Graph Convolutional Networks

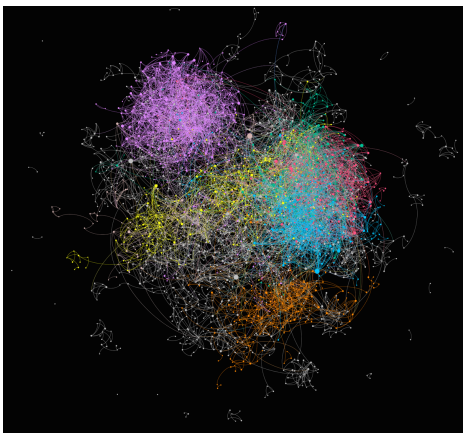
Giovanni Bianco

September 2024

1 Graph Data

Graphs are highly versatile mathematical structures consisting of nodes and edges, useful for modeling a wide range of scenarios. For instance, a school class, a collection of movies, and various networks like social media platforms and web browsers can all be represented as graphs. In a graph, edges provide information about the relationships between nodes. For example, in a graph representing Instagram, nodes could symbolize individual profiles, with edges indicating followership. Moreover, each node can carry additional data, often encapsulated in vectors, which, in the previous example, might include attributes like the age, gender, and name of a user in a social network.

Data scientists utilize different algorithms tailored to the specific nature of the data to extract meaningful information. Just as convolutional neural networks have revolutionized image processing, the concept of Graph Convolutional Networks (GCNs) was introduced to extend these principles to graph data. GCNs leverage the structural information of graphs to perform complex operations, allowing for effective analysis and insights into interconnected data.



2 Understanding Graph Convolutional Networks (GCNs)

A Graph Convolutional Network (GCN) is a type of convolutional neural network that operates on graphs. This framework allows us to work with data in a non-Euclidean domain and capture the connectivity patterns among data points effectively.

2.1 GCN Layer

One of the fundamental operations in a GCN is the **GCN layer**, which processes features from nodes and their neighbors. The operation at each layer ℓ can be described mathematically as:

$$\mathbf{x}_v^{(\ell+1)} = \mathbf{W}^{(\ell+1)} \sum_{w \in \mathcal{N}(v) \cup \{v\}} \frac{1}{c_{w,v}} \cdot \mathbf{x}_w^{(\ell)}$$

Here, $\mathbf{W}^{(\ell+1)}$ is a trainable weight matrix with dimensions $[\text{num\_output\_features}, \text{num\_input\_features}]$, and $c_{w,v}$ is a normalization coefficient defined for each edge, which stabilizes the learning process by preventing the scale of the feature vectors from growing too large and to be too much dependent on the degree of a node. The normalization coefficient $c_{w,v}$ is calculated as:

$$c_{w,v} = \sqrt{(d_w + 1)(d_v + 1)}$$

where d_w and d_v are the degrees of nodes w and v, respectively, plus one, to account for node v itself.

2.2 Graph Convolution Process

In the graph convolution process:

1. **Normalization:** Each neighbor's features ($\mathbf{x}_w^{(\ell)}$) are divided by $c_{w,v}$, effectively normalizing the data.
2. **Aggregation:** The normalized features of all neighbors and the node itself are summed together, resulting in a vector of the same dimensionality as the original features.
3. **Transformation:** This aggregated feature vector is then multiplied by the weight matrix $\mathbf{W}^{(\ell)}$.

2.3 Matrix Representation

The entire process can be represented more compactly using matrix operations:

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}\right)$$

Here, $\hat{\mathbf{A}}$ is the adjacency matrix with self-loops, $\hat{\mathbf{D}}$ is the diagonal degree matrix of $\hat{\mathbf{A}}$, $\mathbf{H}^{(\ell)}$ represents the feature matrix at layer ℓ, and σ is a non-linear activation function, such as ReLU.

2.4 Model structure

We define a Graph Convolutional Network (GCN) model with three layers of graph convolution operations. The first layer transforms the features of the nodes from their original dimensionality to a dimension of 4. The second layer keeps the dimensionality at 4, while the third layer reduces it to 2 dimensions, allowing us to visualize the nodes in 2D space. A linear classifier is then applied to the 2D embeddings to predict the class of each node. The architecture is implemented as follows:

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

class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        torch.manual_seed(1234)
        self.conv1 = GCNConv(dataset.num_features, 4)
        self.conv2 = GCNConv(4, 4)
        self.conv3 = GCNConv(4, 2)
        self.classifier = Linear(2, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        h = h.tanh()
        out = self.classifier(h)
        return out, h

model = GCN()
print(model)
```

2.5 Training and Loss Calculation

In our semi-supervised learning setup, we only calculate the loss on a subset of the nodes, called the train subset, using a mask applied to both the nodes and the labels. This is because we want to avoid overfitting by using only a part of the nodes for training, while updating the embeddings of all the nodes.

This allows us to utilize the information contained in all the nodes, even the non-training ones.

We define the loss function as Cross Entropy Loss and use the Adam optimizer for updating the model's parameters. The training process is implemented as follows:

```
import time
from IPython.display import Javascript
display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight:
↪  430})'''))

model = GCN()
criterion = torch.nn.CrossEntropyLoss()  # Define loss criterion
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)  # Define optimizer

def train(data):
    optimizer.zero_grad()  # Clear gradients
    out, h = model(data.x, data.edge_index)  # Forward pass
    loss = criterion(out[data.train_mask], data.y[data.train_mask])  # Compute
    ↪  loss for training nodes
    loss.backward()  # Backpropagate
    optimizer.step()  # Update parameters
    return loss, h

```

2.6 Training Loop

Each time the `train` function is called, a step is taken. The number of steps is determined by the number of epochs chosen. Every 100 epochs, the current embeddings are visualized in 2D space, with each color corresponding to a class. The training loop is defined as follows:

```
for epoch in range(3000):
    loss, h = train(data)
    if epoch % 100 == 0:
        visualize_embedding(h, color=data.y, epoch=epoch, loss=loss)
        time.sleep(0.3)
```

This training loop allows us to track the progress of the model and visualize the embeddings in real-time, ensuring that we are effectively training the GCN.

3 Implementing GCN on Karate Data

In this section, we implement a Graph Convolutional Network (GCN) on the Karate Club dataset. This dataset consists of 34 nodes, each represented with 34 features. In this case, the feature matrix is simply the adjacency matrix of the graph. The dataset contains 4 distinct classes.

3.1 Loading the Dataset

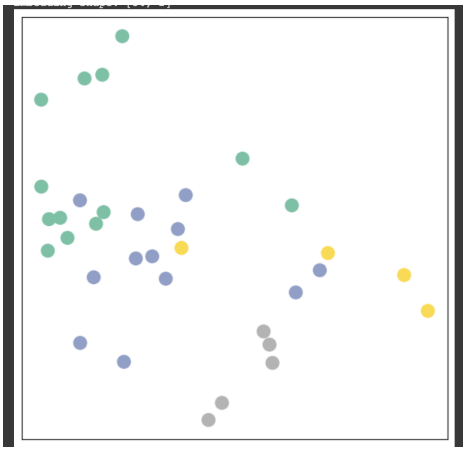
We begin by importing the Karate Club dataset and displaying the key properties of the dataset.

```python
# Import necessary libraries and the Karate Club dataset
from torch_geometric.datasets import KarateClub

# Load the dataset
dataset = KarateClub()

# Display dataset details
print(f'Dataset: {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}')
```

The output shows that there are 34 nodes, each with 34 features, and the dataset has 4 classes. In this specific case, the feature matrix corresponds to the adjacency matrix of the graph.

3.2 Initial Embedding Without Training

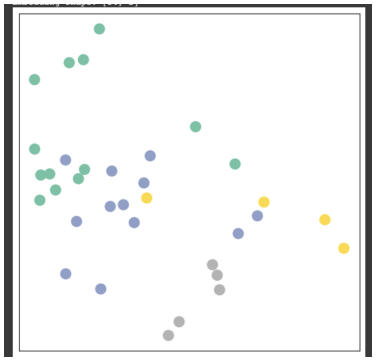
To begin, we visualize the initial embeddings of the nodes in the graph before any training is conducted. This provides insight into how the Graph Convolutional Network (GCN) positions the nodes in the embedding space, solely based on the graph structure.



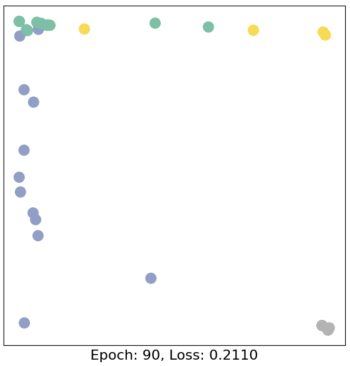
As shown, nodes belonging to the same cluster are already relatively close to one another. This is due to the inherent inductive bias of GCNs: neighboring nodes in the original graph tend to be embedded close together, even in the absence of learned features. This behavior stems from the GCN's mechanism, which aggregates information from neighboring nodes.

3.3 Training with Different Parameters

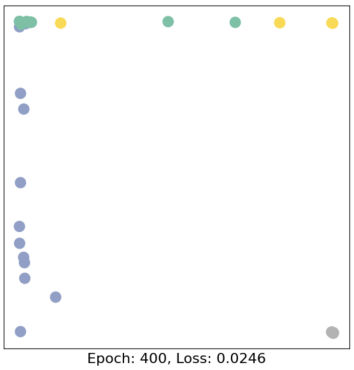
Next, we explore the effect of training the GCN model over multiple epochs, adjusting various parameters to observe changes in the node embeddings.



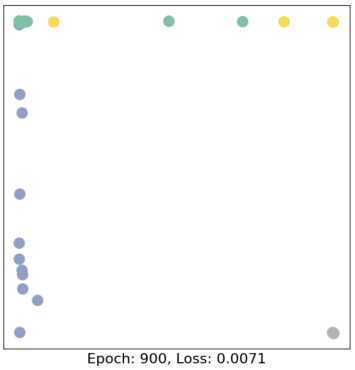
(a) Initial Embedding (No Training)



(b) After 90 Epochs



(c) After 400 Epochs



(d) After 900 Epochs

3.3.1 Training with 90 Epochs

After 90 epochs of training, the communities become more distinct compared to the initial embedding. However, there is still room for improvement in terms of separating the clusters more clearly.

3.3.2 Training with 400 Epochs

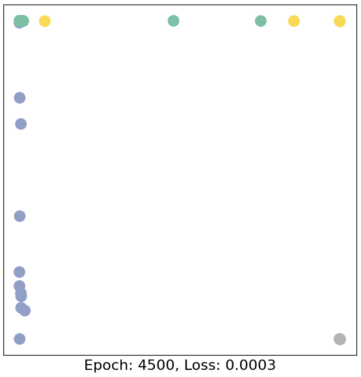
Increasing the number of epochs to 400 results in more cohesive clusters, especially for the blue class, whose nodes are now positioned closer together. However, some nodes remain misclassified—particularly in the green cluster, where a few blue and yellow nodes are still incorrectly embedded.

3.3.3 Training with 900 Epochs

Despite increasing the number of training epochs to 900, the improvement remains marginal. The model still struggles to clearly separate the green cluster, and a few misclassified nodes persist across the other clusters.

3.3.4 Training with 4500 epopchs and 0.01 of learning rate

To further refine the model, we reduce the learning rate to 0.01 and significantly increase the number of training epochs to 5000.



Unfortunately, even with a smaller learning rate and more extensive training, there is no noticeable improvement. The clusters remain similar to previous attempts, indicating that further adjustments to other hyperparameters may be necessary to achieve better classification results.

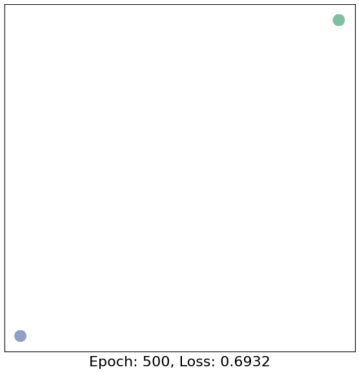
3.4 Why Does Misclassification Occur?

The misclassification likely arises due to the inductive bias we discussed earlier. Upon inspecting the initial graph, we observe that a yellow node, which is potentially the misclassified one, is more connected to green nodes than to other yellow nodes. This connection pattern leads to confusion in classification, as the GCN tends to embed neighboring nodes closely together, regardless of their actual labels.

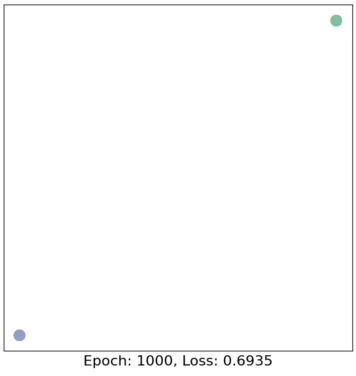
Additionally, the misclassification may also be influenced by the features of the node itself. In our case, the features represent the nodes' connections, meaning the yellow node, having more green connections, may appear more similar to a green node than to other yellow nodes, leading to further confusion.

3.5 Effect of Increasing the Learning Rate

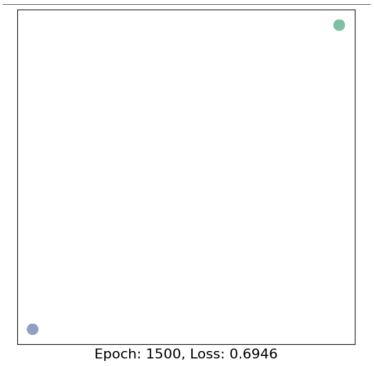
When we increase the learning rate to a large value, such as 1, the model fails to converge, and the nodes essentially "disappear" from the visualized embedding space. This can be observed in the following visualization:



This issue likely occurs because the model takes excessively large steps during optimization, causing the weights to diverge rather than converge to a meaningful solution. This divergence can also be detected by observing the loss function, which increases drastically after certain training steps. For example, between steps 1000 and 1500, the loss increases significantly, indicating that the model is not learning effectively.



(a) Embedding at step 1000.



(b) Embedding at step 1500.

This behavior underscores the importance of selecting an appropriate learning rate to ensure proper convergence during training.

4 Friend recomendation on KarateClub data

In this section, we explore how to recommend friends to each node based on cosine similarity. The similarity is calculated using the node features, which represent the connections each node has. Nodes with similar connections are more likely to be suggested as potential friends.

4.1 Cosine Similarity with the Feature Matrix

We start by defining a cosine similarity function. The following is vectorized version that leverages on numpy to speed up computationa, but that only works if the matrix is symmetric. This makes it applicable to the original feature matrix (34 nodes, 34 features), but not to the embedded matrix after training (34 nodes, 2 features).

```
# This only works if the matrix x is symmetric, meaning we cannot use it with the embedding matrix
def cosine_similarity(x):
    # Step 1: Normalize each row to have unit norm
    norms = np.linalg.norm(x, axis=1, keepdims=True)
    x_normalized = x / norms

    # Step 2: Compute cosine similarity matrix
    csim = np.dot(x_normalized, x_normalized.T)
    return csim

csim = cosine_similarity(x)
print(csim)
```

This function returns a matrix csim, where csim$[i, j]$ gives the cosine similarity between node i and node j.

4.2 Friend Suggestions Based on Cosine Similarity

Next, we define a function to suggest 10 potential friends based on the cosine similarity matrix. The suggestions are ranked from the node with the highest cosine similarity to the lowest.

```
def suggestions(node, csim):
    x = csim  # Assuming csim is defined elsewhere as a cosine similarity matrix
    # Use argsort to get indices sorted by their cosine similarity values in ascending order
    indices = np.argsort(x[node])

    # Select the last 10 indices (the ones with the highest values) and reverse to make highest first
    top_indices = indices[-10:][::-1]

    # Print top indices for debug purposes
    for i in top_indices:
```

```
        print(f'node: {i}, cosine similarity: {x[node][i]:.4f}')  # formatted
        ↪  to 4 decimal places

suggestions(10, csim)
```

This function outputs 10 friend suggestions for the specified node, ranked by cosine similarity. The highest-ranked suggestion will always be the input node itself, as it has a perfect cosine similarity with itself.

For node 10, the result is as follows:

```
index: 10, cosine similarity: 1.0000
index:  6, cosine similarity: 0.8660
index: 11, cosine similarity: 0.5774
index: 16, cosine similarity: 0.4082
index: 21, cosine similarity: 0.4082
index: 12, cosine similarity: 0.4082
index: 17, cosine similarity: 0.4082
index:  4, cosine similarity: 0.3333
index: 19, cosine similarity: 0.3333
index:  5, cosine similarity: 0.2887
```

Of course the node with highest cosine similarity is, trivially, node 10 itself. For a more meaningful insights we might think to remove it.

4.3 Using the Embedding Instead of the Feature Matrix

After training the GCN model with a learning rate of 0.1 and 500 epochs, we need a new function to compute cosine similarity because the embedding matrix is no longer symmetric.

```
def ce_similarity(vec1, vec2):
    cos_sim = np.dot(vec1, vec2) / (np.linalg.norm(vec1) *
    ↪  np.linalg.norm(vec2))
    return cos_sim

# Assuming h is the matrix of embeddings where each row is a vector
h = h.detach().numpy()
# Initialize dictionary to hold cosine similarities
csim = {}
for i in range(h.shape[0]):  # Iterate over each vector (row)
    for j in range(h.shape[0]):
        csim[(i, j)] = c_similarity(h[i, :], h[j, :])

# Create for each row a list of the 10 most similar rows
top = {}
for i in range(h.shape[0]):
    top_i = np.zeros([10, 2]) - 1  # Initialize with -1 to handle minimum
    ↪  correctly
    for j in range(h.shape[0]):
        current_similarity = csim[(i, j)]
```

```
        if current_similarity > np.min(top_i[:, 1]):
            min_index = np.argmin(top_i[:, 1])
            top_i[min_index] = [j, current_similarity]
    top[i] = top_i

results = top[10]
for key, value in results:
    print(f'node: {key}, cosine similarity: {value[1]:.4f}')  # formatted to 4
    ↪  decimal places
```

For node 10, the result is as follows:

```
node: 10, cosine similarity: 1.0000
node:  4, cosine similarity: 1.0000
node:  5, cosine similarity: 1.0000
node:  6, cosine similarity: 1.0000
node: 16, cosine similarity: 1.0000
node: 11, cosine similarity: 0.0119
node:  0, cosine similarity: 0.0007
node: 24, cosine similarity: -0.0007
node: 25, cosine similarity: -0.0009
node:  3, cosine similarity: -0.0423
```

4.4 Comparison of Results

The results are similar, but the order of the friend suggestions and the cosine similarities differs when using the feature matrix versus the embedding matrix. This difference is expected since the embedding process organizes the nodes more distinctly based on their features and class. Nodes with initially similar features but different classes are now further apart, While nodes with different features but the same class are now much closer, The high cosine cimilarity values probably corresponds to nodes that belongs to the same class as node 10, while the other selected nodes are nodes with similar features belonging to differet classes.

Before Embedding	**After Embedding**
node: 10, cosine similarity: 1.0000	node: 10, cosine similarity: 1.0000
node: 6, cosine similarity: 0.8660	node: 4, cosine similarity: 1.0000
node: 11, cosine similarity: 0.5774	node: 5, cosine similarity: 1.0000
node: 16, cosine similarity: 0.4082	node: 6, cosine similarity: 1.0000
node: 21, cosine similarity: 0.4082	node: 16, cosine similarity: 1.0000
node: 12, cosine similarity: 0.4082	node: 11, cosine similarity: 0.0119
node: 17, cosine similarity: 0.4082	node: 0, cosine similarity: 0.0007
node: 4, cosine similarity: 0.3333	node: 24, cosine similarity: -0.0007
node: 19, cosine similarity: 0.3333	node: 25, cosine similarity: -0.0009
node: 5, cosine similarity: 0.2887	node: 3, cosine similarity: -0.0423

The discrepancies between the two highlight how the training process separates the nodes into more distinct clusters, improving the recommendation process based on class information.

4.5 Further improvements

I did not have time to filter the data in order to only consider non-connected nodes for recommendation purposes. Since nodes that are already connected represent existing friendships, it would not make sense to recommend them as new friends. A potential improvement for future iterations would be to incorporate this filtering step, ensuring that recommendations are made only for nodes that are not already connected.

5 Research Network Data

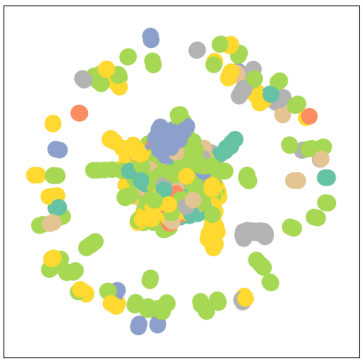
5.1 Analysis of the GCN

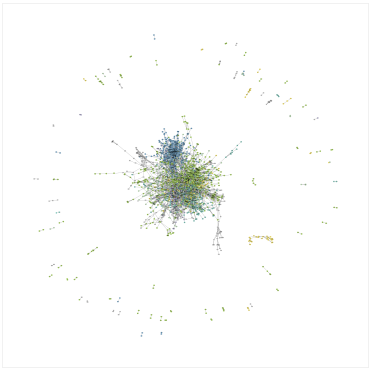
We begin by importing the dataset:

```
from torch_geometric.datasets import Planetoid
dataset = Planetoid("", "Cora")
```

The dataset contains 2708 nodes, each with 1433 features. Unlike the Karate Club dataset, this time the feature matrix is not symmetric,leading to the hypothesis that it no longer matches the adjacency matrix. Upon closer inspection, we can observe that the feature matrix consists solely of 0s and 1s, representing binary relationships and further analysis reveals that the feature matrix still effectively represents the adjacency matrix.

By visualizing the graph, we can see that the nodes are divided into two distinct groups. One is a large connected component, composed of 1433 nodes, while the other consists of the remaining 1275 nodes (2708 - 1433), which are not connected. Since these nodes have no connections, their corresponding columns in the adjacency matrix have been deleted.



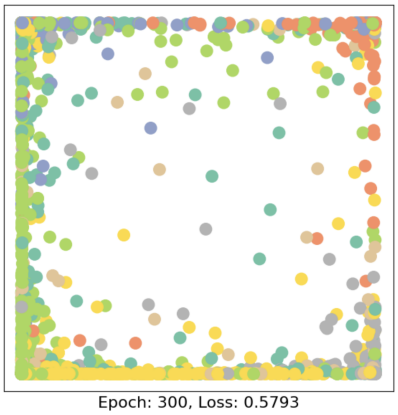


Given the higher complexity of this graph compared to previous examples, the initial embedding appears quite messy, as expected due to the larger number of nodes and features.

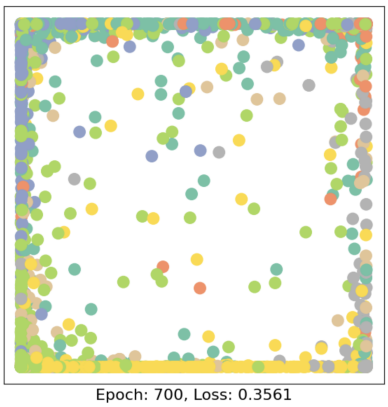
5.2 Analysis of Training Progress

Next, we train the GCN model with a learning rate of 0.1 and observe the results at different stages of training: We observe that the model's performance improves steadily up to approximately 2000 training steps. Beyond this point, however, the gains become negligible. Interestingly, as training continues, the embeddings start to become more disorganized, suggesting signs of overfitting or instability in the optimization process. This behavior can be attributed to the presence of the community of unconnected nodes. Since these nodes have

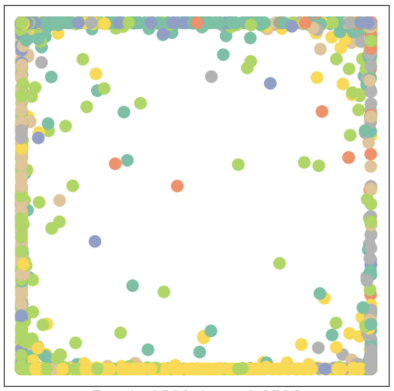
virtually no features, the model struggles to learn meaningful representations for them, even if they belong to a specific class.



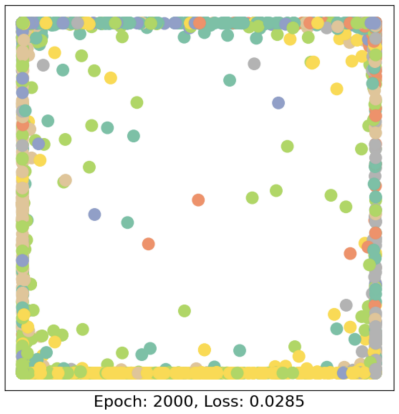
Epoch: 300, Loss: 0.5793



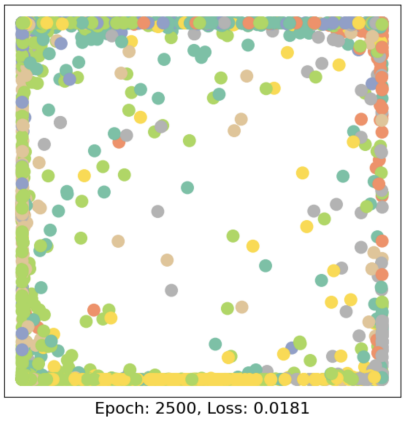
Epoch: 700, Loss: 0.3561



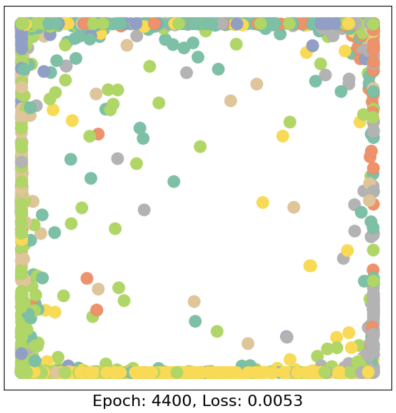
Epoch: 1500, Loss: 0.0516



Epoch: 2000, Loss: 0.0285



Epoch: 2500, Loss: 0.0181



Epoch: 4400, Loss: 0.0053

6 Recommendation

With the normal feature matrix, the results for node 10 are:

```
index: 10, cosine similarity: 1.0000
index: 1140, cosine similarity: 0.7071
index: 1800, cosine similarity: 0.4082
index: 1986, cosine similarity: 0.0877
index: 306, cosine similarity: 0.0811
index: 899, cosine similarity: 0.0000
index: 906, cosine similarity: 0.0000
index: 905, cosine similarity: 0.0000
index: 904, cosine similarity: 0.0000
index: 903, cosine similarity: 0.0000
```

6.1 Trying with the Embedding

Trained with 2000 epochs and 0.1 of learning rate:

```
node: 1692, cosine similarity: 1.0000
node: 1848, cosine similarity: 1.0000
node: 2671, cosine similarity: 1.0000
node: 62, cosine similarity: 1.0000
node: 65, cosine similarity: 1.0000
node: 94, cosine similarity: 1.0000
node: 415, cosine similarity: 1.0000
node: 603, cosine similarity: 1.0000
node: 716, cosine similarity: 1.0000
node: 851, cosine similarity: 1.0000
```

The cosine similarity increased significantly, likely because all these nodes belong to the same class. As a result, they become very close in the embedding space due to the extended training period.

Additionally, reducing the number of features from 1400 to 2 leads to a substantial loss of information. Since the goal is to cluster nodes from the same class together, longer training shifts the focus towards class labels rather than individual features. With such a large number of nodes, in the reduced **plane**, many nodes end up positioned very close to each other, which naturally results in high similarity scores.

7 Task 2: More Advanced Graph Neural Networks

In this task, we explore two popular graph neural network architectures: **GraphSAGE** and **Graph Attention Networks (GATs)**. Both architectures are designed to handle large-scale graph data, but they use different mechanisms to aggregate and process node information. Below, we provide an in-depth look at both architectures, focusing on their mechanisms, use cases, and key mathematical formulations.

7.1 GraphSAGE

GraphSAGE is a scalable and efficient method for generating node embeddings for large graphs. Unlike traditional methods that rely on full-batch aggregation of node features, GraphSAGE trains considering only a fixed number of neighbors for each node. Differently from basic GCN it can also use different aggregation functions such as mean, LSTM, or max-pooling.

Use Cases GraphSAGE excels at generating embeddings for previously unseen nodes by leveraging both the node's local features and the structure of the graph. This makes it particularly well-suited for dynamic and large-scale graphs, such as those encountered in social networks or biological networks.

Limitations The effectiveness of GraphSAGE heavily depends on the choice of the aggregator and the sampling strategy. If the sampling fails to capture the diversity of the neighborhood, the quality of the learned embeddings can degrade.

7.2 Graph Attention Networks (GATs)

Graph Attention Networks (GATs) introduce an attention mechanism to graph neural networks, allowing the model to assign different levels of importance to each node within a neighborhood. This enables more fine-grained, dynamic feature integration and has proven effective in a variety of graph-based tasks. The attention mechanism used in Graph Attention Networks (GATs) is indeed based on the Bahdanau attention (also known as additive attention), which differs from the attention mechanism used in Transformers, known as scaled dot-product attention.

7.2.1 Mathematical Formulation

Given a set of node features $\mathbf{h}_i \in \mathbb{R}^F$ where F is the feature dimension, the goal is to transform these features into a new feature space $\mathbb{R}^{F'}$ using a learned weight matrix $\mathbf{W}$.

Attention Mechanism The attention mechanism computes attention coefficients between a node i and its neighbors j as follows:

$$e_{ij} = a(\mathbf{W}\mathbf{h}_i, \mathbf{W}\mathbf{h}_j)$$

These raw coefficients are then normalized using the softmax function:

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

Complete Attention Formula To introduce non-linearity, we use a **LeakyReLU** activation function:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_k]))}$$

Here, $\mathbf{a}$ is a learnable weight vector, and $||$ represents the concatenation operation.

Embedded vector Finally, the new feature representation for node i is obtained by applying the learned attention coefficients to the features of its neighbors:

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\mathbf{h}_j\right)$$

Multi-Head Attention To increase the model's capacity and stabilize training, GATs often employ multi-head attention. The feature update for node i with multi-head attention is given, in the hidden layers, by:

$$\mathbf{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \mathbf{h}_j\right)$$

where $\mathbf{h}'_i$ will consist of $K \times F'$ features for each node.

If we perform it on the final layer, we want a vector with F' features, so:

$$\mathbf{h}'_i = \sigma\left(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \mathbf{h}_j\right)$$

8 Implementation

8.1 GraphSAGE

```python
from torch_geometric.nn import SAGEConv, to_hetero, GCNConv

class GNNEncoder(torch.nn.Module):
    def __init__(self, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = SAGEConv((-1, -1), hidden_channels)
        self.conv2 = SAGEConv((-1, -1), out_channels)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index)
        return x

class EdgeDecoder(torch.nn.Module):
    def __init__(self, hidden_channels):
        super().__init__()
        self.lin1 = torch.nn.Linear(2 * hidden_channels, hidden_channels)
        self.lin2 = torch.nn.Linear(hidden_channels, 1)

    def forward(self, z_dict, edge_label_index):
        row, col = edge_label_index
        z = torch.cat([z_dict['user'][row], z_dict['movie'][col]], dim=-1)

        z = self.lin1(z).relu()
        z = self.lin2(z)
        return z.view(-1)

class Model(torch.nn.Module):
    def __init__(self, hidden_channels):
        super().__init__()
        self.encoder = GNNEncoder(hidden_channels, hidden_channels)
        self.encoder = to_hetero(self.encoder, data.metadata(), aggr='sum')
        self.decoder = EdgeDecoder(hidden_channels)

    def forward(self, x_dict, edge_index_dict, edge_label_index):
        z_dict = self.encoder(x_dict, edge_index_dict)
        return self.decoder(z_dict, edge_label_index)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = Model(hidden_channels=32).to(device)

print(model)
```

The `GNNEncoder` class implements a two-layer Graph Neural Network (GNN) encoder using `SAGEConv` layers from the PyTorch Geometric library. This class is responsible for learning node embeddings based on graph structure and node features.

The constructor method, `__init__`, accepts two parameters: `hidden_channels`

and `out_channels`, which define the dimensionality of the hidden and output layers, respectively. In this method, two `SAGEConv` layers are initialized. The first layer transforms the input node features into a space of `hidden_channels` dimensions. The second layer further transforms the resulting node embeddings into a space of `out_channels` dimensions.

The `forward` method takes two inputs: the node feature matrix `x` and the edge information `edge_index`, which defines the graph's structure. The node features are first passed through the initial `SAGEConv` layer, followed by a ReLU activation function to introduce non-linearity. The output is then passed through the second `SAGEConv` layer to produce the final learned node embeddings.

8.2 GAT

The implementation is very similar to the one of GraphSAGE

```
class GNNEncoder(torch.nn.Module):
    def __init__(self, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = GATConv((-1, -1), hidden_channels, add_self_loops=False)
        # IMPORTANT: I had to set self-loops to False to avoid an error
        self.conv2 = GATConv(hidden_channels, out_channels,
        ↪  add_self_loops=False)

    def forward(self, x, edge_index):
        # Implement the forward pass of the GNN encoder.
        x = self.conv1(x, edge_index).relu()  # Apply the first GAT layer and
        ↪  ReLU activation
        x = self.conv2(x, edge_index)         # Apply the second GAT layer
        return x
```

It is important to make sure to set the `self_loop=False` when using layers. Otherwise, you may encounter errors related to self-loops that could interfere with the graph's structure. The rest of the implementation is the same as the one showed before.

9 Results on Movielens

I inputed the following movies and the following reviews:

- **The Shining** : 4.4
- **In Time** :3.8
- **American Psycho** : 4.3
- **Shutter Island** :5.5
- **Eternal Sunshine of the Spotless Mind**: 4.8

The movie suggested by the model trained using **GraphSAGE** is **Man in the White Suit**, with the following importance given to each of my imputed movies:

- In Time (2011) 0.086657975
- Shutter Island (2010) 0.084278290
- Eternal Sunshine of the Spotless Mind (2004) 0.070820077
- The Shining (1997) 0.041193223

Instead the movie suggested by the model trained using **GAT** is **Godzilla 1985: The Legend Is Reborn (Gojira)**, with the following importance given to each of my imputed movies:

- The Shining (1997) 0.097372542
- In Time (2011) 0.088697740
- American Psycho (2000) 0.042602697
- Shutter Island (2010) 0.022686744
- Eternal Sunshine of the Spotless Mind (2004) 0.019592901