GRAPH NEURAL NETWORKS

June 15, 2020

1 GRAPH NEURAL NETWORKS

1.1 Summary:

Keypoints

- Understanding graph data.
- Inputs and output of the GCN
- How information propagated through the hidden layers of a GCN.
- How the GCN aggregates information from the previous layers.
- How this mechanism produces useful feature representations of nodes in graphs.

1.2 Introduction to GCN and semi-supervised learning with GCN

- GCN can produce useful feature representations of nodes in networks.
- GCN is a neural network that operates on graphs. Given a graph $G = (V, E)$.

- **Input features** could be stored in a matrix (number_of_nodes lines and number_of_features columns). lets call it feature matrix X, N be the number of nodes and F is the number of input features for each node
- **Graph structure** could be stored in an N Œ N matrix such as the adjacency matrix A

- A hidden layer in the GCN can thus be written as $H = f(H, A))$ where $H = X$ and f is a propagation
- Each layer H corresponds to an NF 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

1.3 Propagation rule

- Lets W is the weight matrix for layer i and is a non-linear activation(ReLu), The weight matrix has dimensions FF
- Propagation rule: $f(H, A) = (AHW)$
- The second dimension of the weight matrix determines the number of features at the next layer

1.4 Simple experimentation with the above graph

```
In [1]: #Imports
        import numpy as np
```

Adjacency matrix

```
In [2]: A = np.matrix([
            [0, 1, 0, 0],
            [0, 0, 1, 1],
            [0, 1, 0, 0],
            [1, 0, 1, 0]])
        print('The adjacency matrix of the given graph is: \n\n',A)
```

```
The adjacency matrix of the given graph is:

 [[0 1 0 0]
 [0 0 1 1]
 [0 1 0 0]
 [1 0 1 0]]
```

Features of the nodes

```
In [3]: # lets generate two features for each node
        X = np.array([[i, -i]for i in range(A.shape[0])])
        print('The feature matrix :\n\n',X)
```

```
The feature matrix :

 [[ 0  0]
 [ 1 -1]
 [ 2 -2]
 [ 3 -3]]
```

Applying propagation rule lets assume that W and activation are unit and one layer: $AHW = AXW = AX$

Therefore: $f(X, A) = AX$

```
In [4]: A.dot(X)
```

```
Out[4]: matrix([[ 1, -1],
                [ 5, -5],
                [ 1, -1],
                [ 2, -2]])
```

==> In the above result shows that the features of each node became the summation of neibhouring features, however **does not include the node it self**.

==> Nodes with large degrees would have large values in their feature representation while nodes with small degrees will have small values. This could cause vanishing or exploding gradients in backprop.

Self loop This is done by adding the identity matrix I to the adjacency matrix A before applying the propagation rule.

```
In [5]: I = np.matrix(np.eye(A.shape[0]))
        print('The identity matrix:\n\n',I)

The identity matrix:

 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]


In [6]: A_ = A + I
        A_.dot(X)

Out[6]: matrix([[ 1., -1.],
                [ 6., -6.],
                [ 3., -3.],
                [ 5., -5.]])
```

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

$f(X, A) = DAX$

```
In [7]: D = np.array(np.sum(A, axis=0))[0]
        #print(D)
        D = np.matrix(np.diag(D))
        D

Out[7]: matrix([[1, 0, 0, 0],
                [0, 2, 0, 0],
                [0, 0, 2, 0],
                [0, 0, 0, 1]])

In [8]: D**-1 * A

Out[8]: matrix([[0. , 1. , 0. , 0. ],
                [0. , 0. , 0.5, 0.5],
                [0. , 0.5, 0. , 0. ],
                [1. , 0. , 1. , 0. ]])

In [9]: ((np.linalg.inv(D)).dot(A)).dot(X)

Out[9]: matrix([[ 1. , -1. ],
                [ 2.5, -2.5],
                [ 0.5, -0.5],
                [ 2. , -2. ]])
```

Lets add the weight matrix

```
In [10]: W = np.matrix([[1, -1],
                        [-1, 1] ])
         W

Out[10]: matrix([[ 1, -1],
                 [-1,  1]])

In [11]: D**-1 * A_*D**-1 * X * W

Out[11]: matrix([[ 1. , -1. ],
                 [ 4.5, -4.5],
                 [ 1.5, -1.5],
                 [ 8. , -8. ]])
```

Activation Function

```
In [12]: def relu(x):
             k = x > 0
             return np.multiply(k,x)

In [13]: relu(D**-1 * A_*D**-1 * X * W)

Out[13]: matrix([[ 1. , -0. ],
                 [ 4.5, -0. ],
                 [ 1.5, -0. ],
                 [ 8. , -0. ]])
```

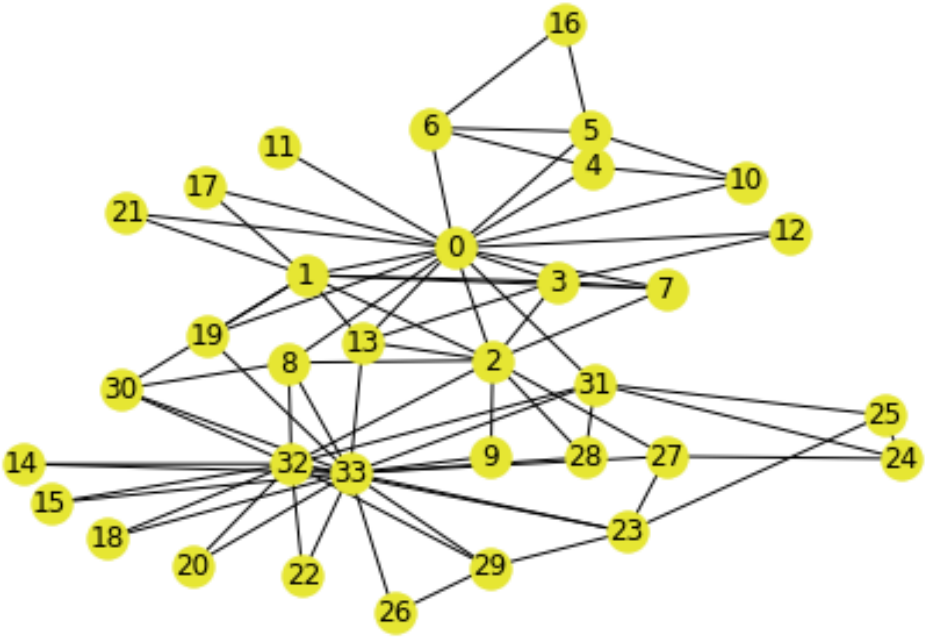
1.5 Karate club Example

```
In [14]: import dgl
         import torch
         import numpy as np
         import pickle

         import matplotlib.pyplot as plt
         import networkx as nx

         import torch.nn as nn
         import torch.nn.functional as F

Using backend: pytorch


In [15]: node_count = 34

In [16]: with open('edge_list.pickle', 'rb') as f:
             edge_list = pickle.load(f)
         print(edge_list,'\n\n We have ',len(edge_list),' Edges')
```

```
[('0', '8'), ('1', '17'), ('24', '31'), ('13', '33'), ('0', '1'), ('2', '8'), ('0', '19'), ('25
```

```
 We have  78  Edges
```

```
In [17]: def build_karate_club_graph(edges):
             g = dgl.DGLGraph()
             g.add_nodes(node_count)
             src, dst = tuple(zip(*edges))
             g.add_edges(src, dst)
             # edges are directional in DGL; make them bidirectional
             g.add_edges(dst, src)
             return g
         G = build_karate_club_graph(edge_list)
         G

Out[17]: DGLGraph(num_nodes=34, num_edges=156,
                  ndata_schemes={}
                  edata_schemes={})

In [18]: print('We have %d nodes.' % G.number_of_nodes())
         print('We have %d edges.' % G.number_of_edges())

We have 34 nodes.
We have 156 edges.

In [19]: # Since the actual graph is undirected, we convert it for visualization purpose.
         nx_G = G.to_networkx().to_undirected()
         # Kamada-Kawaii layout usually looks pretty for arbitrary graphs
         pos = nx.kamada_kawai_layout(nx_G)
         nx.draw(nx_G, pos, with_labels=True, node_color=[[.9, .9, .2]])

/usr/lib/python3/dist-packages/networkx/drawing/nx_pylab.py:611: MatplotlibDeprecationWarning:
  if cb.is_numlike(alpha):
```




```
In [20]: inputs = torch.eye(node_count)
         labeled_nodes = torch.tensor([0, 33])  # only the instructor and the president nodes
         labels = torch.tensor([0,1])  # their labels are different

In [21]: # Define the message and reduce function
         # NOTE: We ignore the GCN's normalization constant c_ij for this tutorial.
         def gcn_message(edges):
             # The argument is a batch of edges.
             # This computes a (batch of) message called 'msg' using the source node's feature
             return {'msg' : edges.src['h']}

         def gcn_reduce(nodes):
             # The argument is a batch of nodes.
             # This computes the new 'h' features by summing received 'msg' in each node's mai
             return {'h' : torch.sum(nodes.mailbox['msg'], dim=1)}

         # Define the GCNLayer module
         class GCNLayer(nn.Module):
             def __init__(self, in_feats, out_feats):
                 super(GCNLayer, self).__init__()
                 self.linear = nn.Linear(in_feats, out_feats)

             def forward(self, g, inputs):
                 # g is the graph and the inputs is the input node features
```

```
                # first set the node features
                g.ndata['h'] = inputs
                # trigger message passing on all edges
                g.send(g.edges(), gcn_message)
                # trigger aggregation at all nodes
                g.recv(g.nodes(), gcn_reduce)
                # get the result node features
                h = g.ndata.pop('h')
                # perform linear transformation
                return self.linear(h)
```

```
In [22]: class GCN(nn.Module):
             def __init__(self, in_feats, hidden_size, num_classes):
                 super(GCN, self).__init__()
                 self.gcn1 = GCNLayer(in_feats, hidden_size)
                 self.gcn2 = GCNLayer(hidden_size, num_classes)
                 self.softmax = nn.Softmax()

             def forward(self, g, inputs):
                 h = self.gcn1(g, inputs)
                 h = torch.relu(h)
                 h = self.gcn2(g, h)
                 h = self.softmax(h)
                 return h
```

```
In [23]: net = GCN(34, 5, 2)
         print(net)
```

```
GCN(
  (gcn1): GCNLayer(
    (linear): Linear(in_features=34, out_features=5, bias=True)
  )
  (gcn2): GCNLayer(
    (linear): Linear(in_features=5, out_features=2, bias=True)
  )
  (softmax): Softmax(dim=None)
)
```

```
In [24]: optimizer = torch.optim.Adam(net.parameters(), lr=0.1)
         all_preds = []
         epochs = 20

         for epoch in range(epochs):
             preds = net(G, inputs)
             all_preds.append(preds)
             # we only compute loss for labeled nodes
             loss = F.cross_entropy(preds[labeled_nodes], labels)
```

```
            optimizer.zero_grad() # PyTorch accumulates gradients by default
            loss.backward()
            optimizer.step()

            print('Epoch %d | Loss: %.4f' % (epoch, loss.item()))
```

```
Epoch 0 | Loss: 0.8805
Epoch 1 | Loss: 0.4259
Epoch 2 | Loss: 0.3162
Epoch 3 | Loss: 0.3133
Epoch 4 | Loss: 0.3133
Epoch 5 | Loss: 0.3133
Epoch 6 | Loss: 0.3133
Epoch 7 | Loss: 0.3133
Epoch 8 | Loss: 0.3133
Epoch 9 | Loss: 0.3133
Epoch 10 | Loss: 0.3133
Epoch 11 | Loss: 0.3133
Epoch 12 | Loss: 0.3133
Epoch 13 | Loss: 0.3133
Epoch 14 | Loss: 0.3133
Epoch 15 | Loss: 0.3133
Epoch 16 | Loss: 0.3133
Epoch 17 | Loss: 0.3133
Epoch 18 | Loss: 0.3133
Epoch 19 | Loss: 0.3133
```

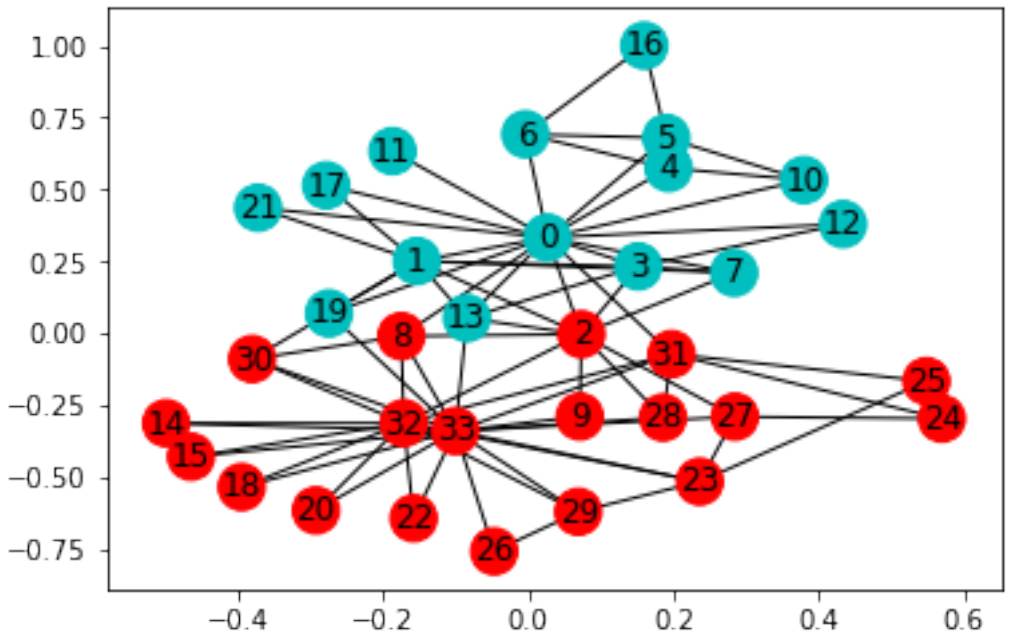
```
/usr/lib/python3/dist-packages/ipykernel_launcher.py:12: UserWarning: Implicit dimension choice
  if sys.path[0] == '':
```

```
In [25]: last_epoch = all_preds[epochs-1].detach().numpy()
         predicted_class = np.argmax(last_epoch, axis=-1)
         color = np.where(predicted_class==0, 'c', 'r')

         print(predicted_class)
         print(color)
```

```
[0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1]
['c' 'c' 'r' 'c' 'c' 'c' 'c' 'c' 'r' 'r' 'c' 'c' 'c' 'c' 'r' 'r' 'c' 'c'
 'r' 'c' 'r' 'c' 'r' 'r' 'r' 'r' 'r' 'r' 'r' 'r' 'r' 'r' 'r' 'r']
```

```
In [26]: nx.draw_networkx(nx_G, pos, node_color=color, with_labels=True, node_size=300)
```

```
/usr/lib/python3/dist-packages/networkx/drawing/nx_pylab.py:611: MatplotlibDeprecationWarning:
  if cb.is_numlike(alpha):
```




```
In [27]: inputs = torch.eye(node_count)
         labeled_nodes = torch.tensor([1,5,27, 32])  # only the instructor and the president n
         labels = torch.tensor([0,1,2,3])  # their labels are different
         #labeled_nodes = torch.tensor([21,16,20, 24])
```

```
In [28]: net2 = GCN(34, 10, 4)
         print(net2)
```

```
GCN(
  (gcn1): GCNLayer(
    (linear): Linear(in_features=34, out_features=10, bias=True)
  )
  (gcn2): GCNLayer(
    (linear): Linear(in_features=10, out_features=4, bias=True)
  )
  (softmax): Softmax(dim=None)
)
```

```
In [29]: optimizer = torch.optim.Adam(net2.parameters(), lr=0.1)
         all_preds = []
         epochs = 30

         for epoch in range(epochs):
             preds = net2(G, inputs)
             all_preds.append(preds)
```

```
            # we only compute loss for labeled nodes
            #print(labels,preds[labeled_nodes])
            loss = F.cross_entropy(preds[labeled_nodes], labels)
            optimizer.zero_grad() # PyTorch accumulates gradients by default
            loss.backward()
            optimizer.step()

            print('Epoch %d | Loss: %.4f' % (epoch, loss.item()))
```

```
/usr/lib/python3/dist-packages/ipykernel_launcher.py:12: UserWarning: Implicit dimension choic
  if sys.path[0] == '':
```

```
Epoch 0 | Loss: 1.3650
Epoch 1 | Loss: 1.2303
Epoch 2 | Loss: 1.1733
Epoch 3 | Loss: 1.0842
Epoch 4 | Loss: 0.8916
Epoch 5 | Loss: 0.8181
Epoch 6 | Loss: 0.8392
Epoch 7 | Loss: 0.8259
Epoch 8 | Loss: 0.7672
Epoch 9 | Loss: 0.7495
Epoch 10 | Loss: 0.7457
Epoch 11 | Loss: 0.7444
Epoch 12 | Loss: 0.7440
Epoch 13 | Loss: 0.7438
Epoch 14 | Loss: 0.7437
Epoch 15 | Loss: 0.7437
Epoch 16 | Loss: 0.7437
Epoch 17 | Loss: 0.7437
Epoch 18 | Loss: 0.7437
Epoch 19 | Loss: 0.7437
Epoch 20 | Loss: 0.7437
Epoch 21 | Loss: 0.7437
Epoch 22 | Loss: 0.7437
Epoch 23 | Loss: 0.7437
Epoch 24 | Loss: 0.7437
Epoch 25 | Loss: 0.7437
Epoch 26 | Loss: 0.7437
Epoch 27 | Loss: 0.7437
Epoch 28 | Loss: 0.7437
Epoch 29 | Loss: 0.7437
```

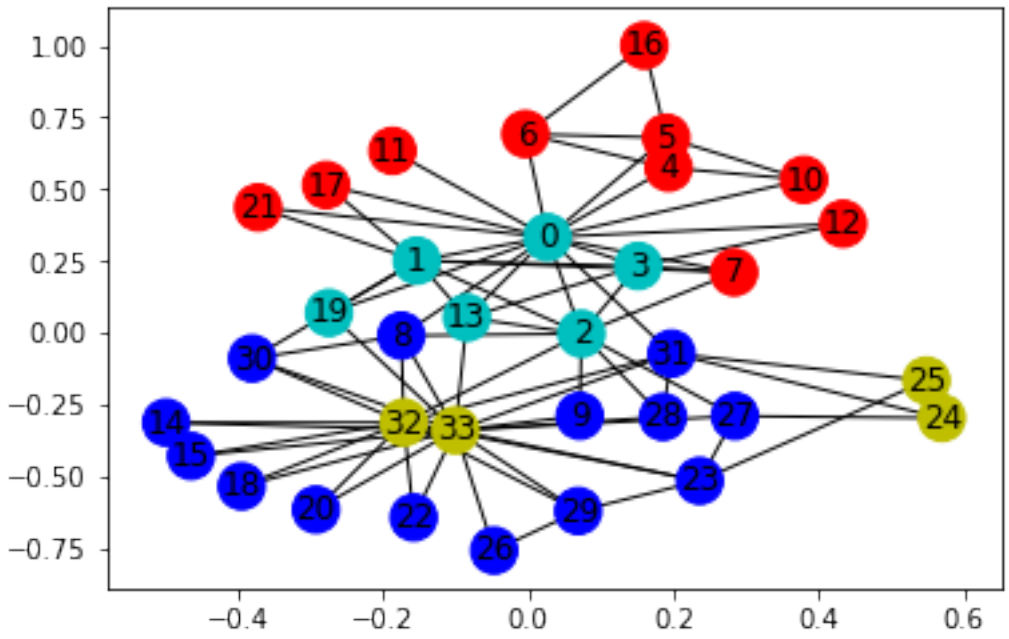
```
In [30]: last_epoch = all_preds[epochs-1].detach().numpy()
         predicted_class = np.argmax(last_epoch, axis=-1)
         colors = ['c','r','b','y']
```

```
        color = []
        for indx,class_ in enumerate(predicted_class):
            color.append(colors[class_])
        print(predicted_class)
        print(color)
```

```
[0 0 0 0 1 1 1 1 2 2 1 1 1 0 2 2 1 1 2 0 2 1 2 2 3 3 2 2 2 2 2 2 3 3]
['c', 'c', 'c', 'c', 'r', 'r', 'r', 'r', 'b', 'b', 'r', 'r', 'r', 'c', 'b', 'b', 'r', 'r', 'b'
```

In [31]:
```
nx.draw_networkx(nx_G, pos, node_color=color, with_labels=True, node_size=300)
```

```
/usr/lib/python3/dist-packages/networkx/drawing/nx_pylab.py:611: MatplotlibDeprecationWarning:
  if cb.is_numlike(alpha):
```



In [111]:
```
net.state_dict
```

Out[111]:
```
<bound method Module.state_dict of GCN(
  (gcn1): GCNLayer(
    (linear): Linear(in_features=34, out_features=10, bias=True)
  )
  (gcn2): GCNLayer(
    (linear): Linear(in_features=10, out_features=4, bias=True)
  )
  (softmax): Softmax(dim=None)
)>
```

The number of parameters will be equal : n_nodes * dim_output_features

In [ ]: