Tutorial problem description

Problem Description:

We have seen a case study on the applications of Graph Convolutional Neural Network (GCN) in Smart Grids domain. Here we will try to solve a similar problem on a smaller smart grid network. We have IEEE 14 bus system as test case. In IEEE 14 bus system there are 14 bus (Nodes) and 20 transmission lines (edges). Our goal is to represent IEEE 14 bus system as a graph (structured data) and apply GCN to detect and locate line failure. Below is the online diagram of IEEE 14 bus system...

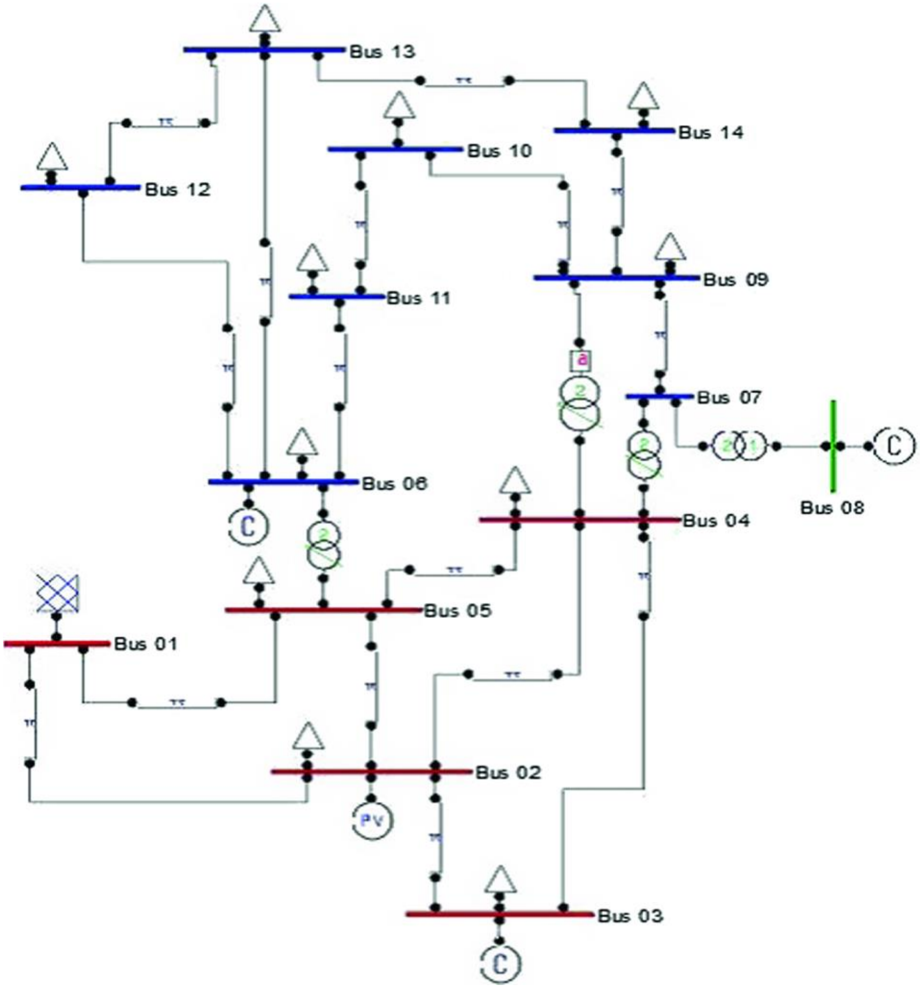


Fig: IEEE 14 Bus system

Dataset Description:

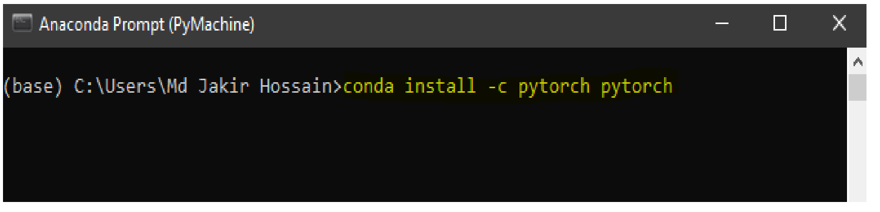
Our data set has 6 features per node (Voltage Magnitude, Voltage Angle, Real Power Demand, Reactive Power Demand, Maximum Voltage, Minimum Voltage) and 2 labels (1 = failed 0 = normal). And we have results from 20 different simulations. Thus, the shape of the feature matrix is [Num_Node, Num_Feat, Num_Simu] and the shape of the labels is [Num_node, Num_Simu]

Instructions to Install Required Tools

We will use python and machine learning libraries on Jupyter Notebook. If you have [Anaconda] installed in your system than you already have [Python] and [Jupyter Notebook] installed, which you can access from Anaconda Navigator.

Step 1: Open Anaconda prompt from your start menu, you will see a window like --------→

Step 2: Installing machine learning libraries, to install the library copy the code and paste it to your anaconda prompt and press enter. ---------→



- **To install PyTorch copy:** `conda install -c pytorch pytorch`
- **To install NetworkX copy:** `conda install -c anaconda networkx`
- **To install DGL copy:** `conda install -c dglteam dgl`

The rest of the libraries should be included in your anaconda package by default. In case, some libraries are not installed than follow the same procedure

Some general notes:
- Wait till the installation is finished and do not close the prompt window.
- Install another after current installation is finished.

If you want to open Jupyter Notebook from any folder on your hard drive other than default folder. Copy this code in anaconda prompt

```
jupyter notebook --notebook-dir=D:/
```

Directory to your desired folder

Step 1: Load necessary packages

we will use "Deep Graph Library(DGL)" to create graph convolutional layer. And "NetworkX" package to plot graphs

```python
import dgl
import numpy as np
import pandas as pd
import networkx as nx
from scipy.io import loadmat
import itertools
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

We used “Deep Graph Library” for Graph Convolution Layer

Networkx for Graph visualization

PyTorch in the Backend

Step 2: Create graph from list of edges

```python
src = np.array([1, 1, 2, 2, 2, 3, 4, 4, 4, 5, 6, 6, 6, 7, 7, 9, 9, 10, 12, 13])-1 # List of source nodes
dst = np.array([2, 5, 3, 4, 5, 4, 5, 7, 9, 6, 11, 12, 13, 8, 9, 10, 14, 11, 13, 14])-1 #List of destination nodes

# Edges are directional in DGL; Make them bi-directional.
u = np.concatenate([src, dst])
v = np.concatenate([dst, src])
#G = dgl.DGLGraph((u, v))
G = dgl.graph((u, v))
G = dgl.add_self_loop(G)
print('We have %d nodes.' % G.number_of_nodes())
print('We have %d edges.' % G.number_of_edges())
```

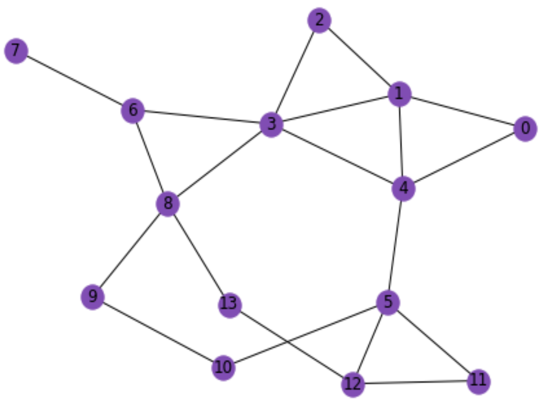
Source and destination node list from figure 1

Creating Graph structure in DGL

```
We have 14 nodes.
We have 54 edges.
```

Step 3: Lets see how the graph looks like

```python
# 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=[[.5, .3, .7]])
```



Step 4: Lets load our dataset

```python
NodeFeature = loadmat('NodeFeatures.mat')              # Loading feature matrix from external files
X = NodeFeature['NodeFeatures'].transpose((1, 2, 0))   # Extracting data values from dictionary object
NodeClass = loadmat('NodeLabels.mat')                  # Loading labels from external files
Y = NodeClass['NodeLabels'].transpose((1,0))           # Extracting data values from dictionary object
print(X.shape,Y.shape)        # print shape of feature and label matrix
n_simulation = X.shape[2]     # Number of simulation in dataset
n_features = X.shape[1]       # total number of features per node
n_node = X.shape[0]           # total number of nodes
n_classes = np.max(Y) + 1     # total number of classes
User_defined = 14             # hidden parameter input to the 2nd convolution layer
```

```
(14, 6, 20) (14, 20)
```

Step 5: Add node features and labels to the graph

```python
G.ndata['feat'] = torch.LongTensor(X)                  # creating the graph data
G.ndata['label'] = torch.LongTensor(Y)                 # creating the graph data
print(G.ndata['feat'][:,:,2],G.ndata['label'][:,2])    # print out input features of all node at simulation 2
```

```
tensor([[  1,   0,   0,   0,   1,   0],
        [  1,  -4,  21,  12,   1,   0],
        [  1, -24,  94,  19,   1,   0],
        [  1, -13,  43,  -3,   1,   0],
        [  1, -10,   7,   1,   1,   0],
        [  1, -15,  11,   7,   1,   0],
        [  1, -16,   0,   0,   1,   0],
        [  1, -16,   0,   0,   1,   0],
        [  1, -18,  27,  15,   1,   0],
        [  1, -18,   8,   5,   1,   0],
        [  1, -18,   3,   1,   1,   0],
        [  1, -16,   6,   1,   1,   0],
        [  1, -16,  12,   5,   1,   0],
        [  1, -18,  13,   4,   1,   0]]) tensor([0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0])
```

Creating graph data by adding node features and labels

Number of features, number of nodes,
number of classes,
and hidden parameter respectively

Step 6: Define a Graph Convolutional Network (GCN)

```
from dgl.nn.pytorch import GraphConv

# creating a 2/3-layer graph convolutional network model
class GCN(nn.Module):
    def __init__(self, in_feats, hidden_size, num_classes):
        super(GCN, self).__init__()
        self.conv1 = GraphConv(in_feats, hidden_size)
        self.conv2 = GraphConv(hidden_size, hidden_size)   # you can add an extra convolution layer amd see the effect
        self.fc1 = nn.Linear(hidden_size,hidden_size)      # First fully connected layer
        self.conv3 = GraphConv(hidden_size, num_classes)   # Output convolution layer

    def forward(self, g, inputs):
        h = self.conv1(g, inputs)
        h = torch.relu(h)
        h = self.conv2(g, h)  # comment/uncomment to add/remove additional Conv layer
        h = torch.relu(h)     # comment/uncomment to add/remove additional Conv layer
        h = self.fc1(h)       # comment/uncomment to add/remove a fully connected layer
        h = torch.relu(h)     # comment/uncomment to add/remove a fully connected layer
        h = self.conv3(g, h)
        return h

# The first layer transforms input features of size of "Num_feat" to a hidden size of "User_define".
# The second layer transforms the hidden layer "User_defined" to size of "Num_classes"
net = GCN(n_features,User_defined,n_classes)
print(net)
```

```
GCN(
  (conv1): GraphConv(in=6, out=14, normalization=both, activation=None)
  (conv2): GraphConv(in=14, out=14, normalization=both, activation=None)
  (fc1): Linear(in_features=14, out_features=14, bias=True)
  (conv3): GraphConv(in=14, out=2, normalization=both, activation=None)
)
```

Defining how data will be processing through each layer

Definition of layers in the network

Model Summery

Step 7: Train-Test split

```
n_train = 15                   # number of training samples Maximum 20 since number of simulation is 20
n_test = 20 - n_train          # number of test samples
Train_feat = X[:,:,:n_train]   # training set
Test_feat = X[:,:,n_train:]    # test set
Train_labels = Y[:,:n_train]   # training label
Test_labels = Y[:,n_train:]    # testing label
inputs = torch.LongTensor(X)   # Initialize features
labels = torch.LongTensor(Y)   # Initialize labels
```

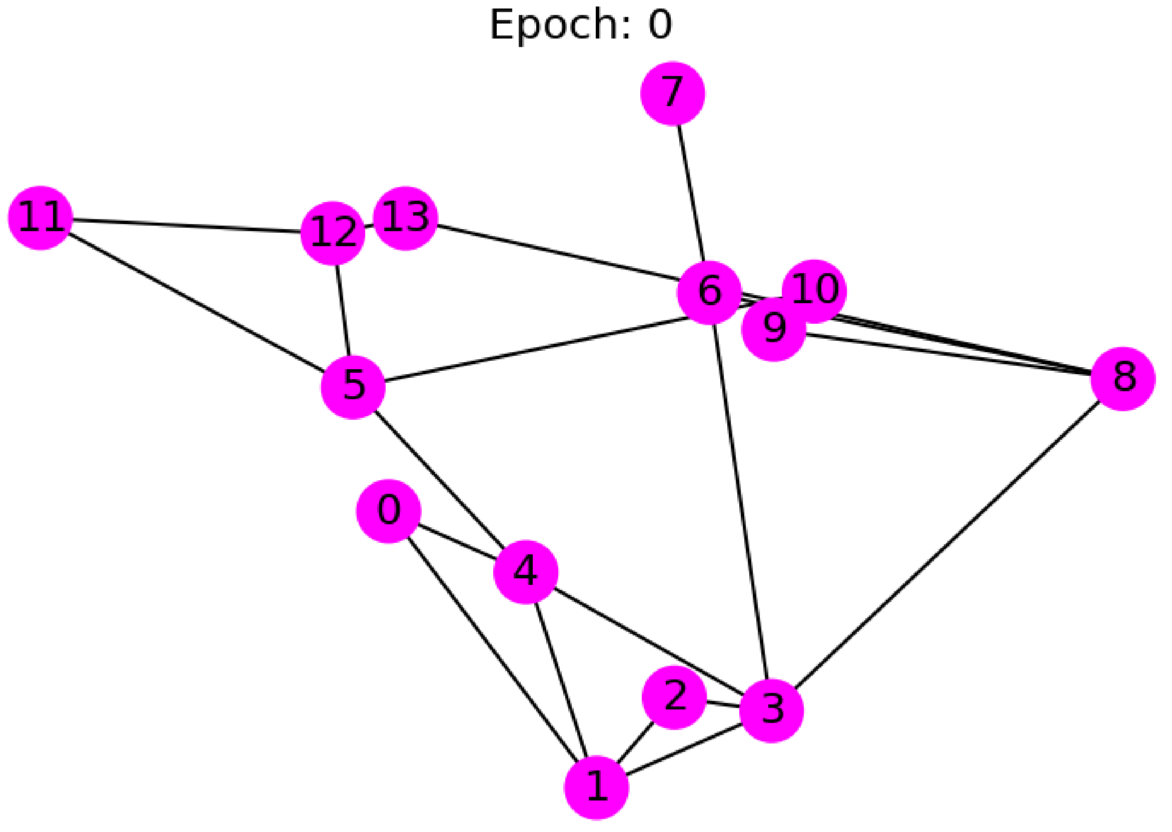
Step 8: Setting up the training loop

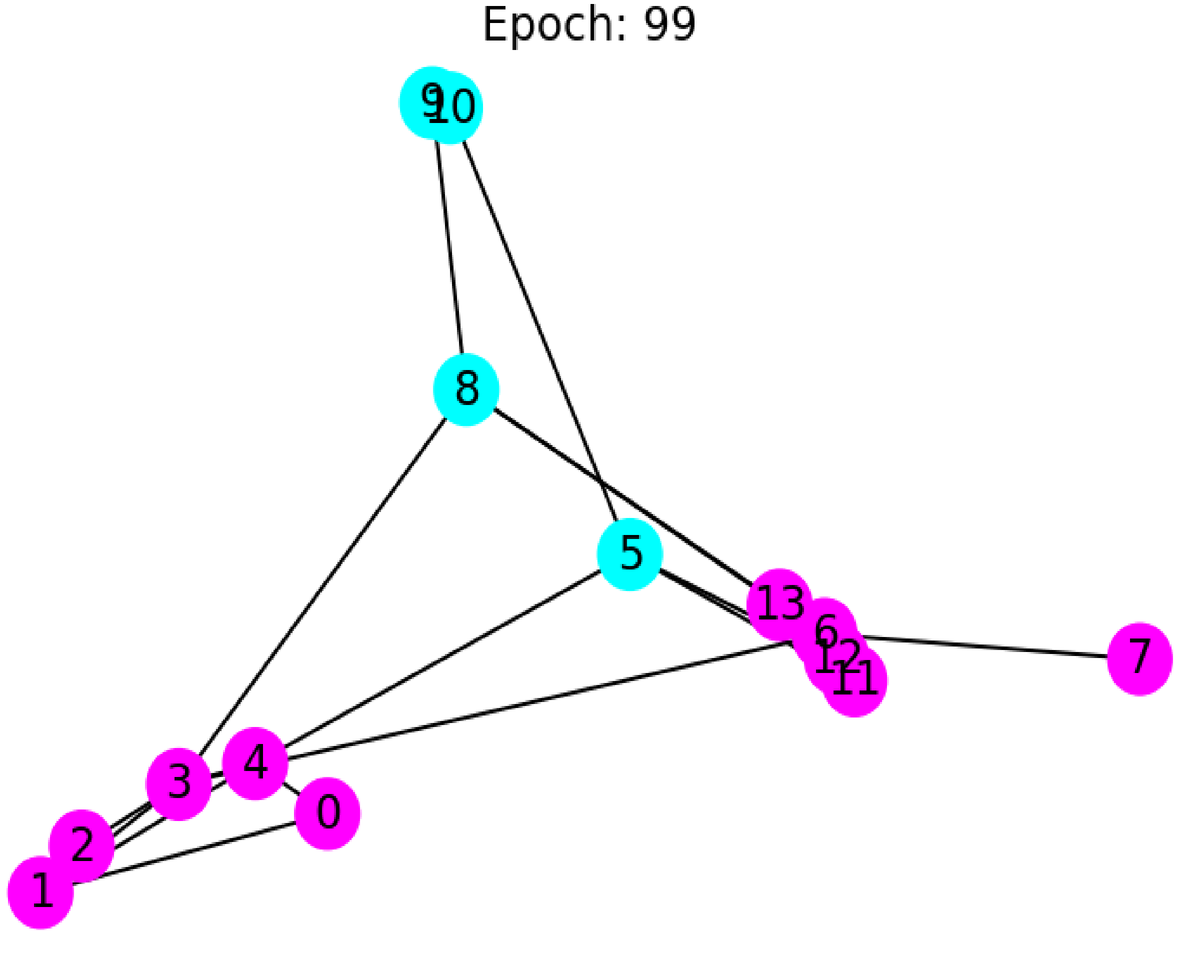
```
# training the model
import torch.optim as optim
learning_rate = 0.001                                    # change learning rate and see the effect on accuracy

optimizer = optim.Adam(net.parameters(), lr= learning_rate)  # We are using ADAM optimizer
all_logits = []
n_epoch = 1000                                           # change number of epoch and see the effect on accuracy

for i in range(n_train):
    for epoch in range(n_epoch):
        logits = net(G, inputs[:,:,i])
        # we save the logits for visualization later
        all_logits.append(logits.detach())
        logp = F.log_softmax(logits, 1)
        # Let's compute the model loss
        loss = F.nll_loss(logp, labels[:,i])

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    #if epoch > n_epoch-2:
    print('Epoch %d | Loss: %.4f' % (epoch, loss.item()))  # printing only the last iteration of every epoch
```

Step 9: Visulaizing the iterations

```
# this function will help us draw the result of each epoch
def draw(i):
    cls1color = '#00FFFF'
    cls2color = '#FF00FF'
    pos = {}
    colors = []
    for v in range(14):
        pos[v] = all_logits[i][v].numpy()
        cls = pos[v].argmax()
        colors.append(cls1color if cls else cls2color)
    ax.cla()
    ax.axis('off')
    ax.set_title('Epoch: %d' % i)
    nx.draw_networkx(nx_G.to_undirected(), pos, node_color=colors,
            with_labels=True, node_size=300, ax=ax)

# lets draw the first epoch
fig = plt.figure(dpi=150)
fig.clf()
ax = fig.subplots()
draw(0)  # draw the prediction of the first epoch
plt.show()
```

```
# lets draw the 100th epoch
fig = plt.figure(dpi=150)
fig.clf()
ax = fig.subplots()
draw(99)  # draw the prediction of the 100th epoch
plt.show()
```


Epoch: 0
7
11
12
13
6
10
9
8
5
0
4
2
3
1


Epoch: 99
9
10
8
5
13
6
12
11
7
4
3
0
2
1

Step 10: Testing model performance

```
p = np.random.randint(n_train, n_simulation)                       # randomly selecting a case from test set
logits = net(G, torch.LongTensor(X[:,:,p]))                        # save output from the model
labels = torch.LongTensor(Y[:,p])                                  # etract true labels

idx_test  = torch.LongTensor([0,1,2,3,4,5,6,7,8,9,10,11,12,13]) # generating node indexes

lbl2idx = {k:v for v,k in enumerate(sorted(np.unique(labels)))} # given node labels find index
idx2lbl = {v:k for k,v in lbl2idx.items()}                         # given idexes find label

df = pd.DataFrame({'Real': [idx2lbl[e] for e in labels.tolist()],
                   'Pred': [idx2lbl[e] for e in logits.argmax(1).tolist()]})
df['Comp'] = np.where(df['Real']==df['Pred'], np.ones(14),np.zeros(14))
Accuracy = df['Comp'].sum()/n_node*100                             # Claculating classification accuracy
print('Selected Simulation = %d | Accuracy = (Number of Correctly Predicted Labels/ Total Node) =  %4f ' % (p, Accuracy))
print(df)
```

```
Selected Simulation = 17 | Accuracy = (Number of Correctly Predicted Labels/ Total Node) =  71.428571
    Real  Pred  Comp
0      1     1   1.0
1      1     0   0.0
2      0     0   1.0
3      0     0   1.0
4      1     1   1.0
5      0     1   0.0
6      0     0   1.0
7      0     0   1.0
8      0     0   1.0
9      0     0   1.0
10     0     1   0.0
11     1     1   1.0
12     1     0   0.0
13     0     0   1.0
```

3 out of 3 Fault Located

Miss classification

Note: Model training accuracy and model performance accuracy is entirely different concept. Model training accuracy is how well the model is fit to data. Model performance accuracy is depending on our definition of accuracy and what we expect to get from the model. Here the performance is moderate. This depends on various factors (Number of data sample, How good is data for the desired task, model hidden parameters, etc.)