HW3 Report:

Daniel Dahan - 345123624
Simon Bellilty - 345233563

Table of Contents

Introduction:

In this exercise, we focused on using machine learning algorithms and statistical tools for network analysis on a network of academic citations. Specifically, we aimed to predict the categories or research fields of scholarly articles using a graph neural network (GNN). The motivation behind this task is to develop automatic tools that can help sort and make articles accessible to researchers, especially in the context of the increasing pace of article publications.

Analyzing:

To begin our analysis, we examined the characteristics of the academic citation network and the dataset used for training the graph neural network (GNN):

- The academic citation network is represented as a **directed graph** with 100000 nodes and 444288 edges. This means there are 100000 articles represented as nodes in the graph, and the edges represent the citations between these articles. Additionally, the graph contains **isolated nodes**, which are articles that do not have any citations to or from other articles.
- The dataset used for training the GNN is an instance of the HW3Dataset class, containing one graph with 100000 nodes, 128 features per node, and 40 classes for prediction.

The dataset's data object contains the following attributes:

x: Feature vectors for each node, with a shape of [100000, 128]. These features represent the characteristics or attributes associated with each article in the graph.
edge_index: Graph edge connectivity, with a shape of [2, 444288]. It indicates the indices of the nodes that are connected by edges in the graph.
y: Node-level target/label values, with a shape of [100000, 1]. This represents the ground-truth labels or categories of the articles.
node_year: Year associated with each node, with a shape of [100000, 1].
train_mask: Indices for training nodes, with a shape of [80000]. These indices indicate which nodes are used for training the model.
val_mask: Indices for validation nodes, with a shape of [20000]. These indices represent the nodes used for validating the model during training.

Preprocess:

The data preprocessing involves two steps:

1. Feature Normalization:
   The features are normalized by subtracting the mean and dividing by the standard deviation. This process centers the data around zero and scales it, making it easier for the model to learn and reducing disparities between features. It aid convergence and improve model performance.

2. Data Splitting:
   The dataset is split into training and validation sets using an 80-20 ratio. The train_mask and val_mask attributes in the dataset object are used to indicate the indices of nodes assigned for training and validation, respectively. This split allows for model training on a subset of the data while reserving a portion for evaluating the model's performance.

These preprocessing steps ensure that the features are normalized for effective learning and that the data is divided into appropriate subsets for training and validation purposes.

Model selection and evaluation

We tested three models and each one with different hyperparameters:
-Multi-Layer Perceptron
-Graph Convolutional Network
-Graph Attention Network

For each model we used

- Cross entropy loss that combines the softmax activation function and the negative log likelihood loss, making it suitable for optimizing models that predict class probabilities,
- Adam optimizer. It adapts the learning rate for each parameter based on the magnitude of their gradients, incorporates momentum to accelerate optimization, applies bias correction, and includes weight decay for regularization.
  We tested a large range of the learning rate variable and got our best result in all the models for a learning rate of 0.005.
  The chosen weight decay was of 5e-4

Multi-Layer Perceptron

MLP is a powerful neural network with multiple interconnected layers used for tasks like classification and regression.
It excels in learning complex relationships but has limitations, including the potential for a high number of parameters, redundancy in high dimensions, and disregard for spatial information when taking flattened inputs.

MLP Layers:

- Input layer: The feature vectors for each node are passed through a **Linear** module, which maps the input features from **dataset.num_features** (128 in this case) to **hidden_channels** (16 in this case) output channels.
- Hidden layer: The output of the input layer is passed through a Rectified Linear Unit (ReLU) activation function, introducing non-linearity to the model.
- Dropout: A dropout layer is applied to reduce overfitting. Dropout randomly zeros out elements with a probability of 0.5 during training.
- Output layer: The output of the hidden layer is passed through another **Linear** module, which maps the hidden channels (16) to the number of output classes (**dataset.num_classes**).

We trained it for 99 epochs with hidden_channel=8,16,32 and 64.

The training phase takes about 5 seconds on the server with 100 epochs and the inference phase takes even less time if the data is already loaded.

Our best result was with an hidden_channel of 16 who got a **train loss of 2.5170** (on epoch 99) and an **accuracy of 0.3880** for the validation set.

```
Epoch: 093, Loss: 2.5291
Epoch: 094, Loss: 2.5281
Epoch: 095, Loss: 2.5260
Epoch: 096, Loss: 2.5208
Epoch: 097, Loss: 2.5215
Epoch: 098, Loss: 2.5201
Epoch: 099, Loss: 2.5170
Test Accuracy: 0.3880
```

It was not sufficient so we tried a second model.

Graph Convolutional Network

GCN is a neural network architecture designed for graph-structured data.
It is effective because it incorporates the graph structure into the learning process, enabling the model to capture relationships and dependencies between nodes.
By performing localized information aggregation and iterative feature learning, GCN can extract expressive node representations that capture complex patterns and structural information in the graph. GCN's ability to generalize well to unseen data, scalability to large graphs, and flexibility for various tasks make it a powerful and widely applicable architecture in graph-based learning.

Our GCN architecture:

```
GCN( (conv1): GCNConv(128, 64) (conv2): GCNConv(64, 32) (conv3): GCNConv(32, 8) (conv4): GCNConv(8, 40) )
```

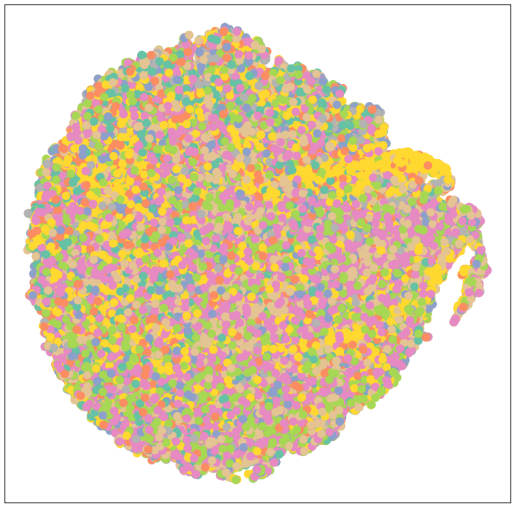
Furthemore,the GCN model applies multiple graph convolutional layers with ReLU activations and dropout layers to learn node representations and make predictions based on the graph structure and node features.
The training phase takes about 15 seconds on the server with 100 epochs and the inference phase takes a few seconds if the data is already loaded.

We trained it for 100 epoch with hidden_channel=8,16,32 and 64.Our best result was for hidden_channel=16 with the **train loss 2.0605** (on epoch 98) and an **accuracy of 0.5038** for the validation set.

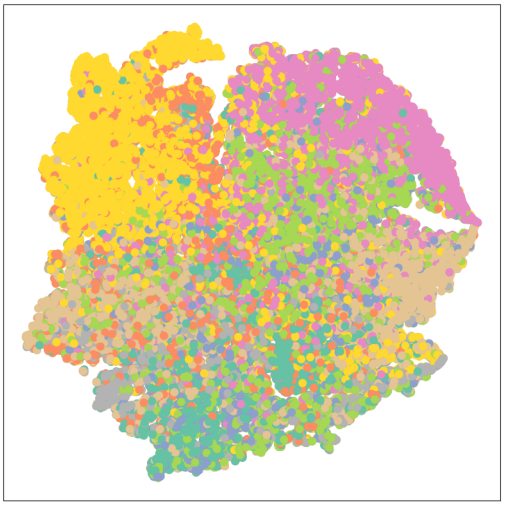
```
Epoch: 097, Loss: 2.0692
Epoch: 098, Loss: 2.0605
Epoch: 099, Loss: 2.0620
Epoch: 100, Loss: 2.0641
```

```
Test Accuracy: 0.5035
```

The GCN results can be visualized with the embedding of the nodes as follow:



Untrained GCN network



Trained GCN network

So we improved our result in a significant way but we wanted to get better accuracy on validation set and a lower train loss.

Graph Attention Network: *(chosen model was 2-layer GAT)*

The 2-layer GAT model is also a neural network architecture for graph-structured data. His specificity is that it applies attention mechanisms in two layers to update node representations by attending to important neighbors. This allows the model to capture complex patterns and relationships within the graph. The final output of the model represents the learned node representations, which can be used for node classification.

Our 2-layer GAT Architecture:

```
GATModel( (conv1): GATConv(128, 64, heads=40) (convs): ModuleList( (0): GATConv(2560, 64, heads=40) ) (lin): Linear(in_features=2560, out_features=40, bias=True) )
```

A short explanation of how it's work can be:

- It utilizes multiple GATConv layers with graph attention mechanisms to update node representations.
- The first layer takes input features and graph connectivity as input.
- Subsequent layers are created using nn.ModuleList and applied sequentially.
- The number of layers and hidden dimension are determined by num_layers and hidden_dim arguments.
- Attention heads are used in each layer, specified by num_heads.
- Dropout is applied to input features and hidden representations for regularization.
- The F.elu activation function introduces non-linearity.
- The final layer is a linear layer mapping hidden representations to output classes.
- Log-softmax activation is applied to obtain class probabilities.

We used a 2_layers GATModel with heads=8 because we didn't have enough RAM on the server or on our computers. We trained it for first 100 epoch. We saw that the result was improved so we chose to train it for 200 epochs and save the model with the highest accuracy on the validation set.
The training phase takes about 10 min on the server but 30 mins minimum on our computers or google colab with 200 epochs and the inference phase takes a few seconds if the data is already loaded.
We tried several hidden_dim like 32 or 64, but our best result was for an hidden_channel of 16 with a **train loss of 1.6248** (on epoch 186) and an **accuracy of 0.5834** for the validation set.

```
:Epoch: 178, Loss: 1.6273, Val: 0.5799
:Epoch: 179, Loss: 1.6259, Val: 0.5800
:Epoch: 180, Loss: 1.6299, Val: 0.5806
:Epoch: 181, Loss: 1.6277, Val: 0.5823
:Epoch: 182, Loss: 1.6316, Val: 0.5834
:Epoch: 183, Loss: 1.6280, Val: 0.5833
:Epoch: 184, Loss: 1.6271, Val: 0.5824
:Epoch: 185, Loss: 1.6256, Val: 0.5819
:Epoch: 186, Loss: 1.6248, Val: 0.5834
:Epoch: 187, Loss: 1.6184, Val: 0.5833
:Epoch: 188, Loss: 1.6237, Val: 0.5826
:Epoch: 189, Loss: 1.6160, Val: 0.5814
:Epoch: 190, Loss: 1.6306, Val: 0.5816
:Epoch: 191, Loss: 1.6206, Val: 0.5831
:Epoch: 192, Loss: 1.6226, Val: 0.5821
:Epoch: 193, Loss: 1.6222, Val: 0.5828
:Epoch: 194, Loss: 1.6224, Val: 0.5816
:Epoch: 195, Loss: 1.6273, Val: 0.5822
:Epoch: 196, Loss: 1.6213, Val: 0.5829
:Epoch: 197, Loss: 1.6154, Val: 0.5824
:Epoch: 198, Loss: 1.6208, Val: 0.5796
:Epoch: 199, Loss: 1.6197, Val: 0.5810
:Epoch: 200, Loss: 1.6148, Val: 0.5815
```

Conclusion

Through the application of different models, we observed improvements in accuracy and loss metrics, showcasing the effectiveness of GNNs in analyzing graph-structured data. The choice of the GAT model and its attention mechanisms proved to be the most successful in capturing intricate patterns and achieving higher prediction accuracy.

We've been limited by the computational resources (lack of RAM) and by the lack of time we had to train more models because GAT is relatively long to run.

References:

Colab Notebooks and Video Tutorials — pytorch_geometric documentation (pytorch-geometric.readthedocs.io)

https://colab.research.google.com/drive/14OvFnAXggxB8vM4e8vSURUp1TaKnovzX?usp=sharing#scrollTo=0YgHcLXMLk4o

https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.GATConv