Graph Neural Network: SoTR

Emre Anakok & Pierre Barbillon
15/11/24

Outline

Outline

Data at hand

Convolution layers for networks

What to do with GCN

Data

Graph $G = (V, E, W)$ with

- a set of nodes $V = \{1, \ldots, N\}$,
- a set of edges $E \subset V^2$, particular cases: (un)directed, with(out) loop,...
- additional information on edges, $w \in W$ containing weights (number of interactions, positive or negative interaction,...)

Equivalence of list of edges, adjacency matrices...

Additional attributes for nodes: covariates for any $i \in V$, X_i attributes of a node (taxon, gender, age, social group,...), or information derived from the edges: degree of i.

Outline

Convolution on graphs

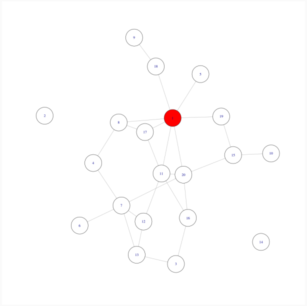
- particular structure,
- isomorphism of graphs up to relabelling the nodes,
- large graphs but sparse,
- convolution on graph, convolution on images (images can be seen as graph with fixed number of neighbors)

Convolution with neighbors: x features on nodes:

$$h_i = \sum_{j \in \mathcal{N}(i)} x_j$$

$\mathcal{N}(i)$ is the set of neighbors of node i.

convolutional layer



Graph Convolution Networks

[Kipf and Welling, 2016]

$$h_i^{(\ell+1)} = \sigma\left(\mathbf{W}^{(\ell+1)} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{c_{i,j}} \cdot \mathbf{h}_j^{(\ell)}\right)$$

Importance of normalization $c_{i,j}$.

Matrix form

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}\right)$$

with

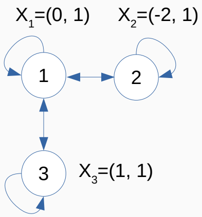
- $W^{(l)}$ a matrix of trainable parameters,
- $\tilde{A} = A + I$,
- D the diagonal matrix of degrees of $\tilde{A}$.

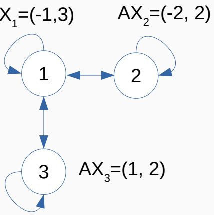
Importance of normalization

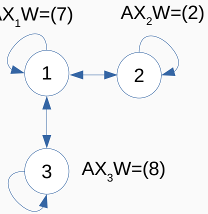
Different choices:

- No normalization $\tilde{A}$
  - $h_i^{l+1} = \sum_{j, j \in \mathcal{N}(i)} A_{ij} h_j^l$,
  - Eigenvalue of $\tilde{A}$ larger than $1 \Rightarrow$ exploding largest eigenvalue when stacking layers,
- row normalization $A_{\text{row}} = D^{-1}A$,
  - $h_i^{l+1} = \sum_{j, j \in \mathcal{N}(i)} A_{ij} \frac{h_j^l}{d_i}$
  - largest eigenvalue is 1 but not taken into account connectivity of neighbors,
- col normalization $A_{\text{col}} = AD^{-1}$
  - $h_i^{l+1} = \sum_{j, j \in \mathcal{N}(i)} A_{ij} \frac{h_j^l}{d_j}$
  - largest eigenvalue is 1 but put too much weight on well connected nodes,
- Naive normalization $A_{\text{naive}} = D^{-1}AD^{-1}$
  - $h_i^{l+1} = \sum_{j, j \in \mathcal{N}(i)} A_{ij} \frac{h_j^l}{d_j d_i}$
  - largest eigenvalue is < 1 and vanishes when stacking layers,
- symmetric normalization $A_{\text{sym}} = D^{-1/2}AD^{-1/2}$
  - $h_i^{l+1} = \sum_{j, j \in \mathcal{N}(i)} A_{ij} \frac{h_j^l}{\sqrt{d_j d_i}}$
  - largest eigenvalue is 1, combine row and col normalization.

Graph Convolutional Network







$$\tilde{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, X = \begin{pmatrix} 0 & 1 \\ -2 & 1 \\ 1 & 1 \end{pmatrix}, W = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Other kinds of GNN

- Graph Convolution Networks as we have seen,
- Graph Attention Networks (GAT) [Casanova et al., 2018],
  - $h_i^l = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha^l(i,j) W h_j^{l-1} \right)$,
  - $\alpha^l(i,j)$ is the attention function,
  - $\alpha^l(i,j) = \text{softmax}\left(\sigma'\left(a^\top \cdot (Wh_i, Wh_j)\right)\right)$.
- Graph SAGE (SAmple and agGrEgate) [Hamilton et al., 2017],
  - $h_{\mathcal{N}(i)}^l = AGGREGATE_k(\{h_j^{l-1}, j \in \mathcal{N}(i)\})$,
  - $h_i^l = \sigma(W^l \cdot CONCAT(h_i^{l-1}, h_{\mathcal{N}(i)}^l))$,
  - $h_i^l = h_i^l / \|h_i^l\|$.
- Graph Isomorphism Network (GIN) [Xu et al., 2018].

see https://distill.pub/2021/understanding-gnns/

Outline

Semi-supervised learning on nodes

Data: $G = (V, E)$ a network with N nodes, $m\%$ of nodes with an observed labels in $\{1, \dots, Q\}$, V set of edges is known, (features on nodes X).

Goal: Classify nodes without labels.

Architecture:

- X can be a vector of the degrees of nodes, a number for each node, or an identity matrix...
- 2 or 3 GCN layers with given numbers of features,
- Last layer is a linear transformation in a K dimension space : for each p point from the dataset $(h^L_{p1}, \dots, h^L_{pK})$.

Loss: Cross entropy:

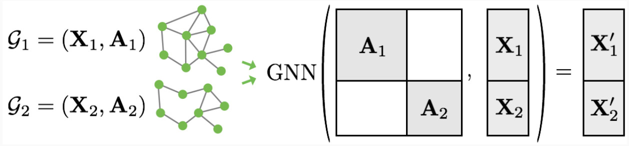
$$loss(x, y) = \frac{1}{n_{\text{train}}} \sum_{p=1}^{n_{\text{train}}} \log \left(\frac{\exp(h^L_{p,y_p})}{\sum_{k=1}^{K} \exp(h^L_{p,k})} \right) .$$

Graph classification

Data: $G_1, \dots G_n$ and labels on graphs.

Goal: Learn the Classification function $f : G \mapsto \{1, \dots, K\}$

Architecture:



$\mathcal{G}_1 = (\mathbf{X}_1, \mathbf{A}_1)$
$\mathcal{G}_2 = (\mathbf{X}_2, \mathbf{A}_2)$
GNN
$\mathbf{A}_1$
$\mathbf{A}_2$
$\mathbf{X}_1$
$\mathbf{X}_2$
=
$\mathbf{X}'_1$
$\mathbf{X}'_2$

and batches

Average over nodes in the same graph in order to have a layer at the graph level and use a classifier.

Loss: cross entropy.

Link Prediction

Data: $G = (V, E)$, V is incomplete.
Goal: Find edges that are likely to exist for a given set of non-observed edges...

Architecture: GCN layers with V as the set of edges... Last layer uses a "decoder" for dyads:

$$g(\text{Dist}(h_i^l, h_j^l)) \text{ or } h_i^{l\top} h_j^l$$

Loss: Cross entropy computed on a set of trainable DYADS (usually half of edges and half of non edges).

Remark: Autoencoder directly derived from link prediction task by using h_i^l as the embedding.

GVAE

Data: $G = (V, E)$.

Goal: Find an embedding of nodes in a small dimension (Euclidean) space as a conditional distribution.

Architecture: GCN layers to embed the nodes in the parameters of a Gaussian distribution, simulation under the distribution and a last decoder layer to predict edges.

$$(X_i)_i \to (m_i, s_i)_i \to (Z_i = m_i + s_i \cdot \mathcal{N}(0,1))_i \to (Z_i^\top Z_j)_{ij}$$

Loss: Cross entropy with a KL on the set of trainable DYADS:

$$\mathbb{E}_{q(Z|X,A)}\big(\log p(A_{\text{train}}|Z)\big) - KL(q(Z|X,A)||p(Z))$$

where $p(Z)$ is a prior distribution chosen as $\mathcal{N}(0,1)$.

Many resources online

- introduction to GNN https://distill.pub/2021/gnn-intro/,
- convolution on graphs https://distill.pub/2021/understanding-gnns/,
- google colabs for pytorch geometric https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html.

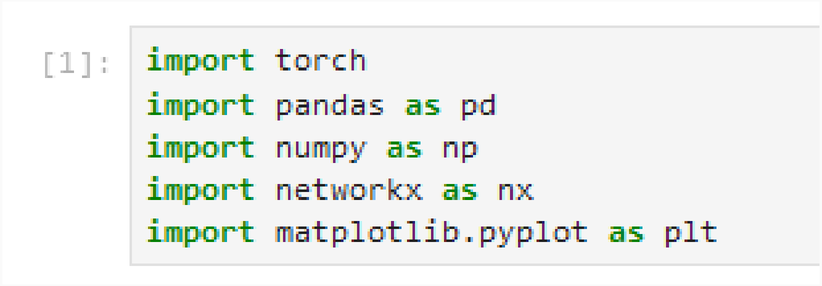
Casanova, P., Lio, A. R. P., and Bengio, Y. (2018).
Graph attention networks.
ICLR. Petar Velickovic Guillem Cucurull Arantxa Casanova Adriana Romero Pietro Liò and Yoshua Bengio.

Hamilton, W., Ying, Z., and Leskovec, J. (2017).
Inductive representation learning on large graphs.
Advances in neural information processing systems, 30.

Kipf, T. N. and Welling, M. (2016).
Semi-supervised classification with graph convolutional networks.
arXiv preprint arXiv:1609.02907.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2018).
How powerful are graph neural networks?
arXiv preprint arXiv:1810.00826.

Required packages

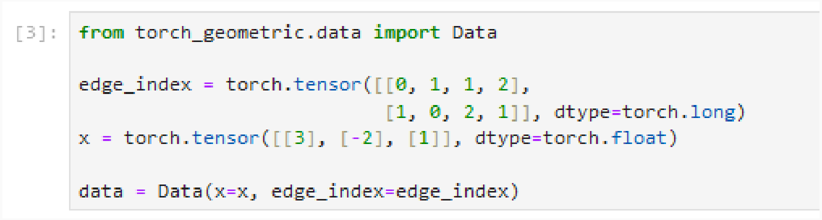


```
[1]: import torch
     import pandas as pd
     import numpy as np
     import networkx as nx
     import matplotlib.pyplot as plt
```

Data format

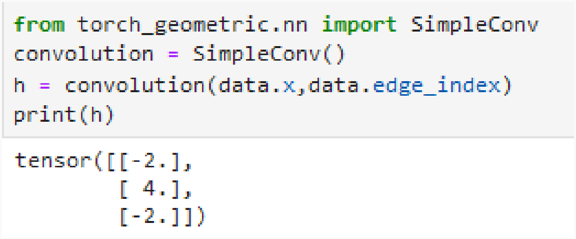
Format:

- $2 \times |E|$ Tensor: edge index.
- $|V| \times d$ Tensor: node features.



```
[3]: from torch_geometric.data import Data

     edge_index = torch.tensor([[0, 1, 1, 2],
                                [1, 0, 2, 1]], dtype=torch.long)
     x = torch.tensor([[3], [-2], [1]], dtype=torch.float)

     data = Data(x=x, edge_index=edge_index)
```

Simple convolution

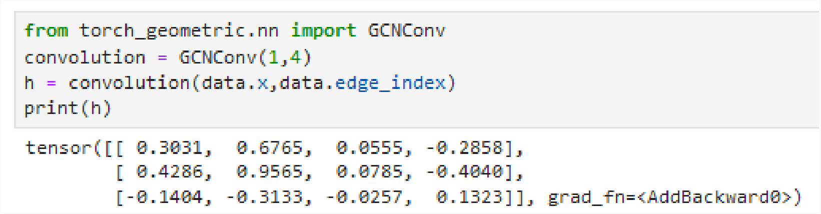


```
from torch_geometric.nn import SimpleConv
convolution = SimpleConv()
h = convolution(data.x,data.edge_index)
print(h)
```

```
tensor([[-2.],
        [ 4.],
        [-2.]])
```

More convolution

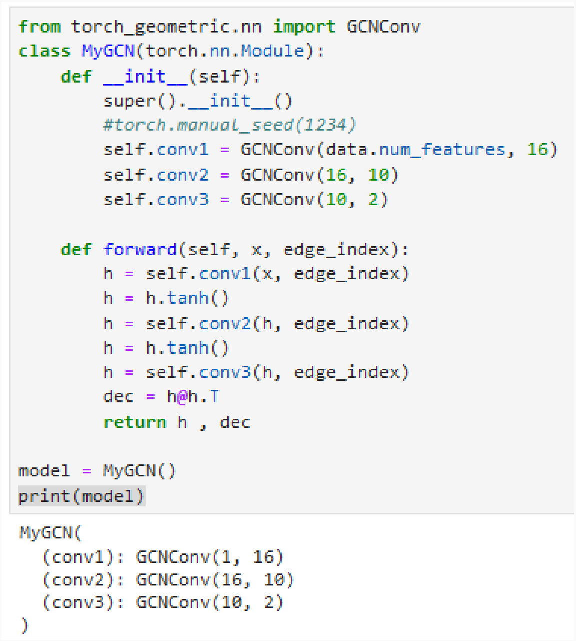
All the convolution are available at https://pytorch-geometric.readthedocs.io/en/latest/cheatsheet/gnn_cheatsheet.html



```
from torch_geometric.nn import GCNConv
convolution = GCNConv(1,4)
h = convolution(data.x,data.edge_index)
print(h)
```

```
tensor([[ 0.3031,  0.6765,  0.0555, -0.2858],
        [ 0.4286,  0.9565,  0.0785, -0.4040],
        [-0.1404, -0.3133, -0.0257,  0.1323]], grad_fn=<AddBackward0>)
```

Simple architecture

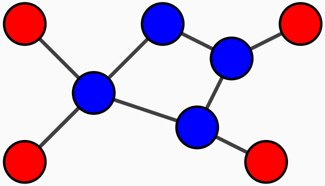


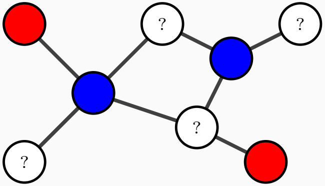
```python
from torch_geometric.nn import GCNConv
class MyGCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #torch.manual_seed(1234)
        self.conv1 = GCNConv(data.num_features, 16)
        self.conv2 = GCNConv(16, 10)
        self.conv3 = GCNConv(10, 2)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        dec = h@h.T
        return h , dec

model = MyGCN()
print(model)
```

```
MyGCN(
  (conv1): GCNConv(1, 16)
  (conv2): GCNConv(16, 10)
  (conv3): GCNConv(10, 2)
)
```

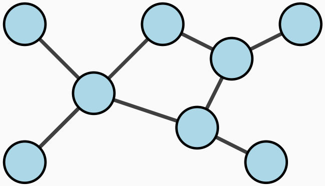
Mask

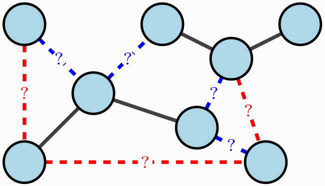
Mask nodes



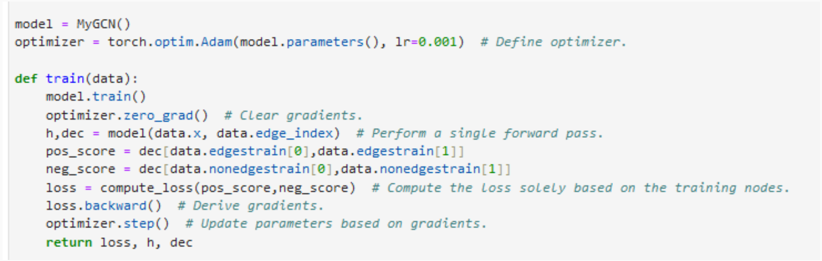


Mask edges





Pytorch training framework

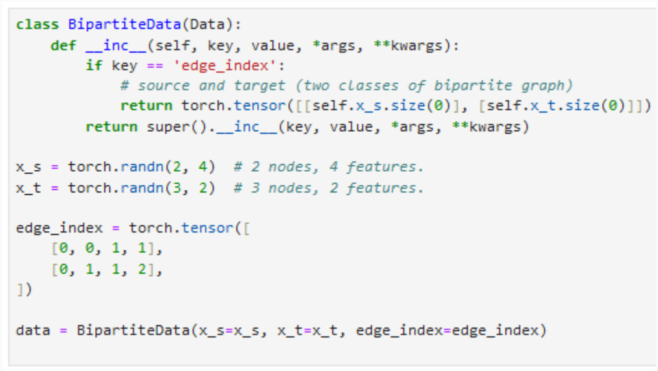


```
model = MyGCN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  # Define optimizer.

def train(data):
    model.train()
    optimizer.zero_grad()  # Clear gradients.
    h,dec = model(data.x, data.edge_index)  # Perform a single forward pass.
    pos_score = dec[data.edgestrain[0],data.edgestrain[1]]
    neg_score = dec[data.nonedgestrain[0],data.nonedgestrain[1]]
    loss = compute_loss(pos_score,neg_score)  # Compute the loss solely based on the training nodes.
    loss.backward()  # Derive gradients.
    optimizer.step()  # Update parameters based on gradients.
    return loss, h, dec
```

Specificity for bipartite network

Format:

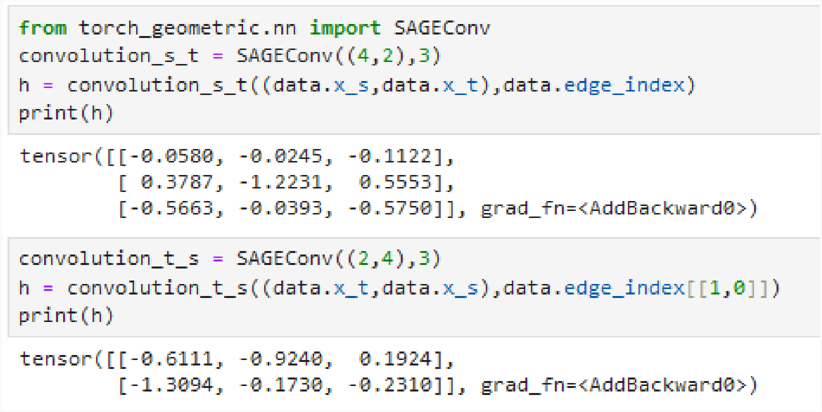
- $2 \times |E|$ Tensor: edge index.
- $|V_s| \times d_s$ Tensor: source node features.
- $|V_t| \times d_t$ Tensor: target node features.



```
class BipartiteData(Data):
    def __inc__(self, key, value, *args, **kwargs):
        if key == 'edge_index':
            # source and target (two classes of bipartite graph)
            return torch.tensor([[self.x_s.size(0)], [self.x_t.size(0)]])
        return super().__inc__(key, value, *args, **kwargs)

x_s = torch.randn(2, 4)  # 2 nodes, 4 features.
x_t = torch.randn(3, 2)  # 3 nodes, 2 features.

edge_index = torch.tensor([
    [0, 0, 1, 1],
    [0, 1, 1, 2],
])

data = BipartiteData(x_s=x_s, x_t=x_t, edge_index=edge_index)
```

Bipartite convolution

Bipartite convolution are directed !



```
from torch_geometric.nn import SAGEConv
convolution_s_t = SAGEConv((4,2),3)
h = convolution_s_t((data.x_s,data.x_t),data.edge_index)
print(h)
```

```
tensor([[-0.0580, -0.0245, -0.1122],
        [ 0.3787, -1.2231,  0.5553],
        [-0.5663, -0.0393, -0.5750]], grad_fn=<AddBackward0>)
```

```
convolution_t_s = SAGEConv((2,4),3)
h = convolution_t_s((data.x_t,data.x_s),data.edge_index[[1,0]])
print(h)
```

```
tensor([[-0.6111, -0.9240,  0.1924],
        [-1.3094, -0.1730, -0.2310]], grad_fn=<AddBackward0>)
```