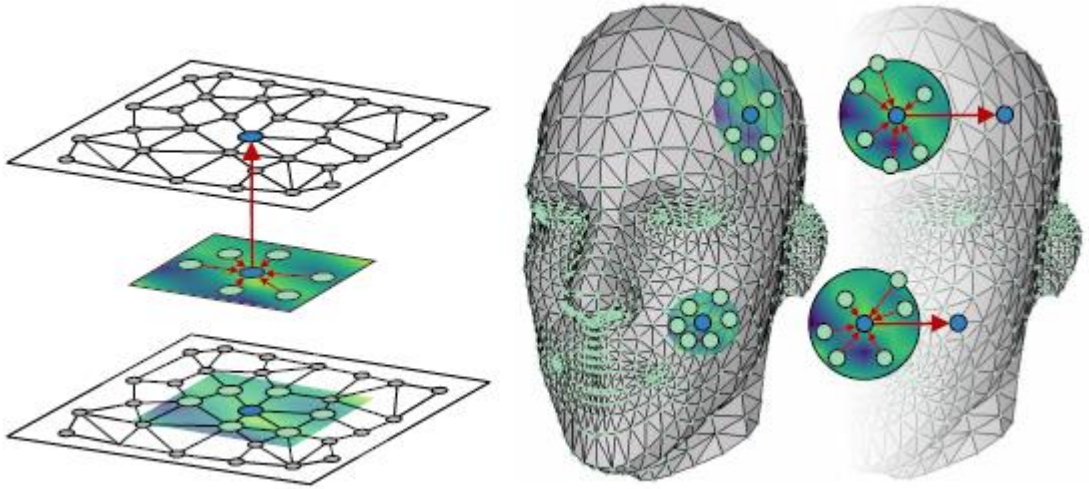
Geometric Graph Convolutional Neural Networks

Bong Jun Lee

Seoul National University

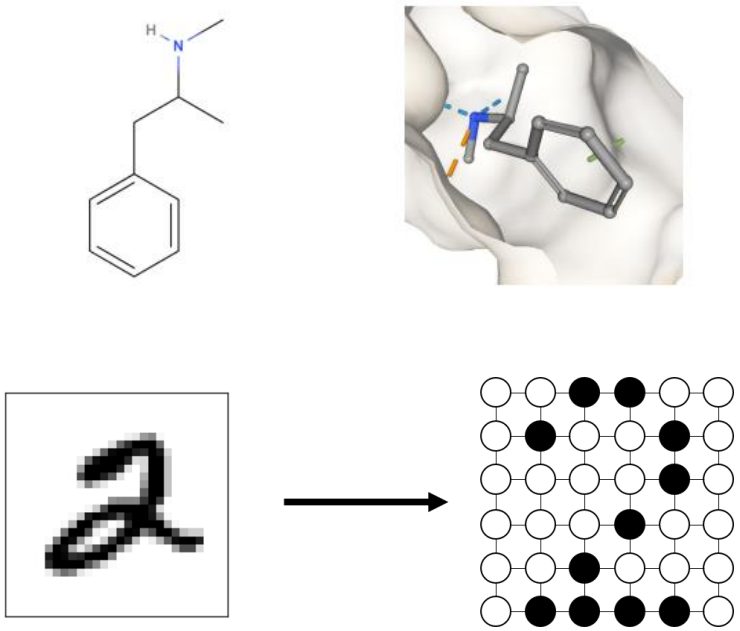
Geometric Graph Convolutional Neural Networks
Przemysław Spurek, Tomasz Danel, Jacek Tabor, Marek Śmieja, Łukasz Struski, Agnieszka Słowik, Łukasz Maziarka
https://arxiv.org/abs/1909.05310

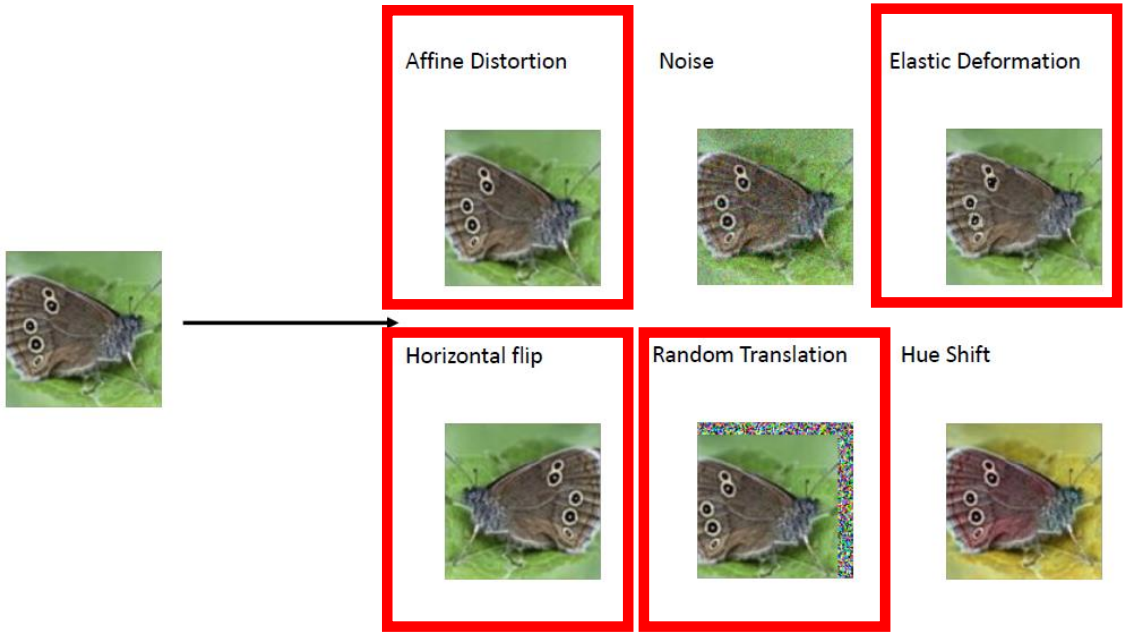
Geometric Graph Convolutional Neural Networks



Geometric Deep Learning: geo-GCN

- How to use geometric features (spatial coordinates) in GCNs
- A proper generalization of GCNs and CNNs
- Perform graph augmentation, which further improves performance of the model

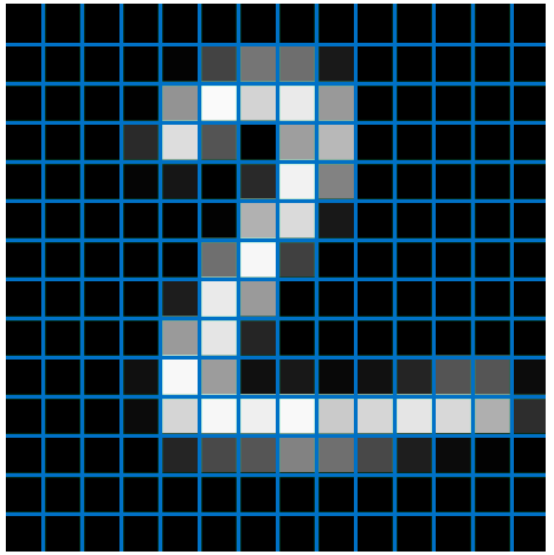




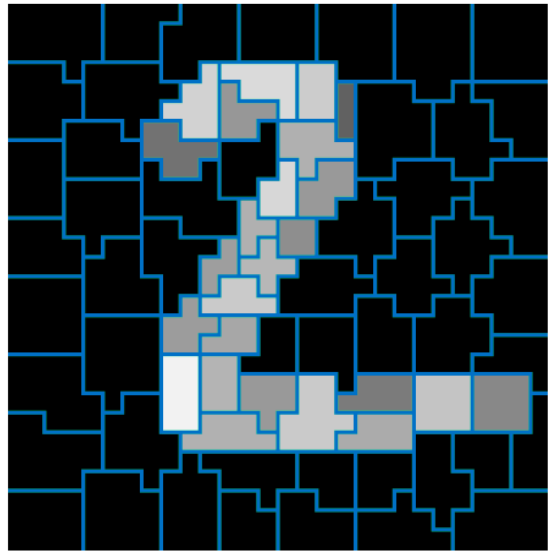
Datasets

Superpixels

- Provide an efficient low/mid-level representation of image data, which greatly reduces the number of image primitives for subsequent vision tasks.
- Vertices correspond to (super)pixels and edges represent their spatial relations
- Each image is represented as an embedded graph of 75 nodes defining the centroids of superpixels



Regular grid



Superpixels

Image

Superpixels

Segmented Labels

Formalization of geo-GCN

- Assume each node v_i is additionally identified with its coordinates $p_i \in \mathbb{R}^T$

$$\bar{h}_i(u,b) = \sum_{j \in N_p} ReLU\left(u^T(p_j - p_i) + b\right) h_j$$

where $u \in \mathbb{R}^T, b \in \mathbb{R}$ are trainable

- Let $U = [u_1, \dots, u_k]$ and $b = [b_1, \dots, b_k]$ define k-filters
- The intermediate representation h_i is a vector defined by:

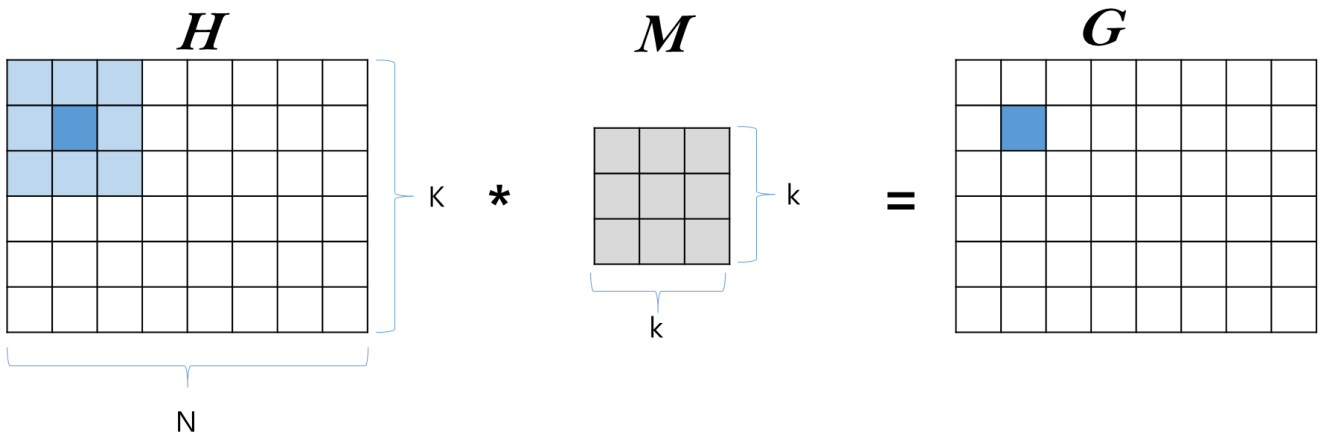
$$\bar{h}_i = \left[\bar{h}_i(u_1, b_1), \dots, \bar{h}_i(u_k, b_k)\right]$$

$$\text{MLP}\,(\bar{H}; W) = ReLU(W^T \bar{H} + b)$$

Theoretical Analysis of geo-GCN

- Theorem
  - Let $M = [m_{i'j'}]_{i',j' \in \{-k..k\}}$ be a given convolutional mask
  - Let $n = (2k+1)^2$ (number of elements of M)
  - Then there exist $u \in \mathbb{R}^2, b_1, \ldots, b_n \in \mathbb{R}$ and $w \in \mathbb{R}^2$ such that

$$M * H = \sum_{i=1}^{n} w_i \bar{H}(u, b_i)$$



Theoretical Analysis of geo-GCN

- Proof
  - Let $P \subset \mathbb{R}^2$ denote all possible positions in the mask M, i.e. $P = [i'j']^T : i', j' \in \{-k..k\}$
  - Let $u \in \mathbb{R}^2$ denote an arbitrary vector which is not orthogonal to any element from $P - P$. Then

$$u^T p \neq u^T q \text{ for } p, q \in P, p \neq q$$

  - Consequently, we may order the elements of P so that $u^T p_1 > \ldots > u^T p_n$.
  - Let M_i denote the convolutional mask, which has value one at the position p_i, and zero otherwise.
  - Now we can choose arbitrary b_i such that

$$b_i \in (-u^T p_i, -u^T p_{i+1}) \text{ for } i = 1..n-1, b_n > -u^T p_n$$

$$\text{for example } b_i = -u^T \frac{p_i + p_{i+1}}{2} \text{ for } i < n, b_n > -u^T p_n + 1$$

  - Then observe that

Theoretical Analysis of geo-GCN (cont.)

$$\bar{H}(u, b_1) = (u^T p_1 + b_1) \cdot M_1 * H$$
$$\bar{H}(u, b_2) = (u^T p_1 + b_2) \cdot M_1 + (u^T p_2 + b_2) \cdot M_2 * H$$

- and generally for every $k = 1..n$ we get

$$\bar{H}(u, b_k) = \sum_{i=1}^{k} (u^T p_i + b_k) \cdot M_i * H$$

- where all the coefficients in the above sum are strictly positive. Consequently,

$$M_1 * H = \frac{\bar{H}(u, b_1)}{u^T p_1 + b_1}$$

$$M * H = \sum_{i=1}^{n} w_i \bar{H}(u, b_i)$$

- and we obtain recursively that

$$M_k * H = \frac{1}{u^T p_k + b_k} \bar{H}(u, b_k) - \frac{1}{u^T p_k + b_k} \sum_{i=1}^{k-1} (u^T p_i + b_k) \cdot M_i * H$$

- which trivially implies that every convolution $M_k * H$ can be obtained as a linear combination of $\bar{H}(u, b_i)_{i=1..k}$

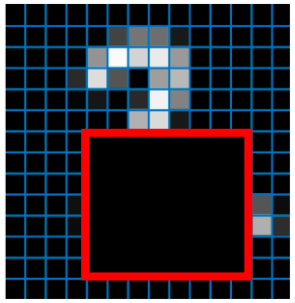
Experiment Result #1

- Image graph classification
  - Grid: regular grid with connections between adjacent pixels which have 2-dimensional location, and it is characterized by a 1- dimensional pixel intensity.
  - Superpixels: irregular grid consisting of 75 superpixels, where the edges are determined by spatial relations between nodes using k-nearest neighbors.

Table 1: Classification accuracy on two graph representations of MNIST.

Method	Grid	Superpixels
ChebNet	99.14%	75.62%
MoNet	99.19%	91.11%
SplineCNN	99.22%	95.22%
geo-GCN	**99.36%**	**95.95%**

Experiment Result #2



- Incomplete image classification
  - MNIST dataset, where a square patch of the size 13x13 was removed from each image
  - The location of the patch was uniformly sampled for each image
  - Imputation Methods
    - mean: Missing features were replaced with mean values of those features computed for all (incomplete) training samples.
    - k-nn: Missing attributes were filled with mean values of those features computed fro m the k nearest training samples (we used K = 5). Neighborhood was measured usin g Euclidean distance in the subspace of observed features.
    - mice: This method fills absent pixels in an iterative process using Multiple Imputation by Chained Equation (mice), where several imputations are drawing from the conditi onal distribution of data by Markov chain Monte Carlo techniques

Experiment Result #2 (cont.)

- Incomplete image classification

Table 2: Classification accuracy of graph representations of incomplete MNIST images.

Method	Accuracy
FCNet + mean	87.59%
FCNet + k-NN	87.10%
FCNet + mice	88.59%
ConvNet + mean	90.95%
ConvNet + k-NN	90.67%
ConvNet + mice	92.10%
geo-GCN	**92.40%**

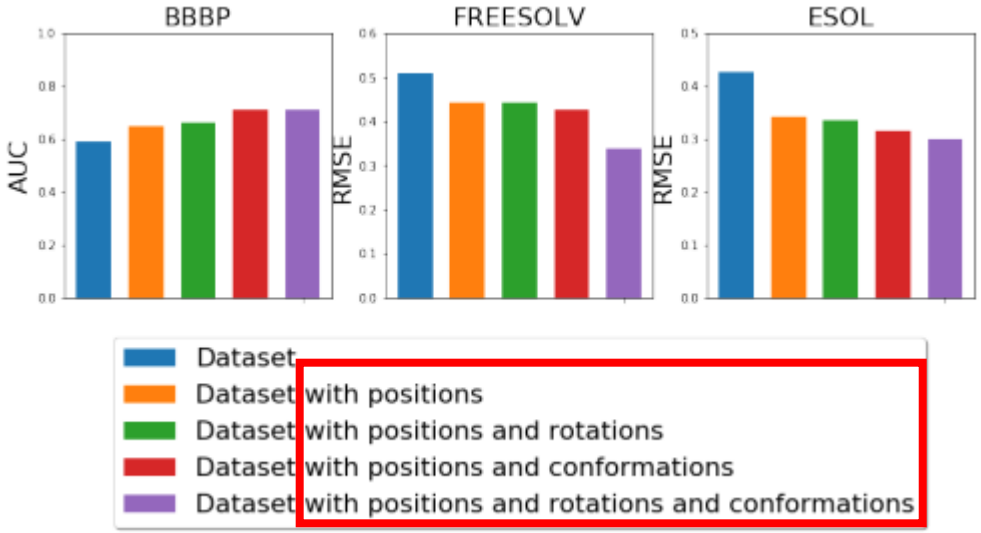
Experiment Result #3

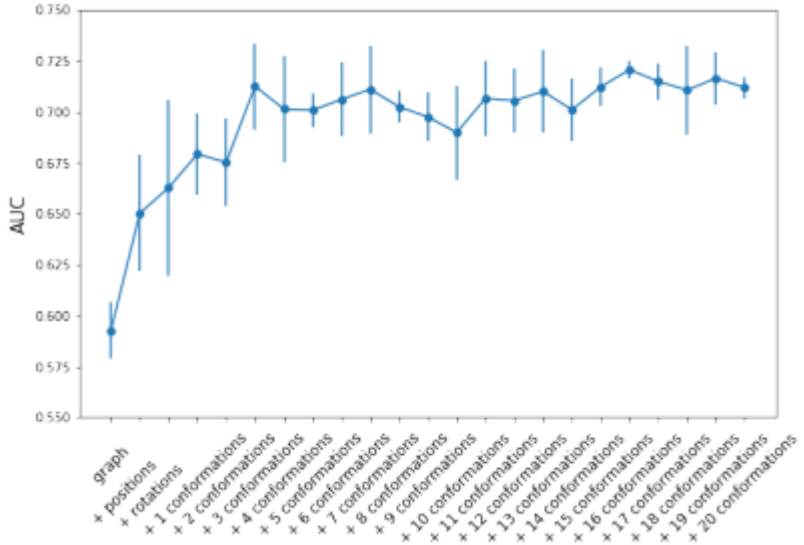
- Learning from molecules
  - 3 datasets from MoleculeNet
    - Blood-Brain Barrier Permeability (BBBP) is a binary classification task of predicting wh ether or not a given compound is able to pass through the barrier between blood a nd the brain, allowing the drug to impact the central nervous system. The ability of a molecule to penetrate this border depends on many different properties such as lipo philicity, molecule size, and its flexibility.
    - Another 2 datasets, ESOL and FreeSolv, are solubility prediction tasks with continuou s targets.

Method	BBBP	ESOL	FreeSolv
SVM	0.603 ± 0.000	0.493 ± 0.000	0.391 ± 0.000
RF	0.551 ± 0.005	0.533 ± 0.003	0.550 ± 0.004
GC	0.690 ± 0.015	0.334 ± 0.017	**0.336 ± 0.043**
Weave	0.703 ± 0.012	0.389 ± 0.045	0.403 ± 0.035
MPNN	0.700 ± 0.019	0.303 ± 0.012	**0.299 ± 0.038**
EAGCN	0.664 ± 0.007	0.459 ± 0.019	0.410 ± 0.014
pos-GCN	0.696 ± 0.008	0.301 ± 0.011	**0.278 ± 0.024**
geo-GCN	**0.743 ± 0.004**	**0.270 ± 0.005**	**0.299 ± 0.033**

Ablation study of the data augmentation

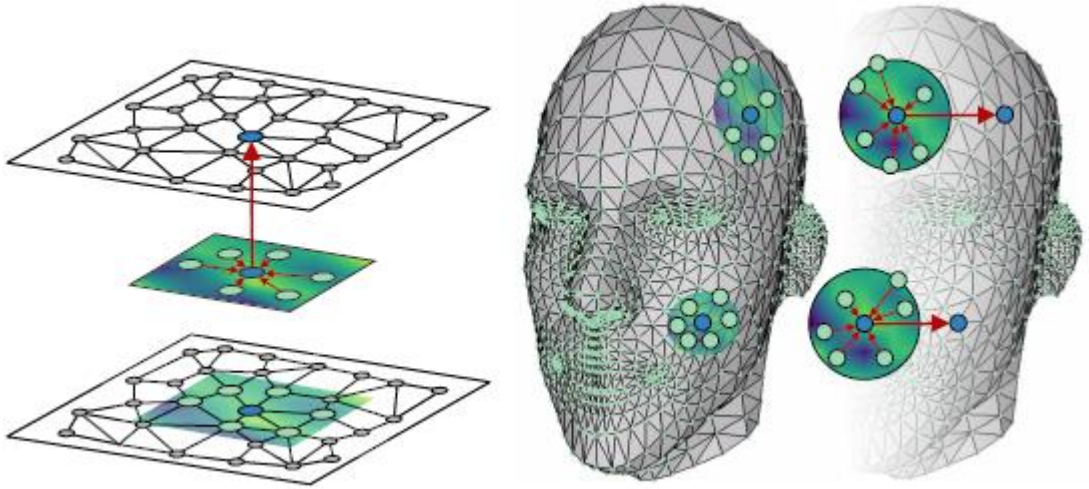
- Removing predicted positions, and thus setting all positional vectors to zero in $\bar{h}_i(u,b) = \sum_{j \in N_p} ReLU\left(u^T(p_j - p_i) + b\right)h_j$
- The outcome of augmenting the data with random rotations and 30 predicted molecule conformations





Thank you

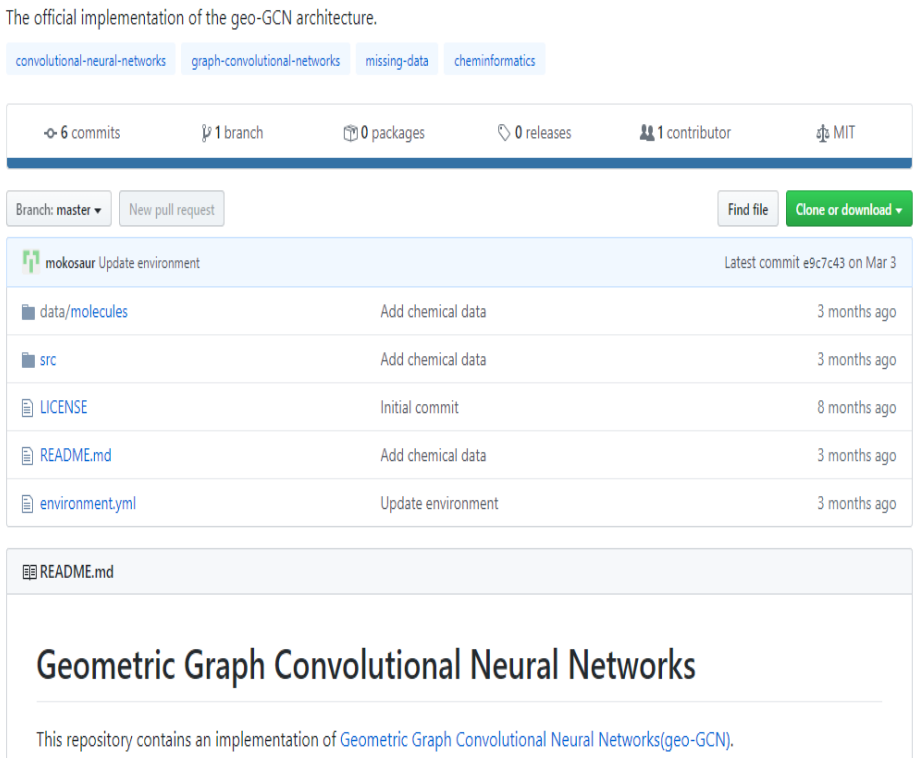
User Guide & Reproduction



How to Set Up: Source Code

geo-GCN

- Install anaconda environment
- Download the official implementation of the geo-GCN architecture made by author.
  - $ git clone https://github.com/gmum/geo-gcn.git
- Go to the src directory and edit environment.yml as follows
  - - python=3.8.2=h9d8afe_1_cpython
  - + python
- Create a new environment
  - $ conda env create -f environment.yml
- Activate geogcn environment
  - $ conda activate geogcn

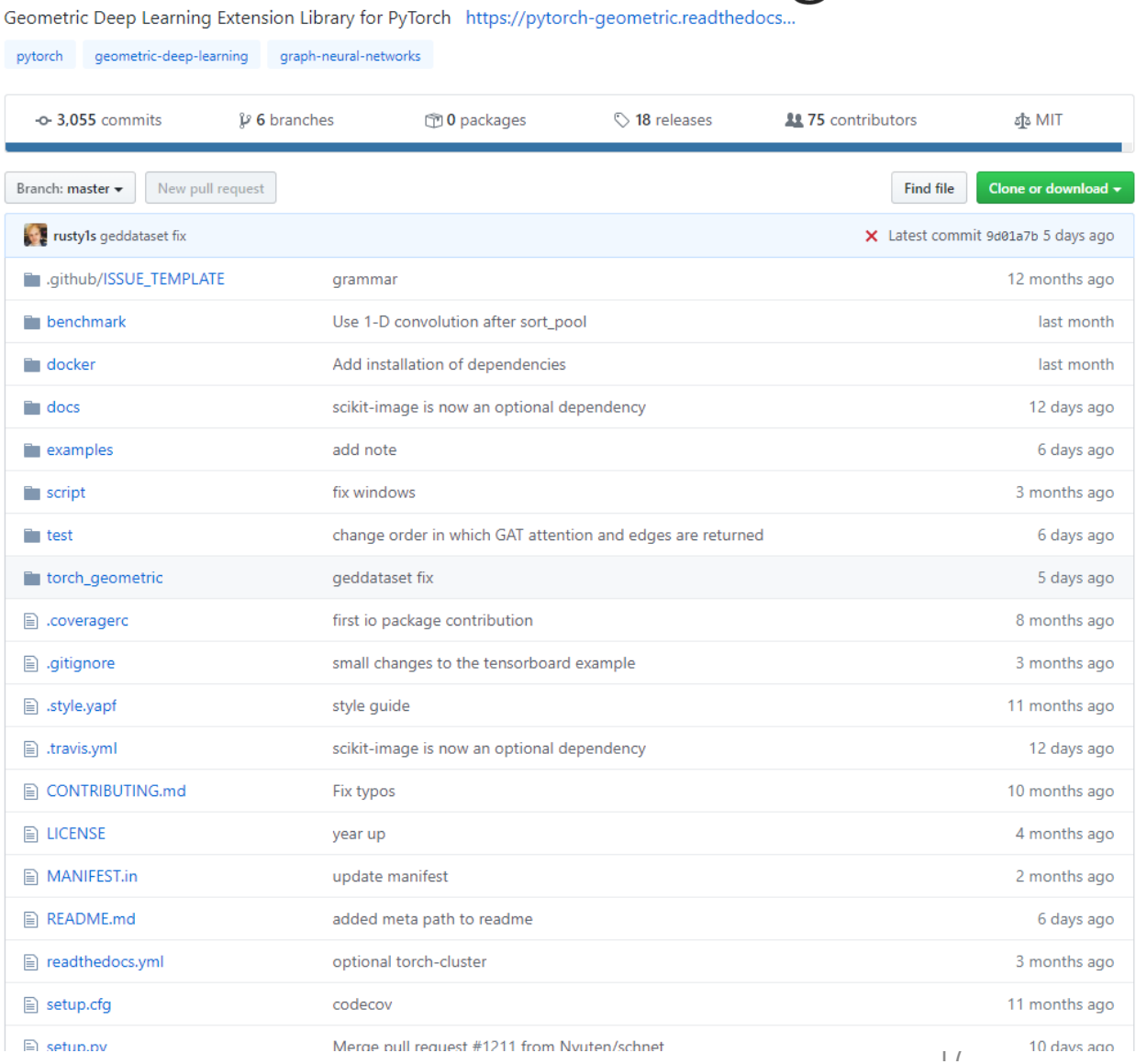


link: https://github.com/gmum/geo-gcn

How to Set Up: Extension Library

pytorch_geometric

- Visit the extension library webpage
  - https://github.com/rusty1s/pytorch_geometric
- Check pytorch version and set variable as fool ows
  - $ python -c “import torch; print(torch.__version__)”
  - $ export CUDA=cu92
- Install pytorch_geometric extension following th e instruction based on pytorch version
  - $ pip install torch-scatter==latest+${CUDA} -f https: //pytorch-geometric.com/whl/torch-1.4.0.html
  - $ pip install torch-sparse==latest+${CUDA} -f https: //pytorch-geometric.com/whl/torch-1.4.0.html
  - $ pip install torch-cluster==latest+${CUDA} -f https: //pytorch-geometric.com/whl/torch-1.4.0.html
  - $ pip install torch-spline-conv==latest+${CUDA} -f https://pytorch-geometric.com/whl/torch-1.4.0.html
  - $ pip install torch-geometric



How to Run & Reproducing Result (Superpixels)

Execution of the source code

- Run geo-GCN on MNISTSuperpixels with default parameters
  - $ python train_models.py MNISTSuperpixels
  - ► [note] other options are not available in the official github except ‘MNISTSuperpixels’.

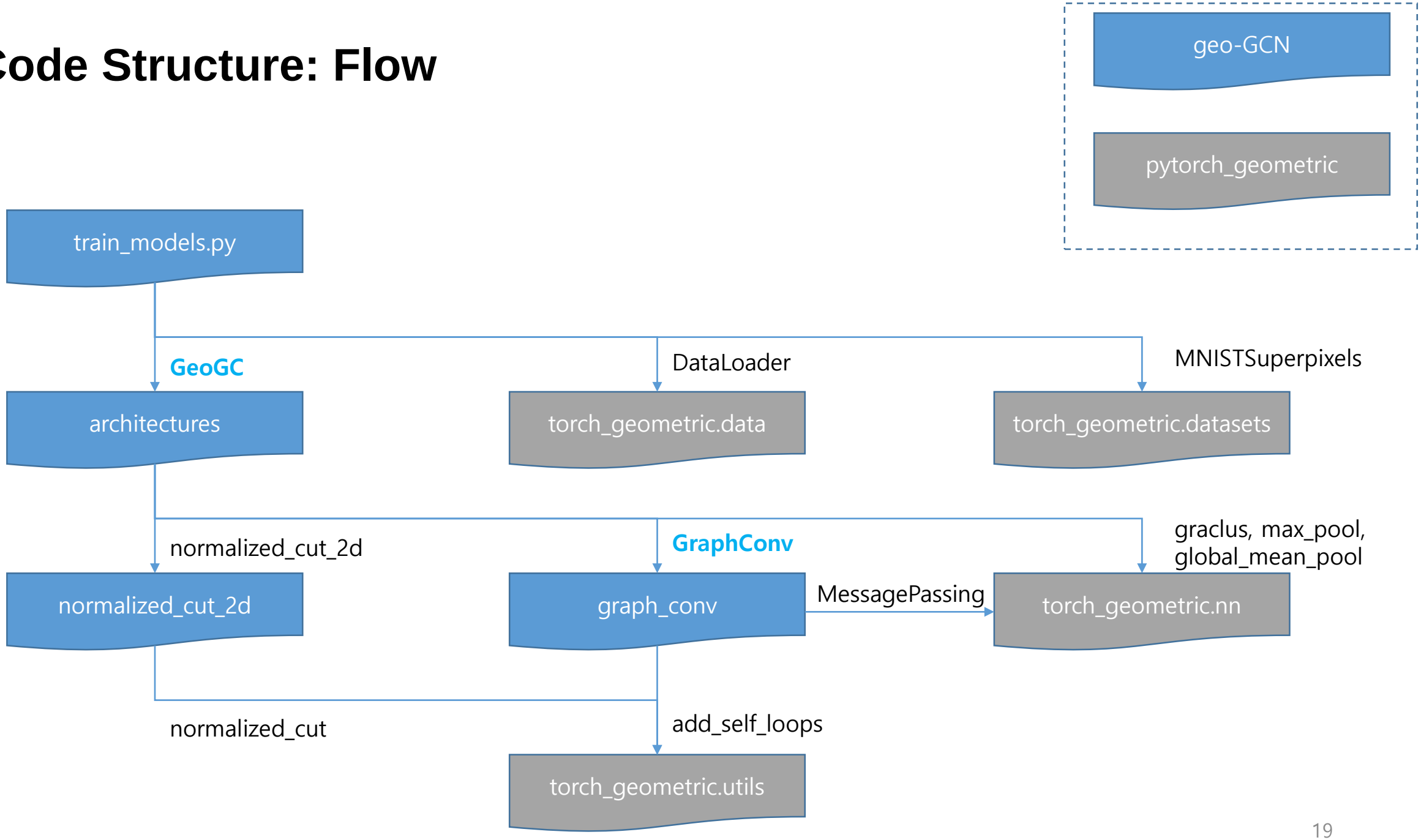
Reproducing Result

- The result outputs with ‘Loss’, ‘Train Acc’, ‘Test Acc’

Reproducing Output

```
Epoch: 001, Loss: 6.06750, Train Acc: 0.11232, Test Acc: 0.11340
Epoch: 002, Loss: 2.30311, Train Acc: 0.11235, Test Acc: 0.11350
Epoch: 003, Loss: 2.30176, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 004, Loss: 2.30125, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 005, Loss: 2.30138, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 006, Loss: 2.30143, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 007, Loss: 2.30123, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 008, Loss: 2.30128, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 009, Loss: 2.30131, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 010, Loss: 2.30144, Train Acc: 0.11237, Test Acc: 0.11350
…
Epoch: 090, Loss: 2.30123, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 091, Loss: 2.30124, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 092, Loss: 2.30124, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 093, Loss: 2.30124, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 094, Loss: 2.30123, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 095, Loss: 2.30124, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 096, Loss: 2.30124, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 097, Loss: 2.30123, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 098, Loss: 2.30123, Train Acc: 0.11237, Test Acc: 0.11350
Epoch: 099, Loss: 2.30123, Train Acc: 0.11237, Test Acc: 0.11350
```

Code Structure: Flow



Code Structure: GeoGC

```
from graph_conv import GraphConv
from normalized_cut_2d import normalized_cut_2d
from torch_geometric.nn import graclus, max_pool, global_mean_pool

class GeoGC(torch.nn.Module):
    def __init__(self, dim_coor, out_dim, input_features,
                 layers_num, model_dim, out_channels_1, dropout,
                 use_cluster_pooling):
        self.conv_layers = [GraphConv(coors=dim_coor,
                                      …
                                      dropout=dropout)] + \
                           [GraphConv(coors=dim_coor,
                                      …
                                      dropout=dropout) for _ in range(layers_num - 1)]
        self.conv_layers = torch.nn.ModuleList(self.conv_layers)
        self.fc1 = torch.nn.Linear(model_dim, out_dim)
    def forward(self, data):
        for i in range(self.layers_num):
            data.x = self.conv_layers[i](data.x, data.pos, data.edge_index)

            if self.use_cluster_pooling:
                weight = normalized_cut_2d(data.edge_index, data.pos)
                cluster = graclus(data.edge_index, weight, data.x.size(0))
                data = max_pool(cluster, data, transform=T.Cartesian(cat=False))

        data.x = global_mean_pool(data.x, data.batch)
        x = self.fc1(data.x)

        return F.log_softmax(x, dim=1)
```

Graph Conv Layers

FC Layer

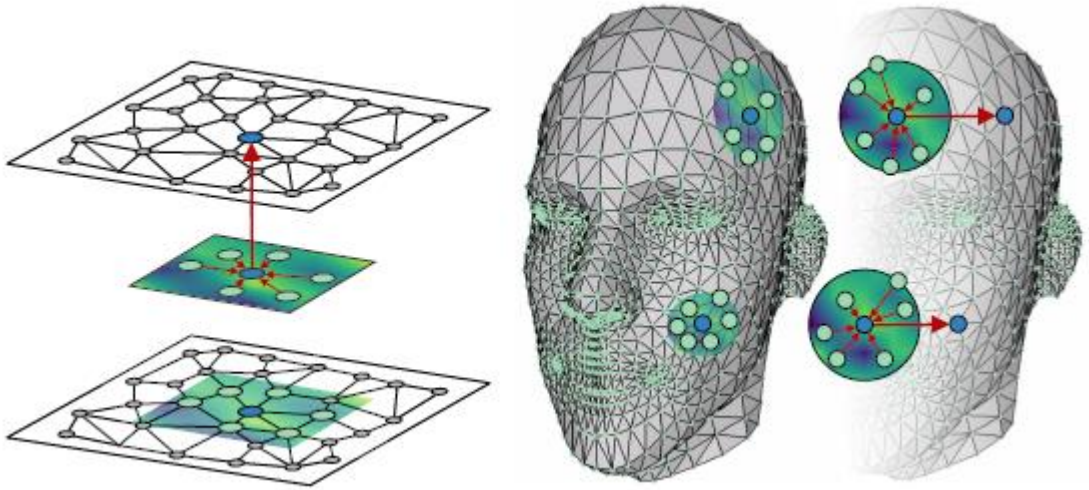
Code Structure: GraphConv

```python
class GraphConv(MessagePassing):
    def __init__(self, coors, out_channels_1, out_features, label_dim=1, dropout=0):

        super(GraphConv, self).__init__(aggr='add')
        self.lin_in = torch.nn.Linear(coors, label_dim * out_channels_1)
        self.lin_out = torch.nn.Linear(label_dim * out_channels_1, out_features)
        self.dropout = dropout

    def forward(self, x, pos, edge_index):

        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))  # num_edges = num_edges + num_nodes

        return self.propagate(edge_index=edge_index, x=x, pos=pos, aggr='add')  # [N, out_channels, label_dim]

    def message(self, pos_i, pos_j, x_j):

        tmp = pos_j - pos_i
        L = self.lin_in(tmp)  # [num_edges, out_channels]
        num_nodes, label_dim = list(x_j.size())
        label_dim_out_channels_1 = list(L.size())[1]

        X = F.relu(L)
        Y = x_j
        X = torch.t(X)
        X = F.dropout(X, p=self.dropout, training=self.training)
        result = torch.t(
            (X.view(label_dim, -1, num_nodes) * torch.t(Y).unsqueeze(1)).reshape(label_dim_out_channels_1, num_nodes))
        return result

    def update(self, aggr_out):

        aggr_out = self.lin_out(aggr_out)  # [num_nodes, label_dim, out_features]
        aggr_out = F.relu(aggr_out)
        aggr_out = F.dropout(aggr_out, p=self.dropout, training=self.training)

        return aggr_out
```

Convolution Mask M

NN

Appendix



pytorch_geometric

PyTorch Geometric (PyG) is a geometric d eep learning extension library for PyTorch.

- Various methods for deep learning on gr aphs and other irregular structures, also known as geometric deep learning, from a variety of published papers
- An easy-to-use mini-batch loader for ma ny small and single giant graphs, multi g pu-support, a large number of common b enchmark datasets (based on simple int erfaces to create your own), and helpful t ransforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds.

- **SplineConv** from Fey *et al.*: SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels (CVPR 2018)
- **GCNConv** from Kipf and Welling: Semi-Supervised Classification with Graph Convolutional Networks (ICLR 2017)
- **ChebConv** from Defferrard *et al.*: Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering (NIPS 2016)
- **NNConv** from Gilmer *et al.*: Neural Message Passing for Quantum Chemistry (ICML 2017)
- **CGConv** from Xie and Grossman: Crystal Graph Convolutional Neural Networks for an Accurate and Interpretable Prediction of Material Properties (Physical Review Letters 120, 2018)
- **ECConv** from Simonovsky and Komodakis: Edge-Conditioned Convolution on Graphs (CVPR 2017)
- **GATConv** from Veličković *et al.*: Graph Attention Networks (ICLR 2018)
- **SAGEConv** from Hamilton *et al.*: Inductive Representation Learning on Large Graphs (NIPS 2017)
- **GraphConv** from, *e.g.*, Morris *et al.*: Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks (AAAI 2019)
- **GatedGraphConv** from Li *et al.*: Gated Graph Sequence Neural Networks (ICLR 2016)
- **GINConv** from Xu *et al.*: How Powerful are Graph Neural Networks? (ICLR 2019)
- **GINEConv** from Wu *et al.*: Strategies for Pre-training Graph Neural Networks (ICLR 2020)
- **ARMAConv** from Bianchi *et al.*: Graph Neural Networks with Convolutional ARMA Filters (CoRR 2019)
- **SGConv** from Wu *et al.*: Simplifying Graph Convolutional Networks (CoRR 2019)
- **APPNP** from Klicpera *et al.*: Predict then Propagate: Graph Neural Networks meet Personalized PageRank (ICLR 2019)
- **AGNNConv** from Thekumparampil *et al.*: Attention-based Graph Neural Network for Semi-Supervised Learning (CoRR 2017)
- **TAGConv** from Du *et al.*: Topology Adaptive Graph Convolutional Networks (CoRR 2017)
- **RGCNConv** from Schlichtkrull *et al.*: Modeling Relational Data with Graph Convolutional Networks (ESWC 2018)
- **SignedConv** from Derr *et al.*: Signed Graph Convolutional Network (ICDM 2018)
- **DNAConv** from Fey: Just Jump: Dynamic Neighborhood Aggregation in Graph Neural Networks (ICLR-W 2019)
- **EdgeConv** from Wang *et al.*: Dynamic Graph CNN for Learning on Point Clouds (CoRR, 2018)
- **PointConv** (including **Iterative Farthest Point Sampling**, dynamic graph generation based on **nearest neighbor** or **maximum distance**, and **k-NN interpolation** for upsampling) from Qi *et al.*: PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation (CVPR 2017) and PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space (NIPS 2017)
- **XConv** from Li *et al.*: PointCNN: Convolution On X-Transformed Points (official implementation) (NeurIPS 2018)
- **PPFConv** from Deng *et al.*: PPFNet: Global Context Aware Local Features for Robust 3D Point Matching (CVPR 2018)
- **GMMConv** from Monti *et al.*: Geometric Deep Learning on Graphs and Manifolds using Mixture Model CNNs (CVPR 2017)
- **FeaStConv** from Verma *et al.*: FeaStNet: Feature-Steered Graph Convolutions for 3D Shape Analysis (CVPR 2018)
- **HypergraphConv** from Bai *et al.*: Hypergraph Convolution and Hypergraph Attention (CoRR 2019)
- **GravNetConv** from Qasim *et al.*: Learning Representations of Irregular Particle-detector Geometry with Distance-weighted Graph Networks (European Physics Journal C, 2019)
- A **MetaLayer** for building any kind of graph network similar to the TensorFlow Graph Nets library from Battaglia *et al.*: