# CPSC 552 – Assignment #13 - Graph Neural Networks (GNN)

In this assignment, we will program two kinds of Graph Neural Networks, one is Graph Convolution Network and the other Graph Attention Network.

In a GNN, each node in the graph performs an update on its feature vector *H* as,

$$H^{(l+1)} = f(H^{(l)}, A)$$

- A feature description $x_i$ for every node $i$; summarized in a $N \times D$ feature matrix $X$ ($N$: number of nodes, $D$: number of input features)
- A representative description of the graph structure in matrix form; typically in the form of an adjacency matrix $A$ (or some function thereof)

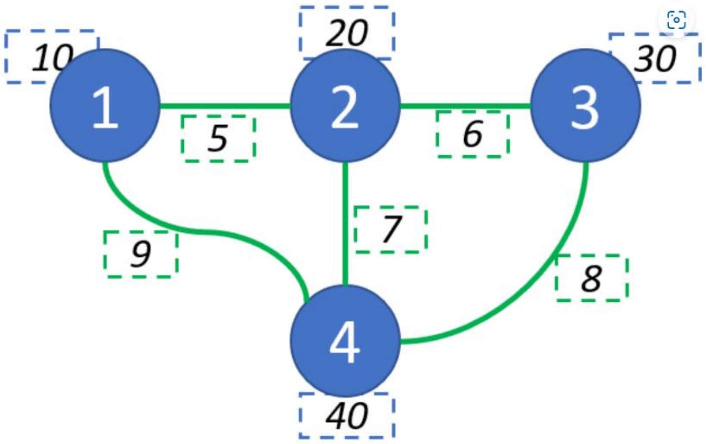
In a Graph Neural Network (GNN), the features in a node are updated as:

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right),$$

with $\hat{A} = A + I$, where $I$ is the identity matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

σ is the non-linearity function e.g., RELU.

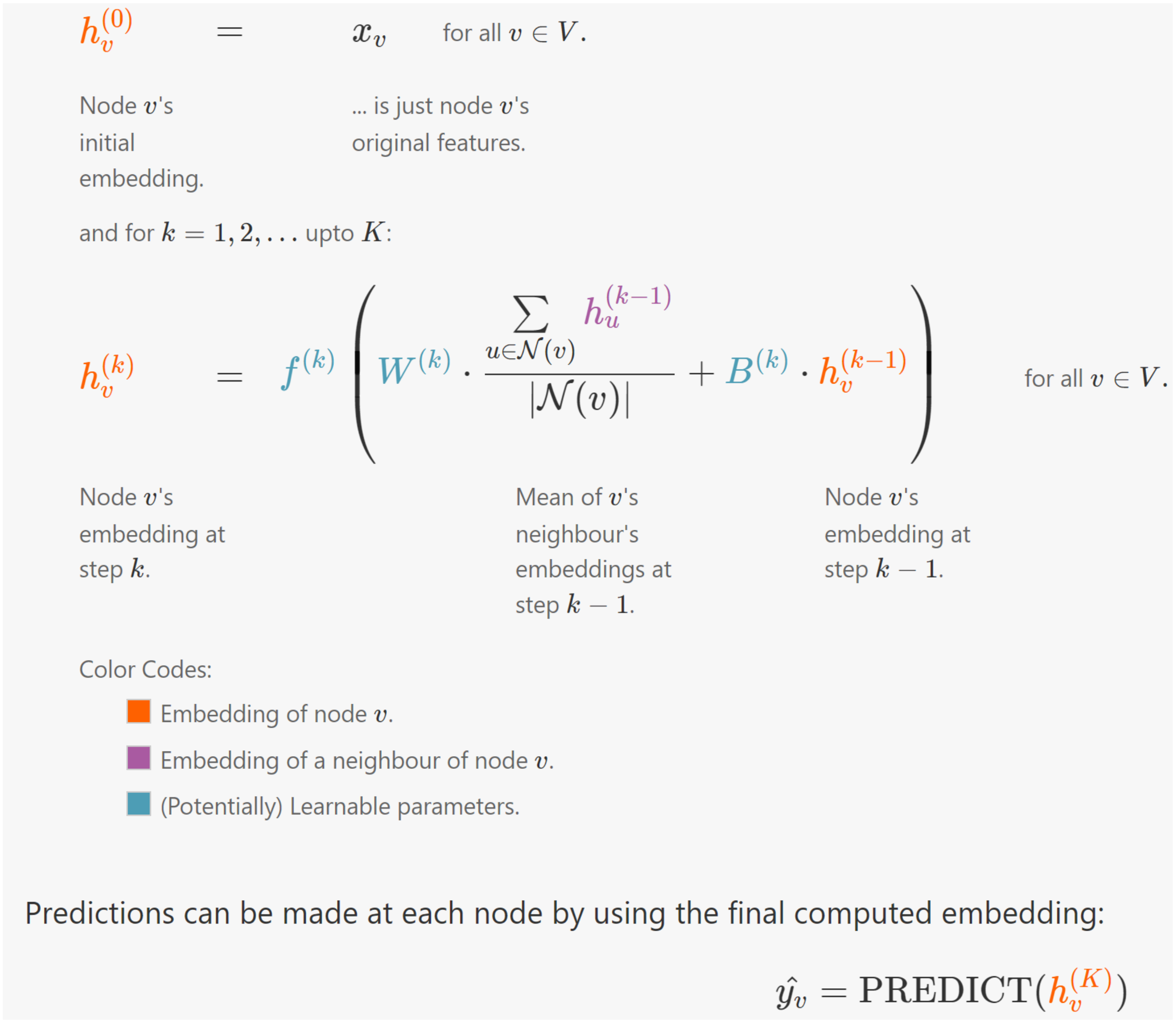
Example:



$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & 0 & 9 \\ 2 & 5 & 0 & 6 & 7 \\ 3 & 0 & 6 & 0 & 8 \\ 4 & 9 & 7 & 8 & 0 \end{array} ; H = \begin{array}{c|c} 1 & 10 \\ 2 & 20 \\ 3 & 30 \\ 4 & 40 \end{array} \quad D = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 \\ 3 & 0 & 0 & 2 & 0 \\ 4 & 0 & 0 & 0 & 3 \end{array}$$

Reference: [Tutorial Graph Neural Networks on Social Networks | Kaggle]

**Graph Convolution Network (GCN):** The GCN shares the weight matrix with all nodes. Each layer of GCN has its own weight matrix. Just like a CNN network, we can have many layers, and attach a classifier at the end, if we wanted to do node classification. In a GCN, each node v updates its feature vector according to its connected neighbors as (k is the layer number):



$$h_v^{(0)} = x_v \quad \text{for all } v \in V.$$

Node $v$'s initial embedding.

... is just node $v$'s original features.

and for $k = 1, 2, \ldots$ upto $K$:

$$h_v^{(k)} = f^{(k)}\left(W^{(k)} \cdot \frac{\sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}}{|\mathcal{N}(v)|} + B^{(k)} \cdot h_v^{(k-1)}\right) \quad \text{for all } v \in V.$$

Node $v$'s embedding at step $k$.

Mean of $v$'s neighbour's embeddings at step $k-1$.

Node $v$'s embedding at step $k-1$.

Color Codes:

- 🟧 Embedding of node $v$.
- 🟪 Embedding of a neighbour of node $v$.
- 🟦 (Potentially) Learnable parameters.

Predictions can be made at each node by using the final computed embedding:

$$\hat{y}_v = \text{PREDICT}(h_v^{(K)})$$

Ref: Understanding Convolutions on Graphs (distill.pub)

GCN is programmed as a layer in the Pytorch geometric library. We will program the classification of the Karate club dataset using GCNs. Karate club dataset has 34 nodes with each node having 34 features. The features in this dataset are actually meaning less and are simply one hot vector representing the node number. There are 78 total edges and each node is classified into one of 4 categories.

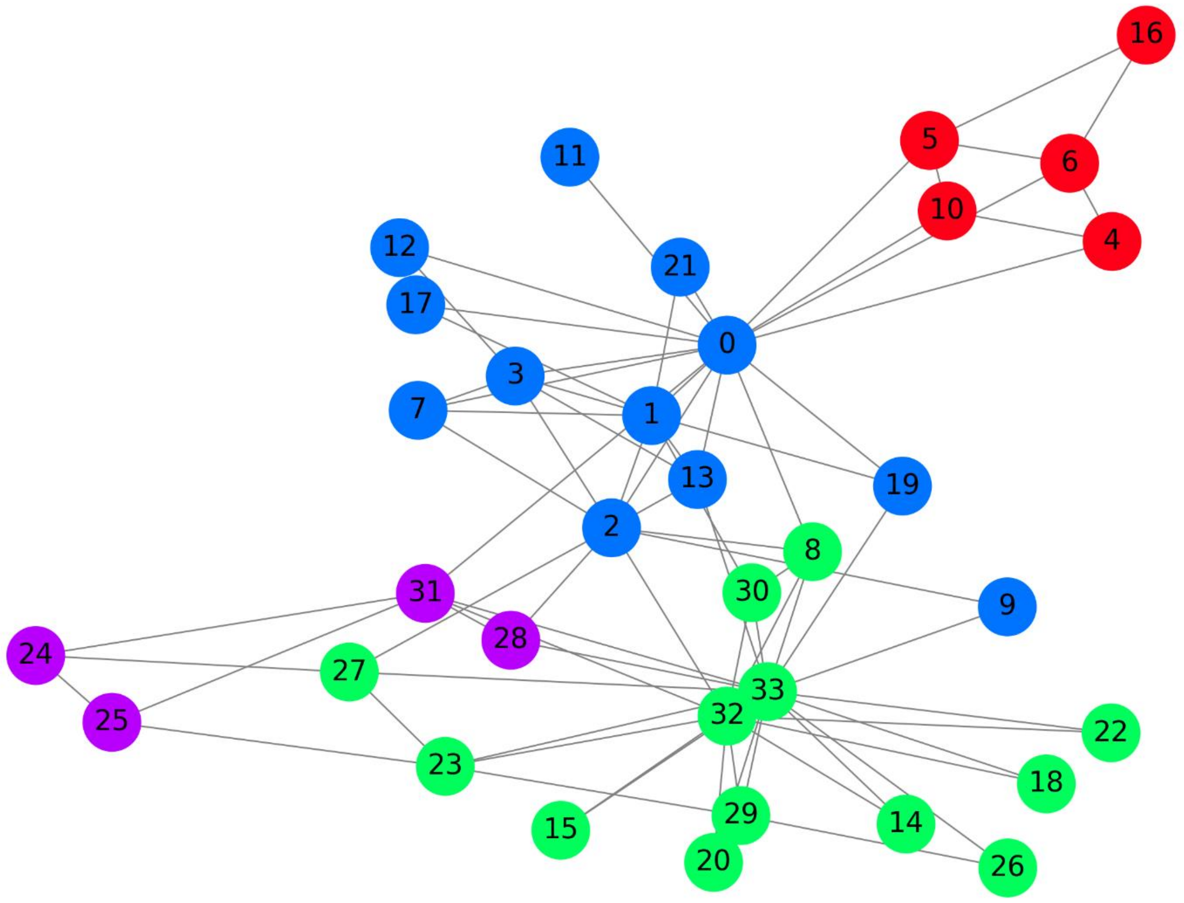
Create a Python application called GCNKarate. Add a class called MyGCN with the following code in it.

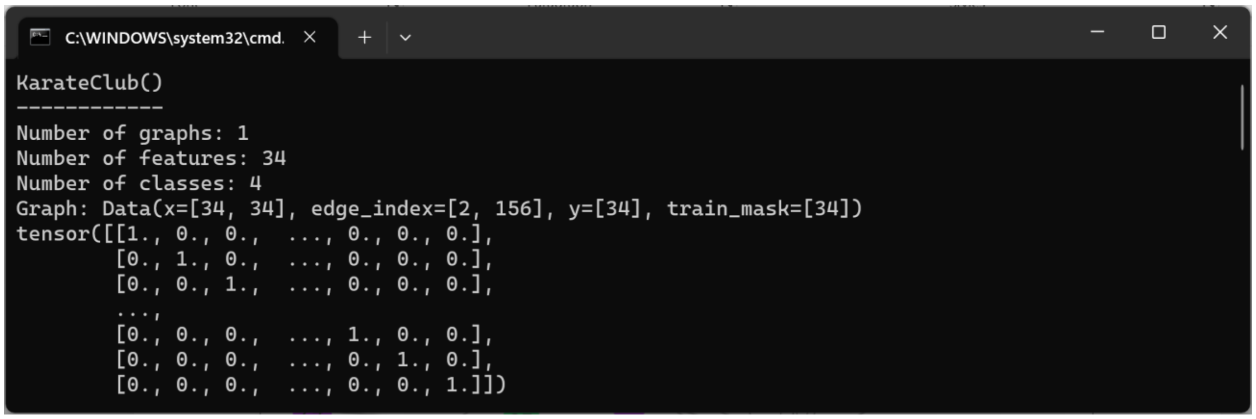
```
from torch_geometric.nn import GCNConv
from torch import nn

class MyGCN(nn.Module):
    def __init__(self,num_features, num_hidden, num_classes):
        super().__init__()
        self.gcn = GCNConv(num_features, num_hidden)
        self.linearnet = nn.Linear(num_hidden, num_classes)

    def forward(self, x, edge_index):
        h = self.gcn(x, edge_index).relu()
        out = self.linearnet(h)
        return h, out
```

Type the following code in the KarateGCN.py file.

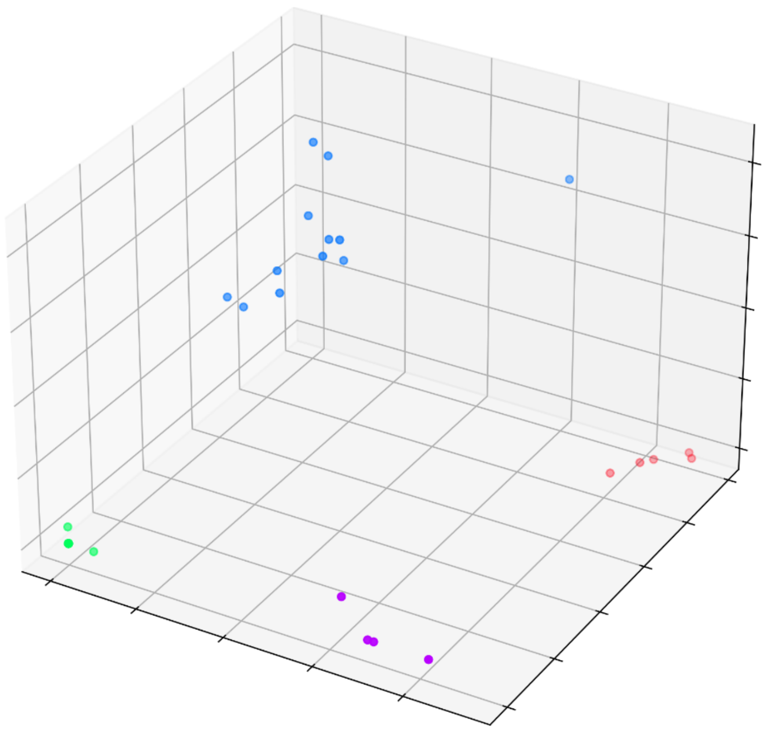
```
import numpy as np
import sys

# pip install networkx
# pip install torch-geometric

from torch_geometric.datasets import KarateClub
import matplotlib.pyplot as plt
import networkx as nx
from torch_geometric.utils import to_networkx
from MyGCN import MyGCN
import torch

def plot_graph(data):
    G = to_networkx(data, to_undirected=True)
    plt.figure(figsize=(12,12))
    plt.axis('off')
    nx.draw_networkx(G,
                     pos=nx.spring_layout(G, seed=0),
                     with_labels=True,
                     node_size=800,
                     node_color=data.y,
                     cmap="hsv",
                     vmin=-2,
                     vmax=3,
                     width=0.8,
                     edge_color="grey",
                     font_size=14
                     )
    plt.show()

def plot_embeddings(data, embeddings):
    fig = plt.figure(figsize=(12, 12))
    ax = fig.add_subplot(projection='3d')
    ax.patch.set_alpha(0)
    plt.tick_params(left=False,
                    bottom=False,
                    labelleft=False,
                    labelbottom=False)
    ax.scatter(embeddings[:, 0], embeddings[:, 1], embeddings[:, 2],
               s=200, c=data.y, cmap="hsv", vmin=-2, vmax=3)
    plt.show()

def compute_accuracy(actual_y, expected_y):
    return (expected_y == actual_y).sum() / len(actual_y)

def main():
    dataset = KarateClub()

    # Print information
    print(dataset)
    print('------------')
    print(f'Number of graphs: {len(dataset)}')
    print(f'Number of features: {dataset.num_features}')
    print(f'Number of classes: {dataset.num_classes}')
    print(f'Graph: {dataset[0]}')
    # 34 nodes with each node having 34 fatures, 78 edges
    data = dataset[0]
    print(data.x)  # identity matrix, the node itself does not
    # contain any info in this example, but can contain features
    # such as age, weight, height

    print(data.edge_index) # 2x78 = 156 edges
    # we will use edge info to classify a node
    # edge indicates which node is connected to which node
    # e.g., 0 -> 1,2,3,4,5,6,7,8,10

    print(data.y) # 34 values
    print(data.train_mask)  # true to use for training, false for testing

    plot_graph(data)

    model = MyGCN(num_features=34,num_hidden=3,num_classes=4)
    print(model)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.02)

    num_epochs = 100

    # train GCN
    for epoch in range(num_epochs):
        optimizer.zero_grad()
        h, out = model(data.x, data.edge_index) # h is learnt embedding
        print(h.shape)
        loss = criterion(out, data.y)
        acc = compute_accuracy(out.argmax(dim=1), data.y)
        loss.backward()
        optimizer.step()
        if epoch % 10 == 0:
            print(f'Epoch {epoch:>3} | Loss: {loss:.2f} | Acc: {acc*100:.2f}%')
    # plot final learnt embedding for each train node
    plot_embeddings(data, h.detach().cpu().numpy())

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

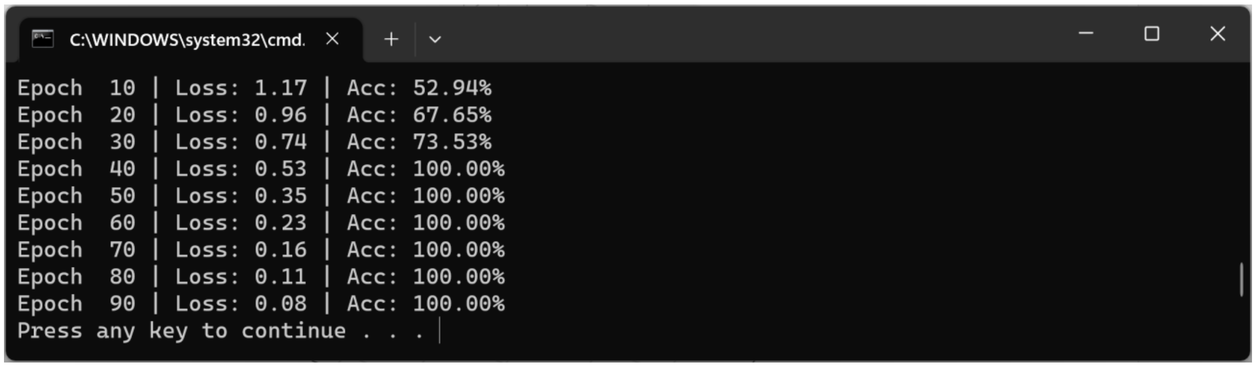
If you run the above program, the output will appear as:





```
KarateClub()
------------
Number of graphs: 1
Number of features: 34
Number of classes: 4
Graph: Data(x=[34, 34], edge_index=[2, 156], y=[34], train_mask=[34])
tensor([[1., 0., 0.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        [0., 0., 1.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 1., 0., 0.],
        [0., 0., 0.,  ..., 0., 1., 0.],
        [0., 0., 0.,  ..., 0., 0., 1.]])
```





```
Epoch  10 | Loss: 1.17 | Acc: 52.94%
Epoch  20 | Loss: 0.96 | Acc: 67.65%
Epoch  30 | Loss: 0.74 | Acc: 73.53%
Epoch  40 | Loss: 0.53 | Acc: 100.00%
Epoch  50 | Loss: 0.35 | Acc: 100.00%
Epoch  60 | Loss: 0.23 | Acc: 100.00%
Epoch  70 | Loss: 0.16 | Acc: 100.00%
Epoch  80 | Loss: 0.11 | Acc: 100.00%
Epoch  90 | Loss: 0.08 | Acc: 100.00%
Press any key to continue . . .
```

The above program used a single layer GCN follower by a linear network (MyGCN.py file). Just like with CNN networks, we can have multiple layers of GCNs. Add a file called MyGCN2 with the following code in it. This model will use two GCN layers before the classification linear network.

```
from torch_geometric.nn import GCNConv
from torch import nn

class MyGCN2(nn.Module):  # 2 layer GCN
    def __init__(self,num_features, num_hidden, num_classes):
        super().__init__()
        self.gcn1 = GCNConv(num_features, 10)
        self.gcn2 = GCNConv(10, num_hidden)
        self.linearnet = nn.Linear(num_hidden, num_classes)
    def forward(self, x, edge_index):
        h1 = self.gcn1(x, edge_index).relu()
        h2 = self.gcn2(h1, edge_index).relu()
        out = self.linearnet(h2)
        return h2, out
```

Add an import for the MyGCN2 in the main file as:

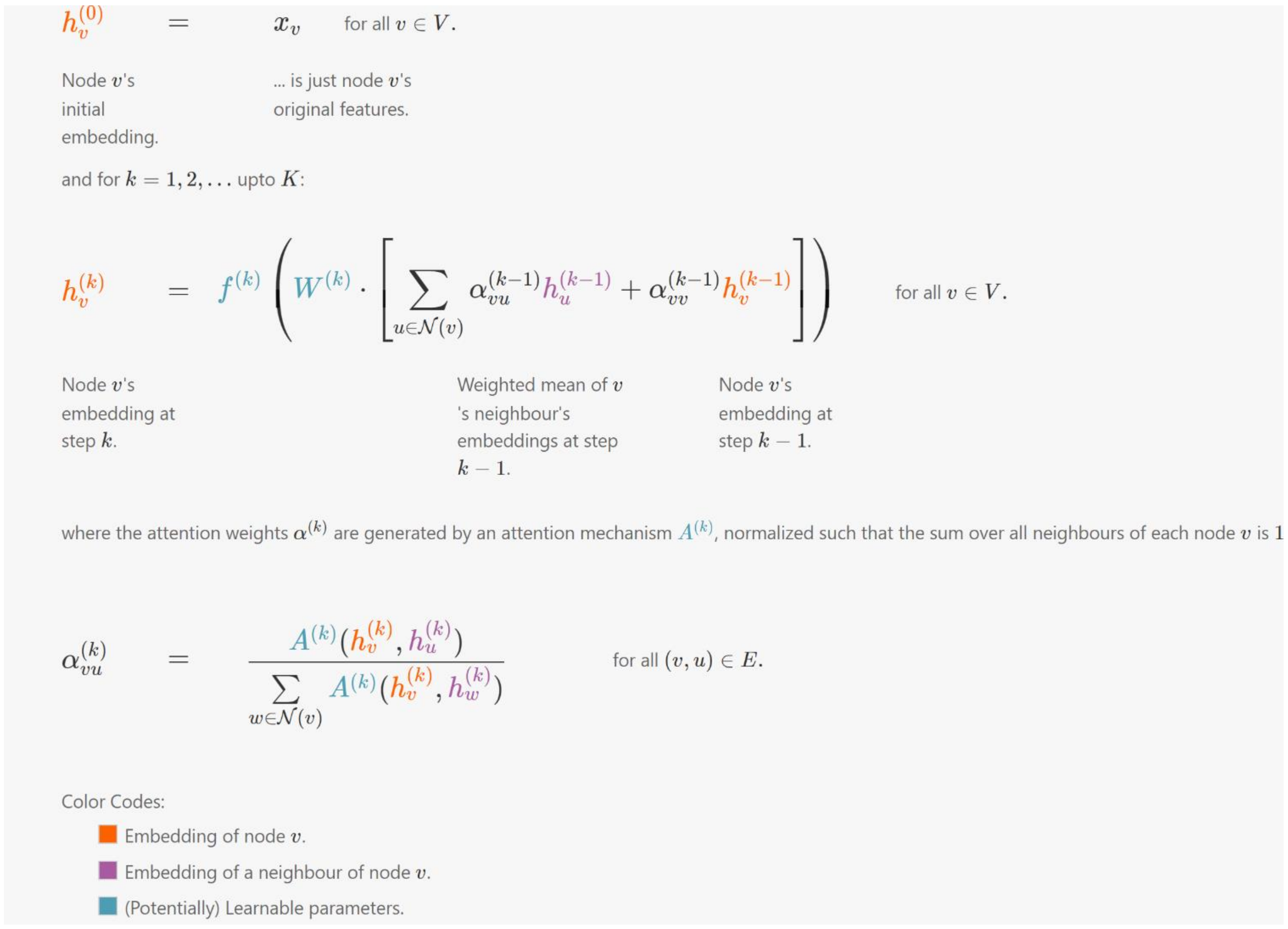
```
from MyGCN2 import MyGCN2 # 2 layer GCN
```

Change the line that creates an object of MyGCN to:

```
    #model = MyGCN(num_features=34,num_hidden=3,num_classes=4)
    model = MyGCN2(num_features=34,num_hidden=3,num_classes=4)
```

Run the program again to see that it still classifies each node correctly.

**Graph Attention Network (GAT):** Graph Attention Networks use the attention mechanism to determine the neighboring node similarity in updating the feature vector in each node as:



$$h_v^{(0)} = x_v \quad \text{for all } v \in V.$$

Node $v$'s initial embedding.

... is just node $v$'s original features.

and for $k = 1, 2, \ldots$ upto $K$:

$$h_v^{(k)} = f^{(k)}\left(W^{(k)} \cdot \left[\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(k-1)} h_u^{(k-1)} + \alpha_{vv}^{(k-1)} h_v^{(k-1)}\right]\right) \quad \text{for all } v \in V.$$

Node $v$'s embedding at step $k$.

Weighted mean of $v$'s neighbour's embeddings at step $k-1$.

Node $v$'s embedding at step $k-1$.

where the attention weights $\alpha^{(k)}$ are generated by an attention mechanism $A^{(k)}$, normalized such that the sum over all neighbours of each node $v$ is 1

$$\alpha_{vu}^{(k)} = \frac{A^{(k)}(h_v^{(k)}, h_u^{(k)})}{\sum_{w \in \mathcal{N}(v)} A^{(k)}(h_v^{(k)}, h_w^{(k)})} \quad \text{for all } (v, u) \in E.$$

Color Codes:

- 🟧 Embedding of node $v$.
- 🟪 Embedding of a neighbour of node $v$.
- 🟦 (Potentially) Learnable parameters.

Graph Attention Networks work better in more complex cases than GCNs. We will program the comparison of the GCN and the GAT on the Cora Citation dataset. This dataset has 2708 nodes in it with each node having 1433 features. The feature vector indicates if the paper has the word appearing in the dictionary of 1433 keywords (1 in the position if the word appears in the paper otherwise 0). The dataset has 7 classes in it according to the category of the paper.

```
0: "Theory",
1: "Reinforcement_Learning",
2: "Genetic_Algorithms",
3: "Neural_Networks",
4: "Probabilistic_Methods",
5: "Case_Based",
6: "Rule_Learning"
```

Create a Python application called CoraGCN. Add a class called MyGCN with the following code in it.

```
from torch_geometric.nn import GCNConv
import torch.nn.functional as F
from torch import nn
import torch

class MyGCN(nn.Module):
    def __init__(self, num_features, num_hidden, num_classes):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GCNConv(num_features, 50)
        self.conv2 = GCNConv(50, num_hidden)
        self.linearnet = nn.Linear(num_hidden, num_classes)

    def forward(self, x, edge_index):
        o1 = self.conv1(x, edge_index)
        o2 = o1.relu()
        o3 = F.dropout(o2, p=0.5, training=self.training) # dropout should only
be used in training
        h = self.conv2(o3, edge_index)
        out = self.linearnet(h)
        return out, h
```

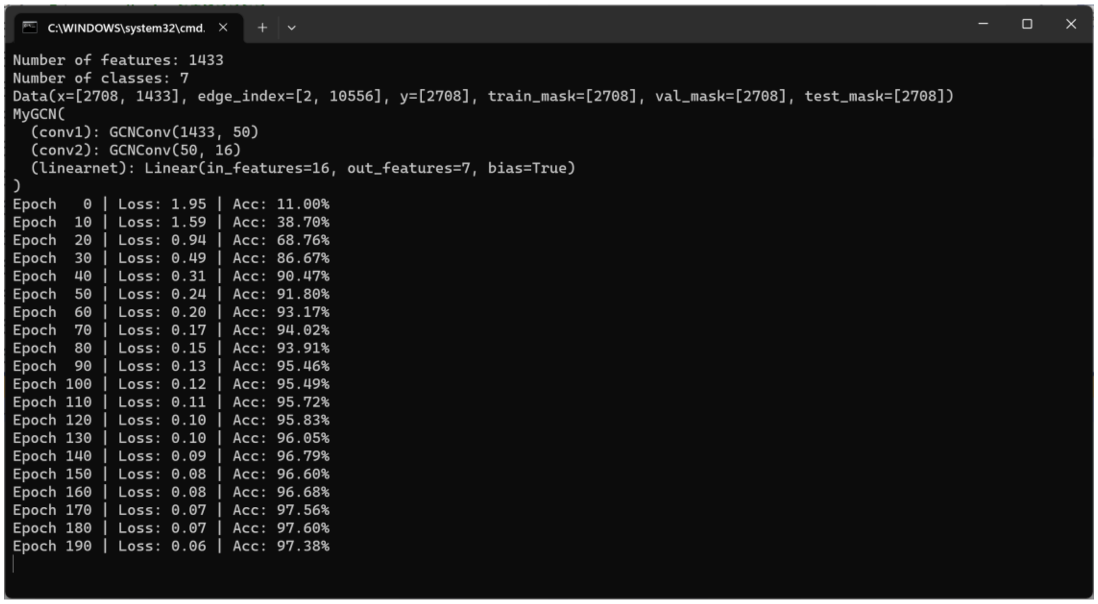
Add another class called MyGATCN with the following code in it.

```
from torch_geometric.nn import GATConv
from torch import nn
import torch
import torch.nn.functional as F

class MyGATCN(nn.Module):
    def __init__(self, num_features, num_hidden, num_classes, heads=6):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GATConv(num_features, 30, heads)
        self.conv2 = GATConv(heads*30, num_hidden, heads)
        self.linearnet = nn.Linear(num_hidden*heads, num_classes)

    def forward(self, x, edge_index):
        x = F.dropout(x, p=0.2, training=self.training)
        x = self.conv1(x, edge_index)
        x = F.elu(x)
        x = F.dropout(x, p=0.2, training=self.training)
        h = self.conv2(x, edge_index)
        out = self.linearnet(h)
        return out, h
```

Type the following code in CoraGCN.py.

```
import sys
from torch_geometric.datasets import Planetoid # Cora Planetoid dataset
from torch_geometric.transforms import NormalizeFeatures
from MyGCN import MyGCN
from MyGATCN import MyGATCN
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import umap
import torch
from matplotlib.colors import ListedColormap

def visualize_TSNE_nodes(h, color):
    z = TSNE(n_components=2).fit_transform(h.detach().cpu().numpy())
    plt.figure(figsize=(10,10))
    plt.title('Visualization of Node Embedding - TSNE')
    plt.xticks([])
    plt.yticks([])
    plt.scatter(z[:, 0], z[:, 1], s=10, c=color, cmap="Set2")
    plt.show()

def visualize_UMAP_nodes(h, color):
    z = umap.UMAP(n_neighbors=5,n_components=2,
                      min_dist=0.3,

metric='correlation').fit_transform(h.detach().cpu().numpy())
    plt.figure(figsize=(10,10))
    plt.title('Visualization of Node Embedding - UMAP')
    plt.xticks([])
    plt.yticks([])
    plt.scatter(z[:, 0], z[:, 1], s=10, c=color, cmap="Set2")
    plt.show()

def compute_accuracy(actual_y, expected_y):
    return (expected_y == actual_y).sum() / len(actual_y)

def plot_embeddings(data, embeddings):
    custom_colors = ['blue', 'yellow', 'green', 'black',
'red','purple','orange']
    custom_cmap = ListedColormap(custom_colors)
    fig = plt.figure(figsize=(12, 12))
    ax = fig.add_subplot(projection='3d')
    ax.patch.set_alpha(0)
    plt.tick_params(left=False,
                    bottom=False,
                    labelleft=False,
                    labelbottom=False)
    ax.scatter(embeddings[:, 0], embeddings[:, 1], embeddings[:, 2],
               s=5, c=data.y, cmap=custom_cmap, vmin=-2, vmax=3)
    plt.show()

def main():
    dataset = Planetoid(root='data/Planetoid',
name='Cora',transform=NormalizeFeatures())
    # 2708 nodes with 1433 features in each node
    # 1 if the word is included in the paper else 0.
    # 7 classes
    # 0: "Theory",
    # 1: "Reinforcement_Learning",
    # 2: "Genetic_Algorithms",
    # 3: "Neural_Networks",
    # 4: "Probabilistic_Methods",
    # 5: "Case_Based",
    # 6: "Rule_Learning"

    print(f'Number of features: {dataset.num_features}')
    print(f'Number of classes: {dataset.num_classes}')

    data = dataset[0]  # first graph object.
    print(data)
    model = MyGCN(num_features=data.num_features,num_hidden=16, num_classes=7)
    #model = MyGATCN(num_features=data.num_features,num_hidden=16, num_classes=7)
print(model)

    model.eval()

    out, _ = model(data.x, data.edge_index)
    #visualize_TSNE_nodes(out, color=data.y)
    #visualize_UMAP_nodes(out, color=data.y)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.02)

    num_epochs = 200

    # train GCN
    model.train()
    for epoch in range(num_epochs):
        optimizer.zero_grad()
        out, h = model(data.x, data.edge_index) # h is learnt embedding
        #print(h.shape)
        loss = criterion(out, data.y)
        acc = compute_accuracy(out.argmax(dim=1), data.y)
        loss.backward()
        optimizer.step()
        if epoch % 10 == 0:
            print(f'Epoch {epoch:>3} | Loss: {loss:.2f} | Acc: {acc*100:.2f}%')
    # plot final learnt embedding for each train node
    plot_embeddings(data, h.detach().cpu().numpy())
    visualize_UMAP_nodes(out, color=data.y)
    visualize_TSNE_nodes(out, color=data.y)
if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Study the above code. If you run the program, the outputs appear as:

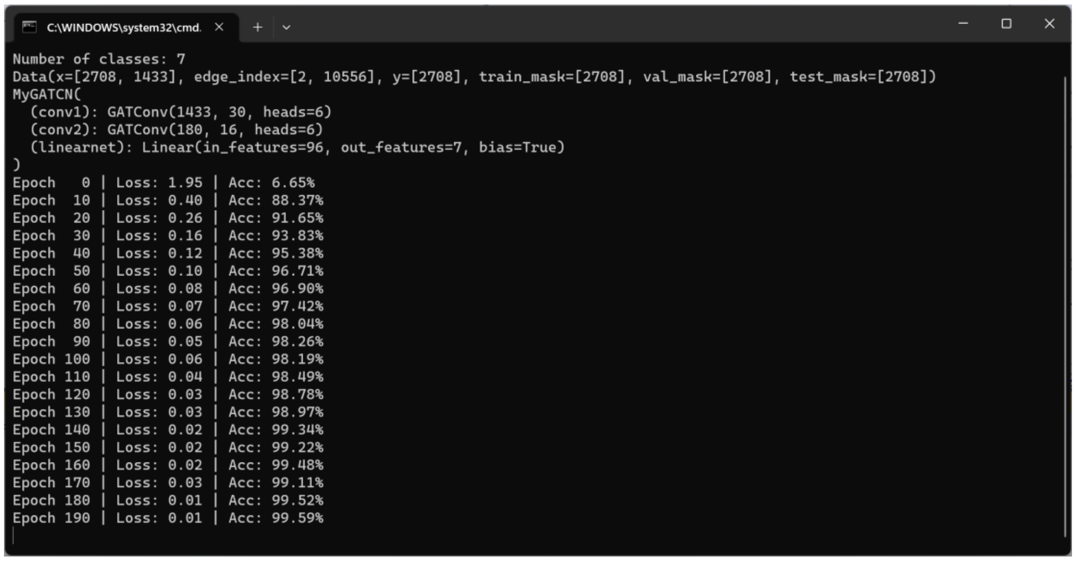


```
Number of features: 1433
Number of classes: 7
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
MyGCN(
  (conv1): GCNConv(1433, 50)
  (conv2): GCNConv(50, 16)
  (linearnet): Linear(in_features=16, out_features=7, bias=True)
)
Epoch   0 | Loss: 1.95 | Acc: 11.00%
Epoch  10 | Loss: 1.59 | Acc: 38.70%
Epoch  20 | Loss: 0.94 | Acc: 68.76%
Epoch  30 | Loss: 0.49 | Acc: 86.67%
Epoch  40 | Loss: 0.31 | Acc: 90.47%
Epoch  50 | Loss: 0.24 | Acc: 91.80%
Epoch  60 | Loss: 0.20 | Acc: 93.17%
Epoch  70 | Loss: 0.17 | Acc: 94.02%
Epoch  80 | Loss: 0.15 | Acc: 93.91%
Epoch  90 | Loss: 0.13 | Acc: 95.46%
Epoch 100 | Loss: 0.12 | Acc: 95.49%
Epoch 110 | Loss: 0.11 | Acc: 95.72%
Epoch 120 | Loss: 0.10 | Acc: 95.83%
Epoch 130 | Loss: 0.10 | Acc: 96.05%
Epoch 140 | Loss: 0.09 | Acc: 96.79%
Epoch 150 | Loss: 0.08 | Acc: 96.60%
Epoch 160 | Loss: 0.08 | Acc: 96.68%
Epoch 170 | Loss: 0.07 | Acc: 97.56%
Epoch 180 | Loss: 0.07 | Acc: 97.60%
Epoch 190 | Loss: 0.06 | Acc: 97.38%
```

The above output is when GCN is being used as the model. If you replace the model with GAT by modifying the lines in the main as:

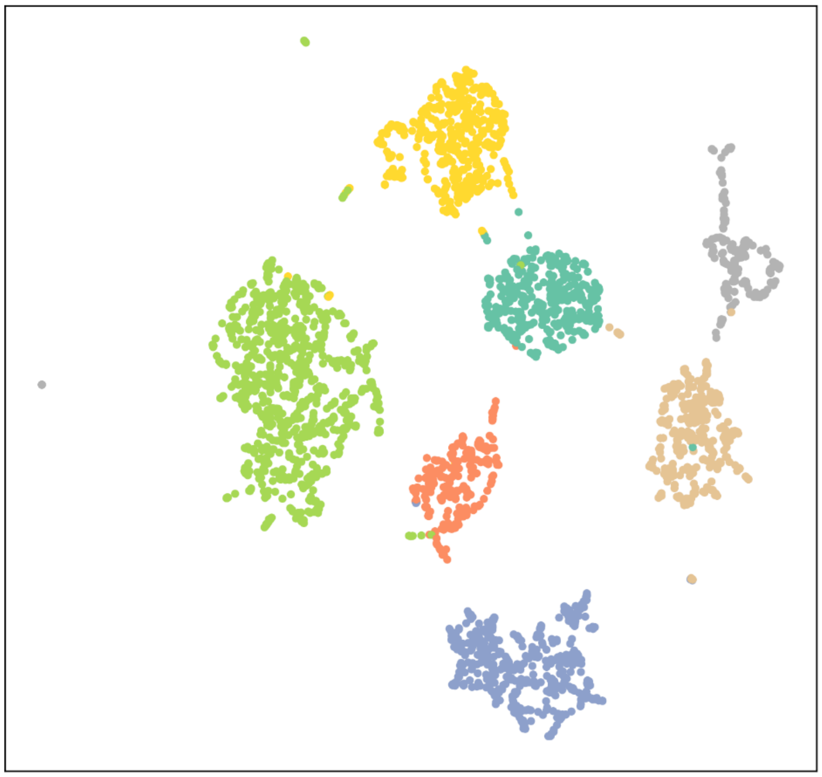
```
   #model = MyGCN(num_features=data.num_features,num_hidden=16, num_classes=7)
    model = MyGATCN(num_features=data.num_features,num_hidden=16, num_classes=7)
```

Now if you run the program, output will appear as:



```
Number of classes: 7
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[2708])
MyGATCN(
  (conv1): GATConv(1433, 30, heads=6)
  (conv2): GATConv(180, 16, heads=6)
  (linearnet): Linear(in_features=96, out_features=7, bias=True)
)
Epoch   0 | Loss: 1.95 | Acc: 6.65%
Epoch  10 | Loss: 0.40 | Acc: 88.37%
Epoch  20 | Loss: 0.26 | Acc: 91.65%
Epoch  30 | Loss: 0.16 | Acc: 93.83%
Epoch  40 | Loss: 0.12 | Acc: 95.38%
Epoch  50 | Loss: 0.10 | Acc: 96.71%
Epoch  60 | Loss: 0.08 | Acc: 96.90%
Epoch  70 | Loss: 0.07 | Acc: 97.42%
Epoch  80 | Loss: 0.06 | Acc: 98.04%
Epoch  90 | Loss: 0.05 | Acc: 98.26%
Epoch 100 | Loss: 0.06 | Acc: 98.19%
Epoch 110 | Loss: 0.04 | Acc: 98.49%
Epoch 120 | Loss: 0.03 | Acc: 98.78%
Epoch 130 | Loss: 0.03 | Acc: 98.97%
Epoch 140 | Loss: 0.02 | Acc: 99.34%
Epoch 150 | Loss: 0.02 | Acc: 99.22%
Epoch 160 | Loss: 0.02 | Acc: 99.48%
Epoch 170 | Loss: 0.03 | Acc: 99.11%
Epoch 180 | Loss: 0.01 | Acc: 99.52%
Epoch 190 | Loss: 0.01 | Acc: 99.59%
```

Visualization of Node Embedding - UMAP



Visualization of Node Embedding - TSNE