COLAB 0 – 5 FINAL REPORT (SUMMARY)

Vedant Tewari

Summer 2022

COMPSCI 699

COLAB 0:

Recently, deep learning on graphs has emerged as one of the hottest research fields in the deep learning community. **Graph Neural Networks (GNNs)** aim to generalize classical deep learning concepts to structured relational data (distinct from images or texts), enabling neural networks to reason about objects and their relations.
This lab is a practice to get familiar with the basic concepts of graph mining and Graph Neural Networks.

In this Collab, two packages are introduced, NetworkX and PyTorch Geometric.

NetworkX is one of the most frequently used Python packages to create, manipulate, and mine graphs.

NetworkX provides several classes to store different types of graphs, such as directed and undirected graphs. It also provides classes to create multigraphs (both directed and undirected).

PyTorch Geometric is an extension to the popular deep learning framework PyTorch and consists of various methods and utilities to ease the implementation of Graph Neural Networks.

This lab is based on a simple graph-structured example, the well-known **Zachary's karate club network**. This graph describes a social network of 34 members of a karate club where a link exists between members if they have interacted outside the club. For this exploration, we are interested in detecting communities that arise from the member's interaction.

Using the NetworkX library:

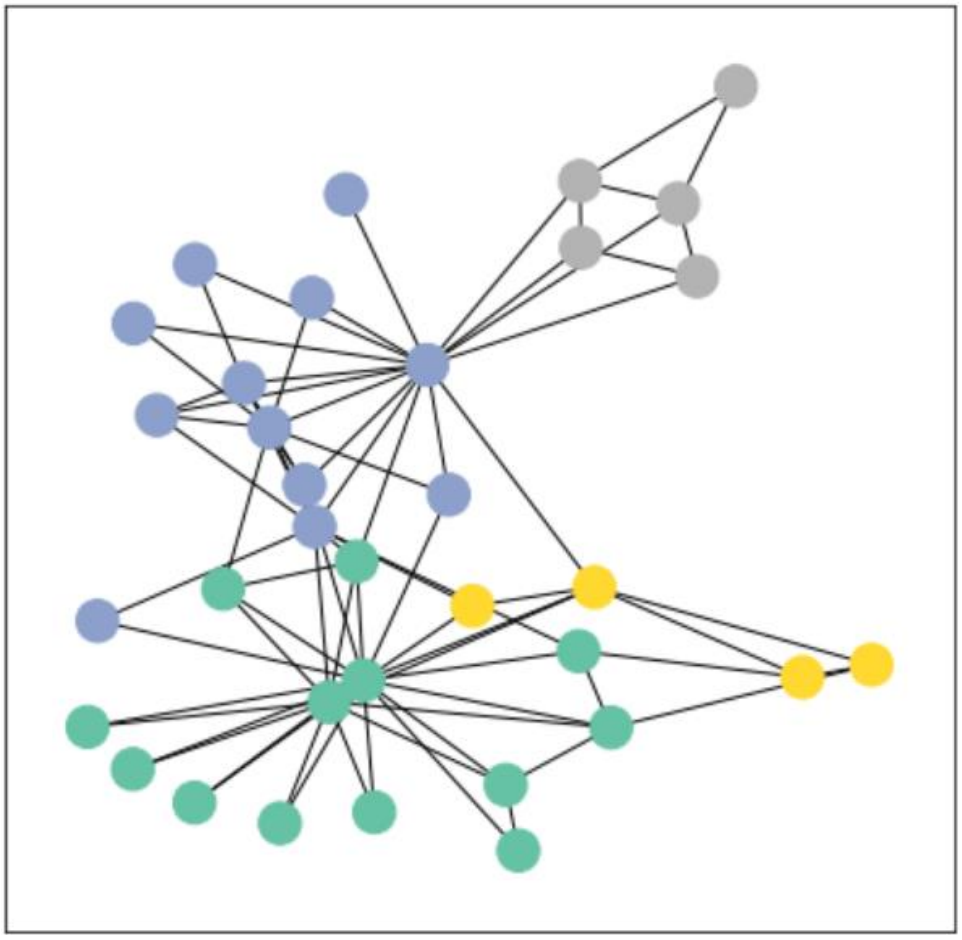
Tutorial:

1. Setup (importing libraries and packages)
2. Creating the graph
3. Adding Nodes (with attributes)
4. Adding Edges (with attributes)
5. Visualizing
6. Exploring Node degrees and neighbors
7. Exploring Other Functionalities (such as PageRank of nodes)

Using PyTorch Geometric Library:

Tutorial:

1. Setup (importing libraries and packages)
2. Visualization
3. Initialization of the Zachary Dataset (since PyTorch Gy provides easy access)
4. Data Exploration (exploring connectivity, edges, labels, features and utility functions through data object attributes)
5. Edge Indexes (in COO format)
6. Visualizing the Graph using NetworkX



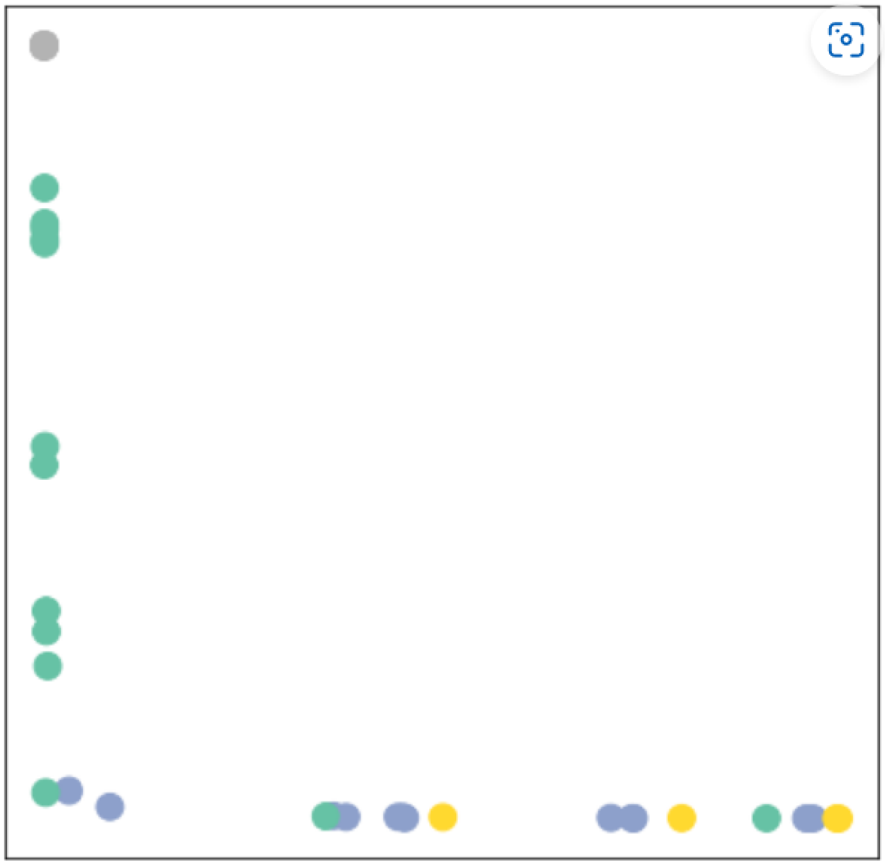
7. Implementing the Graph Neural Network (using PyTorch nn.Module Class, Our GNN is defined by stacking **three graph convolution layers**, which corresponds to aggregating 3-hop neighborhood information around each node (all nodes up to 3 "hops" away). In addition, the GCNConv layers reduce the node feature dimensionality to 2, *i.e.*, $34 \rightarrow 4 \rightarrow 4 \rightarrow 2$. Each GCNConv layer is enhanced by a tanh non-linearity. After that, we apply a single linear transformation (`torch.nn.Linear`) that acts as a classifier to map our nodes to 1 out of the 4 classes/communities. We return both the output of the final classifier as well as the final node embeddings produced by our GNN. We proceed to initialize our final model via GCN function (simple GNN operator), and printing our model produces a summary of all its used sub-modules.
8. Visualizing Node Embeddings generated by GCN
9. Training the Model:
    a. Since everything in our model is differentiable and parameterized, we can add node labels, train the model, and observe how the embeddings react. Here, we make use of a semi-

supervised learning procedure: We train with the supervision of one node per class but are allowed to make use of the complete input graph data for the generation of node embeddings. Our goal is to then predict the labels of the unknown nodes.

b. Training our model is very similar to any other PyTorch model. In addition to defining our network architecture, we define a loss criterion (`CrossEntropyLoss`) And initialize a stochastic gradient optimizer (`Adam`). After that, we perform multiple rounds of optimization, where each round consists of a forward and backward pass to compute the gradients of our model parameters w.r.t. to the loss derived from the forward pass. Note that our semi-supervised learning scenario is achieved by the following line:
loss = criterion(out[data.train_mask], data.y[data.train_mask])

c. While we compute node embeddings for all our nodes, we **only make use of the training node embeddings for computing the loss**. This is implemented by filtering the output of the classifier out and ground-truth labels data.y to only contain the nodes in the train_mask.

10. Visualize results per epoch


Epoch: 400, Loss: 0.1522

COLAB 1:

In this Collab, we will write a full pipeline for **learning node embeddings**. We will go through the following 3 steps.

Step 1: To start, we will load a classic graph in network science, the Karate Club Network. We will explore multiple graph statistics for that graph.

Step 2: We will then work together to transform the graph structure into a PyTorch tensor, so that we can perform machine learning over the graph.

Step 3: Finally, we will finish the first learning algorithm on graphs: a node embedding model. For simplicity, our model here is simpler than DeepWalk / node2vec algorithms taught in the lecture. But it's still rewarding and challenging, as we will write it from scratch via PyTorch.

Graph Basics:

1. Setup

2. Visualize Zachary's Network (The Karate Club Network is a graph describing a social network of 34 members of a karate club and documents links between members who interacted outside the club.)

3. TODO: -

Finding Average Degree:

```
# TODO: Implement this function that takes number of edges
          # and number of nodes, and returns the average node degree of
            # the graph. Round the result to nearest integer (for example
       # 3.3 will be rounded to 3 and 3.7 will be rounded to 4)
```

Took ratio of edges and nodes and multiplied by 2 since counting edges twice per node, rounding the value as instructed

Finding Average Clustering Co-efficient:

```
       # TODO: Implement this function that takes a nx.Graph
       # and returns the average clustering coefficient. Round
       # the result to 2 decimal places (for example 3.333 will
       # be rounded to 3.33 and 3.7571 will be rounded to 3.76)
```

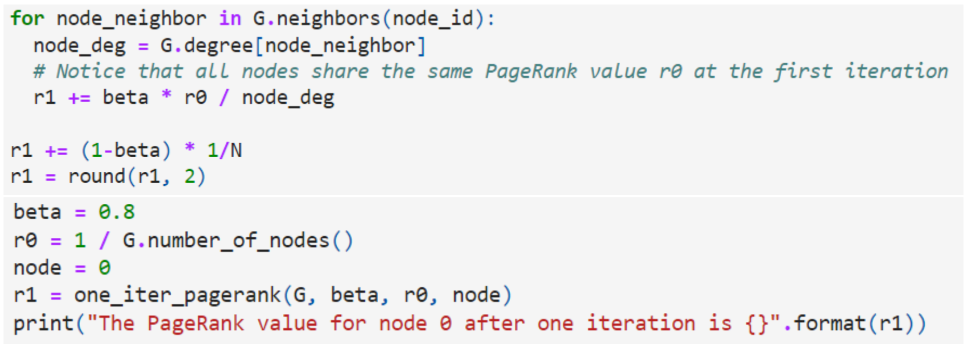
Initialized variable storing average clustering co-efficient and then calling the NetworkX clustering function on the Graph and rounding the value to 2 d.p.

Finding PageRank for Node 0 after one iteration:

implementing the PageRank equation: $r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$

```
# TODO: Implement this function that takes a nx.Graph, beta, r0 and node id.
# The return value r1 is one iteration PageRank value for the input node.
# Please round r1 to 2 decimal places.
```

First getting the number of nodes of G, and then looping through all the neighbors of a starting node (by ID) storing the node degree per iteration and updating the PageRank value:



```
for node_neighbor in G.neighbors(node_id):
  node_deg = G.degree[node_neighbor]
  # Notice that all nodes share the same PageRank value r0 at the first iteration
  r1 += beta * r0 / node_deg

r1 += (1-beta) * 1/N
r1 = round(r1, 2)
```

```
beta = 0.8
r0 = 1 / G.number_of_nodes()
node = 0
r1 = one_iter_pagerank(G, beta, r0, node)
print("The PageRank value for node 0 after one iteration is {}".format(r1))
```

Beta is usually taken as 0.7 or 0.8

Raw Closeness Centrality:

The equation for closeness centrality is $c(v) = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$

```
# TODO: Implement the function that calculates closeness centrality
# for a node in karate club network. G is the input karate club
# network and node are the node id in the graph. Please round the
# closeness centrality result to 2 decimal places.

## Note:
## 1: You can use networkx closeness centrality function.
## 2: Notice that networkx closeness centrality returns the normalized
## closeness directly, which is different from the raw (unnormalized)
## one that we learned in the lecture.
# Normalized version from NetworkX
# closeness = nx.closeness_centrality(G, node)
```

Closeness Centrality -> is a measure of the average shortest distance from each vertex to each other vertex. Specifically, it is the inverse of the average shortest distance between the vertex and all other vertices in the network. The formula is 1/ (average distance to all other vertices).

For each path in the list finding the shortest path per Node (by ID) *[Computes shortest path between source and all other nodes reachable from source.]* *and appending to a list, the path length is calculated by taking the length of the list and subtracting the starting point, then we store the closeness centrality in a variable and round to 2 d.p.*

2. Graph to Tensor:

transform the graph G into a PyTorch tensor, so that we can perform machine learning over the graph.

1. Setup (import libraries and packages)

2. Generate PyTorch tensor with all zeros, ones or random values.

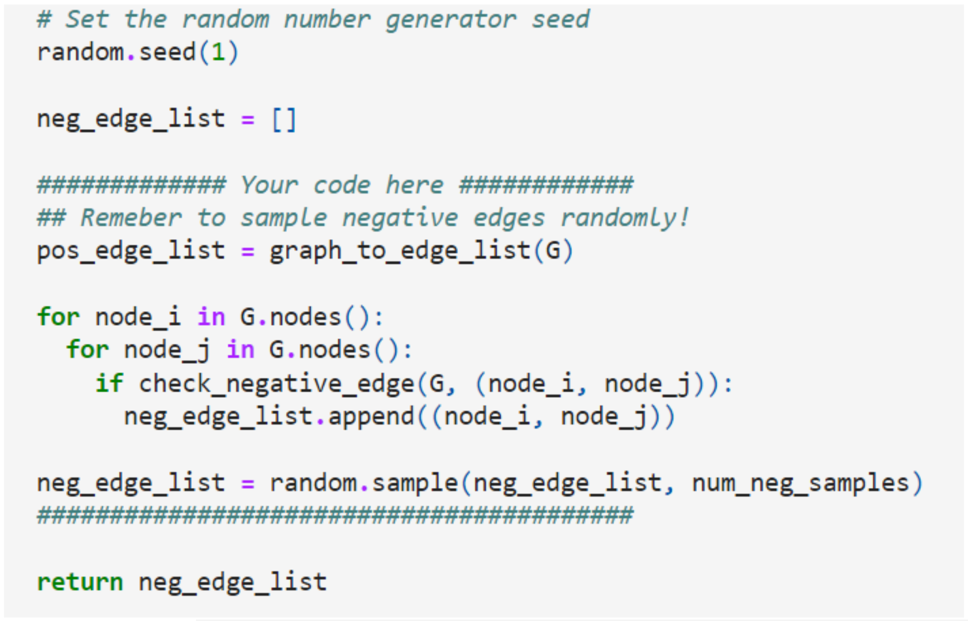
3. Getting Edge List of Network and transforming it into a Long Tensor:

Finding torch.sum of pos_edge_index:

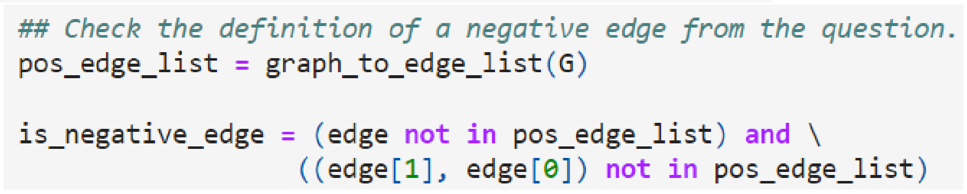
```
       # TODO: Implement the function that returns the edge list of
            # an nx.Graph. The returned edge_list should be a list of tuples
              # where each tuple is a tuple representing an edge connected
              # by two nodes.

edge_list = list(G.edges())

# TODO: Implement the function that transforms the edge_list to
              # tensor. The input edge_list is a list of tuples and the resulting
              # tensor should have the shape [2 x len(edge_list)].

edge_index = torch.tensor(np.array(edge_list), dtype=torch.long)
                edge_index = edge_index.T [accessing attribute of edge_index which is the
                                          edge list]
```

4. Using negative sampling to get negative edges and then determining possible negative edges:

```
# TODO: Implement the function that returns a list of negative edges.
            # The number of sampled negative edges is num_neg_samples. You do not
              # need to consider the corner case when the number of possible negative
edges
              # is less than num_neg_samples. It should be ok as long as your
implementation
              # works on the karate club network.
```



```
# Set the random number generator seed
random.seed(1)

neg_edge_list = []

############# Your code here ############
## Remeber to sample negative edges randomly!
pos_edge_list = graph_to_edge_list(G)

for node_i in G.nodes():
  for node_j in G.nodes():
    if check_negative_edge(G, (node_i, node_j)):
      neg_edge_list.append((node_i, node_j))

neg_edge_list = random.sample(neg_edge_list, num_neg_samples)
#########################################

return neg_edge_list
```



```
## Check the definition of a negative edge from the question.
pos_edge_list = graph_to_edge_list(G)

is_negative_edge = (edge not in pos_edge_list) and \
                   ((edge[1], edge[0]) not in pos_edge_list)
```

3. Node Embedding Learning

1. Setup (import packages such as sci-kit learn and matplotlib)

2. Initializing an embedding layer

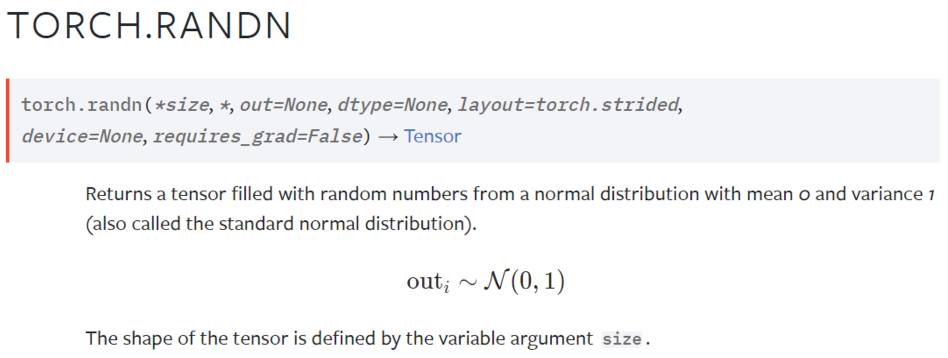
3. Selecting items from the embedding layer matrix

4. Creating Node Embedding Matrix from given graph

-> 16-dimensional vector for each node and initializing matrix under uniform distribution

```
emb = nn.Embedding(num_embeddings=num_node, embedding_dim=embedding_dim)
emb.weight.data = torch.rand(num_nodes, embedding_dim)
```

Using torch.randn() to do so:



TORCH.RANDN

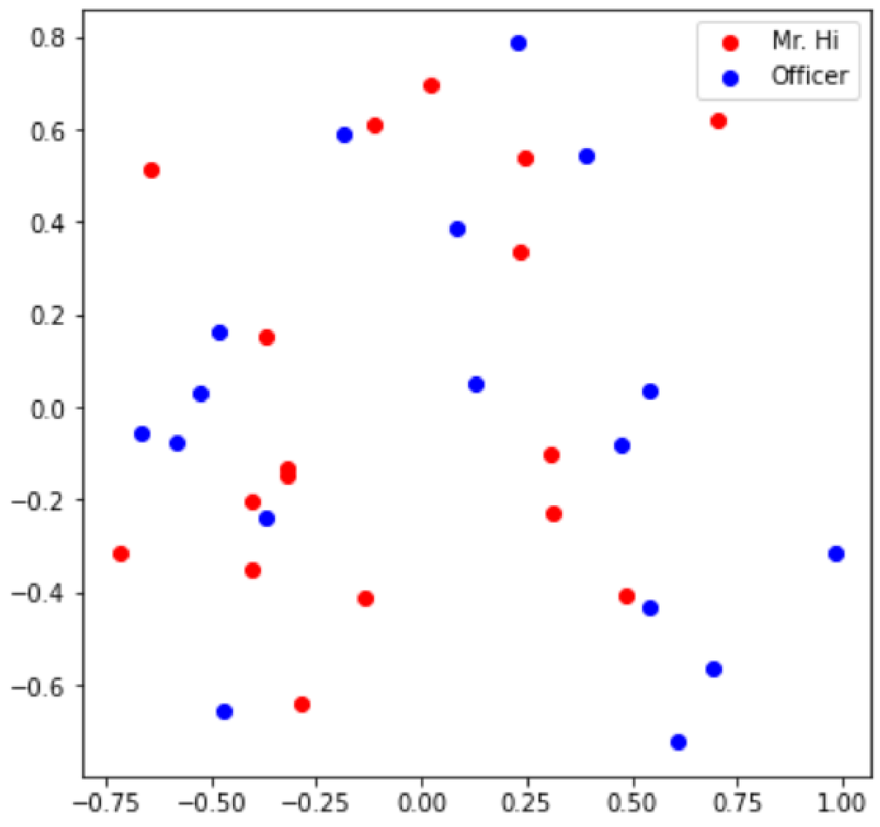
```
torch.randn(*size, *, out=None, dtype=None, layout=torch.strided,
device=None, requires_grad=False) → Tensor
```

Returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution).

$$\text{out}_i \sim \mathcal{N}(0, 1)$$

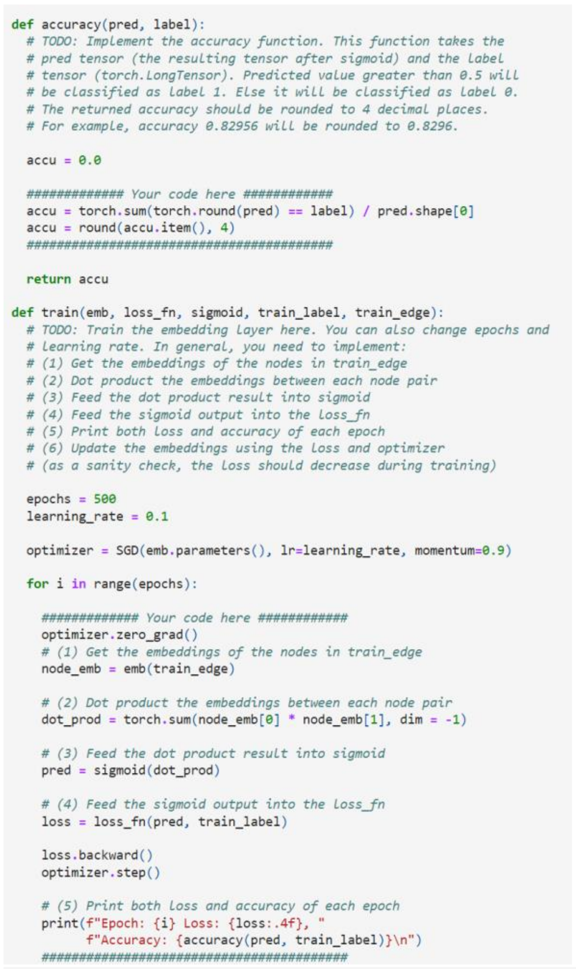
The shape of the tensor is defined by the variable argument `size`.

5. Visualizing Initial Node Embeddings (conducting PCA to reduce the dimensionality of embeddings to a 2D space. Then we visualize each point, colored by the community it belongs to)

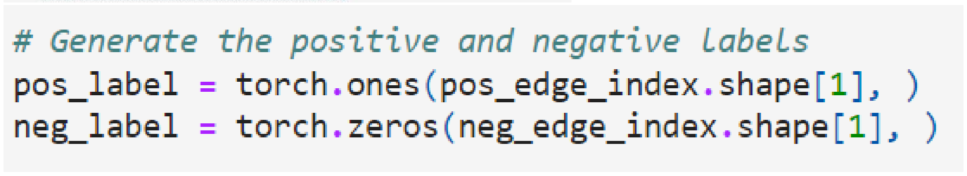


6. Training the embedding:

We want to optimize our embeddings for the task of classifying edges as positive or negative. Given an edge and the embeddings for each node, the dot product of the embeddings, followed by a sigmoid, should give us the likelihood of that edge being either positive (output of sigmoid > 0.5) or negative (output of sigmoid < 0.5).

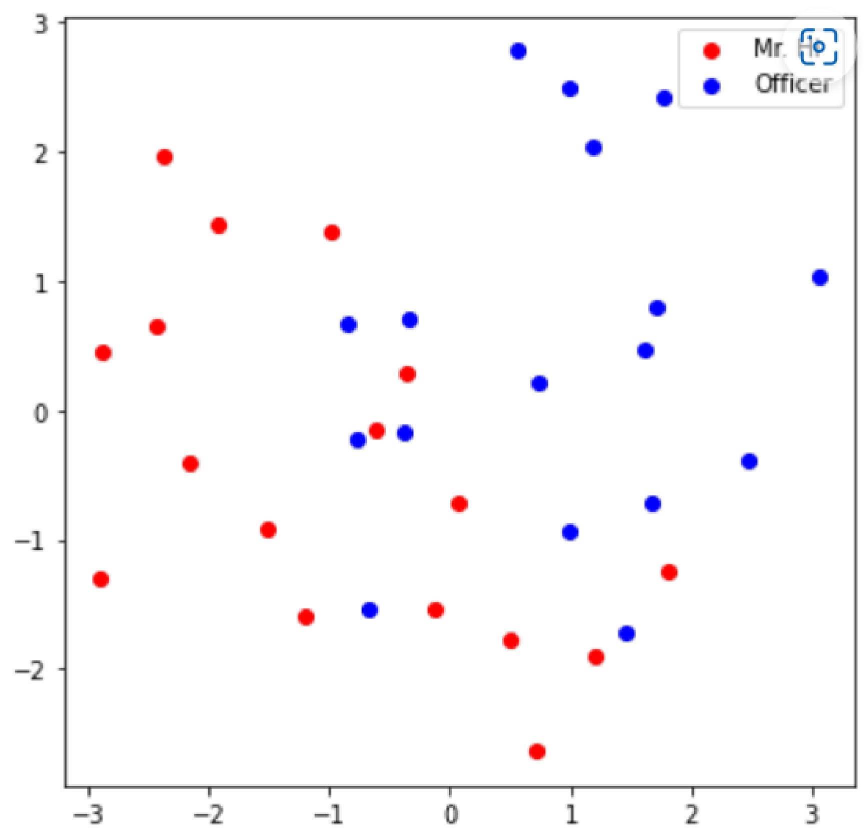


```python
def accuracy(pred, label):
  # TODO: Implement the accuracy function. This function takes the
  # pred tensor (the resulting tensor after sigmoid) and the label
  # tensor (torch.LongTensor). Predicted value greater than 0.5 will
  # be classified as label 1. Else it will be classified as label 0.
  # The returned accuracy should be rounded to 4 decimal places.
  # For example, accuracy 0.82956 will be rounded to 0.8296.

  accu = 0.0

  ############# Your code here ############
  accu = torch.sum(torch.round(pred) == label) / pred.shape[0]
  accu = round(accu.item(), 4)
  #########################################

  return accu

def train(emb, loss_fn, sigmoid, train_label, train_edge):
  # TODO: Train the embedding layer here. You can also change epochs and
  # learning rate. In general, you need to implement:
  # (1) Get the embeddings of the nodes in train_edge
  # (2) Dot product the embeddings between each node pair
  # (3) Feed the dot product result into sigmoid
  # (4) Feed the sigmoid output into the loss_fn
  # (5) Print both loss and accuracy of each epoch
  # (6) Update the embeddings using the loss and optimizer
  # (as a sanity check, the loss should decrease during training)

  epochs = 500
  learning_rate = 0.1

  optimizer = SGD(emb.parameters(), lr=learning_rate, momentum=0.9)

  for i in range(epochs):

    ############# Your code here ############
    optimizer.zero_grad()
    # (1) Get the embeddings of the nodes in train_edge
    node_emb = emb(train_edge)

    # (2) Dot product the embeddings between each node pair
    dot_prod = torch.sum(node_emb[0] * node_emb[1], dim = -1)

    # (3) Feed the dot product result into sigmoid
    pred = sigmoid(dot_prod)

    # (4) Feed the sigmoid output into the loss_fn
    loss = loss_fn(pred, train_label)

    loss.backward()
    optimizer.step()

    # (5) Print both loss and accuracy of each epoch
    print(f"Epoch: {i} Loss: {loss:.4f}, "
          f"Accuracy: {accuracy(pred, train_label)}\n")
    #########################################
```



```python
# Generate the positive and negative labels
pos_label = torch.ones(pos_edge_index.shape[1], )
neg_label = torch.zeros(neg_edge_index.shape[1], )
```

8. Visualizing final node embeddings and saving model predictions



COLAB 2:

In Colab 2, we will work to construct our own graph neural network using PyTorch Geometric (PyG) and then apply that model on two Open Graph Benchmark (OGB) datasets. These two datasets will be used to benchmark your model's performance on two different graph-based tasks: 1) node property prediction, predicting properties of single nodes and 2) graph property prediction, predicting properties of entire graphs or subgraphs.

Step 1: First, we will learn how PyTorch Geometric stores graphs as PyTorch tensors.

Step 2: Then, we will load and inspect one of the Open Graph Benchmark (OGB) datasets by using the ogb package.

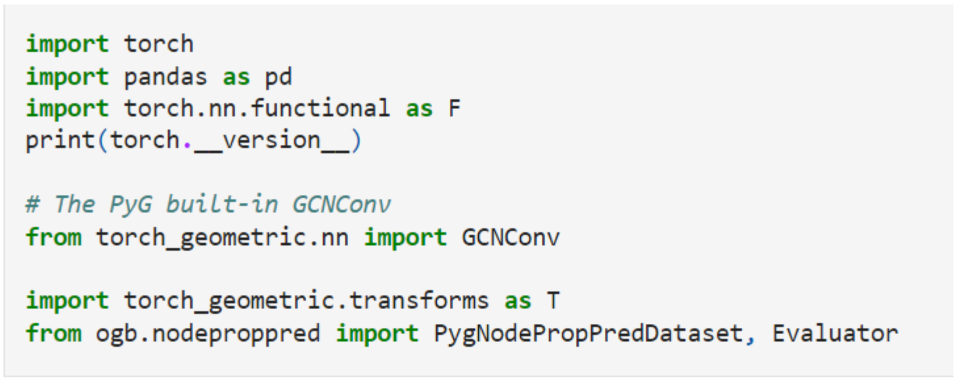
(OGB is a collection of realistic, large-scale, and diverse benchmark datasets for machine learning on graphs. The ogb package not only provides data loaders for each dataset but also model evaluators.)

Step 3: Lastly, we will build our own graph neural network using PyTorch Geometric. We will then train and evaluate our model on the OGB node property prediction and graph property prediction tasks.

1. Setup (importing PyTorch and checking for GPU at runtime)
2. Working with PyTorch Geometric Library for storing and transforming graphs

    a. PyTorch Geometric has two classes for storing and/or transforming graphs into tensor format. One is `torch_geometric.datasets`, which contains a variety of common graph datasets. Another is `torch_geometric.data`, which provides the data handling of graphs in PyTorch tensors.
    b. In this section, we will learn how to use `torch_geometric.datasets` and `torch_geometric.data` together.
    c. Each PyG dataset stores a list of `torch_geometric.data.Data` objects, where each `torch_geometric.data.Data` object represents a graph. We can easily get the `Data` object by indexing it into the dataset.

3. Working with Open Graph Benchmark (The Open Graph Benchmark (OGB) is a collection of realistic, large-scale, and diverse benchmark datasets for machine learning on graphs. Its datasets are automatically downloaded, processed, and split using the OGB Data Loader. The model performance can then be evaluated by using the OGB Evaluator in a unified manner.)
4. GNN: Node Prediction Property
    a. Building First GNN using PyTorch Geometric
    b. Applying it task of node property prediction
    c. Using GCN as the foundation of graph neural network and working with GCNConv layer.
        i. Setup



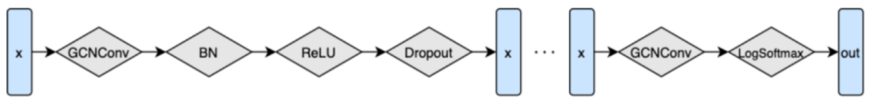
```
import torch
import pandas as pd
import torch.nn.functional as F
print(torch.__version__)

# The PyG built-in GCNConv
from torch_geometric.nn import GCNConv

import torch_geometric.transforms as T
from ogb.nodeproppred import PygNodePropPredDataset, Evaluator
```

        ii. Loading and Preprocessing dataset

```
dataset_name = 'ogbn-arxiv'
dataset = PygNodePropPredDataset(name=dataset_name,
                                transform=T.ToSparseTensor())
data = dataset[0]

# Make the adjacency matrix to symmetric
data.adj_t = data.adj_t.to_symmetric()

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

# If you use GPU, the device should be cuda
print('Device: {}'.format(device))

data = data.to(device)
split_idx = dataset.get_idx_split()
train_idx = split_idx['train'].to(device)
```

iii. Implementing GCN Model:

Please follow the figure below to implement the `forward` function.



Different layers of the model between the input x and output y

1. First we stack GCN layers on top of each other and construct the initial model:

```
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers,
                 dropout, return_embeds=False):
        # TODO: Implement a function that initializes self.convs,
        # self.bns, and self.softmax.

        super(GCN, self).__init__()

        # A list of GCNConv layers
        self.convs = None

        # A list of 1D batch normalization layers
        self.bns = None

        # The log softmax layer
        self.softmax = None
```

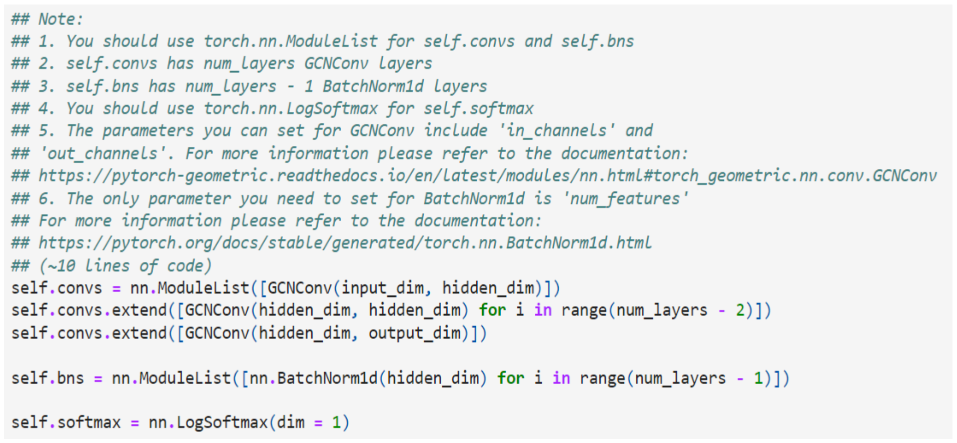
What is Batch Normalization layer:

Batch Normalization is defined as the process of training the neural network which normalizes the input to the layer for each of the small batches. This process stables the learning process and reduces the number of epochs require to train the model.

Why do we use 1D batch Normalization?

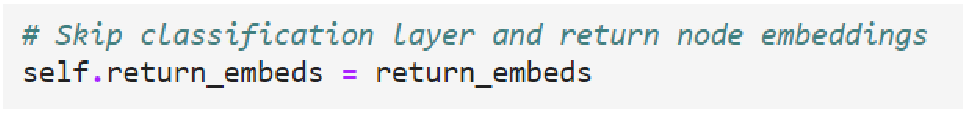
PyTorch batch normalization 1d is a technique used to build a neural network faster and more stable.

2. Setting up attributes:



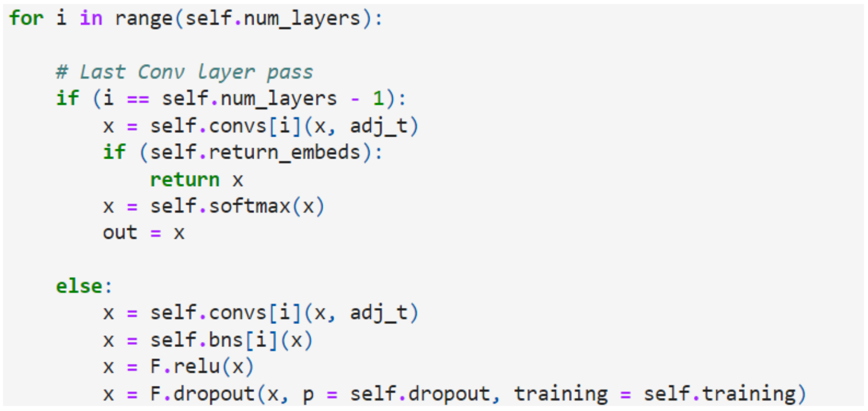
```python
## Note:
## 1. You should use torch.nn.ModuleList for self.convs and self.bns
## 2. self.convs has num_layers GCNConv layers
## 3. self.bns has num_layers - 1 BatchNorm1d layers
## 4. You should use torch.nn.LogSoftmax for self.softmax
## 5. The parameters you can set for GCNConv include 'in_channels' and
## 'out_channels'. For more information please refer to the documentation:
## https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.GCNConv
## 6. The only parameter you need to set for BatchNorm1d is 'num_features'
## For more information please refer to the documentation:
## https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html
## (~10 lines of code)
self.convs = nn.ModuleList([GCNConv(input_dim, hidden_dim)])
self.convs.extend([GCNConv(hidden_dim, hidden_dim) for i in range(num_layers - 2)])
self.convs.extend([GCNConv(hidden_dim, output_dim)])

self.bns = nn.ModuleList([nn.BatchNorm1d(hidden_dim) for i in range(num_layers - 1)])

self.softmax = nn.LogSoftmax(dim = 1)
```

Skip Classification Layer:



```python
# Skip classification layer and return node embeddings
self.return_embeds = return_embeds
```

Setting up Neural Network according to diagram:



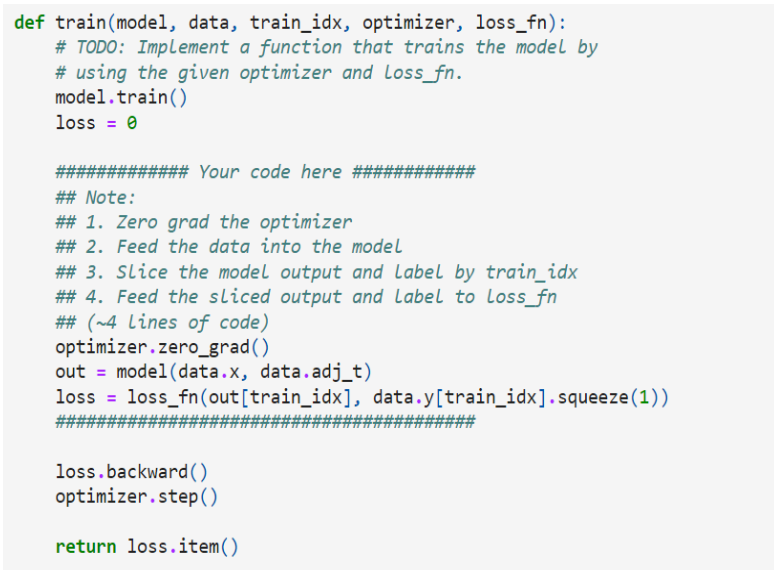
```python
for i in range(self.num_layers):

    # Last Conv layer pass
    if (i == self.num_layers - 1):
        x = self.convs[i](x, adj_t)
        if (self.return_embeds):
            return x
        x = self.softmax(x)
        out = x

    else:
        x = self.convs[i](x, adj_t)
        x = self.bns[i](x)
        x = F.relu(x)
        x = F.dropout(x, p = self.dropout, training = self.training)
```

Why is the dropout module used?

Dropout is designed to be only applied during training, so when doing predictions or evaluation of the model you want dropout to be turned off.

The dropout module `nn.Dropout` conveniently handles this and shuts dropout off as soon as your model enters evaluation mode, while the functional dropout does not care about the evaluation / prediction mode.

3. Training the model according to loss function:



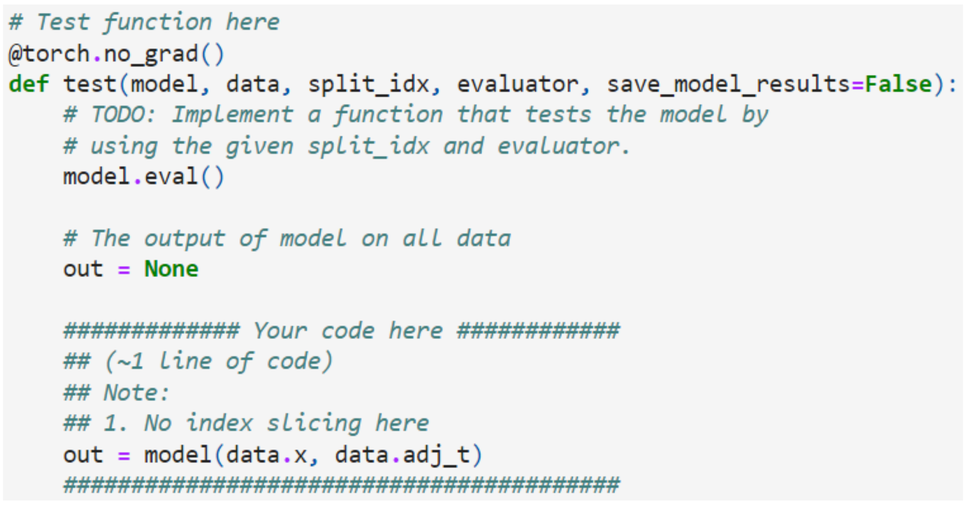
```python
def train(model, data, train_idx, optimizer, loss_fn):
    # TODO: Implement a function that trains the model by
    # using the given optimizer and loss_fn.
    model.train()
    loss = 0

    ############# Your code here ############
    ## Note:
    ## 1. Zero grad the optimizer
    ## 2. Feed the data into the model
    ## 3. Slice the model output and label by train_idx
    ## 4. Feed the sliced output and label to loss_fn
    ## (~4 lines of code)
    optimizer.zero_grad()
    out = model(data.x, data.adj_t)
    loss = loss_fn(out[train_idx], data.y[train_idx].squeeze(1))
    #########################################

    loss.backward()
    optimizer.step()

    return loss.item()
```

Why do set gradient to zero?

In `PyTorch`, for every mini batch during the *training* phase, we typically want to explicitly set the gradients to zero before starting to do backpropagation (i.e., updating the ***W****eights* and ***b****iases*) because PyTorch *accumulates the gradients* on subsequent backward passes. This accumulating behavior is convenient while training RNNs or when we want to compute the gradient of the loss summed up over multiple *mini batches*. So, the default action has been set to accumulate (i.e. sum) the gradients on every `loss.backward()` call.

Because of this, when you start your training loop, ideally you should `zero out the gradients` so that you do the parameter update correctly. Otherwise, the gradient would be a combination of the old gradient, which you have already used to update your model parameters, and the newly computed gradient. It would therefore point in some other direction than the intended direction towards the *minimum* (or *maximum*, in the case of maximization objectives).

4. Testing the Model:



```
# Test function here
@torch.no_grad()
def test(model, data, split_idx, evaluator, save_model_results=False):
    # TODO: Implement a function that tests the model by
    # using the given split_idx and evaluator.
    model.eval()

    # The output of model on all data
    out = None

    ############# Your code here ############
    ## (~1 line of code)
    ## Note:
    ## 1. No index slicing here
    out = model(data.x, data.adj_t)
    #########################################
```

Why do we need the split index and evaluator?

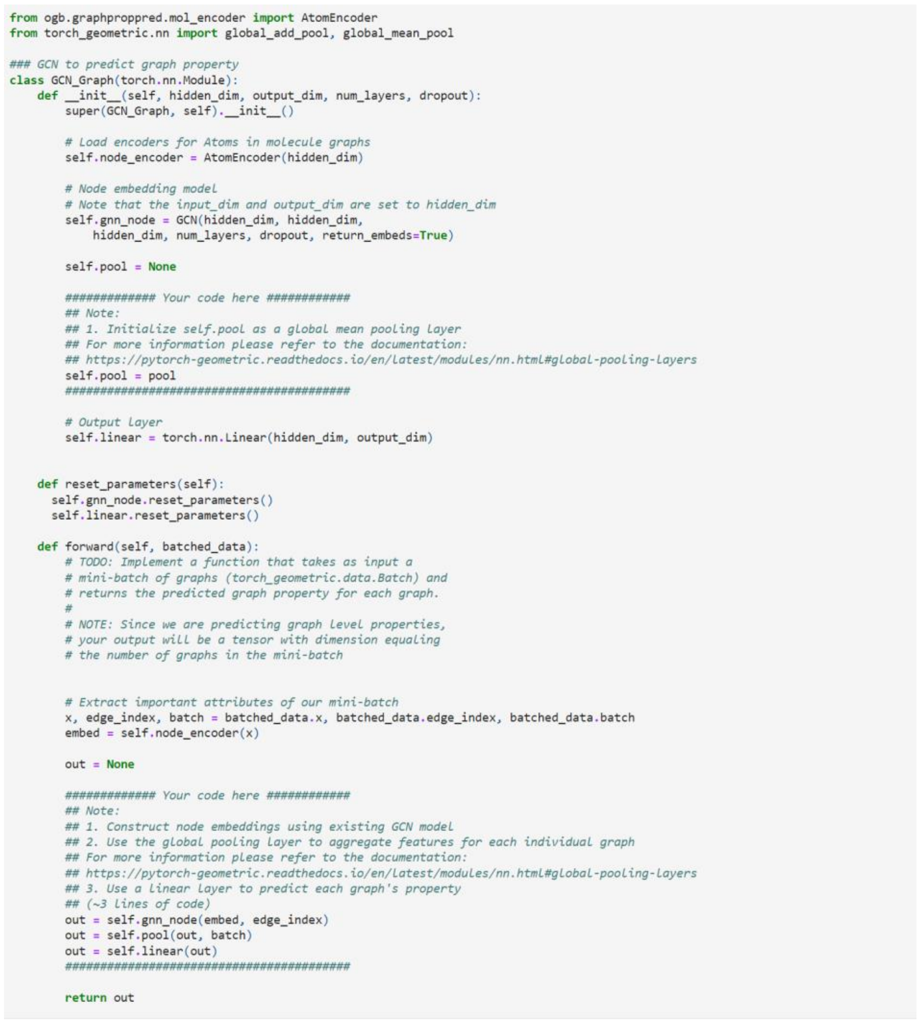
The train-test split procedure is used to estimate the performance of machine learning algorithms when they are used to make predictions on data not used to train the model.

It is a fast and easy procedure to perform, the results of which allow you to compare the performance of machine learning algorithms for your predictive modeling problem. Although simple to use and interpret, there are times when the procedure should not be used, such as when you have a small dataset and situations where additional configuration is required, such as when it is used for classification and the dataset is not balanced.

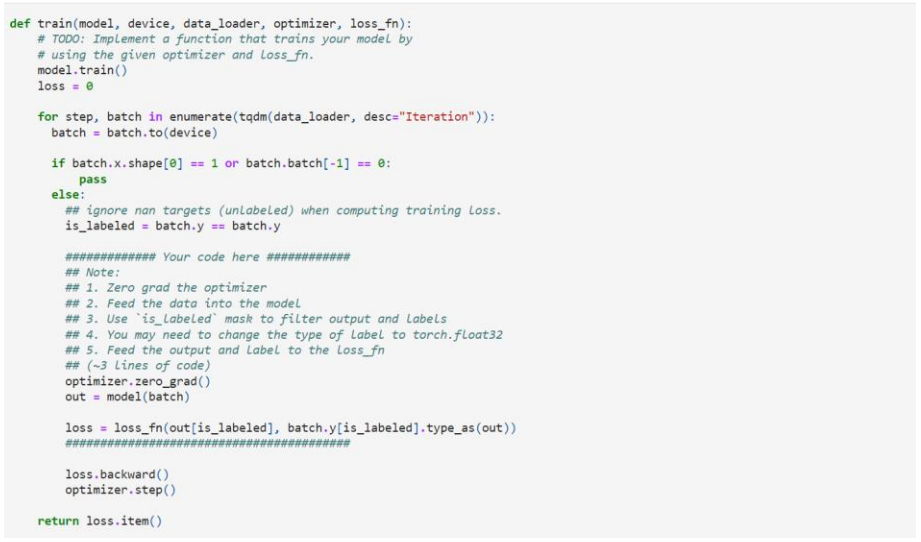
5. GNN: Graph Property Prediction
   a. Load and Preprocess dataset
   b. Implementing Graph Prediction Model
      i. Graph Mini-Batching:
         1. Before diving into the actual model, we introduce the concept of mini batching with graphs. In order to parallelize the processing of a mini batch of graphs, PyG combines the graphs into a single disconnected graph data object (*torch_geometric.data.Batch*). *torch_geometric.data.Batch* inherits from *torch_geometric.data.Data* (introduced earlier) and contains an additional attribute called `batch`.
         2. The `batch` attribute is a vector mapping each node to the index of its corresponding graph within the mini batch:
            `batch = [0, ..., 0, 1, ..., n - 2, n - 1, ..., n - 1]`
         3. This attribute is crucial for associating which graph each node belongs to and can be used to e.g., average the node embeddings for each graph individually to compute graph level embeddings.
      ii. Implementation:
         1. We will reuse the existing GCN model to generate `node_embeddings` and then use `Global Pooling` over the nodes to create graph level embeddings that can be used to predict properties for each graph.

Remember that the `batch` attribute will be essential for performing Global Pooling over our mini batch of graphs.

2. What is Global Pooling and Pooling Layers in Neural Networks and why are they required?
    a. Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.
    b. **Global Average Pooling** is a pooling operation designed to replace fully connected layers in classical CNNs. The idea is to generate one feature map for each corresponding category of the classification task in the last mlpconv layer. Instead of adding fully connected layers on top of the feature maps, we take the average of each feature map, and the resulting vector is fed directly into the softmax layer.



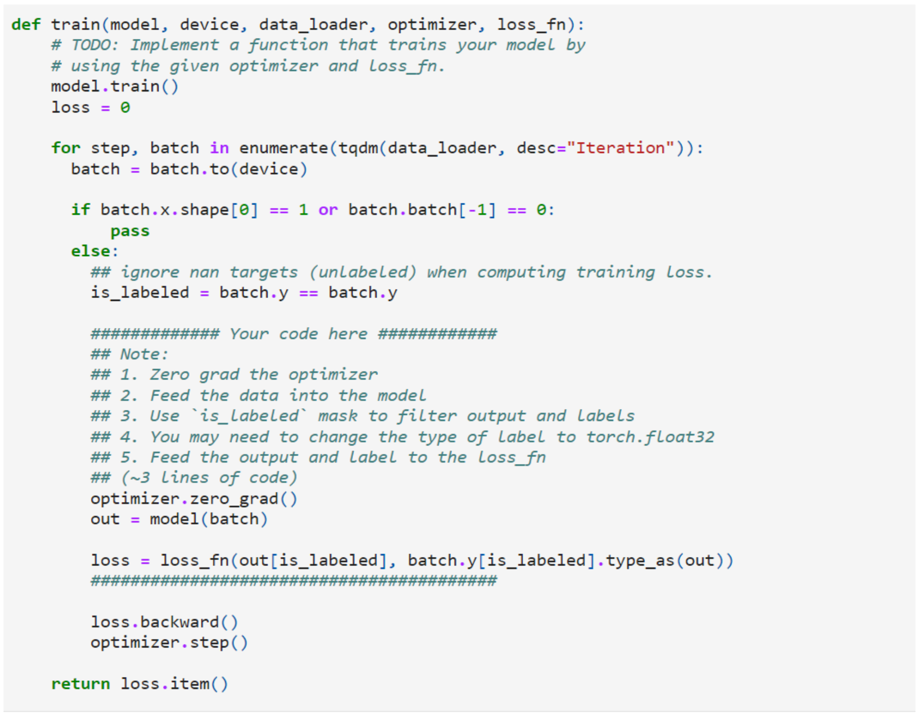
```python
from ogb.graphproppred.mol_encoder import AtomEncoder
from torch_geometric.nn import global_add_pool, global_mean_pool

### GCN to predict graph property
class GCN_Graph(torch.nn.Module):
    def __init__(self, hidden_dim, output_dim, num_layers, dropout):
        super(GCN_Graph, self).__init__()

        # Load encoders for Atoms in molecule graphs
        self.node_encoder = AtomEncoder(hidden_dim)

        # Node embedding model
        # Note that the input_dim and output_dim are set to hidden_dim
        self.gnn_node = GCN(hidden_dim, hidden_dim,
            hidden_dim, num_layers, dropout, return_embeds=True)

        self.pool = None

        ############# Your code here ############
        ## Note:
        ## 1. Initialize self.pool as a global mean pooling layer
        ## For more information please refer to the documentation:
        ## https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#global-pooling-layers
        self.pool = pool
        #########################################

        # Output layer
        self.linear = torch.nn.Linear(hidden_dim, output_dim)


    def reset_parameters(self):
      self.gnn_node.reset_parameters()
      self.linear.reset_parameters()

    def forward(self, batched_data):
        # TODO: Implement a function that takes as input a
        # mini-batch of graphs (torch_geometric.data.Batch) and
        # returns the predicted graph property for each graph.
        #
        # NOTE: Since we are predicting graph level properties,
        # your output will be a tensor with dimension equaling
        # the number of graphs in the mini-batch


        # Extract important attributes of our mini-batch
        x, edge_index, batch = batched_data.x, batched_data.edge_index, batched_data.batch
        embed = self.node_encoder(x)

        out = None

        ############# Your code here ############
        ## Note:
        ## 1. Construct node embeddings using existing GCN model
        ## 2. Use the global pooling layer to aggregate features for each individual graph
        ## For more information please refer to the documentation:
        ## https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#global-pooling-layers
        ## 3. Use a linear layer to predict each graph's property
        ## (~3 lines of code)
        out = self.gnn_node(embed, edge_index)
        out = self.pool(out, batch)
        out = self.linear(out)
        #########################################

        return out
```



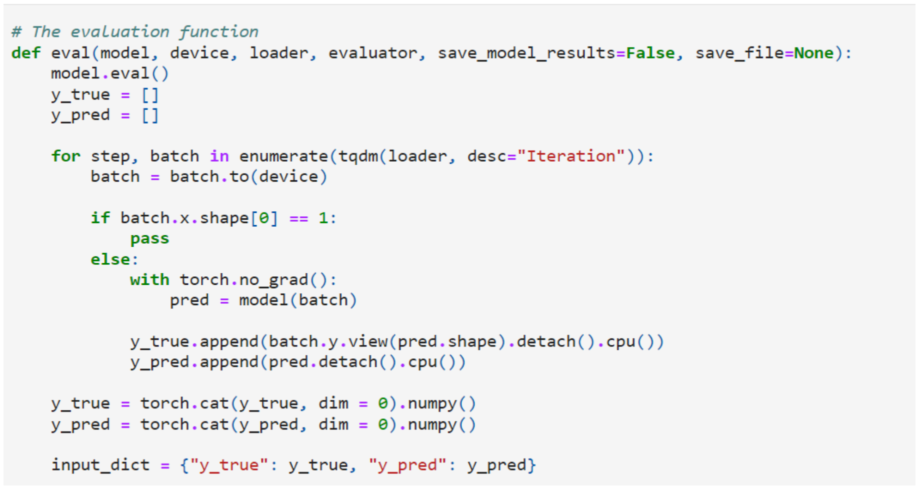
```python
def train(model, device, data_loader, optimizer, loss_fn):
    # TODO: Implement a function that trains your model by
    # using the given optimizer and loss_fn.
    model.train()
    loss = 0

    for step, batch in enumerate(tqdm(data_loader, desc="Iteration")):
      batch = batch.to(device)

      if batch.x.shape[0] == 1 or batch.batch[-1] == 0:
          pass
      else:
        ## ignore nan targets (unlabeled) when computing training loss.
        is_labeled = batch.y == batch.y

        ############# Your code here ############
        ## Note:
        ## 1. Zero grad the optimizer
        ## 2. Feed the data into the model
        ## 3. Use `is_labeled` mask to filter output and labels
        ## 4. You may need to change the type of label to torch.float32
        ## 5. Feed the output and label to the loss_fn
        ## (~3 lines of code)
        optimizer.zero_grad()
        out = model(batch)

        loss = loss_fn(out[is_labeled], batch.y[is_labeled].type_as(out))
        #########################################

        loss.backward()
        optimizer.step()

    return loss.item()
```

3.

4. Training and evaluation functions:



```
def train(model, device, data_loader, optimizer, loss_fn):
    # TODO: Implement a function that trains your model by
    # using the given optimizer and loss_fn.
    model.train()
    loss = 0

    for step, batch in enumerate(tqdm(data_loader, desc="Iteration")):
      batch = batch.to(device)

      if batch.x.shape[0] == 1 or batch.batch[-1] == 0:
          pass
      else:
        ## ignore nan targets (unlabeled) when computing training loss.
        is_labeled = batch.y == batch.y

        ############# Your code here ############
        ## Note:
        ## 1. Zero grad the optimizer
        ## 2. Feed the data into the model
        ## 3. Use `is_labeled` mask to filter output and labels
        ## 4. You may need to change the type of label to torch.float32
        ## 5. Feed the output and label to the loss_fn
        ## (~3 lines of code)
        optimizer.zero_grad()
        out = model(batch)

        loss = loss_fn(out[is_labeled], batch.y[is_labeled].type_as(out))
        #########################################

        loss.backward()
        optimizer.step()

    return loss.item()
```

5. 

```
# The evaluation function
def eval(model, device, loader, evaluator, save_model_results=False, save_file=None):
    model.eval()
    y_true = []
    y_pred = []

    for step, batch in enumerate(tqdm(loader, desc="Iteration")):
        batch = batch.to(device)

        if batch.x.shape[0] == 1:
            pass
        else:
            with torch.no_grad():
                pred = model(batch)

            y_true.append(batch.y.view(pred.shape).detach().cpu())
            y_pred.append(pred.detach().cpu())

    y_true = torch.cat(y_true, dim = 0).numpy()
    y_pred = torch.cat(y_pred, dim = 0).numpy()

    input_dict = {"y_true": y_true, "y_pred": y_pred}
```

6. Predictions and decided best accuracy

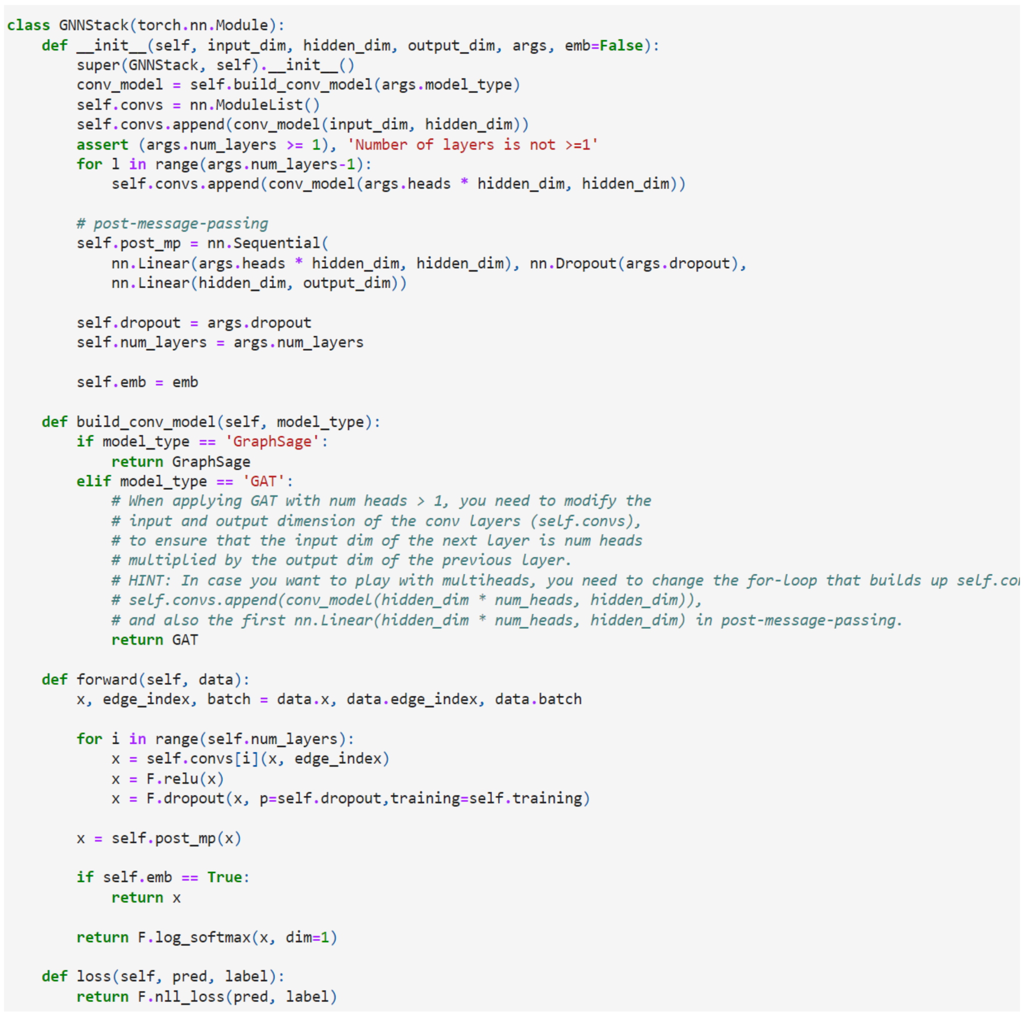
COLAB 3:

In Colab 2 we constructed GNN models by using PyTorch Geometrics built in GCN layer, GCNConv. In this Colab we will go a step deeper and implement the **GraphSAGE** (Hamilton et al. (2017)) and **GAT** (Veličković et al. (2018)) Layers directly. Then we will run and test our models on the CORA dataset, a standard citation network benchmark dataset.

Next, we will use DeepSNAP, a Python library assisting efficient deep learning on graphs, to split the graphs in different settings and apply dataset transformations.

Lastly, using DeepSNAP's transductive link prediction dataset splitting functionality, we will construct a simple GNN model for the task of edge property prediction (link prediction).

1. GNN Layers:
    a. Implementing Layer Modules
        i. In Colab 2, we implemented a GCN model for node and graph classification tasks. However, for that notebook we took advantage of PyG's built in GCN module. For Colab 3, we provide a build upon a general Graph Neural Network Stack, into which we will be able to plug in our own module implementations: GraphSAGE and GAT.
        ii. We will then use our layer implementations to complete node classification on the CORA dataset, a standard citation network benchmark. In this dataset, nodes correspond to documents and edges correspond to undirected citations. Each node or document in the graph is assigned a class label and features based on the documents binarized bag-of-words representation. Specifically, the Cora graph has 2708 nodes, 5429 edges, 7 prediction classes, and 1433 features per node.
    b. What is GraphSAGE and GAT?
        i. GraphSAGE, is a general, inductive framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings for previously unseen data. Instead of training individual embeddings for each node, we learn a function that generates embeddings by sampling and aggregating features from a node's local neighborhood.
        ii. Graph attention networks (GATs), novel neural network architectures that operate on graph-structured data, leveraging masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. By stacking layers in which nodes can attend over their neighborhoods' features, we enable (implicitly) specifying different weights to different nodes in a neighborhood, without requiring any kind of costly matrix operation (such as inversion) or depending on knowing the graph structure upfront. In this way, we address several key challenges of spectral-based graph neural networks simultaneously and make our model readily applicable to inductive as well as transductive problems.
    c. GNN Stack Module:
        i. Below is the implementation of a general GNN stack, where we can plug in any GNN layer, such as **GraphSage**, **GAT**, etc.



```
class GNNStack(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, args, emb=False):
        super(GNNStack, self).__init__()
        conv_model = self.build_conv_model(args.model_type)
        self.convs = nn.ModuleList()
        self.convs.append(conv_model(input_dim, hidden_dim))
        assert (args.num_layers >= 1), 'Number of layers is not >=1'
        for l in range(args.num_layers-1):
            self.convs.append(conv_model(args.heads * hidden_dim, hidden_dim))

        # post-message-passing
        self.post_mp = nn.Sequential(
            nn.Linear(args.heads * hidden_dim, hidden_dim), nn.Dropout(args.dropout),
            nn.Linear(hidden_dim, output_dim))

        self.dropout = args.dropout
        self.num_layers = args.num_layers

        self.emb = emb

    def build_conv_model(self, model_type):
        if model_type == 'GraphSage':
            return GraphSage
        elif model_type == 'GAT':
            # When applying GAT with num heads > 1, you need to modify the
            # input and output dimension of the conv layers (self.convs),
            # to ensure that the input dim of the next layer is num heads
            # multiplied by the output dim of the previous layer.
            # HINT: In case you want to play with multiheads, you need to change the for-loop that builds up self.co
            # self.convs.append(conv_model(hidden_dim * num_heads, hidden_dim)),
            # and also the first nn.Linear(hidden_dim * num_heads, hidden_dim) in post-message-passing.
            return GAT

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch

        for i in range(self.num_layers):
            x = self.convs[i](x, edge_index)
            x = F.relu(x)
            x = F.dropout(x, p=self.dropout,training=self.training)

        x = self.post_mp(x)

        if self.emb == True:
            return x

        return F.log_softmax(x, dim=1)

    def loss(self, pred, label):
        return F.nll_loss(pred, label)
```

ii.

d. Creating Message Passing Layer:

i. Working through this part will help us become acutely familiar with the behind the scenes work of implementing Pytorch Message Passing Layers, allowing us to build our own GNN models. To do so, we work with and implement 3 critical functions needed to define a PyG Message Passing Layer: `forward`, `message`, and `aggregate`.

ii. Reviewing the key components of the message passing process. To do so, we will focus on a single round of message passing with respect to a single central node x. Before message passing, x is associated with a feature vector $xl-1$, and the goal of message passing is to update this feature vector as xl. To do so, we implement the following steps:

1. Each neighboring node v passes its current message $vl-1$ across the edge (x,v) -
2. For the node x, we aggregate all the messages of the neighboring nodes (for example through a sum or mean) - and
3. We transform the aggregated information by, for example, applying linear and non-linear transformations. Altogether, the message passing process is applied such that every node u in our graph updates its embedding by acting as the central node x in step 1-3 described above.

iii. Now, extending this process to that of a single message passing layer, the job of a message passing layer is to update the current feature representation or embedding of each node in a graph by propagating and transforming information within the graph.

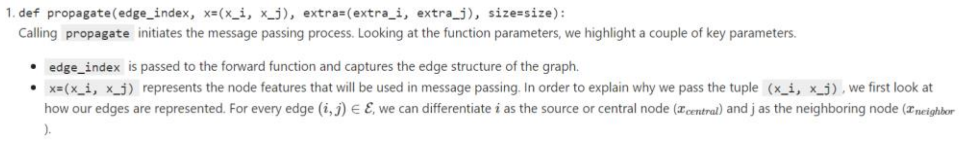
iv. Overall, the general paradigm of a message passing layers is:
   1. pre-processing
   2. **message passing** / propagation
   3. post-processing.

v. The `forward` function that we implement for our message passing layer captures this execution logic. Namely, the `forward` function handles the pre- and post-processing of node features / embeddings, as well as initiates message passing by calling the `propagate` function.

vi. The `propagate` function encapsulates the message passing process! It does so by calling three important functions: 1) `message`, 2) `aggregate`, and 3) `update`. Our implementation will vary slightly from this, as we will not explicitly implement `update`, but instead place the logic for updating node embeddings after message passing and within the `forward` function.

vii. Lastly, before starting to implement our own layer, let us dig a bit deeper into each of the functions described above:

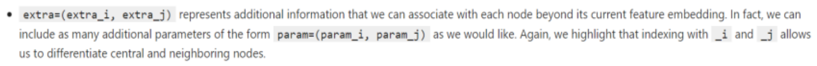
viii.



1. `def propagate(edge_index, x=(x_i, x_j), extra=(extra_i, extra_j), size=size):`

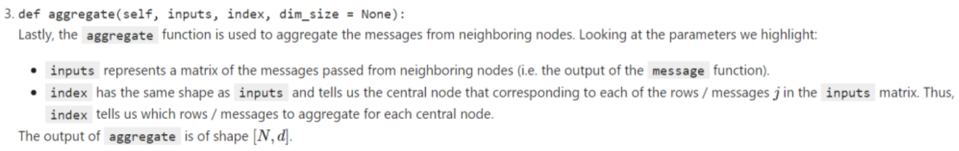
Calling `propagate` initiates the message passing process. Looking at the function parameters, we highlight a couple of key parameters.

- `edge_index` is passed to the forward function and captures the edge structure of the graph.
- `x=(x_i, x_j)` represents the node features that will be used in message passing. In order to explain why we pass the tuple `(x_i, x_j)`, we first look at how our edges are represented. For every edge $(i, j) \in \mathcal{E}$, we can differentiate i as the source or central node ($x_{central}$) and j as the neighboring node ($x_{neighbor}$).



- `extra=(extra_i, extra_j)` represents additional information that we can associate with each node beyond its current feature embedding. In fact, we can include as many additional parameters of the form `param=(param_i, param_j)` as we would like. Again, we highlight that indexing with `_i` and `_j` allows us to differentiate central and neighboring nodes.

ix.



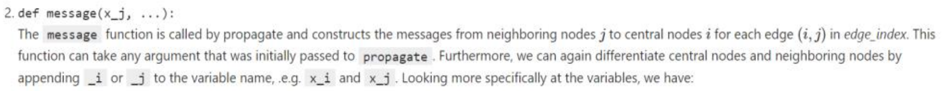
3. `def aggregate(self, inputs, index, dim_size = None):`

Lastly, the `aggregate` function is used to aggregate the messages from neighboring nodes. Looking at the parameters we highlight:

- `inputs` represents a matrix of the messages passed from neighboring nodes (i.e. the output of the `message` function).
- `index` has the same shape as `inputs` and tells us the central node that corresponding to each of the rows / messages j in the `inputs` matrix. Thus, `index` tells us which rows / messages to aggregate for each central node.

The output of `aggregate` is of shape $[N, d]$.

x.



2. `def message(x_j, ...):`

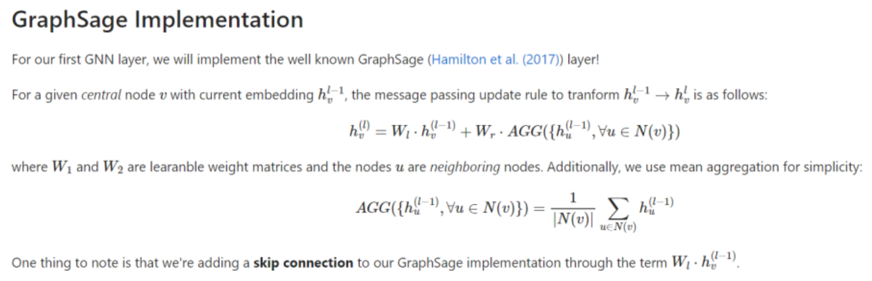
The `message` function is called by propagate and constructs the messages from neighboring nodes j to central nodes i for each edge (i, j) in *edge_index*. This function can take any argument that was initially passed to `propagate`. Furthermore, we can again differentiate central nodes and neighboring nodes by appending `_i` or `_j` to the variable name, .e.g. `x_i` and `x_j`. Looking more specifically at the variables, we have:

e. GraphSage Implementation:

i. This is the given formula: We use *Aggregation functions or aggregators* which accept a neighborhood as input and combine each neighbor's embedding with weights to create a neighborhood embedding. In other words, they aggregate information from the node's neighborhood. Aggregator weights are either learned or fixed depending on the function.

ii. To learn embeddings with aggregators, we first initialize embeddings of all nodes to node features. In turn, for each neighborhood depth until *K,* we create a neighborhood embedding with the aggregator function for each node and concatenate it with the existing embedding of the node. We pass the concatenated vector through a neural network layer to update the node embedding. When each node is processed, we normalize the embeddings to have unit norm.

iii. The advantage of learning aggregator functions to generate node embeddings, instead of learning the embeddings themselves, is inductivity, compared to algorithms such as DeepWalk which require the entire graph as input to be available to learn the embedding of some node.



GraphSage Implementation

For our first GNN layer, we will implement the well known GraphSage (Hamilton et al. (2017)) layer!

For a given *central* node v with current embedding h_v^{l-1}, the message passing update rule to tranform $h_v^{l-1} \rightarrow h_v^{l}$ is as follows:

$$h_v^{(l)} = W_l \cdot h_v^{(l-1)} + W_r \cdot AGG(\{h_u^{(l-1)}, \forall u \in N(v)\})$$

where W_1 and W_2 are learanble weight matrices and the nodes u are *neighboring* nodes. Additionally, we use mean aggregation for simplicity:

$$AGG(\{h_u^{(l-1)}, \forall u \in N(v)\}) = \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(l-1)}$$

1. One thing to note is that we're adding a **skip connection** to our GraphSage implementation through the term $W_l \cdot h_v^{(l-1)}$.

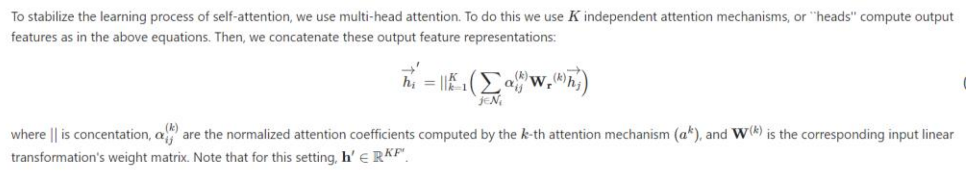
2. GAT Implementation:
   a. Attention mechanisms have become state-of-the-art in many sequence-based tasks such as machine translation and learning sentence representations. One of the major benefits of attention-based mechanisms is their ability to focus on the most relevant parts of the input to make decisions. In this problem, we will see how attention mechanisms can be used to perform node classification over graph-structured data through the usage of Graph Attention Networks (GATs)
   b. The building block of the Graph Attention Network is the graph attention layer, which is a variant of the aggregation function. Let N be the number of nodes and F be the dimension of the feature vector for each node. The input to each graph attentional layer is a set of node features: $h = \{h1\rightarrow, h2\rightarrow, \ldots, hN\rightarrow\}$, $hi\rightarrow \in RF$. The output of each graph attentional layer is a new set of node features, which may have a new dimension F': $h' = \{h1'\rightarrow, h2'\rightarrow, \ldots, hN'\rightarrow\}$, with $hi'\rightarrow \in RF'$.
   c. We will now describe how this transformation is performed for each graph attention layer. First, a shared linear transformation parametrized by the weight matrix $W \in RF' \times F$ is applied to every node.
   d. Next, we perform self-attention on the nodes. We use a shared attention function a: $(3) a : RF' \times RF' \rightarrow R.$
   e. that computes the attention coefficients capturing the importance of node j's features to node i:

$$e_{ij} = a(\mathbf{W_l}\overrightarrow{h_i}, \mathbf{W_r}\overrightarrow{h_j})$$

   f. The most general formulation of self-attention allows every node to attend to all other nodes which drops all structural information. However, to utilize graph structure in the attention mechanisms, we use **masked attention**. In masked attention, we only compute attention coefficients eij for nodes $j \in Ni$, where Ni is some neighborhood of node i in the graph.
   g. To easily compare coefficients across different nodes, we normalize the coefficients across j using a softmax function: $(5) \alpha ij = softmaxj(eij) = exp(eij) \sum k \in Niexp(eik)$
   h. For this problem, our attention mechanism a will be a single-layer feedforward neural network parametrized by a weight vector $a\rightarrow \in RF'$ and $a\rightarrow \in RF'$, followed by a LeakyReLU nonlinearity (with negative input slope 0.2). Let $\cdot T$ represent transposition and $//$ represent concatenation. The coefficients computed by our attention mechanism may be expressed as:

i. Now, we use the normalized attention coefficients to compute a linear combination of the features corresponding to them. These aggregated features will serve as the final output features for every node.

$$h_i' = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W_r} \overrightarrow{h_j}.$$

j. 

To stabilize the learning process of self-attention, we use multi-head attention. To do this we use K independent attention mechanisms, or ``heads'' compute output features as in the above equations. Then, we concatenate these output feature representations:

$$\overrightarrow{h_i}' = ||_{k=1}^{K} \Big(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} \mathbf{W_r}^{(k)} \overrightarrow{h_j} \Big) \quad ($$

where || is concentation, $\alpha_{ij}^{(k)}$ are the normalized attention coefficients computed by the k-th attention mechanism (a^k), and $\mathbf{W}^{(k)}$ is the corresponding input linear transformation's weight matrix. Note that for this setting, $\mathbf{h}' \in \mathbb{R}^{KF'}$.

k. Building Optimizers (using the Adam Optimizer in-built)
l. Training and Testing

3. DeepSNAP Graph
   a. DeepSNAP, a package that combines the benefits of both graph representations and offers a full pipeline for GNN training / validation / and testing. Namely, DeepSNAP includes a graph class representation to allow for more efficient graph manipulation and analysis in addition to a tensor-based representation for efficient message passing computation.
   b. In general, DeepSNAP is a Python library to assist efficient deep learning on graphs. DeepSNAP enables flexible graph manipulation, standard graph learning pipelines, heterogeneous graphs, and overall represents a simple graph learning API.
   c. In more detail:
      i. DeepSNAP allows for sophisticated graph manipulations, such as feature computation, pretraining, subgraph extraction etc. during/before training.
      ii. DeepSNAP standardizes the pipelines for node, edge, and graph-level prediction tasks under inductive or transductive settings. Specifically, DeepSNAP removes previous non-trivial / repetitive design choices left to the user, such as how to split datasets. DeepSNAP thus greatly saves repetitive often non-trivial coding efforts and enables fair model comparison.
      iii. Many real-world graphs are heterogeneous in nature (i.e., include different node types or edge types). However, most packages lack complete support for heterogeneous graphs, including data storage and flexible message passing. DeepSNAP provides an efficient and flexible heterogeneous graph that supports both node and edge heterogeneity.
   d. We focus on working with DeepSNAP for graph manipulation and dataset splitting.
   e. Setup:

```
import torch
import networkx as nx
import matplotlib.pyplot as plt

from deepsnap.graph import Graph
from deepsnap.batch import Batch
from deepsnap.dataset import GraphDataset
from torch_geometric.datasets import Planetoid, TUDataset

from torch.utils.data import DataLoader

def visualize(G, color_map=None, seed=123):
  if color_map is None:
    color_map = '#c92506'
  plt.figure(figsize=(8, 8))
  nodes = nx.draw_networkx_nodes(G, pos=nx.spring_layout(G, seed=seed), \
                                 label=None, node_color=color_map, node_shape='o', node_size=150)
  edges = nx.draw_networkx_edges(G, pos=nx.spring_layout(G, seed=seed), alpha=0.5)
  if color_map is not None:
    plt.scatter([],[], c='#c92506', label='Nodes with label 0', edgecolors="black", s=140)
    plt.scatter([],[], c='#fcec00', label='Nodes with label 1', edgecolors="black", s=140)
    plt.legend(prop={'size': 13}, handletextpad=0)
  nodes.set_edgecolor('black')
  plt.show()
```

f. The `deepsnap.graph.Graph` class is the core class of DeepSNAP. It not only represents a graph in tensor format but also includes a graph object from a graph manipulation package. Currently DeepSNAP supports NetworkX and Snap.py as back-end graph manipulation packages. In this Colab, we will focus on using NetworkX as the back end graph manipulation package.

g. Converting NetworkX Graph to DeepSNAP Graph

h. DeepSNAP features and attributes, mainly at node and edge levels

i. Lastly, DeepSNAP provides functionality to automatically transform a PyG dataset into a list of DeepSNAP graphs.

    i.

```
# The Cora dataset
pyg_dataset= Planetoid(root, name)

# PyG dataset to a list of deepsnap graphs
graphs = GraphDataset.pyg_to_graphs(pyg_dataset)

# Get the first deepsnap graph (CORA only has one graph)
graph = graphs[0]
print(graph)
```

j. Creating a DeepSNAP Dataset:

    i. Now, we will learn how to create DeepSNAP datasets. A `deepsnap.dataset.GraphDataset` contains a list of `deepsnap.graph.Graph` objects. In addition to the list of graphs, we specify what task the dataset will be used on, such as node level task (`task=node`), edge level task (`task=link_pred`) and graph level task (`task=graph`).

    ii. The Graph Dataset class contains many other useful parameters that can be specified during initialization. If you are interested, you can take a look at the documentation.

    iii. As an example, we will first look at the COX2 dataset, which contains 467 graphs. In initializing our dataset, we convert the PyG dataset into its corresponding DeepSNAP dataset and specify the task to `graph`.

k. Exploring Data Split in DeepSNAP graphs:
    i. In general, data splitting is divided into two settings, **inductive** and **transductive**.
    ii. In an inductive setting, we split a list of multiple graphs into disjoint training/validation and test sets. E.g., Graph level Tasks
    iii. In the transductive setting, the training /validation / test sets are all over the same graph. E.g., Node Level Tasks
    iv. Edge Level Splitting: Compared to node and graph level splitting, edge level splitting is a little bit tricky ;)
        1. For edge level splitting we need to consider several different tasks:
        2. Splitting positive edges into train / validation / test datasets.
        3. Sampling / re-sampling negative edges (i.e., edges not present in the graph).
        4. Splitting edges into message passing and supervision edges.
        5. With regard to point 3, for edge level data splitting we classify edges into two types. The first is `message passing` edges, edges that are used for message passing by our GNN. The second is `supervision`, edges that are used in the loss function for backpropagation. DeepSNAP allows for two different modes, where the `message passing` and `supervision` edges are either the same or disjoint.
        6. All Edge Splitting Mode:
            a. First, we explore the `edge_train_mode="all"` mode for edge level splitting, where the `message passing` and `supervision` edges are shared during training.
        7. Disjoint Edge Splitting Mode:
            a. Now we will look at a relatively more complex transductive edge split setting, the `edge_train_mode="disjoint"` mode in DeepSNAP. In this setting, the `message passing` and `supervision` edges are completely disjoint
        8. For edge level tasks, sampling negative edges is critical. Moreover, during each training iteration, we want to resample the negative edges.
        9. The other core functionality of DeepSNAP is graph transformation / feature computation.
        10. In DeepSNAP, we divide graph transformation / feature computation into two different types. The first includes transformations before training (e.g., transform the whole dataset before training directly), and the second includes transformations during training (transform batches of graphs).
            a. An Example that uses the NetworkX back end to calculate the PageRank value for each node and subsequently transforms the node features by concatenating each node's PageRank score (transform the dataset before training).

4. Edge Predictions

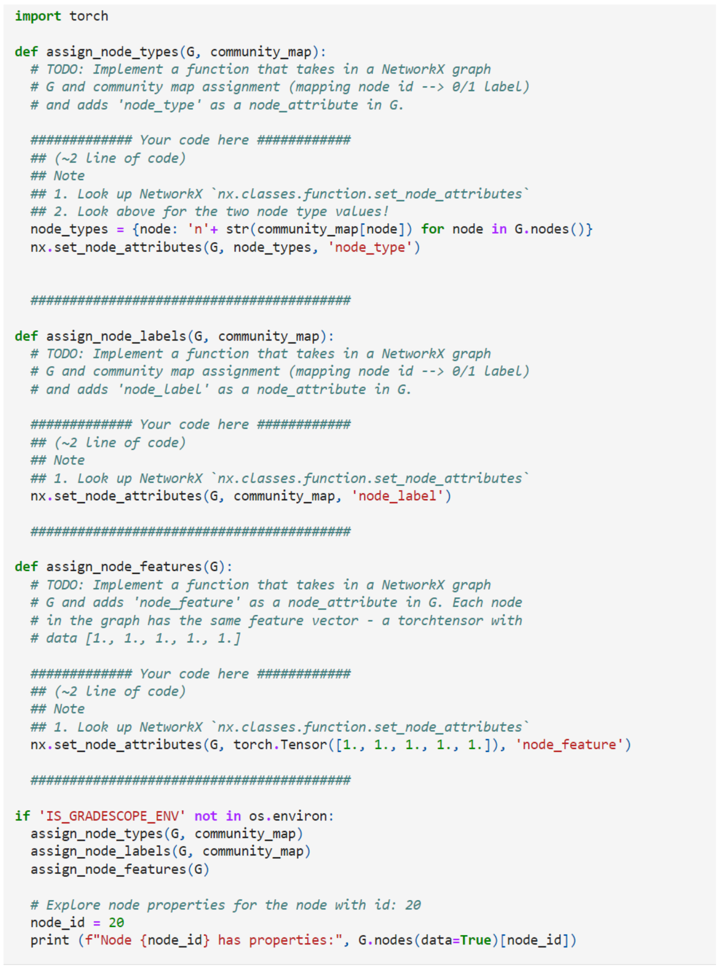
COLAB 4:

In this Colab, we will shift our focus from homogenous graphs to heterogeneous graphs. Heterogeneous graphs extend the traditional homogenous graphs by incorporating different node and edge types. This additional information allows us to extend the graph neural network models that we have worked with before. Namely, we can apply heterogenous message passing, where different message types now exist between different node and edge type relationships.

In this colab, we learn how to transform NetworkX graphs into DeepSNAP representations. Then, we will dive deeper into how DeepSNAP stores and represents heterogeneous graphs as PyTorch Tensors.

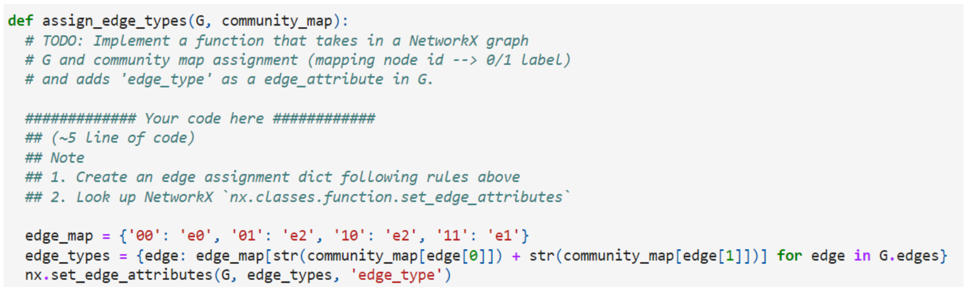
Lastly, we will build our own heterogenous graph neural network models using PyTorch Geometric and DeepSNAP. We will then apply our models for a node property prediction task; specifically, we will evaluate these models on the heterogeneous ACM node prediction dataset.

1. DeepSNAP Heterogeneous Graphs:
    a. DeepSNAP extends its traditional graph representation to include heterogeneous graphs by including the following graph property features:
        i. `node_feature`: The feature of each node (`torch.tensor`)
        ii. `edge_feature`: The feature of each edge (`torch.tensor`)
        iii. `node_label`: The label of each node (`int`)
        iv. `node_type`: The node type of each node (`string`)
        v. `edge_type`: The edge type of each edge (`string`)
            1. where the key **new** features we add are `node_type` and `edge_type`, which enables us to perform heterogenous message passing.
    b. Differentiating Node Types through `node_type`
    c. Assigning Node Types and Node Labels using nx.classes.function.set_node_attributes function (Sets node attributes from a given value or dictionary of values).



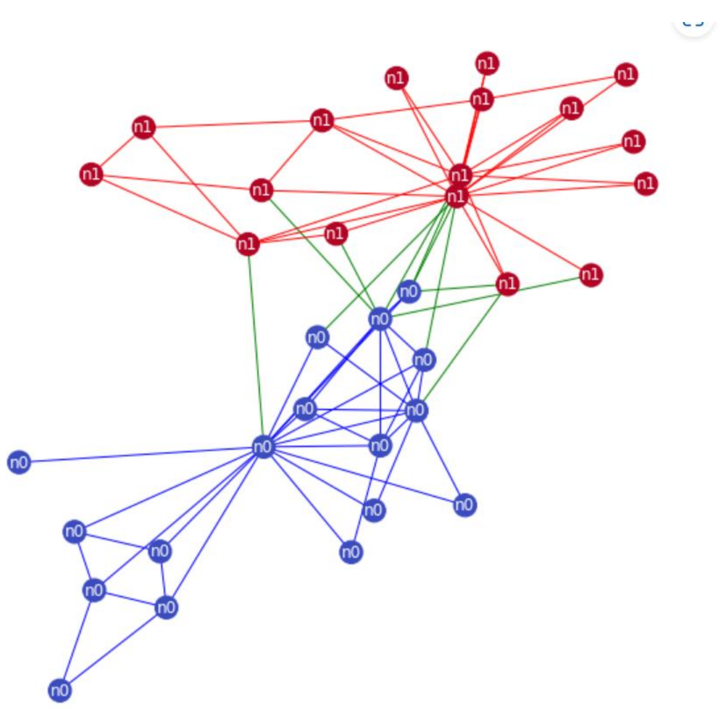
```python
import torch

def assign_node_types(G, community_map):
  # TODO: Implement a function that takes in a NetworkX graph
  # G and community map assignment (mapping node id --> 0/1 label)
  # and adds 'node_type' as a node_attribute in G.

  ############# Your code here ############
  ## (~2 line of code)
  ## Note
  ## 1. Look up NetworkX `nx.classes.function.set_node_attributes`
  ## 2. Look above for the two node type values!
  node_types = {node: 'n'+ str(community_map[node]) for node in G.nodes()}
  nx.set_node_attributes(G, node_types, 'node_type')


  #########################################

def assign_node_labels(G, community_map):
  # TODO: Implement a function that takes in a NetworkX graph
  # G and community map assignment (mapping node id --> 0/1 label)
  # and adds 'node_label' as a node_attribute in G.

  ############# Your code here ############
  ## (~2 line of code)
  ## Note
  ## 1. Look up NetworkX `nx.classes.function.set_node_attributes`
  nx.set_node_attributes(G, community_map, 'node_label')

  #########################################

def assign_node_features(G):
  # TODO: Implement a function that takes in a NetworkX graph
  # G and adds 'node_feature' as a node_attribute in G. Each node
  # in the graph has the same feature vector - a torchtensor with
  # data [1., 1., 1., 1., 1.]

  ############# Your code here ############
  ## (~2 line of code)
  ## Note
  ## 1. Look up NetworkX `nx.classes.function.set_node_attributes`
  nx.set_node_attributes(G, torch.Tensor([1., 1., 1., 1., 1.]), 'node_feature')

  #########################################

if 'IS_GRADESCOPE_ENV' not in os.environ:
  assign_node_types(G, community_map)
  assign_node_labels(G, community_map)
  assign_node_features(G)

  # Explore node properties for the node with id: 20
  node_id = 20
  print (f"Node {node_id} has properties:", G.nodes(data=True)[node_id])
```

d. Assigning Edge Types using nx.classes.function.set_edge_attributes (Sets edge attributes from a given value or dictionary of values).



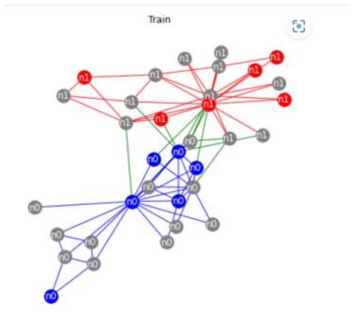
```python
def assign_edge_types(G, community_map):
  # TODO: Implement a function that takes in a NetworkX graph
  # G and community map assignment (mapping node id --> 0/1 label)
  # and adds 'edge_type' as a edge_attribute in G.

  ############# Your code here ############
  ## (~5 line of code)
  ## Note
  ## 1. Create an edge assignment dict following rules above
  ## 2. Look up NetworkX `nx.classes.function.set_edge_attributes`

  edge_map = {'00': 'e0', '01': 'e2', '10': 'e2', '11': 'e1'}
  edge_types = {edge: edge_map[str(community_map[edge[0]]) + str(community_map[edge[1]])] for edge in G.edges}
  nx.set_edge_attributes(G, edge_types, 'edge_type')
```

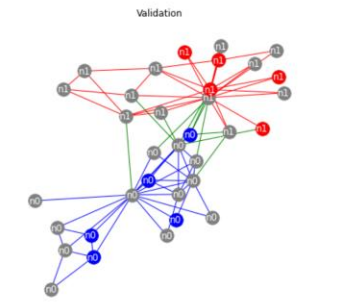
e. Visualization using Matplotlib and transforming NetworkX Object G to DeepSNAP Heterogeneous graph representation using deepsnap.hetero_graph.HeteroGraph

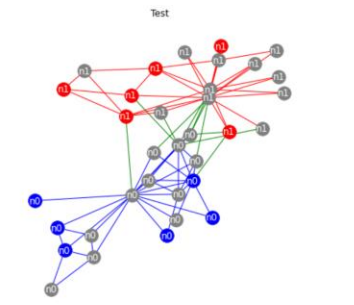


f. Node Types, Message Types and Dataset Split using prior techniques

g. Visualizing Train, Test and Validation sets

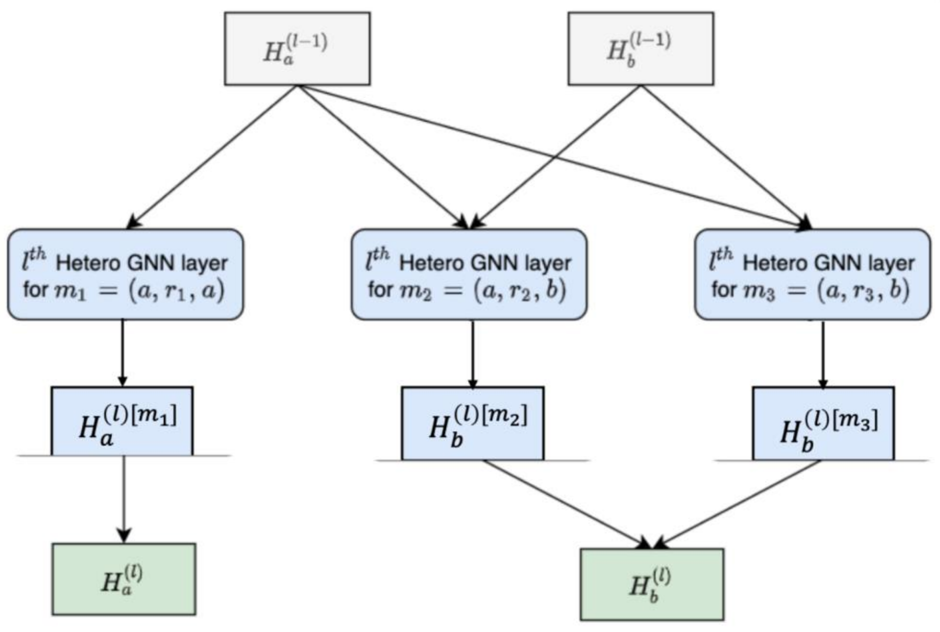






2. Heterogeneous Graph Node Property Prediction
    a. Using PyTorch Geometric and DeepSNAP we implement a GNN model for heterogeneous graph node property prediction (node classification). We will draw upon our understanding of heterogeneous graphs from lecture and previous work in implementing GNN layers using PyG (introduced in Colab 3).
    b. General structure of a heterogeneous GNN layer by working through an example:
    c. Let's assume we have a graph G, which contains two node types a and b, and three message types $m1=(a,r1,a)$, $m2=(a,r2,b)$ and $m3=(a,r3,b)$. Note: during message passing we view each message as (source, relation, destination), where messages "flow" from src to dst node types. For example, during message passing, updating node type b relies on two different message types $m2$ and $m3$.
    d. When applying message passing in heterogenous graphs, we separately apply message passing over each message type. Therefore, for the graph, a heterogeneous GNN layer contains three separate Heterogeneous Message Passing layers (`HeteroGNNConv` in this Colab), where each `HeteroGNNConv` layer performs message passing and aggregation with respect to *only one message type*. Since a message type is viewed as (src, relation, dst) and messages "flow" from src to dst, each `HeteroGNNConv` layer only computes embeddings for the *dst* nodes of a given message type. For example, the `HeteroGNNConv` layer for message type outputs updated embedding representations *only* for nodes with type b.

An overview of the heterogeneous layer:



$H_a^{(l-1)}$
$H_b^{(l-1)}$
l^{th} Hetero GNN layer for $m_1 = (a, r_1, a)$
l^{th} Hetero GNN layer for $m_2 = (a, r_2, b)$
l^{th} Hetero GNN layer for $m_3 = (a, r_3, b)$
$H_a^{(l)[m_1]}$
$H_b^{(l)[m_2]}$
$H_b^{(l)[m_3]}$
$H_a^{(l)}$
$H_b^{(l)}$

- $H_a^{(l)[m_1]}$ is the intermediate matrix of of node embeddings for node type a, generated by the lth `HeteroGNNConv` layer for message type m_1.
- $H_a^{(l)}$ is the matrix with current embeddings for nodes of type a after the lth layer of our Heterogeneous GNN model. Note that these embeddings can rely on one or more intermediate `HeteroGNNConv` layer embeddings(i.e. $H_b^{(l)}$ combines $H_b^{(l)[m_2]}$ and $H_b^{(l)[m_3]}$).

e. Implementation Steps:
   i. Implement `HeteroGNNConv`.
   ii. Implement **just** `mean` aggregation within `HeteroGNNWrapperConv`.
   iii. Implement `generate_convs`.
   iv. Implement the `HeteroGNN` model and the `train` function.
   v. Train the model with `mean` aggregation and test your model to make sure your model has reasonable performance.
   vi. Once you are confident in your mean aggregation model, implement `attn` aggregation in `HeteroGNNWrapperConv`.
   vii. Train the model with `attn` aggregation and test your model to make sure your model has reasonable performance.

We begin by defining the `HeteroGNNConv` layer with respect to message type m:

$$m = (s, r, d)$$

The message passing update rule that we implement is very similar to that of GraphSAGE, except we now need to include the node types and the edge relation type. The update rule for message type m is described below:

$$h_v^{(l)[m]} = W^{(l)[m]} \cdot \mathrm{CONCAT}\Big(W_d^{(l)[m]} \cdot h_v^{(l-1)}, W_s^{(l)[m]} \cdot AGG(\{h_u^{(l-1)}, \forall u \in N_m(v)\})\Big) \qquad (2)$$

- $W_s^{(l)[m]}$ - linear transformation matrix for the messages of neighboring source nodes of type s along message type m.
- $W_d^{(l)[m]}$ - linear transformation matrix for the message from the node v itself of type d.
- $W^{(l)[m]}$ - linear transformation matrix for the concatenated messages from neighboring node's and the central node.
- $h_u^{(l-1)}$ - the hidden embedding representation for node u after the $(l-1)$th `HeteroGNNWrapperConv` layer. Note, that this embedding is not associated with a particular message type (see layer diagrams above).
- $N_m(v)$ - the set of neighbor source nodes s for the node v that we are embedding along message type $m = (s, r, d)$.

Lastly, for simplicity, we use mean aggregations for AGG where:

$$AGG(\{h_u^{(l-1)}, \forall u \in N_m(v)\}) = \frac{1}{|N_m(v)|} \sum_{u \in N_m(v)} h_u^{(l-1)}$$

f. Applying a GNN Wrapper to manage and aggregate node embedding results

  i. Option 1: Simple Mean Aggregation

  where node v has node type d and we sum over the M message types that have destination node type d.

  The first one is simply mean aggregation over message types:

$$h_v^{(l)} = \frac{1}{M} \sum_{m=1}^{M} h_v^{(l)[m]}$$

  ii. Option 2: Semantic Level Attention

Instead of directly averaging on the message type aggregation results, we use attention to learn which message type result is more important, then aggregate across all the message types.

$$e_m = \frac{1}{|V_d|} \sum_{v \in V_d} q_{attn}^T \cdot tanh\Big(W_{attn}^{(l)} \cdot h_v^{(l)[m]} + b\Big)$$

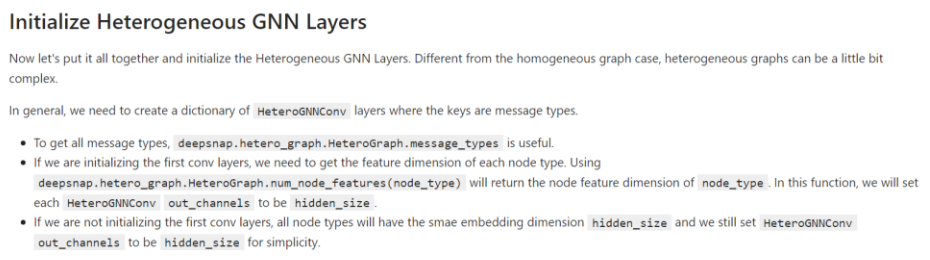
  iii. Computing the normalized attention weights and update $hv(l)$:

$$\alpha_m = \frac{\exp(e_m)}{\sum_{m=1}^{M} \exp(e_m)}$$

$$h_v^{(l)} = \sum_{m=1}^{M} \alpha_m \cdot h_v^{(l)[m]}$$

  iv. Implementation: Initializing the Sequential Model using two linear layers and tanh, then setting parameters and constructing the feed-forward layers in the network and lastly building the aggregate function using torch.stack() and torch.mean() and translating the equation above into code.

  v. Initializing the Heterogeneous layers:



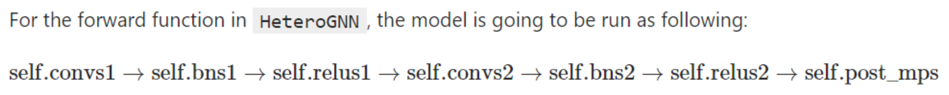
Initialize Heterogeneous GNN Layers

Now let's put it all together and initialize the Heterogeneous GNN Layers. Different from the homogeneous graph case, heterogeneous graphs can be a little bit complex.

In general, we need to create a dictionary of `HeteroGNNConv` layers where the keys are message types.

- To get all message types, `deepsnap.hetero_graph.HeteroGraph.message_types` is useful.
- If we are initializing the first conv layers, we need to get the feature dimension of each node type. Using `deepsnap.hetero_graph.HeteroGraph.num_node_features(node_type)` will return the node feature dimension of `node_type`. In this function, we will set each `HeteroGNNConv` `out_channels` to be `hidden_size`.
- If we are not initializing the first conv layers, all node types will have the smae embedding dimension `hidden_size` and we still set `HeteroGNNConv` `out_channels` to be `hidden_size` for simplicity.

vi. Creating the Model using the forward function:



For the forward function in `HeteroGNN`, the model is going to be run as following:

$$\text{self.convs1} \rightarrow \text{self.bns1} \rightarrow \text{self.relus1} \rightarrow \text{self.convs2} \rightarrow \text{self.bns2} \rightarrow \text{self.relus2} \rightarrow \text{self.post\_mps}$$

vii. Training and Testing

viii. Dataset and Preprocessing

ix. In the next, we will load the data and create a tensor backend (without a NetworkX graph) `deepsnap.hetero_graph.HeteroGraph` object.

3. Training and Testing mean aggregation

COLAB 5:

In this final Colab we will experiment with advanced topics in GNNs. Specifically, we will look at different techniques for scaling up GNNs using PyTorch Geometric, DeepSNAP and NetworkX.

First, we will work with PyTorch Geometric's `NeighborSampler` to scale up training and testing on the OGB `arxiv` dataset.

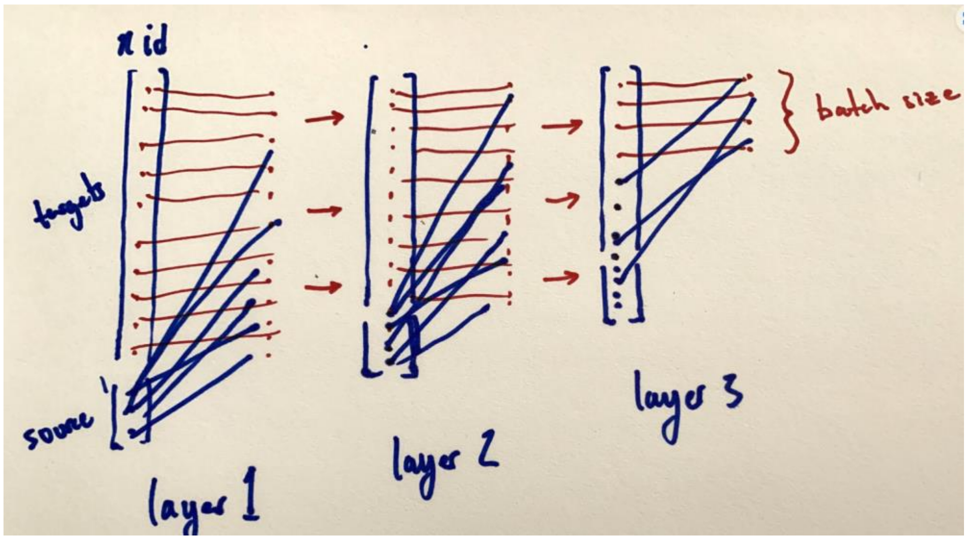
Then, using DeepSNAP and NetworkX, we will implement our own simplified version of `NeighborSampler` and run experiments with different sampling ratios on the Cora graph.

Lastly, we will partition the Cora graph into clusters by using different partition algorithms and then train the models using a vanilla Cluster-GCN.

1. Neighborhood Sampling:
   a. Definition: Neighbor Sampling, originally proposed in **GraphSAGE** (Hamilton et al. (2017)), is a representative method to scale up GNNs. As we learned in the lectures, rather than loading the entire graph into memory for each training loop, we can instead sample a mini-batch of the nodes we want to embed and **only** load the K-hop graph neighborhoods needed to embed these nodes. In this way we take advantage of the fact that the embedding of a node u only depends on its K-hop neighborhood. To further reduce the memory footprint and computational cost, we can choose to sample only a subset of a node's neighborhood during message passing and aggregation.
   b. Setup:

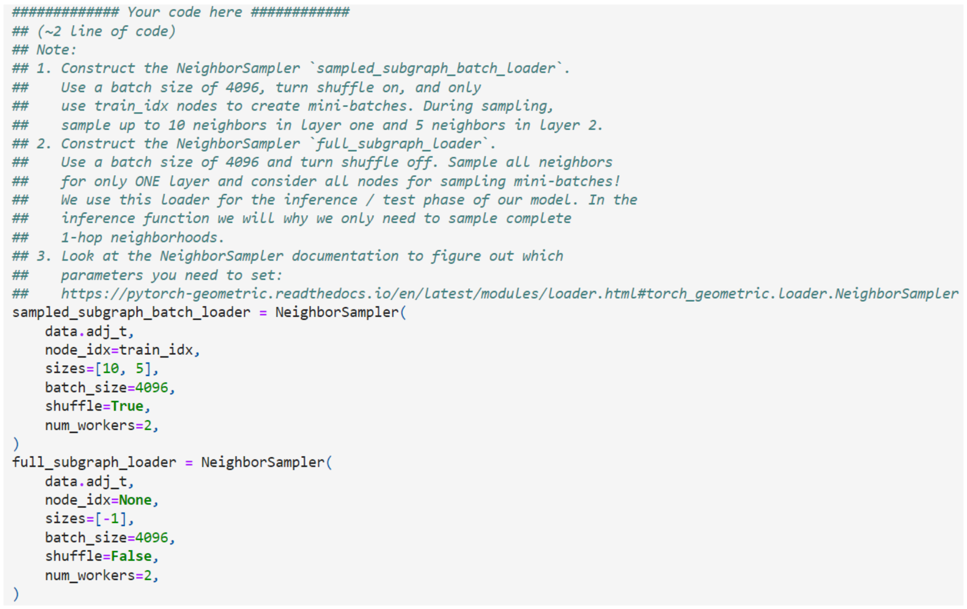
```
import copy
import torch
import torch.nn.functional as F
import torch_geometric.transforms as T

from torch_geometric.nn import SAGEConv
from torch_geometric.loader import NeighborSampler
from ogb.nodeproppred import PygNodePropPredDataset, Evaluator
```

c. PyTorch Geometric has implemented Neighbor Sampling through the NeighborSampler class. Neighbor sampling is based on building a node's computation graph without storing irrelevant information for a given node, thus making it more efficient. Each node produces a single computation graph, where for each node in a k-hop neighborhood, at most, Hk neighbors are randomly sampled. Each node's computation graph will therefore involve $\prod k=1KHk$ leaf nodes for a K-layer GNN. The successive layers of each node's computation graph can be conceptualized as bi-partite graphs that represent each layer of message passing in the GNN as shown in figure below. The blue (or black) dots are source nodes residing at layer $k-1$, while the red dots represent target nodes at subsequent layer k. It is important to stress the target nodes that are embedded in each layer are the final nodes in the left-hand side of the bi-partite graph. Node computation graphs are combined when subsequently forming a batch. If you'd like to learn more about information on neighborhood sampling, this **blog** provides an excellent description.



d. Py Docs:
   i. The neighbor sampler from the "Inductive Representation Learning on Large Graphs" paper, which allows for mini-batch training of GNNs on large-scale graphs where full-batch training is not feasible.
   ii. Given a GNN with layers and a specific mini batch of nodes node_idx for which we want to compute embeddings, this module iteratively samples neighbors and constructs bipartite graphs that simulate the actual computation flow of GNNs.

iii. More specifically, sizes denote how many neighbors we want to sample for each node in each layer. This module then takes in these sizes and iteratively samples sizes[l] for each node involved in layer l. In the next layer, sampling is repeated for the union of nodes that were already encountered. The actual computation graphs are then returned in reverse-mode, meaning that we pass messages from a larger set of nodes to a smaller one, until we reach the nodes for which we originally wanted to compute embeddings.



```
############# Your code here ############
## (~2 line of code)
## Note:
## 1. Construct the NeighborSampler `sampled_subgraph_batch_loader`.
##    Use a batch size of 4096, turn shuffle on, and only
##    use train_idx nodes to create mini-batches. During sampling,
##    sample up to 10 neighbors in layer one and 5 neighbors in layer 2.
## 2. Construct the NeighborSampler `full_subgraph_loader`.
##    Use a batch size of 4096 and turn shuffle off. Sample all neighbors
##    for only ONE layer and consider all nodes for sampling mini-batches!
##    We use this loader for the inference / test phase of our model. In the
##    inference function we will why we only need to sample complete
##    1-hop neighborhoods.
## 3. Look at the NeighborSampler documentation to figure out which
##    parameters you need to set:
##    https://pytorch-geometric.readthedocs.io/en/latest/modules/loader.html#torch_geometric.loader.NeighborSampler
sampled_subgraph_batch_loader = NeighborSampler(
    data.adj_t,
    node_idx=train_idx,
    sizes=[10, 5],
    batch_size=4096,
    shuffle=True,
    num_workers=2,
)
full_subgraph_loader = NeighborSampler(
    data.adj_t,
    node_idx=None,
    sizes=[-1],
    batch_size=4096,
    shuffle=False,
    num_workers=2,
)
```

e. Implementing GNN Model for mini-batch training:

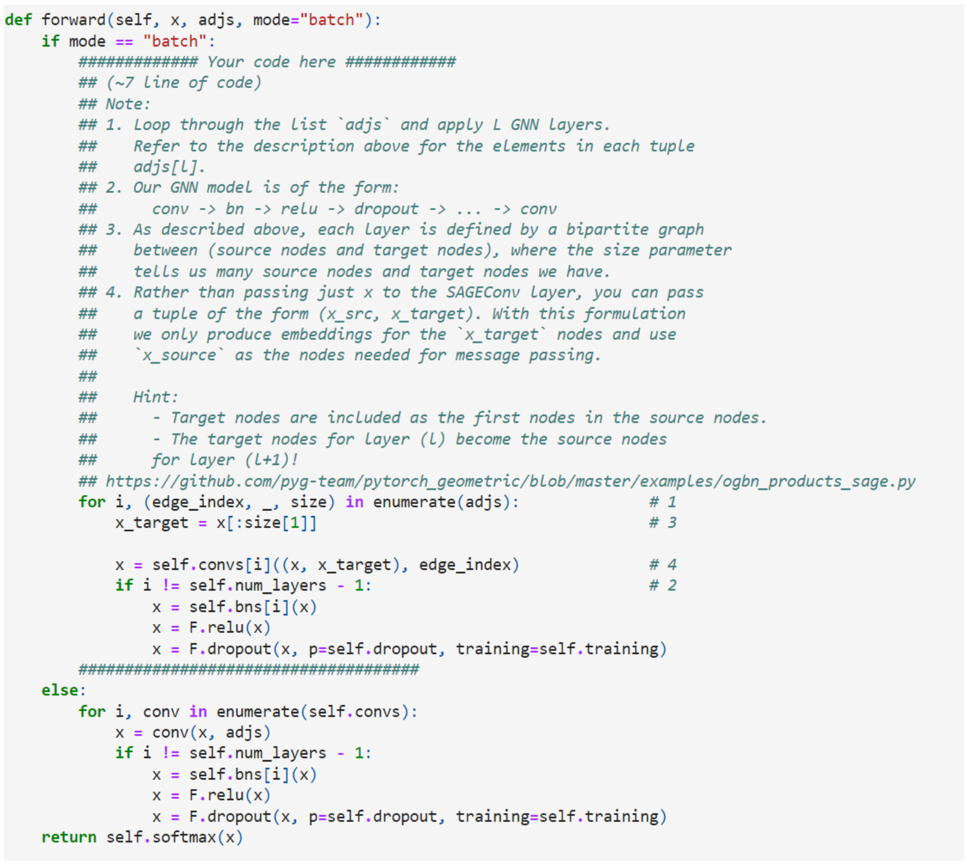
i. The `forward` function will take the node feature `x` and a list of three-element tuples `adjs`. Each element in `adjs` contains the following elements:

ii. `edge_index`: The edge index tensor between source and destination nodes, which forms a bipartite graph.

iii. `e_id`: The indices of the edges in the original graph.

iv. `size`: The shape of the bipartite graph, in (*number of source nodes*, *number of destination nodes*) format.

v. Implementation is like Colab 4



```python
def forward(self, x, adjs, mode="batch"):
    if mode == "batch":
        ############# Your code here ############
        ## (~7 line of code)
        ## Note:
        ## 1. Loop through the list `adjs` and apply L GNN layers.
        ##    Refer to the description above for the elements in each tuple
        ##    adjs[l].
        ## 2. Our GNN model is of the form:
        ##      conv -> bn -> relu -> dropout -> ... -> conv
        ## 3. As described above, each layer is defined by a bipartite graph
        ##    between (source nodes and target nodes), where the size parameter
        ##    tells us many source nodes and target nodes we have.
        ## 4. Rather than passing just x to the SAGEConv layer, you can pass
        ##    a tuple of the form (x_src, x_target). With this formulation
        ##    we only produce embeddings for the `x_target` nodes and use
        ##    `x_source` as the nodes needed for message passing.
        ##
        ##    Hint:
        ##      - Target nodes are included as the first nodes in the source nodes.
        ##      - The target nodes for layer (l) become the source nodes
        ##      for layer (l+1)!
        ## https://github.com/pyg-team/pytorch_geometric/blob/master/examples/ogbn_products_sage.py
        for i, (edge_index, _, size) in enumerate(adjs):              # 1
            x_target = x[:size[1]]                                    # 3

            x = self.convs[i]((x, x_target), edge_index)              # 4
            if i != self.num_layers - 1:                              # 2
                x = self.bns[i](x)
                x = F.relu(x)
                x = F.dropout(x, p=self.dropout, training=self.training)
        #####################################
    else:
        for i, conv in enumerate(self.convs):
            x = conv(x, adjs)
            if i != self.num_layers - 1:
                x = self.bns[i](x)
                x = F.relu(x)
                x = F.dropout(x, p=self.dropout, training=self.training)
    return self.softmax(x)
```

f. Training and Testing:
   i. In both training and testing, we need to sample batches from the data loader.
   ii. Each batch in the `NeighborSampler` data loader holds three elements:
   iii. `batch_size`: The batch size specified in the data loader.
   iv. `n_id`: All nodes (in index format) used in the adjacency matrices.
   v. `adjs`: The three-element tuples.

g. Testing model on mini batch using Negative Sampling

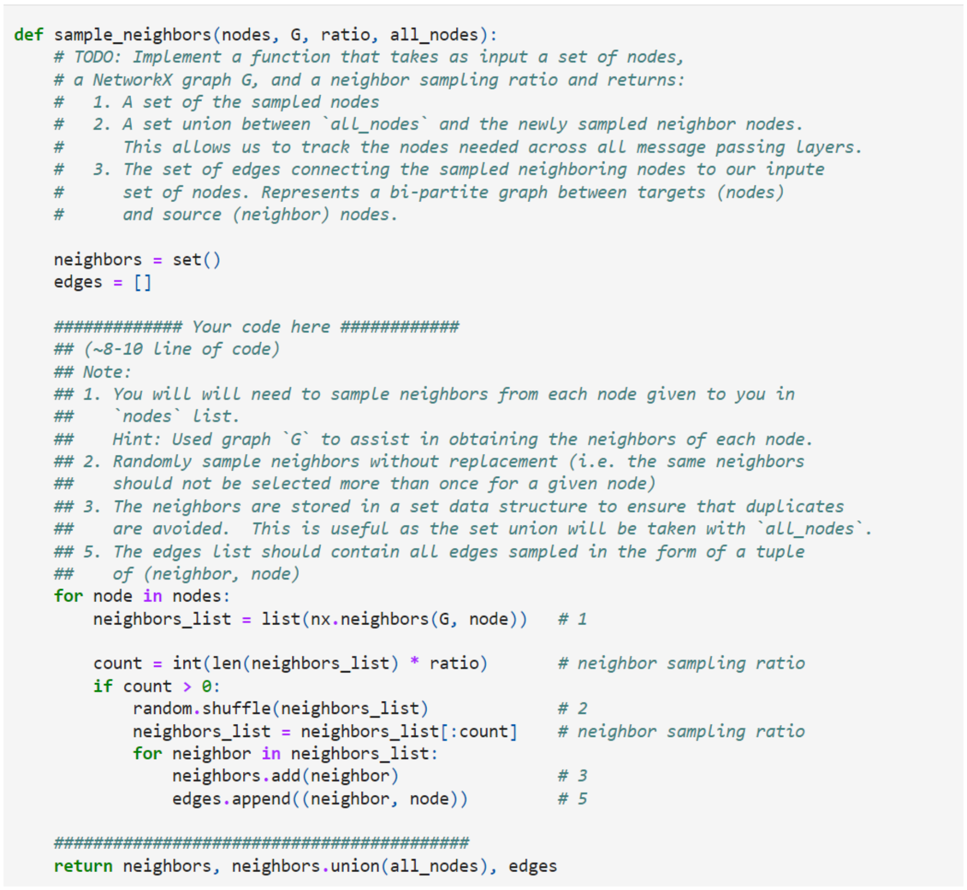
h. Testing model on full batch

i. Visualization

2. Neighborhood Sampling with Different Ratios
   a. Implementing simplified version of Neighbor Sampling using DeepSNAP and NetworkX. Then we will use our sampler to train models with different neighborhood sampling ratios and compare their performance.
   b. To make our experiments faster, we will use the Cora graph.
   c. Setup:

```python
import copy
import torch
import random
import numpy as np
import networkx as nx
import torch.nn as nn
import torch.nn.functional as F

from torch_geometric.nn import SAGEConv
from torch.utils.data import DataLoader
from torch_geometric.datasets import Planetoid
from torch.nn import Sequential, Linear, ReLU
from deepsnap.dataset import GraphDataset
from deepsnap.graph import Graph
```

d. Implementing GNN Model:
   Using simple GraphSage GNN model. Similar to in section one, notice the slightly different implementations of the forward method depending on the data `mode`. When `mode = "batch"` we use Neighbor sampling. Thus, the data parameter contains our graphs node features (x) and a list `edge_indeces` containing the connectivity of each GNN layer (i.e., an edge_index for each layer, defining the bipartite neighborhood computation graph).
e. Implementing Neighborhood Sampling using DeepSNAP and NetworkX:



```
def sample_neighbors(nodes, G, ratio, all_nodes):
    # TODO: Implement a function that takes as input a set of nodes,
    # a NetworkX graph G, and a neighbor sampling ratio and returns:
    #   1. A set of the sampled nodes
    #   2. A set union between `all_nodes` and the newly sampled neighbor nodes.
    #      This allows us to track the nodes needed across all message passing layers.
    #   3. The set of edges connecting the sampled neighboring nodes to our inpute
    #      set of nodes. Represents a bi-partite graph between targets (nodes)
    #      and source (neighbor) nodes.

    neighbors = set()
    edges = []

    ############# Your code here ############
    ## (~8-10 line of code)
    ## Note:
    ## 1. You will will need to sample neighbors from each node given to you in
    ##    `nodes` list.
    ##    Hint: Used graph `G` to assist in obtaining the neighbors of each node.
    ## 2. Randomly sample neighbors without replacement (i.e. the same neighbors
    ##    should not be selected more than once for a given node)
    ## 3. The neighbors are stored in a set data structure to ensure that duplicates
    ##    are avoided.  This is useful as the set union will be taken with `all_nodes`.
    ## 5. The edges list should contain all edges sampled in the form of a tuple
    ##    of (neighbor, node)
    for node in nodes:
        neighbors_list = list(nx.neighbors(G, node))   # 1

        count = int(len(neighbors_list) * ratio)       # neighbor sampling ratio
        if count > 0:
            random.shuffle(neighbors_list)             # 2
            neighbors_list = neighbors_list[:count]    # neighbor sampling ratio
            for neighbor in neighbors_list:
                neighbors.add(neighbor)                # 3
                edges.append((neighbor, node))         # 5

    ###########################################
    return neighbors, neighbors.union(all_nodes), edges
```

f. Tensor transformation and Node Relabeling
g. Training and Testing: Given a node classification task on Cora is a semi-supervised classification task, here we keep all the labeled training nodes (140 nodes) by setting the last ratio to 1.
h. Full-batch Training of GNN Model without using the Negative Sampler
i. Finding Test Accuracy with different sampling ratios

3. Cluster Sampling:
   a. In this final section, we implement a vanilla Cluster-GCN and experiment with 3 different community partition algorithms.
   b. Setup:

```python
import copy
import torch
import random
import numpy as np
import networkx as nx
import torch.nn as nn
import torch.nn.functional as F
import community.community_louvain as community_louvain

from torch_geometric.nn import SAGEConv
from torch.utils.data import DataLoader
from torch_geometric.datasets import Planetoid
from torch.nn import Sequential, Linear, ReLU
from deepsnap.dataset import GraphDataset
from deepsnap.graph import Graph

if 'IS_GRADESCOPE_ENV' not in os.environ:
  pyg_dataset = Planetoid('./tmp', "Cora")
```

c. We will experiment with three community detection / partition algorithms to partition our graph into different clusters:
   i. Kernighan–Lin algorithm (bisection)
   ii. Clauset-Newman-Moore greedy modularity maximization
   iii. Louvain algorithm
      1. As a preprocessing step, we partition our graph into a list of separate subgraphs using one of the three community detection algorithms above. Then during training, we iteratively train our vanilla Cluster-GNN model on a randomly selected subgraph, rather than on over the entire graph at once. To make training more stable, we discard any communities that have less than 10 nodes.

```python
def partition(G, method="louvain"):
    # TODO: Implement a function that takes a Networkx graph G and
    # partitions the graph into communities using the specified graph
    # partition algorithm.
    #
    # Return: A list of sets of nodes, one for each community!

    communities = None

    if method == "louvain":
        ############# Your code here #############
        ## (~9 line of code)
        ## Note:
        ## 1. Find a community mapping corresponding to the partition of the
        ##    graph nodes which maximizes the modularity for the Louvain algorithm.
        ##    Set your resolution to 10.
        ## 2. Create a mapping of communities to a set of member nodes.
        ## 3. Extract the node sets from each community and return
        ##    as a list of sets.
        ##    Hint: Perhaps a dictionary structure can assist.
        ## 4. SET random_state = 8!
        partitions = community_louvain.best_partition(
            G,
            resolution=10,
            random_state=8,
        )
        communities = {}
        for node in partitions:
            community = partitions[node]
            if community in communities:
                communities[community].add(node)
            else:
                communities[community] = set([node])
        communities = communities.values()

        ###########################################
    elif method == "bisection":
        ############# Your code here #############
        ## (~1 line of code)
        ## Note:
        ## 1. The Kernigan-Lin algorithm ensures that nodes are partitioned into two
        ##    primary communities.
        ## 2. Ensure that the resultant data structure is consistent with expected
        ##    output.
        ## 3. SET random_state = 8!
        communities = nx.algorithms.community.kernighan_lin_bisection(
            G=G,
            seed=8
        )

        ###########################################
    elif method == "greedy":
        ############# Your code here #############
        ## (~1 line of code)
        ## Note:
        ## 1. Clauset-Newman-Moore greedy modularity maximization joins pair
        ##    of communities nodes that most increases modularity until no such
        ##    pair exists.
        communities = nx.algorithms.community.greedy_modularity_communities(G)

        ###########################################

    return communities


def preprocess(G, node_label_index, method="louvain"):
    graphs = []
    labeled_nodes = set(node_label_index.tolist())

    communities = partition(G, method)

    for community in communities:
        nodes = set(community)
        subgraph = G.subgraph(nodes)
        # Make sure each subgraph has more than 10 nodes
        if subgraph.number_of_nodes() > 10:
            node_mapping = {node : i for i, node in enumerate(subgraph.nodes())}
            subgraph = nx.relabel_nodes(subgraph, node_mapping)
            # Get the id of the training set labeled node in the new graph
            train_label_index = []
            for node in labeled_nodes:
                if node in node_mapping:
                    # Append relabeled labeled node index
                    train_label_index.append(node_mapping[node])
```

d. Exploring and experimenting with different graph algorithms:

We experiment with the three graph partition algorithms, using the resulting graph clusters to train our vanilla Cluster-GNN. We will first observe how each partition algorithm partitions the original graph. Then we will compare the performance of our vanilla Cluster-GNN trained using the different graph clustering techniques. Lastly, we will compare against training a vanilla GCN over the entire graph (referred to as Full-Batch training).

i. How does the Louvain algorithm partition our graph?

1.
```
Partitioning the graph into 5 communities
Each community has 497 nodes on average
Each community has 973 edges on average
```

ii. Using Louvain clustering, what is the maximum test accuracy obtained by your vanilla Cluster-GCN?

1. `Best model: Train: 92.14%, Valid: 44.20% Test: 44.80%`

iii. How does the Bisection algorithm partition our graph?

1.
```
Partition the graph into 2 communities
Each community has 1354 nodes on average
Each community has 2397 edges on average
```

iv. Using the Bisection algorithm to partition the graph, what is the maximum test accuracy obtained by your vanilla Cluster-GCN?

1. `Best model: Train: 100.00%, Valid: 69.80% Test: 70.80%`

v. How does Greedy preprocessing partition our graph?

1.
```
Partition the graph into 20 communities
Each community has 121 nodes on average
Each community has 222 edges on average
```

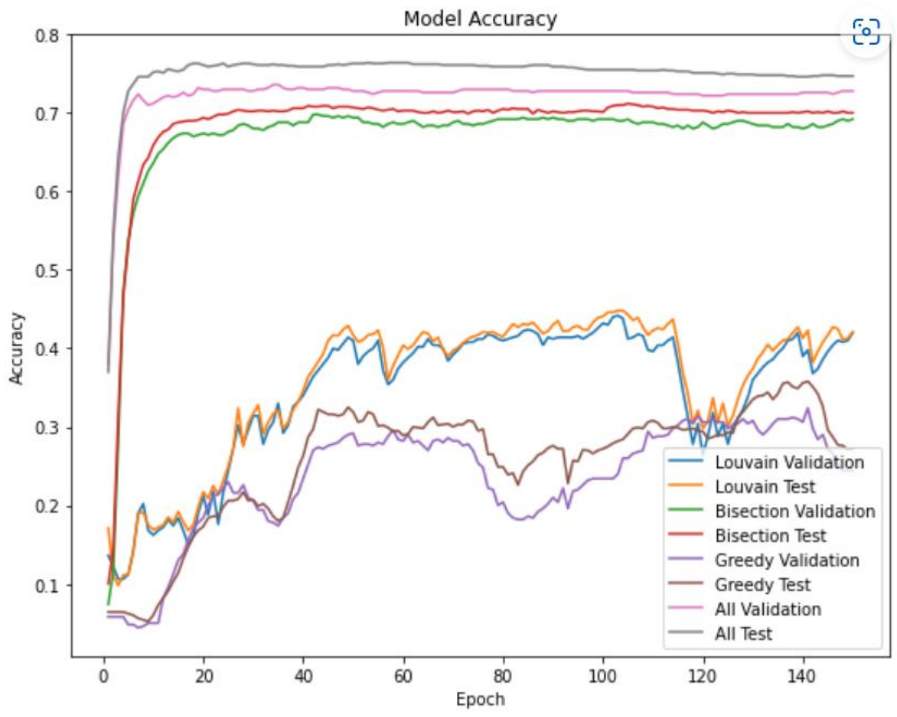
vi. Using Greedy preprocessing to partition the graph, what is the maximum test accuracy obtained by your vanilla Cluster-GCN?

1. `Best model: Train: 90.00%, Valid: 32.40% Test: 35.80%`

e. Full Batch Training using Adam Optimizer:

i. `Best model: Train: 100.00%, Valid: 73.60% Test: 76.20%`

f. Visualization:



4. Getting Predicted Accuracies and subsequent results