Published in Towards Data Science

Inneke Mayachita Follow

Jun 10, 2020 · 9 min read · ✨ · Listen

Save

Understanding Graph Convolutional Networks for Node Classification

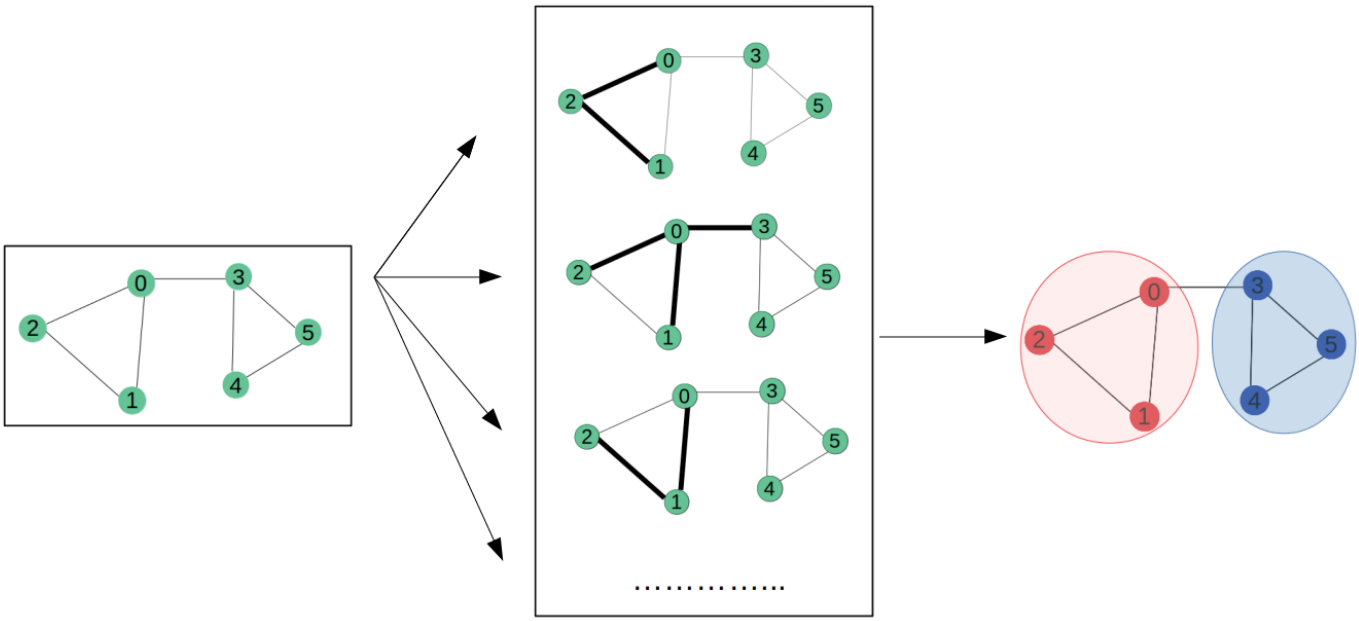


Illustration of Graph Convolutional Networks (image by author)

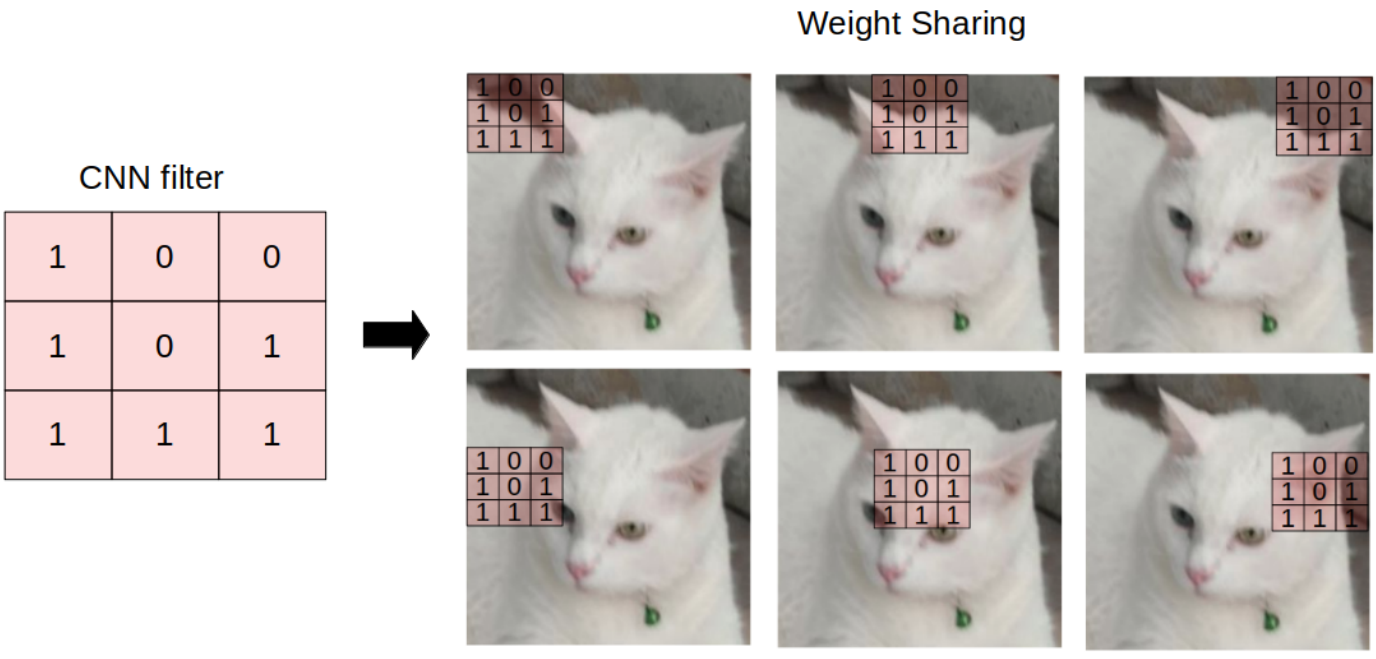
Neural Networks have gained massive success in the last decade. However, early variants of Neural Networks could only be implemented using regular or Euclidean data, while a lot of data in the real world have underlying graph structures which are non-Euclidean. The non-regularity of data structures have led to recent advancements in Graph Neural Networks. In the past few years, different variants of Graph Neural Networks are being developed with Graph Convolutional Networks (GCN) being one of

building our first graph using NetworkX. By the end of this article, I hope we can gain deeper understanding on the mechanisms inside Graph Convolutional Networks.

If you are not familiar with the basic concepts of Graph Neural Networks, I recommend reading my previous article here.

Convolution in Graph Neural Networks

If you are familiar with convolution layers in Convolutional Neural Networks, 'convolution' in GCNs is basically the same operation. It refers to multiplying the input neurons with a set of weights that are commonly known as *filters* or *kernels*. The filters act as a sliding window across the whole image and enable CNNs to learn features from neighboring cells. Within the same layer, the same filter will be used throughout image, this is referred to as **weight sharing**. For example, using CNN to classify images of cats vs non-cats, the same filter will be used in the same layer to detect the nose and the ears of the cat.



The same weight (or kernel, or filter in CNNs) is applied throughout the image (image by author)

specially built to operate on regular (Euclidean) structured data, while GNNs are the generalized version of CNNs where the numbers of nodes connections vary and the nodes are unordered (irregular on non-Euclidean structured data).

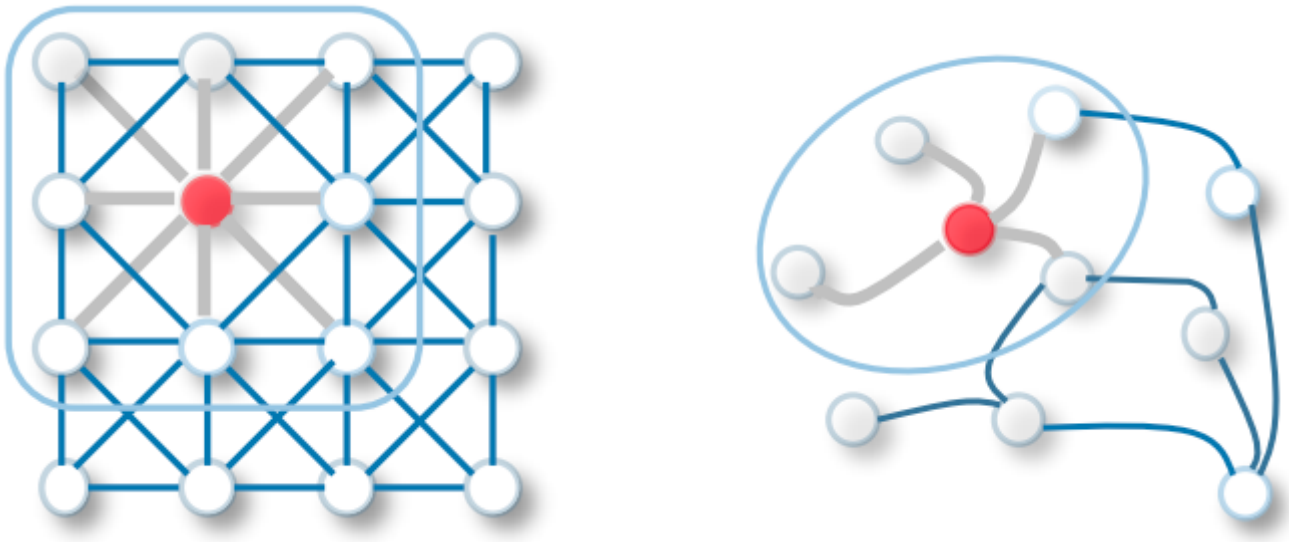


Illustration of 2D Convolutional Neural Networks (left) and Graph Convolutional Networks (right), via source

GCNs themselves can be categorized into 2 major algorithms, **Spatial Graph Convolutional Networks** and **Spectral Graph Convolutional Networks.** In this article, we will be focusing on Fast Approximation Spectral-based Graph Convolutional Networks.

Before diving into the calculations happening inside GCNs, let's briefly recap the concept of forward propagation in Neural Networks first. You can skip the following section if you're familiar with it.

Neural Networks Forward Propagation Brief Recap

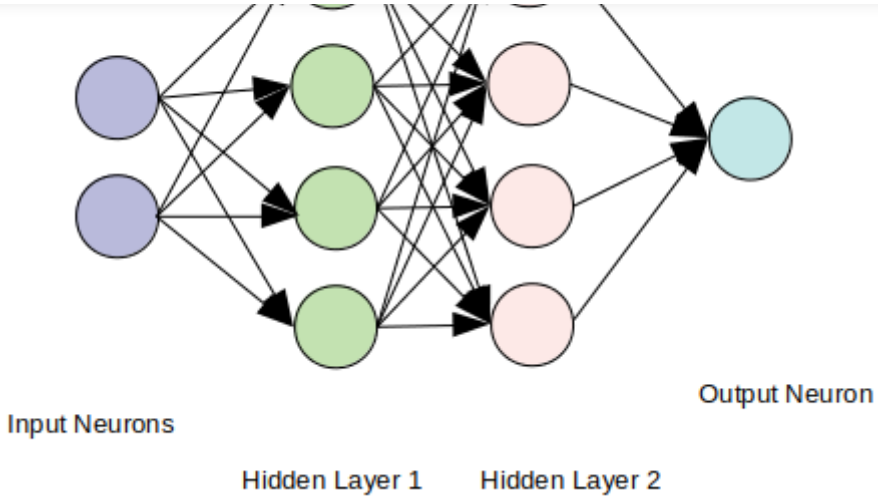


Illustration of Fully-Connected Neural Networks (image by author)

In Neural Networks, in order to propagate the features representation to the next layer (forward pass), we perform the equation below:

$$H^{[i+1]} = \sigma(W^{[i]} H^{[i]} + b^{[i]})$$

- $H^{[i+1]}$: feature representation at layer i+1
- σ: activation function
- $W^{[i]}$: weights at layer i
- $H^{[i]}$: feature representation at layer i
- $b^{[i]}$: bias at layer i

Equation 1 — Forward Pass in Neural Networks

This is basically equivalent to **y = mx+b** in **Linear Regression**, where:

m is equivalent to the **weights**

x is the **input features**

b is the **bias**

What distinguishes the forward pass equation above from Linear Regression is that Neural Networks apply **non-linear activation functions** in order to represent the non-

$$H^{[1]} = \sigma(W^{T\,[0]} X + b^{[0]})$$

Equation 2 — Forward Pass in Neural Networks at the first layer

where **features representation at layer 0** is basically the **input features (X)**.

How does this equation differ in Graph Convolutional Networks?

Fast Approximate Spectral Graph Convolutional Networks

The original idea behind Spectral GCN was inspired by signal/wave propagation. We can think of information propagation in Spectral GCN as signal propagation along the nodes. Spectral GCNs make use of the Eigen-decomposition of graph Laplacian matrix to implement this method of information propagation. To put it simply, the Eigen-decomposition helps us understand the graph structure, hence, classifying the nodes of the graphs. This is somewhat similar to the basic concept of Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) where we use Eigen-decomposition to reduce dimensionality and perform clustering. If you have never heard of Eigen-decomposition and Laplacian matrix, *don't worry!* In this Fast Approximation method, we are not going to use them explicitly.

In this approach, we will take into account the **Adjacency Matrix (A)** in the forward propagation equation in addition to the node features (or so-called input features). **A** is a matrix that represents the edges or connection between the nodes in the forward propagation equation. The insertion of **A** in the forward pass equation enables the model to learn the feature representations based on nodes connectivity. For the sake of simplicity, the bias **b** is omitted. The resulting GCN can be seen as the first-order approximation of Spectral Graph Convolution in the form of a message passing network where the information is propagated along the neighboring nodes within the graph.

By adding the adjacency matrix as an additional element, the forward pass equation

Equation 3— Forward Pass in Graph Convolutional Networks

Wait.. You said A, what is A?*

A* is the normalized version of **A**. To get better understanding on why we need to normalize **A** and what happens during forward pass in GCNs, let's do an experiment.

Building Graph Convolutional Networks

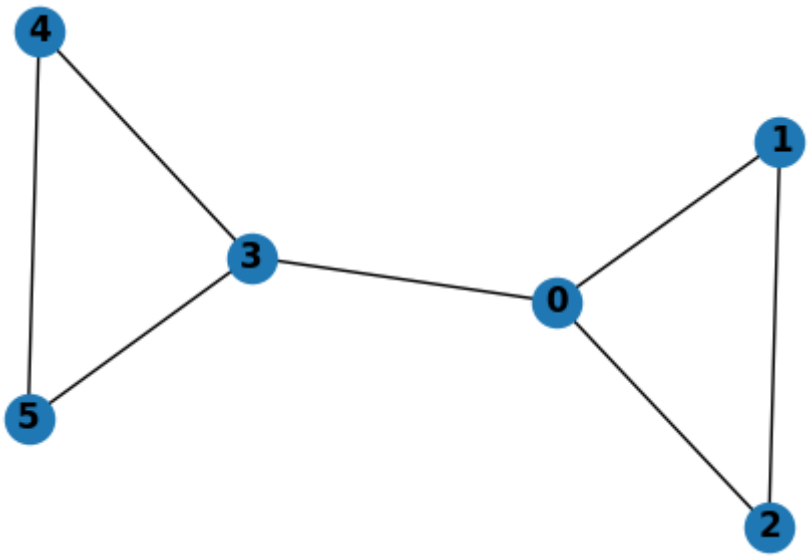
Initializing the Graph G

Let's start by building a simple undirected graph (**G**) using NetworkX. The graph **G** will consist of 6 nodes and the feature of each node will correspond to that particular node number. For example, node 1 will have a node feature of 1, node 2 will have a node feature of 2, and so on. To simplify, we are not going to assign edge features in this experiment.

Output:

```
Graph Info:
 Name: G
Type: Graph
Number of nodes: 6
Number of edges: 7
Average degree:   2.3333

Graph Nodes:  [(0, {'name': 0}), (1, {'name': 1}), (2, {'name': 2}), (3, {'name': 3}), (4, {'name': 4}), (5, {'nam
e': 5})]
```



Graph G visualization

Since we only have 1 graph, this data configuration is an example of a **Single Mode** representation. We will build a GCN that will learn the nodes features representation.

Inserting Adjacency Matrix (A) to Forward Pass Equation

Output:




```
Adjacency Matrix (A):
 [[0. 1. 1. 1. 0. 0.]
 [1. 0. 1. 0. 0. 0.]
 [1. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 1. 1.]
 [0. 0. 0. 1. 0. 1.]
 [0. 0. 0. 1. 1. 0.]]

Node Features Matrix (X):
 [[0]
 [1]
 [2]
 [3]
 [4]
 [5]]
```

Now, let's investigate how by inserting **A** into the forward pass equation adds to richer feature representation of the model. We are going to perform dot product of **A** and **X**. Let's call the result of this dot product operation as *AX* in this article.




Output:

```
Dot product of A and X (AX):
 [[6.]
 [2.]
 [1.]
 [9.]
 [8.]
 [7.]]
```

From the results, it is apparent that *AX* represents the **sum of neighboring nodes features**. For example, the first row of *AX* corresponds to the sum of nodes features connected to node 0, which is node 1, 2, and 3. This gives us an idea how the propagation mechanism is happening in GCNs and how the node connectivity impacts the hidden features representation seen by GCNs.

> *The dot product of Adjacency Matrix and Node Features Matrix represents the sum of neighboring node features.*

But, if we think about it more, we will realize that while *AX* sums up the adjacent node features, **it does not take into account the features of the node itself.**

Oops, problem detected! How to solve it?

Inserting Self-Loops and Normalizing A

To address this problem, we now add self-loops to each node of **A.** Adding self-loops is basically a mechanism to connect a node to itself. That being said, all the diagonal elements of Adjacency Matrix **A** will now become 1 because each node is connected to itself. Let's call **A** with self-loops added as **A_hat** and recalculate *AX*, which is now the dot product of **A_hat** and **X**:

Output:

```
Edges of G with self-loops:
 [(0, 1), (0, 2), (0, 3), (0, 0), (1, 2), (1, 1), (2, 2), (3, 4), (3, 5), (3, 3), (4, 5), (4, 4), (5, 5)]
Adjacency Matrix of added self-loops G (A_hat):
 [[1. 1. 1. 1. 0. 0.]
 [1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 0. 0. 0.]
 [1. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]]
AX:
 [[ 6.]
 [ 3.]
 [ 3.]
 [12.]
 [12.]
 [12.]]
```

Great! One problem solved!




normalize the features to prevent numerical instabilities and vanishing/exploding gradients in order for the model to converge. In GCNs, we normalize our data by **calculating the Degree Matrix (D) and performing dot product operation of the inverse of D with** *AX*

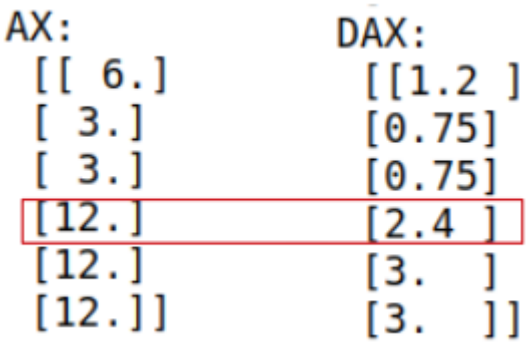
$$\text{normalized features} = D^{-1}AX$$

which we will call *DAX* in this article. In graph terminology, the term “degree” refers to the number of edges a node is connected to.

Output:

```
Degree Matrix of added self-loops G (D):  [(0, 5), (1, 4), (2, 4), (3, 5), (4, 4), (5, 4)]
Degree Matrix of added self-loops G as numpy array (D):
 [[5 0 0 0 0 0]
 [0 4 0 0 0 0]
 [0 0 4 0 0 0]
 [0 0 0 5 0 0]
 [0 0 0 0 4 0]
 [0 0 0 0 0 4]]
Inverse of D:
 [[0.2  0.   0.   0.   0.   0.  ]
 [0.   0.25 0.   0.   0.   0.  ]
 [0.   0.   0.25 0.   0.   0.  ]
 [0.   0.   0.   0.2  0.   0.  ]
 [0.   0.   0.   0.   0.25 0.  ]
 [0.   0.   0.   0.   0.   0.25]]
DAX:
 [[1.2 ]
 [0.75]
 [0.75]
 [2.4 ]
 [3.  ]
 [3.  ]]
```

If we compare *DAX* with *AX*, we will notice that:



We can see the impact normalization has on DAX, where the element that corresponds to node 3 has lower values compared to node 4 and 5. But why would node 3 have different values after normalization if it has the same initial value as node 4 and 5?

Let's take a look back at our graph. Node 3 has **3 incident edges**, while nodes 4 and 5 only have **2 incident edges**. The fact that **node 3 has a higher degree** than node 4 and 5 leads to a **lower weighting of node 3's features in *DAX***. In other words, the lower the

$$\text{normalizing term} = D^{-1} A$$

to

$$\text{normalizing term} = D^{-1/2} A D^{-1/2}$$

Let's calculate the normalized values using the new symmetric normalization equation:

Output:

```
DADX:
 [[1.27082039]
 [0.75      ]
 [0.75      ]
 [2.61246118]
 [2.92082039]
 [2.92082039]]
```

Looking back at Equation 3 in the previous section, we will realize that we now have the answers to what is A*! In the paper, A* is referred to as *renormalization trick*.

Having finished with features handling, it's time to finalize our GCN.

Adding Weights and Activation Function

We are going to build a 2-layer GCN using ReLu as the activation function. To initialize the weights, we will use random seeds so we can replicate the results. Just keep in mind that the weight initialization cannot be 0. In this experiment, we are going to set 4 neurons for the hidden layer. As we will be plotting the feature representations in 2 dimensions, there will be 2 output neurons.

Just to make it simpler, we will re-write the *renormalization trick* equation using numpy, just to make it simpler.




Output:

```
Features Representation from GCN output:
 [[0.00027758 0.        ]
 [0.00017298 0.        ]
 [0.00017298 0.        ]
 [0.00053017 0.        ]
 [0.00054097 0.        ]
 [0.00054097 0.        ]]
```

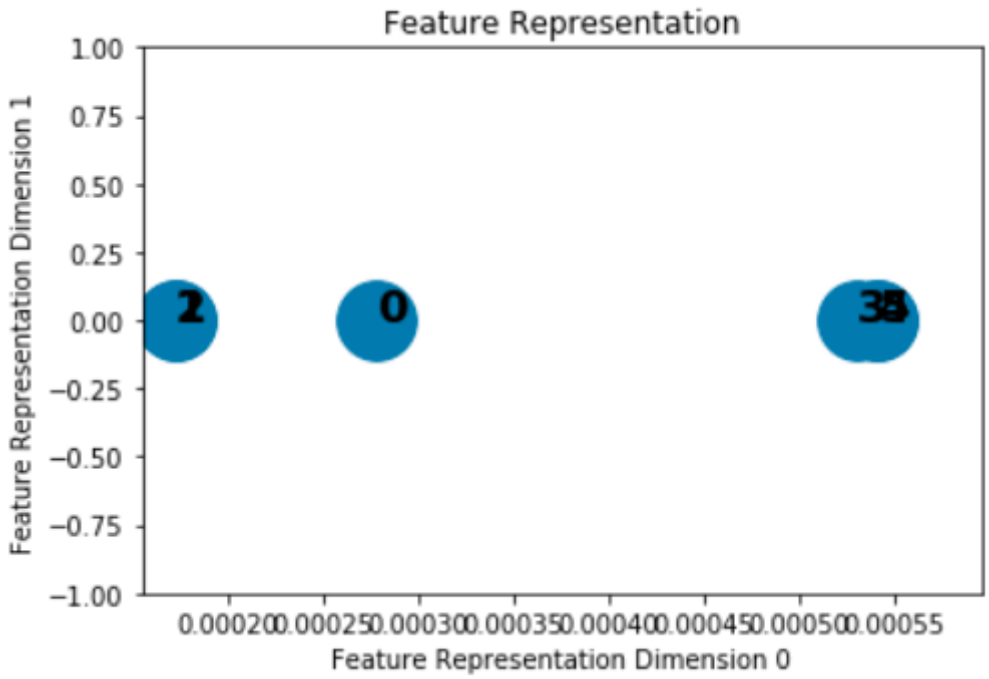
Done! We have just built our first feed-forward GCN model!

Plotting the Features Representations

The 'magic' of GCN is that it can learn features representation even without training. Let's visualize the features representations after passing through 2-layer GCN.




Output:



Features Representation from Feed-Forward GCN

Key Takeaways

- The term 'convolution' in Graph Convolutional Networks is similar to Convolutional Neural Networks in terms of *weight sharing*. The main difference lies in the data structure, where GCNs are the generalized version of CNN that can work on data with underlying non-regular structures.
- The insertion of Adjacency Matrix (**A**) in the forward pass equation of GCNs enable the model to learn the features of neighboring nodes. This mechanism can be seen as a message passing operation along the nodes within the graph.
- *Renormalization trick* is used to normalize the features in Fast Approximate Spectral-based Graph Convolutional Networks by Thomas Kipf and Max Welling (2017).
- GCNs can learn features representation even before training.

Thanks for reading! If you want to read about how to train a GCN on node classification task using CORA dataset, you can read the next article in this series.

Any comment, feedback, or want to discuss? Just drop me a message. You can reach me on LinkedIn.

You can get the full code on GitHub.

References

[1] T. Kipf and M. Welling, Semi-Supervised Classification with Graph Convolutional Networks (2017). arXiv preprint arXiv:1609.02907. ICLR 2017

[2] T. Kipf, https://tkipf.github.io/graph-convolutional-networks/

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Emails will be sent to xbarryzexin@gmail.com. Not you?

Get this newsletter