TS-DiffuGen

Release 2023

Sacha Raffaud

Aug 31, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to the TS-DiffuGen documentation. This guide provides an overview of the TS-DiffuGen package, which is designed for optimizing reaction transition states using diffusion models.

CHAPTER

TWO

DATASETS

Explore the dataset classes in the data directory, which contain various datasets used for training and evaluation.

2.1 Setting up the dataset files

This section covers the process of setting up dataset files using the *Dataset_W93* module. It provides functions to prepare the necessary data for the models.

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

The *Dataset_W93* module also includes a simple dataset class without graph information. This section details its functionalities.

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

For scenarios where graph structures are used in training, this section introduces the *Dataset_W93* module's dataset class that includes reaction graph information.

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

Explore the *Dataset_TX1* module for loading the TX1 dataset. It includes functionalities using a dataset classes that loads data from a dataloader.

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

Learn about the *Dataset_RGD1* module that provides functions to parse data for the RGD1 dataset. The associated dataset class is also explained here.

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

Theoretical background of equivariance, invariance, and the model architecture can be refered from Sacha's thesis. The EGNN architecture was used as a denoising framework. For this it was implemented within dynamics classes that wrap around the EGNN to make it suitable for a denoising task.

3.1 Dynamics without Graphs

Learn about the core elements of the Equivariant Graph Neural Network (EGNN) without graph inputs.

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

This part focuses on incorporating edge attributes into EGNN's dynamics.

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.

CHAPTER

FOUR

UTILITY FUNCTIONS

The EGNN process involves several utility functions. This section provides insights into these functions.

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

FIVE

THE DIFFUSION MODEL BACKBONE

This section describes the functions used for the equivariant diffusion model. One diffusion class includes edge attributes and another does not.

class src.Diffusion.equivariant_diffusion.**DiffusionModel**(*dynamics:* EGNN_dynamics_QM9, *in_node_nf: int*, *n_dims: int = 3*, *device: str = 'cpu'*, *timesteps: int = 1000*, *noise_schedule: str = 'cosine'*, *noise_precision: float = 0.0001*)

The E(n) Diffusion Module.

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

SNR(*gamma*)

Computes signal to noise ratio (alpha^2/sigma^2) given gamma.

alpha(*gamma*, *target_tensor*)

Computes alpha given gamma.

The alpha variable controls how much of the original sample is kept at a given timestep.

check_issues_norm_values(*num_stdevs=8*)

Checks that the number of timesteps is large enough to converge to white noise.

compute_error(*net_out*, *eps*)

Computes the error (MSE) between the true and predicted noise

compute_loss(*x*, *h*, *node_mask*, *edge_mask*)

Computes the loss using simple MSE loss.

compute_x_pred(*net_out*, *zt*, *gamma_t*)

Commputes x_pred, i.e. the most likely prediction of x.

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

Computes the loss (type l2 or NLL) if training. And if eval then always computes NLL.

inflate_batch_array(*array*, *target*)

Inflates the batch array (array) with only a single axis (i.e. shape = (batch_size,), or possibly more empty axes (i.e. shape (batch_size, 1, ..., 1)) to match the target shape.

log_constants_p_x_given_z0(*x*, *node_mask*)

Computes p(x|z0).

phi(*x*, *t*, *node_mask*, *edge_mask*)

Function to get Predicted noise from denoising model.

sample(*h*, *n_samples*, *n_nodes*, *node_mask*, *edge_mask*, *context_size=0*)

Draw samples from the generative model.

sample_chain(*h*, *n_samples*, *n_nodes*, *node_mask*, *edge_mask*, *keep_frames=None*, *context_size=0*)

Draw samples from the generative model, keep the intermediate states for visualization purposes.

sample_normal(*mu*, *sigma*, *node_mask*)

Samples from a Normal distribution and returns the noisy X_t sample.

sample_p_xh_given_z0(*z0*, *node_mask*, *edge_mask*)

Samples x ~ p(x|z0).

sample_p_zs_given_zt(*s*, *t*, *zt*, *node_mask*, *edge_mask*)

Samples from zs ~ p(zs | zt). Only used during sampling.

sample_position(*n_samples*, *n_nodes*, *node_mask*)

Samples mean-centered normal noise for z_x.

sigma(*gamma*, *target_tensor*)

Computes sigma given gamma.

The sigma variable controls how much noise is added at a given timestep.

sigma_and_alpha_t_given_s(*gamma_t: Tensor*, *gamma_s: Tensor*, *target_tensor: Tensor*)

Computes sigma t given s, using gamma_t and gamma_s. Used during sampling.

These are defined as:

alpha t given s = alpha t / alpha s, sigma t given s = sqrt(1 - (alpha t given s) ^2).

class src.Diffusion.equivariant_diffusion.**DiffusionModel_graph**(*dynamics:* EGNN_dynamics_graph, *in_node_nf: int*, *n_dims: int = 3*, *device: str = 'cpu'*, *timesteps: int = 1000*, *noise_schedule: str = 'cosine'*, *noise_precision: float = 0.0001*)

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

compute_loss(*x*, *h*, *node_mask*, *edge_mask*, *edge_attributes*)

Computes the the simple loss (MSE).

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

Computes the loss (type l2 or NLL) if training. And if eval then always computes NLL. # noqa

phi(*x*, *t*, *node_mask*, *edge_mask*, *edge_attributes*)

Function to get Predicted noise from denoising model.

sample(*h*, *edge_attributes*, *n_samples*, *n_nodes*, *node_mask*, *edge_mask*)

Draw samples from the generative model.

sample_p_xh_given_z0(*z0*, *node_mask*, *edge_mask*, *edge_attributes=None*)

Samples x ~ p(x|z0).

sample_p_zs_given_zt(*s*, *t*, *zt*, *node_mask*, *edge_mask*, *edge_attributes=None*)

Samples from zs ~ p(zs | zt). Only used during sampling.

src.Diffusion.equivariant_diffusion.**get_node_features**(*remove_hydrogens=False*, *include_context=False*, *no_product=False*)

Determine the number of node features based on input options.

This function calculates the number of node features for a given molecular node based on the input options provided. The node features are determined by the presence or absence of hydrogen atoms, inclusion of context information, and the no_product option.

Parameters

- **remove_hydrogens** (*bool, optional*) – If True, hydrogens are removed, and the resulting node features will not include a 4D one-hot encoding for hydrogen positions.
- **include_context** (*bool, optional*) – If True, an additional dimension for context information is added to the node features.
- **no_product** (*bool, optional*) – If True, a few dimensions are excluded from the node features to account for the no_product option.

Returns
The calculated number of node features based on the provided options.

Return type
int

5.1 Noise Schedule for Diffusion

Learn about the noise schedule used in the diffusion process. A separate class defines the noise schedule to enhance modularity.

class src.Diffusion.noising.**PredefinedNoiseSchedule**(*noise_schedule*, *timesteps*, *precision*)

Predefined noise schedule.

This class creates a lookup array for predefined (non-learned) noise schedules based on the input noise_schedule. # noqa The class supports three predefined noise schedules: 'cosine', 'polynomial_power', and 'sigmoid_steepness'.

Parameters

- **noise_schedule** (*str*) – The predefined noise schedule to use. Possible values are: 'cosine', 'polynomial_<power>', or 'sigmoid_<steepness>'.
- **timesteps** (*int*) – The total number of time steps for the noise schedule.
- **precision** (*float*) – A parameter to control the noise schedule's behavior. Used in the 'polynomial_power' and 'sigmoid_steepness' schedules.

Raises
ValueError – If the noise_schedule provided is not one of the predefined options.

gamma

The parameter storing the lookup array for the noise schedule.

Type
torch.nn.Parameter

Note: The class does not use learned parameters; instead, it creates predefined noise schedules using the specified options.

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

forward(*t*)

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.Diffusion.noising.**alpha**(*gamma*, *target_tensor*)

Compute the alpha function given gamma.

The alpha function is defined as the square root of the sigmoid of negative gamma.

Parameters

- **gamma** (*torch.Tensor*) – Input tensor containing gamma values.
- **target_tensor** (*torch.Tensor*) – The target tensor with the desired shape.

Returns

The output tensor representing the computed alpha values with the same shape as the target tensor.

Return type

torch.Tensor

src.Diffusion.noising.**clip_noise_schedule**(*alphas2*, *clip_value=0.001*)

Clips the noise schedule given by alpha^2 to improve stability during sampling.

Parameters

- **alphas2** (*numpy.ndarray*) – 1-dimensional array containing the values of alpha^2 for the noise schedule.
- **clip_value** (*float, optional*) – The lower bound to which alpha_t / alpha_t-1 will be clipped. Defaults to 0.001.

Returns

A 1-dimensional array containing the clipped noise schedule, where alpha_t / alpha_t-1 is clipped to be greater than or equal to 'clip_value'.

Return type

numpy.ndarray

src.Diffusion.noising.**cosine_beta_schedule**(*timesteps*, *s=0.008*, *raise_to_power: float = 1*)

Generate a cosine beta schedule as proposed in the paper: (https://openreview.net/forum?id=-NEXDKk8gZ)

Parameters

- **timesteps** (*int*) – The total number of time steps for the beta schedule.
- **s** (*float, optional*) – A scaling parameter to control the schedule's behavior. Defaults to 0.008.
- **raise_to_power** (*float, optional*) – A power value to raise the beta values to. Defaults to 1.

Returns

A 1-dimensional array representing the cosine beta schedule.

Return type
numpy.ndarray

src.Diffusion.noising.**inflate_batch_array**(*array*, *target*)

Inflate the batch array to match the target shape.

This function inflates the batch array 'array' with only a single axis (i.e., shape=(batch_size,), or possibly more empty axes (i.e., shape=(batch_size, 1, ..., 1)) to match the shape of the target tensor.

Parameters

- **array** (*torch.Tensor*) – The input batch array with a single axis (shape=(batch_size,)).
- **target** (*torch.Tensor*) – The target tensor with the desired shape.

Returns
The inflated batch array with the same shape as the target tensor.

Return type
torch.Tensor

src.Diffusion.noising.**polynomial_schedule**(*timesteps: int*, *s=0.0001*, *power=3.0*)

Generate a noise schedule based on a simple polynomial equation: 1 - (x / timesteps)^power.

Parameters

- **timesteps** (*int*) – The total number of time steps for the noise schedule.
- **s** (*float, optional*) – The minimum value added to the noise schedule. Defaults to 1e-4.
- **power** (*float, optional*) – The power value in the polynomial equation. Defaults to 3.

Returns
A 1-dimensional array representing the noise schedule, containing alpha^2 values.

Return type
numpy.ndarray

src.Diffusion.noising.**sigma**(*gamma*, *target_tensor*)

Compute the sigma function given gamma.

The sigma function is defined as the square root of the sigmoid of gamma.

Parameters

- **gamma** (*torch.Tensor*) – Input tensor containing gamma values.
- **target_tensor** (*torch.Tensor*) – The target tensor with the desired shape.

Returns
The output tensor representing the computed sigma values with the same shape as the target tensor.

Return type
torch.Tensor

src.Diffusion.noising.**sigmoid_beta_schedule**(*timesteps*, *raise_to_power: float = 1*, *S: float = 6*)

Generate a sigmoid beta schedule with optional power scaling.

The sigmoid schedule transitions from low to high values based on the sigmoid function.

Parameters

- **timesteps** (*int*) – The total number of time steps for the beta schedule.

- **raise_to_power** (*float, optional*) – A power value to raise the beta values to. Defaults to 1.
- **S** (*float, optional*) – Controls the steepness of the sigmoid function. Lower values make the function less steep, and it slowly transitions from low to high values. Higher values make the function steeper, and it quickly transitions from low to high values. Defaults to 6.

Returns
A 1-dimensional array representing the sigmoid beta schedule.

Return type
numpy.ndarray

Examples

- If you choose S = 1, you will be returned a linear noise schedule.

5.2 Utility Functions for Diffusion

This section covers utility functions that are shared between the different diffusion classes, contributing to code modularity.

src.Diffusion.utils.**assert_mean_zero**(*x*)

Asserts that the mean of each row in the input tensor *x* is approximately zero.

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

Raises
AssertionError – If the absolute value of the maximum mean is >= 1e-6.

src.Diffusion.utils.**random_rotation**(*x*, *h*)

Algorithm: 1. Input sample with coordinates x and node features H 2. Seperate coordinates of reactant, product and TS 1. Sample random angle Theta 2. Rotate Reactant, product and TS coordinates depending on rotation matrix and angle Theta: 3. Return rotated coordinates and same node features

Adapted from: https://github.com/ehoogeboom/e3_diffusion_for_molecules/blob/main/utils.py

Apply random rotations to 3D coordinates and corresponding reactant and product information.

This function performs random rotations on input 3D coordinates along with associated reactant and product information for data augmentation during training. It generates random rotation matrices around each axis, applies these rotations to the input data, and returns the rotated data.

Parameters

- **x** (*torch.Tensor*) – Input tensor of 3D coordinates with shape (batch_size, num_nodes, 3).
- **h** (*torch.Tensor*) – Input tensor containing additional information, including reactant and product details.

Returns
Rotated 3D coordinates tensor and updated information tensor.

Return type
torch.Tensor, torch.Tensor

src.Diffusion.utils.**sample_center_gravity_zero_gaussian_with_mask**(*size*, *device*, *node_mask*)

Generates a tensor following a Gaussian distribution, centered at zero, while considering a node mask.

Parameters

- **size** (*tuple*) – Size of the tensor to generate (batch_size, num_nodes, features).
- **device** (*torch.device*) – Device for tensor placement.
- **node_mask** (*torch.Tensor*) – Node mask of shape (batch_size, num_nodes).

Returns

Generated tensor with masked center gravity.

Return type

x_projected (torch.Tensor)

CHAPTER

SIX

PYTORCH LIGHTNING FOR SIMPLICITY

Explore how PyTorch Lightning is used to simplify the training, testing, and sampling processes. GPU usage is recommended due to the models' computational demands.

Simple diffusion and graph-based diffusion classes are explained here.

class src.lightning_setup.**LitDiffusionModel**(*dataset_to_use*, *in_node_nf*, *hidden_features*, *n_layers*, *device*, *lr*, *remove_hydrogens*, *test_sampling_number*, *save_samples*, *save_path*, *timesteps*, *noise_schedule*, *random_rotations=False*, *augment_train_set=False*, *include_context=False*, *learning_rate_schedule=False*, *no_product=False*, *batch_size=64*, *pytest_time=False*)

augment_reactants_products()

This function will augment the train set

configure_optimizers()

Setup Optimiser and learning rate scheduler if needed.

forward(*x*, *h*, *node_mask*, *edge_mask*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns

Your model's output

sample_and_test(*true_h*, *true_x*, *node_mask*, *edge_mask*, *folder_path*, *remove_hydrogens=False*, *device=None*)

Generate samples from the test set.

This function generates samples using the diffusion model and performs various processing steps on the provided data. It handles tasks such as inflating the input data, performing sampling, and saving the results.

Parameters

- **true_h** (*torch.Tensor*) – The true input graph data (node features and edge features).
- **true_x** (*torch.Tensor*) – The true target graph data (node features).
- **node_mask** (*torch.Tensor*) – The node mask for the input graph.
- **edge_mask** (*torch.Tensor*) – The edge mask for the input graph.
- **folder_path** (*str*) – Path to the folder where results will be saved.

- **remove_hydrogens** (*bool, optional*) – Flag indicating whether to remove hydrogens from data (default is False).
- **device** (*torch.device, optional*) – The device on which to perform computations (default is None).

Returns: None

`test_dataloader()`

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `test()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Note: If you don't need a test dataset and a *test_step()*, you don't need to implement this method.

`test_step(batch, batch_idx)`

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Returns

Any of.

- Any object or value
- `None` - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, *test_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the *test_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

train_dataloader()

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

> **Warning:** do not assign state in prepare_data

- `fit()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False
```

(continues on next page)

(continued from previous page)

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

val_dataloader()

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Note: If you don't need a validation dataset and a *validation_step()*, you don't need to implement this method.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value

- None - Validation will skip to the next batch

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

class src.lightning_setup.**LitDiffusionModel_With_graph**(*in_node_nf*, *in_edge_nf*, *hidden_features*, *n_layers*, *device*, *lr*, *test_sampling_number*, *save_samples*, *save_path*, *timesteps*, *noise_schedule*, *learning_rate_schedule=False*, *no_product=False*, *batch_size=64*, *pytest_time=False*)

configure_optimizers()

Setup Optimiser and learning rate scheduler if needed.

forward(*x*, *h*, *node_mask*, *edge_mask*, *edge_attributes*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns
Your model's output

test_dataloader()

An iterable or collection of iterables specifying test samples.

For more information about multiple dataloaders, see this section.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `test()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Note: If you don't need a test dataset and a *test_step()*, you don't need to implement this method.

test_step(*batch*, *batch_idx*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

Parameters

- **batch** – The output of your `DataLoader`.

- **batch_idx** – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Returns

Any of.

- Any object or value
- None - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, *test_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the *test_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`train_dataloader()`

An iterable or collection of iterables specifying training samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```
def __init__(self):
    super().__init__()
    self.automatic_optimization = False


# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

`val_dataloader``()`

An iterable or collection of iterables specifying validation samples.

For more information about multiple dataloaders, see this section.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`
- `prepare_data()`
- `setup()`

Note: Lightning tries to add the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Note: If you don't need a validation dataset and a *`validation_step()`*, you don't need to implement this method.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
```

(continues on next page)

(continued from previous page)

```
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

6.1 Saving and Writing Functions

Learn about the functions dedicated to saving and writing different components of the diffusion models and their results.

src.Diffusion.saving_sampling_functions.**return_xyz**(*sample*, *ohe_dictionary*, *remove_hydrogen=False*)

Sets up sample into XYZ format.

Parameters

- **sample** (*list of lists*) – A list of molecular structures (samples), where each sample is a list of atoms.# noqa Each atom is represented as a list with either 4 or 5 elements depending on the value of *remove_hydrogen*. If *remove_hydrogen* is False, each atom list contains the atom name (str), x-coordinate (float), y-coordinate (float), z-coordinate (float), and optionally an atom property (int). If *remove_hydrogen* is True, each atom list contains the atom name (str) and the three-dimensional coordinates (x, y, z) as floats.
- **there** (*Dictionary with atom-encoding in*) –
- **remove_hydrogen** (*bool, optional*) – A flag indicating whether hydrogen atoms should be removed from the output. If True, hydrogen atoms are excluded from the output, and the resulting XYZ format contains only non-hydrogen atoms. Default is False.

Returns

A list of molecular structures (samples) converted to the XYZ format.
Each sample is a list of atoms, where each atom is represented as a list containing: - The atom name (str) obtained from the one-hot encoding dictionary. - The x-coordinate (float) of the atom's position in 3D space. - The y-coordinate (float) of the atom's position in 3D space. - The z-coordinate (float) of the atom's position in 3D space. If *remove_hydrogen* is False, the atom list also contains: - An atom property (int) obtained from the original atom data.

Return type
list of lists

src.Diffusion.saving_sampling_functions.**write_xyz_file**(*data*, *filename*)

Writes Molecule Conformation data into an XYZ file in PyMol Format.# noqa

Parameters

- **data** (*list of lists*) – A list containing the conformation data of the molecule. Each element in the list represents an atom in the molecule and is itself a list with four elements: atom name (str), x-coordinate (str), y-coordinate (str), and z-coordinate (str).
- **filename** (*str*) – The name of the XYZ file to be created.

Returns
This function does not return anything. It writes the data into the specified file.

Return type
None

CHAPTER

SEVEN

TRAINING AND SAMPLING FROM THE TEST SET

This section covers the training and sampling procedures using config files located in the configs directory. Key functions are explained, offering insights into the process.

src.train_test.**main**(*args*, *pytest_time=False*)

Main function to manage training and testing of the diffusion model.

This function serves as the entry point to the script, allowing for both training and testing modes based on the provided command-line arguments. It sets up the necessary configurations, loads or trains the model, and handles the flow accordingly.

Parameters

- **args** (*argparse.Namespace*) – Command-line arguments parsed using argparse.
- **pytest_time** (*bool, optional*) – Flag indicating whether the code is being run in a pytest environment (default is False).

Returns

None

CHAPTER

EIGHT

EVALUATING SAMPLES

Understand how sample evaluation is performed in the context of molecule and conformation generation. Metrics such as COV (Coverage) and MAT (Matching) are discussed in Sacha's thesis and their code implementations are defined below:

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

CHAPTER

NINE

SUPPLEMENTARY SCRIPTS AND FUNCTIONS

This section includes additional scripts and functions that supplement the main package functionalities. It highlights the role of scripts like *sample_diffusion_chain* and others in generating GIFs or saving molecule images in .XYZ format using PyMol.

Additional scripts can also be found in the Extra_scripts directory.

PYTHON MODULE INDEX

d

s

INDEX

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W