phast

Victor Schmidt, Alexandre Duval

Sep 14, 2023

CONTENTS

This repository contains implementations for 2 of the PhAST components presented in the paper:

- `PhysEmbedding` that allows one to create an embedding vector from atomic numbers that is the concatenation of:
  - A learned embedding for the atom's group
  - A learned embedding for the atom's period
  - A fixed or learned embedding from a set of known physical properties, as reported by `mendeleev`
  - In the case of the OC20 dataset, a learned embedding for the atom's tag (adsorbate, catalyst surface or catalyst sub-surface)
- Tag-based **graph rewiring** strategies for the OC20 dataset:
  - `remove_tag0_nodes` deletes all nodes in the graph associated with a tag 0 and recomputes edges
  - `one_supernode_per_graph` replaces all tag 0 atoms with a single new atom
  - `one_supernode_per_atom_type` replaces all tag 0 atoms *of a given element* with its own super node

Also: https://github.com/vict0rsch/faenet

CHAPTER

ONE

INSTALLATION

```
pip install phast
```

The above installation does not include `torch_geometric` which is a complex and very variable dependency you have to install yourself if you want to use the graph re-wiring functions of `phast`.

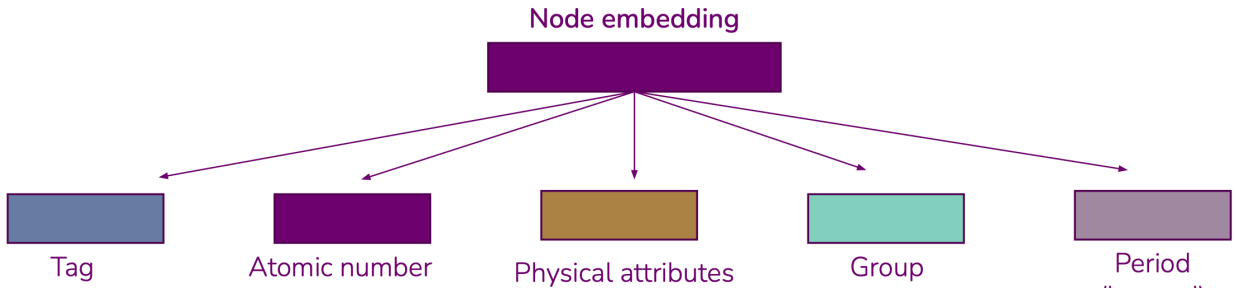
Ignore `torch_geometric` if you only care about the `PhysEmbeddings`.

CHAPTER

TWO

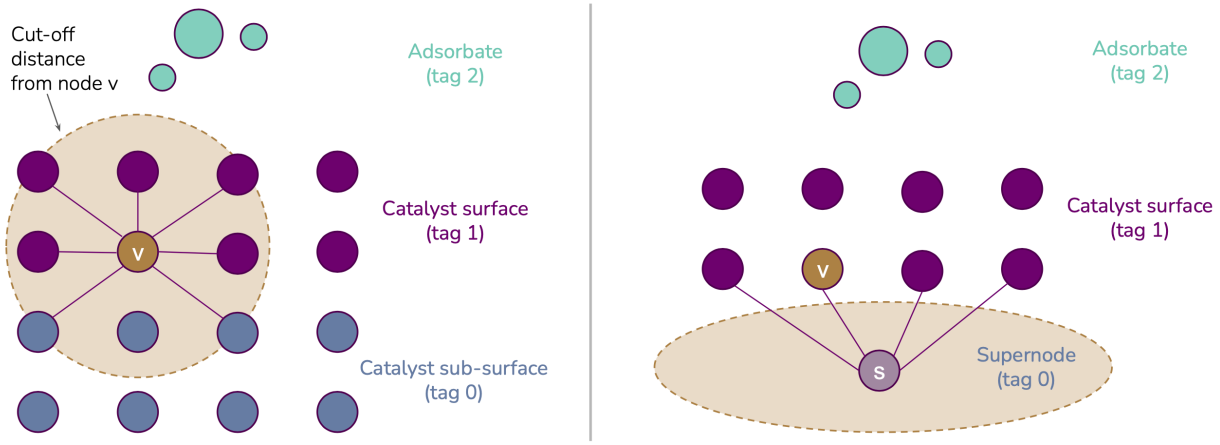
GETTING STARTED

2.1 Physical embeddings




```python
import torch
from phast.embedding import PhysEmbedding

z = torch.randint(1, 85, (3, 12)) # batch of 3 graphs with 12 atoms each
phys_embedding = PhysEmbedding(
    z_emb_size=32, # default
    period_emb_size=32, # default
    group_emb_size=32, # default
    properties_proj_size=32, # default is 0 -> no learned projection
    n_elements=85, # default
)
h = phys_embedding(z) # h.shape = (3, 12, 128)

tags = torch.randint(0, 3, (3, 12))
phys_embedding = PhysEmbedding(
    tag_emb_size=32, # default is 0, this is OC20-specific
    final_proj_size=64, # default is 0, no projection, just the concat. of embeds.
)

h = phys_embedding(z, tags) # h.shape = (3, 12, 64)

# Assuming torch_geometric is installed:
data = torch.load("examples/data/is2re_bs3.pt")
h = phys_embedding(data.atomic_numbers.long(), data.tags) # h.shape = (261, 64)
```

2.2 Graph rewiring




```
from copy import deepcopy
import torch
from phast.graph_rewiring import (
    remove_tag0_nodes,
    one_supernode_per_graph,
    one_supernode_per_atom_type,
)

data = torch.load("./examples/data/is2re_bs3.pt")  # 3 batched OC20 IS2RE data samples
print(
    "Data initially contains {} graphs, a total of {} atoms and {} edges".format(
        len(data.natoms), data.ptr[-1], len(data.cell_offsets)
    )
)
rewired_data = remove_tag0_nodes(deepcopy(data))
print(
    "Data without tag-0 nodes contains {} graphs, a total of {} atoms and {} edges".
↪format(
        len(rewired_data.natoms), rewired_data.ptr[-1], len(rewired_data.cell_offsets)
    )
)
rewired_data = one_supernode_per_graph(deepcopy(data))
print(
    "Data with one super node per graph contains a total of {} atoms and {} edges".
↪format(
        rewired_data.ptr[-1], len(rewired_data.cell_offsets)
    )
)
rewired_data = one_supernode_per_atom_type(deepcopy(data))
print(
    "Data with one super node per atom type contains a total of {} atoms and {} edges".
↪format(
        rewired_data.ptr[-1], len(rewired_data.cell_offsets)
    )
)
```

```
Data initially contains 3 graphs, a total of 261 atoms and 11596 edges
Data without tag-0 nodes contains 3 graphs, a total of 64 atoms and 1236 edges
Data with one super node per graph contains a total of 67 atoms and 1311 edges
Data with one super node per atom type contains a total of 71 atoms and 1421 edges
```

CHAPTER

THREE

TESTS

This requires poetry. Make sure to have torch and torch_geometric installed in your environment before you can run the tests. Unfortunately because of CUDA/torch compatibilities, neither torch nor torch_geometric are part of the explicit dependencies and must be installed independently.

```
git clone git@github.com:vict0rsch/phast.git
poetry install --with dev
pytest --cov=phast --cov-report term-missing
```

Testing on Macs you may encounter a Library Not Loaded Error

Requires Python <3.12 because

```
mendeleev (0.14.0) requires Python >=3.8.1,<3.12
```

CHAPTER

FOUR

API REFERENCE

This page contains auto-generated API reference documentation[1].

4.1 phast

`phast` Python package structure:

- `phast.embedding` Physics-based embedding of atomic graphs, notably *phast.embedding.PhysEmbedding*
- `phast.graph_rewiring` OC20-specific graph rewiring functions, notably *phast.graph_rewiring.remove_tag0_nodes()*

To use `phast.graph_rewiring`, you must install PyTorch Geometric.

4.1.1 Submodules

4.1.1.1 phast.embedding

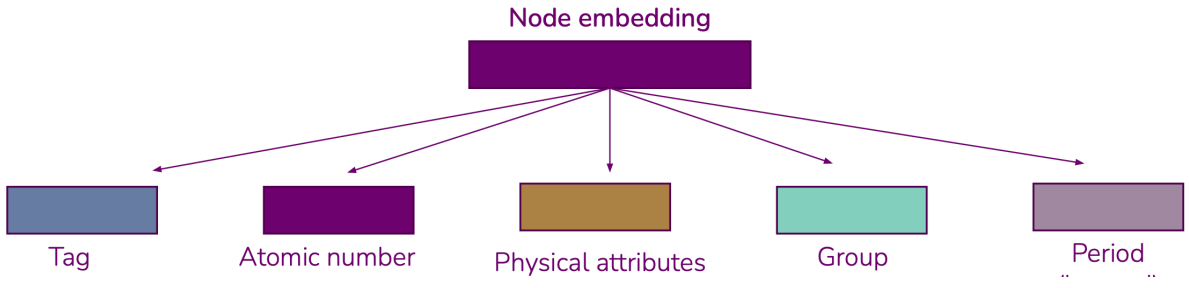
A Python module that endows graph neural networks with physical priors as part of the embeddings of atoms from their characteristic number.

This package contains the implementation of a set of classes that are used to create atomic embeddings from physical properties of periodic table elements.

The physical embeddings are learned or kept fixed depending on the specific use-case. The embeddings can also include information regarding the group and period of the elements.

In the context of the Open Catalyst datasets, tag embeddings can also be used.

This implementation relies on Mendeleev package to access the physical properties of elements from the periodic table.



[1] Created with sphinx-autoapi

```
import torch
from phast.embedding import PhysEmbedding

z = torch.randint(1, 85, (3, 12)) # batch of 3 graphs with 12 atoms each
phys_embedding = PhysEmbedding(
    z_emb_size=32, # default
    period_emb_size=32, # default
    group_emb_size=32, # default
    properties_proj_size=32, # default is 0 -> no learned projection
    n_elements=85, # default
)
h = phys_embedding(z) # h.shape = (3, 12, 128)

tags = torch.randint(0, 3, (3, 12))
phys_embedding = PhysEmbedding(
    tag_emb_size=32, # default is 0, this is OC20-specific
    final_proj_size=64, # default is 0, no projection, just the concat. of embeds.
)

h = phys_embedding(z, tags) # h.shape = (3, 12, 64)

# Assuming torch_geometric is installed:

data = torch.load("examples/data/is2re_bs3.pt")
h = phys_embedding(data.atomic_numbers.long(), data.tags) # h.shape = (261, 64)
```

Classes

PhysEmbedding	This module embeds inputs for use in a neural network, using both
PhysRef	This class implements an interface to access physical properties, period and
PropertiesEmbedding	A class for retrieving physical properties from atomic numbers.

class phast.embedding.**PhysEmbedding**(*z_emb_size=32*, *tag_emb_size=0*, *period_emb_size=32*, *group_emb_size=32*, *properties=PhysRef.default_properties*, *properties_grad=False*, *properties_proj_size=0*, *final_proj_size=0*, *n_elements=85*)

Bases: `torch.nn.Module`

This module embeds inputs for use in a neural network, using both standard embeddings and physical properties. The input to the embedding module can be a set of compositions, atomic numbers and tags, in addition to any extra physical properties specified.

You can disable embeddings by setting their size to 0.

Parameters

- **z_emb_size** (`int`) – Size of the embedding for atomic number.
- **tag_emb_size** (`int`) – Size of the embedding for tags.
- **period_emb_size** (`int`) – Size of the embedding for periods.

- **group_emb_size** (int) – Size of the embedding for groups.
- **properties** (list) – List of the physical properties to include in the embedding. Each property is specified as a string, and should correspond to a valid attribute of the Pymatgen Composition class.
- **properties_proj_size** (int) – Projection size of the physical properties embedding.
- **properties_grad** (bool) – Whether to set the physical properties to be trainable or not.
- **final_proj_size** (int) – Projection size for the final embedding.
- **n_elements** (int) – Number of elements in the periodic table.

Raises

- **ValueError** – if *self.properties_proj_size* is greater than 0 and *self.properties* is empty
- **ValueError** – if *self.full_emb_size* is 0, i.e. all sizes were set to 0.

z_emb_size

Size of the embedding for atomic number.

Type
int

tag_emb_size

Size of the embedding for tags.

Type
int

period_emb_size

Size of the embedding for periods.

Type
int

group_emb_size

Size of the embedding for groups.

Type
int

properties

List of the physical properties to include in the embedding. Each property must be a string as per the `elements` or `fetch_ionization_energies` Mendeleev tables.

Type
list

properties_grad

Whether to set the physical properties to be trainable or not.

Type
bool

n_elements

Number of elements in the periodic table to consider.

Type
int

phys_ref

Reference physical information interface.

Type
PhysRef

full_emb_size

Total size of the concatenated embeddings.

Type
int

final_emb_size

Output size: either the final_proj_size or full_emb_size.

Type
int

embeddings

Dictionary containing the different embeddings.

Type
nn.ModuleDict

phys_lin

A linear layer to project the physical properties to the given size, if projection is requested.

Type
nn.Linear

final_proj

A linear layer to project the final embedding to the requested size.

Type
nn.Linear

forward(*z*, *tag=None*)

Embeds the input(s) using the available embeddings. Final embedding size is the sum of the individual embedding sizes, except if *final_proj_size* is provided, in which case the final embedding is projected to the requested size with an unbiased linear layer.

Parameters

- **z** (*torch.Tensor*) – Tensor of (long) atomic numbers.
- **tag** (*Optional[torch.Tensor]*) – Open Catalyst Project-style tags. Defaults to None.

Returns
Embedded representation of the input(s).

Return type
torch.Tensor

reset_parameters()

Resets the parameters of the linear layers, and the embeddings.

class phast.embedding.**PhysRef**(*properties=[]*, *period=True*, *group=True*, *short=False*, *n_elements=85*)

Bases: torch.nn.Module

This class implements an interface to access physical properties, period and group ids of elements from the periodic table.

Parameters

- **properties** (*list*) –
- **period** (*bool*) –
- **group** (*bool*) –
- **short** (*bool*) –
- **n_elements** (*int*) –

default_properties

A list of the default properties part of atom embeddings.

Type
list

properties_list

A list of the properties that are actually used for creating the embeddings.

Type
list

n_groups

The number of groups of the elements.

Type
int

n_periods

The number of periods of the elements.

Type
int

n_properties

The number of properties of the elements that are used to create the embeddings.

Type
int

properties

Whether to create an embedding of physical embeddings.

Type
bool

properties_grad

Whether the physical properties embedding should be learned or kept fixed.

Type
bool

period

Whether to use period embeddings.

Type
bool

group
Whether to use group embeddings.

Type
bool

short
A boolean flag indicating whether to keep only the columns that do not have NaN values.

Type
bool

group_mapping
A tensor containing the mapping from the element atomic number to the corresponding group embedding.

Type
torch.Tensor

period_mapping
A tensor containing the mapping from the element atomic number to the corresponding period embedding.

Type
torch.Tensor

properties_mapping
A tensor containing the mapping from the element atomic number to the corresponding physical properties embedding.

Type
torch.Tensor

__init__()
Initializes the PhysRef class.

Parameters

- **properties** (*list*) –
- **period** (*bool*) –
- **group** (*bool*) –
- **short** (*bool*) –
- **n_elements** (*int*) –

Return type
None

__repr__()
Returns a string representation of the class instance.

period_and_group()
Returns the period and group embeddings of the elements.

```
default_properties = ['atomic_radius', 'atomic_volume', 'density',
'dipole_polarizability', 'electron_affinity',...
```

period_and_group(*z*)

class phast.embedding.**PropertiesEmbedding**(*properties*, *grad=False*)

Bases: `torch.nn.Module`

A class for retrieving physical properties from atomic numbers.

Parameters

- **properties** (`torch.Tensor`) – A tensor containing the properties to be embedded.
- **grad** (`bool`) – Whether to enable gradient computation or not.

properties

A parameter or buffer storing the properties.

Type
nn.Parameter or nn.Buffer

forward(*z*)

Returns the embedded properties at the specified indices.

Parameters
z (*torch.Tensor*) –

reset_parameters()

Does nothing in this class.

forward(*z*)

Returns a properties for each atom in the batch according to (1-based) atomic numbers.

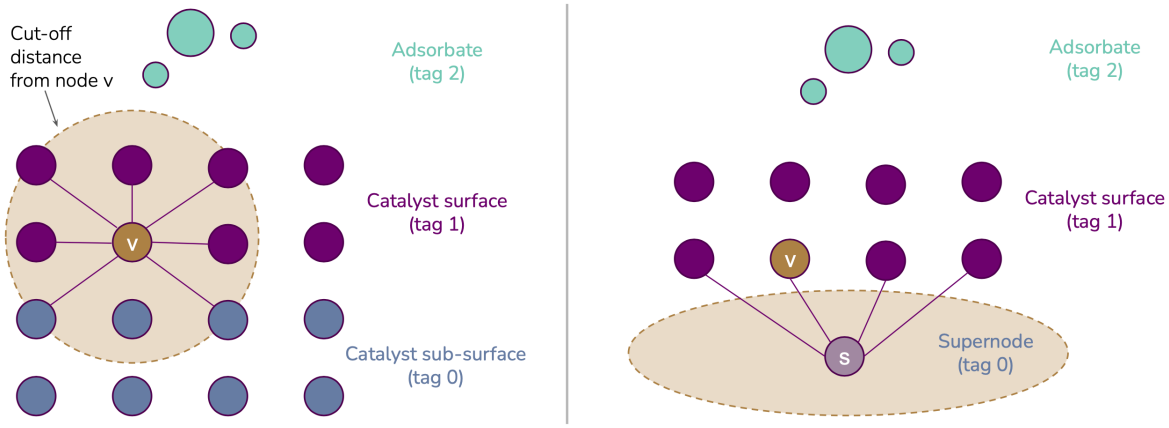
Parameters
z (*torch.Tensor*) – Tensor of atomic numbers as `torch.Long`.

Returns
The properties for each atom.

reset_parameters()

4.1.1.2 phast.graph_rewiring

In the context of the OC20 dataset, rewire each 3D molecular graph according to 1 of 3 strategies: remove all tag-0 atoms, aggregate all tag-0 atoms into a single super-node, or aggregate all tag-0 atoms of a given element into a single super-node (hence, up to 3 super nodes will be created since 0C20 catalysts can have up to 3 elements).



```
from phast.graph_rewiring import remove_tag0_nodes

data = load_oc20_data_batch() # Yours to define
rewired_data = remove_tag0_nodes(data)
```

Warning: This modules expects `torch_geometric` to be installed.

Functions

adjust_cutoff_distances(data, sn_indxes[, cutoff])	Because of rewiring, some edges could be now longer than
one_supernode_per_atom_type(data[, cutoff])	For each graph independently, replace all tag-0 atoms of a given element by a new
one_supernode_per_graph(data[, cutoff, num_elements])	Replaces all tag-0 atom with a single super-node S representing them, per graph.
remove_tag0_nodes(data)	Delete sub-surface (`data.tag == 0`) nodes and rewire the graph accordingly.

phast.graph_rewiring.**adjust_cutoff_distances**(*data*, *sn_indxes*, *cutoff=6.0*)

Because of rewiring, some edges could be now longer than the allowed cutoff distance. This function removes them.

Modified attributes: * `edge_index` * `cell_offsets` * `distances` * `neighbors`

Warning: This function modifies the input data in-place.

Parameters

- **data** (*torch_geometric.Data*) – The rewired graph data.
- **sn_indxes** (*torch.Tensor[torch.Long]*) – Indices of the supernodes.
- **cutoff** (*float, optional*) – Maximum edge length. Defaults to 6.0.

Returns

The updated graph.

Return type

torch_geometric.Data

phast.graph_rewiring.**one_supernode_per_atom_type**(*data*, *cutoff=6.0*)

For each graph independently, replace all tag-0 atoms of a given element by a new super node S_i, $i \in \{1..3\}$. As per *one_supernode_per_graph()*, each S_i is positioned at the center of mass of the atoms it replaces in x and y dimensions but at the maximum height of the atoms it replaces in the z dimension.

Expected `data` attributes are the same as for *remove_tag0_nodes()*.

Note: S_i conserves the atomic number of the tag-0 atoms it replaces.

Warning: This function modifies the input data in-place.

Parameters

- **data** (*torch_geometric.Data*) – the data batch to re-wire
- **cutoff** (*float*) –

Returns
the data rewired data batch

Return type
torch_geometric.Data

phast.graph_rewiring.**one_supernode_per_graph**(*data*, *cutoff=6.0*, *num_elements=83*)

Replaces all tag-0 atom with a single super-node S representing them, per graph. For each graph, S is the last node in the graph. S is positioned at the center of mass of all tag-0 atoms in x and y directions but at the maximum z coordinate of all tag-0 atoms. All atoms previously connected to a tag-0 atom are now connected to S unless that would create an edge longer than `cutoff`.

Expected `data` attributes are the same as for *remove_tag0_nodes()*.

Note: S will be created with a new atomic number $Z_S = num\_elements + 1$, so this should be set to the number of elements expected to be present in the dataset, not that of the current graph.

Warning: This function modifies the input data in-place.

Parameters

- **data** (*data.Data*) – single batch of graphs
- **cutoff** (*float*) –
- **num_elements** (*int*) –

Return type
Union[torch_geometric.data.Batch, torch_geometric.data.Data]

phast.graph_rewiring.**remove_tag0_nodes**(*data*)

Delete sub-surface (`data.tag == 0`) nodes and rewire the graph accordingly.

Warning: This function modifies the input data in-place.

Expected `data` tensor attributes:

- `pos`: node positions
- `atomic_numbers`: atomic numbers
- `batch`: mini-batch id for each atom
- `tags`: atom tags
- `edge_index`: edge indices as a $2imesE$ tensor

- `force`: force vectors per atom (optional)
- `pos_relaxed`: relaxed atom positions (optional)
- `fixed`: mask for fixed atoms (optional)
- `natoms`: number of atoms per graph
- `ptr`: cumulative sum of `natoms`
- `cell_offsets`: unit cell directional offset for each edge
- `distances`: distance between each edge's atoms

Parameters
data (*torch_geometric.Data*) – the data batch to re-wire

Return type
Union[torch_geometric.data.Batch, torch_geometric.data.Data]

4.1.1.3 phast.utils

Functions

ensure_pyg_ok(func)	Decorator to ensure that torch_geometric is installed when

Attributes

PYG_OK

phast.utils.**PYG_OK = True**

phast.utils.**ensure_pyg_ok**(*func*)

Decorator to ensure that torch_geometric is installed when using a function that requires it.

Parameters
func (*callable*) – Function to decorate.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

D

E

F

G

M

N

O

P

R

S

T

Z