TS-DiffuGen

Release 2023

Sacha Raffaud

Aug 31, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION:

This file contains simple documentation regarding the TS-DiffuGen package.

CHAPTER

TWO

DATASETS:

The dataset classes can be found within the data directory.

2.1 Setting up file:

`data.Dataset_W93.setup_dataset_files.`**`create_pdb_from_file_path`**(*path_to_raw_log*, *path_to_final*)

Create a .XYZ file from a .log DFT file containing atomic coordinates.

This function reads a DFT .log file, extracts atomic symbols and coordinates, and writes them into a .PDB file format.

Parameters

- **`path_to_raw_log`** (*`str`*) – Path to the input DFT .log file.
- **`path_to_final`** (*`str`*) – Path to the output .PDB file to be created.

Returns

None

`data.Dataset_W93.setup_dataset_files.`**`extract_geometry`**(*logs*)

Extract geometry from DFT .log files.

This function extracts atomic symbols and coordinates from a list of lines obtained from a DFT .log file. It searches for the section indicating the standard nuclear orientation and extracts the atom labels and corresponding coordinates in the next line.

Parameters

`logs` (*`list`*) – List of lines from the DFT .log file.

Returns

A tuple containing a list of atomic symbols and a numpy array of atomic coordinates.

Return type

tuple

`data.Dataset_W93.setup_dataset_files.`**`process_reactions`**(*dataframe*, *directory*)

Process reaction data from .log files and create various output files.

This function processes reaction data stored in .log files and organizes the data into clean geometry files in .xyz format. It also generates images of the reactants and products and saves them, along with the geometry files, for each reaction.

Parameters

- **`dataframe`** (*`pd.DataFrame`*) – DataFrame containing reaction data, including reactant and product SMILES.

- **directory** (*str*) – Path to the main directory containing the raw log files.

Returns
None

2.2 Simple W93 Dataset without Graphs:

class data.Dataset_W93.dataset_class.**W93_TS**(**args: Any*, ***kwargs: Any*)

atom_count()
Counts the occurrences of different atom types in the reactions.

convert_to_float(*data*)
Convert nested list elements to floats.

convert_to_list(*data*)
Convert nested list elements to Lists.

count_data()
Counts the number of reactions in the dataset.

create_data_array()
Creates data arrays with context information based on the selected context or without context.

create_graph()
Creates fully connected graphs from the data.

delete_centre_gravity()
Removes the center of gravity from molecule coordinates. Needed to keep model roto-translation invariant

extract_data(*reaction_number*)
Extracts reactant, product, and transition state information from specified reaction.

generate_one_hot_encodings(*num_of_atoms*)
Generates one-hot encodings for atom types.

get(*idx*)
Retrieves a data sample and its corresponding node mask.

get_keys_from_value(*search_value*)
Retrieves atom types based on their one-hot encoded vectors.

len()
Returns the length of the dataset.

load_data()
Loads the dataset by extracting data from reaction files and performing preprocessing steps.

one_hot_encode()
Performs one-hot encoding of atom types and prepares other data-processing.

pad_data(*data*, *max_length*)

Pads molecule so that all have the same size
and can be fed through batch_loader

plot_molecule_size_distribution()

Plots the distribution of molecule sizes in the dataset.

read_xyz_file(*file_path*)

Reads an XYZ file and returns its data.

replace_atom_types_with_ohe_vectors(*molecule_list*)

Replaces atom types in molecule data with their one-hot encoded vectors.

2.3 W93 Dataset with Graphs:

class data.Dataset_W93.dataset_reaction_graph.**W93_TS_coords_and_reacion_graph**(*directory='data/Dataset_W93/da* *run-* *ning_pytest=False*)

atom_count()

Counts the occurrences of different atom types in the reactions.

convert_to_float(*data*)

Convert nested list elements to floats.

convert_to_list(*data*)

Convert nested list elements to Lists.

count_data()

Counts the number of reactions in the dataset.

create_data_array()

Creates data arrays.

delete_centre_gravity()

Removes the center of gravity from molecule coordinates. Needed to keep model roto-translation invariant

extract_data(*reaction_number*)

Extracts reactant, product, and transition state information from specified reaction.

generate_one_hot_encodings(*num_of_atoms*)

Generates one-hot encodings for atom types.

get(*idx*)

Retrieves a data sample and its corresponding node mask.

get_keys_from_value(*search_value*)

Retrieves atom types based on their one-hot encoded vectors.

len()

Returns the length of the dataset.

load_data()

# noqa Loads the dataset by extracting data from reaction files and performing preprocessing steps.

one_hot_encode()

Performs one-hot encoding of atom types and prepares other data-processing.

`pad_data`(*data*, *max_length*)

Pads molecule so that all have the same size
and can be fed through batch_loader

`read_xyz_file`(*file_path*)

Reads an XYZ file and returns its data.

`replace_atom_types_with_ohe_vectors`(*molecule_list*)

Replaces atom types in molecule data with their one-hot encoded vectors.

`data.Dataset_W93.dataset_reaction_graph.`**`bond_to_edge`**(*h*, *edge_index*)

Generate bond information between atoms in a reactant and product configuration.

This function takes atom information and edge indices to create bond information for a reaction graph based on reactant and product configurations.

Parameters

- **h** (*torch.Tensor*) – Tensor containing atom and configuration information. Shape: [num_atoms, num_features]
- **edge_index** (*torch.Tensor*) – Tensor containing edge indices of the graph. Shape: [2, num_edges]

Returns

Tensors containing bond information for reactant and product edges.
Each element represents a bond type: 1 for a bond present, and 0 for no bond present.

Return type
torch.Tensor, torch.Tensor

Note:

- The input tensor h contains atom information and configuration details. It is structured as follows:
  - h[:, :4] contains atom type encodings.
  - h[:, 4:7] contains reactant atom coordinates.
  - h[:, 7:10] contains product atom coordinates.
- The edge_index tensor contains pairs of atom indices representing edges in the graph.
- The function calculates bond information based on atom types and distances, using predefined bond lengths for various atom combinations.
- Bond types are determined by comparing distances between atoms with predefined bond lengths.
- The function is currently hardcoded for hydrogen atoms and assumes no context.

`data.Dataset_W93.dataset_reaction_graph.`**`get_adj_matrix_no_batch`**(*n_nodes*)

Get the adjacency matrix for a fully connected graph with the specified number of nodes.

This function returns the adjacency matrix as edge indices (rows and columns) for a graph with the given number of nodes. If the adjacency matrix for the specified number of nodes has already been generated, it is retrieved; otherwise, the function recursively generates the adjacency matrix.

Parameters
n_nodes (*int*) – Number of nodes in the graph.

Returns

A list containing two LongTensors representing the rows
and columns of the edge indices of the adjacency matrix.

Return type
list of torch.LongTensor

`data.Dataset_W93.dataset_reaction_graph.`**`get_bond_type`**(*atom_1_OHE*, *atom_2_OHE*, *distances*)

This function takes two one-hot encoded representations of atoms (atom_1_OHE and atom_2_OHE) and a list of distances between atom pairs. It calculates the bond types between these atoms based on their types and the provided distances.

Parameters

- **`atom_1_OHE`** (*`torch.Tensor`*) – One-hot encoded representation of the first atom.
- **`atom_2_OHE`** (*`torch.Tensor`*) – One-hot encoded representation of the second atom.
- **`distances`** (*`list`*) – List of distances between atom pairs.

Returns

A tensor containing the bond types between the atom pairs.
Each element in the tensor represents a bond type: 1 for a bond present, and 0 for no bond present.

Return type
torch.Tensor

2.4 TX1 Dataset:

`class` `data.Dataset_TX1.TX1_dataloader.`**`Dataloader`**(*hdf5_file*, *datasplit='data'*, *only_final=False*)

Iterates through an HDF5 dataset, providing access to reaction data.

This class allows iteration through an HDF5 dataset containing reaction data. It provides the ability to loop through different configurations for each reaction and return the data in dictionaries. The class can also be configured to iterate only through reactants, products, and transition states.

Parameters

- **`hdf5_file`** (*`str`*) – Path to the HDF5 file containing the dataset.
- **`datasplit`** (*`str, optional`*) – Specifies the data split to iterate through (e.g., “data”, “train”, “val”, “test”). (default: “data”)
- **`only_final`** (*`bool, optional`*) – If True, the iterator will only loop through reactants, products, and transition states. (default: False)

`hdf5_file`

Path to the HDF5 file containing the dataset.

Type
str

`only_final`

Flag indicating whether to iterate only through reactants, products, and transition states.

Type
bool

`datasplit`

Specifies the data split being iterated through.

Type
str

Returns
A dictionary containing data for each configuration.

Return type
dict

`data.Dataset_TX1.TX1_dataloader.`**`generator`**`(`*formula*, *rxn*, *grp*`)`

# noqa Iterate through an HDF5 group, processing energy and force data.

This generator function iterates through an HDF5 group containing energy, force, atomic numbers, and position data for a molecular system. It processes this data to calculate various properties and returns dictionaries with the calculated information for each step.

Parameters

- **`formula`** (*`str`*) – Chemical formula of the molecule.
- **`rxn`** (*`str`*) – Reaction identifier or description.
- **`grp`** (*`h5py.Group`*) – The HDF5 group containing the data.

Returns
A dictionary containing calculated properties for each step.

Return type
dict

`data.Dataset_TX1.TX1_dataloader.`**`get_molecular_reference_energy`**`(`*atomic_numbers*`)`

Calculate the molecular reference energy based on atomic numbers.

This function takes a list of atomic numbers and calculates the molecular reference energy by summing up the individual reference energies of the atoms.

Parameters
`atomic_numbers` (*`list`*) – List of atomic numbers of the atoms in the molecule.

Returns
The calculated molecular reference energy.

Return type
float

`class data.Dataset_TX1.dataset_TX1_class.`**`TX1_dataset`**`(`*directory='data/Dataset_TX1/Transition1x.h5'*, *split='data'*`)`

`get(`*idx*`)`

Gets the data object at index `idx`.

`len()`

Returns the number of graphs stored in the dataset.

`setup()`

Setup the Data and the node masks

2.5 RGD1 Dataset:

`data.Dataset_RGD1.parse_data.`**`checking_duplicates`**`()`

Check for duplicate reactants and products based on SMILES strings.

This function reads reaction data from an HDF5 file, parses the SMILES strings of reactants and products, and identifies duplicate reactions. It keeps track of duplicate reaction information and their counts.

Returns
None

`data.Dataset_RGD1.parse_data.`**`main`**`()`

Main Function to save reaction geometries in required XYZ format.

This function reads a dataset containing reaction geometries from an HDF5 file, parses the elements and geometries of reactants, products, and transition states, and saves them as XYZ files.

Returns
None

`data.Dataset_RGD1.parse_data.`**`save_reactions_with_multiple_ts`**`(`*save_even_single=False*`)`

Save reactions with multiple TS conformers to separate directories.

This function reads reaction data from an HDF5 file along with activation energy data from a CSV file. It identifies reactions with multiple transition state (TS) conformers based on identical reaction SMILES strings. It then organizes and saves these reactions and their geometries into separate directories.

Parameters
`save_even_single` (*`bool, optional`*) – Save reactions with only one TS conformer as well. (default: False)

Returns
None

`data.Dataset_RGD1.parse_data.`**`save_xyz_file`**`(`*path*, *atoms*, *coordinates*`)`

Save XYZ file with atomic elements and corresponding coordinates.

Parameters

- **`file_path`** (*`str`*) – Path to the output XYZ file.
- **`elements`** (*`list`*) – List of atomic symbols.
- **`coordinates`** (*`numpy.ndarray`*) – Array of atomic coordinates.

Returns
None

`class` `data.Dataset_RGD1.RGD1_dataset_class.`**`RGD1_TS`**`(`*directory='data/Dataset_RGD1/data/Clean_Geometries/'*, *remove_hydrogens=False*, *plot_distribution=False*`)`

`atom_count``()`

Counts the occurrences of different atom types in the reactions.

`convert_to_float``(`*data*`)`

Convert nested list elements to floats.

`convert_to_list``(`*data*`)`

Convert nested list elements to Lists.

`count_data()`

Counts the number of reactions in the dataset.

`create_data_array()`

Creates data arrays with context information based on the selected context or without context.

`delete_centre_gravity()`

Removes the center of gravity from molecule coordinates. Needed to keep model roto-translation invariant

`extract_data(`*reaction_number*`)`

Extracts reactant, product, and transition state information from specified reaction.

`generate_one_hot_encodings(`*num_of_atoms*`)`

Generates one-hot encodings for atom types.

`get(`*idx*`)`

Retrieves a data sample and its corresponding node mask.

`get_keys_from_value(`*search_value*`)`

Retrieves atom types based on their one-hot encoded vectors.

`len()`

Returns the length of the dataset.

`load_data()`

Loads the dataset by extracting data from reaction files and performing preprocessing steps.

`one_hot_encode()`

Performs one-hot encoding of atom types and prepares other data-processing.

`pad_data(`*data*, *max_length*`)`

Pads molecule so that all have the same size

and can be fed through batch_loader

`plot_molecule_size_distribution()`

Plots the distribution of molecule sizes in the dataset.

`read_xyz_file(`*file_path*`)`

Reads an XYZ file and returns its data.

`replace_atom_types_with_ohe_vectors(`*molecule_list*`)`

Replaces atom types in molecule data with their one-hot encoded vectors.

CHAPTER

THREE

THE EQUIVARIANT GRAPH NEURAL NETWORK:

Background about equivariance and invariance as well as the architecture of the model is described in Sacha's thesis.

The main EGNN files used for the diffusion process are the dynamics files:

3.1 Dynamics without Graphs:

class src.EGNN.egnn.**EGNN**(*in_node_nf*, *in_edge_nf*, *hidden_nf*, *device='cpu'*, *act_fn=SiLU()*, *n_layers=3*, *attention=False*, *out_node_nf=None*, *tanh=False*, *coords_range=5*, *norm_constant=1*, *inv_sublayers=2*, *sin_embedding=False*, *normalization_factor=100*, *aggregation_method='sum'*)

Equivariant Graph Neural Network (EGNN) module.

Parameters

- **in_node_nf** (*int*) – Number of input node features.
- **in_edge_nf** (*int*) – Number of input edge features.
- **hidden_nf** (*int*) – Number of hidden node features.
- **device** (*str*) – Device to run the module on.
- **act_fn** (*nn.Module*) – Activation function applied to MLP layers.
- **n_layers** (*int*) – Number of layers in the EGNN.
- **attention** (*bool*) – Whether to apply attention mechanism.
- **norm_diff** (*bool*) – Whether to normalize coordinate differences.
- **out_node_nf** (*int*) – Number of output node features.
- **tanh** (*bool*) – Whether to apply hyperbolic tangent to coordinates.
- **coords_range** (*float*) – Range of coordinate values.
- **norm_constant** (*int*) – Normalization constant for coordinate differences.
- **inv_sublayers** (*int*) – Number of layers in the EquivariantBlock.
- **sin_embedding** (*bool*) – Whether to use sinusoid embeddings.
- **normalization_factor** (*int*) – Normalization factor for aggregation.
- **aggregation_method** (*str*) – Aggregation method for aggregating edge information.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*h*, *x*, *edge_index*, *node_mask=None*, *edge_mask=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`class src.EGNN.egnn.`**`EquivariantBlock`**(*hidden_nf*, *edge_feat_nf=2*, *device='cpu'*, *act_fn=SiLU()*, *n_layers=2*, *attention=True*, *tanh=False*, *coords_range=15*, *norm_constant=1*, *sin_embedding=None*, *normalization_factor=100*, *aggregation_method='sum'*)

Equivariant Block module for EGNN.

Parameters

- **hidden_nf** (*int*) – Number of hidden node features.
- **edge_feat_nf** (*int*) – Number of edge feature dimensions.
- **device** (*str*) – Device to run the module on.
- **act_fn** (*nn.Module*) – Activation function applied to MLP layers.
- **n_layers** (*int*) – Number of layers in the block.
- **attention** (*bool*) – Whether to apply attention mechanism.
- **norm_diff** (*bool*) – Whether to normalize coordinate differences.
- **tanh** (*bool*) – Whether to apply hyperbolic tangent to coordinates.
- **coords_range** (*float*) – Range of coordinate values.
- **norm_constant** (*int*) – Normalization constant for coordinate differences.
- **sin_embedding** (*nn.Module*) – Sinusoid embedding for distances.
- **normalization_factor** (*int*) – Normalization factor for aggregation.
- **aggregation_method** (*str*) – Aggregation method for aggregating edge information.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*h*, *x*, *edge_index*, *node_mask=None*, *edge_mask=None*, *edge_attr=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`class src.EGNN.egnn.`**`EquivariantUpdate`**(*hidden_nf*, *normalization_factor*, *aggregation_method*, *edges_in_d=1*, *act_fn=SiLU()*, *tanh=False*, *coords_range=10.0*)

Equivariant Update module for EGNN.

Parameters

- **hidden_nf** (*int*) – Number of hidden node features.

- **normalization_factor** (*int*) – Normalization factor for aggregation.
- **aggregation_method** (*str*) – Aggregation method for aggregating edge information. # noqa
- **edges_in_d** (*int*) – Dimensionality of additional edge attributes.
- **act_fn** (*nn.Module*) – Activation function applied to MLP layers.
- **tanh** (*bool*) – Whether to apply hyperbolic tangent to coordinates.
- **coords_range** (*float*) – Range of coordinate values.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*h*, *coord*, *edge_index*, *coord_diff*, *edge_attr=None*, *node_mask=None*, *edge_mask=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class src.EGNN.egnn.**GCL**(*input_nf*, *output_nf*, *hidden_nf*, *normalization_factor*, *aggregation_method*, *edges_in_d=0*, *nodes_att_dim=0*, *act_fn=SiLU()*, *attention=False*)

Graph Convolutional Layer (GCL) module.

Parameters

- **input_nf** (*int*) – Number of input node features.
- **output_nf** (*int*) – Number of output node features.
- **hidden_nf** (*int*) – Number of hidden node features.
- **normalization_factor** (*int*) – Normalization factor for aggregation.
- **aggregation_method** (*str*) – Aggregation method for aggregating edge information.
- **edges_in_d** (*int*) – Dimensionality of additional edge attributes.
- **nodes_att_dim** (*int*) – Dimensionality of additional node attributes.
- **act_fn** (*nn.Module*) – Activation function applied to MLP layers.
- **attention** (*bool*) – Whether to apply attention mechanism.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*h*, *edge_index*, *edge_attr=None*, *node_attr=None*, *node_mask=None*, *edge_mask=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class `src.EGNN.egnn.`**`SinusoidsEmbeddingNew`**(*max_res=15.0*, *min_res=0.0075*, *div_factor=4*)

Sinusoidal embedding module for EGNN.

Parameters

- **`max_res`** (*`float`*) – Maximum resolution.
- **`min_res`** (*`float`*) – Minimum resolution.
- **`div_factor`** (*`int`*) – Division factor.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`src.EGNN.egnn.`**`coord2diff`**(*x*, *edge_index*, *norm_constant=1*)

Converts node coordinates to differences and radial distances.

Parameters

- **`x`** (*`Tensor`*) – Node coordinates.
- **`edge_index`** (*`Tensor`*) – Edge indices.
- **`norm_constant`** (*`int, optional`*) – Normalization constant for coordinate differences.

Returns

Radial distances. Tensor: Coordinate differences.

Return type

Tensor

`src.EGNN.egnn.`**`get_adj_matrix`**(*n_nodes*, *batch_size*, *device*)

Generates adjacency matrix for a batch of graphs with a given number of nodes.

Parameters

- **`n_nodes`** (*`int`*) – Number of nodes in each graph.
- **`batch_size`** (*`int`*) – Batch size.
- **`device`** (*`str`*) – Device to run the operation on.

Returns

Edge information (edge indices and attributes).

Return type

Tuple

`src.EGNN.egnn.`**`get_edges`**(*n_nodes*)

Generates edges for a graph with a given number of nodes.

Parameters

`n_nodes` (*`int`*) – Number of nodes.

Returns
List of rows and columns representing edges.

Return type
List

`src.EGNN.egnn.`**`get_edges_batch`**`(`*n_nodes*, *batch_size*`)`

Generates edges for a batch of graphs with a given number of nodes.

Parameters

- **`n_nodes`** (*`int`*) – Number of nodes in each graph.
- **`batch_size`** (*`int`*) – Batch size.

Returns
Edge information (edge indices and attributes).

Return type
Tuple

`src.EGNN.egnn.`**`unsorted_segment_sum`**`(`*data*, *segment_ids*, *num_segments*, *normalization_factor: int*, *aggregation_method: str = 'sum'*`)`

Performs unsorted segment sum operation with normalization.

Parameters

- **`data`** (*`Tensor`*) – Data to be segmented.
- **`segment_ids`** (*`Tensor`*) – Indices indicating segments. rows
- **`num_segments`** (*`int`*) – Number of segments.
- **`normalization_factor`** (*`int`*) – Normalization factor for aggregation.
- **`aggregation_method`** (*`str`*) – Aggregation method ('sum' or 'mean').

Returns
Result of the operation.

Return type
Tensor

`class src.EGNN.dynamics.`**`EGNN_dynamics_QM9`**`(`*in_node_nf: int*, *n_dims: int = 3*, *out_node: int = 3*, *hidden_nf: int = 64*, *device: str = 'cpu'*, *act_fn=SiLU()*, *n_layers: int = 4*, *attention: bool = True*, *condition_time: bool = True*, *tanh: bool = False*, *norm_constant: int = 0*, *inv_sublayers: int = 2*, *sin_embedding: bool = False*, *normalization_factor: int = 100*, *aggregation_method: str = 'sum'*`)`

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`forward``(`*t*, *xh*, *node_mask*, *edge_mask*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

3.2 Dynamics with Graphs:

The main difference being that the EGNN takes extra inputs of edge attributes.

`class src.EGNN.egnn_with_graph.`**`EGNN_with_bond`**(*in_node_nf*, *in_edge_nf*, *hidden_nf*, *device='cpu'*, *act_fn=SiLU()*, *n_layers=3*, *attention=False*, *out_node_nf=None*, *tanh=False*, *coords_range=5*, *norm_constant=1*, *inv_sublayers=2*, *sin_embedding=False*, *normalization_factor=100*, *aggregation_method='sum'*)

Enhanced Graph Neural Network (EGNN) with Bond Information.

This class defines an EGNN model with support for bond information in addition to node and edge features. It applies equivariant graph neural network layers to process node and edge features, considering bond information and interaction between nodes. The model supports multiple layers, attention mechanisms, and various configuration options to customize the behavior of the network.

Parameters

- **`in_node_nf`** (*`int`*) – Dimensionality of input node features.
- **`in_edge_nf`** (*`int`*) – Dimensionality of input edge features.
- **`hidden_nf`** (*`int`*) – Dimensionality of hidden features in the equivariant blocks.
- **`device`** (*`str, optional`*) – Device to which the model will be moved (default is "cpu").
- **`act_fn`** (*`torch.nn.Module, optional`*) – Activation function to be used in the equivariant blocks (default is nn.SiLU()).
- **`n_layers`** (*`int, optional`*) – Number of equivariant blocks (default is 3).
- **`attention`** (*`bool, optional`*) – Whether to use attention mechanism in the equivariant blocks (default is False).
- **`out_node_nf`** (*`int, optional`*) – Dimensionality of output node features. If not provided, it will be set to in_node_nf.
- **`tanh`** (*`bool, optional`*) – Whether to apply tanh activation to the output node features (default is False).
- **`coords_range`** (*`float, optional`*) – Range of coordinate values for positional encodings (default is 5).
- **`norm_constant`** (*`float, optional`*) – Normalization constant for positional encodings (default is 1).
- **`inv_sublayers`** (*`int, optional`*) – Number of layers in the equivariant block's inverse feature extraction (default is 2).
- **`sin_embedding`** (*`bool, optional`*) – Whether to use sinusoidal embeddings for edge features (default is False).
- **`normalization_factor`** (*`int, optional`*) – Normalization factor for equivariant block's aggregation (default is 100).
- **`aggregation_method`** (*`str, optional`*) – Method for aggregating node information (default is "sum").

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*h*, *x*, *edge_index*, *node_mask=None*, *edge_mask=None*, *edge_attributes=None*)

Forward pass of the EGNN_with_bond model.

Parameters

- **h** (*torch.Tensor*) – Input node features.
- **x** (*torch.Tensor*) – Input edge features.
- **edge_index** (*torch.Tensor*) – Graph edge indices.
- **node_mask** (*torch.Tensor, optional*) – Mask for node features (default is None).
- **edge_mask** (*torch.Tensor, optional*) – Mask for edge features (default is None).
- **edge_attributes** (*torch.Tensor, optional*) – Additional attributes associated with edges (default is None).

Returns

Output node features. x (torch.Tensor): Output edge features.

Return type

h (torch.Tensor)

class src.EGNN.dynamics_with_graph.**EGNN_dynamics_graph**(*in_node_nf: int*, *in_edge_nf: int*, *n_dims: int = 3*, *out_node: int = 3*, *hidden_nf: int = 64*, *device: str = 'cpu'*, *act_fn=SiLU()*, *n_layers: int = 4*, *attention: bool = True*, *condition_time: bool = True*, *tanh: bool = False*, *norm_constant: int = 0*, *inv_sublayers: int = 2*, *sin_embedding: bool = False*, *normalization_factor: int = 100*, *aggregation_method: str = 'sum'*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*t*, *xh*, *node_mask*, *edge_mask*, *edge_attributes*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

3.3 Utility Functions:

The Utility functions used in the EGNN process are outlined bellow:

src.EGNN.utils.**assert_correctly_masked**(*variable*, *node_mask*)

Asserts that the selected elements in the input tensor *variable* are correctly masked out based on the provided node mask.

Parameters

- **variable** (*torch.Tensor*) – The input tensor of shape (batch_size, num_nodes, . . .).
- **node_mask** (*torch.Tensor*) – A mask tensor of shape (batch_size, num_nodes) with values 0 or 1, indicating which nodes' values should be masked out.

Raises
`AssertionError` – If any unmasked elements are found in the masked-out regions.

`src.EGNN.utils.`**`assert_mean_zero_with_mask`**(*x*, *node_mask*, *eps=1e-10*)

Asserts that the mean of selected elements in the input tensor *x* is approximately zero, considering the mask provided by *node_mask*.

Parameters

- **x** (*torch.Tensor*) – The input tensor of shape (batch_size, num_nodes, . . .).
- **node_mask** (*torch.Tensor*) – A mask tensor of shape (batch_size, num_nodes) with values 0 or 1, indicating which nodes' values to consider for mean calculation.
- **eps** (*float, optional*) – A small constant to avoid division by zero in relative error calculation.

Raises
`AssertionError` – If the mean of selected elements is not approximately zero.

`src.EGNN.utils.`**`remove_mean`**(*x*)

Remove the mean along a specified dimension from the input tensor.

Parameters
x (*torch.Tensor*) – The input tensor.

Returns
A new tensor with the mean along the specified dimension subtracted.

Return type
torch.Tensor

`src.EGNN.utils.`**`remove_mean_with_mask`**(*x*, *node_mask*)

Remove the mean along a specified dimension from the input tensor.

Parameters
x (*torch.Tensor*) – The input tensor.

Returns
A new tensor with the mean along the specified dimension subtracted.

Return type
torch.Tensor

`src.EGNN.utils.`**`setup_device`**()

Set up the computing device for PyTorch operations.

Returns
The selected computing device (GPU if available, otherwise CPU).

Return type
torch.device

`src.EGNN.utils.`**`sum_except_batch`**(*x*)

Sums the elements of each tensor in the input *x*, except the batch dimension.

Parameters
x (*torch.Tensor*) – Input tensor of shape (batch_size, *).

Returns

A tensor of shape (batch_size,) containing the sum of elements in each tensor of *x*,
excluding the batch dimension.

Return type
torch.Tensor

CHAPTER

FOUR

EXAMPLE FUNCTIONS AND CLASSES FROM THE PACKAGE INCLUDE:

See *src.evaluate_samples* for details.

src.evaluate_samples.**calc_cov_mat**(*results_matrix*, *cov_threshold=0.1*)

Calculate COV and MAT scores based on D-MAE matrix.

Parameters

- **results_matrix** (*np.ndarray*) – D-MAE matrix.
- **cov_threshold** (*float, optional*) – COV threshold. Defaults to 0.1.

Returns

Calculated MAT-R mean, median, and COV-R scores.

Return type

tuple

src.evaluate_samples.**calculate_DMAE**(*gen_mol*, *true_mol*)

Calculate D-MAE between inter-atomic distance matrices.

Parameters

- **gen_mol** (*list*) – Inter-atomic distance matrix of generated molecule.
- **true_mol** (*list*) – Inter-atomic distance matrix of true molecule.

Returns

D-MAE value.

Return type

float

src.evaluate_samples.**calculate_best_rmse**(*gen_mol*, *ref_mol*, *max_iters=100000*, *use_hydrogens=False*)

Calculate Best RMSD between RDKit Molecule Objects.

Parameters

- **gen_mol** (*Chem.Mol*) – RDKit molecule object representing generated molecule.
- **ref_mol** (*Chem.Mol*) – RDKit molecule object representing reference molecule.
- **max_iters** (*int, optional*) – Maximum atom matches. Defaults to 100_000.
- **use_hydrogens** (*bool, optional*) – True to include hydrogens. Defaults to False.

Returns

Best RMSD value.

Return type
float

`src.evaluate_samples.`**`calculate_distance_matrix`**(*coordinates*)

Calculate pairwise distance matrix from 3D coordinates.

Parameters
`coordinates` (*`list`*) – List of 3D coordinates for each atom.

Returns
Pairwise distance matrix.

Return type
np.ndarray

`src.evaluate_samples.`**`create_lists`**(*original_path*, *RMSD=False*)

Create lists of true and generated molecules from the given path.

Parameters

- **`original_path`** (*`str`*) – Path to the original directory containing molecule files.
- **`RMSD`** (*`bool, optional`*) – True if RMSD format, False if standard XYZ format.

Returns
Two lists containing RDKit molecule objects.

Return type
tuple

`src.evaluate_samples.`**`create_table`**(*true_mols*, *gen_mols*, *max_iters=1*, *metric='RMSD'*)

Create comparison table between molecules.

Parameters

- **`true_mols`** (*`list`*) – List of true molecule RDKit objects.
- **`gen_mols`** (*`list`*) – List of lists containing generated molecule RDKit objects.
- **`max_iters`** (*`int`*) – Maximum atom matches for RMSD calculation.
- **`metric`** (*`str`*) – Metric choice, “RMSD” or “DMAE”.

Returns
DataFrame of comparison metrics.

Return type
pd.DataFrame

`src.evaluate_samples.`**`evaluate`**(*sample_path*, *evaluation_type*, *cov_threshold=0.1*)

Evaluate generated samples using RMSE or D-MAE metrics.

Parameters

- **`sample_path`** (*`str`*) – Path to sample directory.
- **`evaluation_type`** (*`str`*) – Metric choice, “RMSD” or “DMAE”.
- **`cov_threshold`** (*`float, optional`*) – COV threshold. Defaults to 0.1.

`src.evaluate_samples.`**`get_paths`**(*sample_path*)

Load molecule files from a sample path and organize them into true and generated samples.

Parameters
`sample_path` (*`str`*) – Path to the sample directory containing molecule files.

Returns
Two lists containing true and generated sample file paths.

Return type
tuple

`src.evaluate_samples.`**`import_xyz_file`**(*molecule_path*, *RMSD=False*)

Import an XYZ file as an RDKit molecule object.

Parameters

- **`molecule_path`** (*`str`*) – File path of the XYZ file to be imported.
- **`RMSD`** (*`bool, optional`*) – True if RMSD format, False if standard XYZ format.

Returns
RDKit molecule object or None if loading fails.

Return type
Chem.Mol or None

PYTHON MODULE INDEX

INDEX