Training Deeper GNNs

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실
NETWORK SCIENCE LAB

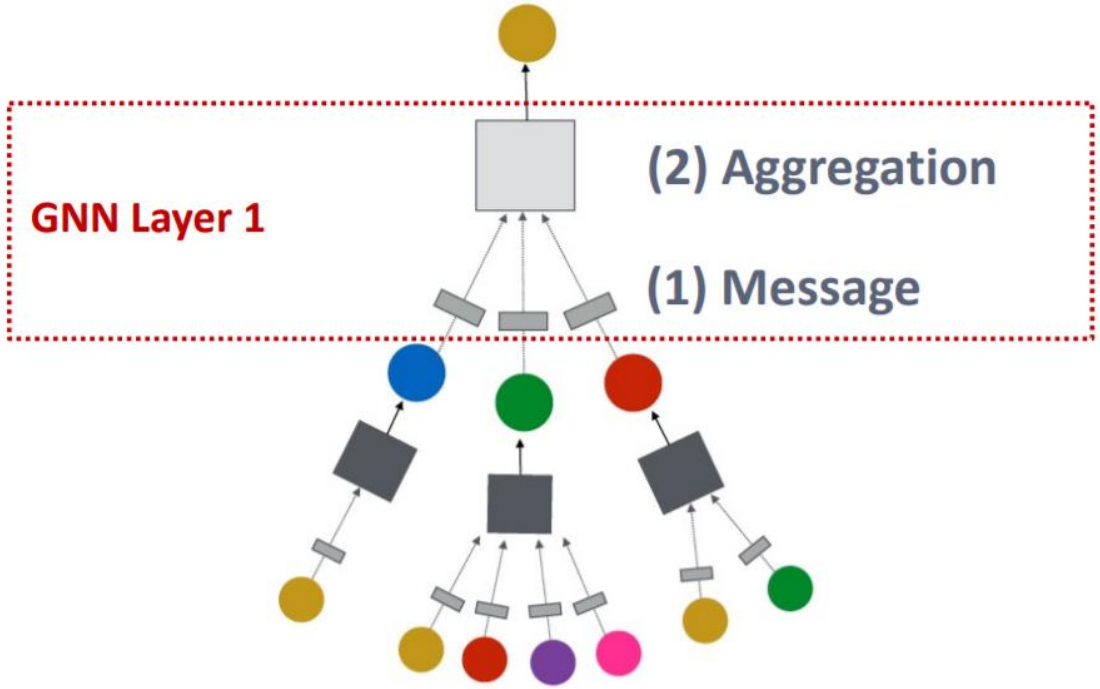
가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA

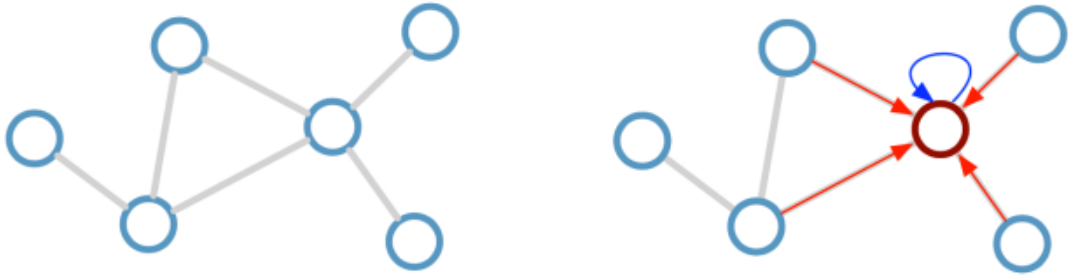
Contents

- Why do we need deeper GNNs?
- What impedes GNNs to go deeper?
- Solutions for alleviating over-smoothing and over-fitting
  - Dropping Nodes and Edges.
  - JK networks with Skip Connections.
  - DeeperGCN with residual connection.
  - GCNII with residual connection and identity mapping.

Recap: Graph Convolutional Networks (GCNs)

- GNN Layer = **Message** + **Aggregation**
  - Message COMPUTATION
    - How to make each neighborhood node as embedding?
  - Message AGGERGATION
    - How to combine those embeddings?





Update rule: $$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right)$$

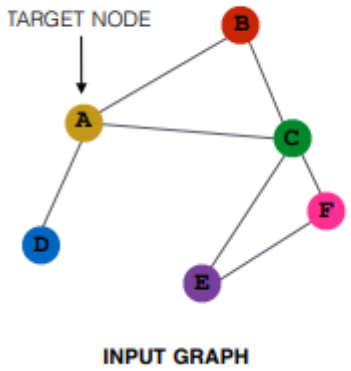
Kipf & Welling (ICLR 2017), related previous works by Duvenaud et al. (NIPS 2015) and Li et al. (ICLR 2016)

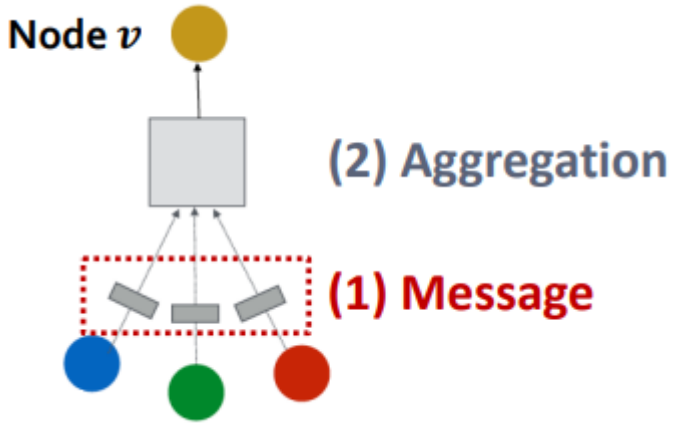
Recap: Message Computation

- Intuition: Each node will create a message, which will be sent to other nodes later

- Example: A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
  - Multiply node features with weight matrix $\mathbf{W}^{(l)}$

Message function: $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$





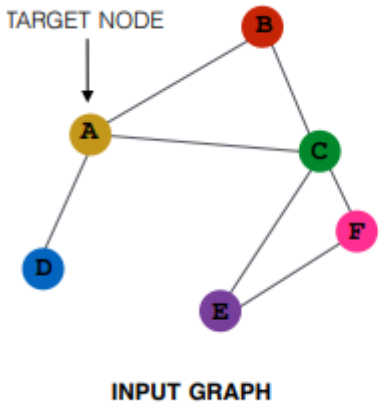
Recap: Message Aggregation

- **Intuition**: Each node will aggregate the messages from node v's neighbors

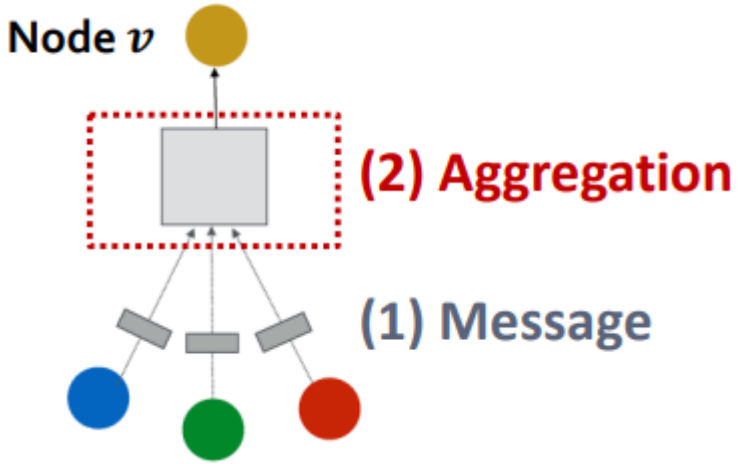
$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

- **Example**: Sum(·), Mean(·) or Max(·) aggregator

$$\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$$



INPUT GRAPH



Recap: Message Aggregation Issues

- **Issue**: Information from node v itself could get lost
  - Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$

- **Solution**: Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$
  - (1) Message: compute message from node v itself

$$\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)} \qquad \mathbf{m}_v^{(l)} = \mathbf{B}^{(l)}\mathbf{h}_v^{(l-1)}$$

  - (2) Aggregation: After aggregating from neighbors, we can aggregate the message from node v itself
    - Via concatenation or summation

Then aggregate from node itself

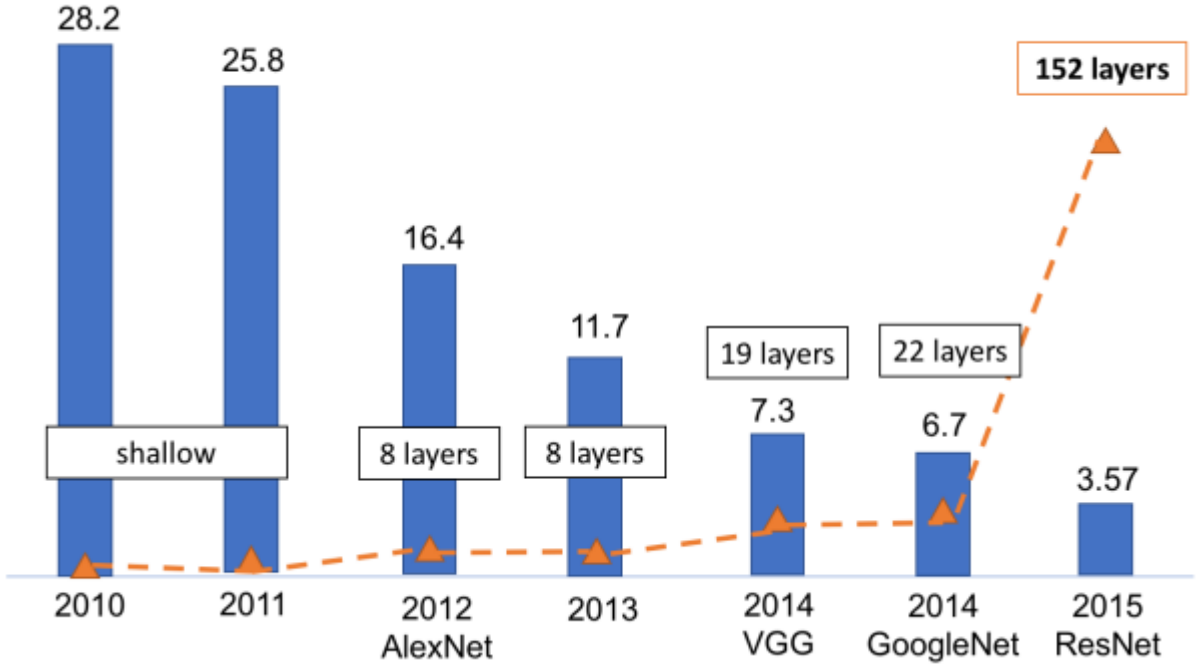
$$\mathbf{h}_v^{(l)} = \text{CONCAT}\left(\text{AGG}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right), \mathbf{m}_v^{(l)}\right)$$

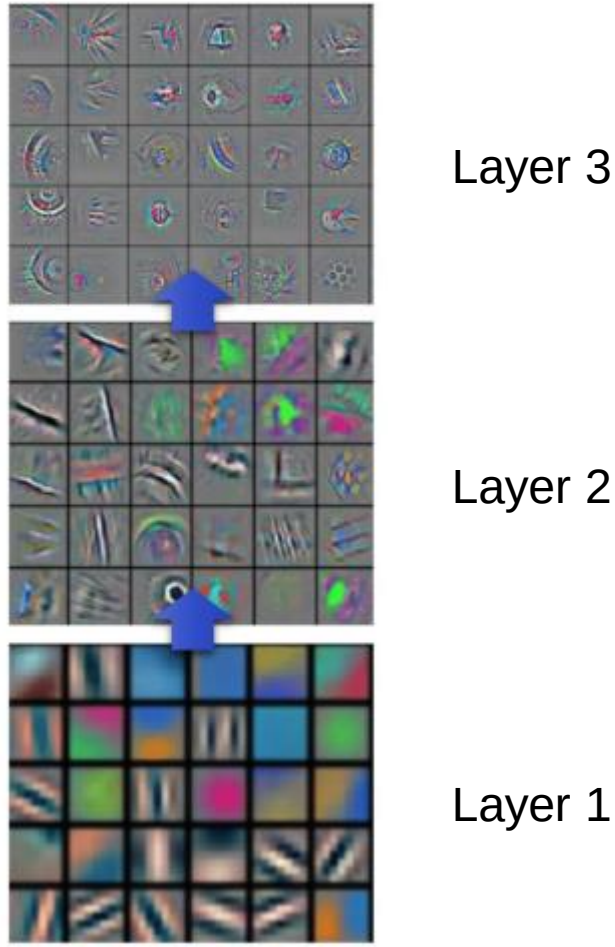
First aggregate from neighbors

https://web.stanford.edu/class/cs224w/slides/04-GNN2.pdf

The Power of Deeper DNNs

- Unprecedented success of deep DNNs in computer vision
- Deeper DNNs enable larger receptive fields

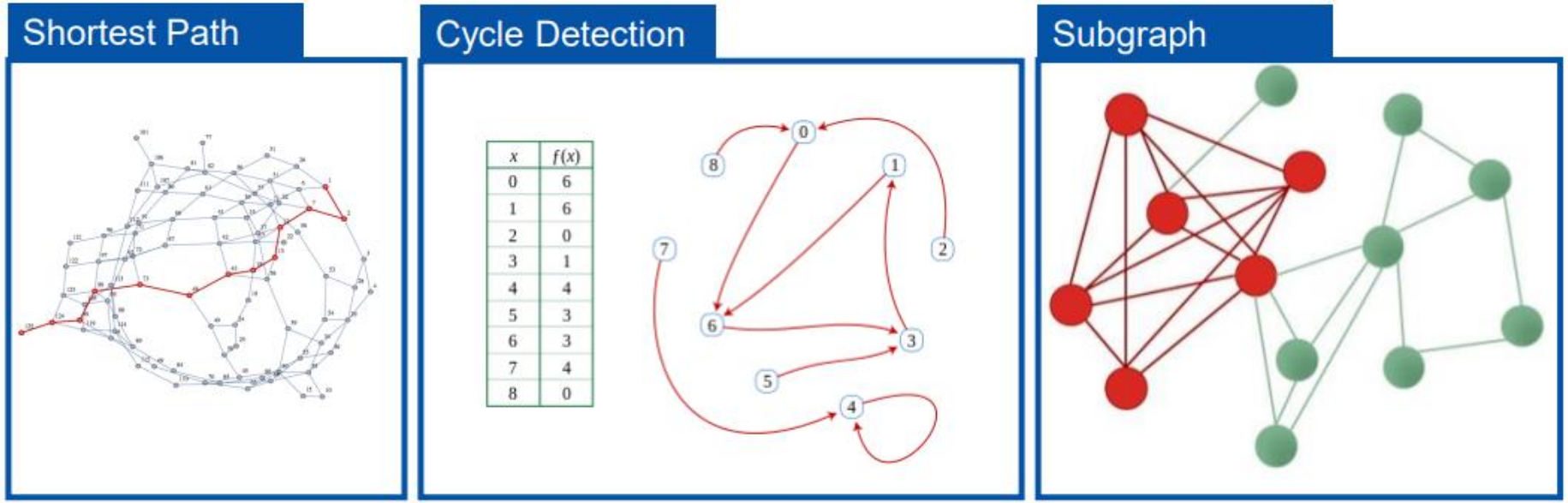




Yu Rong, Wenbing Huang, Tingyang Xu, Hong Cheng, Junzhou Huang 2020

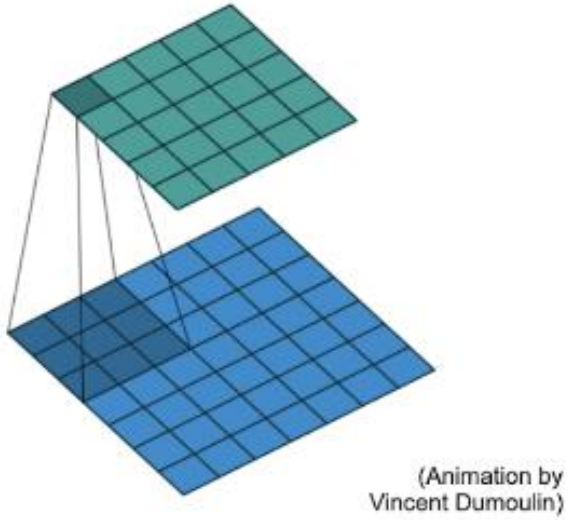
The Power of Deeper DNNs

- Do GNNs need deeper structures to enable larger receptive fields, too?
- What limits the expressive power of GNNs?
  - The depth d.
  - The width w.
- GNNs significantly lose their power when capacity, dw, is restricted

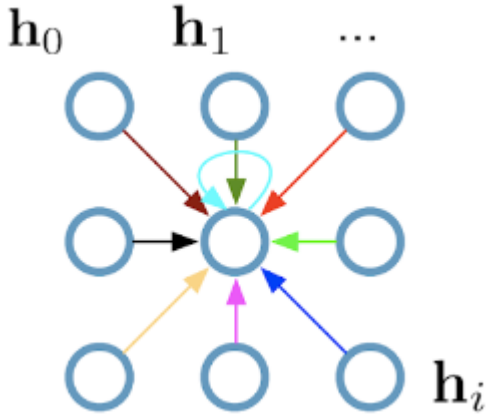


Loukas, Andreas. "What graph neural networks cannot learn: depth vs width." International Conference on Learning Representations. 2020

- Single CNN layer with 3x3 filter:



(Animation by Vincent Dumoulin)



Update for a single pixel:

- Transform messages individually $\mathbf{W}_i\mathbf{h}_i$
- Add everything up $\sum_i \mathbf{W}_i\mathbf{h}_i$

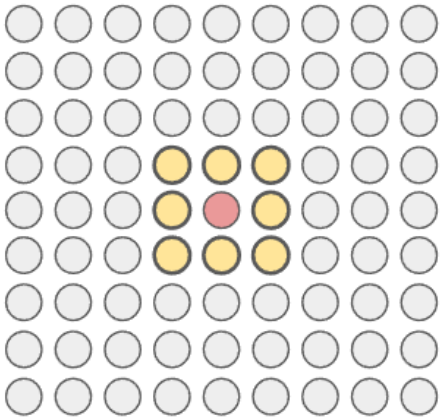
$\mathbf{h}_i \in \mathbb{R}^F$ are (hidden layer) activations of a pixel/node

Full update:

$$\mathbf{h}_4^{(l+1)} = \sigma\left(\mathbf{W}_0^{(l)}\mathbf{h}_0^{(l)} + \mathbf{W}_1^{(l)}\mathbf{h}_1^{(l)} + \cdots + \mathbf{W}_8^{(l)}\mathbf{h}_8^{(l)}\right)$$

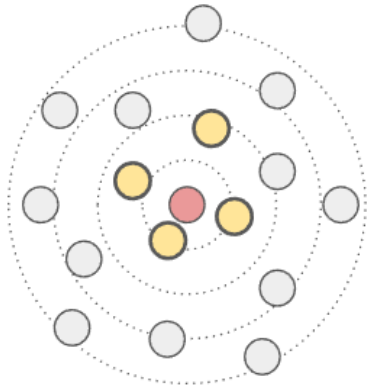
Recap: CNN vs. GCN

- The key differences between CNNs and GCNs is how the convolution is computed.
- For CNNs, we simply do a convolution with our neighbors on the grid with a given kernel size.
- On graphs, we need to define our neighbors first, and then do the convolution in the neighborhood. This is done by finding the k nearest neighbors (with respect to some distance metric) .
- **So, where are the deep GCNs?**



$$\sigma\left(\sum_i w_i x_i + b\right)$$

Convolutional Neural Network (CNN)



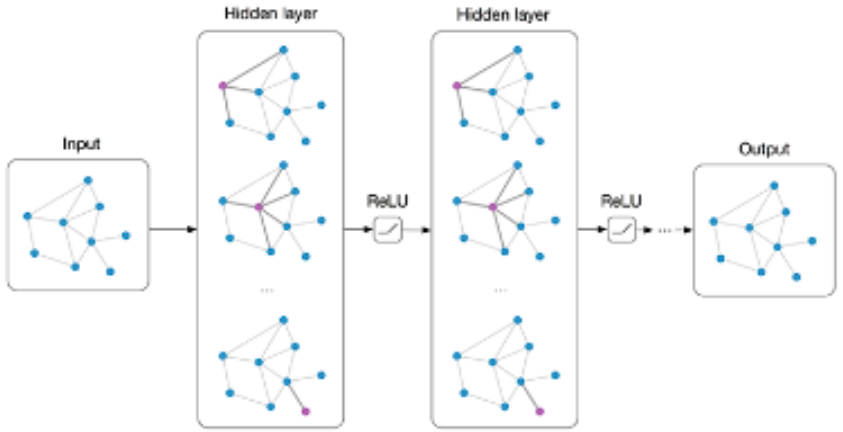
$$\sigma\left(\sum_{u \in \mathcal{N}(v)} w x_u + w' x_v + b\right)$$

Graph Convolutional Network (GCN)

Why current GCNs cannot go deeper?

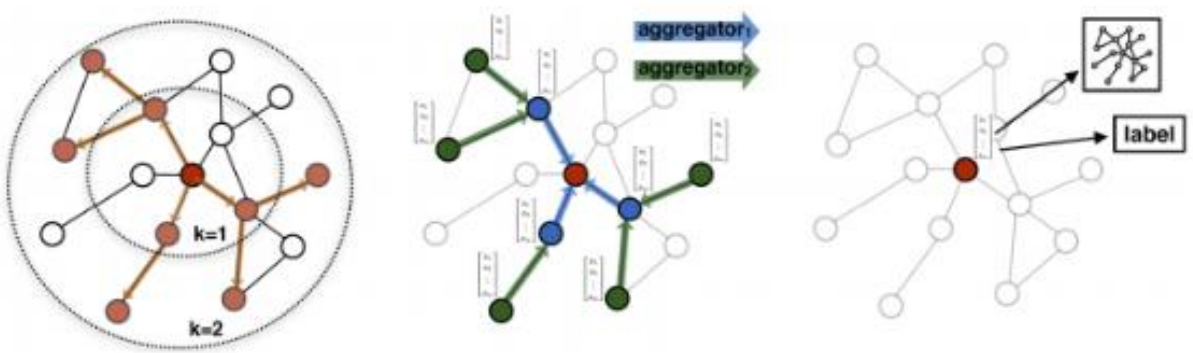
- Most SOTA GCN models are normally no deeper than 3 or 4 layers.

GCN



Kipf, T.N. and Welling, M., 2016. Semi-Supervised Classification with Graph Convolutional Networks.

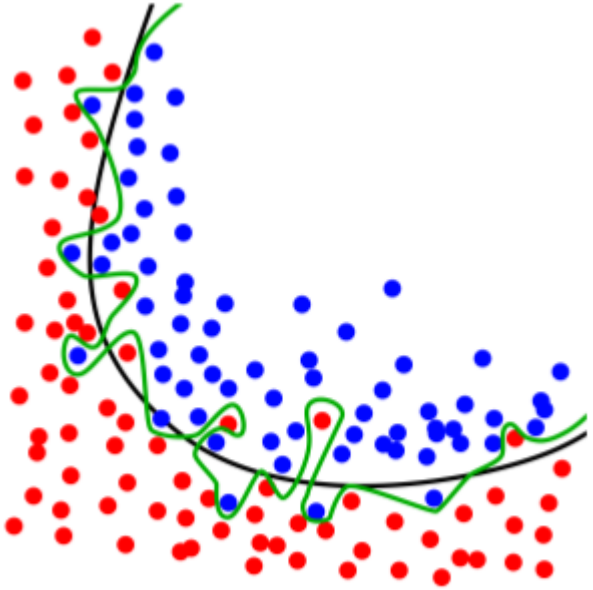
GraphSage



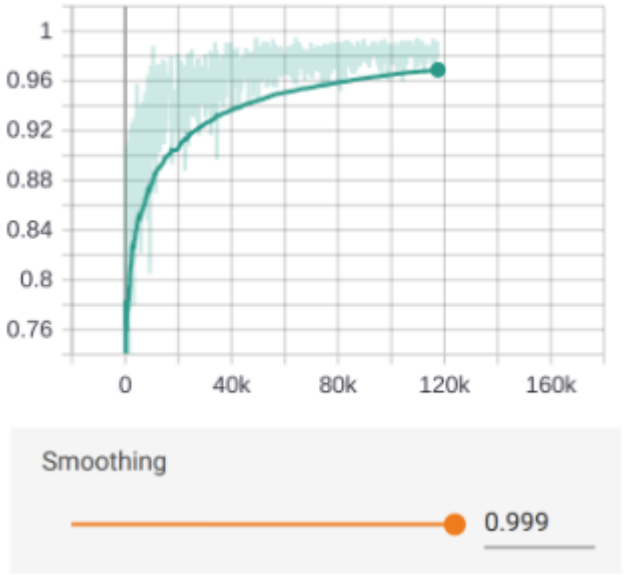
Hamilton, W.L., Ying, R. and Leskovec, J., 2017. Inductive Representation Learning on Large Graphs.

Why GCNs are limited to shallow structures?

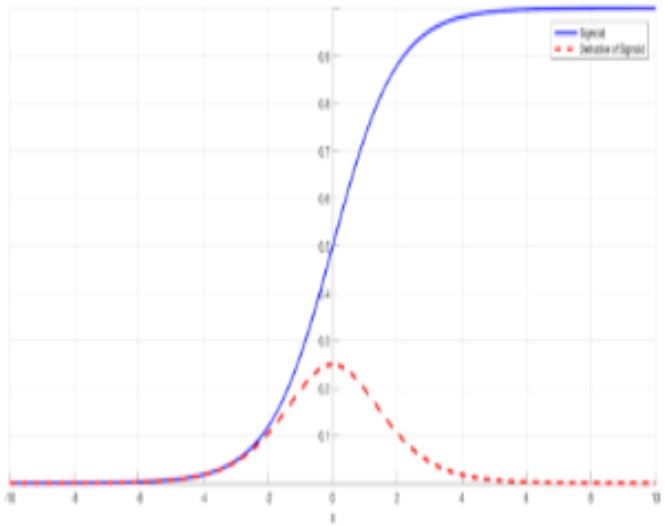
- It turns out that this is not trivial.
- The main difficulties here are Over-fitting, Over-smoothing, and Vanishing gradient.



Over-fitting



Over-smoothing

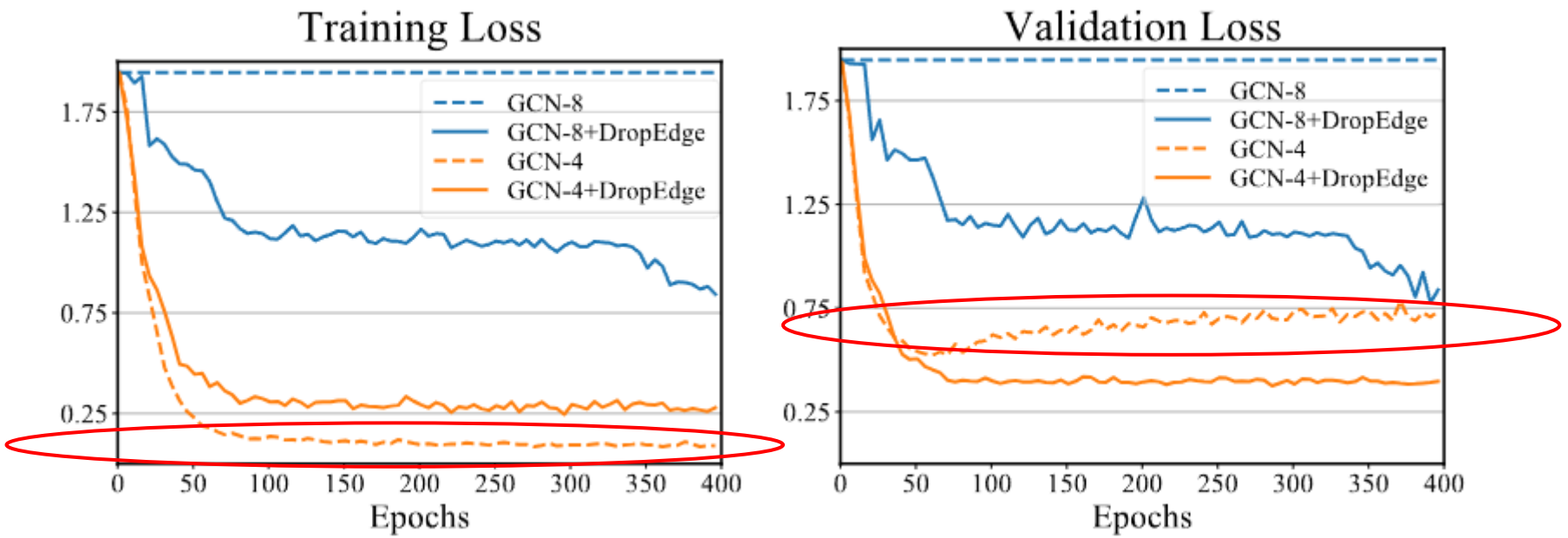


Vanishing Gradient

Figures from https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

Why current GCNs cannot go deeper?

- **Overfitting**: GNNs suffer from Overfitting
  - Too many parameters are established but only few of data points are provided

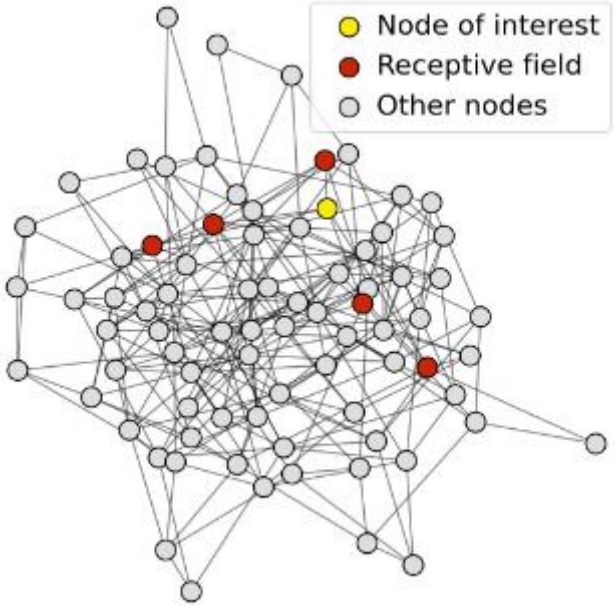


Rong, Yu, et al. "Dropedge: Towards deep graph convolutional networks on node classification." *arXiv preprint arXiv:1907.10903* (2019).

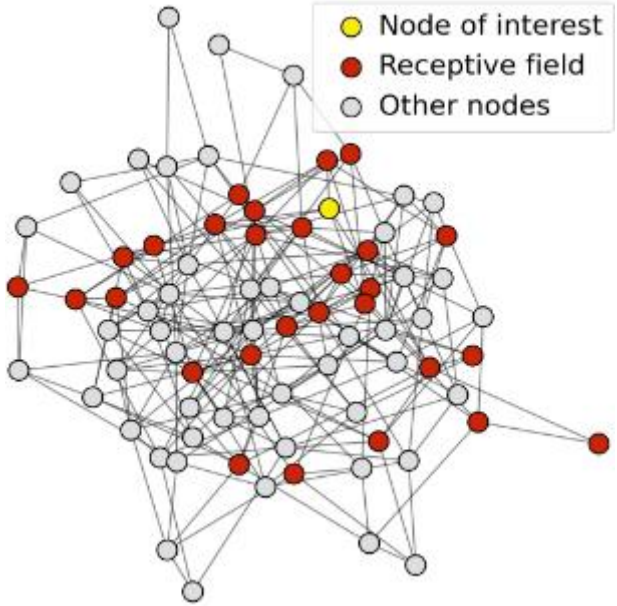
Why current GCNs cannot go deeper?

- **Over-smoothing problem**: all the node embeddings converge to the same value
  - This is bad because we want to use node embeddings to differentiate nodes.
- Why does the over-smoothing problem happen?
  - **Receptive field**: the set of nodes that determine the embedding of a node of interest.
    - In a K-layer GNN, each node has a receptive field of K-hop neighborhood

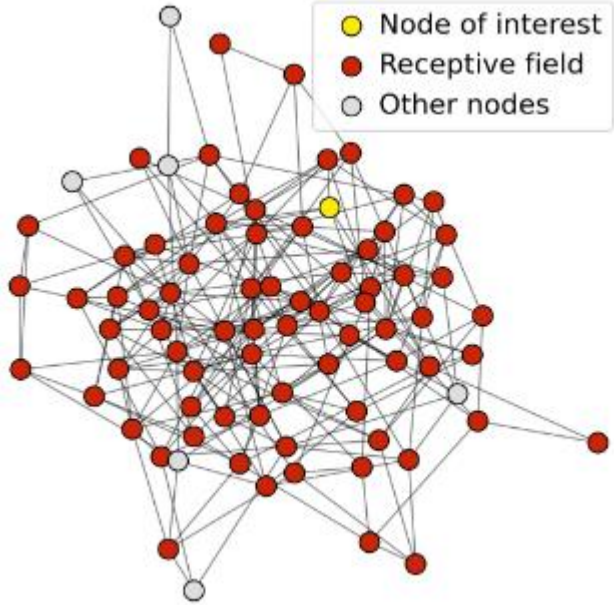
Receptive field for 1-layer GNN



Receptive field for 2-layer GNN



Receptive field for 3-layer GNN

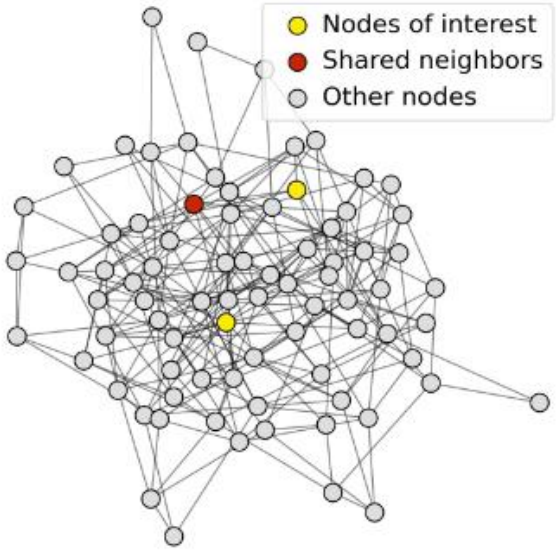


Jure Leskovec, Stanford CS224W: Machine Learning with Graph

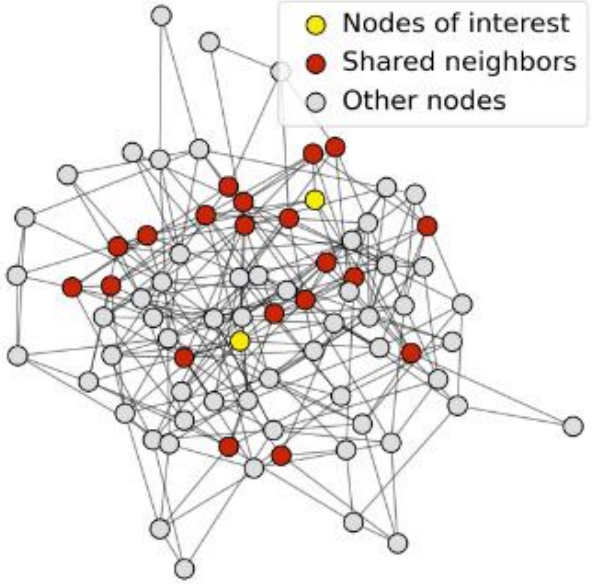
Why current GCNs cannot go deeper?

- **Over-smoothing**:
  - Receptive field overlap for two nodes: shared neighbors quickly grows when increase the number of hops (number of GNN layers).
    - If two nodes have highly-overlapped receptive fields, then their embeddings are highly similar.

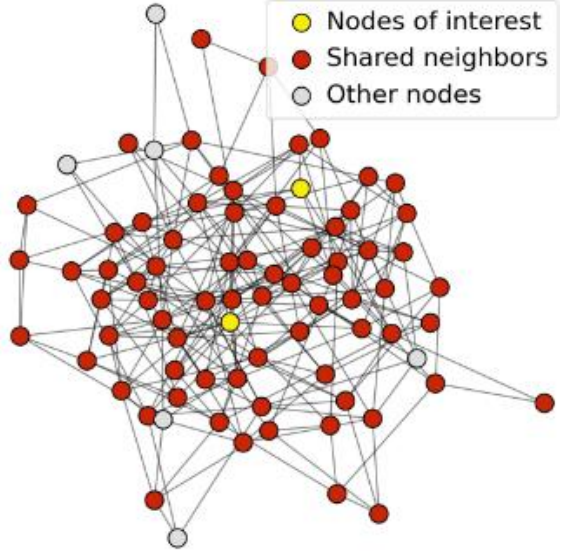
1-hop neighbor overlap
Only 1 node



2-hop neighbor overlap
About 20 nodes



3-hop neighbor overlap
Almost all the nodes!



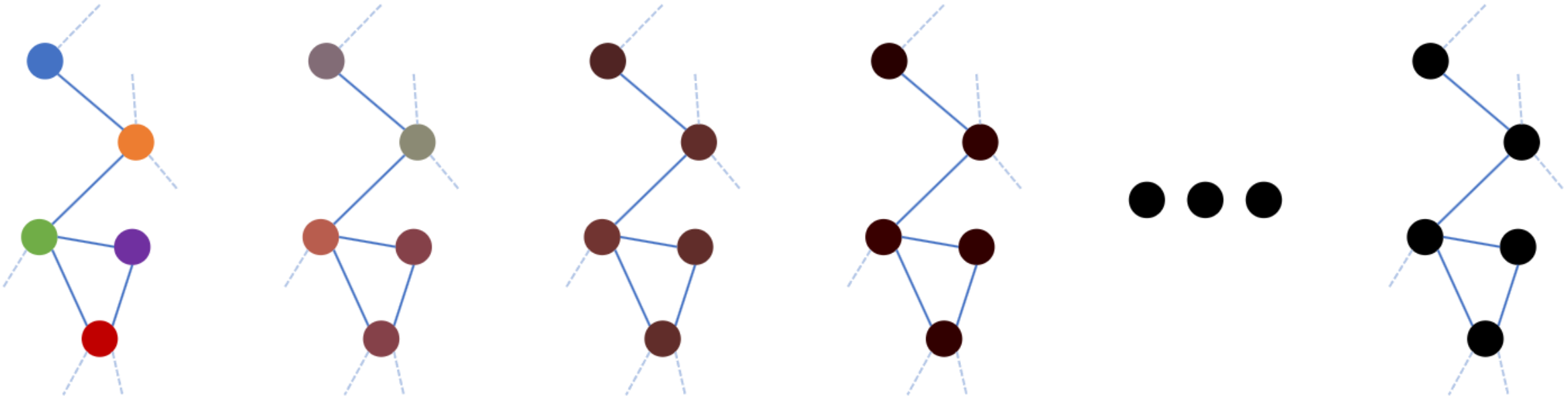
Jure Leskovec, Stanford CS224W: Machine Learning with Graph

Why current GCNs cannot go deeper?

- **Vanishing Gradient**:
  - l-layers gradients

$$\frac{dH_{l+1}}{dH_l} \cdot \frac{dH_l}{dH_{l-1}} \cdot \cdots \cdot \frac{dH_0}{dW_0} \leq (s_l \lambda_{m+1}) \cdot (s_{l-1} \lambda_{m+1}) \cdot \cdots \cdot \frac{dH_0}{dW_0}$$

  - The gradients vanish as the model go deeper because $s_{1..l}\lambda_{m+1} < 1$



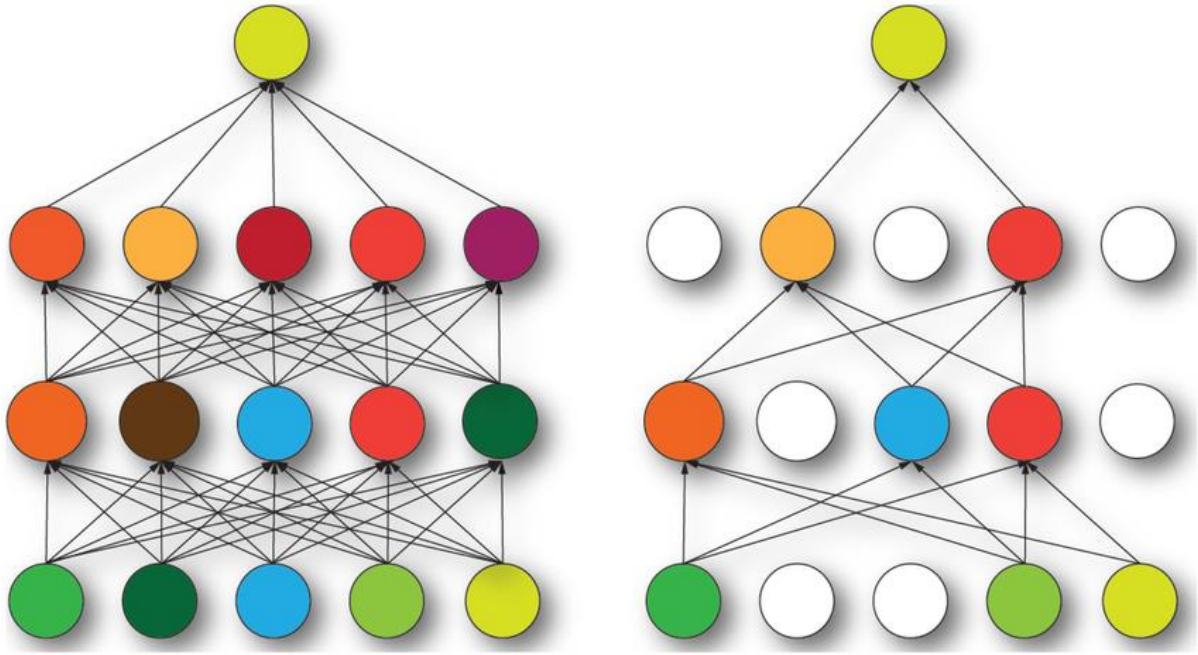
RGB as Features | Layer 1 | Layer 100 | Layer 200 | Layer 500
RGB = [0,0,0]

Solutions for training Deeper GNNs

- Randomly Dropping
  - Dropout().
  - DropEdges and DropNodes.
- PairNorm.
- JK-Net networks with Skip Connection.
- GCNII with Initial residual and Identity mapping.
- DeeperGCN with residual connection.

Randomly Dropping: Dropout()

- **Similar to standard neural networks:** Dropout is used as a regularization technique — it prevents overfitting by ensuring that no units are codependent (more on this later).
- During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.



N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting

Randomly Dropping: Dropout() Sample code

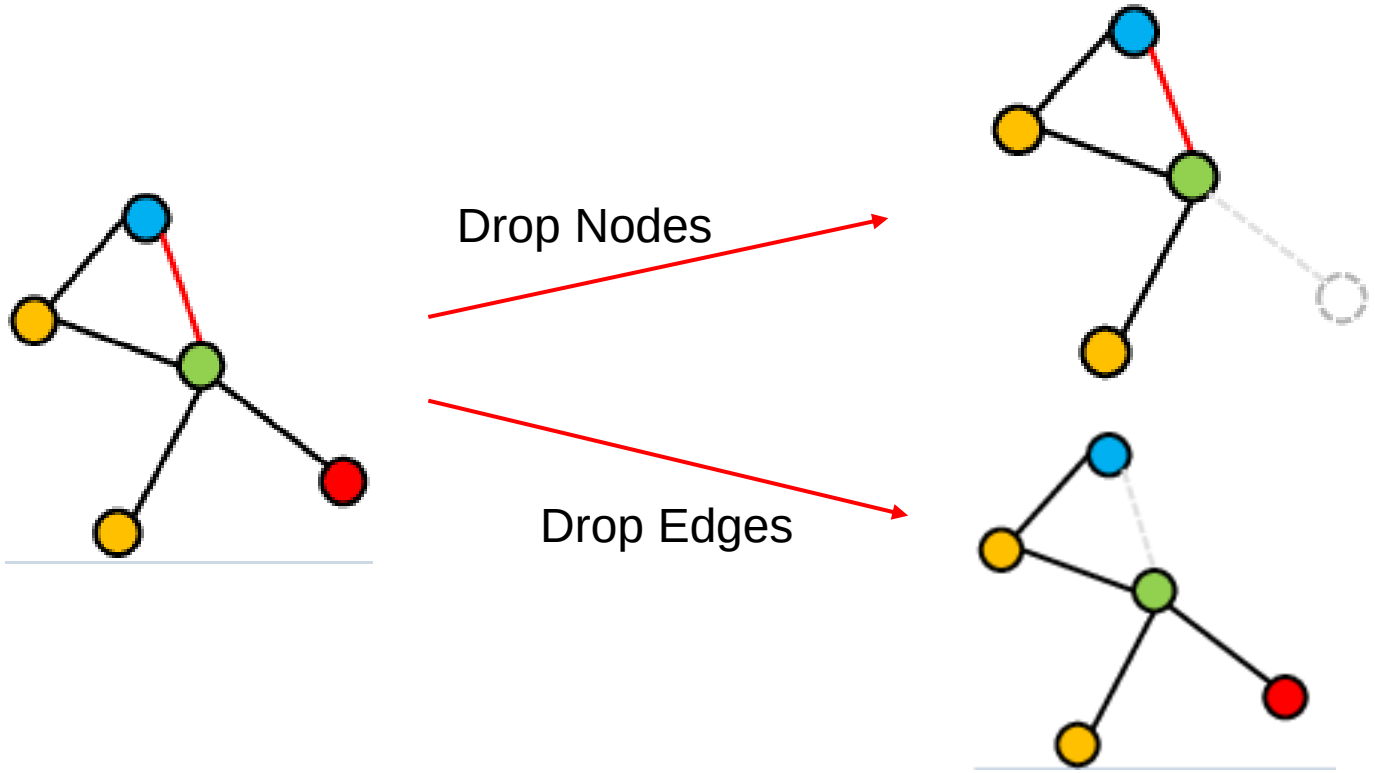
➢ Dropout: From torch.nn.Dropout

- ➢ During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.
- ➢ Each channel will be zeroed out independently on every forward call.

```
36 class GCN(torch.nn.Module):
37     def __init__(self, num_features, num_classes, dim=16, drop=0.5):
38         super(GCN, self).__init__()
39         self.conv1 = GCNConv(num_features, dim)
40         self.conv2 = GCNConv(dim, num_classes)
41         self.drop = torch.nn.Dropout(p=drop)
42         self.h = None
43     def forward(self, x, edge_index,  g, Kindices):
44         x = F.relu(self.conv1(x, edge_index))
45         x = self.drop(x)
46         x = self.conv2(x, edge_index)
47         self.h = x
48         return F.log_softmax(x, dim=1)
49
```

Dropping: Edges and Nodes

- Randomly remove a certain number of edges or nodes from the input graph at each training epoch.
- Alternatively, they could be regarded as data augmentation methods, helping relieve both the over-fitting and over-smoothing issues in training very deep GNNs.

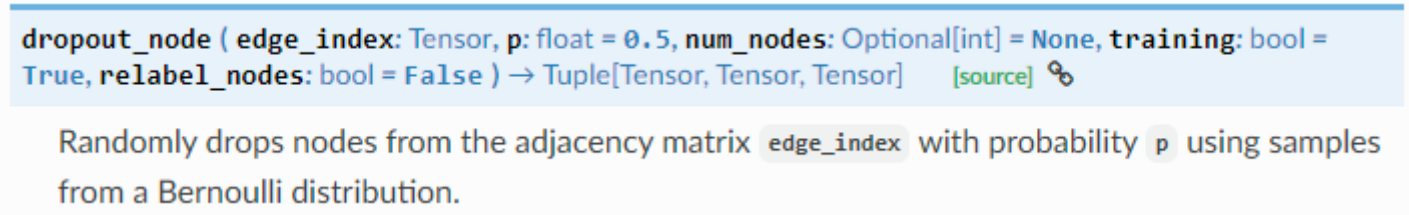


Rong, Yu, et al. "Dropedge: Towards deep graph convolutional networks on node classification." *arXiv preprint arXiv:1907.10903* (2019).

Dropping Edges/Nodes: Sample code

➢ Dropout: From torch.nn.Dropout

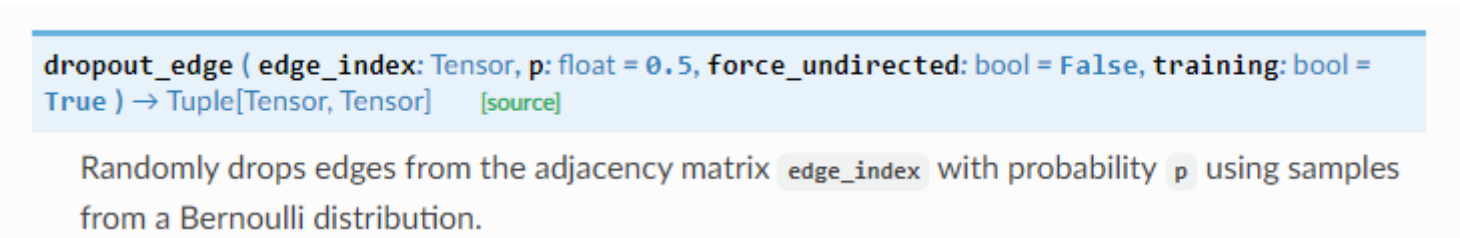
- ➢ Randomly drops nodes from the adjacency matrix edge_index with probability p using samples from a Bernoulli distribution.



dropout_node (**edge_index**: Tensor, **p**: float = `0.5`, **num_nodes**: Optional[int] = `None`, **training**: bool = `True`, **relabel_nodes**: bool = `False`) → Tuple[Tensor, Tensor, Tensor] [source]

Randomly drops nodes from the adjacency matrix `edge_index` with probability `p` using samples from a Bernoulli distribution.

- ➢ Randomly drops edges from the adjacency matrix edge_index with probability p using samples from a Bernoulli distribution.



dropout_edge (**edge_index**: Tensor, **p**: float = `0.5`, **force_undirected**: bool = `False`, **training**: bool = `True`) → Tuple[Tensor, Tensor] [source]

Randomly drops edges from the adjacency matrix `edge_index` with probability `p` using samples from a Bernoulli distribution.

Random Dropping: DropEdges Sample code

- Randomly drops edges/nodes for each epochs
- Let's do some sample codes

```python
def train_dropEdges(model, data):
    """Train a GNN model and return the trained model."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = model.optimizer
    epochs = 15

    model.train()
    for epoch in range(epochs+1):
        # Training
        optimizer.zero_grad()
        edge_index1, _ = dropout_edge(data.edge_index, p=0.5)
        _, out = model(data.x, edge_index1 )
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.train_mask])
        loss.backward()
        optimizer.step()

        # Validation
        val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
        val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.val_mask])

        # Print metrics every 10 epochs
        if(epoch % 1 == 0):
            print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc: '
                  f'{acc*100:>6.2f}% | Val Loss: {val_loss:.2f} | '
                  f'Val Acc: {val_acc*100:.2f}%')

    return model
```

```python
def train_dropNodes(model, data):
    """Train a GNN model and return the trained model."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = model.optimizer
    epochs = 15

    model.train()
    for epoch in range(epochs+1):
        # Training
        optimizer.zero_grad()
        edge_index1, _, _ = dropout_node(data.edge_index)
        _, out = model(data.x, edge_index1 )
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.train_mask])
        loss.backward()
        optimizer.step()

        # Validation
        val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
        val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.val_mask])

        # Print metrics every 10 epochs
        if(epoch % 1 == 0):
            print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc: '
                  f'{acc*100:>6.2f}% | Val Loss: {val_loss:.2f} | '
                  f'Val Acc: {val_acc*100:.2f}%')

    return model
```

- **Idea:**
  1) Combining a proper normalizer and a residual node update formulation addresses oversmoothing (in many cases).
  2) Maintain constant total pairwise feature distances across layers to prevent distant node embeddings from becoming too similar.

- Types of oversmoothing in GCN:
  - **Node-wise oversmoothing**: Features of connected nodes become too similar.
  - **Feature-wise oversmoothing**: All node features become indistinguishable across the graph.

Zhao, Lingxiao, and Leman Akoglu. "Pairnorm: Tackling oversmoothing in gnns." *ICLR 2020*

PairNorm: Tackling Oversmoothing in GNNs

- **Normalization:**

$$\tilde{\mathbf{x}}_i^c = \tilde{\mathbf{x}}_i - \frac{1}{n}\sum_{i=1}^{n}\tilde{\mathbf{x}}_i \qquad \text{(Center)}$$

$$\dot{\mathbf{x}}_i = s \cdot \frac{\tilde{\mathbf{x}}_i^c}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}\|\tilde{\mathbf{x}}_i^c\|_2^2}} = s\sqrt{n} \cdot \frac{\tilde{\mathbf{x}}_i^c}{\sqrt{\|\tilde{\mathbf{X}}^c\|_F^2}} \qquad \text{(Scale)}$$

- **Residual** update:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right) \longrightarrow \mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right) + \mathbf{h}_i^{(l)}$$

- **Advantage:**
  - Fast and easy to implement.
  - Compatible with existing GNN architectures without requiring additional parameters.
  - General enough to be applied to various GNN models.

Zhao, Lingxiao, and Leman Akoglu. "Pairnorm: Tackling oversmoothing in gnns." *ICLR 2020*

JK-Net networks

- **Idea:**
  1) Makes each layer increase the size of the influence distribution by aggregating neighborhoods from the previous layers
  2) Combines some of the previous layers' representations independently for each node

- At the last layer, for each node, carefully select from all of those intermediate representations (which "jump" to the last layer).
- If this is done independently for each node, then the model can adapt the effective neighborhood size for each node as needed, resulting in exactly the desired adaptivity.

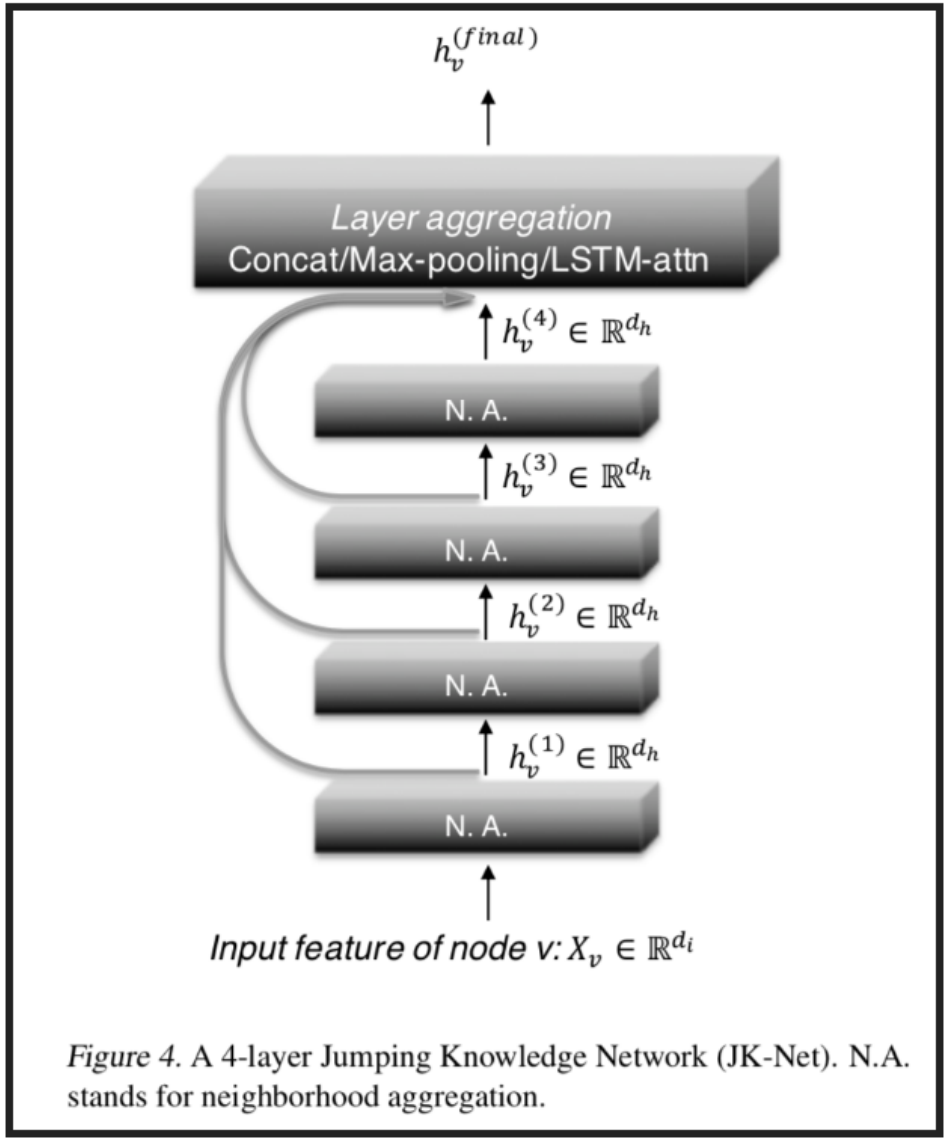


Figure 4. A 4-layer Jumping Knowledge Network (JK-Net). N.A. stands for neighborhood aggregation.

JK networks: Aggregation Mechanisms with Skip Connections

- **CONCATENATION**:

$$\text{CONCATENATION}\left[h_v^{(1)}, \ldots, h_v^{(k)}\right]$$

- **MAX-POOLING**:
  - Select the most informative layer for each feature coordinate.
- **LSTM-ATTENTION**:
  - Input $h_v^{(1)}, \ldots, h_v^{(k)}$ into a bi-directional LSTM to generate forward and backward $f_v^{(l)}$ and $b_v^{(l)}$ for each layer.

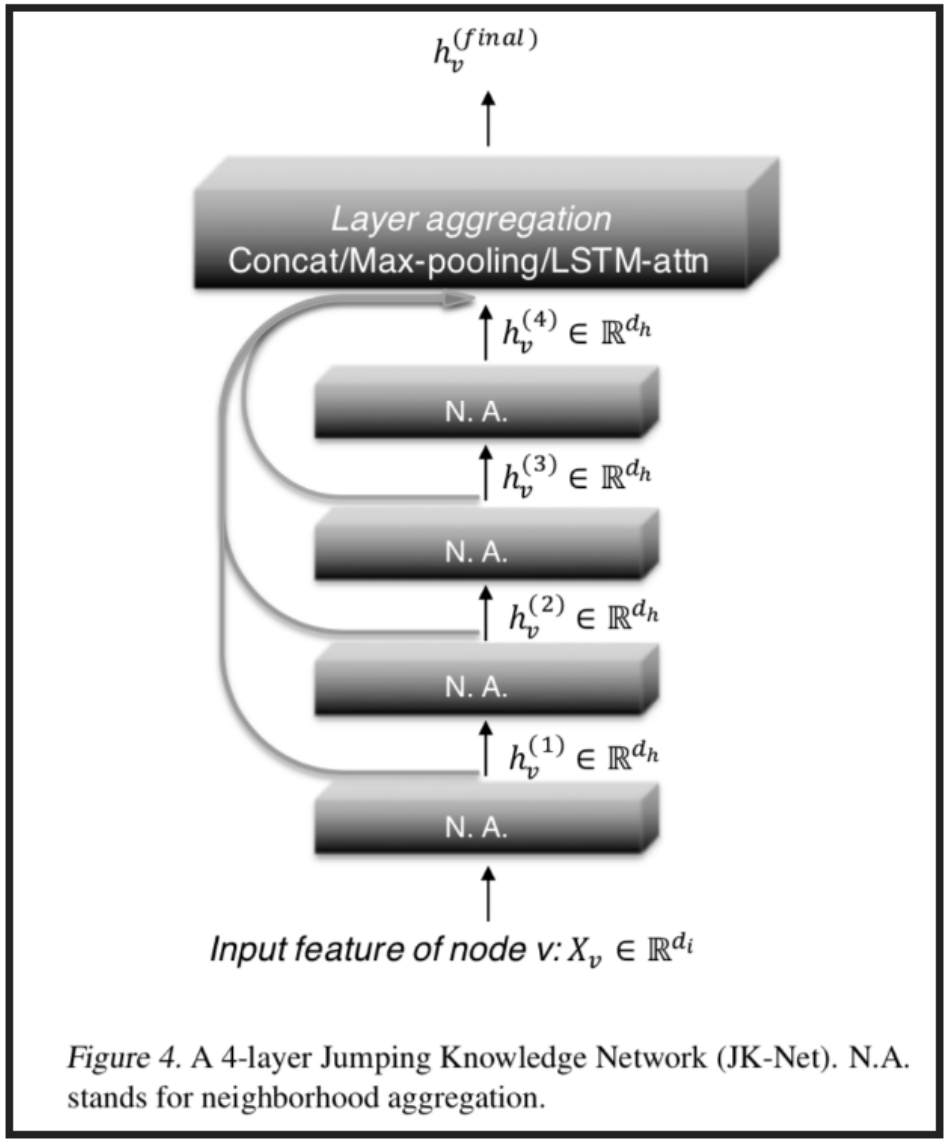
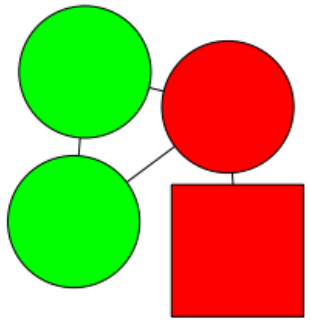


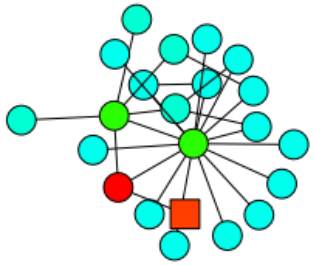
Figure 4. A 4-layer Jumping Knowledge Network (JK-Net). N.A. stands for neighborhood aggregation.

JK-Net networks: Example

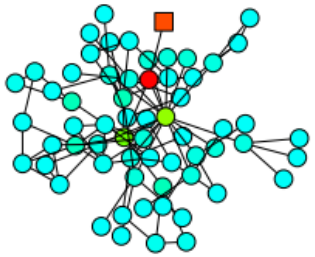
- A 6-layer JK-Net learns to adapt to different subgraph structures



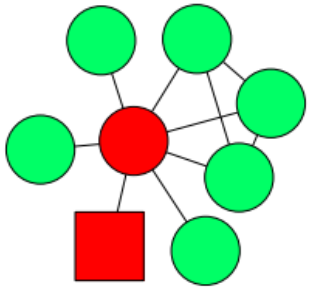
(g) 2-layer



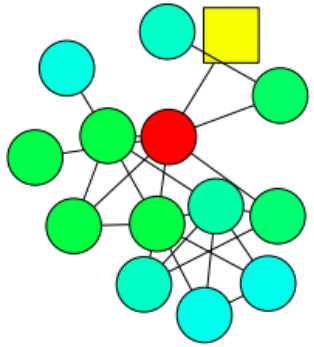
(h) 3-layer



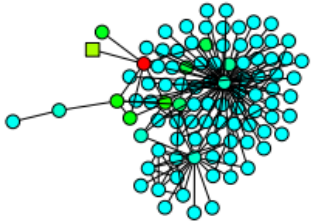
(i) 4-layer



(j) 2-layer

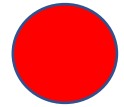


(k) 3-layer



(l) 4-layer

Target node



1-hop neighbours

2-hop neighbours

JK-Net networks: Sample code

JK-Net in PyTorch Geometric:

```
from torch_geometric.nn import JumpingKnowledge
```

models.JumpingKnowledge

class **JumpingKnowledge** (**mode**: str, **channels**: Optional[int] = None, **num_layers**: Optional[int] = None)
[source]

Bases: `Module`

The Jumping Knowledge layer aggregation module from the "Representation Learning on Graphs with Jumping Knowledge Networks" paper.

Jumping knowledge is performed 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"`).

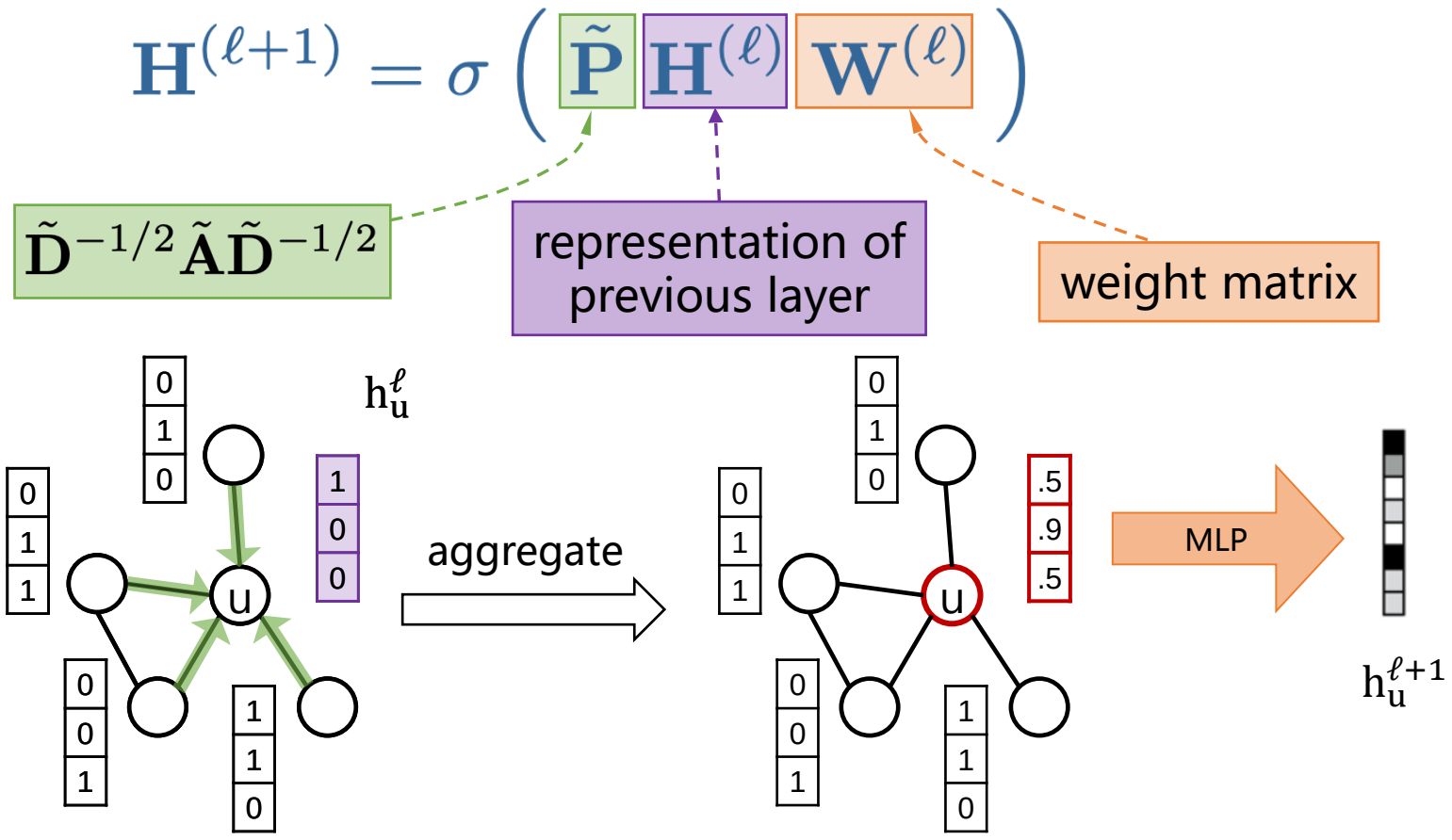
https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.models.JumpingKnowledge.html

JK-Net networks: Sample code


```python
class JKNet(torch.nn.Module):
    def __init__(self, dataset, mode='max', num_layers=6, hidden=16):
        super(JKNet, self).__init__()
        self.num_layers = num_layers
        self.mode = mode

        self.conv0 = GCNConv(dataset.num_node_features, hidden)
        self.dropout0 = torch.nn.Dropout(p=0.5)

        for i in range(1, self.num_layers):
            setattr(self, 'conv{}'.format(i), GCNConv(hidden, hidden))
            setattr(self, 'dropout{}'.format(i), torch.nn.Dropout(p=0.5))

        self.jk = JumpingKnowledge(mode=mode)
        if mode == 'max':
            self.fc = torch.nn.Linear(hidden, dataset.num_classes)
        elif mode == 'cat':
            self.fc = torch.nn.Linear(num_layers * hidden, dataset.num_classes)
        self.optimizer = torch.optim.Adam(self.parameters(),
                                          lr=0.01,
                                          weight_decay=5e-4)
    def forward(self, x, edge_index):

        layer_out = []
        for i in range(self.num_layers):
            conv = getattr(self, 'conv{}'.format(i))
            dropout = getattr(self, 'dropout{}'.format(i))
            x = dropout(F.relu(conv(x, edge_index)))
            layer_out.append(x)

        h = self.jk(layer_out)
        h = self.fc(h)
        return h, F.log_softmax(h, dim=1)
```

Graph Convolutional Networks

- Graph Convolutional Layer (Kipf et al., 2017)



GCNII: GCN via Initial residual and Identity mapping

- **Idea**: introduces two key modifications to solve oversmoothing problem
  - Initial Residual Connection: Adds a skip connection from the input layer to each subsequent layer, ensuring the original features of nodes are retained through multiple layers.
  - Identity Mapping: Modifies the weight matrix by adding an identity matrix, which helps retain the expressive power of deep networks by reducing overfitting and feature degradation.

Chen, Ming, et al. "Simple and deep graph convolutional networks." *International conference on machine learning*. PMLR, 2020.

GCNII: GCN via Initial residual and Identity mapping

- Vanilla GCN

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\tilde{\mathbf{P}}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}\right)$$

- GCN + **Initial residual**

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell\mathbf{H}^{(0)}\right)\mathbf{W}^{(\ell)}\right)$$

- **GCNII**: GCN + Initial residual **+ Identity mapping**

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell\mathbf{H}^{(0)}\right)\left((1-\beta_\ell)\mathbf{I}_n + \beta_\ell\mathbf{W}^{(\ell)}\right)\right)$$

Chen, Ming, et al. "Simple and deep graph convolutional networks." *International conference on machine learning*. PMLR, 2020.

GCNII: Sample code

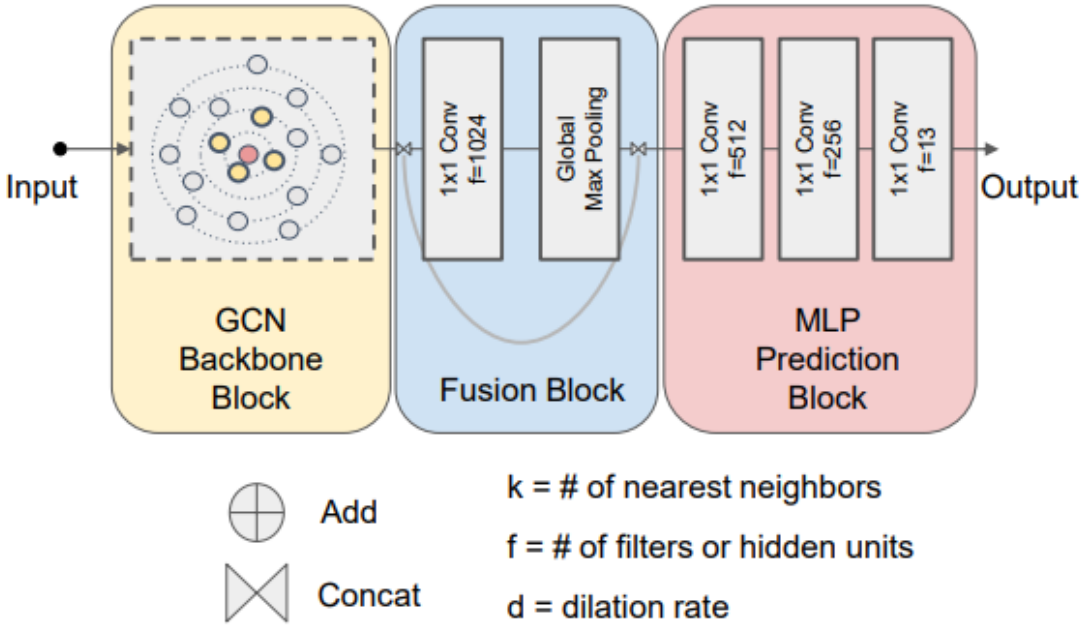
from torch_geometric.nn import GCN2Conv

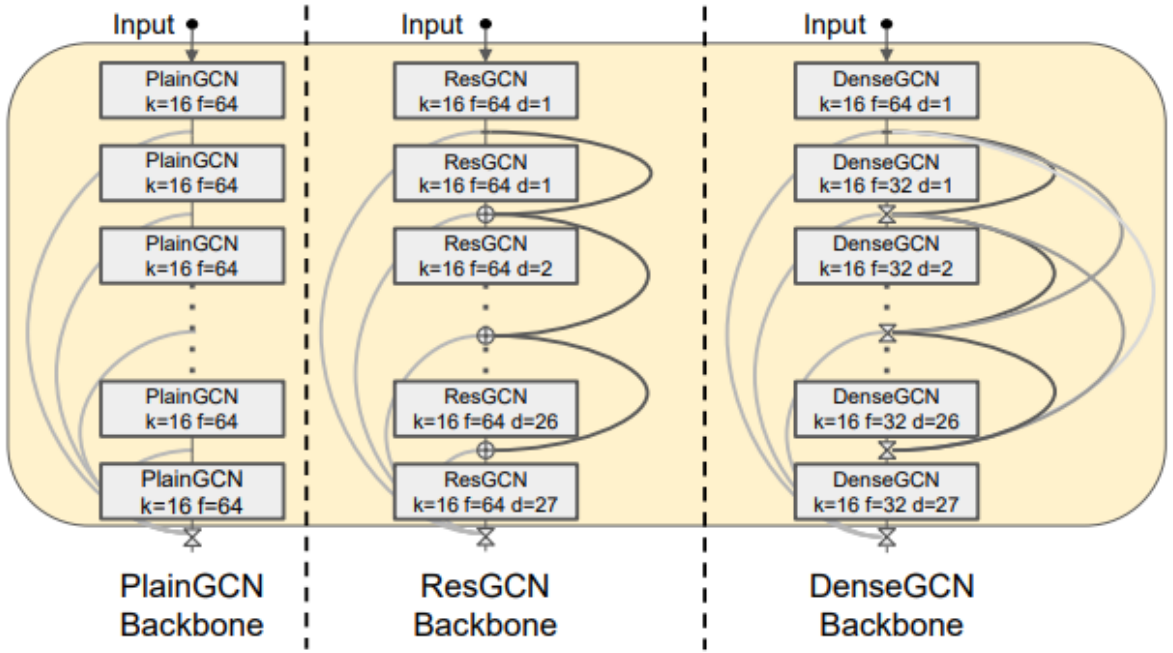
```
class GCNII(torch.nn.Module):
  def __init__(self, num_features, num_classes, hidden_channels = 16, num_layers =2, shared_weights=True, dropout=0.0):
    super().__init__()
    self.lins = torch.nn.ModuleList()
    self.proj = Linear(1, 64)
    self.lins1 = Linear(num_features, hidden_channels)
    self.lins2  = Linear(hidden_channels, num_classes)

    self.convs = torch.nn.ModuleList()
    for layer in range(num_layers):
      self.convs.append(
        GCN2Conv(hidden_channels, 0.1, 0.5, layer + 1, shared_weights, normalize=False))
    self.h = None
    self.dropout = dropout
    self.optimizer = torch.optim.Adam(self.parameters(),lr=0.01,weight_decay=5e-4)

  def forward(self, x, edge_index):

    x = F.dropout(x, self.dropout, training=self.training)
    x =  self.lins1(x)
    x = x_0 = x.relu()

    for conv in self.convs:
      x = F.dropout(x, self.dropout, training=self.training)
      x = conv(x, x_0, edge_index)
      x = x.relu()
    x = F.dropout(x, self.dropout, training=self.training)
    x = self.lins2(x)
    self.h =x
    return x, x.log_softmax(dim=-1)
```

https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.GCN2Conv.html#torch_geometric.nn.conv.GCN2Conv

DeepGCNs framework

- Solving vanishing gradient problem by using Residual Learning for GCNs





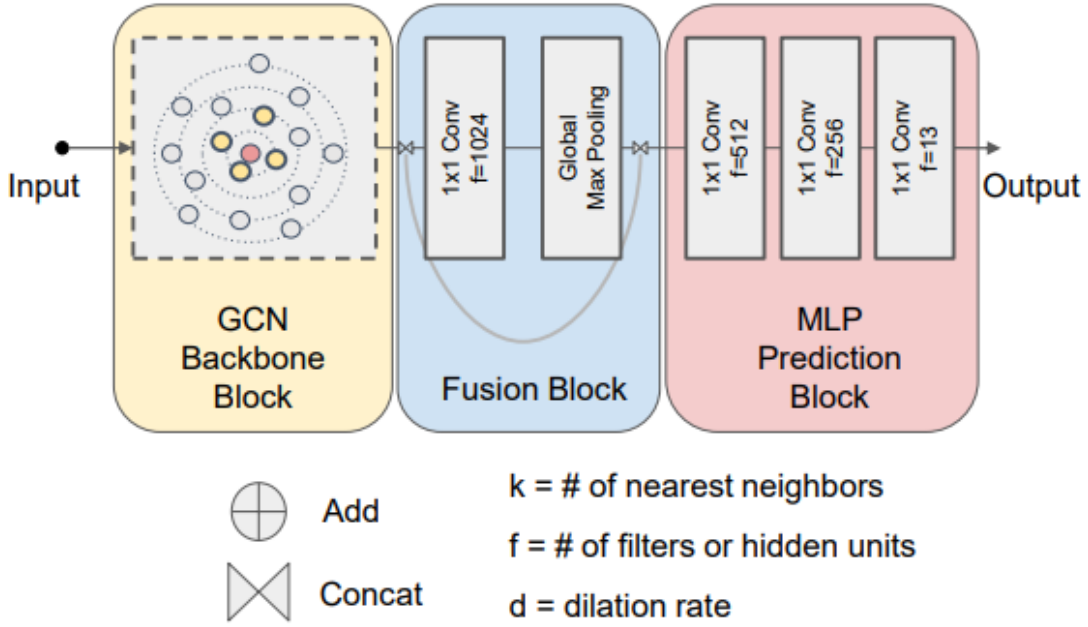
$$\begin{aligned}\mathcal{G}_{l+1} &= \mathcal{H}(\mathcal{G}_l, \mathcal{W}_l) \\ &= \mathcal{F}(\mathcal{G}_l, \mathcal{W}_l) + \mathcal{G}_l = \mathcal{G}^{res}_{l+1} + \mathcal{G}_l.\end{aligned} \tag{3}$$

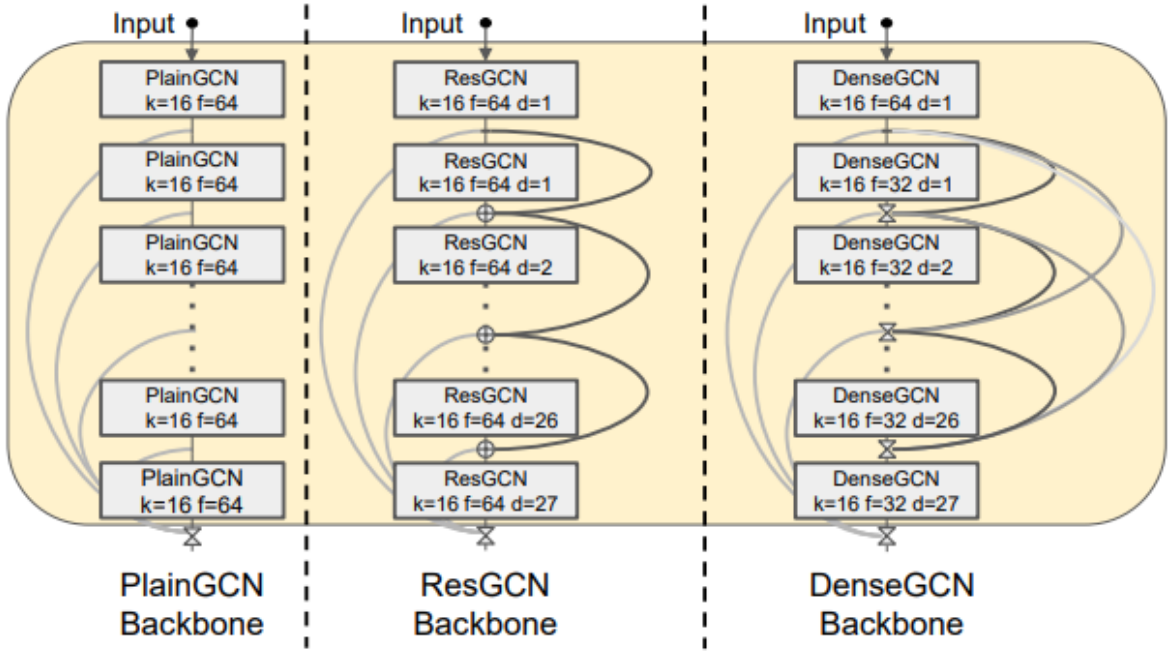
$\mathcal{G}_l = (\mathcal{V}_l, \mathcal{E}_l)$ and $\mathcal{G}_{l+1} = (\mathcal{V}_{l+1}, \mathcal{E}_{l+1})$ are the input and output graphs at the l-th layer

DeepGCNs: Can GCNs Go as Deep as CNNs?; ICCV 2019

DeepGCNs framework

- Framework consists of three blocks (GCN Backbone Block, Fusion Block and MLP Prediction Block).
- Three types of GCN Backbone Blocks (PlainGCN, ResGCN and DenseGCN).
- There are two kinds of GCN skip connections vertex-wise additions and vertex-wise concatenations. k is the number of nearest neighbors in GCN layers.

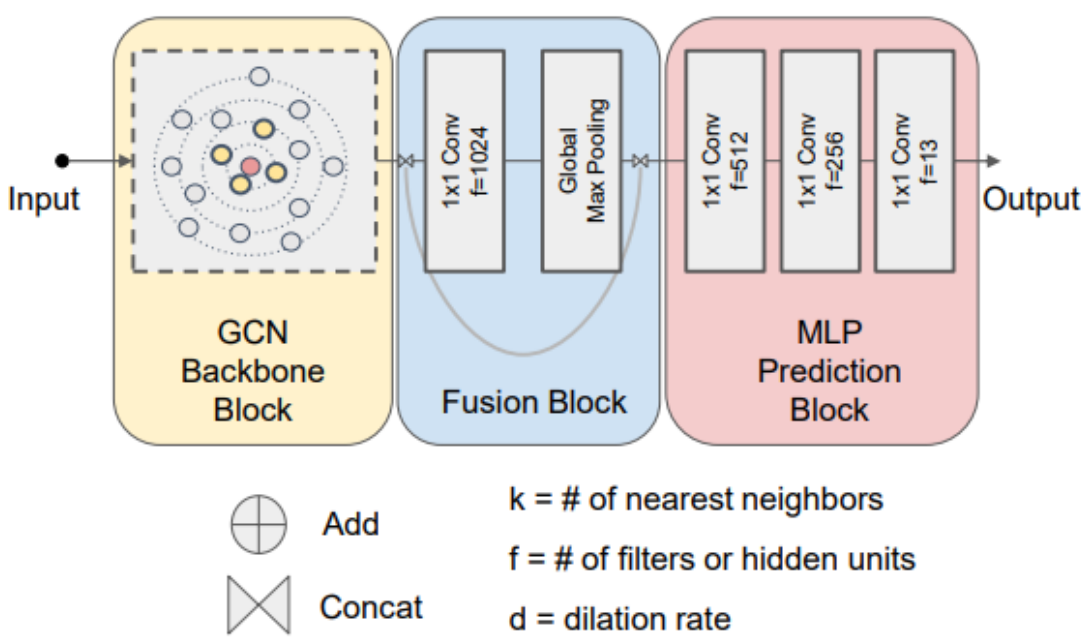


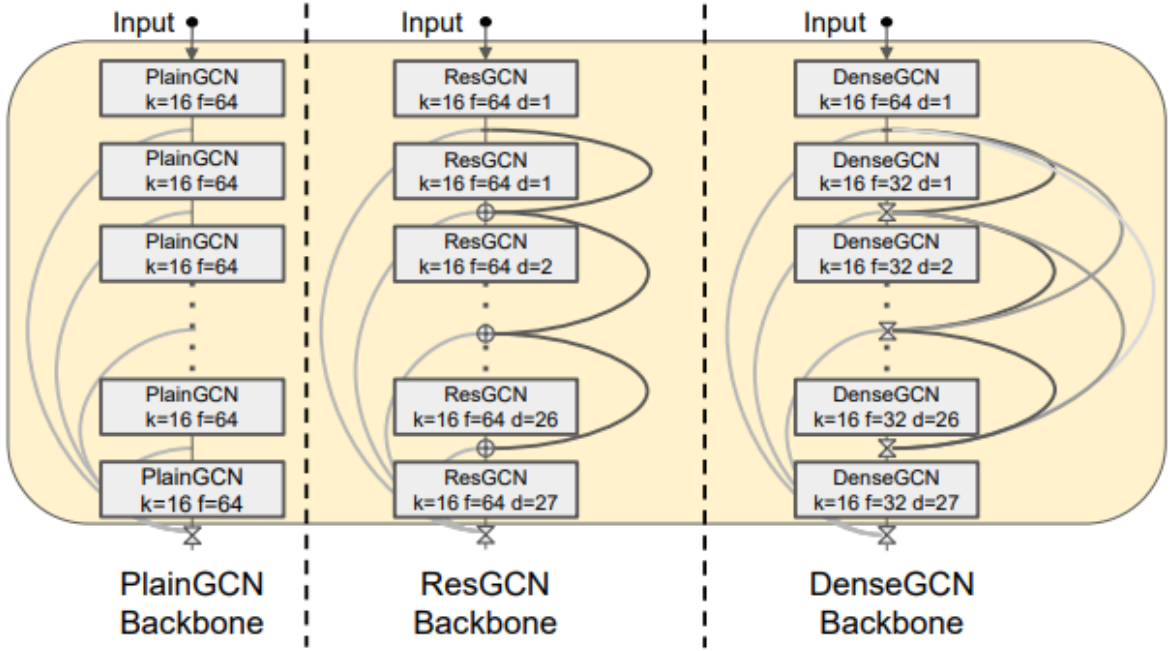


DeepGCNs: Can GCNs Go as Deep as CNNs?; ICCV 2019

DeepGCNs framework

- **PlainGCN**: a fusion block, and a MLP prediction block
- **ResGCN**: construct ResGCN by adding dynamic dilated k-NN and residual graph connections to PlainGCN.
- **DenseGCN**: adding dynamic dilated k-NN and dense graph connections to the PlainGCN.

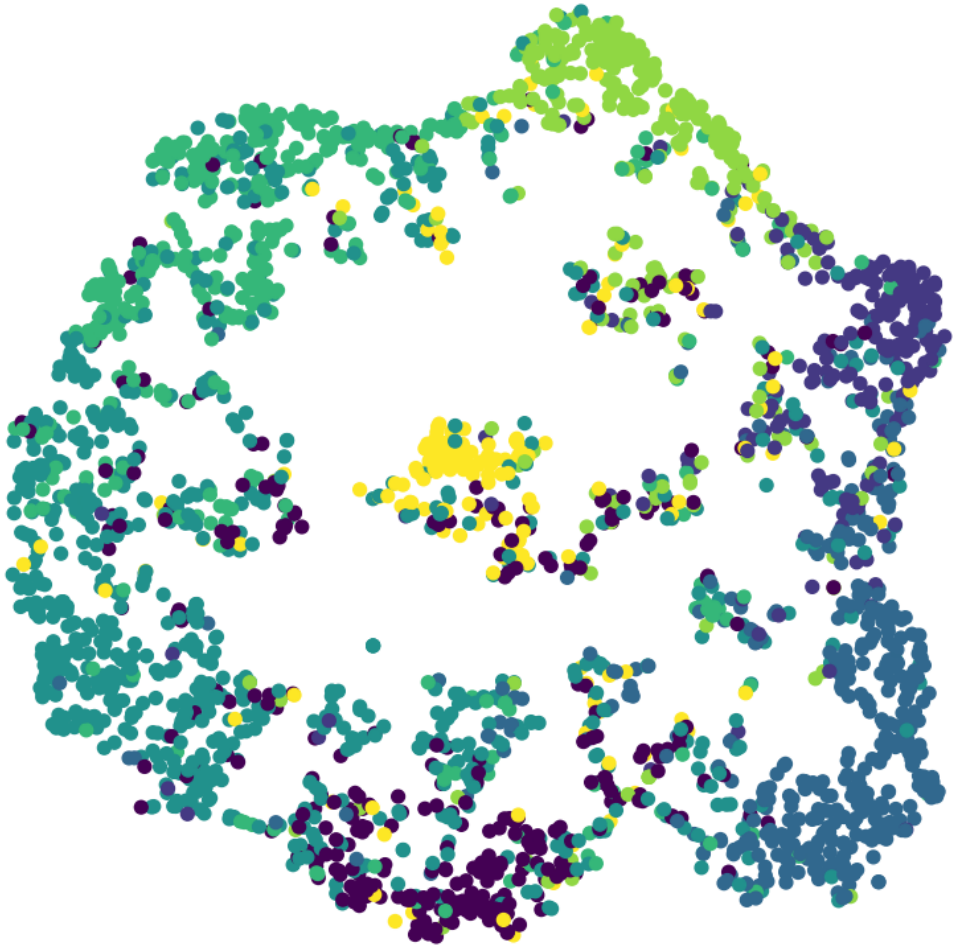




DeepGCNs: Can GCNs Go as Deep as CNNs?; ICCV 2019

DeepGCNs: Sample code


```python
class DeepGCN(torch.nn.Module):
  def __init__(self,num_features, num_classes, dim =16, num_layers =2, drop=0.5):
    super().__init__()

    self.node_encoder = Linear(num_features, dim)
    #self.edge_encoder = Linear(data.edge_attr.size(-1), hidden_channels)
    self.h = None
    self.layers = torch.nn.ModuleList()
    for i in range(1, num_layers + 1):
      conv = GENConv(dim, dim, aggr='softmax', t=1.0, learn_t=True, num_layers=2, norm='layer')
      norm = LayerNorm(dim, elementwise_affine=True)
      act = ReLU(inplace=True)

      layer = DeepGCNLayer(conv, norm, act, block='res+', dropout=0.1,ckpt_grad=i % 3)
      self.layers.append(layer)
    self.mlp = Linear(dim, num_classes)
    self.optimizer = torch.optim.Adam(self.parameters(),lr=0.01,weight_decay=5e-4)


  def forward(self, x, edge_index):
    x = self.node_encoder(x)
    x = self.layers[0].conv(x, edge_index)

    for layer in self.layers[1:]:
      x = layer(x, edge_index)

    x = self.layers[0].act(self.layers[0].norm(x))
    x = F.dropout(x, p=0.1, training=self.training)
    x = self.mlp(x)
    self.h = x
    return x, F.log_softmax(x, dim=1)
```



네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA