Message Passing Graph Neural Networks

We develop a simplified version of orginal GNN models that were proposed by Merkwirth and Lengauer [2005] and Scarselli et al. [2009]. We develop custom MPGNN layers and FFNN layers so that students can understand what is happening under the hood in GNNs. We use the standard model training pipeline (inlcuding optimization) since those were covered in detail in early lectures. For this purpose we use Cora dataset, which is a citation dataset and the task is a node classification. Deep graph Library (DGL) is only used to load the dataset. NetworkX is used for visualizations.

Since PyTroch framework is used for model builting students should be familiar with the model bulding pipeline in that framework. Refer this PyTorch tutorial to get a basic idea about it. Students should read about the dataset before running this code to get an idea about the data as well. Refer the lecture slides [specifically pages 35 -52] to understand the MPGNN model and the equations. Students are also encouraged to experiment with this code.

```
!pip install torch==2.3.0 torchvision==0.18.0 torchaudio==2.3.0 --index-url https://download.pytorch.org/whl/cu121
```

```
Looking in indexes: https://download.pytorch.org/whl/cu121
Requirement already satisfied: torch==2.3.0 in /usr/local/lib/python3.10/dist-packages (2.3.0+cu121)
Requirement already satisfied: torchvision==0.18.0 in /usr/local/lib/python3.10/dist-packages (0.18.0+cu121)
Requirement already satisfied: torchaudio==2.3.0 in /usr/local/lib/python3.10/dist-packages (2.3.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (3.16.0)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (4.12.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (1.13.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (2024.6.1)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (12.1.105
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (12.1.1
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (12.1.105
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (8.9.2.26)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (10.3.2.106
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (11.4.5.1
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (12.1.0.1
Requirement already satisfied: nvidia-nccl-cu12==2.20.5 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (2.20.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (12.1.105)
Requirement already satisfied: triton==2.3.0 in /usr/local/lib/python3.10/dist-packages (from torch==2.3.0) (2.3.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchvision==0.18.0) (1.26.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.10/dist-packages (from torchvision==0.18.0) (10.4.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.10/dist-packages (from nvidia-cusolver-cu12==11.4.5.107->
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch==2.3.0) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy->torch==2.3.0) (1.3.0)
```

```
!pip install  dgl -f https://data.dgl.ai/wheels/torch-2.3/cu121/repo.html
```

```
Looking in links: https://data.dgl.ai/wheels/torch-2.3/cu121/repo.html
Requirement already satisfied: dgl in /usr/local/lib/python3.10/dist-packages (2.4.0+cu121)
Requirement already satisfied: networkx>=2.1 in /usr/local/lib/python3.10/dist-packages (from dgl) (3.3)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.10/dist-packages (from dgl) (1.26.4)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from dgl) (24.1)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from dgl) (2.1.4)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.10/dist-packages (from dgl) (5.9.5)
Requirement already satisfied: pydantic>=2.0 in /usr/local/lib/python3.10/dist-packages (from dgl) (2.9.2)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages (from dgl) (6.0.2)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist-packages (from dgl) (2.32.3)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from dgl) (1.13.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from dgl) (4.66.5)
Requirement already satisfied: torch<=2.4.0 in /usr/local/lib/python3.10/dist-packages (from dgl) (2.3.0+cu121)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.10/dist-packages (from pydantic>=2.0->dgl) (0.7.0)
Requirement already satisfied: pydantic-core==2.23.4 in /usr/local/lib/python3.10/dist-packages (from pydantic>=2.0->dgl) (2.23.4)
Requirement already satisfied: typing-extensions>=4.6.1 in /usr/local/lib/python3.10/dist-packages (from pydantic>=2.0->dgl) (4.12.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->dgl) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->dgl) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->dgl) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->dgl) (2024.8.30)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (3.16.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (1.13.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (2024.6.1)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (12.
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (1
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (12.
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (8.9.2.26
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (12.1.3.
```

```
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (11.0.2.
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (10.3.
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (11.
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (12.
Requirement already satisfied: nvidia-nccl-cu12==2.20.5 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (2.20.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (12.1.105)
Requirement already satisfied: triton==2.3.0 in /usr/local/lib/python3.10/dist-packages (from torch<=2.4.0->dgl) (2.3.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.10/dist-packages (from nvidia-cusolver-cu12==11.4.5.107->
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas->dgl) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->dgl) (2024.2)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas->dgl) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas->dgl) (1.16.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch<=2.4.0->dgl) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy->torch<=2.4.0->dgl) (1.3.0)
```

```
import dgl
import numpy as np
import torch
import pdb
import torch.nn.init as init
import matplotlib.pyplot as plt
import time
from sklearn.metrics import confusion_matrix
import networkx as nx
```

```
torch.__version__
```

```
'2.3.0+cu121'
```

```
torch.Tensor(2,2)
```

```
tensor([[0., 0.],
        [0., 0.]])
```

```
# Load the Cora dataset
dataset = dgl.data.CoraGraphDataset()

# Get the graph, features, and labels
g = dataset[0]  # The graph itself
features = g.ndata['feat']  # Node features
labels = g.ndata['label']  # Node labels

print(f'Number of categories: {dataset.num_classes}')
print(f'Node features shape: {features.shape}')
print(f'Label shape: {labels.shape}')
print(f'object type: {type(g)}')
```

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
Number of categories: 7
Node features shape: torch.Size([2708, 1433])
Label shape: torch.Size([2708])
object type: <class 'dgl.heterograph.DGLGraph'>
```

```
g.edges()[1]
```

```
tensor([ 633, 1862, 2582,  ...,  598, 1473, 2706])
```

```
type(dataset)
features.shape
new_arr_0 = features[0,:].numpy()[np.where(features[0,:].numpy()!=0)]
print(new_arr_0)
```

```
[0.11111111 0.11111111 0.11111111 0.11111111 0.11111111 0.11111111
 0.11111111 0.11111111 0.11111111]
```

Double-click (or enter) to edit

```
# get data split
train_mask = g.ndata['train_mask']
val_mask = g.ndata['val_mask']
test_mask = g.ndata['test_mask']
```

```
val_mask
```

```
tensor([False, False, False,  ..., False, False, False])
```

```
val_mask.numpy()[val_mask.numpy()==True].shape
```

```
(500,)
```

```
val_mask
```

```
tensor([False, False, False,  ..., False, False, False])
```

```
print(type(features))
print(labels.unsqueeze(-1).shape)
print(labels.unsqueeze(-1).dtype)
print(torch.ones(1433).unsqueeze(0).dtype)
labels.unsqueeze(-1)[:8]
```

```
<class 'torch.Tensor'>
torch.Size([2708, 1])
torch.int64
torch.float32
tensor([[3],
        [4],
        [4],
        [0],
        [3],
        [2],
        [0],
        [3]])
```

```
#create a features tensor - Not used in the main code atm
struc_features = torch.matmul((torch.ones(2708)*0.1).unsqueeze(-1), torch.ones(1433).unsqueeze(0))
print(struc_features.shape)
print(struc_features[:8,:])
```

```
torch.Size([2708, 1433])
tensor([[0.1000, 0.1000, 0.1000,  ..., 0.1000, 0.1000, 0.1000],
        [0.1000, 0.1000, 0.1000,  ..., 0.1000, 0.1000, 0.1000],
        [0.1000, 0.1000, 0.1000,  ..., 0.1000, 0.1000, 0.1000],
        ...,
        [0.1000, 0.1000, 0.1000,  ..., 0.1000, 0.1000, 0.1000],
        [0.1000, 0.1000, 0.1000,  ..., 0.1000, 0.1000, 0.1000],
        [0.1000, 0.1000, 0.1000,  ..., 0.1000, 0.1000, 0.1000]])
```

```
#create a features tensor - Not used in the main code atm
struc_features_1 = torch.matmul((labels*0.1).unsqueeze(-1).to(dtype=torch.float32), torch.ones(1433).unsqueeze(0))
print(struc_features_1.shape)
print(struc_features_1[:8,:])
```

```
torch.Size([2708, 1433])
tensor([[0.3000, 0.3000, 0.3000,  ..., 0.3000, 0.3000, 0.3000],
        [0.4000, 0.4000, 0.4000,  ..., 0.4000, 0.4000, 0.4000],
        [0.4000, 0.4000, 0.4000,  ..., 0.4000, 0.4000, 0.4000],
        ...,
        [0.2000, 0.2000, 0.2000,  ..., 0.2000, 0.2000, 0.2000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.3000, 0.3000, 0.3000,  ..., 0.3000, 0.3000, 0.3000]])
```

```
g.edges() # original edges are in an edge list as set of directed edges
```

```
(tensor([   0,    0,    0,  ..., 2707, 2707, 2707]),
 tensor([ 633, 1862, 2582,  ...,  598, 1473, 2706]))
```

```
# create the adjacency matrix
num_nodes = len(g.nodes())
print(num_nodes)
adj = np.zeros((num_nodes, num_nodes))
print(adj.shape)
```

```
for i in range(len(g.edges()[0])):
  # print(i)
  row = g.edges()[0][i]
  col = g.edges()[1][i]

  adj[row, col] = 1
  adj[col, row] = 1

adj_torch = torch.from_numpy(adj)# use in model development
adj_torch = adj_torch.to(dtype=torch.float32)
print(adj_torch.shape)
print(adj_torch.dtype)
print((adj.T - adj).sum()) # symmetric as it should be
```

```
2708
(2708, 2708)
torch.Size([2708, 2708])
torch.float32
0.0
```

```
# Get the adjacency matrix in dense format
adj_matrix_dense = g.adjacency_matrix().to_dense()

# Get the adjacency matrix in sparse format
adj_matrix_sparse = g.adjacency_matrix()

print("Dense adjacency matrix:\n", adj_matrix_dense)
print("Dense adjacency matrix:\n", adj_matrix_dense.shape)
print("Sparse adjacency matrix:\n", adj_matrix_sparse)
print("Sparse adjacency matrix:\n", adj_matrix_sparse.shape)
```

```
Dense adjacency matrix:
 tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 1.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 1.],
        [0., 0., 0.,  ..., 0., 1., 0.]])
Dense adjacency matrix:
 torch.Size([2708, 2708])
Sparse adjacency matrix:
 SparseMatrix(indices=tensor([[   0,    0,    0,  ..., 2707, 2707, 2707],
                              [ 633, 1862, 2582,  ...,  598, 1473, 2706]]),
             values=tensor([1., 1., 1.,  ..., 1., 1., 1.]),
             shape=(2708, 2708), nnz=10556)
Sparse adjacency matrix:
 (2708, 2708)
```

```
# undirected graph it seems
print(np.abs(adj).sum())
print(adj.sum())
# no self loops
print(np.trace(np.abs(adj)))
```

```
10556.0
10556.0
0.0
```

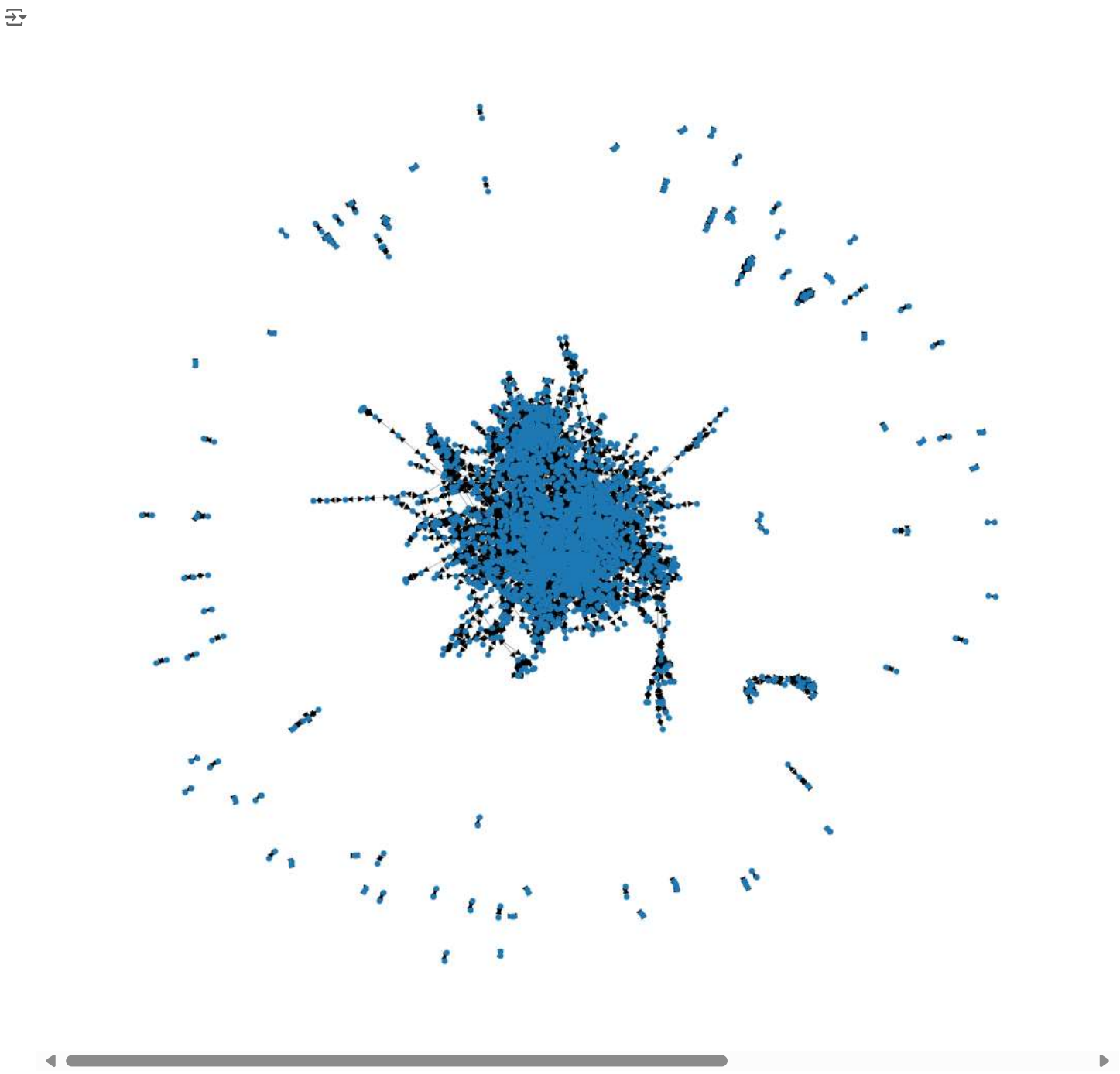
```
print(features[train_mask].shape)
print(labels[train_mask].shape)
labels[train_mask]
tmp = list(features[train_mask][10].numpy())
my_list = [x for x in tmp if x != 0]
print(my_list) # thus feature matrix is normalized row wise
print(len(my_list))
```

```
torch.Size([140, 1433])
torch.Size([140])
[0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.05882353, 0.0
17
```

Visualize the CORA dataset

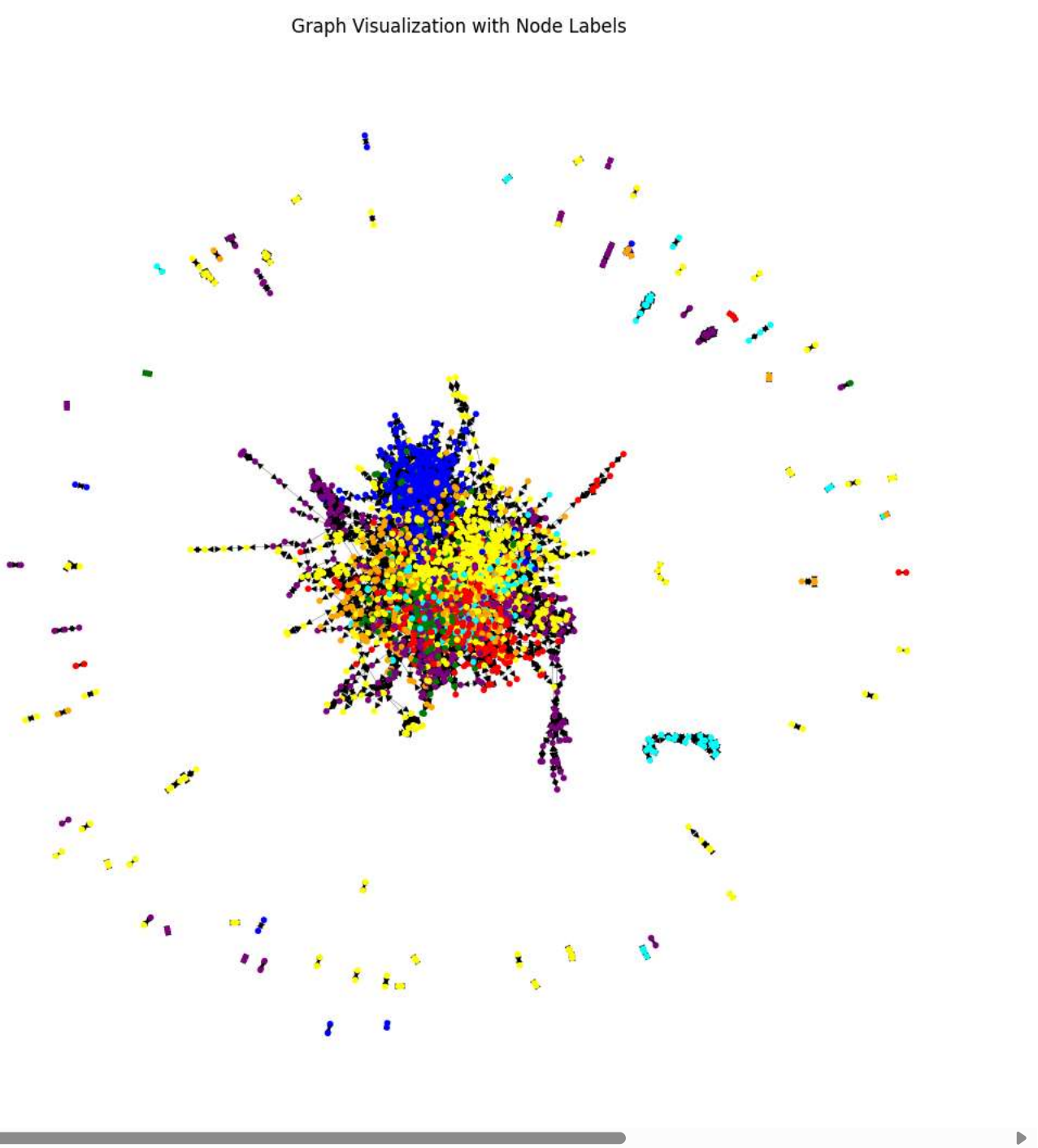
```
#visualize the entire dataset
```

```
#visualize the entire dataset

# Convert the DGL graph to a NetworkX graph
nx_g = g.to_networkx()

plt.figure(figsize=(10, 10))
pos = nx.spring_layout(nx_g, seed=42)  # Use spring layout
nx.draw(nx_g, pos, node_size=10, width=0.1, with_labels=False)
plt.show()
```

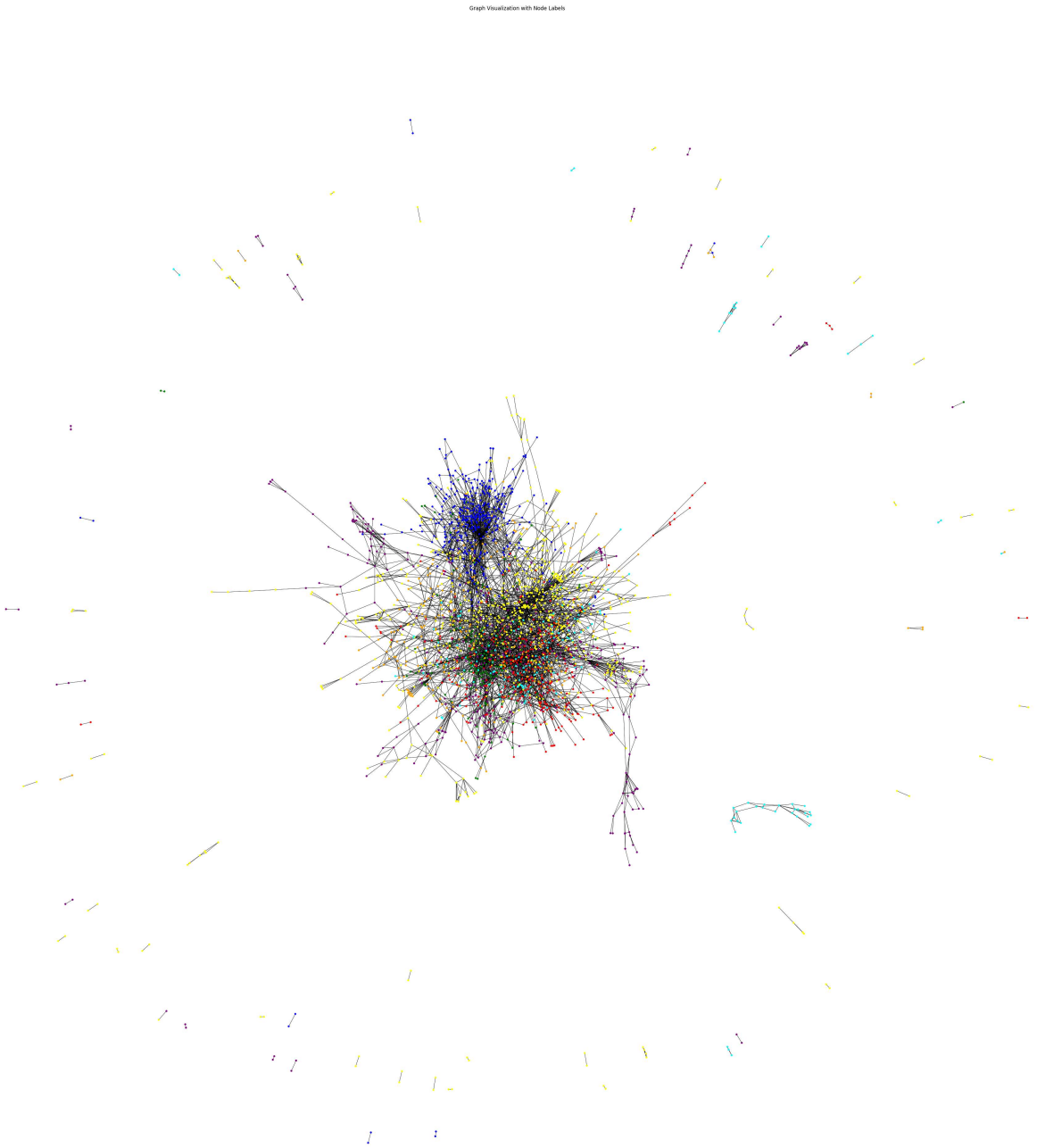


```
#visualize the dataset with labels - each node belong to a class out of 7 classes so 7 colours are used
color_map = [
    'red', 'green', 'blue', 'yellow', 'purple', 'orange', 'cyan'
]

colors = [color_map[label] for label in labels]
nx_g = g.to_networkx()

plt.figure(figsize=(10, 10))
pos = nx.spring_layout(nx_g, seed=42)
nx.draw(nx_g, pos, node_color=colors, node_size=10, width=0.1, with_labels=False)
plt.title("Graph Visualization with Node Labels")
```

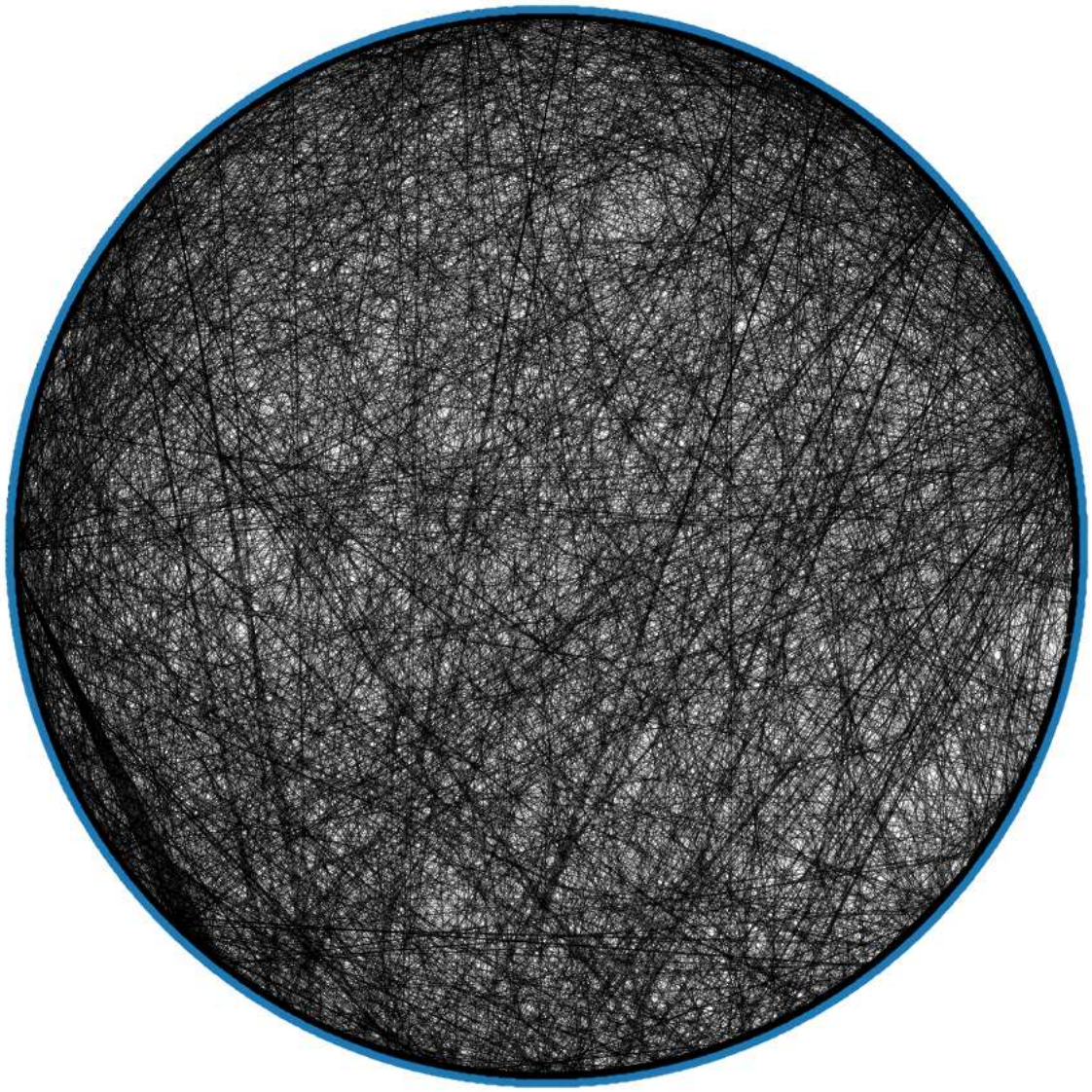
```
plt.title("Graph Visualization with Node Labels")
plt.show()
```



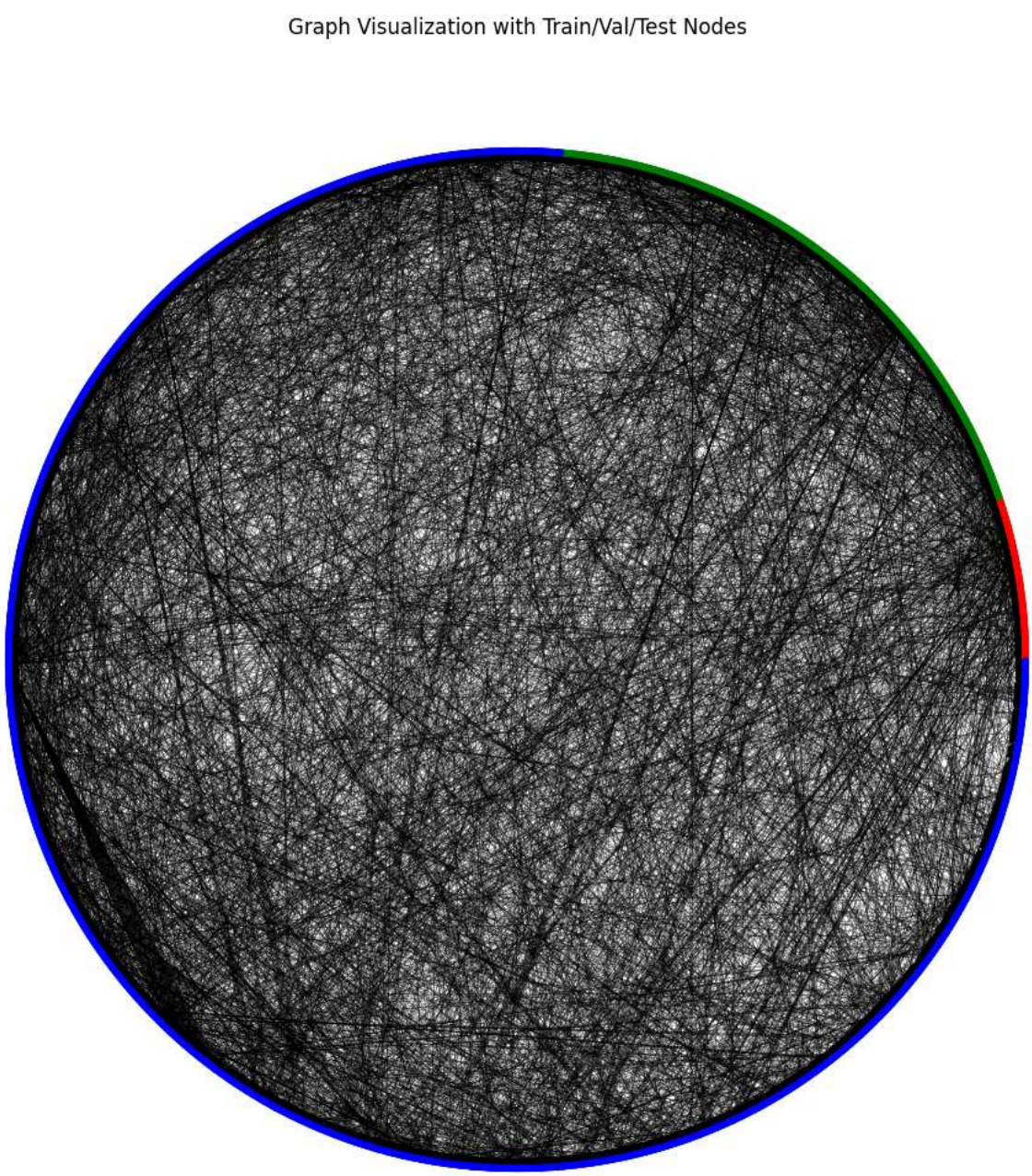

```
#same as before but more detailed - open the image in a new tab for higher resolution
plt.figure(figsize=(40, 40))
nx.draw(nx_g, pos, node_color=colors, node_size=12, width=0.4, with_labels=False, arrows=False)
plt.title("Graph Visualization with Node Labels")
plt.show()
```




```
# visualize using circular layout - less time to visualize
plt.figure(figsize=(10, 10))
pos = nx.circular_layout(nx_g)
nx.draw(nx_g, pos, node_size=10, width=0.1, with_labels=False)
plt.show()
```



```
# visualize graph with each train/ validation / test split using circular layout

# train set in red, validation in green and test is in blue
colors = np.where(train_mask, 'red', np.where(val_mask, 'green', 'blue'))
plt.figure(figsize=(10, 10))
nx.draw(nx_g, pos, node_color=colors, node_size=10, width=0.1, with_labels=False)
plt.title("Graph Visualization with Train/Val/Test Nodes")
plt.show()
```



MPGNN Architecture

Refer the equations of the MPGNN in the lecture slides.

```
# Individual MPGNN layer is built from scratch
class MPGNN_Layer(torch.nn.Module):

    def __init__(self, input_embed_size =-1, output_embed_size = -1, bias_ = True):
        super(MPGNN_Layer, self).__init__()


        self.input_size = input_embed_size
        self.output_size = output_embed_size
        self.bias_ = bias_

        self.adj_M = adj_torch
        # print(self.adj_M.dtype)
```

```
            self.relu = torch.nn.ReLU()
            self.ones_v = torch.ones(num_nodes).unsqueeze(-1)

            self.weights_self = torch.nn.Parameter(torch.Tensor(input_embed_size , output_embed_size))
            self.weights_neigh = torch.nn.Parameter(torch.Tensor(input_embed_size , output_embed_size))

            self.initialize_parameters()

            if self.bias_ :
              self.bias = torch.nn.Parameter(torch.zeros(1, output_embed_size))


        def forward(self, x): # x should be embedding matrix of graph, shape -> (num_nodes, embed_length)

            num_nodes = x.shape[0]

            #msg passing
            msg_neigh = torch.matmul(x ,self.weights_neigh)
            # aggregation
            aggreg_neigh = torch.matmul(self.adj_M, msg_neigh)
            # update
            tmp_emb = aggreg_neigh +  torch.matmul(x ,self.weights_self)

            # pdb.set_trace()
            # non-linear activation and bias adding
            if self.bias_ :
              self.bias_matrix = torch.matmul(self.ones_v , self.bias)
              linear_out = tmp_emb + self.bias_matrix
              output_emb = self.relu(linear_out)
            else:
              output_emb = self.relu(tmp_emb)

            return output_emb

        def initialize_parameters(self):
            # Xavier initialization for weights
            init.xavier_uniform_(self.weights_self)
            init.xavier_uniform_(self.weights_neigh)



    #FFNN layer is built from scratch
    class FC_scratch(torch.nn.Module):
        def __init__(self, input_feat_num =-1,  output_feat_num=-1, activation = None):
            super(FC_scratch, self).__init__()

            self.act = activation
            self.ones_v = torch.ones(num_nodes).unsqueeze(-1)

            self.weights_FC = torch.nn.Parameter(torch.Tensor(input_feat_num , output_feat_num))
            self.bias = torch.nn.Parameter(torch.Tensor(1, output_feat_num))

            self.initialize_parameters()

        def forward(self, x):

            bias_matrix = torch.matmul(self.ones_v , self.bias)
            x = torch.matmul(x , self.weights_FC) + bias_matrix

            if self.act != None:
              return self.act(x)
            else:
              return x # needed  when we only need linear outputs

        def initialize_parameters(self):
            init.xavier_uniform_(self.weights_FC)
            init.xavier_uniform_(self.bias)


    # MPGNN Complete model is built from scratch
    class MPGNN_Model(torch.nn.Module):

        def __init__(self, input_feat_num =1433,  num_classes=7):
            super(MPGNN_Model, self).__init__()

            self.softmax = torch.nn.Softmax()

            self.MPGNN_1 = MPGNN_Layer(input_feat_num, 512)
```

```
        self.MPGNN_2 = MPGNN_Layer(512, 64)
        # self.fc = torch.nn.Linear(64, num_classes)
        self.fc = FC_scratch(64 , num_classes)



    def forward(self, x): # x should be feature matrix of graph, shape -> (num_nodes, feature_length)

        hid_1 = self.MPGNN_1(x)
        hid_2 = self.MPGNN_2(hid_1)

        #classification
        tmp_out = self.fc(hid_2)
        final_out = self.softmax(tmp_out)

        return final_out



#create an instance of the model and optimizer + loss function; currently not using a lr schedular
model = MPGNN_Model()
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)


# Model parameters are calculated here
trainable_params = []
non_trainable_params = []

for name, param in model.named_parameters():
    if param.requires_grad:
        print(f"Trainable parameter: {name} - size: {param.size()}")
        trainable_params.append(param)
    else:
        print(f"Non-trainable parameter: {name} - size: {param.size()}")
        non_trainable_params.append(param)

# Calculate the total number of trainable and non-trainable parameters
num_trainable_params = sum(p.numel() for p in trainable_params) # numel()  returns the total number of elements in a tensor
num_non_trainable_params = sum(p.numel() for p in non_trainable_params)

print(f"Number of trainable parameters: {num_trainable_params}")
print(f"Number of non-trainable parameters: {num_non_trainable_params}")
```

```
Trainable parameter: MPGNN_1.weights_self - size: torch.Size([1433, 512])
Trainable parameter: MPGNN_1.weights_neigh - size: torch.Size([1433, 512])
Trainable parameter: MPGNN_1.bias - size: torch.Size([1, 512])
Trainable parameter: MPGNN_2.weights_self - size: torch.Size([512, 64])
Trainable parameter: MPGNN_2.weights_neigh - size: torch.Size([512, 64])
Trainable parameter: MPGNN_2.bias - size: torch.Size([1, 64])
Trainable parameter: fc.weights_FC - size: torch.Size([64, 7])
Trainable parameter: fc.bias - size: torch.Size([1, 7])
Number of trainable parameters: 1533959
Number of non-trainable parameters: 0
```

```
print(features.shape)
print(labels.shape)
print(labels[train_mask].shape[0])
```

```
torch.Size([2708, 1433])
torch.Size([2708])
140
```

```
# train step function is defined here - since full batch is used in forward pass, this step is actually epoch wise rather than batch wise

running_loss =[]
running_acc = []

def train_one_epoch(epoch_index, features, labels, tb_writer = None):

    inputs = features
    labels = labels


    # Zero the gradients at each start epoch
    optimizer.zero_grad()
```

```
    # Make predictions for the entire graph nodes
    outputs = model(inputs)

    # Compute the loss for training subgraph
    outputs_train = outputs[train_mask]
    labels_train = labels[train_mask]

    loss = loss_fn(outputs_train, labels_train)

    # backpropagation and update weights
    loss.backward()
    optimizer.step()

    # Gather data and print
    total = labels_train.shape[0]
    loss_per_node = loss
    print(f'Train Loss: {loss_per_node}')
    running_loss.append(loss_per_node)


    max_prob , max_prob_index = torch.max(outputs_train, dim=1)
    if epoch_index % 100 == 0:
      print('max_prob_index: ', max_prob_index)
      print('max_prob_values: ', max_prob)

    # Compare predictions to ground truth
    correct = (max_prob_index == labels_train).sum().item()
    # Calculate accuracy
    accuracy = correct / total
    print(f'Train Accuracy: {accuracy * 100:.2f}%')
    running_acc.append(accuracy)

    return loss_per_node, accuracy
```

Standard Model Training

```
# set the input feature matrix for the training / validation / test

# features = struc_features #use this if structural information is only used
features = features # use this if {feature + strutural} informationn is use


# model training is done here

EPOCHS = 500
val_running_loss = []
val_running_acc = []

start_time = time.time()

for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch + 1))

    model.train(True)
    avg_loss, avg_acc = train_one_epoch(epoch, features, labels, None) # training


    # Set the model to evaluation mode, disabling dropout etc.
    model.eval()

    # validation of the model on val dataset
    with torch.no_grad():

      outputs = model(features) #validation

      outputs_val = outputs[val_mask]
      labels_val = labels[val_mask]
      val_loss = loss_fn(outputs_val, labels_val)

      # Gather data and print
      total = labels_val.shape[0]
      loss_per_node = val_loss
      print(f'Val Loss: {loss_per_node}')
      val_running_loss.append(loss_per_node)
```

```
        max_prob , max_prob_index = torch.max(outputs_val, dim=1)
        if epoch % 100 == 0:
          print('max_prob_index: ', max_prob_index)
          print('max_prob_values: ', max_prob)

        # Compare predictions to ground truth
        correct = (max_prob_index == labels_val).sum().item()
        # Calculate accuracy
        accuracy = correct / total
        print(f'Val Accuracy: {accuracy * 100:.2f}%')
        val_running_acc.append(accuracy)

    end_time = time.time()
    time_taken = end_time - start_time
    print(f"Time taken to run the code: {time_taken:.6f} seconds")
```

```
EPOCH 1:
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/module.py:1532: UserWarning: Implicit dimension choice for softmax has been dep
  return self._call_impl(*args, **kwargs)
Train Loss: 1.9524096250534058
max_prob_index:  tensor([0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        5, 5, 5, 0, 0, 5, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0,
        5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 5,
        0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 5, 5, 0, 0, 0, 0, 0,
        0, 0, 0, 5, 0, 5, 0, 5, 0, 0, 0, 0, 0, 0, 5, 0, 5, 5, 0, 0, 0, 0, 0, 5, 0,
        0, 0, 0, 0, 5, 0, 5, 0, 0, 0, 5, 0, 0, 5, 0, 0, 0, 0, 5, 5])
max_prob_values:  tensor([0.2708, 0.2637, 0.3019, 0.2656, 0.2780, 0.2732, 0.2825, 0.2612, 0.2759,
        0.2804, 0.2639, 0.2668, 0.2827, 0.3327, 0.2769, 0.2650, 0.2833, 0.2789,
        0.2688, 0.2671, 0.2787, 0.2614, 0.2973, 0.2700, 0.3829, 0.2616, 0.2665,
        0.2651, 0.2710, 0.2779, 0.3091, 0.2663, 0.2610, 0.2781, 0.3131, 0.2814,
        0.2744, 0.2679, 0.2745, 0.2940, 0.2903, 0.2885, 0.2709, 0.2682, 0.2709,
        0.3058, 0.2741, 0.2724, 0.2820, 0.2685, 0.2883, 0.2802, 0.2870, 0.3164,
        0.2685, 0.2789, 0.2753, 0.2724, 0.2709, 0.3229, 0.2657, 0.2784, 0.2680,
        0.2663, 0.2761, 0.2976, 0.2664, 0.2675, 0.3382, 0.2966, 0.2736, 0.2883,
        0.3147, 0.3328, 0.2843, 0.2943, 0.2714, 0.2680, 0.2702, 0.2708, 0.2701,
        0.2658, 0.2640, 0.2706, 0.2864, 0.2742, 0.2715, 0.2643, 0.3212, 0.2887,
        0.3267, 0.2632, 0.2673, 0.2693, 0.2988, 0.3586, 0.2667, 0.2780, 0.2651,
        0.2644, 0.2715, 0.3145, 0.2906, 0.3133, 0.2771, 0.2682, 0.2669, 0.2724,
        0.2758, 0.3503, 0.2732, 0.3108, 0.3038, 0.2790, 0.2757, 0.2720, 0.2701,
        0.2751, 0.2923, 0.2801, 0.2838, 0.2817, 0.2613, 0.2633, 0.2930, 0.2677,
        0.3291, 0.2702, 0.2738, 0.2732, 0.2660, 0.2655, 0.2757, 0.2939, 0.2705,
        0.2702, 0.2747, 0.2865, 0.2873, 0.3463], grad_fn=<MaxBackward0>)
Train Accuracy: 14.29%
Val Loss: 1.9461935758590698
max_prob_index:  tensor([0, 5, 0, 5, 0, 0, 5, 5, 0, 0, 0, 5, 0, 5, 0, 5, 5, 5, 5, 0, 5, 0, 5, 0,
        0, 5, 0, 0, 0, 0, 5, 0, 5, 5, 5, 0, 0, 0, 0, 5, 0, 5, 0, 0, 0, 0, 5, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 5, 0, 5, 0, 5, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 5, 0, 5, 0, 0, 5, 5, 0, 0, 5,
        5, 0, 0, 0, 0, 0, 0, 0, 5, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0,
        0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 5, 0, 0, 5, 0, 5, 0, 0, 0, 5, 0,
        5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 5, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 5, 0, 0,
        0, 5, 0, 5, 0, 5, 5, 0, 0, 5, 0, 5, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 5, 0, 0, 0, 0, 5, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0,
        5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0,
        5, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 5, 0,
        0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 5, 0, 5, 0, 0,
        5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5,
        0, 0, 0, 5, 0, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 5, 0, 0, 0, 0, 0,
        0, 5, 0, 0, 0, 0, 0, 0, 0, 5, 5, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 0, 0,
        0, 0, 5, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 5, 0, 0, 0, 5, 0, 0, 5, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 5, 0])
max_prob_values:  tensor([0.2641, 0.2675, 0.2982, 0.3780, 0.2777, 0.2787, 0.2632, 0.2615, 0.2701,
        0.2803, 0.2849, 0.3092, 0.2831, 0.3154, 0.3174, 0.3299, 0.3289, 0.3720,
        0.3134, 0.2785, 0.3249, 0.2666, 0.2727, 0.2686, 0.2686, 0.3076, 0.2662,
        0.2811, 0.2708, 0.2667, 0.3120, 0.2727, 0.2648, 0.3240, 0.3192, 0.2842,
        0.2801, 0.2688, 0.2666, 0.2955, 0.2922, 0.2642, 0.2762, 0.2805, 0.2694,
        0.2633, 0.2598, 0.2679, 0.2741, 0.2770, 0.2638, 0.2731, 0.2687, 0.2760,
        0.2910, 0.2745, 0.2698, 0.2657, 0.2656, 0.2671, 0.2714, 0.4311, 0.2670,
        0.2781, 0.2638, 0.3260, 0.2714, 0.2678, 0.2727, 0.2641, 0.2949, 0.2702,
        0.2651, 0.2763, 0.2651, 0.3963, 0.3417, 0.2738, 0.2758, 0.2908, 0.2689,
        0.2660, 0.2674, 0.2760, 0.2811, 0.2685, 0.3176, 0.2698, 0.3210, 0.2757,
        0.2825, 0.3475, 0.3860, 0.2728, 0.2671, 0.3157, 0.3527, 0.2714, 0.2652,
        0.2950, 0.2744, 0.2896, 0.2711, 0.2722, 0.3366, 0.2711, 0.3170, 0.2669,
        0.2784, 0.2805, 0.2676, 0.3012, 0.2811, 0.2880, 0.2829, 0.2805, 0.2693,
        0.2735, 0.2844, 0.2728, 0.2691, 0.3278, 0.2770, 0.2979, 0.2776, 0.2880,
        0.2737, 0.2677, 0.2730, 0.2780, 0.2824, 0.2842, 0.2658, 0.2662, 0.2802,
        0.2715, 0.2799, 0.2759, 0.2616, 0.2805, 0.2741, 0.2697, 0.2812, 0.2651,
        0.2973, 0.2604, 0.2650, 0.2704, 0.2639, 0.3214, 0.2852, 0.2680, 0.2692,
        0.2687, 0.2868, 0.2647, 0.2787, 0.3165, 0.2733, 0.3035, 0.2800, 0.2791,
        0.3034, 0.2808, 0.2609, 0.2733, 0.3864, 0.2734, 0.3286, 0.2702, 0.2928,
        0.2731, 0.2793, 0.2710, 0.2681, 0.2693, 0.3311, 0.2764, 0.3400, 0.2675,
        0.2646, 0.2674, 0.2682, 0.2772, 0.2668, 0.2624, 0.3194, 0.2844, 0.2681,
        0.2896, 0.2706, 0.2640, 0.2686, 0.3144, 0.2703, 0.3692, 0.2694, 0.3227,
        0.2846, 0.2685, 0.2682, 0.3390, 0.2680, 0.3189, 0.2848, 0.2672, 0.3152,
        0.2783, 0.2630, 0.2659, 0.2982, 0.2706, 0.2712, 0.2754, 0.2673, 0.2674,
        0.2636, 0.3267, 0.2725, 0.2831, 0.2640, 0.2660, 0.2719, 0.2752, 0.3329,
        0.2680, 0.3078, 0.2696, 0.2663, 0.2751, 0.2788, 0.3180, 0.2718, 0.2739,
        0.2855, 0.2710, 0.2708, 0.2718, 0.2704, 0.2767, 0.2716, 0.2663, 0.2735,
        0.2738, 0.2848, 0.2780, 0.2667, 0.2602, 0.2801, 0.3053, 0.2695, 0.2890,
        0.2937, 0.2743, 0.2615, 0.2616, 0.2710, 0.2799, 0.2606, 0.2792, 0.2784,
        0.2700, 0.2974, 0.2924, 0.2679, 0.2648, 0.2603, 0.2706, 0.2887, 0.2818,
        0.2649, 0.2664, 0.2746, 0.2638, 0.2719, 0.3260, 0.3248, 0.3171, 0.2641,
        0.2633, 0.2691, 0.2709, 0.2681, 0.2774, 0.2717, 0.2660, 0.3217, 0.2789,
        0.2952, 0.2915, 0.2745, 0.2654, 0.2694, 0.2673, 0.2798, 0.2727, 0.3114,
        0.2624, 0.2691, 0.2747, 0.2912, 0.2913, 0.2719, 0.2834, 0.2786, 0.2746,
        0.2715, 0.2747, 0.2769, 0.2791, 0.2708, 0.2770, 0.2959, 0.2683, 0.3178,
```

```
          0.2675, 0.3168, 0.2679, 0.2917, 0.2870, 0.2926, 0.2762, 0.2693, 0.2902,
          0.3011, 0.2700, 0.3148, 0.3210, 0.2716, 0.2745, 0.2606, 0.2654, 0.2854,
          0.2776, 0.2692, 0.2649, 0.3233, 0.2736, 0.2809, 0.2732, 0.3207, 0.2747,
          0.2819, 0.2767, 0.2708, 0.2695, 0.2815, 0.3204, 0.2733, 0.3105, 0.2912,
          0.2658, 0.2878, 0.2764, 0.2735, 0.2773, 0.2811, 0.2721, 0.2687, 0.2765,
          0.2722, 0.3170, 0.2578, 0.2621, 0.2774, 0.2885, 0.2834, 0.3188, 0.2662,
          0.2889, 0.2691, 0.2791, 0.2699, 0.2774, 0.2830, 0.2850, 0.2745, 0.2685,
          0.2564, 0.4156, 0.2651, 0.2877, 0.2710, 0.2699, 0.3098, 0.3168, 0.2870,
          0.2652, 0.2695, 0.2711, 0.2632, 0.2756, 0.3177, 0.2691, 0.2736, 0.2715,
          0.2766, 0.2771, 0.2690, 0.2824, 0.3030, 0.2702, 0.3066, 0.2723, 0.2792,
          0.2673, 0.2821, 0.3573, 0.2670, 0.2833, 0.2674, 0.2604, 0.3163, 0.2855,
          0.2844, 0.2657, 0.2734, 0.2768, 0.2957, 0.2813, 0.2759, 0.2700, 0.3052,
          0.3353, 0.2782, 0.3094, 0.2983, 0.2720, 0.2702, 0.2743, 0.2646, 0.2695,
          0.2888, 0.3085, 0.2621, 0.2695, 0.2714, 0.2658, 0.2770, 0.2784, 0.3318,
          0.3138, 0.2735, 0.2811, 0.2807, 0.2745, 0.2901, 0.2715, 0.2669, 0.2894,
          0.2954, 0.2742, 0.2639, 0.2684, 0.2694, 0.2716, 0.2776, 0.2748, 0.3764,
          0.2651, 0.2708, 0.2708, 0.2694, 0.2804, 0.2977, 0.2754, 0.2732, 0.2764,
          0.2993, 0.3335, 0.2954, 0.2681, 0.2685, 0.3094, 0.2707, 0.2665, 0.3145,
          0.2682, 0.2746, 0.2781, 0.2684, 0.2780, 0.2656, 0.2809, 0.2733, 0.2690,
          0.3112, 0.2701, 0.2919, 0.2723, 0.2720, 0.2699, 0.2707, 0.3129, 0.2688,
          0.2707, 0.2643, 0.2904, 0.2768, 0.2702])
Val Accuracy: 12.00%
EPOCH 2:
Train Loss: 1.9521502256393433
Train Accuracy: 14.29%
Val Loss: 1.9459271430969238
Val Accuracy: 12.20%
EPOCH 3:
Train Loss: 1.9516496658325195
Train Accuracy: 14.29%
Val Loss: 1.9455398321151733
Val Accuracy: 12.20%
EPOCH 4:
Train Loss: 1.950920581817627
Train Accuracy: 14.29%
Val Loss: 1.9450398683547974
Val Accuracy: 12.40%
EPOCH 5:
Train Loss: 1.9499664306640625
Train Accuracy: 15.00%
Val Loss: 1.9444293975830078
Val Accuracy: 12.40%
EPOCH 6:
Train Loss: 1.9488372802734375
Train Accuracy: 15.00%
Val Loss: 1.943709135055542
Val Accuracy: 12.40%
EPOCH 7:
Train Loss: 1.9475228786468506
Train Accuracy: 14.29%
Val Loss: 1.9428679943084717
Val Accuracy: 12.20%
EPOCH 8:
Train Loss: 1.9460301399230957
Train Accuracy: 14.29%
Val Loss: 1.9419058561325073
Val Accuracy: 12.20%
EPOCH 9:
Train Loss: 1.9444000720977783
Train Accuracy: 14.29%
Val Loss: 1.940821647644043
Val Accuracy: 12.20%
EPOCH 10:
Train Loss: 1.9426367282867432
Train Accuracy: 14.29%
Val Loss: 1.9396061897277832
Val Accuracy: 12.00%
EPOCH 11:
Train Loss: 1.9407501220703125
Train Accuracy: 14.29%
Val Loss: 1.9382563829421997
Val Accuracy: 12.00%
EPOCH 12:
Train Loss: 1.9386993646621704
Train Accuracy: 14.29%
Val Loss: 1.9367274045944214
Val Accuracy: 12.00%
EPOCH 13:
Train Loss: 1.9364323616027832
Train Accuracy: 14.29%
Val Loss: 1.9349912405014038
Val Accuracy: 12.00%
EPOCH 14:
Train Loss: 1.9339362382888794
Train Accuracy: 14.29%
```

```
Val Loss: 1.9330328702926636
Val Accuracy: 15.20%
EPOCH 15:
Train Loss: 1.9312273263931274
Train Accuracy: 18.57%
Val Loss: 1.9308555126190186
Val Accuracy: 17.40%
EPOCH 16:
Train Loss: 1.9283334016799927
Train Accuracy: 20.00%
Val Loss: 1.9284707307815552
Val Accuracy: 17.40%
EPOCH 17:
Train Loss: 1.9252327680587769
Train Accuracy: 20.71%
Val Loss: 1.9258602857589722
Val Accuracy: 17.40%
EPOCH 18:
Train Loss: 1.9219268560409546
Train Accuracy: 20.71%
Val Loss: 1.923136591911316
Val Accuracy: 17.40%
EPOCH 19:
Train Loss: 1.9185022115707397
Train Accuracy: 20.71%
Val Loss: 1.9202640056610107
Val Accuracy: 17.40%
EPOCH 20:
Train Loss: 1.914974570274353
Train Accuracy: 20.71%
Val Loss: 1.9174202680587769
Val Accuracy: 17.40%
EPOCH 21:
Train Loss: 1.9114753007888794
Train Accuracy: 20.71%
Val Loss: 1.9145513772964478
Val Accuracy: 17.40%
EPOCH 22:
Train Loss: 1.9080363512039185
Train Accuracy: 20.71%
Val Loss: 1.9117467403411865
Val Accuracy: 17.40%
EPOCH 23:
Train Loss: 1.9047328233718872
Train Accuracy: 20.71%
Val Loss: 1.9090584516525269
Val Accuracy: 17.40%
EPOCH 24:
Train Loss: 1.901580810546875
Train Accuracy: 20.71%
Val Loss: 1.9065709114074707
Val Accuracy: 17.60%
EPOCH 25:
Train Loss: 1.8986918926239014
Train Accuracy: 20.71%
Val Loss: 1.9042598009109497
Val Accuracy: 17.60%
EPOCH 26:
Train Loss: 1.8960144519805908
Train Accuracy: 20.71%
Val Loss: 1.9021143913269043
Val Accuracy: 17.80%
EPOCH 27:
Train Loss: 1.8935346603393555
Train Accuracy: 20.71%
Val Loss: 1.9001262187957764
Val Accuracy: 18.00%
EPOCH 28:
Train Loss: 1.8912161588668823
Train Accuracy: 20.71%
Val Loss: 1.8982882499694824
Val Accuracy: 18.00%
EPOCH 29:
Train Loss: 1.8890283107757568
Train Accuracy: 20.71%
Val Loss: 1.8965551853179932
Val Accuracy: 18.00%
EPOCH 30:
Train Loss: 1.886904239654541
Train Accuracy: 21.43%
Val Loss: 1.8949073553085327
Val Accuracy: 18.00%
EPOCH 31:
Train Loss: 1.8848042488098145
Train Accuracy: 21.43%
```

```
Val Loss: 1.8933284282684326
Val Accuracy: 18.00%
EPOCH 32:
Train Loss: 1.882717490196228
Train Accuracy: 21.43%
Val Loss: 1.8918063640594482
Val Accuracy: 18.20%
EPOCH 33:
Train Loss: 1.8806222677230835
Train Accuracy: 21.43%
Val Loss: 1.8903343677520752
Val Accuracy: 18.20%
EPOCH 34:
Train Loss: 1.8785114288330078
Train Accuracy: 21.43%
Val Loss: 1.8888964653015137
Val Accuracy: 19.20%
EPOCH 35:
Train Loss: 1.8764088153839111
Train Accuracy: 21.43%
Val Loss: 1.8874759674072266
Val Accuracy: 19.80%
EPOCH 36:
Train Loss: 1.874355673789978
Train Accuracy: 22.14%
Val Loss: 1.886099934577942
Val Accuracy: 19.80%
EPOCH 37:
Train Loss: 1.87240469455719
Train Accuracy: 24.29%
Val Loss: 1.8847908973693848
Val Accuracy: 20.20%
EPOCH 38:
Train Loss: 1.870523452758789
Train Accuracy: 24.29%
Val Loss: 1.8835430145263672
Val Accuracy: 20.60%
EPOCH 39:
Train Loss: 1.8687878847122192
Train Accuracy: 25.00%
Val Loss: 1.8823474645614624
Val Accuracy: 20.60%
EPOCH 40:
Train Loss: 1.8670724630355835
Train Accuracy: 25.71%
Val Loss: 1.881183385848999
Val Accuracy: 20.80%
EPOCH 41:
Train Loss: 1.8653714656829834
Train Accuracy: 25.71%
Val Loss: 1.8800359964370728
Val Accuracy: 20.80%
EPOCH 42:
Train Loss: 1.8636871576309204
Train Accuracy: 26.43%
Val Loss: 1.8789081573486328
Val Accuracy: 21.20%
EPOCH 43:
Train Loss: 1.8620132207870483
Train Accuracy: 26.43%
Val Loss: 1.8777908086776733
Val Accuracy: 21.40%
EPOCH 44:
Train Loss: 1.8603278398513794
Train Accuracy: 26.43%
Val Loss: 1.8766846656799316
Val Accuracy: 21.80%
EPOCH 45:
Train Loss: 1.8586596250534058
Train Accuracy: 26.43%
Val Loss: 1.8755940198898315
Val Accuracy: 22.40%
EPOCH 46:
Train Loss: 1.856994867324829
Train Accuracy: 27.14%
Val Loss: 1.8745105266571045
Val Accuracy: 22.40%
EPOCH 47:
Train Loss: 1.8553235530853271
Train Accuracy: 27.14%
Val Loss: 1.873434066772461
Val Accuracy: 22.60%
EPOCH 48:
Train Loss: 1.8536518812179565
Train Accuracy: 27.86%
```

```
Train Accuracy: 27.86%
Val Loss: 1.8723676204681396
Val Accuracy: 22.80%
EPOCH 49:
Train Loss: 1.8519892692565918
Train Accuracy: 27.86%
Val Loss: 1.8713074922561646
Val Accuracy: 22.80%
EPOCH 50:
Train Loss: 1.850342869758606
Train Accuracy: 27.86%
Val Loss: 1.870255470275879
Val Accuracy: 23.00%
EPOCH 51:
Train Loss: 1.8487186431884766
Train Accuracy: 27.86%
Val Loss: 1.8692127466201782
Val Accuracy: 23.00%
EPOCH 52:
Train Loss: 1.8471171855926514
Train Accuracy: 27.86%
Val Loss: 1.868182897567749
Val Accuracy: 23.20%
EPOCH 53:
Train Loss: 1.8455381393432617
Train Accuracy: 27.86%
Val Loss: 1.867177128791809
Val Accuracy: 23.20%
EPOCH 54:
Train Loss: 1.843995451927185
Train Accuracy: 28.57%
Val Loss: 1.8661792278289795
Val Accuracy: 23.60%
EPOCH 55:
Train Loss: 1.8424853086471558
Train Accuracy: 28.57%
Val Loss: 1.8651939630508423
Val Accuracy: 23.60%
EPOCH 56:
Train Loss: 1.8410147428512573
Train Accuracy: 28.57%
Val Loss: 1.8642233610153198
Val Accuracy: 23.80%
EPOCH 57:
Train Loss: 1.839591383934021
Train Accuracy: 28.57%
Val Loss: 1.8632640838623047
Val Accuracy: 23.80%
EPOCH 58:
Train Loss: 1.8382134437561035
Train Accuracy: 28.57%
Val Loss: 1.8623114824295044
Val Accuracy: 23.80%
EPOCH 59:
Train Loss: 1.836883306503296
Train Accuracy: 28.57%
Val Loss: 1.8613606691360474
Val Accuracy: 23.80%
EPOCH 60:
Train Loss: 1.8355810642242432
Train Accuracy: 28.57%
Val Loss: 1.860410451889038
Val Accuracy: 23.80%
EPOCH 61:
Train Loss: 1.8343100547790527
Train Accuracy: 28.57%
Val Loss: 1.8594592809677124
Val Accuracy: 24.20%
EPOCH 62:
Train Loss: 1.8330754041671753
Train Accuracy: 28.57%
Val Loss: 1.8585060834884644
Val Accuracy: 24.60%
EPOCH 63:
Train Loss: 1.8318653106689453
Train Accuracy: 28.57%
Val Loss: 1.8575475215911865
Val Accuracy: 24.80%
EPOCH 64:
Train Loss: 1.830693244934082
Train Accuracy: 28.57%
Val Loss: 1.8565865755081177
Val Accuracy: 24.80%
EPOCH 65:
Train Loss: 1.8295599222183228
```

```
Train Accuracy: 28.57%
Val Loss: 1.8556259870529175
Val Accuracy: 25.00%
EPOCH 66:
Train Loss: 1.8284614086151123
Train Accuracy: 28.57%
Val Loss: 1.8546689748764038
Val Accuracy: 25.20%
EPOCH 67:
Train Loss: 1.827397108078003
Train Accuracy: 28.57%
Val Loss: 1.8537142276763916
Val Accuracy: 25.20%
EPOCH 68:
Train Loss: 1.8263646364212036
Train Accuracy: 28.57%
Val Loss: 1.852761149406433
Val Accuracy: 25.80%
EPOCH 69:
Train Loss: 1.8253555297851562
Train Accuracy: 28.57%
Val Loss: 1.8518062829971313
Val Accuracy: 25.80%
EPOCH 70:
Train Loss: 1.8243719339370728
Train Accuracy: 28.57%
Val Loss: 1.8508481979370117
Val Accuracy: 25.80%
EPOCH 71:
Train Loss: 1.8234150409698486
Train Accuracy: 28.57%
Val Loss: 1.8498880863189697
Val Accuracy: 26.00%
EPOCH 72:
Train Loss: 1.8224785327911377
Train Accuracy: 28.57%
Val Loss: 1.8489230871200562
Val Accuracy: 26.20%
EPOCH 73:
Train Loss: 1.8215575218200684
Train Accuracy: 28.57%
Val Loss: 1.8479564189910889
Val Accuracy: 26.20%
EPOCH 74:
Train Loss: 1.8206526041030884
Train Accuracy: 28.57%
Val Loss: 1.8469841480255127
Val Accuracy: 26.60%
EPOCH 75:
Train Loss: 1.8197617530822754
Train Accuracy: 28.57%
Val Loss: 1.8460109233856201
Val Accuracy: 26.80%
EPOCH 76:
Train Loss: 1.8188802003860474
Train Accuracy: 28.57%
Val Loss: 1.8450357913970947
Val Accuracy: 27.20%
EPOCH 77:
Train Loss: 1.8180097341537476
Train Accuracy: 28.57%
Val Loss: 1.8440557718276978
Val Accuracy: 27.20%
EPOCH 78:
Train Loss: 1.8171457052230835
Train Accuracy: 28.57%
Val Loss: 1.843070149421692
Val Accuracy: 27.20%
EPOCH 79:
Train Loss: 1.8162941932678223
Train Accuracy: 28.57%
Val Loss: 1.8420774936676025
Val Accuracy: 27.80%
EPOCH 80:
Train Loss: 1.8154478073120117
Train Accuracy: 28.57%
Val Loss: 1.8410776853561401
Val Accuracy: 27.80%
EPOCH 81:
Train Loss: 1.8146089315414429
Train Accuracy: 28.57%
Val Loss: 1.8400706052780151
Val Accuracy: 27.80%
EPOCH 82:
Train Loss: 1.8137761354446411
```

```
Train Accuracy: 29.29%
Val Loss: 1.8390562534332275
Val Accuracy: 27.80%
EPOCH 83:
Train Loss: 1.8129472732543945
Train Accuracy: 29.29%
Val Loss: 1.8380343914031982
Val Accuracy: 28.20%
EPOCH 84:
Train Loss: 1.8121225833892822
Train Accuracy: 29.29%
Val Loss: 1.837004542350769
Val Accuracy: 28.40%
EPOCH 85:
Train Loss: 1.8113031387329102
Train Accuracy: 29.29%
Val Loss: 1.8359696865081787
Val Accuracy: 29.00%
EPOCH 86:
Train Loss: 1.8104853630065918
Train Accuracy: 29.29%
Val Loss: 1.834928035736084
Val Accuracy: 29.20%
EPOCH 87:
Train Loss: 1.8096705675125122
Train Accuracy: 29.29%
Val Loss: 1.833882451057434
Val Accuracy: 29.40%
EPOCH 88:
Train Loss: 1.8088616132736206
Train Accuracy: 29.29%
Val Loss: 1.8328371047973633
Val Accuracy: 29.60%
EPOCH 89:
Train Loss: 1.8080569505691528
Train Accuracy: 30.00%
Val Loss: 1.8317954540252686
Val Accuracy: 30.40%
EPOCH 90:
Train Loss: 1.8072558641433716
Train Accuracy: 30.71%
Val Loss: 1.8307583332061768
Val Accuracy: 30.60%
EPOCH 91:
Train Loss: 1.8064581155776978
Train Accuracy: 31.43%
Val Loss: 1.8297251462936401
Val Accuracy: 30.80%
EPOCH 92:
Train Loss: 1.8056614398956299
Train Accuracy: 31.43%
Val Loss: 1.8286961317062378
Val Accuracy: 31.20%
EPOCH 93:
Train Loss: 1.8048689365386963
Train Accuracy: 32.14%
Val Loss: 1.827669620513916
Val Accuracy: 31.40%
EPOCH 94:
Train Loss: 1.8040744066238403
Train Accuracy: 32.14%
Val Loss: 1.8266481161117554
Val Accuracy: 32.00%
EPOCH 95:
Train Loss: 1.8032835721969604
Train Accuracy: 32.14%
Val Loss: 1.8256309032440186
Val Accuracy: 32.20%
EPOCH 96:
Train Loss: 1.8024961948394775
Train Accuracy: 32.14%
Val Loss: 1.8246170282363892
Val Accuracy: 32.20%
EPOCH 97:
Train Loss: 1.8017090559005737
Train Accuracy: 32.14%
Val Loss: 1.823607087135315
Val Accuracy: 32.80%
EPOCH 98:
Train Loss: 1.8009239435195923
Train Accuracy: 32.14%
Val Loss: 1.8226039409637451
Val Accuracy: 32.80%
EPOCH 99:
Train Loss: 1.8001371622085571
```

```
Train Accuracy: 32.86%
Val Loss: 1.8216078281402588
Val Accuracy: 33.00%
EPOCH 100:
Train Loss: 1.7993520498275757
Train Accuracy: 33.57%
Val Loss: 1.8206170797348022
Val Accuracy: 33.40%
EPOCH 101:
Train Loss: 1.7985676527023315
max_prob_index:  tensor([0, 0, 3, 5, 0, 0, 0, 5, 0, 0, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        3, 0, 0, 3, 0, 5, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 5,
        3, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 5, 2, 0, 0, 0, 3,
        2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 2, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 2, 1, 1, 0, 0, 5, 0, 0, 1, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 1, 0, 1, 0, 5, 5, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1])
max_prob_values:  tensor([0.2553, 0.2929, 0.3697, 0.2565, 0.3030, 0.3073, 0.5544, 0.2530, 0.2710,
        0.2722, 0.2864, 0.3200, 0.2885, 0.5202, 0.2935, 0.2677, 0.3005, 0.2853,
        0.3295, 0.2518, 0.2367, 0.2606, 0.3496, 0.2538, 0.7677, 0.3114, 0.2769,
        0.3054, 0.2510, 0.2385, 0.9842, 0.2567, 0.3439, 0.3596, 0.9922, 0.3473,
        0.3639, 0.2947, 0.2995, 0.3597, 0.3135, 0.4330, 0.2812, 0.3002, 0.2816,
        0.4751, 0.2496, 0.2572, 0.4236, 0.2387, 0.2574, 0.3314, 0.5816, 0.9959,
        0.2907, 0.6128, 0.3801, 0.2780, 0.2529, 0.9952, 0.3563, 0.4833, 0.2675,
        0.2559, 0.2805, 0.6800, 0.2544, 0.2771, 0.9899, 0.4886, 0.2717, 0.3242,
        0.9946, 0.9879, 0.7590, 0.3359, 0.3445, 0.2982, 0.3047, 0.2771, 0.3663,
        0.4469, 0.2513, 0.2715, 0.3139, 0.3252, 0.2678, 0.2835, 0.9673, 0.4493,
        0.9927, 0.3137, 0.2641, 0.3327, 0.5404, 0.9589, 0.2604, 0.2600, 0.2634,
        0.2983, 0.3800, 0.9915, 0.7932, 0.9027, 0.3320, 0.2529, 0.2569, 0.2800,
        0.2810, 0.9998, 0.2711, 0.9982, 0.9724, 0.2702, 0.2779, 0.2905, 0.2877,
        0.2673, 0.5833, 0.3113, 0.3563, 0.3190, 0.2690, 0.2773, 0.9515, 0.2608,
        0.9905, 0.2629, 0.2689, 0.2686, 0.3275, 0.2350, 0.2596, 0.9526, 0.2579,
        0.2945, 0.2528, 0.3280, 0.8715, 0.9918], grad_fn=<MaxBackward0>)
Train Accuracy: 33.57%
Val Loss: 1.819632887840271
max_prob_index:  tensor([0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 3, 0, 1, 2, 2, 2, 3, 3, 0, 3, 0, 0, 0,
        0, 3, 0, 0, 0, 0, 2, 0, 0, 2, 2, 0, 1, 0, 5, 3, 0, 0, 0, 0, 0, 0, 0, 0,
        2, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 0, 0, 3, 0, 3, 0, 3, 0, 5, 5, 0, 0, 0,
        5, 0, 0, 3, 3, 0, 3, 0, 0, 0, 5, 0, 3, 0, 3, 0, 2, 0, 3, 3, 3, 5, 0, 3,
        1, 0, 5, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 3, 0, 2, 5, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 3, 0, 0,
        0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 1, 0, 0, 3, 0, 0, 0, 2, 1, 0, 0, 0, 1, 0,
        1, 0, 0, 0, 0, 0, 1, 0, 3, 0, 1, 5, 0, 0, 5, 0, 0, 0, 2, 3, 0, 1, 0, 0,
        0, 2, 0, 3, 0, 2, 0, 0, 0, 2, 0, 3, 0, 5, 2, 0, 0, 0, 1, 3, 3, 0, 0, 0,
        0, 2, 0, 0, 5, 0, 0, 0, 2, 0, 3, 5, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0,
        0, 0, 0, 3, 1, 0, 0, 0, 0, 0, 0, 3, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 3, 1, 0, 0, 0, 0, 0, 0, 0, 5, 1, 5,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0,
        1, 0, 0, 0, 0, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 0, 2, 3, 5, 0, 0, 0, 0, 0,
        1, 0, 0, 0, 3, 3, 3, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0,
        5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 2, 0, 0, 0, 0, 0, 1, 5, 3, 0, 0,
        2, 0, 2, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 3,
        0, 3, 0, 0, 2, 0, 1, 0, 1, 5, 0, 5, 0, 0, 2, 1, 0, 0, 3, 0, 3, 0, 0, 0,
        0, 1, 0, 0, 3, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 3, 0, 5, 0, 5, 0, 0, 0,
        0, 0, 3, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 1, 2, 1, 5, 5, 2, 0, 0, 2, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 0])
max_prob_values:  tensor([0.2592, 0.2574, 0.6091, 0.6433, 0.2946, 0.3381, 0.3999, 0.2652, 0.3163,
        0.4448, 0.2860, 0.3661, 0.2980, 0.9977, 0.9955, 0.9905, 0.9921, 0.6226,
        0.5547, 0.2620, 0.4203, 0.2768, 0.3929, 0.2527, 0.2882, 0.3870, 0.2703,
        0.3267, 0.2882, 0.2709, 0.9926, 0.2887, 0.3124, 0.9943, 0.9940, 0.5010,
        0.3903, 0.3011, 0.2518, 0.3543, 0.3659, 0.3697, 0.2473, 0.2718, 0.2602,
        0.2674, 0.3680, 0.2601, 0.3016, 0.2422, 0.2750, 0.3845, 0.2571, 0.2672,
        0.3204, 0.2865, 0.2574, 0.2521, 0.2527, 0.2928, 0.2763, 0.8201, 0.2807,
        0.2924, 0.3027, 0.4412, 0.2751, 0.2530, 0.2560, 0.2552, 0.6095, 0.4321,
        0.2553, 0.2940, 0.2665, 0.4937, 0.4308, 0.2859, 0.3946, 0.2971, 0.2783,
        0.2459, 0.2549, 0.2474, 0.3236, 0.2569, 0.3964, 0.2480, 0.9927, 0.2902,
        0.3449, 0.5956, 0.6624, 0.2582, 0.2650, 0.3972, 0.9972, 0.2512, 0.2648,
        0.7018, 0.3489, 0.2588, 0.3513, 0.3371, 0.9932, 0.2727, 0.9931, 0.2610,
        0.3042, 0.3092, 0.2590, 0.3808, 0.3758, 0.3068, 0.2946, 0.3144, 0.2635,
        0.4548, 0.3050, 0.2668, 0.2611, 0.4776, 0.3020, 0.2752, 0.2618, 0.2582,
        0.3278, 0.2955, 0.2335, 0.3007, 0.4096, 0.2916, 0.2635, 0.2595, 0.3055,
        0.2845, 0.3723, 0.3949, 0.2552, 0.5661, 0.3566, 0.2913, 0.2721, 0.2612,
        0.3450, 0.2721, 0.2485, 0.2607, 0.2582, 0.9765, 0.4562, 0.2561, 0.2546,
        0.2624, 0.7434, 0.2890, 0.2665, 0.4109, 0.2475, 0.2697, 0.3733, 0.3487,
        0.9110, 0.2845, 0.3006, 0.3698, 1.0000, 0.2621, 0.9638, 0.2739, 0.4165,
        0.3016, 0.2479, 0.2593, 0.3055, 0.5323, 0.5202, 0.2511, 0.5876, 0.2473,
        0.2703, 0.3018, 0.2503, 0.4228, 0.3288, 0.2892, 0.9954, 0.2677, 0.2608,
        0.8009, 0.2922, 0.3017, 0.3175, 0.9915, 0.4381, 0.5557, 0.2624, 0.9906,
        0.3052, 0.2575, 0.2679, 0.9862, 0.2262, 0.4394, 0.2622, 0.2540, 0.9926,
        0.5162, 0.2979, 0.3026, 0.9904, 0.2525, 0.2513, 0.3035, 0.3279, 0.2549,
        0.3787, 0.9944, 0.2525, 0.2390, 0.2591, 0.2560, 0.2727, 0.2528, 0.9963,
        0.2786, 0.3471, 0.2338, 0.3333, 0.2869, 0.2947, 0.3790, 0.2659, 0.4237,
        0.3463, 0.2827, 0.2968, 0.3141, 0.2597, 0.2644, 0.2791, 0.2696, 0.2564,
        0.2910, 0.4677, 0.3103, 0.2774, 0.2500, 0.2926, 0.2778, 0.2564, 0.3205,
        0.3160, 0.2625, 0.3013, 0.2529, 0.2673, 0.3146, 0.2598, 0.3595, 0.2628,
        0.3171, 0.3790, 0.3276, 0.2721, 0.3715, 0.2847, 0.3452, 0.6526, 0.3085
```

```
          0.3171, 0.3790, 0.3270, 0.2721, 0.3715, 0.2847, 0.3432, 0.6926, 0.3085,
          0.2567, 0.2957, 0.3553, 0.2566, 0.2794, 0.8874, 0.4526, 0.9273, 0.2587,
          0.2608, 0.2961, 0.2677, 0.2549, 0.5237, 0.2837, 0.2527, 0.9872, 0.2310,
          0.2863, 0.4919, 0.2640, 0.2704, 0.2691, 0.2773, 0.4720, 0.3032, 0.6367,
          0.3670, 0.3704, 0.2971, 0.4013, 0.5135, 0.3326, 0.3433, 0.3301, 0.2807,
          0.2858, 0.3763, 0.2612, 0.2650, 0.2733, 0.3220, 0.9642, 0.2532, 0.4811,
          0.2707, 0.7949, 0.3109, 0.3858, 0.7744, 0.3447, 0.3271, 0.2590, 0.3481,
          0.3077, 0.3715, 0.9943, 0.4394, 0.2513, 0.2737, 0.2972, 0.3835, 0.3344,
          0.3128, 0.2615, 0.2719, 0.9020, 0.2740, 0.3336, 0.2712, 0.4299, 0.3016,
          0.4024, 0.3132, 0.3282, 0.4358, 0.3052, 0.9783, 0.2627, 0.9926, 0.5219,
          0.2616, 0.3364, 0.2894, 0.2528, 0.2849, 0.3585, 0.2750, 0.4176, 0.3001,
          0.2457, 0.9969, 0.2924, 0.2765, 0.3844, 0.3731, 0.4090, 0.6282, 0.2503,
          0.2961, 0.2901, 0.2646, 0.3418, 0.2327, 0.3684, 0.3174, 0.3013, 0.2800,
          0.2970, 0.9907, 0.2567, 0.2898, 0.2712, 0.2840, 0.9965, 0.9264, 0.2533,
          0.3427, 0.2672, 0.3084, 0.3161, 0.2712, 0.4226, 0.2276, 0.2866, 0.2758,
          0.2288, 0.3143, 0.2596, 0.4005, 0.3066, 0.2622, 0.9637, 0.3077, 0.6284,
          0.2559, 0.2780, 0.5695, 0.3276, 0.2951, 0.2551, 0.2745, 0.9962, 0.3789,
          0.7917, 0.2802, 0.3669, 0.2279, 0.2594, 0.2684, 0.2744, 0.2556, 0.2815,
          0.6320, 0.2783, 0.3007, 0.3404, 0.2640, 0.2865, 0.2733, 0.2472, 0.3005,
          0.4209, 0.8494, 0.2765, 0.2589, 0.3327, 0.4263, 0.3597, 0.3109, 0.9984,
          0.9735, 0.2895, 0.2964, 0.3128, 0.2597, 0.4934, 0.2573, 0.2641, 0.2499,
          0.2507, 0.3510, 0.2542, 0.3506, 0.2831, 0.2830, 0.4591, 0.3423, 0.8083,
          0.2590, 0.2975, 0.2588, 0.2636, 0.2604, 0.4437, 0.2825, 0.3601, 0.2584,
          0.9705, 0.9948, 0.3727, 0.2557, 0.2586, 0.9928, 0.3065, 0.2431, 0.9947,
          0.2424, 0.2897, 0.3467, 0.2927, 0.3590, 0.2947, 0.3008, 0.3674, 0.2573,
          0.9876, 0.3001, 0.3797, 0.2687, 0.2926, 0.2394, 0.2592, 0.4657, 0.2906,
          0.2555, 0.2591, 0.3164, 0.2917, 0.2539])
Val Accuracy: 33.80%
EPOCH 102:
Train Loss: 1.797785997390747
Train Accuracy: 35.00%
Val Loss: 1.8186548948287964
Val Accuracy: 33.80%
EPOCH 103:
Train Loss: 1.7970081567764282
Train Accuracy: 36.43%
Val Loss: 1.817684531211853
Val Accuracy: 33.80%
EPOCH 104:
Train Loss: 1.7962324619293213
Train Accuracy: 36.43%
Val Loss: 1.8167195320129395
Val Accuracy: 34.20%
EPOCH 105:
Train Loss: 1.795453429222107
Train Accuracy: 36.43%
Val Loss: 1.8157609701156616
Val Accuracy: 34.60%
EPOCH 106:
Train Loss: 1.794672966003418
Train Accuracy: 36.43%
Val Loss: 1.8148082494735718
Val Accuracy: 34.80%
EPOCH 107:
Train Loss: 1.793893814086914
Train Accuracy: 36.43%
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-37-e5d80e2fc245> in <cell line: 9>()
     20     with torch.no_grad():
     21
---> 22       outputs = model(features) #validation
     23
     24       outputs_val = outputs[val_mask]

                                 5 frames
<ipython-input-29-3b39a3239af9> in forward(self, x)
     29
     30         #msg passing
---> 31         msg_neigh = torch.matmul(x ,self.weights_neigh)
     32         # aggregation
     33         aggreg_neigh = torch.matmul(self.adj_M, msg_neigh)

KeyboardInterrupt:
```

Results

```
# Model performance on test set
test_running_loss= []
test_running_acc = []

model.eval()

with torch.no_grad():

  outputs_ = model(features) # test

  outputs_test = outputs_[test_mask]
  labels_test = labels[test_mask]
  test_loss = loss_fn(outputs_test, labels_test)

  total = labels_test.shape[0]
  print(f'Test Loss: {test_loss}')
  test_running_loss.append(test_loss)


  max_prob_ , max_prob_index_ = torch.max(outputs_test, dim=1)
  correct_test = (max_prob_index_ == labels_test).sum().item()

  accuracy_test = correct_test / total
  print(f'Test Accuracy: {accuracy_test * 100:.2f}%')
  test_running_acc.append(accuracy_test)
```

```
Test Loss: 1.8268849849700928
Test Accuracy: 33.70%
```

```
train_losses = [ val.detach().numpy() for val in running_loss ]
train_accuracies = running_acc
val_losses = [ val.detach().numpy() for val in val_running_loss ]
val_accuracies = val_running_acc
test_losses = [ val.detach().numpy() for val in test_running_loss ]
test_accuracies = test_running_acc



# accuracy and cross entropy loss plots for training and validation

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curves')

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy')
plt.plot(val_accuracies, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Curves')

plt.show()
```


```
# confusion matrix on test set

y_true = labels_test   #  Ground truth labels
y_pred = max_prob_index_      #  Predicted labels

cm = confusion_matrix(y_true, y_pred)

classes = np.unique(y_true)

plt.figure(figsize=(6, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()

tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes)
plt.yticks(tick_marks, classes)

for i, j in np.ndindex(cm.shape):
    plt.text(j, i, cm[i, j], horizontalalignment="center", color="white" if cm[i, j] > cm.max() / 2 else "black")

plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.show()
```