GQCML Documentation

Release 0.1

Niels Billiet

Apr 12, 2021

CONTENTS

CHAPTER

ONE

GQCML.DATA_GENERATORS PACKAGE

1.1 Submodules

1.2 gqcml.data_generators.Huckel module

class gqcml.data_generators.Huckel.**HuckelSolver**

Bases: object

ONV(*eigenvals*, *N*, *N_a*, *N_b*)

Constructs the occupation number vectors that determines the filling of the electrons in the spin orbitals. This functions takes degenerate states into account as well

Arguments

param eigenvals (np.array) A 1D array containing the eigenvalues of the system

param N (int) The amount of spin orbitals in the system, i.e. the maximum number of electrons in the ONV

param .. math: N_a (int): The amount of .. math:: lpha electrons in the system

param .. math: N_b (int): The amount of .. math:: beta electrons in the system

return ONV (np.array) The occupation number vector for both .. math:: lpha and .. math:: eta are updated

compute_density_matrix(*N_a*, *N_b*)

Computes the groundstate density of based on the ONV

Parameters

Parameters

- **math:** (.) – N_a (int): The number of .. math:: lpha electrons
- **math:** – N_b (int): The number of .. math:: eta electrons

Return P (np.array) A (N x N) numpy array that represents the total electron density matrix from the system when it contains .. math:: N_a and .. math:: N_b electrons

compute_energy(*N_a*, *N_b*)

Computes the molecular energy in the Huckel model based on the occupation of alpha and beta electrons of the molecular orbitals. This functions assumes that the lowest eigenvalues are the first elements

Parameters

Parameters

- **math:** (.) – N_a (int): The number of .. math:: lpha electrons

- **math:** – N_b (int): The number of .. math:: eta electrons

Return E (float) A floating point value that corresponds to the total energy of the system when contain .. math:: N_a and .. math:: N_b electrons

solve_general(*H*, *S*)

A function that solves the Schrödinger equation working when including a general overlap

Parameters

Parameters

- **(np.array)** (*S*) – A (N x N) numpy array that represents the system that needs to be computed
- **(np.array)** – A (N x N) numpy array that contains the overlap values associated with the given H

Return None The function does computes the eigenvalues and eigenvectors of the given matrix H using the eigh function from scipy. These solutions are stored in the corresponding attributes

solve_ndo(*H*)

A function that solves the Schrödinger equation working under the assumption of non-differential overlap

Parameters :param H (np.array): A (N x N) numpy array that represents the system that needs to be computed :return None: The function does computes the eigenvalues and eigenvectors of the given matrix H

using the eigh function from numpy. These solutions are stored in the corresponding attributes

1.3 gqcml.data_generators.graph_sampler module

class gqcml.data_generators.graph_sampler.**graph_sampler**(*sites*)

Bases: object

convert_triu_to_mat(*triu_vector*)

generate_diagonal_vector(*nmb_el_type*)

A function that generates a set of vectors that specifies all possible combinations of the diagonal elements of the adjacency matrix.

Parameters

Parameters **(list)** (*nmb_el_type*) – A list that contains the number of occurence of each unique element, e.g. suppose we have a diagonal of length 4 with 2 unique elements in equal occurence then this would correspond with [2,2]

Returns A set of vectors that contain all possible permutations of a vector that has the specified ratios of these elements. E.g. for 2 unique elements in a diagonal of length 4 (equal ratios) this would return [[1,1,2,2],[1,2,1,2], [2,1,1,2]

permutate_matrix(*matrix*, *permutations*)

sample_homogeneous_matrix(*triu_vector*, *amount_samples*, *diagonal_interval=[- 5, 0.001]*, *off_diagonal_interval=[- 5, - 0.001]*)

A function that takes a upper triangle of a homogeneous systems, i.e. a graph consisting of a single vertex type, and generates an amount of samples within the specified sampling ranges

Parameters

Parameters

- **(np array)** (*triu_vector*) – A upper triangle np array that describes the structure of the system The array consists of 0's and 1's where the 1 signifies a weight.
- **(int)** (*amount_samples*) – An integer that determines how many samples need to be generated
- **(opt, list)** (*off_diagonal_interval*) – A list of floats that determines the lower and upper range of the uniform random distribution used in the generation of the samples for the diagonal elements of the adjacency matrix
- **(opt, list)** – A list of floats that determines the lower and upper range of the uniform random distribution used in the generation of the samples for the off-diagonal elements of the adjacency matrix

Return sampled_trius (list) A list of numpy arrays containing the upper triangle vector sampled randomly

sample_inhomogeneous_matrix(*diagonal_vector*, *triu_vector*, *amount_samples*, *diagonal_interval=[- 5, 0.001]*, *off_diagonal_interval=[- 5, - 0.001]*)

Constructs a weighted adjacency matrix sampled uniformly based on the symbolic assignment of the elements

- The diagonal elements are assigned symbolic through integers. These integers will be replaced by random values
- Off diagonal elements are sampled based on the the diagonal indices

This function will thus generate an adjacency matrix that has unique weights for every type of bond. A type of bond is considered to be the edge between a pair of diagonal weihgts.

Parameters

Parameters

- **(np array)** (*triu_vector*) – A np array of integers that symbolic denotes the type of vertex.
- **(np array)** – A upper triangle np array that describes the structure of the system The array consists of 0's and 1's where the 1 signifies a weight.
- **(int)** (*amount_samples*) – An integer that determines how many samples need to be generated
- **(opt, list)** (*off_diagonal_interval*) – A list of floats that determines the lower and upper range of the uniform random distribution used in the generation of the samples for the diagonal elements of the adjacency matrix
- **(opt, list)** – A list of floats that determines the lower and upper range of the uniform random distribution used in the generation of the samples for the off-diagonal elements of the adjacency matrix

Return sampled_trius (list) A list of numpy arrays containing the upper triangle vector sampled randomly

1.4 Module contents

CHAPTER

TWO

GQCML.DATA PACKAGE

2.1 Submodules

2.2 gqcml.data.Data module

class gqcml.data.Data.**Preprocessor**(*graph_dim*)

Bases: object

adjacency_matrix(*matrix*, *diagonal=None*, *normalize=False*, *negative_weights=False*)

A function to format the matrix into a adjacency matrix format. The function has the capability to change the diagonal elements of the matrix according to the argument given to 'diagonal'

In addition to this we also provide the option to normalize the given adjacency matrix using the following formula

$$= D^{-1/2} A D^{-1/2}$$

where the normalized is computed using the absolute value of the adjacency matrix

Parameters

Parameters

- **(np.array)** (*matrix*) – A numpy array that represents the graph that is currently being processed The array has the dimension (N x N) where N is the number of vertices in the graph
- **diagonal (str)** (*(opt)*) – An optional parameter that determines the diagonal of this matrix. Standardly we set this value to None resulting in a diagonal that remains unchanged. Alternatively we can give the following keywords

  1) 'ones'-replace the diagonal elements with ones

  2) 'zeros'-replace the diagonal elements with zero
- **normalize (bool)** (*(opt)*) – A boolean that determines whether the matrix should be normalized using its degree matrix (computed using the absolute values of the matrix)
- **negative_weights (bool)** (*(opt)*) – A boolean that is used in the determination of the added one to the diagonal (negative or positive) and how the normalization is computed

Return adjacency_matrix (np.array) The adjacency matrix

binarize_matrix(*matrix*, *diagonal=True*)

A function that binarizes a given matrix with the option to exclude the diagonal elements Parameters

:param matrix (np.array): A 2D numpy array that will be converted to its binary form :param (opt) diagonal (bool): A boolean that indicates whether the diagonal should be 1 or 0 :return binary_matrix (np.array): A binary representation of the input matrix

pdegree_weighted_nf(*matrix*, *num_degrees*)

Constructs a seperate class representation that characterizes the edges that connect to the central vertex. This class representation is constructed as a set of tuples where the first element of the tuple describes the central vertex's degree and the second element describes the connecting vertex's degree.

{(1,1), (1,2),. . . , (2,1), (2,2), . . . }.

as such this class degree will have the dimension (1 X .. math::M^2) where M is the predefined maximum degree number (standarly this will be set to 3 when working with organic molecules).We provide the option to process the following weight types

1) "diagonal", the diagonal elements of the matrix will be used in the neighbourhood description of the central vertex.
2) "off diagonal", the off diagonal elements of the matrix will be used in the neighbourhood description of the central vertex

Parameters **(numpy array)** (*matrix*) – A 2D matrix (N X N) representing the graph that is to be processed. This matrix is a square matrix that contains the weights that will be used in weighting the generated categorical vectors. The weights can either be the edge weights or the self loop weights (diagonal repeated as a row).

triu_to_matrix(*triu*)

Convert a vector of upper triangular values to a full, symmetric matrix.

Parameters :param triu (np.array): A 1D numpy array containing the upper triangle values of a symmetric

matrix

Return matrix (np.array) The matrix form of the upper triangle

vdegree_nf(*matrix*, *categorical=True*, *num_degrees=3*)

A funtion that formats the weighted matrix to generate a vertex degree feature based on the weights in the matrix. The degree is computed as the number of weights that each vertex possesses. In addition we standardly enable the conversion of this integer feature to the categorical form.

Parameters

Parameters

- **(numpy array)** (*matrix*) – A numpy array that represents the adjacency matrix that represents the graph.
- **categorical (bool)** (*(opt)*) – Option to convert the degree feature into the categorical format. Standard value for this argument is True

Return vdegree_nf, matrix (np.array) A numpy array that describes the vertices in terms of their degree

vdegree_weighted_nf(*matrix*, *neighbourhood=False*, *weight_method='average'*)

A vertex degree node feature constructor that weights the categorical vector using different weights types. The weight types that we specify are the following options

If "neighbourhood" is set to true we weight the categorical with the self loops but also weights 2 additional categorical vectors. These 2 vectors take the following weighting into account

- The edges weights

- The self loops of the connecting vertices`

When using the neighbourhood weight type we als can specify how to weight these neighbourhood

1) “average”, the off diagonal elements are average and subsequently used to weight the degree vector of the central vertex
2) “linear combination”, the off diagonal elements are weighted using the degree vector of the connecting vertices which are then subsequently summed and averaged per degree class

Parameters

Parameters

- **(numpy array)** (*matrix*) – A 2D matrix (N X N) representing the graph that is to be processed. This matrix is a square matrix that contains the weights that will be used in weighting the generated categorical vectors. The weights can either be the edge weights or the self loop weights (diagonal repeated as a row)
- **neighbourhood (bool)** (*(opt)*) – Boolean that enables the option to include information about the neighbourhood weights. The edge weights and self loop weights of the neighbouring vertices are formatted in the categorical format. The method which this is done is determined by the weight_method optional argument
- **weight_method (str)** (*(opt)*) – Method by which we incorporate neighbourhood information into the node feature representation.
  1) ”average”, neighbourhood weights are averaged and then used to weight the the degree vector of the central vertex
  2) ”linear combination” neighbourhood weights are weighted with the degree vectors of the connecting vertex and subsequently summed up and averaged, meaning that we count the amount of neighbouring vertices have a specific degree which will be used to rescale the summed combination

Return vdegree_weighted_nf (numpy.array) a numpy array that describes the vertices as a categorical degree vector weighted by its self loop weight

weights_nf(*matrix*, *edge_weights=False*)

A function that formats the weighted matrix as node_features using the self loop weight as the node features. In addition to this we also provide to option to include the average of the edge weights as an additional feature

Parameters

Parameters

- **(numpy array)** (*matrix*) – A 2D numpy array that contains diagonal and off-diagonal weights that parametrize the graph
- **edge_weights** (*(opt)*) – A boolean argument that determines whether the average of the off-diagonal elements should be included as an additional node feature

Return node_features Numpy array where the node features are the self loop weights and possibly the average of the edge weights as well

2.3 Module contents

CHAPTER

THREE

GQCML.DATASETS PACKAGE

3.1 Submodules

3.2 gqcml.datasets.Datasets module

gqcml.datasets.Datasets.**DataLoader_constructor**(*input_tensors*, *output_tensors*, *batch_size*, *shuffle=True*, *pin_memory=False*, *num_workers=0*)

A function that takes in a set of input tensors and a set of output tensor and constructs a data loader that can be used in the optimization of network.

Parameters :param input_tensors (list of torch.Tensor): A list of input tensors :param output_tensors (torch.Tensor): A tensor that contains the corresponding output tensors :param batch_size (int): An integer that determines the number of tensors in each batch :param shuffle (opt, bool): A boolean that enables the option to shuffle the data during batch iteration :param pin_memory (opt, bool): A boolean that enables the option to pin memory in the GPUs utilized during training.

This allows for faster data transfer

Parameters **(opt, int)** (*num_workers*) – An integer that controls how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process

Return loader (torch.data.DataLoader) A DataLoader object that returns batch objects

gqcml.datasets.Datasets.**bin_values**(*target_values*, *num_bins*)

A function to partition a continuous target variable into discrete bins. This function is intended to work together with the train_test_split functionality from sklearn.model_selection to partition the dataset into representative training, validation and test data by ensuring that target values are equally represented

Parameters

- **(np.array)** (*target_values*) – A 2D numpy array containing the continuous target values of the dataset. Each row represents the target value of a datapoint
- **(int)** (*num_bins*) – An integer that determines the number of output classes that the function returns

Return binned_values (np.array) A 2D array where the values of the input array have been classified into discrete bins determined by the minimum and maximum of the input array.

gqcml.datasets.Datasets.**split_dataset**(*input_values*, *target_values*, *data_frac*, *seed*, *stratify=True*, *num_bins=10*, *shuffle=True*)

Split_dataset

A function that splits a dataset into a training, validation and test set.

Parameters

- **(np.array)** (*output_values*) – A 2D numpy array containing the input values of the dataset. Each row represents an input vector of the dataset.
- **(np.array)** – A 2D numpy array containing the continuous target values of the dataset. Each row represents the target value of a datapoint.
- **(float)** (*data_frac*) – The fraction of the dataset that should be reserved for the validation and test set. This fraction will be split in half for both these sets.
- **(int)** (*seed*) – The random seed used to determine the split of the dataset.
- **stratify (bool)** (*(optional)*) – Option to partition the dataset according to the distribution of the target values. This option ensures that all the sets follow the same distribution in their target.
- **num_bins (int)** (*(optional)*) – An integer that determines the number of output classes
- **shuffle (bool)** (*(optional)*) – Option to shuffle the dataset before partitioning it into a training, validation and test set.

Return inputs, outputs (tuple) The function returns a tuple of the split dataset in the order of input data followed by output data. The order in which this happens is training, validation and test set

gqcml.datasets.Datasets.**trius_to_inputs**(*trius*, *preprocessor*, *am_args*, *preprocessor_nf_method*, *nf_args*)

trius_to_input

A function that interacts with the preprocessor class defined in the Data section of gqcml. The function takes as stack of upper triangle values and converts them to their matrix representation. Following the conversion to matrices we utilize methods defined in the preprocessor class to convert the matrices to a suitable input for graph neural networks. The different methods that are defined are

1) (categorical) degree node feature
2) weight node feature
3) weighted categorical degree
4) weighted categorical pair degree

Parameters

Parameters

- **(np.array)** (*trius*) – A numpy array containing the trius of the graphs that are to be processed. This array has the dimension (K x M) where K is the number of graphs contained in the dataset and M=N(N-1)/2 with N the number of vertices in the graph
- **(gqcml.data.Data)** (*preprocessor_am_method*) – A function from the Preprocessor class that processes the matrices to the adjacency matrices
- **(list)** (*args*) – The list of function arguments (excluding the input matrix) for the am_method. When no arguments need to specified the input should be an empty list
- **(gqcml.data.Data.Preprocessor)** (*preprocessor_nf_method*) – A function from the Preprocessor class that processes the matrices into the desired node feature class
- **(list)** – The list of function arguments (excluding the input matrix) for the nf_method. When no arguments need to specified the input should be an empty list

- **(list)** – The arguments for the preprocessor_method excluding the matrix to be processed.

Return tuple of numpy arrays Returns the processed node features and the adjacency matrices

3.3 Module contents

CHAPTER

FOUR

GQCML.NN PACKAGE

4.1 Submodules

4.2 gqcml.nn.blocks module

class gqcml.nn.blocks.**GraphConv_Block**(*node_input_dim*, *nmb_GC_layers*, *nmb_filter_dlayers*, *activation_function*)

Bases: torch.nn.modules.module.Module

A sequential block of GraphConv operators that can be used a building module of a graph convolutional model.

Attributes :var node_input_dim (int): The number of features that are associated with each node in the graph G :var nmb_GraphConv_layers (int): The number of GraphConv layers that need to be present in the block :var nmb_filter_dlayers (int): The number of non-linear transformations that are performed in each graph convolution

after aggregation of the neighbourhood features. The dense layers standardly are constructed to output as many amount of features that go into them

forward(*node_feat*, *adj_matrix*)

Forward propagation function that produces an output

Parameters :param node_feat (torch.Tensor): The initial node feature tensors that are associated with a batch of graphs. This

tensor should have the dimensions (B x N x F) where B is the batch size, N the number of nodes present in the graphs and F the number of features associated with each node present in the graph

Parameters **(torch.Tensor)** (*adj_matrix*) – The weighted adjacency tensors that are associated with a batch of graphs. This tensor should have the dimensions (B x N x N) where B is the batch size and N the number of nodes present in the graph

Return transf_node_feat (torch.Tensor) The transformed node features that are obtained by consecutively performing the graph convolutional operator. This tensor has the dimensions (B x N x F) where B is the batch size, N the number of nodes present in the graph and F is the number of features associated with each node

4.3 gqcml.nn.layers module

class gqcml.nn.layers.**GaussianEmbedding**(*lower*, *upper*, *num_gaussians*, *emb_dim*, *variance=None*, *emb_bias=False*)

Bases: torch.nn.modules.module.Module

A function that combines the GaussianExpansion layer with the concept of embedding. Should the initial features be described by a single continuous variable the problem becomes that learning an embedding for this becomes difficult due to the mathematical construct of embedding where

$$EMB(v_i) = Zv_i$$

Where Z is a learnable set of weights. In the case of a single node feature v_i the dimension of this Z becomes a vector and thus does not contain many learnable parameters. When we use the Gaussian expansion layer we convert this continuous variable to "quasi" binned variable where our bins consist of the different gaussians within the layer and the layer measures "overlap" with the gaussians. Using this expanded representation we provide more freedom to the Z matrix in our embedding due to the increased dimensionality.

Attributes

Variables

- **(float)** (*variance*) – A float that bounds the linear space from which the means are generated. The space spans the interval [lower, upper]
- **(float)** – A float that bounds the linear space from which the means are generated. The space spans the interval [lower, upper]
- **(float)** – A float that controls the width of gaussians that we construct.
- **(int)** (*emb_dim*) – The number of gaussian functions that we place within the linear space.
- **(int)** – The dimension of the embedded node feature.
- **emb_bias (bool)** (*(opt)*) – A boolean that controls the addition of a learnable bias for the embedding

forward(*inp*)

A function that calls the forward propagation

Parameters

Parameters **(torch.Tensor** (*inp*) – A (B x N x 1) tensor, where B is the batch size and N is the number of nodes, that should be expaned in a Gaussian basis

Return embedded tensor (torch.Tensor) A (B x N x E) tensor where E is the embedding dimension

class gqcml.nn.layers.**GaussianExpansion**(*lower*, *upper*, *num_gaussians*, *variance=None*)

Bases: torch.nn.modules.module.Module

A layer that takes an input tensor and expands it into a gaussian basis. The layer takes a tensor consisting of single descriptors and bins this descriptor. The descriptor is fed to a function that computes the output value of a series of gaussians that are equidistantly spaced using a linear space of means and a single variance parameter. The output of this function thus maps a scalar value to a vector with the dimension equal to the number of gaussians that we have chosen to span the linear space of means. .. math:

```
\phi(
```

u_i)= u_i ightarrow ec{ u_i}: mathbb{R} mapsto mathbb{R}^N

phi(

u_i)=egin{bmatrix} e^{ u_i-mu_1}{2sigma^2} & e^{ u_i-mu_2}{2sigma^2} & dots e^{ u_i-mu_N}{2sigma^2} end{bmatrix}

Attributes :var lower (float): A float that bounds the linear space from which the means are generated.

The space spans the interval [lower, upper]

var upper (float) A float that bounds the linear space from which the means are generated. The space spans the interval [lower, upper]

var num_gaussians (int) The number of gaussian functions that we place within the linear space.

var (opt) variance (float) Standardly this is not defined. When this optional parameter is not defined we compute the variance so that the FWHM of neighbouring overlaps. If a float is given the variance as defined will be used.

forward(*inp*)

Forward phase of the layer

Parameters

Parameters **(torch.Tensor** (*inp*) – A (B x N x 1) tensor, where B is the batch size and N is the number of nodes, that should be expaned in a Gaussian basis

Return expanded tensor (torch.Tensor) A (B x N x M) tensor where M is the number of Gaussians

meta()

A function that extracts the meta data from the layer

class gqcml.nn.layers.**GraphConv**(*node_input_dim*, *node_output_dim*, *activation_function*, *bias=True*)

Bases: torch.nn.modules.module.Module

The standard graph convolution operator defined in the paper Semi-Supervised Classification with Graph Convolutional Networks (https://arxiv.org/abs/1609.02907). .. math:

```
n_(i+1) = h(A.n_(i).W_(i+1))
```

Where the current node features are aggregated according to the adjacency matrix A and subsequently weighted by a (set of) weight matrix/matrices. We include the option for a residual connection by adding the transformed node features to the current state

Attributes :var node_input_dim (int): The number of features present on the nodes of the graph G when entering the layer

Variables

- **(int)** (*node_output_dim*) – The number of features present in the layer output
- **(torch.nn)** (*activation_function*) – The activation function to be used for the filter layer(s) after message construction
- **(opt, bool)** (*bias*) – A boolean that enables or disables the bias term in the filter application. In the standard settings this value is True

forward(*node_feat*, *adj_matrix*)

Function that calls the forward pass of the layer

Parameters :param: node_feat (torch.Tensor): The current node features, this tensor should have the dimensions (B x N x F) where

B is the batch size, N is the number of nodes and F is the number of features

Param adj_matrix (torch.Tensor): The adjacency matrices by which the node features need to be aggregated, this tensor should have the dimensions (B x N x N) where B is the batch size and N the number of nodes

Returns transf_node_emb (torch.Tensor): The updated node features with the aggregated features according to the adjacency matrix. This tensor has the same dimensions as

reset_parameters()
A function that initializes the weights and biases of the learnable layer

class gqcml.nn.layers.**ShiftedSoftplus**(*beta=2*)
Bases: torch.nn.modules.module.Module

The shifted softplus activation function

$$ln\left(eta + etae^{x}\right.$$

ight)

Attributes :var beta (opt,float): A shifting factor with standard setting 2

forward(*inp*)
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.

4.4 gqcml.nn.models module

class gqcml.nn.models.**DNN**(*layer_configs*, *activation_function*, *bias=True*)
Bases: torch.nn.modules.module.Module

Dense Neural Network

A standard dense network consisting of multiple hidden layers

Arguments

layer_dims (list) [A list containing the number of nodes in each layer for the network.] The first entry in the list corresponds to the number of features in the input tensor and the last entry in the list corresponds to the number of features in the output tensor

activation_function (torch.nn): The activation used in the network after each hidden layer

forward(*input_tensor*)
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.

meta()
Takes the input arguments of the class and formats them in a dictionary format which can be used in creating an overarching meta file

class gqcml.nn.models.**GraphConv_model**(*node_embedding_nn*, *GC_dimensions*, *activation_function*, *node_prop_nn*, *bias=True*)
Bases: `torch.nn.modules.module.Module`

Graph Convolution model

A standard graph convolution model constructor

Arguments

node_embedding_nn (torch.nn.Module): A torch.nn.Module that processes the initial node features. This torch modu
takes a tensor with dimensions (B x N x f) and should return a tensor with dimensions (B x N x F) where B is the batch size, N is the number of nodes in the graphs, f is the initial number of node features and F is the number of node features associated with each node in the graph. F should be equal to the node_input_dim parameter of the GraphConv layers.

GC_dimensions (list): A list of integers describing the consecutive node embedding dimensions that go into the graph
convolution layers and are subsequently returned.

activation_function (torch.nn): The activation function used non-linear transformation of the aggregated neighbourh
information.

node_prop_nn (torch.nn.Module): A torch.nn.Module that processes the transformed node features after the applica
of the graph convolution operators. The network takes the node features with dimension (B x N x F) where B is the batch size, N is the number of nodes in the graph and F the number of features associated with each node in the graphs. The output of the network is a tensor that has the dimensions (B x N x 1) where the resulting features are mapped to a single node property.

forward(*node_feat*, *adj_matrix*)

Parameters

- **(torch.Tensor)** (*adj_matrix*) – The initial node feature tensor with the dimensions (B x N x f) where B is the batch size, N is the number of nodes in each graph and f is the number of node features associated with each node in the graphs.
- **(torch.Tensor)** – The weighted adjacency tensor with the dimensions (B x N x N) where B is the batch size and N is the number of nodes in each graph.

Return (torch.Tensor) The graph property tensor with dimensions (B x 1) where B is the batch size. The graph property is computed as the sum of the node properties obtained through the site_prop_nn

meta()

site_properties(*node_feat*, *adj_matrix*)

Parameters

- **(torch.Tensor)** (*adj_matrix*) – The initial node feature tensor with the dimensions (B x N x f) where B is the batch size, N is the number of nodes in each graph and f is the number of node features associated with each node in the graphs.

- **(torch.Tensor)** – The weighted adjacency tensor with the dimensions (B x N x N) where B is the batch size and N is the number of nodes in each graph.

Return (torch.Tensor) The site properties tensor with dimensions (B x N x1) where B is the batch size and N is the number of nodes in each graph.

4.5 gqcml.nn.models_test module

class gqcml.nn.models_test.**GCNConv_gqcml**
Bases: torch.nn.modules.module.Module

forward(*node_features*, *adj_matrices*)
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 gqcml.nn.models_test.**GCNConv_tg**
Bases: torch.nn.modules.module.Module

forward(*data*)
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.

4.6 Module contents

CHAPTER

FIVE

GQCML.TORCHGEOM_INTERFACE PACKAGE

5.1 Submodules

5.2 gqcml.torchgeom_interface.auto_encoder module

class gqcml.torchgeom_interface.auto_encoder.**GraphAutoEncoder**(*num_aggr*, *layer_config*, *act_fn*, *dropout=True*, *dropout_rate=0.1*, *feature_transformation=False*)

Bases: torch.nn.modules.module.Module

decode(*encoded_node_features*)

encode(*data*)

forward(*data*)

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 gqcml.torchgeom_interface.auto_encoder.**Neighbourhood_Aggregator**

Bases: torch_geometric.nn.conv.message_passing.MessagePassing

A class that is used in the generation of a feature vector containing raw node features that are aggregated. This function works in a similar fashion to GCNConv but removes the use of any bias and weights.

forward(*x: torch.Tensor*, *edge_index: Union[torch.Tensor, torch_sparse.tensor.SparseTensor]*, *edge_weight: Optional[torch.Tensor] = None*) → torch.Tensor

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.

---

message(*x_j: torch.Tensor*, *edge_weight: Optional[torch.Tensor]*) → torch.Tensor

Constructs messages from node j to node i in analogy to $\phi_{\mathbf{\Theta}}$ for each edge in `edge_index`. This function can take any argument as input which was initially passed to `propagate()`. Furthermore, tensors passed to `propagate()` can be mapped to the respective nodes i and j by appending `_i` or `_j` to the variable name, *.e.g.* `x_i` and `x_j`.

message_and_aggregate(*adj_t: torch_sparse.tensor.SparseTensor*, *x: torch.Tensor*) → torch.Tensor

Fuses computations of *`message()`* and `aggregate()` into a single function. If applicable, this saves both time and memory since messages do not explicitly need to be materialized. This function will only gets called in case it is implemented and propagation takes place based on a `torch_sparse.SparseTensor`.

gqcml.torchgeom_interface.auto_encoder.**sum_similarity_loss**(*inp_tensor*, *decoded_tensor*)

gqcml.torchgeom_interface.auto_encoder.**train_model**(*device*, *model*, *nmb_epochs*, *train_loader*, *val_loader*, *loss_fn*, *optimizer*, *model_logger*, *scheduler=None*, *verbose=False*)

train model

A function that trains a given neural network

Parameters

param device (str) The device to which the model and tensors are moved during training.

param model (torch.Module) The neural network that needs to be trained. This is aa torch.nn.Module class that has a forward function

param nmb_epochs (int) The number of epochs that the model needs to be trained

param train_loader (torch.DataLoader) The dataloader that contains the training dataset and generates batches

param val_loader (torch.DataLoader) The dataloader that contains the validation dataset and generates batches

param loss_fn (torch.nn) The loss function that is used during the optimization of the neural network

param optimizer (torch.optim) The optimizer that is used during the training of the model

param model_logger (gqcml.utils.train) The model logger class that registers the training progress and saves the best model

param (optional) scheduler (torch.optim.lr_scheduler) A scheduler for decreasing the learning rate

5.3 gqcml.torchgeom_interface.blocks module

class gqcml.torchgeom_interface.blocks.**GCNConv_block**(*inp_dim*, *hidden_channels*, *num_layers*, *act_fn*, *add_self_loops=False*, *normalize=False*, *bias=True*, *residual=False*)

Bases: torch.nn.modules.module.Module

forward(*x*, *edge_idx*, *edge_attr*)

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.

---

meta()

class gqcml.torchgeom_interface.blocks.**GraphConv_block**(*inp_dim*, *hidden_channels*, *num_layers*, *act_fn*, *aggr='add'*, *bias=True*, *residual=False*)

Bases: torch.nn.modules.module.Module

forward(*x*, *edge_idx*, *edge_attr*)

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.

---

meta()

5.4 gqcml.torchgeom_interface.datasets module

gqcml.torchgeom_interface.datasets.**TriuDataset**(*dim*, *trius*, *output_values*, *residual=False*)

gqcml.torchgeom_interface.datasets.**graph_to_Data**(*nf*, *am*, *output*)

5.5 gqcml.torchgeom_interface.models module

class gqcml.torchgeom_interface.models.**Graph_Convolution_attentionpool_model**(*emb*, *conv_block*, *prop_dnn*, *gate_nn*, *nn=None*, *max_feature=False*)

Bases: torch.nn.modules.module.Module

forward(*data*)

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.

---

meta()

class gqcml.torchgeom_interface.models.**Graph_Convolution_avgpool_model**(*emb*, *conv_block*, *prop_dnn*, *max_feature=False*)

Bases: torch.nn.modules.module.Module

forward(*data*)

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.

---

meta()

class gqcml.torchgeom_interface.models.**Graph_Convolution_model**(*emb*, *conv_block*, *prop_dnn*)

Bases: torch.nn.modules.module.Module

forward(*data*)

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.

---

meta()

5.6 gqcml.torchgeom_interface.models_3 module

class gqcml.torchgeom_interface.models_3.**GCNConv**(*node_embedding_nn*, *gc_layers*, *activation_function*, *node_prop_nn*, *normalize=False*, *bias=True*, *add_self_loops=False*, *residual=False*)

Bases: torch.nn.modules.module.Module

GCNConv model

A standard graph convolutional network using the torch geomtric interface

Parameters

- **num_sites** – number of sites/atoms in the Hückel model
- **node_embedding_nn** (*torch.nn.Module*) – A torch.nn.Module that processes the initial node features. This torch module takes a tensor with dimensions (B x N x f) and should return a tensor with dimensions (B x N x F) where B is the batch size, N is the number of nodes in the graphs, f is the initial number of node features and F is the number of node features associated with each node in the graph. F should be equal to the node_input_dim parameter of the GraphConv layers.
- **gcn_configs** (*list*) – A list containing the dimensions of the consecutive layers of the embedding section of the network The first value of this list corresponds to the dimensions of the node feature vector and the last value corresponds to the first node feature dimension of the graph convolutions section.
- **activation_function** (*torch.nn*) – The activation function that will be used in after each layer in the embedding, graph convolution and property prediction phase of the network
- **node_prop_nn** (*torch.nn.Module*) – A torch.nn.Module that processes the transformed node features after the application of the graph convolution operators. The network takes the node features with dimension (B x N x F) where B is the batch size, N is the number of nodes in the graph and F the number of features associated with each node in the graphs. The output of the network is a tensor that has the dimensions (B x N x 1) where the resulting features are mapped to a single node property.
- **normalize** (*optional, boolean*) – Option to enable the use of the normalized form of the graph convolution
- **bias** (*optional, boolean*) – Option to add a bias to the layers

forward(*data*)

Forward

The function to compute the model output.

Parameters

param data (torch_geometric.data.Batch) A Batch object from torch geometric obtained from the loader which contains x, edge_index, edge_attr and batch

return graph property (torch.tensor) A 2D tensor containing the output values associated with the input graphs

meta()

Meta

Returns a dictionary that contains all the network parameters sorted according 3 blocks (embedding, gc and property).

node_properties(*data*)
Node properties

Returns the properties of the nodes, i.e. the output of the layer before summing to obtain the graph property

Parameters

param data (torch_geometric.data.Batch) A Batch object from torch geometric obtained from the loader which contains x, edge_index, edge_attr and batch

return node properties (torch.tensor) A 2D tensor containing the output values associated with the nodes of the input graphs

class gqcml.torchgeom_interface.models_3.**GatedGraphConv**(*node_embedding_nn*, *output_channels*, *nmb_layers*, *node_prop_nn*, *bias=True*)

Bases: torch.nn.modules.module.Module

Gated Graph Conv model

A standard graph convolutional network using the torch geomtric interface

Parameters

- **num_sites** – number of sites/atoms in the Hückel model
- **node_embedding_nn** (*torch.nn.Module*) – A torch.nn.Module that processes the initial node features. This torch module takes a tensor with dimensions (B x N x f) and should return a tensor with dimensions (B x N x F) where B is the batch size, N is the number of nodes in the graphs, f is the initial number of node features and F is the number of node features associated with each node in the graph. F should be equal to the node_input_dim parameter of the GraphConv layers.
- **gcn_configs** (*list*) – A list containing the dimensions of the consecutive layers of the embedding section of the network The first value of this list corresponds to the dimensions of the node feature vector and the last value corresponds to the first node feature dimension of the graph convolutions section.
- **activation_function** (*torch.nn*) – The activation function that will be used in after each layer in the embedding, graph convolution and property prediction phase of the network
- **node_prop_nn** (*torch.nn.Module*) – A torch.nn.Module that processes the transformed node features after the application of the graph convolution operators. The network takes the node features with dimension (B x N x F) where B is the batch size, N is the number of nodes in the graph and F the number of features associated with each node in the graphs. The output of the network is a tensor that has the dimensions (B x N x 1) where the resulting features are mapped to a single node property.
- **normalize** (*optional, boolean*) – Option to enable the use of the normalized form of the graph convolution
- **bias** (*optional, boolean*) – Option to add a bias to the layers

forward(*data*)
Forward

The function to compute the model output.

Parameters

param data (torch_geometric.data.Batch) A Batch object from torch geometric obtained from the loader which contains x, edge_index, edge_attr and batch

return graph property (torch.tensor) A 2D tensor containing the output values associated with the input graphs

meta()
Meta

Returns a dictionary that contains all the network parameters sorted according 3 blocks (embedding, gc and property).

node_properties(*data*)
Node properties

Returns the properties of the nodes, i.e. the output of the layer before summing to obtain the graph property

Parameters

param data (torch_geometric.data.Batch) A Batch object from torch geometric obtained from the loader which contains x, edge_index, edge_attr and batch

return node properties (torch.tensor) A 2D tensor containing the output values associated with the nodes of the input graphs

5.7 gqcml.torchgeom_interface.utils module

gqcml.torchgeom_interface.utils.**evaluate_model**(*device*, *model_path*, *error_path*, *modelname*, *model*, *test_loader*, *model_descr='_val.pth'*)

gqcml.torchgeom_interface.utils.**train_model**(*device*, *model*, *nmb_epochs*, *train_loader*, *val_loader*, *loss_fn*, *optimizer*, *model_logger*, *scheduler=None*, *verbose=False*, *reduction='mean'*)

train model

A function that trains a given neural network

Parameters

param device (str) The device to which the model and tensors are moved during training.

param model (torch.Module) The neural network that needs to be trained. This is aa torch.nn.Module class that has a forward function

param nmb_epochs (int) The number of epochs that the model needs to be trained

param train_loader (torch.DataLoader) The dataloader that contains the training dataset and generates batches

param val_loader (torch.DataLoader) The dataloader that contains the validation dataset and generates batches

param loss_fn (torch.nn) The loss function that is used during the optimization of the neural network

param optimizer (torch.optim) The optimizer that is used during the training of the model

param model_logger (gqcml.utils.train) The model logger class that registers the training progress and saves the best model

param (optional) scheduler (torch.optim.lr_scheduler) A scheduler for decreasing the learning rate

5.8 Module contents

CHAPTER

SIX

GQCML.UTILS PACKAGE

6.1 Submodules

6.2 gqcml.utils.test module

gqcml.utils.test.**evaluate_model**(*error_path*, *model_path*, *modelname*, *model*, *test_loader*, *device*, *dataset_descr='test'*, *model_descr='_val.pth'*)

6.3 gqcml.utils.train module

class gqcml.utils.train.**model_logger**(*history_dir*, *model_dir*, *modelname*, *loss_fn*)

Bases: object

add_epoch_metrics(*epoch*, *training_losses*, *nmb_train_datapoints*, *validation_losses*, *nmb_val_datapoints*)

Add epoch metrics

A function used to write the loss metrics of an epoch to the history file.

Parameters

param epoch (int) The number of the epoch the model is currently on

param training_losses (list) A list containing all the batch training losses during the current epoch

param nmb_train_datapoints (int) The total number of datapoints in the training dataset

param validation_losses (list) A list containing all the batch validation losses during the current epoch

param nmb_val_datapoints (int) The total number of datapoints in the validation dataset

return Nothing is returned, the average loss for the training and validation set is computed together with the variances of the batch losses and written to the metrics file

check_improvement(*model*, *epoch*)

Check improvement

A function to checkpoint the model during optimization based on the evolution of the validation loss.

Parameters

param model (torch.Module) The model object that is currently being optimized. This object needs to have the state_dict() function in order to retrieve the parameters.

param epoch (int) The current epoch of the model optimization.

return Nothing is returned, the function checks whether the validation loss has decreased and saved the model

summary()
A function that returns the optimal state of the model during optimization as a dictionary object

gqcml.utils.train.**random_seed**(*seed_value*, *use_cuda*)
random seed

Sets the random seed for all the different packages to the same value

gqcml.utils.train.**train_model**(*device*, *model*, *nmb_epochs*, *train_loader*, *val_loader*, *loss_fn*, *optimizer*, *model_logger*, *scheduler=None*, *verbose=False*)
train model

A function that trains a given neural network

Parameters

param device (str) The device to which the model and tensors are moved during training.

param model (torch.Module) The neural network that needs to be trained. This is aa torch.nn.Module class that has a forward function

param nmb_epochs (int) The number of epochs that the model needs to be trained

param train_loader (torch.DataLoader) The dataloader that contains the training dataset and generates batches

param val_loader (torch.DataLoader) The dataloader that contains the validation dataset and generates batches

param loss_fn (torch.nn) The loss function that is used during the optimization of the neural network

param optimizer (torch.optim) The optimizer that is used during the training of the model

param model_logger (gqcml.utils.train) The model logger class that registers the training progress and saves the best model

param (optional) scheduler (torch.optim.lr_scheduler) A scheduler for decreasing the learning rate

6.4 Module contents

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g