How to do Deep Learning on Graphs with Graph Convolutional Networks

Part 1: A High-Level Introduction to Graph Convolutional Networks

Tobias Skovgaard Jepsen Sep 18, 2018 · 9 min read

Machine learning on graphs is a difficult task due to the highly complex, but also informative graph structure. This post is the first in a series on how to do deep learning on graphs with Graph Convolutional Networks (GCNs), a powerful type of neural network designed to work directly on graphs and leverage their structural information. The posts in the series are:

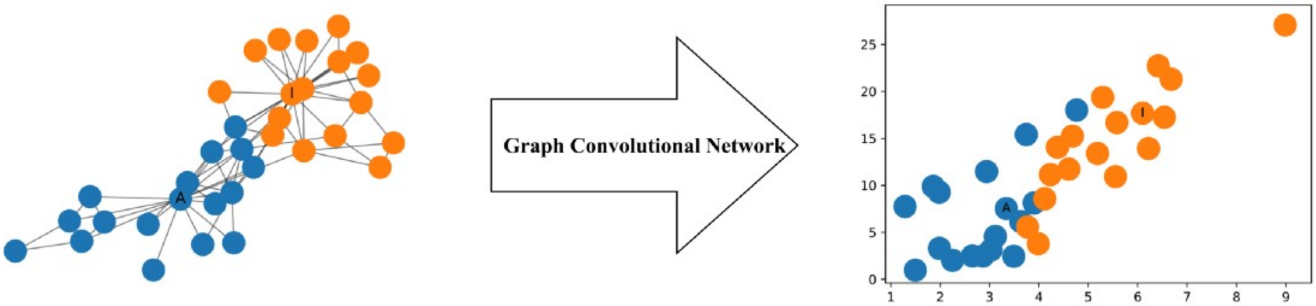
1. A High-Level Introduction to Graph Convolutional Networks (this)
2. Semi-Supervised Learning with Spectral Graph Convolutions

In this post, I will give an introduction to GCNs and illustrate how information is propagated through the hidden layers of a GCN using coding examples. We'll see how the GCN aggregates information from the previous layers and how this mechanism produces useful feature representations of nodes in graphs.

What is a Graph Convolutional Network?

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. The figure below illustrates a 2-dimensional representation of each node in a network produced by such a GCN.

Notice that the relative nearness of nodes in the network is preserved in the 2-dimensional representation even without any training.



More formally, a *graph convolutional network (GCN)* is a neural network that operates on graphs. Given a graph $G = (V, E)$, a GCN takes as input

- 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
- an $N \times N$ matrix representation of the graph structure such as the adjacency matrix $\boldsymbol{A}$ of G.[1]

A hidden layer in the GCN can thus be written as $\boldsymbol{H^i} = f(\boldsymbol{H}^{i-1}, \boldsymbol{A}))$ where $\boldsymbol{H^0} = \boldsymbol{X}$ and f is a propagation [1]. Each layer $\boldsymbol{H^i}$ corresponds to an $N \times F^{\boldsymbol{i}}$ 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 [1].

A Simple Propagation Rule

One of the simplest possible propagation rule is [1]:

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

where $\boldsymbol{W^i}$ is the weight matrix for layer i and σ is a non-linear activation function such as the ReLU function. The weight matrix has dimensions $F^{\boldsymbol{i}} \times F^{i+1}$; in other words the size of the second dimension of the weight matrix determines the number of features at the next layer. If you are familiar with convolutional neural networks, this operation is similar to a filtering operation since these weights are shared across nodes in the graph.

Simplifications

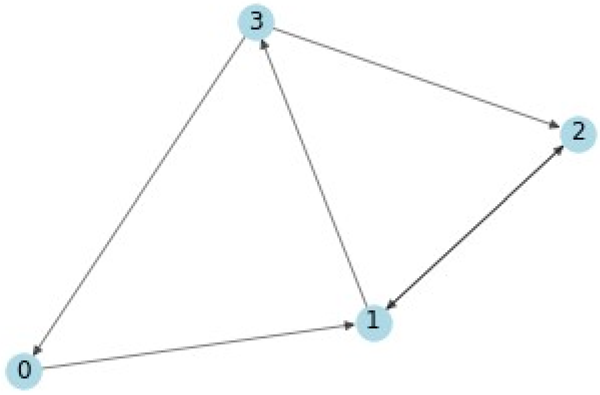
Let's examine the propagation rule at its most simple level. Let

- $i = 1$, s.t. f is a function of the input feature matrix,
- σ be the identity function, and
- choose the weights s.t. $\boldsymbol{AH^0W^0} = \boldsymbol{AXW^0} = \boldsymbol{AX}$.

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

A Simple Graph Example

As a simple example, we'll use the the following graph:



A simple directed graph.

And below is its `numpy` adjacency matrix representation.

```
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.]
        ])
```

Applying the Propagation Rule

Alright! We now have a graph, its adjacency matrix `A` and a set of input features `X`. Let's see what happens when we apply the propagation rule:

```
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. I encourage you to check the calculation for yourself. Note that in this case a node *n* is a neighbor of node *v* if there exists an edge from *v* to *n*.

Uh oh! Problems on the Horizon!

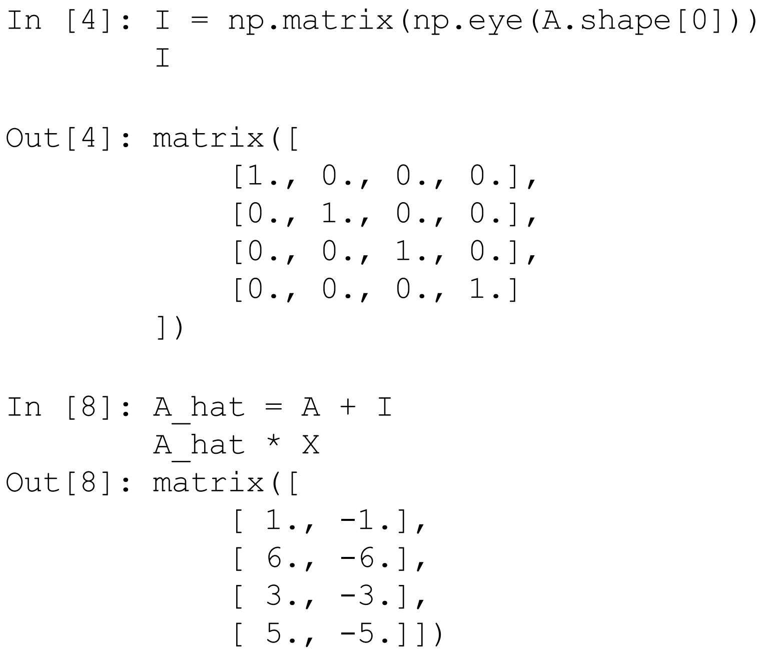
You may have already spotted the problems:

- 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.[1]
- 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 [1, 2], 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.

In the following, I discuss each of these problems separately.

Adding Self-Loops

To address the first problem, one can simply add a self-loop to each node [1, 2]. In practice this is done by adding the identity matrix `I` to the adjacency matrix `A` before applying the propagation rule.



```
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.]])
```

Since the node is now a neighbor of itself, the node's own features is included when summing up the features of its neighbors!

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` [1]. Thus our simplified propagation rule looks like this [1]:

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

Let's see what happens. 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.]
        ])
```

Before applying the rule, let's see what happens to the adjacency matrix after we transform it.

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. ]
])
```

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

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

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. Again, I encourage you to verify this observation for yourself.

Putting it All Together

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.

Adding back the Weights

First order of business is applying the weights. Note that here `D_hat` is the degree matrix of `A_hat = A + I`, i.e., the degree matrix of `A` with forced self-loops.

```
In [45]: W = np.matrix([
             [1, -1],
             [-1, 1]
         ])
         D_hat**-1 * A_hat * X * W
Out[45]: matrix([
            [ 1., -1.],
            [ 4., -4.],
            [ 2., -2.],
            [ 5., -5.]
        ])
```

And if we want to reduce the dimensionality of the output feature representations we can reduce the size of the weight matrix `W`:

```
In [46]: W = np.matrix([
             [1],
             [-1]
         ])
         D_hat**-1 * A_hat * X * W
Out[46]: matrix([[1.],
        [4.],
        [2.],
        [5.]]
)
```

Adding an Activation Function

We choose to preserve the dimensionality of the feature representations and apply the ReLU activation function.

```
In [51]: W = np.matrix([
             [1, -1],
             [-1, 1]
         ])
         relu(D_hat**-1 * A_hat * X * W)
Out[51]: matrix([[1., 0.],
        [4., 0.],
        [2., 0.],
        [5., 0.]])
```

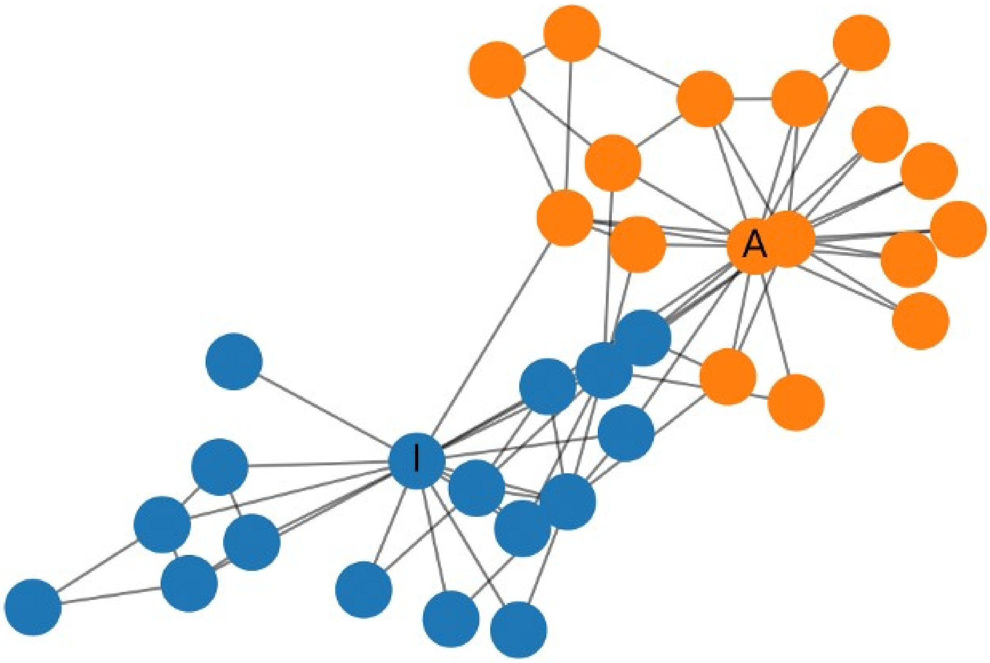
Voila! A complete hidden layer with adjacency matrix, input features, weights and activation function!

Back to Reality

Now, finally, we can apply a graph convolutional network on a real graph. I will show you how to produce the feature representations we saw early in the post.

Zachary's Karate Club

Zachary's karate club is a commonly used social network where nodes represent members of a karate club and the edges their mutual relations. While Zachary was studying the karate club, a conflict arose between the administrator and the instructor which resulted in the club splitting in two. The figure below shows the graph representation of the network and nodes are labeled according to which part of the club. The administrator and instructor are marked with 'A' and 'I', respectively.



Zachary's Karate Club

Building the GCN

Now let us build the graph convolutional network. We won't actually train the network, but simply initialize it at random to produce the feature representations we saw at the start of this post. We will use `networkx` which has a graph representation of the club easily available, and compute the `A_hat` and `D_hat` matrices.

```
from networkx import karate_club_graph, to_numpy_matrix

zkc = karate_club_graph()
order = sorted(list(zkc.nodes()))

A = to_numpy_matrix(zkc, nodelist=order)
I = np.eye(zkc.number_of_nodes())

A_hat = A + I
D_hat = np.array(np.sum(A_hat, axis=0))[0]
D_hat = np.matrix(np.diag(D_hat))
```

Next, we'll initialize weights randomly.

```
W_1 = np.random.normal(
    loc=0, scale=1, size=(zkc.number_of_nodes(), 4))
W_2 = np.random.normal(
    loc=0, size=(W_1.shape[1], 2))
```

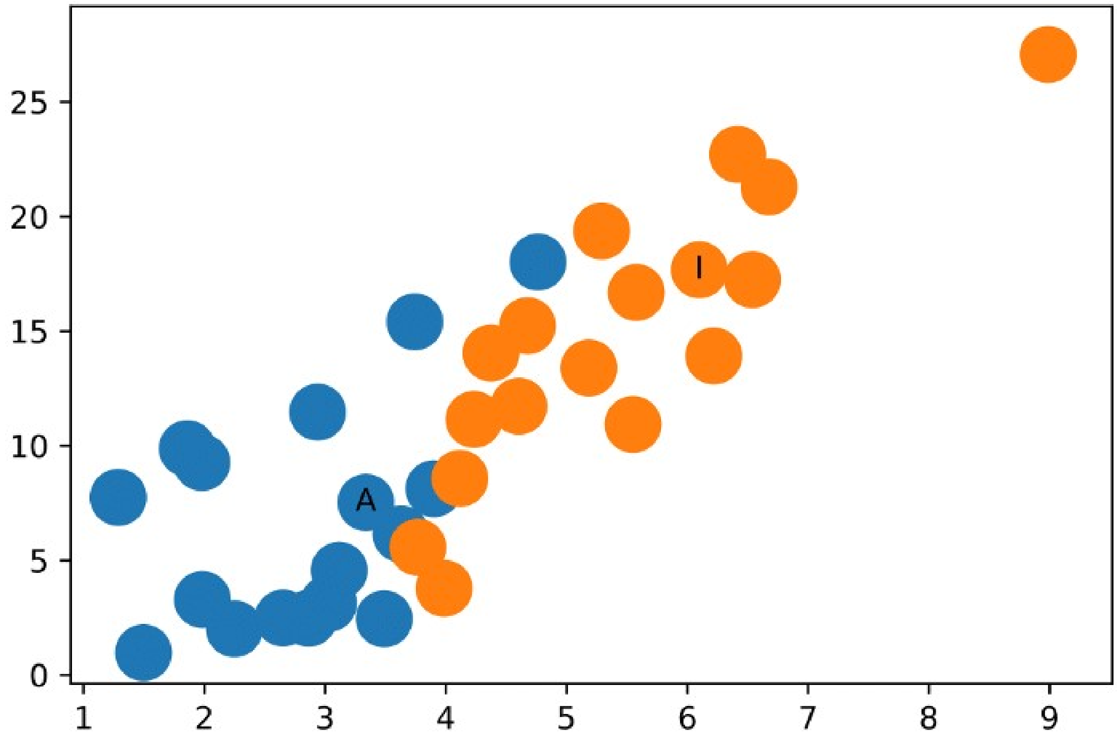
Stack the GCN layers. We here use just the identity matrix as feature representation, that is, each node is represented as a one-hot encoded categorical variable.

```
def gcn_layer(A_hat, D_hat, X, W):
    return relu(D_hat**-1 * A_hat * X * W)

H_1 = gcn_layer(A_hat, D_hat, I, W_1)
H_2 = gcn_layer(A_hat, D_hat, H_1, W_2)

output = H_2
```

We extract the feature representations.

```
feature_representations = {
    node: np.array(output)[node]
    for node in zkc.nodes()}
```

And voila! Feature representations that separate the communities in Zachary's karate club quite well. And we haven't even begun training yet!



Feature Representations of the Nodes in Zachary's Karate Club

I should note that for this example the randomly initialized weights were very likely to give 0 values on either the x- or the y-axis as result of the ReLU function, so it took a few random initializations to produce the figure above.

Conclusion

In this post I have given a high-level introduction to graph convolutional networks and illustrated how the feature representation of a node at each layer in the GCN is based on an aggregate of the its neighborhood. We saw how we can build these networks using numpy and how powerful they are: even randomly initialized GCNs can separate communities in Zachary's Karate Club.

In the next post, I will go a bit more into technical detail and show how to implement and train a recently published GCN using semi-supervised learning. **You find the next post in the series here.**

Liked what you read? Consider following me on Twitter where I share papers, videos, and articles related to the practice, theory, and ethics of data science and machine learning that I find interesting in addition to my own posts.

For professional inquiries, please contact me on LinkedIn or by direct message on Twitter.

References

[1] Blog post on graph convolutional networks by Thomas Kipf.

[2] Paper called *Semi-Supervised Classification with Graph Convolutional Networks* by Thomas Kipf and Max Welling.

Machine Learning   Data Science   Neural Networks   Deep Learning

About   Help   Legal