EI320A(3) 深度學習使用 Python

Instructors

Tipajin Thaipisutikul (t.greentip@gmail.com)

Prof. Huang-Chia Shih (hcshih@Saturn.yzu.edu.tw)

Week	Date	Content	Note	Total
1	2/26	Welcome to the course	Homework (1)	1
2	3/5	Crash Course of Python, NumPy, Pandas, and Matplotlib	In class hands-on (4)	5
3	3/12	Get to know about Data, ML: Classification Models	In class hands-on (5)	10
4	3/19	ML: Regression Models	In class hands-on (5)	15
5	3/26	ML: Clustering/Apriori Models	In class hands-on (5)	20
6	4/2	Holiday		
7	4/9	Introduction to Deep Learning (ANN)		
8	4/16	ANN Labs, Introduction to Convolutional Neural Network (CNN)	In class hands-on (10)	30
9	4/23	Convolutional Neural Network (CNN) & CNN Labs	In class hands-on (5)	35
10	4/30	Introduction to Recurrent Neural Network (RNN)	In class hands-on (5)	40
11	5/7	Recurrent Neural Network (RNN) & RNN Labs	In class hands-on (5)	45
12	5/14	Wrap Up all ANN, CNN, RNN Project Proposal Presentation	Proposal Presentation (10)	55
13	5/21	Generative Adversarial Network (GAN)	In class hands-on (5)	60
14	5/28	Reinforcement Learning (RL)	In class hands-on (5)	65
15	6/4	NLP & S2S & Attention Neural Network	In class hands-on (5)	70
16	6/11	Graph Convolutional Networks (GCN)	In class hands-on (5)	75
17	6/18	Final Project Presentation	Final Presentation (25)	100

GCN Plan of Attack

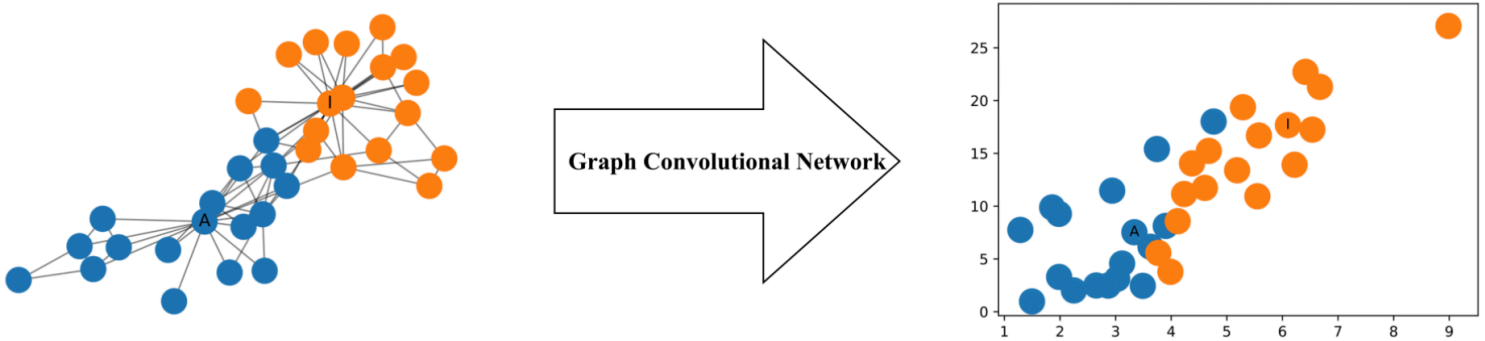
1. What is Graph Convolutional Networks? (GCN)
2. A Simple Propagation Rule & A Graph Example
3. Problem on Horizon & Solution
4. Put Them All Together & Exercise

1. What is Graph Convolutional Networks? (GCN)

- Graph Convolutional Networks (GCNs) is a powerful type of neural network designed to work directly on **graphs and leverage their structural information**
- We'll see **how the GCN aggregates information** from the previous layers and how this mechanism produces useful **feature representations of nodes in graphs.**

- GCNs are a very powerful **neural network** architecture for machine learning on **graphs**. In fact, they are so powerful that even *a randomly initiated 2-layer GCN* can produce useful feature representations of nodes in networks.

1. What is Graph Convolutional Networks? (GCN)



Given a graph $\boldsymbol{G = (V, E)}$, a GCN takes as input

1. an input feature matrix $N \times F^0$ feature matrix, $\boldsymbol{X}$, where N is the number of nodes and F^0 is the number of input features for each node, and
2. an $N \times N$ matrix representation of the graph structure such as the adjacency matrix $\boldsymbol{A}$ of G.

- A hidden layer in the GCN can thus be written as $\boldsymbol{H^i = f(H^{i-1}, A))}$ where $\boldsymbol{H^0 = X}$ and f is a propagation.
- Each layer $\boldsymbol{H^i}$ corresponds to an $N \times F^i$ feature matrix, where each row is a feature representation of a node.
- At each layer, these features are aggregated to form the next layer's features using the propagation rule f. In this way, features become increasingly more abstract at each consecutive layer. In this framework, variants of GCN differ only in the choice of propagation rule f

GCN Plan of Attack

1. What is Graph Convolutional Networks? (GCN)
2. A Simple Propagation Rule & A Graph Example
3. Problem on Horizon & Solution
4. Put Them All Together & Exercise

2. A Simple Propagation Rule & Graph Example

One of the simplest possible propagation rule is

$$f(\boldsymbol{H}^{i}, \boldsymbol{A}) = \sigma(\boldsymbol{A}\boldsymbol{H}^{i}\boldsymbol{W}^{i})$$

where $\boldsymbol{W}^{i}$ is the weight matrix for layer i and σ is a non-linear activation function such as ReLu

Simplifications

Let's examine the propagation rule at its most simple level.

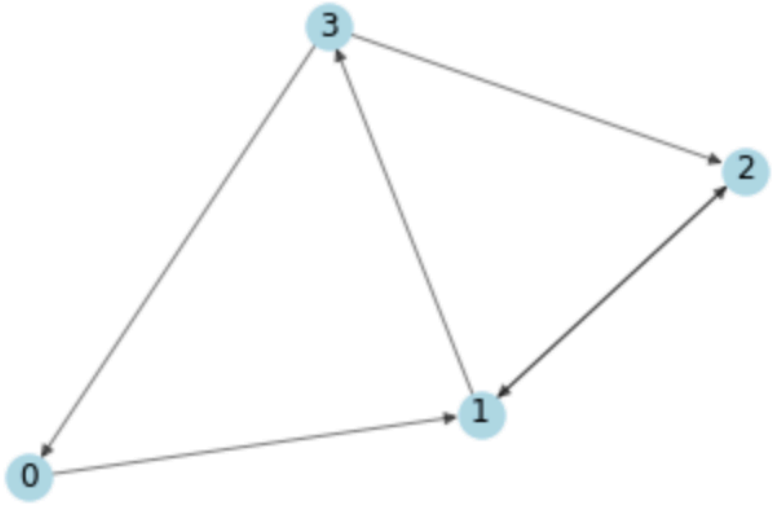
Let

1. $i = 1$, s.t. f is a function of the input feature matrix,
2. σ be the identity function, and
3. choose the weights: $\boldsymbol{A}\boldsymbol{H}^{0}\boldsymbol{W}^{0} = \boldsymbol{A}\boldsymbol{X}\boldsymbol{W}^{0} = \boldsymbol{A}\boldsymbol{X}$.

In other words, $f(\boldsymbol{X}, \boldsymbol{A}) = \boldsymbol{A}\boldsymbol{X}$.

- This propagation rule is perhaps a bit too simple, but we will add in the missing parts later.
- As a side note, **AX** is now equivalent to the input layer of a multi-layer perceptron.

2. A Simple Propagation Rule & Graph Example



A simple directed graph.

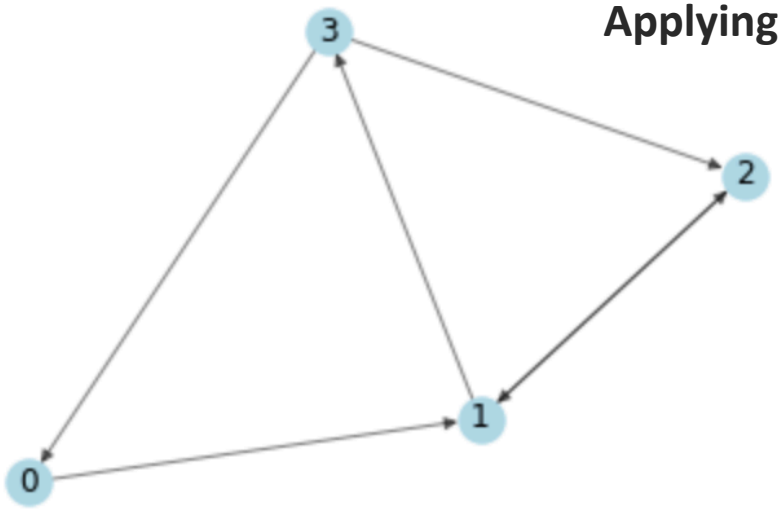
```
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

Next, we need features! We generate 2 integer features for every node based on its index. This makes it easy to confirm the matrix calculations manually later.

```
In [3]: X = np.matrix([
            [i, -i]
            for i in range(A.shape[0])
        ], dtype=float)
        X


Out[3]: matrix([
           [ 0.,  0.],
           [ 1., -1.],
           [ 2., -2.],
           [ 3., -3.]
        ])
```

2. A Simple Propagation Rule & Graph Example



A simple directed graph.

Applying the Propagation Rule

$$f(H^{i}, A) = \sigma(AH^{i}W^{i})$$

```
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

```
Out[3]: matrix([
            [ 0.,  0.],
            [ 1., -1.],
            [ 2., -2.],
            [ 3., -3.]
        ])
```

```
In [6]: A * X
Out[6]: matrix([
            [ 1., -1.],
            [ 5., -5.],
            [ 1., -1.],
            [ 2., -2.]]
```

- What happened? The representation of each node (each row) is now a sum of its neighbors features!
- In other words, the graph convolutional layer represents each node as an aggregate of its neighborhood.
- Note that in this case a node n is a neighbor of node v if there exists an edge from v to n.

GCN Plan of Attack

1. What is Graph Convolutional Networks? (GCN)
2. A Simple Propagation Rule & A Graph Example
3. Problem on Horizon & Solution
4. Put Them All Together & Exercise

3. Problem on Horizon & Solution

Problems

1. The aggregated representation of a node does **not include its own features**!
   - The representation is an aggregate of the features of neighbor nodes, so only nodes that has a self-loop will include their own features in the aggregate.

2. Nodes with **large degrees** will have **large values** in their feature representation while nodes with **small degrees** will have **small values.**
   - This can cause vanishing or exploding gradients, but is also problematic for stochastic gradient descent algorithms which are typically used to train such networks and are sensitive to the scale (or range of values) of each of the input features.

3. Problem on Horizon & Solution

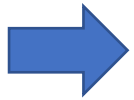
Solutions

1. **Adding Self-Loops**

- We can simply add a self-loop to each node. In practice this is done by adding the **identity matrix I** to **the adjacency matrix A** before applying the propagation rule.
- Since the node is now a neighbor of itself, the node's own features is included when summing up the features of its neighbors!

```
In [4]: I = np.matrix(np.eye(A.shape[0]))
        I

Out[4]: matrix([
            [1., 0., 0., 0.],
            [0., 1., 0., 0.],
            [0., 0., 1., 0.],
            [0., 0., 0., 1.]
        ])
```



```
In [8]: A_hat = A + I
        A_hat * X
Out[8]: matrix([
            [ 1., -1.],
            [ 6., -6.],
            [ 3., -3.],
            [ 5., -5.]])
```

Previously (Without Self-Loops)

```
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

```
Out[3]: matrix([
            [ 0.,  0.],
            [ 1., -1.],
            [ 2., -2.],
            [ 3., -3.]
        ])
```

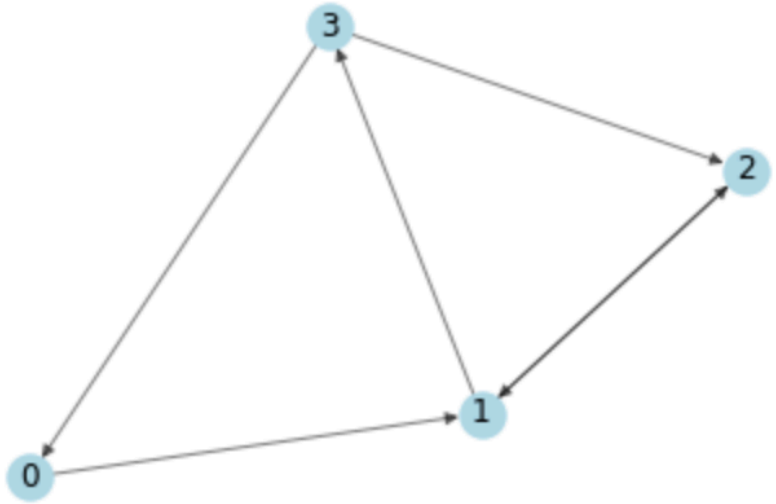
```
In [6]: A * X
Out[6]: matrix([
            [ 1., -1.],
            [ 5., -5.],
            [ 1., -1.],
            [ 2., -2.]]
```

3. Problem on Horizon & Solution

Solutions

2. Normalizing the Feature Representations

- The feature representations can be normalized by **node degree** by **transforming the adjacency matrix A** by multiplying it with **the inverse degree matrix D.**
- Thus our simplified propagation rule looks like this:



A simple directed graph.

$$f(X, A) = D^{-1}AX$$

We first compute the degree matrix.

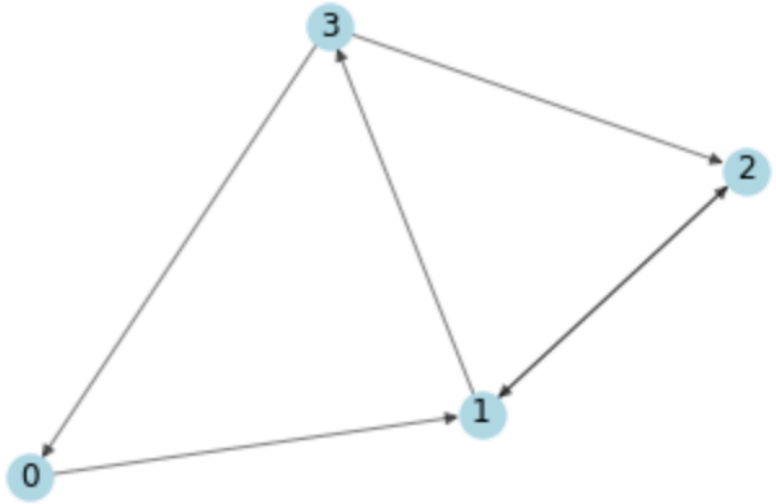
```
In [9]: D = np.array(np.sum(A, axis=0))[0]
        D = np.matrix(np.diag(D))
        D
Out[9]: matrix([
            [1., 0., 0., 0.],
            [0., 2., 0., 0.],
            [0., 0., 2., 0.],
            [0., 0., 0., 1.]
        ])
```

3. Problem on Horizon & Solution

Solutions

2. Normalizing the Feature Representations

- The feature representations can be normalized by **node degree** by **transforming the adjacency matrix A** by multiplying it with **the inverse degree matrix D.**
- Thus our simplified propagation rule looks like this:



A simple directed graph.

$$f(X, A) = D^{-1}AX$$

We first compute the degree matrix.

Before

```
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype=float
)
```

After

```
In [10]: D**-1 * A
Out[10]: matrix([
             [0. , 1. , 0. , 0. ],
             [0. , 0. , 0.5, 0.5],
             [0. , 0.5, 0. , 0. ],
             [0.5, 0. , 0.5, 0. ]
])
```

3. Problem on Horizon & Solution

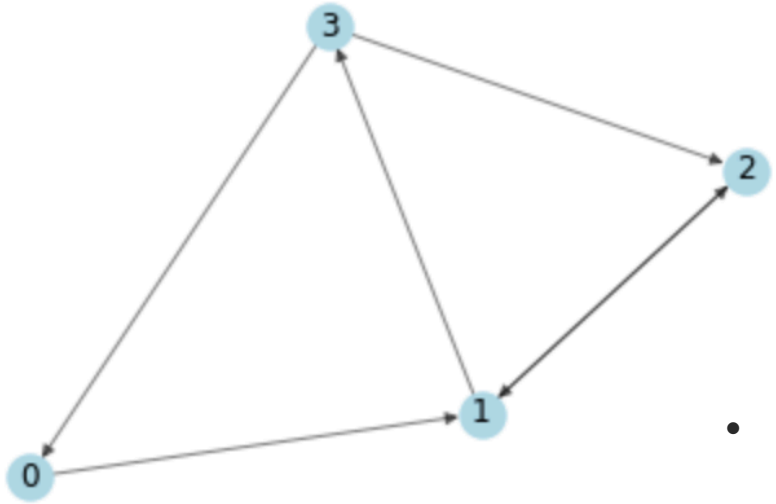
Solutions

2. Normalizing the Feature Representations

- The feature representations can be normalized by **node degree** by **transforming the adjacency matrix A** by multiplying it with **the inverse degree matrix D.**
- Thus our simplified propagation rule looks like this:

$$f(X, A) = D^{-1}AX$$

```
In [11]: D**-1 * A * X
Out[11]: matrix([
                [ 1. , -1. ],
                [ 2.5, -2.5],
                [ 0.5, -0.5],
                [ 2. , -2. ]
         ])
```



A simple directed graph.

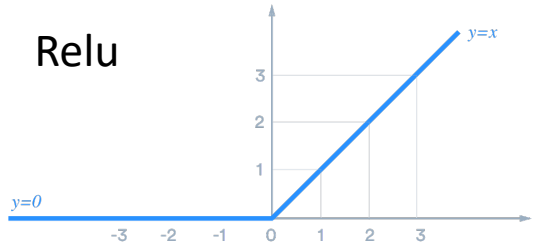
- Observe that the weights (the values) in each row of the adjacency matrix have been divided by the degree of the node corresponding to the row.
- We apply the propagation rule with the transformed adjacency matrix and get node representations corresponding to the mean of the features of neighboring nodes.
- This is because the weights in the (transformed) adjacency matrix correspond to weights in a weighted sum of the neighboring nodes' features.

GCN Plan of Attack

1. What is Graph Convolutional Networks? (GCN)
2. A Simple Propagation Rule & A Graph Example
3. Problem on Horizon & Solution
4. Put Them All Together & Exercise

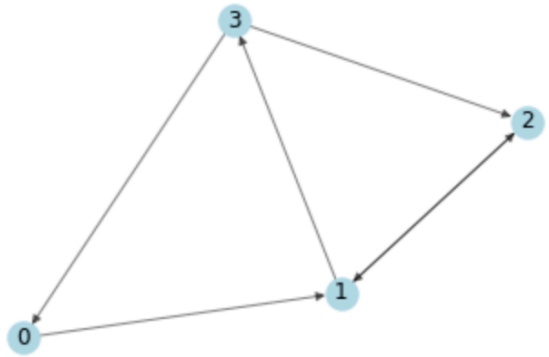
4. Put Them All Together & Exercise

Relu



We now combine the self-loop and normalization tips.
In addition, we'll **reintroduce the weights and activation function** that we previously discarded to simplify the discussion.

$$f(H^i, A) = \sigma(D^{-1}\hat{A}H^iW^i)$$



A simple directed graph.

$A =$
```
matrix([[0, 1, 0, 0],
        [0, 0, 1, 1],
        [0, 1, 0, 0],
        [1, 0, 1, 0]])
```

$I =$
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

$D =$
```
matrix([[1, 0, 0, 0],
        [0, 2, 0, 0],
        [0, 0, 2, 0],
        [0, 0, 0, 1]])
```

$\hat{A} =$
```
matrix([[1., 1., 0., 0.],
        [0., 1., 1., 1.],
        [0., 1., 1., 0.],
        [1., 0., 1., 1.]])
```

$D^{-1} =$
```
matrix([[1. , 0. , 0. , 0. ],
        [0. , 0.5, 0. , 0. ],
        [0. , 0. , 0.5, 0. ],
        [0. , 0. , 0. , 1. ]])
```

$X =$
```
matrix([[ 0,  0,  0],
        [ 1, -1,  1],
        [ 2, -2,  2],
        [ 3, -3,  3]])
```

$W =$
```
matrix([[ 1, -1],
        [-1,  1],
        [ 1, -1]])
```

$D^{-1}\hat{A} =$
```
matrix([[1. , 1. , 0. , 0. ],
        [0. , 0.5, 0.5, 0.5],
        [0. , 0.5, 0.5, 0. ],
        [1. , 0. , 1. , 1. ]])
```

$D^{-1}\hat{A}H^i =$
```
matrix([[ 1. , -1. ,  1. ],
        [ 3. , -3. ,  3. ],
        [ 1.5, -1.5,  1.5],
        [ 5. , -5. ,  5. ]])
```

$D^{-1}\hat{A}H^iW^i =$
```
matrix([[  3. ,  -3. ],
        [  9. ,  -9. ],
        [  4.5,  -4.5],
        [ 15. , -15. ]])
```

$\sigma(D^{-1}\hat{A}H^iW^i) =$

Code Examples

- https://towardsdatascience.com/kegra-deep-learning-on-knowledge-graphs-with-keras-98e340488b93
- https://pypi.org/project/keras-gcn/
- https://github.com/CyberZHG/keras-gcn