torch_geometric.nn

Contents

Convolutional Layers

class **MessagePassing**(*aggr='add', flow='source_to_target'*) [source]

Base class for creating message passing layers

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\mathbf{x}_i, \square_{j \in \mathcal{N}(i)} \, \phi_{\mathbf{\Theta}} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{i,j} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and $\gamma_{\mathbf{\Theta}}$ and $\phi_{\mathbf{\Theta}}$ denote differentiable functions such as MLPs. See here for the accompanying tutorial.

Parameters:

- **aggr** (*string, optional*) – The aggregation scheme to use (`"add"` , `"mean"` or `"max"`). (default: `"add"`)
- **flow** (*string, optional*) – The flow direction of message passing (`"source_to_target"` or `"target_to_source"`). (default: `"source_to_target"`)

message(*x_j*) [source]

Constructs messages in analogy to $\phi_{\mathbf{\Theta}}$ for each edge in $(i,j) \in \mathcal{E}$. Can take any argument which was initially passed to `propagate()` . In addition, features can be lifted to the source node i and target node j by appending `_i` or `_j` to the variable name, .*e.g.* `x_i` and `x_j` .

propagate(*edge_index, size=None, **kwargs*) [source]

The initial call to start propagating messages.

Parameters:

- **edge_index** (*Tensor*) – The indices of a general (sparse) assignment matrix with shape `[N, M]` (can be directed or undirected).
- **size** (*list or tuple, optional*) – The size `[N, M]` of the assignment matrix. If set to `None`, the size is tried to get automatically inferrred. (default: `None`)
- ****kwargs** – Any additional data which is needed to construct messages and to update node embeddings.

update(*aggr_out*) [source]

Updates node embeddings in analogy to $\gamma_{\mathbf{\Theta}}$ for each node $i \in \mathcal{V}$. Takes in the output of aggregation as first argument and any argument which was initially passed to `propagate()`.

---

class **GCNConv**(*in_channels, out_channels, improved=False, cached=False, bias=True*) [source]

The graph convolutional operator from the "Semi-supervised Classfication with Graph Convolutional Networks" paper

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **improved** (*bool, optional*) – If set to `True`, the layer computes $\hat{\mathbf{A}}$ as $\mathbf{A} + 2\mathbf{I}$. (default: `False`)
- **cached** (*bool, optional*) – If set to `True`, the layer will cache the computation of $\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\right)$. (default: `False`)
- **bias** (*bool, optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x, edge_index, edge_weight=None*) [source]

static **norm**(*edge_index, num_nodes, edge_weight, improved=False, dtype=None*) [source]

reset_parameters() [source]

---

class **ChebConv**(*in_channels, out_channels, K, bias=True*) [source]

The chebyshev spectral graph convolutional operator from the "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering" paper

$$\mathbf{X}' = \sum_{k=0}^{K-1} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

where $\mathbf{Z}^{(k)}$ is computed recursively by

$$\begin{aligned}\mathbf{Z}^{(0)} &= \mathbf{X}\\ \mathbf{Z}^{(1)} &= \hat{\mathbf{L}} \cdot \mathbf{X}\\ \mathbf{Z}^{(k)} &= 2 \cdot \hat{\mathbf{L}} \cdot \mathbf{Z}^{(k-1)} - \mathbf{Z}^{(k-2)}\end{aligned}$$

and $\hat{\mathbf{L}}$ denotes the scaled and normalized Laplacian.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **K** (*int*) – Chebyshev filter size, *i.e.* number of hops K.
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*, *edge_weight=None*) [source]

reset_parameters() [source]

class **SAGEConv**(*in_channels*, *out_channels*, *normalize=True*, *bias=True*) [source]

The GraphSAGE operator from the "Inductive Representation Learning on Large Graphs" paper

$$\begin{aligned}\hat{\mathbf{x}}_i &= \mathbf{\Theta} \cdot \mathrm{mean}_{j \in \mathcal{N}(i) \cup \{i\}} \left(\mathbf{x}_j \right)\\ \mathbf{x}'_i &= \frac{\hat{\mathbf{x}}_i}{\| \hat{\mathbf{x}}_i \|_2}.\end{aligned}$$

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **normalize** (*bool*, *optional*) – If set to `False`, output features will not be ℓ_2-normalized.
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*) [source]

reset_parameters() [source]

class **GraphConv**(*in_channels*, *out_channels*, *aggr='add'*, *bias=True*) [source]

The graph neural network operator from the "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks" paper

$$\mathbf{x}_i' = \mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \mathbf{\Theta}\mathbf{x}_j.$$

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **aggr** (*string*) – The aggregation scheme to use (`"add"` , `"mean"` , `"max"`). (default: `"add"`)
- **bias** (*bool*, *optional*) – If set to `False` , the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*) [source]

reset_parameters() [source]

---

class **GatedGraphConv**(*out_channels*, *num_layers*, *aggr='add'*, *bias=True*) [source]

The gated graph convolution operator from the "Gated Graph Sequence Neural Networks" paper

$$\begin{aligned}
\mathbf{h}_i^{(0)} &= \mathbf{x}_i \,\|\, \mathbf{0} \\
\mathbf{m}_i^{(l+1)} &= \sum_{j \in \mathcal{N}(i)} \mathbf{\Theta} \cdot \mathbf{h}_j^{(l)} \\
\mathbf{h}_i^{(l+1)} &= \mathrm{GRU}(\mathbf{m}_i^{(l+1)}, \mathbf{h}_i^{(l)})
\end{aligned}$$

up to representation $\mathbf{h}_i^{(L)}$. The number of input channels of $\mathbf{x}_i$ needs to be less or equal than `out_channels` .

Parameters:

- **out_channels** (*int*) – Size of each input sample.
- **num_layers** (*int*) – The sequence length L.
- **aggr** (*string*) – The aggregation scheme to use (`"add"` , `"mean"` , `"max"`). (default: `"add"`)
- **bias** (*bool*, *optional*) – If set to `False` , the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*) [source]

reset_parameters() [source]

---

class **GATConv**(*in_channels*, *out_channels*, *heads=1*, *concat=True*, *negative_slope=0.2*, *dropout=0*, *bias=True*) [source]

The graph attentional operator from the "Graph Attention Networks" paper

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k \in \mathcal{N}(i)\cup\{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}.$$

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **heads** (*int*, *optional*) – Number of multi-head-attentions. (default: `1`)
- **concat** (*bool*, *optional*) – If set to `False`, the multi-head
- **are averaged instead of concatenated. (default** (*attentions*) – `True`)
- **negative_slope** (*float*, *optional*) – LeakyReLU angle of the negative slope. (default: `0.2`)
- **dropout** (*float*, *optional*) – Dropout probability of the normalized attention coefficients which exposes each node to a stochastically sampled neighborhood during training. (default: `0`)
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*) [source]

reset_parameters() [source]

class **AGNNConv**(*requires_grad=True*) [source]

Graph attentional propagation layer from the "Attention-based Graph Neural Network for Semi-Supervised Learning" paper

$$\mathbf{X}' = \mathbf{P}\mathbf{X},$$

where the propagation matrix $\mathbf{P}$ is computed as

$$P_{i,j} = \frac{\exp(\beta \cdot \cos(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k \in \mathcal{N}(i)\cup\{i\}} \exp(\beta \cdot \cos(\mathbf{x}_i, \mathbf{x}_k))}$$

with trainable parameter β.

Parameters: **requires_grad** (*bool*, *optional*) – If set to `False`, β will not be trainable. (default: `True`)

forward(*x*, *edge_index*) [source]

reset_parameters() [source]

---

class **GINConv**(*nn, eps=0, train_eps=False*) [source]

The graph isomorphism operator from the "How Powerful are Graph Neural Networks?" paper

$$\mathbf{x}'_i = h_{\mathbf{\Theta}}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right),$$

here $h_{\mathbf{\Theta}}$ denotes a neural network, *.i.e.* a MLP.

Parameters:

- **nn** (*nn.Sequential*) – Neural network $h_{\mathbf{\Theta}}$.
- **eps** (*float*, *optional*) – (Initial) ϵ value. (default: `0`)
- **train_eps** (*bool*, *optional*) – If set to `True`, ϵ will be a trainable parameter. (default: `False`)

forward(*x, edge_index*) [source]

reset_parameters() [source]

---

class **ARMAConv**(*in_channels, out_channels, num_stacks=1, num_layers=1, shared_weights=False, act=<function relu>, dropout=0, bias=True*) [source]

The ARMA graph convolutional operator from the "Graph Neural Networks with Convolutional ARMA Filters" paper

$$\mathbf{X}' = \frac{1}{K}\sum_{k=1}^{K}\mathbf{X}_k^{(T)},$$

with $\mathbf{X}_k^{(T)}$ being recursively defined by

$$\mathbf{X}_k^{(t+1)} = \sigma\left(\hat{\mathbf{L}}\mathbf{X}_k^{(t)}\mathbf{W} + \mathbf{X}^{(0)}\mathbf{V}\right),$$

where $\hat{\mathbf{L}} = \mathbf{I} - \mathbf{L} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ denotes the modified Laplacian $\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.

Parameters:

- **in_channels** (*int*) – Size of each input sample $\mathbf{x}^{(t)}$.
- **out_channels** (*int*) – Size of each output sample $\mathbf{x}^{(t+1)}$.
- **num_stacks** (*int*, *optional*) – Number of parallel stacks K. (default: `1`).
- **num_layers** (*int*, *optional*) – Number of layers T. (default: `1`)
- **act** (*callable*, *optional*) – Activiation function σ. (default: `torch.nn.functional.ReLU()`)
- **shared_weights** (*int*, *optional*) – If set to `True` the layers in each stack will share the same parameters. (default: `False`)
- **dropout** (*float*, *optional*) – Dropout probability of the skip connection. (default: `0`)
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*, *edge_weight=None*) [source]

reset_parameters() [source]

class **SGConv**(*in_channels*, *out_channels*, *K=1*, *cached=False*, *bias=True*) [source]

The simgple graph convolutional operator from the "Simplifying Graph Convolutional Networks" paper

$$\mathbf{X}' = \left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\right)^K\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **K** (*int*, *optional*) – Number of hops K. (default: `1`)
- **cached** (*bool*, *optional*) – If set to `True`, the layer will cache the computation of $\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\right)^K\mathbf{X}$. (default: `False`)
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*, *edge_weight=None*) [source]

reset_parameters() [source]

class **APPNP**(*K*, *alpha*, *bias=True*) [source]

The approximate personalized propagation of neural predictions layer from the "Predict then Propagate: Graph Neural Networks meet Personalized PageRank" paper

$$\begin{aligned}\mathbf{X}^{(0)} &= \mathbf{X} \\ \mathbf{X}^{(k)} &= (1-\alpha)\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}^{(k-1)} + \alpha\mathbf{X}^{(0)} \\ \mathbf{X}' &= \mathbf{X}^{(K)},\end{aligned}$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix.

Parameters:

- **K** (*int*) – Number of iterations K.
- **alpha** (*float*) – Teleport probability α.
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*, *edge_weight=None*) [source]

---

class **RGCNConv**(*in_channels*, *out_channels*, *num_relations*, *num_bases*, *bias=True*) [source]

The relational graph convolutional operator from the "Modeling Relational Data with Graph Convolutional Networks" paper

$$\mathbf{x}_i' = \mathbf{\Theta}_0 \cdot \mathbf{x}_i + \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_r(i)} \frac{1}{|\mathcal{N}_r(i)|} \mathbf{\Theta}_r \cdot \mathbf{x}_j,$$

where $\mathcal{R}$ denotes the set of relations, *i.e.* edge types. Edge type needs to be a one-dimensional `torch.long` tensor which stores a relation identifier $\in \{0, \ldots, |\mathcal{R}| - 1\}$ for each edge.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **num_relations** (*int*) – Number of relations.
- **num_bases** (*int*) – Number of bases used for basis-decomposition.
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*, *edge_type*, *edge_norm=None*) [source]

reset_parameters() [source]

---

class **SignedConv**(*in_channels*, *out_channels*, *first_aggr*, *bias=True*) [source]

The signed graph convolutional operator from the "Signed Graph Convolutional Network" paper

$$\mathbf{x}_v^{(\text{pos})} = \mathbf{\Theta}^{(\text{pos})} \left[\frac{1}{|\mathcal{N}^+(v)|} \sum_{w \in \mathcal{N}^+(v)} \mathbf{x}_w, \mathbf{x}_v \right]$$

$$\mathbf{x}_v^{(\text{neg})} = \mathbf{\Theta}^{(\text{neg})} \left[\frac{1}{|\mathcal{N}^-(v)|} \sum_{w \in \mathcal{N}^-(v)} \mathbf{x}_w, \mathbf{x}_v \right]$$

if `first_aggr` is set to `True`, and

$$\mathbf{x}_v^{(\text{pos})} = \mathbf{\Theta}^{(\text{pos})} \left[\frac{1}{|\mathcal{N}^+(v)|} \sum_{w \in \mathcal{N}^+(v)} \mathbf{x}_w^{(\text{pos})}, \frac{1}{|\mathcal{N}^-(v)|} \sum_{w \in \mathcal{N}^-(v)} \mathbf{x}_w^{(\text{neg})}, \mathbf{x}_v^{(\text{pos})} \right]$$

$$\mathbf{x}_v^{(\text{neg})} = \mathbf{\Theta}^{(\text{pos})} \left[\frac{1}{|\mathcal{N}^+(v)|} \sum_{w \in \mathcal{N}^+(v)} \mathbf{x}_w^{(\text{neg})}, \frac{1}{|\mathcal{N}^-(v)|} \sum_{w \in \mathcal{N}^-(v)} \mathbf{x}_w^{(\text{pos})}, \mathbf{x}_v^{(\text{neg})} \right]$$

otherwise. In case `first_aggr` is `False`, the layer expects `x` to be a tensor where `x[:, :in_channels]` denotes the positive node features $\mathbf{X}^{(\text{pos})}$ and `x[:, in_channels:]` denotes the negative node features $\mathbf{X}^{(\text{neg})}$.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **first_aggr** (*bool*) – Denotes which aggregation formula to use.
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *pos_edge_index*, *neg_edge_index*) [source]

reset_parameters() [source]

---

class **DNAConv**(*channels*, *heads=1*, *groups=1*, *dropout=0*, *cached=False*, *bias=True*) [source]

The dynamic neighborhood aggregation operator from the "Just Jump: Towards Dynamic Neighborhood Aggregation in Graph Neural Networks" paper

$$\mathbf{x}_v^{(t)} = h_{\mathbf{\Theta}}^{(t)} \left(\mathbf{x}_{v \leftarrow v}^{(t)}, \left\{ \mathbf{x}_{v \leftarrow w}^{(t)} : w \in \mathcal{N}(v) \right\} \right)$$

based on (multi-head) dot-product attention

$$\mathbf{x}_{v \leftarrow w}^{(t)} = \text{Attention} \left(\mathbf{x}_v^{(t-1)} \, \mathbf{\Theta}_Q^{(t)}, [\mathbf{x}_w^{(1)}, \ldots, \mathbf{x}_w^{(t-1)}] \, \mathbf{Q}_K^{(t)}, [\mathbf{x}_w^{(1)}, \ldots, \mathbf{x}_w^{(t-1)}] \, \mathbf{Q}_V^{(t)} \right)$$

with $\mathbf{\Theta}_Q^{(t)}, \mathbf{\Theta}_K^{(t)}, \mathbf{\Theta}_V^{(t)}$ denoting (grouped) projection matrices for query, key and value information, respectively. $h_{\mathbf{\Theta}}^{(t)}$ is implemented as a non-trainable version of `torch_geometric.nn.conv.GCNConv`.

Note

In contrast to other layers, this operator expects node features as shape `[num_nodes, num_layers, channels]`.

Parameters:

- **channels** (*int*) – Size of each input/output sample.
- **heads** (*int*, *optional*) – Number of multi-head-attentions. (default: `1`)
- **groups** (*int*, *optional*) – Number of groups to use for all linear projections. (default: `1`)
- **dropout** (*float*, *optional*) – Dropout probability of attention coefficients. (default: `0`)
- **cached** (*bool*, *optional*) – If set to `True`, the layer will cache the computation of $\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\right)$. (default: `False`)
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *edge_index*, *edge_weight=None*) [source]

reset_parameters() [source]

---

class **PointConv**(*local_nn=None*, *global_nn=None*) [source]

The PointNet set layer from the "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation" and "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space" papers

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\max_{j \in \mathcal{N}(i) \cup \{i\}} h_{\mathbf{\Theta}}(\mathbf{x}_j, \mathbf{p}_j - \mathbf{p}_i) \right),$$

where $\gamma_{\mathbf{\Theta}}$ and $h_{\mathbf{\Theta}}$ denote neural networks, *.i.e.* MLPs, and $\mathbf{P} \in \mathbb{R}^{N \times D}$ defines the position of each point.

Parameters:

- **local_nn** (*nn.Sequential*, *optional*) – Neural network $h_{\mathbf{\Theta}}$. (default: `None`)
- **global_nn** (*nn.Sequential*, *optional*) – Neural network $\gamma_{\mathbf{\Theta}}$. (default: `None`)

forward(*x*, *pos*, *edge_index*) [source]

Parameters:

- **x** (*Tensor*) – The node feature matrix. Allowed to be `None`.
- **pos** (*Tensor or tuple*) – The node position matrix. Either given as tensor for use in general message passing or as tuple for use in message passing in bipartite graphs.
- **edge_index** (*LongTensor*) – The edge indices.

reset_parameters() [source]

---

class **GMMConv**(*in_channels, out_channels, dim, kernel_size, bias=True*) [source]

The gaussian mixture model convolutional operator from the "Geometric Deep Learning on Graphs and Manifolds using Mixture Model CNNs" paper

$$\mathbf{x}_i' = \mathbf{\Theta} \cdot \sum_{j \in \mathcal{N}(i) \cup \{i\}} \sum_{k=1}^{K} \mathbf{w}_k(\mathbf{e}_{i,j}) \odot \mathbf{x}_j,$$

where

$$\mathbf{w}_k(\mathbf{e}) = \exp\left(-\frac{1}{2}(\mathbf{e} - \mu_k)^\top \Sigma_k^{-1} (\mathbf{e} - \mu_k)\right)$$

denotes a weighting function based on trainable mean vector μ_k and diagonal covariance matrix $\mathbf{\Sigma}_k$.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **dim** (*int*) – Pseudo-coordinate dimensionality.
- **kernel_size** (*int*) – Number of kernels K.
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x, edge_index, pseudo*) [source]

reset_parameters() [source]

---

class **SplineConv**(*in_channels, out_channels, dim, kernel_size, is_open_spline=True, degree=1, norm=True, root_weight=True, bias=True*) [source]

The spline-based convolutional operator from the "SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels" paper

$$\mathbf{x}_i' = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j \cdot h_{\mathbf{\Theta}}(\mathbf{e}_{i,j}),$$

where $h_{\mathbf{\Theta}}$ denotes a kernel function defined over the weighted B-Spline tensor product basis.

Note

Pseudo-coordinates must lay in the fixed interval $[0, 1]$ for this method to work as intended.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **dim** (*int*) – Pseudo-coordinate dimensionality.
- **kernel_size** (*int or [int]*) – Size of the convolving kernel.
- **is_open_spline** (*bool or [bool], optional*) – If set to `False`, the operator will use a closed B-spline basis in this dimension. (default `True`)
- **degree** (*int, optional*) – B-spline basis degrees. (default: `1`)
- **norm** (*bool, optional*) – If set to `False`, output node features will not be degree-normalized. (default: `True`)
- **root_weight** (*bool, optional*) – If set to `False`, the layer will not add transformed root node features to the output. (default: `True`)
- **bias** (*bool, optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x, edge_index, pseudo*) [source]

reset_parameters() [source]

---

class **NNConv**(*in_channels, out_channels, nn, aggr='add', root_weight=True, bias=True*) [source]

The continuous kernel-based convolutional operator adapted from the "Neural Message Passing for Quantum Chemistry" paper

$$\mathbf{x}'_i = \mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j \cdot h_{\mathbf{\Theta}}(\mathbf{e}_{i,j}),$$

where $h_{\mathbf{\Theta}}$ denotes a neural network, *.i.e.* a MLP.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **nn** (*nn.Sequential*) – Neural network $h_{\mathbf{\Theta}}$.
- **aggr** (*string*) – The aggregation operator to use (`"add"`, `"mean"`, `"max"`). (default: `"add"`)
- **root_weight** (*bool, optional*) – If set to `False`, the layer will not add the transformed root node features to the output. (default: `True`)
- **bias** (*bool, optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x, edge_index, edge_attr*) [source]

reset_parameters() [source]

---

class **EdgeConv**(*nn, aggr='max'*) [source]

The edge convolutional operator from the "Dynamic Graph CNN for Learning on Point Clouds" paper

$$\mathbf{x}_i' = \sum_{j \in \mathcal{N}(i)} h_{\mathbf{\Theta}}(\mathbf{x}_i \,\|\, \mathbf{x}_j - \mathbf{x}_i),$$

where $h_{\mathbf{\Theta}}$ denotes a neural network, *.i.e.* a MLP.

Parameters:

- **nn** (*nn.Sequential*) – Neural network $h_{\mathbf{\Theta}}$.
- **aggr** (*string*) – The aggregation operator to use (`"add"`, `"mean"`, `"max"`). (default: `"max"`)

forward(*x*, *edge_index*) [source]

reset_parameters() [source]

---

class **XConv**(*in_channels*, *out_channels*, *dim*, *kernel_size*, *hidden_channels=None*, *dilation=1*, *bias=True*) [source]

The convolutional operator on $\mathcal{X}$-transformed points from the "PointCNN: Convolution On X-Transformed Points" paper

$$\mathbf{x}_i' = \mathrm{Conv}\left(\mathbf{K}, \gamma_{\mathbf{\Theta}}(\mathbf{P}_i - \mathbf{p}_i) \times \left(h_{\mathbf{\Theta}}(\mathbf{P}_i - \mathbf{p}_i) \,\|\, \mathbf{x}_i\right)\right),$$

where $\mathbf{K}$ and $\mathbf{P}_i$ denote the trainable filter and neighboring point positions of $\mathbf{x}_i$, respectively. $\gamma_{\mathbf{\Theta}}$ and $h_{\mathbf{\Theta}}$ describe neural networks, *i.e.* MLPs, where $h_{\mathbf{\Theta}}$ individually lifts each point into a higher-dimensional space, and $\gamma_{\mathbf{\Theta}}$ computes the $\mathcal{X}$- transformation matrix based on *all* points in a neighborhood.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **dim** (*int*) – Point cloud dimensionality.
- **kernel_size** (*int*) – Size of the convolving kernel, *i.e.* number of neighbors including self-loops.
- **hidden_channels** (*int*, *optional*) – Output size of $h_{\mathbf{\Theta}}$, *i.e.* dimensionality of lifted points. If set to `None`, will be automatically set to `in_channels / 4`. (default: `None`)
- **dilation** (*int*, *optional*) – The factor by which the neighborhood is extended, from which `kernel_size` neighbors are then uniformly sampled. Can be interpreted as the dilation rate of classical convolutional operators. (default: `1`)
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)

forward(*x*, *pos*, *batch=None*) [source]

`reset_parameters`() [source]

class **PPFConv**(*local_nn=None*, *global_nn=None*) [source]

The PPFNet operator from the "PPFNet: Global Context Aware Local Features for Robust 3D Point Matching" paper

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\max_{j \in \mathcal{N}(i) \cup \{i\}} h_{\mathbf{\Theta}}(\mathbf{x}_j, \|\mathbf{d_{j,i}}\|, \angle(\mathbf{n}_i, \mathbf{d_{j,i}}), \angle(\mathbf{n}_j, \mathbf{d_{j,i}}), \angle(\mathbf{n}_i, \mathbf{n}_j)) \right)$$

where $\gamma_{\mathbf{\Theta}}$ and $h_{\mathbf{\Theta}}$ denote neural networks, *.i.e.* MLPs, which takes in node features and `torch_geometric.transforms.PointPairFeatures`.

Parameters:

- **local_nn** (*nn.Sequential*, *optional*) – Neural network $h_{\mathbf{\Theta}}$. (default: `None`)
- **global_nn** (*nn.Sequential*, *optional*) – Neural network $\gamma_{\mathbf{\Theta}}$. (default: `None`)

`forward`(*x*, *pos*, *norm*, *edge_index*) [source]

Parameters:

- **x** (*Tensor*) – The node feature matrix. Allowed to be `None`.
- **pos** (*Tensor or tuple*) – The node position matrix. Either given as tensor for use in general message passing or as tuple for use in message passing in bipartite graphs.
- **norm** (*Tensor or tuple*) – The normal vectors of each node. Either given as tensor for use in general message passing or as tuple for use in message passing in bipartite graphs.
- **edge_index** (*LongTensor*) – The edge indices.

`reset_parameters`() [source]

class **MetaLayer**(*edge_model=None*, *node_model=None*, *global_model=None*) [source]

A meta layer for building any kind of graph network, inspired by the "Relational Inductive Biases, Deep Learning, and Graph Networks" paper.

A graph network takes a graph as input and returns an updated graph as output (with same connectivity). The input graph has node features `x`, edge features `edge_attr` as well as global-level features `u`. The output graph has the same structure, but updated features.

Edge features, node features as well as global features are updated by calling the functions `edge_model`, `node_model` and `global_model`, respectively.

To allow for batch-wise graph processing, all callable functions take an additional argument `batch`, which determines the assignment of edges or nodes to their specific graphs.

Parameters:

- **edge_model** (*func, optional*) – A callable which updates a graph's edge features based on its source and target node features, its current edge features and its global features. (default: `None`)
- **node_model** (*func, optional*) – A callable which updates a graph's node features based on its current node features, its graph connectivity, its edge features and its global features. (default: `None`)
- **global_model** (*func, optional*) – A callable which updates a graph's global features based on its node features, its graph connectivity, its edge features and its current global features.

```
from torch.nn import Sequential as Seq, Linear as Lin, ReLU
from torch_scatter import scatter_mean
from torch_geometric.nn import MetaLayer

class MyLayer(torch.nn.Module):
    def __init__(self):
        super(MyLayer, self).__init__()

        self.edge_mlp = Seq(Lin(..., ...), ReLU(), Lin(..., ...))
        self.node_mlp_1 = Seq(Lin(..., ...), ReLU(), Lin(..., ...))
        self.node_mlp_2 = Seq(Lin(..., ...), ReLU(), Lin(..., ...))
        self.global_mlp = Seq(Lin(..., ...), ReLU(), Lin(..., ...))

        def edge_model(src, dest, edge_attr, u, batch):
            # source, target: [E, F_x], where E is the number of edges.
            # edge_attr: [E, F_e]
            # u: [B, F_u], where B is the number of graphs.
            # batch: [E] with max entry B - 1.
            out = torch.cat([src, dest, edge_attr, u[batch]], 1)
            return self.edge_mlp(out)

        def node_model(x, edge_index, edge_attr, u, batch):
            # x: [N, F_x], where N is the number of nodes.
            # edge_index: [2, E] with max entry N - 1.
            # edge_attr: [E, F_e]
            # u: [B, F_u]
            # batch: [N] with max entry B - 1.
            row, col = edge_index
            out = torch.cat([x[col], edge_attr], dim=1)
            out = self.node_mlp(out)
            out = scatter_mean(out, row, dim=0, dim_size=x.size(0))
            out = torch.cat([out, u[batch]], dim=1)
            return self.node_mlp_2(out)

        def global_model(x, edge_index, edge_attr, u, batch):
            # x: [N, F_x], where N is the number of nodes.
            # edge_index: [2, E] with max entry N - 1.
            # edge_attr: [E, F_e]
            # u: [B, F_u]
            # batch: [N] with max entry B - 1.
            out = torch.cat([u, scatter_mean(x, batch, dim=0)], dim=1)
            return self.global_mlp(out)

        self.op = MetaLayer(edge_model, node_model, global_model)

    def forward(self, x, edge_index, edge_attr, u, batch):
        return self.op(x, edge_index, edge_attr, u, batch)
```

forward(*x*, *edge_index*, *edge_attr=None*, *u=None*, *batch=None*) [source]

reset_parameters() [source]

class **DenseSAGEConv**(*in_channels*, *out_channels*, *normalize=True*, *bias=True*) [source]

See `torch_geometric.nn.conv.SAGEConv`.

Return type: `Tensor`

forward(*x*, *adj*, *mask=None*, *add_loop=True*) [source]

Parameters:
- **x** (*Tensor*) – Node feature tensor $\mathbf{X} \in \mathbb{R}^{B \times N \times F}$, with batch-size B, (maximum) number of nodes N for each graph, and feature dimension F.
- **adj** (*Tensor*) – Adjacency tensor $\mathbf{A} \in \mathbb{R}^{B \times N \times N}$.
- **mask** (*ByteTensor*, *optional*) – Mask matrix $\mathbf{M} \in \{0, 1\}^{B \times N}$ indicating the valid nodes for each graph. (default: `None`)
- **add_loop** (*bool*, *optional*) – If set to `False`, the layer will not automatically add self-loops to the adjacency matrices. (default: `True`)

reset_parameters() [source]

Global Pooling Layers

global_add_pool(*x*, *batch*, *size=None*) [source]

Returns batch-wise graph-level-outputs by adding node features across the node node dimension, so that for a single graph $\mathcal{G}_i$ its output is computed by

$$\mathbf{r}_i = \sum_{n=1}^{N_i} \mathbf{x}_n$$

Parameters:
- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{(N_1 + \ldots + N_B) \times F}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example.
- **size** (*int*, *optional*) – Batch-size B. Automatically calculated if not given. (default: `None`)

Return type: `Tensor`

global_mean_pool(*x*, *batch*, *size=None*) [source]

Returns batch-wise graph-level-outputs by averaging node features across the node node dimension, so that for a single graph $\mathcal{G}_i$ its output is computed by

$$\mathbf{r}_i = \frac{1}{N_i} \sum_{n=1}^{N_i} \mathbf{x}_n$$

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{(N_1 + \ldots + N_B) \times F}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example.
- **size** (*int*, *optional*) – Batch-size B. Automatically calculated if not given. (default: `None`)

Return type: `Tensor`

`global_max_pool`(*x, batch, size=None*) [source]

Returns batch-wise graph-level-outputs by taking the channel-wise maximum across the node dimension, so that for a single graph $\mathcal{G}_i$ its output is computed by

$$\mathbf{r}_i = \max_{n=1}^{N_i} \mathbf{x}_n$$

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{(N_1 + \ldots + N_B) \times F}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example.
- **size** (*int*, *optional*) – Batch-size B. Automatically calculated if not given. (default: `None`)

Return type: `Tensor`

`global_sort_pool`(*x, batch, k*) [source]

The global pooling operator from the "An End-to-End Deep Learning Architecture for Graph Classification" paper, where node features are first sorted individually and then sorted in descending order based on their last features. The first k nodes form the output of the layer.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example.
- **k** (*int*) – The number of nodes to hold for each graph.

Return type: `Tensor`

---

class **GlobalAttention**(*gate_nn*, *nn=None*) [source]

Global soft attention layer from the "Gated Graph Sequence Neural Networks" paper

$$\mathbf{r}_i = \sum_{n=1}^{N_i} \mathrm{softmax}\left(h_{\mathrm{gate}}(\mathbf{x}_n)\right) \odot h_{\mathbf{\Theta}}(\mathbf{x}_n),$$

where $h_{\mathrm{gate}} \colon \mathbb{R}^F \to \mathbb{R}$ and $h_{\mathbf{\Theta}}$ denote neural networks, *i.e.* MLPS.

Parameters:
- **gate_nn** (*nn.Sequential*) – Neural network $h_{\mathrm{gate}} \colon \mathbb{R}^F \to \mathbb{R}$.
- **nn** (*nn.Sequential*, *optional*) – Neural network $h_{\mathbf{\Theta}}$. (default: `None`)

forward(*x*, *batch*, *size=None*) [source]

reset_parameters() [source]

class **Set2Set**(*in_channels*, *processing_steps*, *num_layers=1*) [source]

The global pooling operator based on iterative content-based attention from the "Order Matters: Sequence to sequence for sets" paper

$$\begin{aligned}
\mathbf{q}_t &= \mathrm{LSTM}(\mathbf{q}^*_{t-1}) \\
\alpha_{i,t} &= \mathrm{softmax}(\mathbf{x}_i \cdot \mathbf{q}_t) \\
\mathbf{r}_t &= \sum_{i=1}^{N} \alpha_{i,t} \mathbf{x}_i \\
\mathbf{q}^*_t &= \mathbf{q}_t \, \Vert \, \mathbf{r}_t,
\end{aligned}$$

where $\mathbf{q}^*_T$ defines the output of the layer with twice the dimensionality as the input.

Parameters:
- **in_channels** (*int*) – Size of each input sample.
- **processing_steps** (*int*) – Number of iterations T.
- **num_layers** (*int*, *optional*) – Number of recurrent layers, *.e.g*, setting `num_layers=2` would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. (default: `1`)

forward(*x*, *batch*) [source]

reset_parameters() [source]

Pooling Layers

class **TopKPooling**(*in_channels*, *ratio=0.5*) [source]

top_k pooling operator from the "Graph U-Net" and "Towards Sparse Hierarchical Graph Classifiers" papers

$$\begin{aligned}\mathbf{y} &= \frac{\mathbf{X}\mathbf{p}}{\|\mathbf{p}\|} \\ \mathbf{i} &= \text{top}_k(\mathbf{y}) \\ \mathbf{X}' &= (\mathbf{X} \odot \tanh(\mathbf{y}))_{\mathbf{i}} \\ \mathbf{A}' &= \mathbf{A}_{\mathbf{i},\mathbf{i}},\end{aligned}$$

where nodes are dropped based on a learnable projection score $\mathbf{p}$.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **ratio** (*float*) – Graph pooling ratio, which is used to compute $k = \lceil \text{ratio} \cdot N \rceil$. (default: `0.5`)

forward(*x, edge_index, edge_attr=None, batch=None*) [source]

reset_parameters() [source]

max_pool(*cluster, data, transform=None*) [source]

Pools and coarsens a graph given by the `torch_geometric.data.Data` object according to the clustering defined in `cluster`. All nodes within the same cluster will be represented as one node. Final node features are defined by the *maximum* features of all nodes within the same cluster, node positions are averaged and edge indices are defined to be the union of the edge indices of all nodes within the same cluster.

Parameters:

- **cluster** (*LongTensor*) – Cluster vector $\mathbf{c} \in \{0, \ldots, N-1\}^N$, which assigns each node to a specific cluster.
- **data** (*Data*) – Graph data object.
- **transform** (*callable, optional*) – A function/transform that takes in the coarsened and pooled `torch_geometric.data.Data` object and returns a transformed version. (default: `None`)

Return type: `torch_geometric.data.Data`

avg_pool(*cluster, data, transform=None*) [source]

Pools and coarsens a graph given by the `torch_geometric.data.Data` object according to the clustering defined in `cluster`. Final node features are defined by the *average* features of all nodes within the same cluster. See `torch_geometric.nn.pool.max_pool()` for more details.

Parameters:

- **cluster** (*LongTensor*) – Cluster vector $\mathbf{c} \in \{0, \ldots, N - 1\}^N$, which assigns each node to a specific cluster.
- **data** (*Data*) – Graph data object.
- **transform** (*callable*, *optional*) – A function/transform that takes in the coarsened and pooled `torch_geometric.data.Data` object and returns a transformed version. (default: `None`)

Return type: `torch_geometric.data.Data`

---

max_pool_x(*cluster*, *x*, *batch*, *size=None*) [source]

Max-Pools node features according to the clustering defined in `cluster`.

Parameters:

- **cluster** (*LongTensor*) – Cluster vector $\mathbf{c} \in \{0, \ldots, N - 1\}^N$, which assigns each node to a specific cluster.
- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{(N_1 + \ldots + N_B) \times F}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B - 1\}^N$, which assigns each node to a specific example.
- **size** (*int*, *optional*) – The maximum number of clusters in a single example. This property is useful to obtain a batch-wise dense representation, *e.g.* for applying FC layers, but should only be used if the size of the maximum number of clusters per example is known in advance. (default: `None`)

Return type: (`Tensor`, `LongTensor`) if `size` is `None`, else `Tensor`

---

avg_pool_x(*cluster*, *x*, *batch*, *size=None*) [source]

Average pools node features according to the clustering defined in `cluster`. See `torch_geometric.nn.pool.max_pool_x()` for more details.

Parameters:

- **cluster** (*LongTensor*) – Cluster vector $\mathbf{c} \in \{0, \ldots, N - 1\}^N$, which assigns each node to a specific cluster.
- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{(N_1 + \ldots + N_B) \times F}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B - 1\}^N$, which assigns each node to a specific example.
- **size** (*int*, *optional*) – The maximum number of clusters in a single example. (default: `None`)

Return type: (`Tensor`, `LongTensor`) if `size` is `None`, else `Tensor`

---

graclus(*edge_index*, *weight=None*, *num_nodes=None*) [source]

A greedy clustering algorithm from the "Weighted Graph Cuts without Eigenvectors: A Multilevel Approach" paper of picking an unmarked vertex and matching it with one of its unmarked neighbors (that maximizes its edge weight). The GPU algoithm is adapted from the "A GPU Algorithm for Greedy Graph Matching" paper.

Parameters:

- **edge_index** (*LongTensor*) – The edge indices.
- **weight** (*Tensor*, *optional*) – One-dimensional edge weights. (default: `None`)
- **num_nodes** (*int*, *optional*) – The number of nodes, *i.e.* `max_val + 1` of `edge_index`. (default: `None`)

Return type: `LongTensor`

`voxel_grid`(*pos, batch, size, start=None, end=None*) [source]

Voxel grid pooling from the, *e.g.*, Dynamic Edge-Conditioned Filters in Convolutional Networks on Graphs paper, which overlays a regular grid of user-defined size over a point cloud and clusters all points within the same voxel.

Parameters:

- **pos** (*Tensor*) – Node position matrix $\mathbf{X} \in \mathbb{R}^{(N_1 + \ldots + N_B) \times D}$.
- **batch** (*LongTensor*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example.
- **size** (*float* or *[float]*) – Size of a voxel (in each dimension).
- **start** (*float* or *[float]*, *optional*) – Start coordinates of the grid (in each dimension). If set to `None`, will be set to the minimum coordinates found in `pos`. (default: `None`)
- **end** (*float* or *[float]*, *optional*) – End coordinates of the grid (in each dimension). If set to `None`, will be set to the maximum coordinates found in `pos`. (default: `None`)

Return type: `LongTensor`

`fps`(*x, batch=None, ratio=0.5, random_start=True*) [source]

"A sampling algorithm from the "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space" paper, which iteratively samples the most distant point with regard to the rest points.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **batch** (*LongTensor*, *optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example. (default: `None`)
- **ratio** (*float*, *optional*) – Sampling ratio. (default: `0.5`)
- **random_start** (*bool*, *optional*) – If set to `False`, use the first node in $\mathbf{X}$ as starting node. (default: obj:*True*)

Return type: `LongTensor`

```
>>> x = torch.Tensor([[-1, -1], [-1, 1], [1, -1], [1, 1]])
>>> batch = torch.tensor([0, 0, 0, 0])
>>> index = fps(x, batch, ratio=0.5)
```

---

knn(*x, y, k, batch_x=None, batch_y=None*) [source]

Finds for each element in `y` the `k` nearest points in `x`.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **y** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{M \times F}$.
- **k** (*int*) – The number of neighbors.
- **batch_x** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example. (default: `None`)
- **batch_y** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^M$, which assigns each node to a specific example. (default: `None`)

Return type: `LongTensor`

```
>>> x = torch.Tensor([[-1, -1], [-1, 1], [1, -1], [1, 1]])
>>> batch_x = torch.tensor([0, 0, 0, 0])
>>> y = torch.Tensor([[-1, 0], [1, 0]])
>>> batch_x = torch.tensor([0, 0])
>>> assign_index = knn(x, y, 2, batch_x, batch_y)
```

---

knn_graph(*x, k, batch=None, loop=False*) [source]

Computes graph edges to the nearest `k` points.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **k** (*int*) – The number of neighbors.
- **batch** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example. (default: `None`)
- **loop** (*bool, optional*) – If `True`, the graph will contain self-loops. (default: `False`)

Return type: `LongTensor`

```
>>> x = torch.Tensor([[-1, -1], [-1, 1], [1, -1], [1, 1]])
>>> batch = torch.tensor([0, 0, 0, 0])
>>> edge_index = knn_graph(x, k=2, batch=batch, loop=False)
```

radius(*x, y, r, batch_x=None, batch_y=None, max_num_neighbors=32*) [source]

Finds for each element in `y` all points in `x` within distance `r`.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **y** (*Tensor*) – Node feature matrix $\mathbf{Y} \in \mathbb{R}^{M \times F}$.
- **r** (*float*) – The radius.
- **batch_x** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example. (default: `None`)
- **batch_y** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^M$, which assigns each node to a specific example. (default: `None`)
- **max_num_neighbors** (*int, optional*) – The maximum number of neighbors to return for each element in `y`. (default: `32`)

Return type: `LongTensor`

```
>>> x = torch.Tensor([[-1, -1], [-1, 1], [1, -1], [1, 1]])
>>> batch_x = torch.tensor([0, 0, 0, 0])
>>> y = torch.Tensor([[-1, 0], [1, 0]])
>>> batch_y = torch.tensor([0, 0])
>>> assign_index = radius(x, y, 1.5, batch_x, batch_y)
```

radius_graph(*x, r, batch=None, loop=False, max_num_neighbors=32*) [source]

Computes graph edges to all points within a given distance.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **r** (*float*) – The radius.
- **batch** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example. (default: `None`)
- **loop** (*bool, optional*) – If `True`, the graph will contain self-loops. (default: `False`)
- **max_num_neighbors** (*int, optional*) – The maximum number of neighbors to return for each element in `y`. (default: `32`)

Return type: `LongTensor`

```
>>> x = torch.Tensor([[-1, -1], [-1, 1], [1, -1], [1, 1]])
>>> batch = torch.tensor([0, 0, 0, 0])
>>> edge_index = radius_graph(x, r=1.5, batch=batch, loop=False)
```

nearest(*x, y, batch_x=None, batch_y=None*) [source]

Clusters points in `x` together which are nearest to a given query point in `y`.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **y** (*Tensor*) – Node feature matrix $\mathbf{Y} \in \mathbb{R}^{M \times F}$.
- **batch_x** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^N$, which assigns each node to a specific example. (default: `None`)
- **batch_y** (*LongTensor, optional*) – Batch vector $\mathbf{b} \in \{0, \ldots, B-1\}^M$, which assigns each node to a specific example. (default: `None`)

```
>>> x = torch.Tensor([[-1, -1], [-1, 1], [1, -1], [1, 1]])
>>> batch_x = torch.tensor([0, 0, 0, 0])
>>> y = torch.Tensor([[-1, 0], [1, 0]])
>>> batch_y = torch.tensor([0, 0])
>>> cluster = nearest(x, y, batch_x, batch_y)
```

dense_diff_pool(*x, adj, s, mask=None*) [source]

Differentiable pooling operator from the "Hierarchical Graph Representation Learning with Differentiable Pooling" paper

$$\mathbf{X}' = \mathrm{softmax}(\mathbf{S})^\top \cdot \mathbf{X}$$
$$\mathbf{A}' = \mathrm{softmax}(\mathbf{S})^\top \cdot \mathbf{A} \cdot \mathrm{softmax}(\mathbf{S})$$

based on dense learned assignments $\mathbf{S} \in \mathbb{R}^{B \times N \times C}$. Returns pooled node feature matrix, coarsened adjacency matrix and the auxiliary link prediction objective $\|\mathbf{A} - \mathrm{softmax}(\mathbf{S}) \cdot \mathrm{softmax}(\mathbf{S})^\top\|_F$.

Parameters:

- **x** (*Tensor*) – Node feature tensor $\mathbf{X} \in \mathbb{R}^{B \times N \times F}$ with batch-size B, (maximum) number of nodes N for each graph, and feature dimension F.
- **adj** (*Tensor*) – Adjacency tensor $\mathbf{A} \in \mathbb{R}^{B \times N \times N}$.
- **s** (*Tensor*) – Assignment tensor $\mathbf{S} \in \mathbb{R}^{B \times N \times C}$ with number of clusters C. The softmax does not have to be applied beforehand, since it is executed within this method.
- **mask** (*ByteTensor, optional*) – Mask matrix $\mathbf{M} \in \{0, 1\}^{B \times N}$ indicating the valid nodes for each graph. (default: `None`)

Return type: (`Tensor`, `Tensor`, `Tensor`, `Tensor`)

Unpooling Layers

knn_interpolate(*x, pos_x, pos_y, batch_x=None, batch_y=None, k=3*) [source]

The k-NN interpolation from the "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space" paper. For each point y with position $\mathbf{p}(y)$, its interpolated features $\mathbf{f}(y)$ are given by

$$\mathbf{f}(y) = \frac{\sum_{i=1}^{k} w(x_i)\mathbf{f}(x_i)}{\sum_{i=1}^{k} w(x_i)}, \text{ where } w(x_i) = \frac{1}{d(\mathbf{p}(y), \mathbf{p}(x_i))^2}$$

and $\{x_1, \ldots, x_k\}$ denoting the k nearest points to y.

Parameters:

- **x** (*Tensor*) – Node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$.
- **pos_x** (*Tensor*) – Node position matrix $\in \mathbb{R}^{N \times d}$.
- **pos_y** (*Tensor*) – Upsampled node position matrix $\in \mathbb{R}^{M \times d}$.
- **batch_x** (*LongTensor, optional*) – Batch vector $\mathbf{b_x} \in \{0, \ldots, B-1\}^N$, which assigns each node from $\mathbf{X}$ to a specific example. (default: `None`)
- **batch_y** (*LongTensor, optional*) – Batch vector $\mathbf{b_y} \in \{0, \ldots, B-1\}^N$, which assigns each node from $\mathbf{Y}$ to a specific example. (default: `None`)
- **k** (*int, optional*) – Number of neighbors. (default: `3`)

Models

class **`JumpingKnowledge`**(*mode, channels=None, num_layers=None*) [source]

The Jumping Knowledge layer aggregation module from the "Representation Learning on Graphs with Jumping Knowledge Networks" paper based on either **concatenation** (`"cat"`)

$$\mathbf{x}_v^{(1)} \, \| \, \ldots \, \| \, \mathbf{x}_v^{(T)}$$

max pooling (`"max"`)

$$\max\left(\mathbf{x}_v^{(1)}, \ldots, \mathbf{x}_v^{(T)}\right)$$

or **weighted summation**

$$\sum_{t=1}^{T} \alpha_v^{(t)} \mathbf{x}_v^{(t)}$$

with attention scores $\alpha_v^{(t)}$ obtained from a bi-directional LSTM (`"lstm"`).

Parameters:

- **mode** (*string*) – The aggregation scheme to use (`"cat"`, `"max"` or `"lstm"`).
- **channels** (*int*, *optional*) – The number of channels per representation. Needs to be only set for LSTM-style aggregation. (default: `None`)
- **num_layers** (*int*, *optional*) – The number of layers to aggregate. Needs to be only set for LSTM-style aggregation. (default: `None`)

`forward`(*xs*) [source]

Aggregates representations across different layers.

Parameters: **xs** (*list* or *tuple*) – List containing layer-wise representations.

`reset_parameters`() [source]

---

class `DeepGraphInfomax`(*hidden_channels*, *encoder*, *summary*, *corruption*) [source]

The Deep Graph Infomax model from the "Deep Graph Infomax" paper based on user-defined encoder and summary model $\mathcal{E}$ and $\mathcal{R}$ respectively, and a corruption function $\mathcal{C}$.

Parameters:

- **hidden_channels** (*int*) – The latent space dimensionality.
- **encoder** (*Module*) – The encoder module $\mathcal{E}$.
- **summary** (*callable*) – The readout function $\mathcal{R}$.
- **corruption** (*callable*) – The corruption function $\mathcal{C}$.

`discriminate`(*z*, *summary*, *sigmoid=True*) [source]

Given the patch-summary pair `z` and `summary`, computes the probability scores assigned to this patch-summary pair.

Parameters:

- **z** (*Tensor*) – The latent space.
- **sigmoid** (*bool*, *optional*) – If set to `False`, does not apply the logistic sigmoid function to the output. (default: `True`)

`forward`(**args*, ***kwargs*) [source]

Returns the latent space for the input arguments, their corruptions and their summary representation.

`loss`(*pos_z*, *neg_z*, *summary*) [source]

Computes the mutal information maximization objective.

`reset_parameters`() [source]

test(*train_z*, *train_y*, *test_z*, *test_y*, *solver='lbfgs'*, *multi_class='auto'*, **args*, ***kwargs*) [source]

Evaluates latent space quality via a logistic regression downstream task.

---

class **InnerProductDecoder** [source]

The inner product decoder from the "Variational Graph Auto-Encoders" paper

$$\sigma(\mathbf{Z}\mathbf{Z}^\top)$$

where $\mathbf{Z} \in \mathbb{R}^{N \times d}$ denotes the latent space produced by the encoder.

forward(*z*, *edge_index*, *sigmoid=True*) [source]

Decodes the latent variables `z` into edge probabilties for the given node-pairs `edge_index`.

Parameters:

- **z** (*Tensor*) – The latent space $\mathbf{Z}$.
- **sigmoid** (*bool*, *optional*) – If set to `False`, does not apply the logistic sigmoid function to the output. (default: `True`)

forward_all(*z*, *sigmoid=True*) [source]

Decodes the latent variables `z` into a probabilistic dense adjacency matrix.

Parameters:

- **z** (*Tensor*) – The latent space $\mathbf{Z}$.
- **sigmoid** (*bool*, *optional*) – If set to `False`, does not apply the logistic sigmoid function to the output. (default: `True`)

---

class **GAE**(*encoder*, *decoder=None*) [source]

The Graph Auto-Encoder model from the "Variational Graph Auto-Encoders" paper based on user-defined encoder and decoder models.

Parameters:

- **encoder** (*Module*) – The encoder module.
- **decoder** (*Module*, *optional*) – The decoder module. If set to `None`, will default to the `torch_geometric.nn.models.InnerProductDecoder`. (default: `None`)

decode(**args*, ***kwargs*) [source]

Runs the decoder and computes edge probabilties.

encode(**args*, ***kwargs*) [source]

Runs the encoder and computes node-wise latent variables.

recon_loss(*z*, *pos_edge_index*) [source]

Given latent variables `z`, computes the binary cross entropy loss for positive edges `pos_edge_index` and negative sampled edges.

Parameters:
- **z** (*Tensor*) – The latent space $\mathbf{Z}$.
- **pos_edge_index** (*LongTensor*) – The positive edges to train against.

reset_parameters() [source]

split_edges(*data*, *val_ratio=0.05*, *test_ratio=0.1*) [source]

Splits the edges of a `torch_geometric.data.Data` object into positve and negative train/val/test edges.

Parameters:
- **data** (*Data*) – The data object.
- **val_ratio** (*float*, *optional*) – The ratio of positive validation edges. (default: `0.05`)
- **test_ratio** (*float*, *optional*) – The ratio of positive test edges. (default: `0.1`)

test(*z*, *pos_edge_index*, *neg_edge_index*) [source]

Given latent variables `z`, positive edges `pos_edge_index` and negative edges `neg_edge_index`, computes area under the ROC curve (AUC) and average precision (AP) scores.

Parameters:
- **z** (*Tensor*) – The latent space $\mathbf{Z}$.
- **pos_edge_index** (*LongTensor*) – The positive edges to evaluate against.
- **neg_edge_index** (*LongTensor*) – The negative edges to evaluate against.

class **VGAE**(*encoder*, *decoder=None*) [source]

The Variational Graph Auto-Encoder model from the "Variational Graph Auto-Encoders" paper.

Parameters:
- **encoder** (*Module*) – The encoder module to compute μ and $\log \sigma^2$.
- **decoder** (*Module*, *optional*) – The decoder module. If set to `None`, will default to the `torch_geometric.nn.models.InnerProductDecoder`. (default: `None`)

encode(**args*, ***kwargs*) [source]

kl_loss(*mu=None*, *logvar=None*) [source]

Computes the KL loss, either for the passed arguments `mu` and `logvar`, or based on latent variables from last encoding.

Parameters:

- **mu** (*Tensor*, *optional*) – The latent space for μ. If set to `None`, uses the last computation of mu. (default: `None`)
- **logvar** (*Tensor*, *optional*) – The latent space for $\log \sigma^2$. If set to `None`, uses the last computation of $\log \sigma^2$.(default: `None`)

reparametrize(*mu*, *logvar*) [source]

class **ARGA**(*encoder*, *discriminator*, *decoder=None*) [source]

The Adversarially Regularized Graph Auto-Encoder model from the "Adversarially Regularized Graph Autoencoder for Graph Embedding" paper. paper.

Parameters:

- **encoder** (*Module*) – The encoder module.
- **discriminator** (*Module*) – The discriminator module.
- **decoder** (*Module*, *optional*) – The decoder module. If set to `None`, will default to the `torch_geometric.nn.models.InnerProductDecoder`. (default: `None`)

discriminator_loss(*z*) [source]

Computes the loss of the discriminator.

Parameters: **z** (*Tensor*) – The latent space $\mathbf{Z}$.

reg_loss(*z*) [source]

Computes the regularization loss of the encoder.

Parameters: **z** (*Tensor*) – The latent space $\mathbf{Z}$.

reset_parameters() [source]

class **ARGVA**(*encoder*, *discriminator*, *decoder=None*) [source]

The Adversarially Regularized Variational Graph Auto-Encoder model from the "Adversarially Regularized Graph Autoencoder for Graph Embedding" paper. paper.

Parameters:

- **encoder** (*Module*) – The encoder module to compute μ and $\log \sigma^2$.
- **discriminator** (*Module*) – The discriminator module.
- **decoder** (*Module*, *optional*) – The decoder module. If set to `None`, will default to the `torch_geometric.nn.models.InnerProductDecoder`. (default: `None`)

encode(**args*, ***kwargs*) [source]

kl_loss(*mu=None*, *logvar=None*) [source]

reparametrize(*mu*, *logvar*) [source]

---

class **SignedGCN**(*in_channels*, *hidden_channels*, *num_layers*, *lamb=5*, *bias=True*) [source]

The signed graph convolutional network model from the "Signed Graph Convolutional Network" paper. Internally, this module uses the `torch_geometric.nn.conv.SignedConv` operator.

Parameters:

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **num_layers** (*int*) – Number of layers.
- **lamb** (*float*, *optional*) – Balances the contributions of the overall objective. (default: `5`)
- **bias** (*bool*, *optional*) – If set to `False`, all layers will not learn an additive bias. (default: `True`)

create_spectral_features(*pos_edge_index*, *neg_edge_index*, *num_nodes=None*) [source]

Creates `in_channels` spectral node features based on positive and negative edges.

Parameters:

- **pos_edge_index** (*LongTensor*) – The positive edge indices.
- **neg_edge_index** (*LongTensor*) – The negative edge indices.
- **num_nodes** (*int*, *optional*) – The number of nodes, *i.e.* `max_val + 1` of `pos_edge_index` and `neg_edge_index`. (default: `None`)

discriminate(*z*, *edge_index*) [source]

Given node embeddings `z`, classifies the link relation between node pairs `edge_index` to be either positive, negative or non-existent.

Parameters:

- **x** (*Tensor*) – The input node features.
- **edge_index** (*LongTensor*) – The edge indices.

forward(*x*, *pos_edge_index*, *neg_edge_index*) [source]

Computes node embeddings `z` based on positive edges `pos_edge_index` and negative edges `neg_edge_index`.

Parameters:
- **x** (*Tensor*) – The input node features.
- **pos_edge_index** (*LongTensor*) – The positive edge indices.
- **neg_edge_index** (*LongTensor*) – The negative edge indices.

loss(*z*, *pos_edge_index*, *neg_edge_index*) [source]

Computes the overall objective.

Parameters:
- **z** (*Tensor*) – The node embeddings.
- **pos_edge_index** (*LongTensor*) – The positive edge indices.
- **neg_edge_index** (*LongTensor*) – The negative edge indices.

neg_embedding_loss(*z*, *neg_edge_index*) [source]

Computes the triplet loss between negative node pairs and sampled non-node pairs.

Parameters:
- **z** (*Tensor*) – The node embeddings.
- **neg_edge_index** (*LongTensor*) – The negative edge indices.

nll_loss(*z*, *pos_edge_index*, *neg_edge_index*) [source]

Computes the discriminator loss based on node embeddings `z`, and positive edges `pos_edge_index` and negative nedges `neg_edge_index`.

Parameters:
- **z** (*Tensor*) – The node embeddings.
- **pos_edge_index** (*LongTensor*) – The positive edge indices.
- **neg_edge_index** (*LongTensor*) – The negative edge indices.

pos_embedding_loss(*z*, *pos_edge_index*) [source]

Computes the triplet loss between positive node pairs and sampled non-node pairs.

Parameters:
- **z** (*Tensor*) – The node embeddings.
- **pos_edge_index** (*LongTensor*) – The positive edge indices.

reset_parameters() [source]

split_edges(*edge_index*, *test_ratio=0.2*) [source]

Splits the edges `edge_index` into train and test edges.

Parameters:

- **edge_index** (*LongTensor*) – The edge indices.
- **test_ratio** (*float*, *optional*) – The ratio of test edges. (default: `0.2`)

test(*z*, *pos_edge_index*, *neg_edge_index*) [source]

Evaluates node embeddings `z` on positive and negative test edges by computing AUC and F1 scores.

Parameters:

- **z** (*Tensor*) – The node embeddings.
- **pos_edge_index** (*LongTensor*) – The positive edge indices.
- **neg_edge_index** (*LongTensor*) – The negative edge indices.

---

class **RENet**(*num_nodes*, *num_rels*, *hidden_channels*, *seq_len*, *num_layers=1*, *dropout=0*, *bias=True*) [source]

The Recurrent Event Network model from the "Recurrent Event Network for Reasoning over Temporal Knowledge Graphs" paper

$$f_{\Theta}(\mathbf{e}_s, \mathbf{e}_r, \mathbf{h}^{(t-1)}(s, r))$$

based on a RNN encoder

$$\mathbf{h}^{(t)}(s, r) = \mathrm{RNN}(\mathbf{e}_s, \mathbf{e}_r, g(\mathcal{O}_r^{(t)}(s)), \mathbf{h}^{(t-1)}(s, r))$$

where $\mathbf{e}_s$ and $\mathbf{e}_r$ denote entity and relation embeddings, and $\mathcal{O}_r^{(t)}(s)$ represents the set of objects interacted with subject s under relation r at timestamp t. This model implements g as the **Mean Aggregator** and f_{Θ} as a linear projection.

Parameters:

- **num_nodes** (*int*) – The number of nodes in the knowledge graph.
- **num_rel** (*int*) – The number of relations in the knowledge graph.
- **hidden_channels** (*int*) – Hidden size of node and relation embeddings.
- **seq_len** (*int*) – The sequence length of past events.
- **num_layers** (*int*, *optional*) – The number of recurrent layers. (default: `1`)
- **dropout** (*int*) – If non-zero, introduces a dropout layer before the final prediction. (default: `0`)
- **bias** (*bool*, *optional*) – If set to `False`, all layers will not learn an additive bias. (default: `True`)

forward(*data*) [source]

Given a `data` batch, computes the forward pass.

Parameters: **data** (*torch_geometric.data.Data*) – The input data, holding subject `sub`, relation `rel` and object `obj` information with shape `[batch_size]`. In addition, `data` needs to hold history information for subjects, given by a vector of node indices `h_sub` and their relative timestamps `h_sub_t` and batch assignments `h_sub_batch`. The same information must be given for objects (`h_obj`, `h_obj_t`, `h_obj_batch`).

static **pre_transform**(*seq_len*) [source]

Precomputes history objects

$$\{\mathcal{O}_r^{(t-k-1)}(s), \ldots, \mathcal{O}_r^{(t-1)}(s)\}$$

of a `torch_geometric.datasets.icews.EventDataset` with k denoting the sequence length `seq_len`.

reset_parameters() [source]

test(*logits*, *y*) [source]

Given ground-truth `y`, computes Mean Reciprocal Rank (MRR) and Hits at 1/3/10.

DataParallel layers

class **DataParallel**(*module, device_ids=None, output_device=None*) [source]

Implements data parallelism at the module level.

This container parallelizes the application of the given `module` by splitting a list of `torch_geometric.data.Data` objects and copying them as `torch_geometric.data.Batch` objects to each device. In the forward pass, the module is replicated on each device, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are summed into the original module.

The batch size should be larger than the number of GPUs used.

The parallelized `module` must have its parameters and buffers on `device_ids[0]`.

Note

You need to use the `torch_geometric.data.DataListLoader` for this module.

Parameters:

- **module** (*Module*) – Module to be parallelized.
- **device_ids** (*list of int or torch.device*) – CUDA devices. (default: all devices)
- **output_device** (*int or torch.device*) – Device location of output. (default: `device_ids[0]`)

forward(*data_list*) [source]

scatter(*data_list*, *device_ids*) [source]