is a set of *rules* that encode the lifted model architecture. Intuitively, this is somewhat similar to composing modules in the common deep learning frameworks, but more versatile. The versatility follows from the *declarative* nature of the rules, which can be highly abstract and expressive, just like the modules, yet directly reveal an interface to the underlying lower-level principles of the module's computation.

4.2.1 Interpretation of Rules

TODO: Understanding rules

CHAPTER

FIVE

UNDERSTANDING RULES

In PyNeuraLogic, describing a learning model differs from conventional deep learning frameworks. Here, instead of putting together a sequence of modules and operations on numeric tensors, we define a model "template" formed from *rules* operating on *relations*. This template is then used to unfold differentiable computational graphs, which may be tailored for each (relational) learning sample.

But how exactly do these rules translate into computational graphs?

The semantics of this process follows directly from logical inference, and is described in detail in the paper(s) on Lifted Relational Neural Networks However, let us skip the scientific notions here and take a direct look at the process through a simple example instead!

Consider the following relatively simple graph with arbitrarily picked node ids. Usually, we would encode the graph either as an adjacency matrix, or simply a list of edges with two vectors - `[[sources], [destinations]]`. The latter representation for graphs is also available in PyNeuraLogic for convenience, but we will stick with the more general relational representation here, and describe the graph edges as `R.edge(<source>, <destination>)`, that is:

```
[
    R.edge(3, 1), R.edge(2, 1), R.edge(4, 3), R.edge(4, 1),
    R.edge(1, 3), R.edge(1, 2), R.edge(3, 4), R.edge(1, 4),
]
```

And just like that, we encoded our example input graph with bidirectional edges. Let us now define few example templates to operate upon this graph, and dive into how they are being compiled into *computational* graphs.

5.1 An Entry Template

```
R.h(V.X) <= R.edge(V.X, V.Y)
```

The first template is relatively simple; it contains one rule with only one body relation `R.edge(V.X, V.Y)`. The rule roughly translates into plain English as

> "To compute representation `h` of any entity `X`, aggregate all values of relations `R.edge` where the entity `X` is the *"source" node* of the relation edge."

So, for example, for a query `R.h(1)`, there are exactly three instances of the edge relation that satisfy the template rule - `R.edge(1, 2)`, `R.edge(1, 3)`, and `R.edge(1, 4)`, corresponding to the three neighbors of the node 1. So in the end, we end up with a computational graph like the one below. The computation in the graph goes from the bottom (input) level up to the output level, which corresponds to our query.

Note: Notice that the input value of all the edge relations is 1. This value has been set implicitly because we didn't provide any.

This visualization renders only the graph's *structure* without specification of the operations on the values being passed through. However, every node in this graph can be associated with some function.

In this case, let us focus on the only node with multiple inputs, highlighted with the magenta color. This node is the so-called *aggregation* node that aggregates all of its inputs through some aggregation function (*AVG* by default) to produce a single output value.

```
value = AVG (value(...edge(1,2)), value(...edge(1,3)), value(...edge(1,4)))
```

Note how this functionality can be viewed as a basis for the "neighborhood aggregation" operation commonly utilized in Graph Neural Networks.

Note: What if we have a node without any edge and want to compute the `R.h`? We will get an exception because we cannot satisfy the rule. Later in this tutorial, we will look at solutions to such a scenario.

5.2 Multiple Body Relations

Our first template was very limited in what we were able to express. We will often find ourselves declaring rules with multiple body relations to capture more complicated computational patterns. As an example of such a template rule, we could introduce a *feature* relation for the nodes and utilize it in the rule. Also, we will introduce weights to the rule at the same time.

```
R.h(V.X)["a": 1,] <= (R.edge(V.X, V.Y)["b": 1,], R.feature(V.Y)["c": 1,])
```

Note: We used named weights here to clarify how the weights are being mapped into the computational graph. However you can normally omit these names.

Now, let us extend our input data (the encoding of the input graph) with some node features correspondingly. For simplicity, each feature will be a simple scalar value, for example:

```
R.feature(1)[0.2], R.feature(2)[0.3], R.feature(3)[0.4], R.feature(4)[0.5]
```

Now, for the same query `R.h(1)`, we will end up with the computational graph below. Note how the bottom layer expanded with additional inputs (`R.feature`), and how the weights came upe associated with the corresponding edges.

Let us now focus on a different "level" in the computational graph. This time, we highlight the nodes that correspond to the rule's body (these were present in the previous example, too, however they were not so interesting as there was only one body relation at the input). In this case of a multitude of relations in the body of the rule, these again need to be *combined* somehow. By default, this operation is a weighted summation of the inputs with a nonlinearity (tanh) on top. Thus, for example, the value of the leftmost magenta node will be calculated as follows:

```
value = tanh ( (0.3 * c) + (1 * b) )
```

5.3 Multiple Rules

Now that we understand how multiple relations in the body of a rule are combined, and how the different instantiations of the body are aggregated, let us look at a scenario with two different rules with the same head relation.

```
R.h(V.X) <= (R.edge(V.X, V.Y), R.feature(V.Y)),
R.h(V.X) <= R.feature(V.X),
```

Up until now, to successfully derive `R.h`, the nodes were required to have edges. To mitigate this, we can add a second rule which will be satisfied for any node with some features. Let us take a look at how the mapping changed for this template on the same query `R.h(1)`

Now, this additional rule introduced the rightmost branch highlighted with the magenta color. Note that this branch has the same structure as the left one, i.e. there is an aggregation node and node that "combines" the body relations. Nevertheles, in this case, there isn't much to combine nor aggregate.

Another interesting point to note here is the operation of the topmost node that corresponds to the query, which now has multiple inputs, too. Consequently, these need to be combined somehow which, by default, is a (weighted) summation again.

5.4 Graph Readout

Up until now, we have been working with queries on top of one entity - node. What if we wanted to compute the value of relation `R.h` for all available nodes and then somehow aggregate them into one value, i.e., do a "graph readout"?

For that, we can yet again leverage the elegant expressiveness of relational logic. We can simply state, "Aggregate all values of the relation `R.h` for all entities `X` that satisfy the relation." Let us use a different query, `R.q`, for the readout in this case.

```
R.h(V.X) <= (R.edge(V.X, V.Y), R.feature(V.Y)),
R.h(V.X) <= R.feature(V.X),
R.q <= R.h(V.X),
```

In this case, there are no new operations to be discussed in the computational graph shown below. All of the `R.h` node computation will be unfolded into their respective subgraphs, e.g., the `R.h(1)` node will be unfolded to the graph from the previous example above.

Note: Note that the computational subgraphs for the individual nodes here will not be completely separate, i.e. the computational graph will not be a tree anymore, since the nodes share some of their neighbors in the input graph, too.

5.5 Activation and Aggregation functions

So far we focused solely on the *structure* of the computational graph, without specificying the indivudal operations/functions associated with the nodes. Let us now demonstrate how to customize these. For that, let is consider again the graph/template from the first (entry) example.

```
R.h(V.X) <= R.edge(V.X, V.Y)
```

If we would like to change the *aggregation* function of the rule, i.e. how all the values of the edges of each node are being aggregated, we can append that information to the rule as

```
(R.h(V.X) <= R.edge(V.X, V.Y)) | [Aggregation.MAX]
```

Should we want to further change the non-linear activation of the rule nodes, combining the rule body relations we would add:

```
(R.h(V.X) <= R.edge(V.X, V.Y)) | [Aggregation.MAX, Activation.SIGMOID]
```

Finally, to change the activation function of the *head* of the rule in the case with multiple rules with the same head:

```
R.h(V.X) <= (R.edge(V.X, V.Y), R.feature(V.Y)),
R.h(V.X) <= R.feature(V.X),
```

we would append that information to the head relation itself as:

```
R.h / 1 | [Activation.SIGMOID]
```

Note: The `/ 1` here defines the "arity" of the relation, which is necessary to uniquely identify the relation, since we can have multiple relations of the same name with different arities (and activation functions).

CHAPTER

SIX

MODEL EVALUATION

6.1 Model Building

When we have the template, examples, and queries ready, we need to 'compile' them together to retrieve a model that can be trained and evaluated.

The 'compilation' is done in two steps. Firstly, we retrieve a model instance for the specified backend.

```
from neuralogic.core import Backend, Settings

settings = Settings()
model = template.build(settings)
```

Then we can 'build' the examples and queries (dataset), yielding a multitude of computational graphs to be trained.

```
built_dataset = model.build_dataset(dataset)
```

6.2 Saving and Loading Model

When our model is trained, or we want to persist the model's state (e.g., make a checkpoint), we can utilize the model instance method *state_dict()* (or *parameters()*). The method puts all parameters' values into a dictionary that can be later saved (e.g., in JSON or in binary) or somehow manipulated.

When we want to load a state into our model, we can then simply pass the state into *load_state_dict()* method.

Note: Evaluators offer the same interface for saving/loading of the model.

6.3 Utilizing Evaluators

Writing custom training loops and handling different backends can be cumbersome and repetitive. The library offers 'evaluators' that encapsulate the training loop and testing evaluation. Evaluators also handle other responsibilities, such as building datasets.

```
from neuralogic.nn import get_evaluator


evaluator = get_evaluator(template, settings, Backend.JAVA)
```

Once you have an evaluator, you can evaluate or train the model on a dataset. The dataset doesn't have to be pre-built, as in the case of classical evaluation - the evaluator handles that for you.

Note: If it is used more than once, it is more efficient to pass a pre-built dataset into the evaluator (this will prevent redundant dataset building).

6.3.1 Settings Instance

The *Settings* instance contains all the settings used to customize the behavior of different parts of the library.

Most importantly, it affects the behavior of the model building (e.g., specify default rule/relation activation functions), evaluators (e.g., error function, number of epochs, learning rate, optimizer), and the model itself (e.g., initialization of the learnable parameters).

```
from neuralogic.core import Settings, Optimizer, Initializer
from neuralogic.nn.init import Uniform


Settings(
    initializer=Uniform(),
    optimizer=Optimizer.SGD,
    learning_rate=0.1,
    epochs=100,
)
```

In the example above, we define settings to ensure that initial values of learnable parameters (of the model these settings are used for) are sampled from the uniform distribution. We also set properties utilized by evaluators: the number of epochs (100) and the optimizer, which is set to Stochastic gradient descent (SGD) with a learning rate of 0.1.

6.3.2 Evaluator Training/Testing Interface

The evaluator's basic interface consists of two methods - `train` and `test` for training on a dataset and evaluating on a dataset, respectively. Both methods have the same interface and are implemented in two modes - generator and non-generator.

The generator mode (default mode) yields a tuple of two elements (total loss and number of instances/samples) per each epoch. This mode can be useful when we want to, for example, visualize, log or do some other manipulations in real-time during the training (or testing).

```
for total_loss, seen_instances in neuralogic_evaluator.train(dataset):
    pass
```

The non-generator mode, on the other hand, returns only a tuple of metrics from the last epoch.

```
results = neuralogic_evaluator.train(dataset, generator=False)
```

CHAPTER

SEVEN

MODULE ZOO

Welcome to our module zoo, the place where we discuss all pre-defined modules and outline how they are mapped to logic programs.

All modules listed here are defined in the `neuralogic.nn.module` package, and their usage is quite similar to the usage of regular rules. You can add them to your template via the += operator or `add_module` method, e.g.:

```
from neuralogic.nn.module import GCNConv

template += GCNConv(...)
# or
template.add_module(GCNConv(...))
```

Right after adding a module into a template, it is expanded into logic form - rules. This allows you to build upon pre-defined modules and create new variations by adding your own custom rules or just mixing modules together.

7.1 Pre-defined Modules

GNN

General Blocks

Meta

Module	Edge formats
GCNConv	R.<edge_name>(<source>, <target>)
SAGEConv	R.<edge_name>(<source>, <target>)
GINConv	R.<edge_name>(<source>, <target>)
RGCNConv	R.<edge_name>(<source>, <relation>, <target>) or R.<relation>(<source>, <target>)
TAGConv	R.<edge_name>(<source>, <target>)
GATv2Conv	R.<edge_name>(<source>, <target>)
SGConv	R.<edge_name>(<source>, <target>)
APPNPConv	R.<edge_name>(<source>, <target>)
ResGatedGraphConv	R.<edge_name>(<source>, <target>)

Module	
Linear	
MLP	

Recurrent/Recursive module	
RvNN	
RNN	
GRU	
LSTM	

Pooling module	
Pooling	
SumPooling	
AvgPooling	
MaxPooling	

Module	
MetaConv	
MAGNNMean	
MAGNNLinear	

7.2 GNN Modules

class GCNConv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Graph Convolutional layer from "Semi-supervised Classification with Graph Convolutional Networks". Which can be expressed as:

$$\mathbf{x}'_i = act(\mathbf{W} \cdot agg_{j \in \mathcal{N}(i)}(\mathbf{x}_j))$$

Where *act* is an activation function, *agg* aggregation function and W is a learnable parameter. This equation is translated into the logic form as:

```
(R.<output_name>(V.I)[<W>] <= (R.<feature_name>(V.J), R.<edge_name>(V.J, V.I))) | [
↪<aggregation>, Activation.IDENTITY]
R.<output_name> / 1 | [<activation>]
```

Examples

The whole computation of this module (parametrized as `GCNConv(2, 3, "h1", "h0", "_edge")`) is as follows:

```
(R.h1(V.I)[3, 2] <= (R.h0(V.J), R._edge(V.J, V.I)) | [Aggregation.SUM, Activation.
↪IDENTITY]
R.h1 / 1 | [Activation.IDENTITY]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.SUM`

class SAGEConv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

GraphSAGE layer from "Inductive Representation Learning on Large Graphs". Which can be expressed as:

$$\mathbf{x}'_i = act(\mathbf{W}_1\mathbf{x}_i + \mathbf{W}_2 \cdot agg_{j\in\mathcal{N}(i)}(\mathbf{x}_j)))$$

Where *act* is an activation function, *agg* aggregation function and *W*'s are learnable parameters. This equation is translated into the logic form as:

```
(R.<output_name>(V.I)[<W1>] <= (R.<feature_name>(V.J), R.<edge_name>(V.J, V.I))) | [
↪<aggregation>, Activation.IDENTITY]
(R.<output_name>(V.I)[<W2>] <= R.<feature_name>(V.I)) | [Activation.IDENTITY]
R.<output_name> / 1 | [<activation>]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.AVG`

class GINConv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

class RGCNConv(*in_channels: int, out_channels: int, output_name: str, feature_name: str, edge_name: ~typing.Optional[str], relations: ~typing.List[str], activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>, aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Relational Graph Convolutional layer from Modeling Relational Data with Graph Convolutional Networks. Which can be expressed as:

$$\mathbf{x}'_i = act(\mathbf{W_0} \cdot \mathbf{x}_i + \sum_{r \in \mathcal{R}} agg_{j \in \mathcal{N}_r(i)}(\mathbf{W_r} \cdot \mathbf{x}_j))$$

Where *act* is an activation function, *agg* aggregation function (by default average), W_0 is a learnable root parameter and W_r is a learnable parameter for each relation.

The first part of the equation that is "$\mathbf{W_0} \cdot \mathbf{x}_i$" can be expressed in the logic form as:

```
R.<output_name>(V.I) <= R.<feature_name>(V.I)[<W0>]
```

Another part of the equation that is "$agg_{j \in \mathcal{N}_r(i)}(\mathbf{W_r} \cdot \mathbf{x}_j)$" can be expressed as:

```
R.<output_name>(V.I) <= (R.<feature_name>(V.J)[<Wr>], R.<edge_name>(V.J, relation,␣
↪V.I))
```

where "relation" is a constant name, or as:

```
R.<output_name>(V.I) <= (R.<feature_name>(V.J)[<Wr>], R.<relation>(V.J, V.I))
```

The outer summation, together with summing it with the first part, is handled by aggregation of all rules with the same head (and substitution).

Examples

The whole computation of this module (parametrized as `RGCNConv(1, 2, "h1", "h0", "_edge", ["sibling", "parent"])`) is as follows:

```
metadata = Metadata(activation=Activation.IDENTITY, aggregation=Aggregation.AVG)

(R.h1(V.I) <= R.h0(V.I)[2, 1]) | metadata
(R.h1(V.I) <= (R.h0(V.J)[2, 1], R._edge(V.J, sibling, V.I))) | metadata
(R.h1(V.I) <= (R.h0(V.J)[2, 1], R._edge(V.J, parent, V.I))) | metadata
R.h1 / 1 [Activation.IDENTITY]
```

Module parametrized as `RGCNConv(1, 2, "h1", "h0", None, ["sibling", "parent"])` translates into:

```
metadata = Metadata(activation=Activation.IDENTITY, aggregation=Aggregation.AVG)

(R.h1(V.I) <= R.h0(V.I)[2, 1]) | metadata
(R.h1(V.I) <= (R.h0(V.J)[2, 1], R.sibling(V.J, V.I))) | metadata
(R.h1(V.I) <= (R.h0(V.J)[2, 1], R.parent(V.J, V.I))) | metadata
R.h1 / 1 [Activation.IDENTITY]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*Optional[str]*) – Edge predicate name to use for neighborhood relations. When `None`, elements from `relations` are used instead.
- **relations** (*List[str]*) – List of relations' names
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.SUM`

class TAGConv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *k: int = 2*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Topology Adaptive Graph Convolutional layer from "Topology Adaptive Graph Convolutional Networks". Which can be expressed as:

$$\mathbf{x}'_i = act(\sum_{k=0}^{K} \mathbf{W}_k \cdot agg_{j \in \mathcal{N}^k(i)}(\mathbf{x}_j))$$

Where *act* is an activation function, *agg* aggregation function, *Wk* are learnable parameters and $\mathcal{N}^k(i)$ denotes nodes that are *k* hops away from the node *i*. This equation is translated into the logic form as:

This equation is translated into the logic form as:

```
(R.<output_name>(V.I0)[<W0>] <= R.<feature_name>(V.I0)) | [<aggregation>,␣
↪Activation.IDENTITY]
(R.<output_name>(V.I0)[<W1>] <= (R.<feature_name>(V.I1), R.<edge_name>(V.I1, V.
↪I0))) | [<aggregation>, Activation.IDENTITY]
(R.<output_name>(V.I0)[<W2>] <= (R.<feature_name>(V.I2), R.<edge_name>(V.I1, V.I0),␣
↪R.<edge_name>(V.I2, V.I1)) | [<aggregation>, Activation.IDENTITY]
...
(R.<output_name>(V.I0)[<Wk>] <= (R.<feature_name>(V.I<k>), R.<edge_name>(V.I1, V.
↪I0), ..., R.<edge_name>(V.I<k>, V.I<k-1>)) | [<aggregation>, Activation.IDENTITY]
R.<output_name> / 1 | [<activation>]
```

Examples

The whole computation of this module (parametrized as `TAGConv(1, 2, "h1", "h0", "_edge")`) is as follows:

```
(R.h1(V.I0)[2, 2] <= R.h0(V.I0)) | [Aggregation.SUM, Activation.IDENTITY]
(R.h1(V.I0)[2, 1] <= (R.h0(V.I1), R._edge(V.I1, V.I0)) | [Aggregation.SUM,␣
↪Activation.IDENTITY]
(R.h1(V.I0)[2, 1] <= (R.h0(V.I2), R._edge(V.I1, V.I0), R._edge(V.I2, V.I1)) |␣
↪[Aggregation.SUM, Activation.IDENTITY]
R.h1 / 1 | [Activation.IDENTITY]
```

Module parametrized as `TAGConv(1, 2, "h1", "h0", "_edge", 1)` translates into:

```
(R.h1(V.I0)[2, 1] <= R.h0(V.I0)) | [Aggregation.SUM, Activation.IDENTITY]
(R.h1(V.I0)[2, 1] <= (R.h0(V.I1), R._edge(V.I1, V.I0)) | [Aggregation.SUM,␣
↪Activation.IDENTITY]
R.h1 / 1 | [Activation.IDENTITY]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **k** (*int*) – Number of hops. Default: `2`
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.SUM`

class GATv2Conv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *share_weights: bool = False*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*)

GATv2 layer from "How Attentive are Graph Attention Networks?".

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **share_weights** (*bool*) – Share weights in attention. Default: `False`
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`

class SGConv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *k: int = 1*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Simple Graph Convolutional layer from "Simplifying Graph Convolutional Networks". Which can be expressed as:

$$\mathbf{x}_i' = act(\mathbf{W} \cdot agg_{j \in \mathcal{N}^k(i)}(\mathbf{x}_j))$$

Where *act* is an activation function, *agg* aggregation function, W is a learnable parameter and $\mathcal{N}^k(i)$ denotes nodes that are k hops away from the node i. This equation is translated into the logic form as:

```
(R.<output_name>(V.I)[<W>] <= (
    R.<feature_name>(V.I<k>),
    R.<edge_name>(V.I<1>, V.I<0>), R.<edge_name>(V.I<2>, V.I<1>), ..., R.<edge_name>
↪(V.I<k>, V.I<k-1>),
)) | [<aggregation>, Activation.IDENTITY]

R.<output_name> / 1 | [<activation>]
```

Examples

The whole computation of this module (parametrized as `SGConv(2, 3, "h1", "h0", "_edge", 2)`) is as follows:

```
(R.h1(V.I0)[3, 2] <= (R.h0(V.I2), R._edge(V.I1, V.I0), R._edge(V.I2, V.I1))) |␣
↪[Activation.IDENTITY, Aggregation.SUM]
R.h1 / 1 | [Activation.IDENTITY]
```

Module parametrized as `SGConv(2, 3, "h1", "h0", "_edge", 1)` translates into:

```
(R.h1(V.I0)[3, 2] <= (R.h0(V.I1), R._edge(V.I1, V.I0))) | [Activation.IDENTITY,␣
↪Aggregation.SUM]
R.h1 / 1 | [Activation.IDENTITY]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **k** (*int*) – Number of hops. Default: 1
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.SUM`

class APPNPConv(*output_name: str*, *feature_name: str*, *edge_name: str*, *k: int*, *alpha: float*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Approximate Personalized Propagation of Neural Predictions layer from "Predict then Propagate: Graph Neural Networks meet Personalized PageRank". Which can be expressed as:

$$\mathbf{x}_i^0 = \mathbf{x}_i$$

$$\mathbf{x}_i^k = \alpha \cdot \mathbf{x}_i^0 + (1 - \alpha) \cdot agg_{j \in \mathcal{N}(i)}(\mathbf{x}_j^{k-1})$$

$$\mathbf{x}'_i = act(\mathbf{x}_i^K)$$

Where *act* is an activation function and *agg* aggregation function.

The first part of the second equation that is "$\alpha \cdot \mathbf{x}_i^0$" is expressed in the logic form as:

```
R.<output_name>__<k>(V.I) <= R.<feature_name>(V.I)[<alpha>].fixed()
```

The second part of the second equation that is "$(1-\alpha) \cdot agg_{j \in \mathcal{N}(i)}(\mathbf{x}_j^{k-1})$" is expressed as:

```
R.<output_name>__<k>(V.I) <= (R.<output_name>__<k-1>(V.J)[1 - <alpha>].fixed(), R.
↪<edge_name>(V.J, V.I))
```

Examples

The whole computation of this module (parametrized as `APPNPConv("h1", "h0", "_edge", 3, 0.1, Activation.SIGMOID)`) is as follows:

```
metadata = Metadata(activation=Activation.IDENTITY, aggregation=Aggregation.SUM)

(R.h1__1(V.I) <= R.h0(V.I)[0.1].fixed()) | metadata
(R.h1__1(V.I) <= (R.h0(V.J)[0.9].fixed(), R._edge(V.J, V.I))) | metadata
R.h1__1/1 [Activation.IDENTITY]

(R.h1__2(V.I) <= <0.1> R.h0(V.I)) | metadata
(R.h1__2(V.I) <= (<0.9> R.h1__1(V.J), R._edge(V.J, V.I))) | metadata
R.h1__2/1 [Activation.IDENTITY]

(R.h1(V.I) <= <0.1> R.h0(V.I)) | metadata
(R.h1(V.I) <= (<0.9> R.h1__2(V.J), R._edge(V.J, V.I))) | metadata
R.h1 / 1 [Activation.SIGMOID]
```

Parameters

- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **k** (*int*) – Number of iterations
- **alpha** (*float*) – Teleport probability
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.SUM`

class ResGatedGraphConv(*in_channels: int*, *out_channels: int*, *output_name: str*, *feature_name: str*, *edge_name: str*, *gating_activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Residual Gated Graph Convolutional layer from "Residual Gated Graph ConvNets". Which can be expressed as:

$$\mathbf{x}'_i = act(\mathbf{W}_1\mathbf{x}_i + agg_{j\in\mathcal{N}(i)}(\eta_{i,j} \odot \mathbf{W}_2\mathbf{x}_j))$$

$$\eta_{i,j} = gating\_act(\mathbf{W}_3\mathbf{x}_i + \mathbf{W}_4\mathbf{x}_j)$$

Where *act* is an activation function, *agg* aggregation function, *gating_act* is a gating activation function and W_n are learnable parameters. This equation is translated into the logic form as:

```
(R.<output_name>__gate(V.I, V.J) <= (R.<feature_name>(V.I)[<W>], R.<feature_name>(V.
↪J)[<W>])) | [Activation.IDENTITY]
R.<output_name>__gate / 2 | [<activation>]

(R.<output_name>(V.I) <= R.<feature_name>(V.I)[<W>]) | [Activation.IDENTITY]
(R.<output_name>(V.I) <= (
    R.<output_name>__gate(V.I, V.J), R.<feature_name>(V.J)[<W>], R.<edge_name>(V.J,␣
↪V.I))
) | Metadata(activation="elementproduct-identity", aggregation=<aggregation>)

R.<output_name> / 1 | [<activation>]
```

Examples

The whole computation of this module (parametrized as `ResGatedGraphConv(1, 2, "h1", "h0", "_edge")`) is as follows:

```
metadata = Metadata(activation="elementproduct-identity", aggregation=Aggregation.
↪SUM)

(R.h1__gate(V.I, V.J) <= (R.h0(V.I)[2, 1], R.h0(V.J)[2, 1])) | [Activation.IDENTITY]
R.h1__gate / 2 | [Activation.SIGMOID]

(R.h1(V.I) <= R.h0(V.I)[2, 1]) | [Activation.IDENTITY]
(R.h1(V.I) <= (R.h1__gate(V.I, V.J), R.h0(V.J)[2, 1], R._edge(V.J, V.I))) | metadata
R.h1 / 1 | [Activation.IDENTITY]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **edge_name** (*str*) – Edge predicate name to use for neighborhood relations.
- **gating_activation** (*Activation*) – Gating activation function. Default: `Activation.SIGMOID`
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.IDENTITY`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.SUM`

7.3 General Block Modules

class Linear(*in_channels: int*, *out_channels: int*, *output_name: str*, *input_name: str*, *activation: ~neuralogic.core.constructs.function.Function = <neuralogic.core.constructs.function.Activation object>*, *arity: int = 1*)

Apply linear transformation on the input. Can be expressed as:

$$h_{i_0,..,i_n} = W \cdot x_{i_0,..,i_n}$$

Where x is the input, $W \in R^{(out\_channels \times in\_channels)}$ is a learnable parameter, and n is the arity of the input and output.

It is also possible to attach non-linearity via the activation parameter and compute:

$$h_{i_0,..,i_n} = act(W \cdot x_{i_0,..,i_n})$$

Example

The whole computation of this module (parametrized as `Linear(1, 2, "h1", "h0")`) is as follows:

```
(R.h1(V.X0)[2, 1] <= R.h0(V.X0)) | [Activation.IDENTITY]
R.h1 / 1 | [Activation.IDENTITY]
```

Module parametrized as `Linear(1, 2, "h1", "h0", Activation.SIGMOID, 2)` translates into:

```
(R.h1(V.X0, V.X1)[2, 1] <= R.h0(V.X0, V.X1)) | [Activation.IDENTITY]
R.h1 / 2 | [Activation.SIGMOID]
```

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input name.
- **activation** (*Function*) – Activation function of the output. Default: `Activation.IDENTITY`
- **arity** (*int*) – Arity of the input and output predicate. Default: 1

class MLP(*units: ~typing.List[int]*, *output_name: str*, *input_name: str*, *activation: ~typing.Union[~neuralogic.core.constructs.function.Function, ~typing.List[~neuralogic.core.constructs.function.Function]] = <neuralogic.core.constructs.function.Activation object>*)

Parameters

- **units** (*List[int]*) – List of layer sizes.
- **output_name** (*str*) – Output (head) predicate name of the module.

- **input_name** (*str*) – Input name.
- **activation** (*Union[Function, List[Function]]*) – Activation function of all layers or list of activations for each layer. Default: `Activation.RELU`

class RvNN(*input_size: int*, *output_name: str*, *input_name: str*, *parent_map_name: str*, *max_children: int = 2*, *activation: ~neuralogic.core.constructs.function.Function = <neuralogic.core.constructs.function.Activation object>*, *aggregation: ~neuralogic.core.constructs.function.Function = <neuralogic.core.constructs.function.Aggregation object>*, *arity: int = 1*)

Recursive Neural Network (RvNN) module which is computed as:

$$\mathbf{h}_i = act(agg_{j \in \mathcal{C}\langle\langle\rangle\rangle}(\mathbf{W}_{\mathbf{id(j)}}\mathbf{h}_j))$$

Where act is an activation function, agg aggregation function and $\mathbf{W}$'s are learnable parameters. $\mathcal{C}\langle\langle\rangle\rangle$ represents the ordered list of children of node i. The $id(j)$ function maps node j to its index (position) in its parent's children list.

Parameters

- **input_size** (*int*) – Input feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input feature predicate name to get leaf features from.
- **parent_map_name** (*str*) – Name of the predicate to get mapping from parent to children
- **max_children** (*int*) – Maximum number of children (specify which <max_children>-ary tree will be considered). Default: `2`
- **activation** (*Function*) – Activation function of all layers. Default: `Activation.TANH`
- **aggregation** (*Function*) – Aggregation function of a layer. Default: `Activation.SUM`
- **arity** (*int*) – Arity of the input and output predicate (doesn't include the node id term). Default: `1`

class RNN(*input_size: int*, *hidden_size: int*, *sequence_length: int*, *output_name: str*, *input_name: str*, *hidden_0_name: str*, *activation: ~neuralogic.core.constructs.function.Function = <neuralogic.core.constructs.function.Activation object>*, *arity: int = 1*, *next_name: str = '_next__positive'*)

One-layer Recurrent Neural Network (RNN) module which is computed as:

$$h_t = act(\mathbf{W}_{ih}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1})$$

where $t \in (1, sequence\_length + 1)$ is a time step. In the template, the t is referred to as `V.T`, and $t - 1$ is referred to as `V.Z`. This module expresses the first equation as:

```
(R.<output_name>(<...terms>, V.T) <= (
    R.<input_name>(<...terms>, V.T)[<hidden_size>, <input_size>],
    R.<hidden_input_name>(<...terms>, V.Z)[<hidden_size>, <hidden_size>],
    R.<next_name>(V.Z, V.T),
)) | [<activation>]

R.<output_name> / <arity> + 1 | [Activation.IDENTITY]
```

Additionally, we define rules for the recursion purpose (the positive integer sequence `R.<next_name>(V.Z, V.T)`) and the "stop condition", that is:

```
(R.<output_name>(<...terms>, 0) <= R.<hidden_0_name>(<...terms>)) | [Activation.
↪IDENTITY]
```

Parameters

- **input_size** (*int*) – Input feature size.
- **hidden_size** (*int*) – Output and hidden feature size.
- **sequence_length** (*int*) – Sequence length.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input feature predicate name to get features from.
- **hidden_0_name** (*str*) – Predicate name to get initial hidden state from.
- **activation** (*Function*) – Activation function. Default: `Activation.TANH`
- **arity** (*int*) – Arity of the input and output predicate. Default: `1`
- **next_name** (*str*) – Predicate name to get positive integer sequence from. Default: `_next__positive`

class GRU(*input_size: int*, *hidden_size: int*, *sequence_length: int*, *output_name: str*, *input_name: str*, *hidden_0_name: str*, *arity: int = 1*, *next_name: str = '_next__positive'*)

One-layer Gated Recurrent Unit (GRU) module which is computed as:

$$r_t = \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1})$$

$$z_t = \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1})$$

$$n_t = \tanh(\mathbf{W}_{xn}\mathbf{x}_t + r_t \odot (\mathbf{W}_{hn}\mathbf{h}_{t-1}))$$

$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{t-1}$$

where $t \in (1, sequence\_length + 1)$ is a time step. In the template, the t is referred to as `V.T`, and $t - 1$ is referred to as `V.Z`. This module expresses the first equation as:

```
(R.<output_name>__r(<...terms>, V.T) <= (
    R.<input_name>(<...terms>, V.T)[<hidden_size>, <input_size>],
    R.<hidden_input_name>(<...terms>, V.Z)[<hidden_size>, <hidden_size>],
    R.<next_name>(V.Z, V.T),
)) | [Activation.SIGMOID]

R.<output_name>__r / <arity> + 1 | [Activation.IDENTITY]
```

The second equation is expressed in the same way, except for a different head predicate name. The third equation is split into three rules. The first two computes the element-wise product - $r_t * (\mathbf{W}_{hn}\mathbf{h}_{t-1})$.

```
(R.<output_name>__n_helper_weighted(<...terms>, V.T) <= (
    R.<hidden_input_name>(<...terms>, V.Z)[<hidden_size>, <hidden_size>], R.<next_
↪name>(V.Z, V.T),
)) | [Activation.IDENTITY],

R.<output_name>__n_helper_weighted / (<arity> + 1) | [Activation.IDENTITY],

(R.<output_name>__n_helper(<...terms>, V.T) <= (
```

(continues on next page)

(continued from previous page)

```
    R.<output_name>__r(<..terms>, V.T), R.<>__n_helper_weighted(<...terms>, V.T)
)) | Metadata(activation="elementproduct-identity"),

R.<output_name>__n_helper / (<arity> + 1) | [Activation.IDENTITY],
```

The third one computes the sum and applies the $tanh$ activation function.

```
(R.<output_name>__n(<...terms>, V.T) <= (
    R.<input_name>(<...terms>, V.T)[<hidden_size>, <input_size>],
    R.<output_name>__n_helper(<...terms>, V.T)
)) | [Activation.TANH]
R.<output_name>__n / (<arity> + 1) | [Activation.IDENTITY],
```

The last equation is computed via three rules. The first two rules computes element-wise products. That is:

```
(R.<output_name>__left(<...terms>, V.T) <= (
    R.<output_name>__z(<...terms>, V.T), R.<output_name>__n(<...terms>, V.T)
)) | Metadata(activation="elementproduct-identity")

(R.<output_name>__right(<...terms>, V.T) <= (
    R.<output_name>__z(<...terms>, V.T), R.<hidden_input_name>(<...terms>, V.Z), R.
↪<next_name>(V.Z, V.T),,
)) | Metadata(activation="elementproduct-identity")

R.<output_name>__left / <arity> + 1 | [Activation.IDENTITY]
R.<output_name>__right / <arity> + 1 | [Activation.IDENTITY]
```

The last output rule sums up the element-wise products.

```
(R.<output_name>(<...terms>, V.T) <= (
    R.<output_name>__left(<...terms>, V.T), R.<output_name>__right(<...terms>, V.T)
)) | [Activation.IDENTITY]
R.<output_name> / <arity> + 1 | [Activation.IDENTITY],
```

Additionally, we define rules for the recursion purpose (the positive integer sequence `R.<next_name>(V.Z, V.T)`) and the "stop condition", that is:

```
(R.<output_name>(<...terms>, 0) <= R.<hidden_0_name>(<...terms>)) | [Activation.
↪IDENTITY]
```

Parameters

- **input_size** (*int*) – Input feature size.
- **hidden_size** (*int*) – Output and hidden feature size.
- **sequence_length** (*int*) – Sequence length.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input feature predicate name to get features from.
- **hidden_0_name** (*str*) – Predicate name to get initial hidden state from.
- **arity** (*int*) – Arity of the input and output predicate. Default: 1

- **next_name** (*str*) – Predicate name to get positive integer sequence from. Default: `_next__positive`

class LSTM(*input_size: int*, *hidden_size: int*, *sequence_length: int*, *output_name: str*, *input_name: str*, *hidden_0_name: str*, *cell_state_0_name: str*, *arity: int = 1*, *next_name: str = '_next__positive'*)

One-layer Long Short-Term Memory (LSTM) RNN module which is computed as:

$$i_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1})$$

$$f_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1})$$

$$o_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1})$$

$$g_t = \tanh(\mathbf{W}_{xg}\mathbf{x}_t + \mathbf{W}_{hg}\mathbf{h}_{t-1})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

Parameters

- **input_size** (*int*) – Input feature size.
- **hidden_size** (*int*) – Output and hidden feature size.
- **sequence_length** (*int*) – Sequence length.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input feature predicate name to get features from.
- **hidden_0_name** (*str*) – Predicate name to get initial hidden state from.
- **cell_state_0_name** (*str*) – Predicate name to get initial cell state from.
- **arity** (*int*) – Arity of the input and output predicate. Default: 1
- **next_name** (*str*) – Predicate name to get positive integer sequence from. Default: `_next__positive`

class Pooling(*output_name: str*, *input_name: str*, *aggregation: Function*, *input_arity: int = 1*)

Apply generic pooling over the input specified by the input_name and the input arity parameters. Can be expressed as:

$$h = agg_{i_0,..,i_n \in N}(x_{(i_0,..,i_n)})$$

Where N is a set of tuples of length n (specified by the input arity parameter) that are valid arguments for the input predicate.

For example, a classic pooling over graph nodes represented by relations of arity 1 (node id) would be calculated as:

$$h = agg_{i \in N}(x_{(i)})$$

Here N refers to a set of all node ids. Lifting the restriction of the input arity via the input_arity parameter allows for pooling not only nodes but also edges (`input_arity=2`) and other objects (hyperedges etc.)

Examples

The whole computation of this module (parametrized as `Pooling("h1", "h0", Aggregation.AVG)`) is as follows:

```
(R.h1 <= R.h0(V.X0)) | [Aggregation.AVG, Activation.IDENTITY]
R.h1 / 0 | [Activation.IDENTITY]
```

Module parametrized as `Pooling("h1", "h0", Aggregation.MAX, 2)` translates into:

```
(R.h1 <= R.h0(V.X0, V.X1)) | [Aggregation.MAX, Activation.IDENTITY]
R.h1 / 0 | [Activation.IDENTITY]
```

Parameters

- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input name.
- **aggregation** (*Function*) – Aggregation function.
- **input_arity** (*int*) – Arity of the input predicate `input_name`. Default: 1

class SumPooling(*output_name: str*, *input_name: str*, *input_arity: int = 1*)

Apply sum pooling over the input specified by the input_name and the input arity parameters. Can be expressed as:

$$h = \sum_{i_0,..,i_n \in N} x_{(i_0,..,i_n)}$$

Where N is a set of tuples of length n (specified by the input arity parameter) that are valid arguments for the input predicate.

This module extends the generic pooling `Pooling`.

Examples

The whole computation of this module (parametrized as `SumPooling("h1", "h0")`) is as follows:

```
(R.h1 <= R.h0(V.X0)) | [Aggregation.SUM, Activation.IDENTITY]
R.h1 / 0 | [Activation.IDENTITY]
```

Parameters

- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input name.
- **input_arity** (*int*) – Arity of the input predicate `input_name`. Default: 1

class AvgPooling(*output_name: str*, *input_name: str*, *input_arity: int = 1*)

Apply average pooling over the input specified by the input_name and the input arity parameters. Can be expressed as:

$$h = \frac{1}{|N|} \sum_{i_0,..,i_n \in N} x_{(i_0,..,i_n)}$$

Where N is a set of tuples of length n (specified by the input arity parameter) that are valid arguments for the input predicate.

This module extends the generic pooling `Pooling`.

Examples

The whole computation of this module (parametrized as `AvgPooling("h1", "h0")`) is as follows:

```
(R.h1 <= R.h0(V.X0)) | [Aggregation.AVG, Activation.IDENTITY]
R.h1 / 0 | [Activation.IDENTITY]
```

Parameters

- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input name.
- **input_arity** (*int*) – Arity of the input predicate `input_name`. Default: 1

class MaxPooling(*output_name: str*, *input_name: str*, *input_arity: int = 1*)

Apply max pooling over the input specified by the input_name and the input arity parameters. Can be expressed as:

$$h = max_{i_0,..,i_n \in N}(x_{(i_0,..,i_n)})$$

Where N is a set of tuples of length n (specified by the input arity parameter) that are valid arguments for the input predicate.

This module extends the generic pooling `Pooling`.

Examples

The whole computation of this module (parametrized as `MaxPooling("h1", "h0")`) is as follows:

```
(R.h1 <= R.h0(V.X0)) | [Aggregation.MAX, Activation.IDENTITY]
R.h1 / 0 | [Activation.IDENTITY]
```

Parameters

- **output_name** (*str*) – Output (head) predicate name of the module.
- **input_name** (*str*) – Input name.
- **input_arity** (*int*) – Arity of the input predicate `input_name`. Default: 1

7.4 Meta Modules

class MetaConv(*in_channels: int, out_channels: int, output_name: str, feature_name: str, role_name: ~typing.Optional[str], roles: ~typing.List[str], activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>, aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Metagraph Convolutional Unit layer from Meta-GNN: metagraph neural network for semi-supervised learning in attributed heterogeneous information networks. Which can be expressed as:

$$\mathbf{x}_i' = act(\mathbf{W_0} \cdot \mathbf{x}_i + agg_{j \in \mathcal{N}_r(i)} \sum_{k \in \mathcal{K}} (\mathbf{W_k} \cdot \mathbf{x}_j))$$

Where *act* is an activation function, *agg* aggregation function (by default average), W_0 is a learnable root parameter and W_k is a learnable parameter for each role.

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **role_name** (*Optional[str]*) – Role predicate name to use for role relations. When `None`, elements from `roles` are used instead.
- **roles** (*List[str]*) – List of relations' names
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.SIGMOID`
- **aggregation** (*Aggregation*) – Aggregation function of nodes' neighbors. Default: `Aggregation.AVG`

class MAGNNMean(*output_name: str, feature_name: str, relation_name: str, type_name: ~typing.Optional[str], meta_paths: ~typing.List[str], activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>, aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Intra-metapath Aggregation module with Mean encoder from "MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding". Which can be expressed as:

$$\mathbf{h}_{P(v,u)} = MEAN(\{\mathbf{x}_t | \forall t \in P(v,u)\})$$

$$\mathbf{h}_v^P = act(\sum_{u \in N_v^P} \mathbf{h}_{P(v,u)})$$

Where *act* is an activation function, $P(v,u)$ is a single metapath instance, N_v^P is set of metapath-based neighbors.

Parameters

- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **relation_name** (*str*) – Relation predicate name for connectivity checks between entities.

- **type_name** (*Optional[str]*) – Metapath type predicate name. If none, `meta_paths` will be used instead.
- **meta_paths** (*List[str]*) – Name of types forming a single metapath.
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.SIGMOID`

class MAGNNLinear(*in_channels: int, out_channels: int, output_name: str, feature_name: str, relation_name: str, type_name: ~typing.Optional[str], meta_paths: ~typing.List[str], activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>, aggregation: ~neuralogic.core.constructs.function.Aggregation = <neuralogic.core.constructs.function.Aggregation object>*)

Intra-metapath Aggregation module with Linear encoder from "MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding". Which can be expressed as:

$$\mathbf{h}_{P(v,u)} = \mathbf{W}_p \cdot MEAN(\{\mathbf{x}_t | \forall t \in P(v,u)\})$$

$$\mathbf{h}_v^P = act(\sum_{u \in N_v^P} \mathbf{h}_{P(v,u)})$$

Where *act* is an activation function, $P(v,u)$ is a single metapath instance, N_v^P is set of metapath-based neighbors.

Parameters

- **in_channels** (*int*) – Input feature size.
- **out_channels** (*int*) – Output feature size.
- **output_name** (*str*) – Output (head) predicate name of the module.
- **feature_name** (*str*) – Feature predicate name to get features from.
- **relation_name** (*str*) – Relation predicate name for connectivity checks between entities.
- **type_name** (*Optional[str]*) – Metapath type predicate name. If none, `meta_paths` will be used instead.
- **meta_paths** (*List[str]*) – Name of types forming a single metapath.
- **activation** (*Activation*) – Activation function of the output. Default: `Activation.SIGMOID`

CHAPTER

EIGHT

ADVANCED USAGE

8.1 Heterogeneous Graphs

Most GNN models consider graphs to be homogeneous - that is, all nodes being of one type, despite many possible instances of problems, where it would be beneficial to utilize information about entities' types and their relations' types. In PyNeuraLogic, we can easily encode instances of such heterogeneous graphs with an arbitrary number of nodes' and edges' classes out of the box.

Let's consider the graph in the above figure, where nodes' color represents their type (either *Blue* or *Pink*). We can represent introduced nodes' types in the input examples, for example, as the following list of ground relations:

```
Relation.type(1, Term.BLUE),
Relation.type(2, Term.BLUE),
Relation.type(3, Term.PINK),
Relation.type(4, Term.PINK),
Relation.type(5, Term.PINK),
```

Note: Note that there are many ways to express the same concept. We could, for example, encode types (for nodes 1 and 3) as `Relation.blue(1), Relation.pink(3)`. Nodes are also not limited to have only one type; we can assign multiple types to one node, such as `Relation.type(1, Term.BLUE), Relation.type(1, Term.PINK)`.

Note: In this example, we consider only nodes' types, but we can analogically encode edges' (or any other) types.

We can then utilize the information about types for various use cases. In the following example, we showcase a template rule for the aggregation of neighbor nodes of the central node with the same type as the central node.

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

Since types are just regular constructs (relations) in PyNeuraLogic, we are able to manipulate them as anything else. We can, for example, create hierarchies of types or, as is shown in the following example, attach features to types in input examples and then utilize them in the aggregation.

```
Relation.type_feature(Term.BLUE)[[1, 2, 3]]
```

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

8.2 Utilizing Inference Engine

While translating logic programs into computations graphs, PyNeuraLogic utilizes an inference engine. The inference engine serves for deducing information from the input knowledge base encoded in examples or a template. For convenience, this functionality is also exposed via a high-level interface to be accessible for users.

8.2.1 London Underground Example

The interface for the inference engine is relatively simple. Consider the following example based on the "Simply Logical: Intelligent Reasoning by Example" book by Peter Flach. We have a network based on a part of the London Underground encoded as a directed graph as visualized in the following image.

This graph can be encoded as `connected(From, To, Line)` such as:

```
from neuralogic.core import Template, R, V, T


template = Template()
template += [
    R.connected(T.bond_street, T.oxford_circus, T.central),
    R.connected(T.oxford_circus, T.tottenham_court_road, T.central),
    R.connected(T.bond_street, T.green_park, T.jubilee),
    R.connected(T.green_park, T.charing_cross, T.jubilee),
    R.connected(T.green_park, T.piccadilly_circus, T.piccadilly),
    R.connected(T.piccadilly_circus, T.leicester_square, T.piccadilly),
    R.connected(T.green_park, T.oxford_circus, T.victoria),
    R.connected(T.oxford_circus, T.piccadilly_circus, T.bakerloo),
```

(continues on next page)

(continued from previous page)

```
    R.connected(T.piccadilly_circus, T.charing_cross, T.bakerloo),
    R.connected(T.tottenham_court_road, T.leicester_square, T.northern),
    R.connected(T.leicester_square, T.charing_cross, T.northern),
]
```

This template essentially encodes only direct connections between stations (nodes). We might want to extend this knowledge by deducing which stations are nearby - stations with at most one station between them.

So stations are nearby if they are directly connected, which can be expressed as:

```
template += R.nearby(V.X, V.Y) <= R.connected(V.X, V.Y, V.L)
```

Stations are also nearby if exactly one station lays on the path between those two stations and are on the same line.

```
template += R.nearby(V.X, V.Y) <= (R.connected(V.X, V.Z, V.L), R.connected(V.Z, V.Y, V.
↪L))
```

Now we can ask the inference engine to get all sorts of different information, such as what stations are nearby the Tottenham Court Road station.

```
from neuralogic.inference.inference_engine import InferenceEngine


engine = InferenceEngine(template)

engine.q(R.nearby(T.tottenham_court_road, V.X))
```

Running the `query` (or `q`) will return a generator of dictionaries with all possible substitutions for all variables in the query. In this case, we have only one variable in the query (`V.X`). As you can see, the inference engine found all stations that are nearby the Tottenham Court Road station (Leicester Square and Charing Cross).

```
[
    {"X": "leicester_square"},
    {"X": "charing_cross"},
]
```

We could also ask the inference engine to get all possible nearby stations (`R.nearby(V.X, V.Y)`) and so on.

Finding Path Recursively

We can also define another rule to check a generic path from a station `X` to another station `Y`. We will call this rule `reachable` and use recursion in its definition. The `reachable` rule is satisfied if two stations are directly connected or station `X` is connected to station `Z` from which you can reach `Y`.

```
template += R.reachable(V.X, V.Y) <= R.connected(V.X, V.Y, V.L)
template += R.reachable(V.X, V.Y) <= (R.connected(V.X, V.Z, V.L), R.reachable(V.Z, V.Y))
```

Now we can ask the inference engine what stations we can reach from a station or ask more exact queries such as if two specific stations are reachable.

```
engine = InferenceEngine(template)

if engine.query(R.reachable(T.green_park, T.tottenham_court_road)):
```

(continues on next page)

(continued from previous page)

```
    print("Yes, you can reach Tottenham Court Road from Green Park")
else:
    print("Those two stations are reachable, so this should never be printed out")
```

Changing the Knowledge Base

There might be cases where we want to reuse defined rules on the different knowledge bases (e.g., on different cities' underground systems) or extend the knowledge base for some queries (e.g., add additional routes).

We can extend the current knowledge defined in the template using the `set_knowledge` method.

```
engine.set_knowledge(additional_knowledge)
```

We can also set a knowledge that will extend the knowledge base defined in the template but will ignore the knowledge set by the `set_knowledge` method. This knowledge base will be considered only for the context of the query.

```
engine.query(R.some_query, additional_knowledge)
```

8.3 Fuzzy Relational Inference Engine

In the *Utilizing Inference Engine* section, we introduced a high-level interface for the underlying inference engine that does only minimal work to provide more performance (e.g., it does not construct neural networks). To complement this type of inference engine, PyNeuraLogic also provides an evaluation inference engine that, on top of finding all valid substitutions, runs an evaluation of the provided logic program.

8.3.1 Finding the Shortest Path

As an example of a possible use case of the evaluation inference engine, we will take a look at the example from *Utilizing Inference Engine* but with a slight twist - we introduce weights to connections, representing either distance from stations in time or some unit of length.

The encoding is almost the same, except for added values to each connection, that is `connected(From, To, Line)[Distance]`.

```
from neuralogic.core import Template, R, V, T, Metadata, Aggregation, Activation,␣
↪ActivationAgg
from neuralogic.inference.evaluation_inference_engine import EvaluationInferenceEngine
```

(continues on next page)

(continued from previous page)

```
template = Template()
template += [
    R.connected(T.bond_street, T.oxford_circus, T.central)[7],
    R.connected(T.oxford_circus, T.tottenham_court_road, T.central)[9],
    R.connected(T.bond_street, T.green_park, T.jubilee)[14],
    R.connected(T.green_park, T.charing_cross, T.jubilee)[21],
    R.connected(T.green_park, T.piccadilly_circus, T.piccadilly)[8],
    R.connected(T.piccadilly_circus, T.leicester_square, T.piccadilly)[6],
    R.connected(T.green_park, T.oxford_circus, T.victoria)[15],
    R.connected(T.oxford_circus, T.piccadilly_circus, T.bakerloo)[12],
    R.connected(T.piccadilly_circus, T.charing_cross, T.bakerloo)[11],
    R.connected(T.tottenham_court_road, T.leicester_square, T.northern)[8],
    R.connected(T.leicester_square, T.charing_cross, T.northern)[7],
]
```

We have defined two rules called `shortest_path`. The first rule aggregates connected stations and takes all connections' maximum value (distance). The second rule handles instances when stations are not directly connected - at least one station has to be traversed to get to the goal station. The second rule aggregates all possible instances and finds maximum value while "calling" one of the two rules recursively.

```
metadata = Metadata(aggregation=Aggregation.MIN, activation=Activation.IDENTITY)

template += (R.shortest(V.X, V.Y) <= R.connected(V.X, V.Y, V.L)) | metadata
template += (R.shortest(V.X, V.Y) <= (R.connected(V.X, V.Z, V.L), R.shortest_path(V.Z, V.
↪Y))) | metadata
```

Attention: Notice we are appending metadata with aggregation (Min) and activation (Identity) functions.

It is also necessary to set additional activation functions to identity.

```
template += R.shortest_path / 2 | Metadata(activation=ActivationAgg.MIN + Activation.
↪IDENTITY)
template += R.connected / 3 | Metadata(activation=Activation.IDENTITY)
```

Evaluating Queries

Now when the template and the knowledge base are ready, we can run queries the same way as for the previously introduced instance of `InferenceEngine`. The only difference in the interface for `EvaluationInferenceEngine` are returned values from the generator - instead of returning generator of dictionaries containing substitutions, `EvaluationInferenceEngine` returns a generator of tuple containing the output of evaluation and the dictionary of substitutions.

We can, for example, get the shortest path from the Bond Street station to the Charing Cross station.

```
engine = EvaluationInferenceEngine(template)

result = engine.q(R.shortest_path(T.bond_street, T.charing_cross)

print(list(result))
```

```
[
    (30.0, {})
]
```

The query computed the distance to be 30 units, which is the actual shortest distance for this input. But this query does not bring any additional value compared to evaluation via evaluators or directly on the model.

To fully utilize the fuzzy relational inference engine, we would also want to get some substitutions. For example, we can get the shortest distances from the Green Park station to all reachable stations.

```
result = engine.q(R.shortest_path(T.green_park, V.X))

print(list(result))
```

```
[
    (19.0, {'X': 'charing_cross'}),
    (14.0, {'X': 'leicester_square'}),
    (8.0, {'X': 'piccadilly_circus'}),
    (15.0, {'X': 'oxford_circus'}),
    (24.0, {'X': 'tottenham_court_road'})
]
```

This output then tells us that the shortest path to the Charing Cross station from the Green Park station is 19 units long, to the Leicester Square station it is 14 units long, and so on.

8.4 Modifiers

Modifiers are optional and alter an relations' behavior in some way. Currently, there are two following modifiers, which can be chained together:

8.4.1 Hidden Modifier

Sometimes, there are relations in rules that only define the logic structure and are not beneficial to be included in the computation graph. For those cases, there is a hidden modifier that enforces exactly that - includes relation for the logic part and excludes relation in the resulting computation graph.

For example, consider the following rule. In some instances, it might be counterproductive to include the `edge` relations in the resulting computation graph (e.g., they might not have any edge features), yet those `edge` relations cannot be removed as they define a critical part of the logic structure of the program. Including them in the computation graph will produce a side effect - offsetting the result of relations `h`.

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

This issue can be solved by flagging the predicate `edge` as `hidden`, ensuring that relations with such a predicate will not be included in the computation graph.

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

# can be written also as (prepended _ makes predicate hidden)

Relation.h(Var.X) <= (Relation.feature(Var.Y), Relation._edge(Var.X, Var.Y))
```

8.4.2 Special Modifier

The special modifier changes the relation's behavior depending on its predicate name. We can utilize the following special predicates:

- **`Relation.special.alldiff`**
  A special relation with the `alldiff` predicate ensures that its terms (logic variables) are substituted for different values (unique values). It's also possible to use `...` in place of terms, which is substituted for all variables declared in the current rule - no variable declared in the rule can be substituted for the same value simultaneously.

```
Relation.special.alldiff(Var.X, Var.Y)  # Var.X cannot equal to Var.Y

# Var.X != Var.Y != Var.Z
Relation.h(Var.X) <= (Relation.b(Var.Y, Var.Z), Relation.special.alldiff(...))
```

- `Relation.special.anypred`
- `Relation.special.in`
- `Relation.special.maxcard`
- `Relation.special.true`
- `Relation.special.false`
- `Relation.special.neq`
- `Relation.special.leq`
- `Relation.special.geq`
- `Relation.special.lt`
- `Relation.special.gt`
- `Relation.special.eq`

8.5 Visualization

You can run this page in Jupyter Notebook

PyNeuraLogic offers multiple options for visualization of templates and samples, which can be helpful while investigating how the high-level rule representations are being translated into computation graphs. The usage of visualization tools requires having installed Graphviz.

Depending on the parametrization, the drawing methods can output either graph image in bytes, graph image rendered into a file, or graph image displayed into IPython (Jupyter Notebook).

Additionally, it is also possible to retrieve the generated source of graphs in the DOT format. This format can then be used to display or further customize and manipulate generated graphs in other libraries.

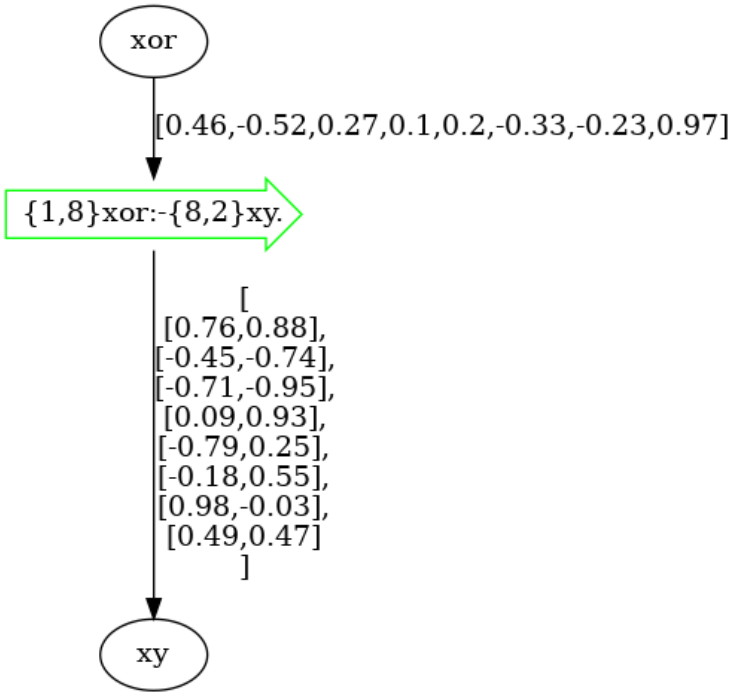
8.5.1 Visualization of the XOR Example

To showcase the usage of visualization tools, we will use the template and the dataset introduced in XOR Example

Model Rendering

All that is needed to visualize the model - the template with current weights' values is to call the `draw` method.

```
from neuralogic.utils.data import XOR_Vectorized
from neuralogic.core import Settings, Backend


template, dataset = XOR_Vectorized()
model = template.build(Backend.JAVA, Settings())

model.draw()
```



Tip: If you are using evaluators, you can draw the model via the `evaluator.draw` method.

Tip: You can also visualize the template by calling the `template.draw` method.

Templates (models) and samples can be drawn into various raster formats (such as PNG or JPEG) or SVG format, which is considerably faster for larger graphs. To set the format, simply use the `img_type` parameter.

The drawing can be further parameterized, for example, with the `value_detail` parameter to display more (or less) decimal places of all values (there are three levels of detail - `0-2`, where `0` has the least number of decimals and `2` the most number of decimals).

The model above was directly drawn into Jupyter Notebook without any parametrization. To draw the model into a file, all we have to do is add the `filename` parameter with a path to the output image, such as:

```
model.draw(filename="my_image.png")
```

We can also get raw images bytes by turning off displaying into IPython:

```
model.draw(draw_ipython=False)
```

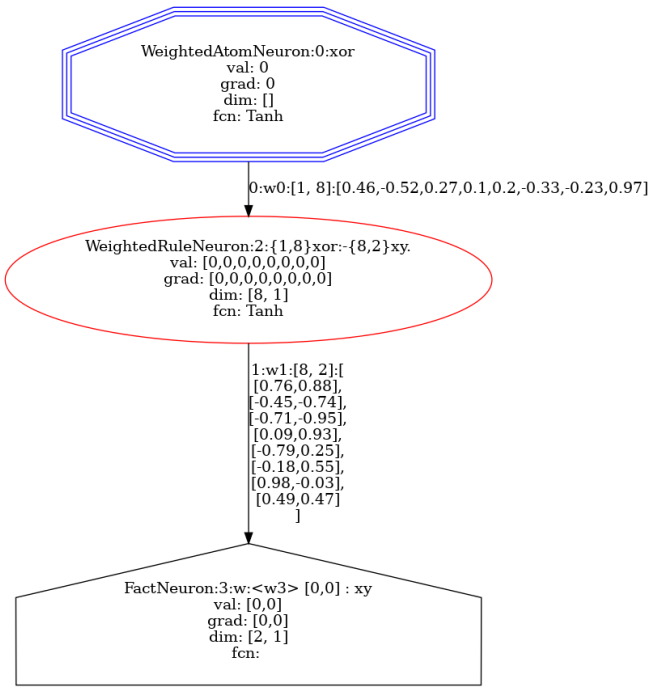
Tip: If you are drawing straight into Jupyter Notebook, you can include additional parameters into drawing functions to customize the underlying Image and SVG objects.

Samples Rendering

Samples can be drawn in the same way and supports the same parametrization as the model drawing.

An example of drawing samples can be seen in the code below, where we render the actual computation graph for the first example (input `[0, 0]`).

```
built_dataset = model.build_dataset(dataset)

built_dataset.samples[0].draw()
```



Getting the DOT Source

To get the DOT source of the model or the sample, all you have to do is call the `model_to_dot_source` function or the `sample_to_dot_source` function, respectively.

```
from neuralogic.utils.visualize import sample_to_dot_source


dot_source = sample_to_dot_source(built_dataset.samples[0])
print(dot_source)
```

```
digraph G {
3 [shape=house, color=black, label="FactNeuron:3:w:<w3> [0,0] : xy
val: [0,0]
grad: [0,0]
dim: [2, 1]
fcn:
"]

2 [shape=ellipse, color=red, label="WeightedRuleNeuron:2:{1,8}xor:-{8,2}xy.
val: [0,0,0,0,0,0,0,0]
grad: [0,0,0,0,0,0,0,0]
dim: [8, 1]
fcn: Tanh
"]
2 -> 3 [label="1:w1:[8, 2]:[
[0.76,0.88],
[-0.45,-0.74],
[-0.71,-0.95],
[0.09,0.93],
[-0.79,0.25],
[-0.18,0.55],
[0.98,-0.03],
[0.49,0.47]
]"]

0 [shape=ellipse, color=blue, label="WeightedAtomNeuron:0:xor
val: 0
grad: 0
dim: []
fcn: Tanh
"]
0 -> 2 [label="0:w0:[1, 8]:[0.46,-0.52,0.27,0.1,0.2,-0.33,-0.23,0.97]"]

0 [shape = tripleoctagon]
}
```

8.6 Recursive XOR Generalization

You can run this page in Jupyter Notebook

In one of our introductory examples we have showcased how to learn the XOR operation for two inputs. In this example, we will generalize the learning of the XOR operation to N inputs while making the use of recursion.

We will define a recursive template, train it on the classic XOR (two inputs) and show an inference of inputs of different lengths.

The template will essentially evaluate $xor_n = xor(val_n, xor_{n-1})$ with shared weights across all depths.

```
from neuralogic.nn import get_evaluator
from neuralogic.core import Settings, R, V, Template, Activation
from neuralogic.dataset import Dataset
```

Before we define rules for the actual learning, we introduce helper relations (facts) `R._next`. Those rules serve for the definition of the sequence of integers, that is $1, 2, ..., N$ (N is defined by `max_number_of_vars`). We have to do that because later on, we will utilize this sequence for the recursion. Integers in PyNeuraLogic are independent entities with no extra meaning or context.

```
max_number_of_vars = 5

template = Template()
template += (R._next(i, i + 1) for i in range(max_number_of_vars))
```

We then define the base case of the recursion, that is, to get the value of xor of length 1 (index 0) return the value of the first (index 0) element.

```
template += R.xor_at(0) <= R.val_at(0)
```

Now when we have the base case ready, we introduce the recursive "calls". The following rule can be interpreted as "To calculate the xor of length N (`V.Y`), calculate the xor of length $N - 1$ (`V.X`) and pipe the result together with the element at index N (`R.val_at(V.Y)`) into xor".

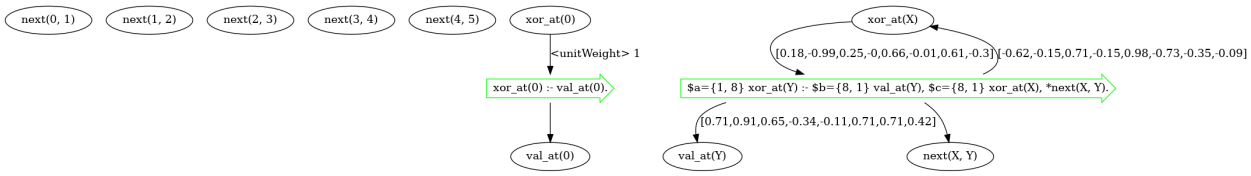
We also assigned three unique vector learnable parameters and named them. Naming is entirely optional and is here only to show the mapping later.

```
template += R.xor_at(V.Y)["a": 1, 8] <= (R.val_at(V.Y)["b": 8, 1], R.xor_at(V.X)["c": 8,␣
↪1], R._next(V.X, V.Y))
```

And that is everything you need to define a template for recursive generalization of XOR!

The recursion, together with weights mapping, can be viewed in the template graph by drawing it.

```
template.draw()
```



The definition of the training data set is straightforward; we train the model on inputs of the length of two. That is, we encode $xor(0, 0) = 0$, $xor(1, 0) = 1$, and so on as the training set.

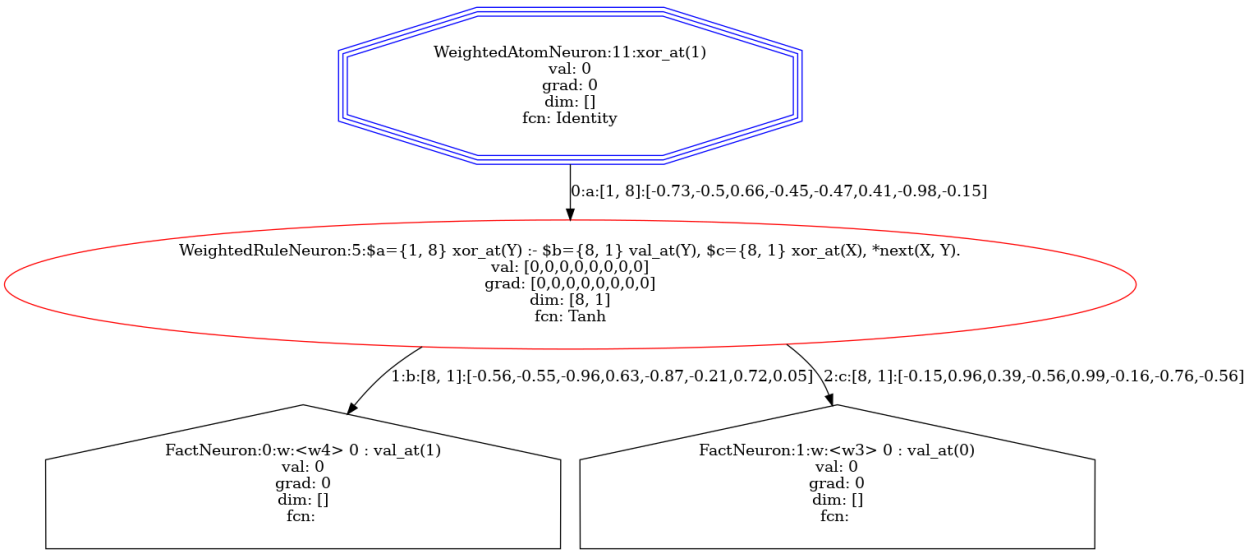
```
examples = [
    [R.val_at(0)[0], R.val_at(1)[0]],  # input: 0, 0
    [R.val_at(0)[0], R.val_at(1)[1]],  # input: 0, 1
    [R.val_at(0)[1], R.val_at(1)[0]],  # input: 1, 0
    [R.val_at(0)[1], R.val_at(1)[1]],  # input: 1, 1
]

queries = [  # outputs: 0, 1, 1, and 0
    R.xor_at(1)[0], R.xor_at(1)[1], R.xor_at(1)[1], R.xor_at(1)[0],
]

train_dataset = Dataset(examples, queries)
```

```
settings = Settings(
    epochs=5000, rule_activation=Activation.TANH, relation_activation=Activation.
↪IDENTITY, iso_value_compression=False
)

evaluator = get_evaluator(template, settings)
built_dataset = evaluator.build_dataset(train_dataset)
```

Note: Notice we turned off compression, so the recursion is clearly visible in the visual representation later on.

Once we build the training dataset, we can visualize each sample. For example, the $xor(0, 0)$ sample will be represented by the following computation graph.

```
built_dataset.samples[0].draw()
```




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

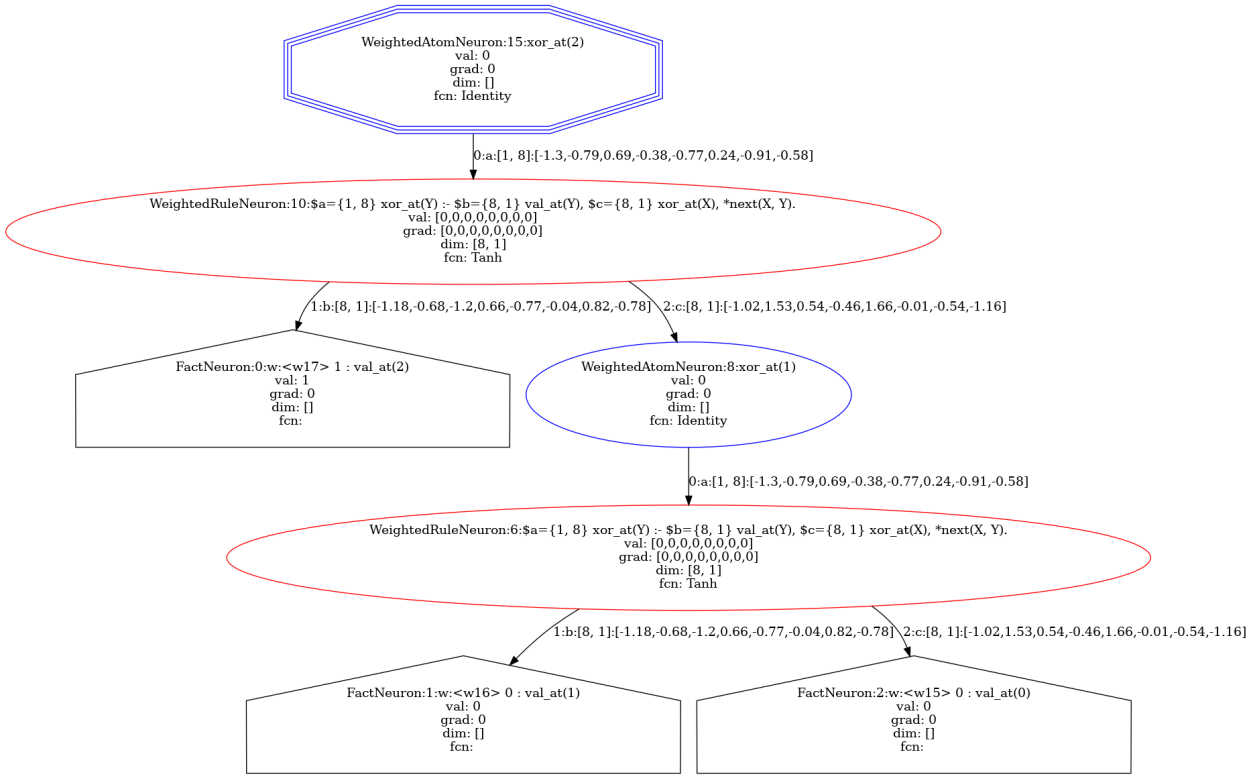
We train the model on the training dataset via the evaluator and then prepare a test dataset. We can put any input of maxi-

mum length of N (`max_number_of_vars`) into the dataset. For this example, we chose $xor(0, 0, 1)$ and $xor(1, 0, 1, 0)$. Feel free to try out other lengths and combinations!

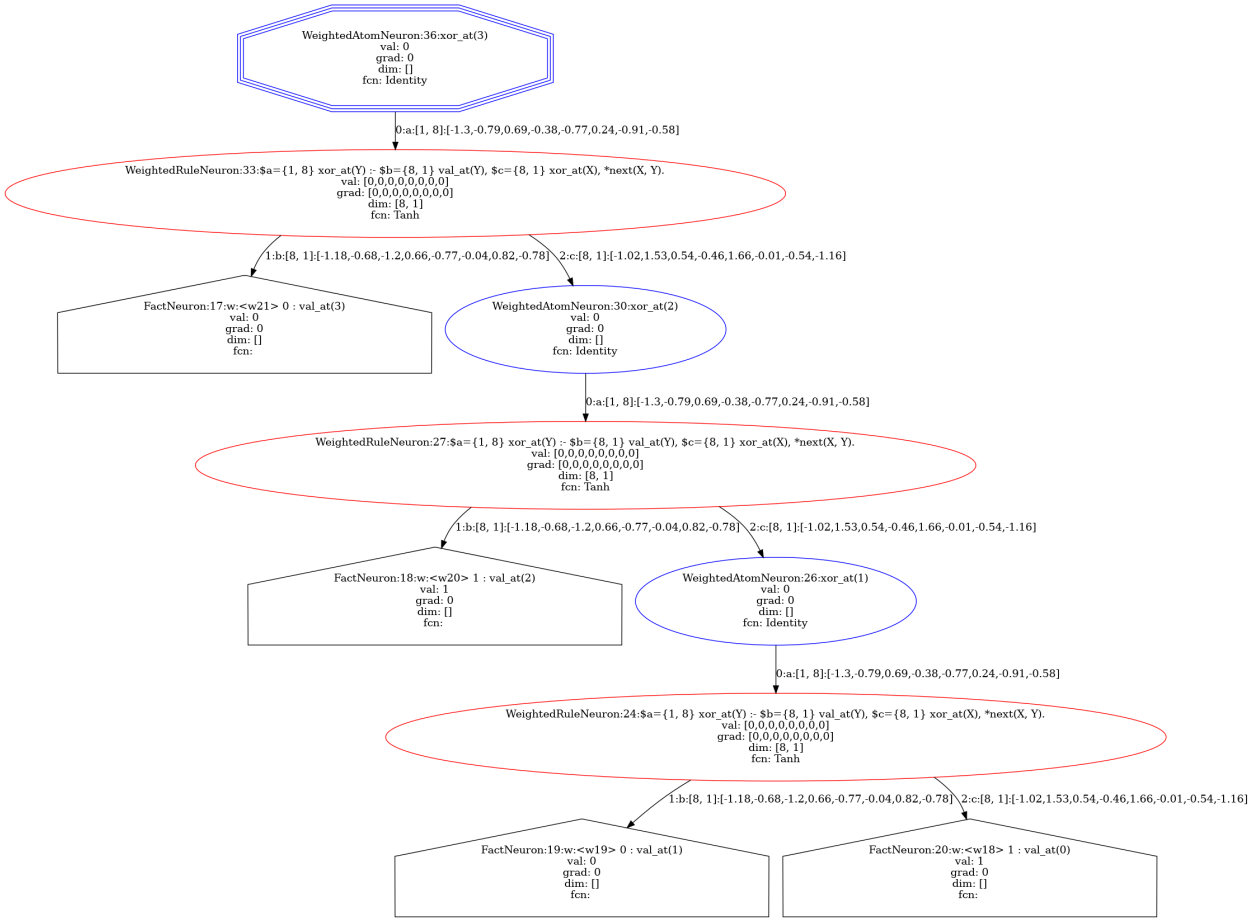
```
test_examples = [
    [R.val_at(0)[0], R.val_at(1)[0], R.val_at(2)[1]],
    [R.val_at(0)[1], R.val_at(1)[0], R.val_at(2)[1], R.val_at(3)[0]],
]

test_queries = [
    R.xor_at(2), R.xor_at(3)
]

test_dataset = Dataset(test_examples, test_queries)
built_test_dataset = evaluator.build_dataset(test_dataset)
```

When we visualize our test samples and compare them, we can clearly see how the template is recursively unrolled into computation graphs (trees) with shared weights across depths.

```
built_test_dataset.samples[0].draw()
```




```
built_test_dataset.samples[1].draw()
```



Running inference on our test dataset yields correct results, that is $xor(0, 0, 1) = 1$ and $xor(1, 0, 1, 0) = 0$.

```
for _, result in evaluator.test(built_test_dataset):
    print(result)  # 1, 0
```

8.7 Java Settings, Logging and Debugging

PyNeuraLogic, at its core, utilizes procedures (such as grounding) running on a Java Virtual Machine (JVM). JVM itself offers plentiful options to set, such as memory limitations, garbage collectors settings, and more.

This section will go through interfaces that allow you to pass your own JVM settings. We will also look into JVM logging and JVM debugging.

8.7.1 JVM Settings

Important: Customizing JVM settings and JVM path is applicable only before a JVM is started. If you want to do some customizations, do them before working with PyNeuraLogic (building model/building samples, etc.)

By default, PyNeuraLogic uses JVM found on your `PATH`. If you want to use a different JVM, you can do that by calling the `neuralogic.set_jvm_path` function, such as:

```
import neuralogic

neuralogic.set_jvm_path("/some/path/my_jvm/")
```

You can also make some adjustments to JVM settings via the `neuralogic.set_jvm_options` function. By default, two options are passed into the JVM - `"-Xms1g"`, which sets the minimum amount of heap memory size to 1 GB, and `"-Xmx64g"`, which sets the maximum amount of heap memory size to 64 GB.

This function overrides already set options, so if you want to keep defaults or previously set options,you will have to specify them again. For example, you can inspect the garbage collector with customizing settings such as:

```
import neuralogic

neuralogic.set_jvm_options(["-Xms1g", "-Xmx64g", "-XX:+PrintGCDetails"])
```

8.7.2 Java Logging

Looking into the Java logs can be valuable practice to get better insight into what is going on in the background. It offers a lot of information about all steps, such as the grounding process. This info is also practical when asking for help in discussion/issues.

You can add a logging handler anytime you want with any level by calling the `add_handler` function. The first argument can be any object that implements a `write(message: str)` method (e.g., file handlers, `sys.stdout`, etc.).

```
import sys
from neuralogic.logging import add_handler, Formatter, Level

add_handler(sys.stdout, Level.FINE, Formatter.COLOR)
```

If you decide you no longer want to subscribe to loggers, you can remove all logging handlers by calling the `clear_handlers` function.

```
from neuralogic.logging import clear_handlers

clear_handlers()
```

8.7.3 Java Debugging

Important: To run PyNeuraLogic in debug mode, you have to run the debug mode before a JVM is started - therefore, run the debug mode before working with PyNeuraLogic (building model/building samples, etc.)

There is a prepared interface to run the JVM in the debug mode, which allows to attach a remote debugger on the JVM and then use breakpoints on the NeuraLogic project, as usual. You can enable the debug mode by calling the `neuralogic.initialize` function with the argument `debug_mode=True`.

```
import neuralogic

neuralogic.initialize(debug_mode=True)
```

```
>>> Listening for transport dt_socket at address: 12999
```

Once you get the message above, the execution of the python program will wait (by default) for you to connect your remote debugger to the port (by default, *12999*). Via other arguments of the initialize function, it is possible to specify further things like debugging port, etc.

Once the remote debugger is attached, the execution of the Python program will continue until the execution hits a breakpoint.

- *Heterogeneous Graphs*
  Learn how to represent heterogeneous graphs and possible ways to incorporate rules utilizing them into your templates.
- *Utilizing Inference Engine*
  PyNeuraLogic offers to utilize its engine only for inference as well. This section goes through an example to showcase the usage of the inference engine, to get all possible substitutions satisfying our queries.
- *Fuzzy Relational Inference Engine*
  You can also extend the inference engine from the previous section and utilize numeric relations' values. For example, to compute the shortest paths between points as in this example!
- *Modifiers*
  Some relations can have special meanings and functionalities. You can find out more about them here.
- *Visualization*
  Having a visual representation of your model can help you get a better insight. Learn how to utilize prepared tools to visualize your models/templates and samples.
- *Recursive XOR Generalization*
  Learn how recursive templates can be defined and utilized!
- *Java Settings, Logging and Debugging*
  In this section, we go through all the different settings of the backend engine, such as using its logging, debugging, passing additional JVM arguments, etc.

CHAPTER

NINE

EXAMPLES

- Molecular GNNs
- Simple XOR example
- Recursive XOR Generalization
- Visualization
- Pattern Matching
- Distinguishing k-regular graphs
- Distinguishing non-regular graphs

CHAPTER

TEN

BENCHMARKS

Here we compare the speed of some popular GNN models encoded in PyNeuraLogic against some of the most popular GNN frameworks in their latest versions, namely (2.0.2), (0.6.1), and (1.0.6).

The benchmarks report comparison of the average training time per epoch of three different architectures - GCN (two GCNConv layers), GraphSAGE (two GraphSAGEConv layers), and GIN (five GINConv layers).

Datasets are picked from the common and are loaded into PyNeuraLogic, DGL, and PyG via PyG's . Spektral benchmark uses Spektral's .

We compare the frameworks in a binary graph classification task with only node's features. This is merely for the sake of simple reusability of the introduced architectures over the frameworks. Statistics of each dataset can be seen down below.

Due to its declarative nature, PyNeuraLogic has to transform each dataset into a logic form and then into a computation graph. The time spent on this preprocessing task is labeled as "Dataset Build Time". Note that this transformation happens only once before the training.

MUTAG

NCI1

PROTEINS

BZR

COX2

DHFR

KKI

Peking_1

	GCN	GraphSAGE	GIN
Spektral	0.1238s	0.1547s	0.2491s
Deep Graph Library	0.1287s	0.1795s	0.5214s
PyTorch Geometric	0.0897s	0.1099s	0.3399s
PyNeuraLogic	**0.0083s**	**0.0119s**	**0.0393s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	1.4265s	1.9372s	2.3662s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
188	~17.9	~19.7	7

	GCN	GraphSAGE	GIN
Spektral	3.0152s	3.1773s	5.1924s
Deep Graph Library	3.1044s	4.3426s	11.3512s
PyTorch Geometric	1.9226s	2.6211s	7.0598s
PyNeuraLogic	**0.2396s**	**0.3461s**	**1.5037s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	24.8405s	25.2125s	57.4115s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
4110	~29.8	~32.3	37

	GCN	GraphSAGE	GIN
Spektral	0.7221s	1.0153s	1.4591s
Deep Graph Library	0.7859s	1.1963s	3.1576s
PyTorch Geometric	0.5047s	0.6455s	1.9786s
PyNeuraLogic	**0.0741s**	**0.1111s**	**0.5524s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	9.9873s	10.0125s	24.2591s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
1113	~39.0	~72.8	3

	GCN	GraphSAGE	GIN
Spektral	0.2730s	0.3238s	0.5144s
Deep Graph Library	0.3035s	0.4288s	1.1171s
PyTorch Geometric	0.1847s	0.2464s	0.7232s
PyNeuraLogic	**0.0293s**	**0.0469s**	**0.1552s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	3.8219s	3.9852s	7.0831s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
405	~35.7	~38.3	53

	GCN	GraphSAGE	GIN
Spektral	0.3411s	0.3705s	0.5975s
Deep Graph Library	0.3513s	0.5124s	1.2988s
PyTorch Geometric	0.2082s	0.2857s	0.8086s
PyNeuraLogic	**0.0321s**	**0.0505s**	**0.1754s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	4.2805s	4.5738s	8.6356s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
467	~41.2	~43.4	35

	GCN	GraphSAGE	GIN
Spektral	0.5578s	0.6058s	0.9708s
Deep Graph Library	0.6063s	0.8010s	2.1136s
PyTorch Geometric	0.3388s	0.4588s	1.3178s
PyNeuraLogic	**0.0572s**	**0.0879s**	**0.3168s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	7.3361s	7.3635s	15.0887s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
467	~42.4	~44.5	53

	GCN	GraphSAGE	GIN
Spektral	0.0565s	0.0797s	0.1200s
Deep Graph Library	0.0611s	0.0887s	0.2292s
PyTorch Geometric	0.0370s	0.0535s	0.1480s
PyNeuraLogic	**0.0262s**	**0.0321s**	**0.0529s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	1.7563s	2.0459s	2.6008s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
83	~26.9	~48.4	190

	GCN	GraphSAGE	GIN
Spektral	0.0597s	0.0851s	0.1244s
Deep Graph Library	0.0654s	0.0923s	0.2335s
PyTorch Geometric	0.0404s	0.0608s	0.1547s
PyNeuraLogic	**0.0371s**	**0.0469s**	**0.0778s**

	GCN	GraphSAGE	GIN
PyNeuraLogic	2.3414s	2.2352s	3.3951s

Num. of Graphs	Avg. num. of nodes	Avg. num. of edges	Num. node of features
85	~39.3	~77.3	190

CHAPTER

ELEVEN

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.

11.1 Representation of hyperedges

When we encode input data (graph) in the form of logic data format (i.e., ground relations), we can represent regular edges, for example, as `Relation.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 $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{\{1, 2\}, \{3, 4, 5\}, \{1, 2, 4, 6\}\}$ can be represented as:

```
Relation.edge(1, 2),
Relation.edge(3, 4, 5),
Relation.edge(1, 2, 4, 6),
```

11.2 Propagation on hyperedges

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

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

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

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

CHAPTER

TWELVE

HETEROPHILY SETTINGS

Regular GNN models usually consider homophily in the graph - frequently, nodes of similar classes are connected with each other. 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.

There have been proposed new methods and models to properly capture problems of such settings, such as *CPGNN* ("Graph Neural Networks with Heterophily") or *H2GCN* ("Beyond Homophily in Graph Neural Networks: Current Limitations and Effective Designs").

We take into consideration the latter one - the *H2GCN* model, which is specifically built to deal with heterophily graphs and implement three key design concepts. All of those concepts can be easily represented in PyNeuraLogic with a few rules. As in other cases, the rule representation can be further manipulated and tweaked without the need of reimplementing the whole model or digging into an already implemented black box.

12.1 1. The Central Node Embedding Separation

The first key design is separating the embedding of the central node from the embedding of neighbor nodes. This behavior can be achieved just with two rules, that can be written in the following form:

```
Relation.layer_1(Var.X) <= (Relation.layer_0(Var.Y), Relation.edge(Var.Y, Var.X)),
Relation.layer_1(Var.X) <= Relation.layer_0(Var.X)
```

The first rule aggregates all features of neighbors of the central node, and then we combine the aggregated value with the value of the second rule, which embeds the features of the central node.

12.2 2. Higher-Order Neighborhoods Embedding

The second concept is to consider not only the direct neighbors but also higher-order neighbors in the computation of the central node's representation, such as second-order neighbors (neighbors of neighbors), as can be represented as the following rule:

```
Relation.layer_1(Var.X) <= (
    Relation.layer_0(Var.Z),
    Relation.edge(Var.Y, Var.X),
    Relation.edge(Var.Z, Var.Y),
    Relation.special.alldiff(...),
)
```

See also:

For more information about the special predicate `alldiff`, see *Special Modifier*.

12.3 3. Combination of Intermediate Representations

The last design concept used in the *H2GCN* model is the combination of intermediate representation. This can also be easily achieved in PyNeuraLogic just by one rule, where we combine all representations of layers, such as:

```
Relation.layer_final(Var.X) <= (
    Relation.layer_0(Var.X),
    Relation.layer_1(Var.X),
    Relation.layer_2(Var.X),
    Relation.layer_n(Var.X),
)
```

CHAPTER

THIRTEEN

NEURALOGIC PACKAGE

13.1 Subpackages

13.1.1 neuralogic.core package

Subpackages

neuralogic.core.builder package

Submodules

neuralogic.core.builder.builder module

class Builder(*settings:* SettingsProxy)

Bases: `object`

static build(*samples*)

build_model(*parsed_template*, *backend:* Backend, *settings:* SettingsProxy)

build_template_from_file(*settings:* SettingsProxy, *filename: str*)

from_logic_samples(*parsed_template*, *logic_samples*, *backend:* Backend)

from_sources(*parsed_template*, *sources:* Sources, *backend:* Backend)

static get_builders(*settings:* SettingsProxy)

stream_to_list(*stream*) → List

neuralogic.core.builder.components module

class BuiltDataset(*samples*)

Bases: `object`

BuiltDataset represents an already built dataset - that is, a dataset that has been grounded and neuralized.

class Neuron(*neuron: Dict[str, Any]*, *index*)

Bases: `object`

static parse_hook_name(*name: str*)

class RawSample(*sample*)

Bases: object

draw(*filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*, **args*, ***kwargs*)

java_sample

class Sample(*sample*, *java_sample*)

Bases: *RawSample*

static deserialize_network(*network*)

id

java_sample

neurons

output_neuron

target

class Weight(*weight*)

Bases: object

static get_unit_weight() → *Weight*

neuralogic.core.builder.dataset_builder module

class DatasetBuilder(*parsed_template*, *java_factory:* JavaFactory)

Bases: object

build_dataset(*dataset: BaseDataset*, *backend:* Backend, *settings:* SettingsProxy, *file_mode: bool = False*) → *BuiltDataset*

Builds the dataset (does grounding and neuralization) for this template instance and the backend

Parameters

- **dataset** –
- **backend** –
- **settings** –
- **file_mode** –

Returns

build_examples(*examples*, *examples_builder*)

build_queries(*queries*, *query_builder*)

static merge_queries_with_examples(*queries*, *examples*, *one_query_per_example*, *example_queries=True*)

Module contents

neuralogic.core.constructs package

Submodules

neuralogic.core.constructs.factories module

class AtomFactory

Bases: object

class Predicate(*hidden=False*, *special=False*)

Bases: object

static get_predicate(*name*, *arity*, *hidden*, *special*) → *Predicate*

property hidden: *Predicate*

property special: *Predicate*

get(*name: str*) → BaseRelation

class ConstantFactory

Bases: object

class VariableFactory

Bases: object

neuralogic.core.constructs.java_objects module

class JavaFactory(*settings: Optional[SettingsProxy] = None*)

Bases: object

atom_to_clause(*atom*)

get_conjunction(*relations*, *variable_factory*, *default_weight=None*, *is_example=False*)

get_generic_relation(*relation_class*, *relation*, *variable_factory*, *default_weight=None*, *is_example=False*)

get_lifted_example(*example*)

get_metadata(*metadata*, *metadata_class*)

get_new_weight_factory()

get_predicate(*predicate*)

get_predicate_metadata_pair(*predicate_metadata*)

get_query(*query*)

get_relation(*relation*, *variable_factory*, *is_example=False*)

get_rule(*rule*)

get_term(*term*, *variable_factory*)

get_value(*weight*)

get_valued_fact(*relation*, *variable_factory*, *default_weight=None*, *is_example=False*)

get_variable_factory()

get_weight(*weight*, *name*, *fixed*)

neuralogic.core.constructs.metadata module

class Metadata(*offset=None*, *learnable: Optional[bool] = None*, *activation: Optional[Union[str, Activation, ActivationAgg]] = None*, *aggregation: Optional[Union[str, Aggregation]] = None*, *duplicit_grounding: bool = False*)

Bases: object

activation

aggregation

duplicit_grounding

static from_iterable(*iterable: Iterable*) → *Metadata*

learnable

offset

neuralogic.core.constructs.predicate module

class Predicate(*name*, *arity*, *hidden=False*, *special=False*)

Bases: object

arity

hidden

name

set_arity(*arity*)

special

to_str()

class PredicateMetadata(*predicate:* Predicate, *metadata:* Metadata)

Bases: object

metadata

predicate

neuralogic.core.constructs.rule module

class Rule(*head*, *body*)

Bases: object

body

head

is_ellipsis_templated() → bool

metadata: Optional[*Metadata*]

Module contents

neuralogic.core.settings package

Submodules

neuralogic.core.settings.settings_proxy module

class SettingsProxy(*, *optimizer:* Optimizer, *learning_rate: float*, *epochs: int*, *error_function: ErrorFunction*, *initializer:* Initializer, *rule_activation: Activation*, *relation_activation: Activation*, *iso_value_compression: bool*, *chain_pruning: bool*)

Bases: object

property chain_pruning: bool

property debug_exporting: bool

property default_fact_value: float

property epochs: int

property error_function

get_activation_function(*activation: Activation*)

property initializer

property initializer_const

property initializer_uniform_scale

property iso_value_compression: bool

property learning_rate: float

property optimizer

property relation_activation: Activation

property rule_activation: Activation

to_json() → str

Module contents

class Settings(*, optimizer: ~neuralogic.core.enums.Optimizer = Optimizer.ADAM, learning_rate: ~typing.Optional[float] = None, epochs: int = 3000, error_function: ~neuralogic.nn.loss.ErrorFunction = <neuralogic.nn.loss.MSE object>, initializer: ~neuralogic.nn.init.Initializer = <neuralogic.nn.init.Uniform object>, rule_activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>, relation_activation: ~neuralogic.core.constructs.function.Activation = <neuralogic.core.constructs.function.Activation object>, iso_value_compression: bool = True, chain_pruning: bool = True)

Bases: object

property chain_pruning: bool

create_disconnected_proxy() → *SettingsProxy*

create_proxy() → *SettingsProxy*

property epochs: int

property error_function: ErrorFunction

property initializer: *Initializer*

property iso_value_compression: bool

property learning_rate: float

property optimizer: *Optimizer*

property relation_activation: Activation

property rule_activation: Activation

Submodules

neuralogic.core.enums module

class Backend(*value*)

Bases: Enum

An enumeration.

DYNET = 'dynet'

JAVA = 'java'

TORCH = 'torch'

class Optimizer(*value*)

Bases: str, Enum

An enumeration.

ADAM = 'ADAM'

SGD = 'SGD'

neuralogic.core.sources module

class Sources(*sources*)

Bases: object

static from_args(*args: List[str]*, *settings:* SettingsProxy) → *Sources*

static from_settings(*settings:* SettingsProxy) → *Sources*

to_json() → str

neuralogic.core.template module

class Template(*, *template_file: Optional[str] = None*)

Bases: object

add_hook(*relation: Union[BaseRelation, str]*, *callback: Callable[[Any], None]*) → None

Hooks the callable to be called with the relation's value as an argument when the value of the relation is being calculated.

Parameters

- **relation** –
- **callback** –

Returns

add_module(*module: Module*)

Expands the module into rules and adds them into the template

Parameters
module –

Returns

add_rule(*rule*) → None

Adds one rule to the template

Parameters
rule –

Returns

add_rules(*rules: List*)

Adds multiple rules to the template

Parameters
rules –

Returns

build(*settings:* Settings, *backend:* Backend = *Backend.JAVA*)

draw(*filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*, **args*, ***kwargs*)

get_parsed_template(*settings:* SettingsProxy, *java_factory:* JavaFactory)

remove_duplicates()

Remove duplicates from the template

remove_hook(*relation: Union[BaseRelation, str]*, *callback*)

Removes the callable from the relation's hooks

Parameters

- **relation** –
- **callback** –

Returns

Module contents

13.1.2 neuralogic.dataset package

Available dataset formats

- *Dataset* (Logic format)
- *FileDataset*
- *TensorDataset*

---

class Dataset(*examples: Optional[List[List[Union[BaseRelation, WeightedRelation, Rule]]]] = None*, *queries: Optional[List[Union[List[Union[BaseRelation, WeightedRelation, Rule]], BaseRelation, WeightedRelation, Rule]]] = None*)

Dataset encapsulating (learning) samples in the form of logic format, allowing users to fully take advantage of the PyNeuraLogic library.

One learning sample consists of: * Example: A list of logic facts and rules representing some instance (e.g., a graph) * Query: A logic fact to mark the output of a model and optionally target label.

Examples and queries in the dataset can be paired in the following ways:

- N:N - Dataset contains N examples and N queries. They will be paired by their index.

```
dataset.add_example(first_example)
dataset.add_example(second_example)

dataset.add_query(first_query)
dataset.add_query(second_query)

# Learning samples: [first_example, first_query], [second_example, second_query]
```

- 1:N - Dataset contains 1 example and N queries. All queries will be run on the example.

```
dataset.add_example(example)

dataset.add_query(first_query)
dataset.add_query(second_query)

# Learning samples: [example, first_query], [example, second_query]
```

- N:M - Dataset contains N examples and M queries (N <= M). It pairs queries similarly to the N: N case but also allows running multiple queries on a specific example (by inserting a list of queries instead of one query).

```
dataset.add_example(first_example)
dataset.add_example(second_example)

dataset.add_query([first_query_0, first_query_1])
dataset.add_query(second_query)

# Learning samples:
#   [first_example, first_query_0], [first_example, first_query_1], [second_example,
↪ second_query]
```

Parameters

- **examples** (*Optional[List]*) – List of examples. Default: None
- **queries** (*Optional[List]*) – List of queries. Default: None

class FileDataset(*examples_file: Optional[str] = None*, *queries_file: Optional[str] = None*)

FileDataset represents samples stored in files in the NeuraLogic (logic) format.

Parameters

- **examples_file** (*Optional[str]*) – Path to the examples file. Default: None
- **queries_file** (*Optional[str]*) – Path to the queries file. Default: None

class Data(*x: Sequence*, *edge_index: Sequence*, *y: Union[Sequence, float, int]*, *edge_attr: Optional[Sequence] = None*, *y_mask: Optional[Sequence] = None*)

The Data instance stores information about one specific graph instance.

Example

For example, the directed graph $G = (V, E)$, where $E = \{(0, 1), (1, 2), (2, 0)\}$, node features $X = \{[0], [1], [0]\}$ and target nodes' labels $Y = \{0, 1, 0\}$ would be represented as:

```
data = Data(
    x=[[0], [1], [0]],
    edge_index=[
        [0, 1, 2],
        [1, 2, 0],
    ],
    y=[0, 1, 0],
)
```

Parameters

- **x** (*Sequence*) – Sequence of node features.
- **edge_index** (*Sequence*) – Edges represented via a graph connectivity format - matrix [[...src], [...dst]].
- **y** (*Union[Sequence, float, int]*) – Sequence of labels of all nodes or one graph label.
- **edge_attr** (*Optional[Sequence]*) – Optional sequence of edge features. Default: None

- **y_mask** (*Optional[Sequence]*) – Optional sequence of node ids to generate queries for. Default: None (all nodes)

static from_pyg(*data*) → List[*Data*]

Converts a PyTorch Geometric Data instance into a list of PyNeuraLogic *Data* instances. The conversion supports train_mask, test_mask and val_mask attributes - for each mask the conversion yields a new data instance.

Parameters
data – The PyTorch Geometric Data instance

Returns
The list of PyNeuraLogic *Data* instances

class TensorDataset(*data: List[Data]*, *one_hot_encode_labels: bool = False*, *one_hot_decode_features: bool = False*, *number_of_classes: int = 1*, *feature_name: str = 'node_feature'*, *edge_name: str = 'edge'*, *output_name: str = 'predict'*)

The TensorDataset holds a list of *Data* instances - a list of graphs represented in a tensor format.

Parameters

- **data** (*List[Data]*) – List of data (graph) instances.
- **one_hot_encode_labels** (*bool*) – Turn numerical labels into one hot encoded vectors - e.g., label 2 would be turned into a vector [0, 0, 1, .., 0] of length number_of_classes. Default: False
- **one_hot_decode_features** (*bool = False*) – Turn one hot encoded feature vectors into a scalar - e.g., feature vector [0, 0, 1] would be turned into a scalar feature 2. Default: False
- **number_of_classes** (*int*) – Specifies the number of classes for converting numerical labels to one hot encoded vectors. Default: 1
- **feature_name** (*str*) – Specify the node feature predicate name used for converting into the logic format. Default: node_feature
- **edge_name** (*str*) – Specify the edge predicate name used for converting into the logic format. Default: edge
- **output_name** (*str*) – Specify the output predicate name used for converting into the logic format. Default: predict

13.1.3 neuralogic.nn package

Subpackages

neuralogic.nn.evaluator package

Submodules

neuralogic.nn.evaluator.dynet module

neuralogic.nn.evaluator.java module

class JavaEvaluator(*problem: Optional[Template]*, *settings: Settings*)

Bases: *AbstractEvaluator*

load_state_dict(*state_dict: Dict*)

reset_dataset(*dataset*)

set_dataset(*dataset: Union[BaseDataset, BuiltDataset]*)

state_dict() → Dict

test(*dataset: Optional[Union[BaseDataset, BuiltDataset]] = None*, ***, *generator: bool = True*)

train(*dataset: Optional[Union[BaseDataset, BuiltDataset]] = None*, ***, *generator: bool = True*, *epochs: Optional[int] = None*)

neuralogic.nn.evaluator.torch module

class TorchEvaluator(*template: Template*, *settings: Settings*)

Bases: *AbstractEvaluator*

error_functions = {'SOFTENTROPY': CrossEntropyLoss(), 'SQUARED_DIFF': MSELoss()}

load_state_dict(*state_dict: Dict*)

state_dict() → Dict

test(*dataset: Optional[Union[BaseDataset, BuiltDataset]] = None*, ***, *generator: bool = True*)

train(*dataset: Optional[Union[BaseDataset, BuiltDataset]] = None*, ***, *generator: bool = True*)

trainers = {Optimizer.ADAM: <function TorchEvaluator.<lambda>>, Optimizer.SGD: <function TorchEvaluator.<lambda>>}

Module contents

Submodules

neuralogic.nn.init module

class Constant(*value: float = 0.1*)

Bases: *Initializer*

Initializes learnable parameters with the `value`.

Parameters

value (*float*) – Value to fill weights with. Default: `0.1`

get_settings() → Dict[str, Any]

class Glorot(*scale: float = 2*)

Bases: *Initializer*

Initializes learnable parameters with samples from a uniform distribution (from the interval `[-scale / 2, scale / 2]`) using the Glorot method.

Parameters
scale (*float*) – Scale of a uniform distribution interval [-scale / 2, scale / 2]. Default: 2

get_settings() → Dict[str, Any]

is_simple() → bool

class **He**(*scale: float = 2*)

Bases: *Initializer*

Initializes learnable parameters with samples from a uniform distribution (from the interval `[-scale / 2, scale / 2]`) using the He method.

Parameters
scale (*float*) – Scale of a uniform distribution interval [-scale / 2, scale / 2]. Default: 2

get_settings() → Dict[str, Any]

is_simple() → bool

class **Initializer**

Bases: `object`

get_settings() → Dict[str, Any]

is_simple() → bool

class **InitializerNames**

Bases: `object`

CONSTANT = 'CONSTANT'

GLOROT = 'GLOROT'

HE = 'HE'

LONGTAIL = 'LONGTAIL'

NORMAL = 'NORMAL'

UNIFORM = 'UNIFORM'

class **Longtail**

Bases: *Initializer*

Initializes learnable parameters with random samples from a long tail distribution

class **Normal**

Bases: *Initializer*

Initializes learnable parameters with random samples from a normal (Gaussian) distribution

class **Uniform**(*scale: float = 2*)

Bases: *Initializer*

Initializes learnable parameters with random uniformly distributed samples from the interval `[-scale / 2, scale / 2]`.

Parameters
scale (*float*) – Scale of the distribution interval [-scale / 2, scale / 2]. Default: 2

get_settings() → Dict[str, Any]

neuralogic.nn.base module

class AbstractEvaluator(*backend:* Backend, *template:* Template, *settings:* Settings)

Bases: object

build_dataset(*dataset: Union[BaseDataset,* BuiltDataset*]*, *file_mode: bool = False*)

draw(*filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*, **args*, ***kwargs*)

load_state_dict(*state_dict: Dict*)

property model: *AbstractNeuraLogic*

parameters() → Dict

reset_parameters()

set_dataset(*dataset: Union[BaseDataset,* BuiltDataset*]*)

state_dict() → Dict

test(*dataset: Optional[Union[BaseDataset,* BuiltDataset*]] = None*, ***, *generator: bool = True*)

train(*dataset: Optional[Union[BaseDataset,* BuiltDataset*]] = None*, ***, *generator: bool = True*)

class AbstractNeuraLogic(*backend:* Backend, *dataset_builder:* DatasetBuilder, *template:* Template, *settings:* SettingsProxy)

Bases: object

build_dataset(*dataset: Union[BaseDataset,* BuiltDataset*]*, *file_mode: bool = False*)

draw(*filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*, **args*, ***kwargs*)

load_state_dict(*state_dict: Dict*)

parameters() → Dict

run_hook(*hook: str*, *value*)

set_hooks(*hooks*)

state_dict() → Dict

sync_template(*state_dict: Optional[Dict] = None*, *weights=None*)

neuralogic.nn.dynet module

neuralogic.nn.java module

class NeuraLogic(*model*, *dataset_builder*, *template*, *settings:* SettingsProxy)

Bases: *AbstractNeuraLogic*

load_state_dict(*state_dict: Dict*)

reset_parameters()

set_training_samples(*samples*)

state_dict() → Dict

test()

train()

neuralogic.nn.torch module

class NeuraLogic(*model: List[*Weight*]*, *dataset_builder*, *template*, *settings: Optional[*SettingsProxy*] = None*)

Bases: *AbstractNeuraLogic*

activations = {'Average': <built-in method mean of type object>, 'Maximum': <built-in method max of type object>, 'Minimum': <built-in method min of type object>, 'ReLu': <built-in method relu of type object>, 'Sigmoid': <built-in method sigmoid of type object>, 'Sum': <built-in method sum of type object>, 'Tanh': <built-in method tanh of type object>}

build_sample(*sample:* Sample)

initializers = {'CONSTANT': <function NeuraLogic.<lambda>>, 'GLOROT': <function NeuraLogic.<lambda>>, 'HE': <function NeuraLogic.<lambda>>, 'LONGTAIL': <function longtail>, 'NORMAL': <function NeuraLogic.<lambda>>, 'UNIFORM': <function NeuraLogic.<lambda>>}

load_state_dict(*state_dict: Dict*)

process_neuron_inputs(*neuron:* Neuron, *neurons: List[Tensor]*, *weights: ParameterList*) → Tuple[List[Union[Tensor, int, float]], List[Tensor], List[Tensor]]

reset_parameters()

state_dict() → Dict

static to_tensor_value(*value*) → Tensor

to_torch_expression(*neuron:* Neuron, *neurons: List[Tensor]*, *weights: ParameterList*) → Tensor

longtail(*tensor: Tensor*, *_:* SettingsProxy)

Module contents

get_evaluator(*template*, *settings=None*, *backend:* Backend = *Backend.JAVA*)

get_neuralogic_layer(*backend:* Backend = *Backend.JAVA*)

13.1.4 neuralogic.utils package

Subpackages

neuralogic.utils.data package

Module contents

`Family`()

`Mutagenesis`()

`Nations`()

`Trains`()

`XOR`()

`XOR_Vectorized`()

neuralogic.utils.visualize package

Module contents

`draw`(*drawer*, *obj*, *filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, **args*, ***kwargs*)

`draw_model`(*model*, *filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*, **args*, ***kwargs*)

Draws model either as an image of type img_type either into:

- a file - if filename is specified),
- an IPython Image - if draw_ipython is True
- or bytes otherwise

Parameters

- `model` –
- `filename` –
- `draw_ipython` –
- `img_type` –
- `value_detail` –
- `graphviz_path` –
- `args` –
- `kwargs` –

Returns

draw_sample(*sample*, *filename: Optional[str] = None*, *draw_ipython=True*, *img_type='png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*, **args*, ***kwargs*)

Draws sample either as an image of type img_type either into:

- a file - if filename is specified),
- an IPython Image - if draw_ipython is True
- or bytes otherwise

Parameters

- **sample** –
- **filename** –
- **draw_ipython** –
- **img_type** –
- **detail** –
- **graphviz_path** –
- **args** –
- **kwargs** –

Returns

get_drawing_settings(*img_type: str = 'png'*, *value_detail: int = 0*, *graphviz_path: Optional[str] = None*) → *SettingsProxy*

Returns the default settings instance for drawing with a specified image type.

Parameters

- **img_type** –
- **value_detail** –
- **graphviz_path** –

Returns

get_sample_drawer(*settings:* SettingsProxy)

get_template_drawer(*settings:* SettingsProxy)

model_to_dot_source(*model*) → str

Renders the model into its dot source representation.

Parameters
model –

Returns

sample_to_dot_source(*sample*, *value_detail: int = 0*) → str

Renders the sample into its dot source representation.

Parameters

- **sample** –
- **value_detail** –

Returns

`to_dot_source`(*drawer*, *obj*) → str

Module contents

13.2 Submodules

13.3 neuralogic.logging module

`class Formatter`(*value*)

Bases: `Enum`

Logged information formatters

`COLOR = 'color'`

`NORMAL = 'normal'`

`class Level`(*value*)

Bases: `Enum`

Logging level

`ALL = 'ALL'`

`CONFIG = 'CONFIG'`

`FINE = 'FINE'`

`FINER = 'FINER'`

`FINEST = 'FINEST'`

`INFO = 'INFO'`

`OFF = 'OFF'`

`SEVERE = 'SEVERE'`

`WARNING = 'WARNING'`

`class TextIOWrapper`(*wrapped_text_io*)

Bases: `object`

`write`(*string*)

`add_handler`(*output*, *level:* Level = *Level.FINER*, *formatter:* Formatter = *Formatter.COLOR*)

Add logger handler for an insight into the java backend

Parameters

- **output** – File-like object (has write(text: str) method)
- **level** – The logging level
- **formatter** – The log formatter

`clear_handlers`()

Clear all handlers

13.4 Module contents

initial_seed() → int

Returns the initial random seed for a random number generator used in the backend

initialize(*debug_mode: bool = False*, *debug_port: int = 12999*, *is_debug_server: bool = True*, *debug_suspend: bool = True*)

is_initialized() → bool

manual_seed(*seed: int*)

Sets the seed for a random number generator used in the backend to the passed **seed**.

Parameters
seed –

seed() → int

Sets the seed for a random number generator used in the backend to a random seed and returns the seed.

set_jvm_options(*options: List[str]*) → None

Set the jvm options - by default [“-Xms1g”, “-Xmx64g”],

set_jvm_path(*path: Optional[str]*) → None

PyNeuraLogic lets you use Python to create Differentiable Logic Programs

---

Logic programming is a declarative coding paradigm in which you declare your logical *variables* and *relations* between them. These can be further composed into so-called *rules* that drive the computation. Such a rule set then forms a *logic program*, and its execution is equivalent to performing logic inference with the rules.

PyNeuralogic, through its NeuraLogic backend, then makes this inference process *differentiable* which, in turn, makes it equivalent to forward propagation in deep learning. This lets you learn numeric parameters that can be associated with the rules, just like you learn weights in neural networks.

CHAPTER

FOURTEEN

WHAT IS THIS GOOD FOR?

Many things! For instance - ever heard of Graph Neural Networks (GNNs)? Well, a *graph* happens to be a special case of a logical relation - a binary one to be more exact. Now, at the heart of any GNN model there is a so-called *propagation rule* for passing 'messages' between the neighboring nodes. Particularly, the representation ('message') of a node `X` is calculated by aggregating the representations of adjacent nodes `Y`, i.e. those with an `edge` between `X` and `Y`.

Or, a bit more 'formally':

```
Relation.node2(Var.X) <= (Relation.node1(Var.Y), Relation.edge(Var.Y, Var.X))
```

...and that's the actual *code*! Now for a classic learnable GNN layer, you'll want to add some numeric parameters, such as

```
Relation.node2(Var.X)[5,10] <= (Relation.node1(Var.Y)[10,20], Relation.edge(Var.Y, Var.
↪X))
```

to project your `[1,20]` input node embeddings through a learnable `[10,20]` layer before the aggregation, and subsequently a `[5,10]` layer after the aggregation. The particular aggregation and activation functions, as well as other details, can naturally be specified further, but you can as well leave it default like we did here with your first, fully functional GNN layer!

CHAPTER

FIFTEEN

HOW IS IT DIFFERENT FROM OTHER GNN FRAMEWORKS?

Naturally, PyNeuralogic is by no means limited to GNN models, as the expressiveness of *relational* logic goes much further beyond graphs. So nothing stops you from playing directly with:

- multiple relations and object types
- hypergraphs, nested graphs, relational databases
- alternative propagation schemes
- direct sub-structure (pattern) matching
- inclusion of logical background knowledge
- and more…

In PyNeuraLogic, all these ideas take the same form of simple small logic programs. These are commonly highly transparent and easy to understand, thanks to their declarative nature. Consequently, there is no need to design a new blackbox class name for each small modification of the GNN rule - you code directly at the level of the logical principles here!

The backend engine then creates the underlying differentiable computation (inference) graphs in a fully automated and dynamic fashion, hence you don't have to care about aligning everything into some (static) tensor operations. This gives you considerably more expressiveness, and, perhaps surprisingly, sometimes even performance.

We hope you'll find the framework useful in designing your own deep **relational** learning ideas beyond the GNNs! Please let us know if you need some guidance or would like to cooperate!

CHAPTER

SIXTEEN

SUPPORTED BACKENDS

Models defined in PyNeuraLogic can be built for and evaluated in different backends. Currently, you can pick and use the following backends, which, except for the Java backend, have to be additionally installed:

- Java
- PyTorch
- DyNet

CHAPTER

SEVENTEEN

EXAMPLES

- Molecular GNNs
- Simple XOR example
- Recursive XOR Generalization
- Visualization
- Pattern Matching
- Distinguishing k-regular graphs
- Distinguishing non-regular graphs

CHAPTER

EIGHTEEN

PAPERS

- Beyond Graph Neural Networks with Lifted Relational Neural Networks Machine Learning Journal, 2021
- Lifted Relational Neural Networks Journal of Artificial Intelligence Research, 2018
- Lossless compression of structured convolutional models via lifting ICLR, 2021

PYTHON MODULE INDEX

n

INDEX

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X