CS224W - 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.

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

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.

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.

Note: Make sure to **sequentially run all the cells in each section**, so that the intermediate variables / packages will carry over to the next cell

We recommend you save a copy of this colab in your drive so you don't lose progress!

Have fun and good luck on Colab 2 :)

Device

You might need to use a GPU for this Colab to run quickly.

Please click `Runtime` and then `Change runtime type`. Then set the `hardware accelerator` to **GPU**.

Setup

As discussed in Colab 0, the installation of PyG on Colab can be a little bit tricky. First let us check which version of PyTorch you are running

In [1]:
```
import torch
import os
print("PyTorch has version {}".format(torch.__version__))
```

```
PyTorch has version 1.10.2
```

Download the necessary packages for PyG. Make sure that your version of torch matches the output from the cell above. In case of any issues, more information can be found on the PyG's installation page.

In [2]:
```
# # Install torch geometric
```

```
# if 'IS_GRADESCOPE_ENV' not in os.environ:
#   !pip install torch-scatter -f https://pytorch-geometric.com/whl/torch-1.9
#   !pip install torch-sparse -f https://pytorch-geometric.com/whl/torch-1.9.
#   !pip install torch-geometric
#   !pip install ogb
```

1) PyTorch Geometric (Datasets and Data)

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.

In this section, we will learn how to use `torch_geometric.datasets` and `torch_geometric.data` together.

PyG Datasets

The `torch_geometric.datasets` class has many common graph datasets. Here we will explore its usage through one example dataset.

In [3]:

```
from torch_geometric.datasets import TUDataset

if 'IS_GRADESCOPE_ENV' not in os.environ:
  root = './enzymes'
  name = 'ENZYMES'

  # The ENZYMES dataset
  pyg_dataset= TUDataset(root, name)

  # You will find that there are 600 graphs in this dataset
  print(pyg_dataset)
```

```
/home/arch/anaconda3/envs/GNN_env/lib/python3.8/site-packages/torch/cuda/__in
it__.py:80: UserWarning: CUDA initialization: CUDA unknown error - this may b
e due to an incorrectly set up environment, e.g. changing env variable CUDA_V
ISIBLE_DEVICES after program start. Setting the available devices to be zero.
(Triggered internally at  /opt/conda/conda-bld/pytorch_1640811757556/work/c1
0/cuda/CUDAFunctions.cpp:112.)
  return torch._C._cuda_getDeviceCount() > 0
ENZYMES(600)
```

Question 1: What is the number of classes and number of features in the ENZYMES dataset? (5 points)

In [4]:

```
def get_num_classes(pyg_dataset):
  # TODO: Implement a function that takes a PyG dataset object
  # and returns the number of classes for that dataset.

    num_classes = 0

  ############# Your code here ############
  ## (~1 line of code)
  ## Note
  ## 1. Colab autocomplete functionality might be useful.
    num_classes = pyg_dataset.num_classes
```

```
    ##########################################

      return num_classes

  def get_num_features(pyg_dataset):
    # TODO: Implement a function that takes a PyG dataset object
    # and returns the number of features for that dataset.

      num_features = 0

    ############# Your code here ############
    ## (~1 line of code)
    ## Note
    ## 1. Colab autocomplete functionality might be useful.
      num_features = pyg_dataset.num_node_features
    ##########################################

      return num_features

  if 'IS_GRADESCOPE_ENV' not in os.environ:
      num_classes = get_num_classes(pyg_dataset)
      num_features = get_num_features(pyg_dataset)
      print("{} dataset has {} classes".format(name, num_classes))
      print("{} dataset has {} features".format(name, num_features))
```

```
ENZYMES dataset has 6 classes
ENZYMES dataset has 3 features
```

PyG Data

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 into the dataset.

For more information such as what is stored in the `Data` object, please refer to the documentation.

Question 2: What is the label of the graph with index 100 in the ENZYMES dataset? (5 points)

Acknowledgement: After getting stuck, I referenced code snippets fro the following notebook: https://github.com/luciusssss/CS224W-Colab/blob/main/CS224W-Colab%202.ipynb

In [22]:

```
  def get_graph_class(pyg_dataset, idx):
    # TODO: Implement a function that takes a PyG dataset object,
    # an index of a graph within the dataset, and returns the class/label
    # of the graph (as an integer).

      label = -1

    ############# Your code here ############
    ## (~1 line of code)
  #    label = pyg_dataset[idx].y.item
      label = pyg_dataset[idx].y[0]

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

      return label
```

```
# Here pyg_dataset is a dataset for graph classification
if 'IS_GRADESCOPE_ENV' not in os.environ:
    graph_0 = pyg_dataset[0]
    print(graph_0)
    idx = 100
    label = get_graph_class(pyg_dataset, idx)
    print('Graph with index {} has label {}'.format(idx, label))
```

```
Data(edge_index=[2, 168], x=[37, 3], y=[1])
Graph with index 100 has label 4
```

Question 3: How many edges does the graph with index 200 have? (5 points)

In [23]:

```
def get_graph_num_edges(pyg_dataset, idx):
  # TODO: Implement a function that takes a PyG dataset object,
  # the index of a graph in the dataset, and returns the number of
  # edges in the graph (as an integer). You should not count an edge
  # twice if the graph is undirected. For example, in an undirected
  # graph G, if two nodes v and u are connected by an edge, this edge
  # should only be counted once.

    num_edges = 0

  ############# Your code here ############
  ## Note:
  ## 1. You can't return the data.num_edges directly
  ## 2. We assume the graph is undirected
  ## 3. Look at the PyG dataset built in functions
  ## (~4 lines of code)
    graph = pyg_dataset[idx]
    edge_index = graph.edge_index.t()
    #print("edge_index: length {} values:\n{}".format(len(edge_index),edge_ir
    #print(edge_index[1])
    #print(list(edge_index[1]))
    sorted_edges, indices = torch.sort(edge_index,dim=1)

    edges = torch.unique(sorted_edges)
    num_edges = len(edges)

    print("original # edges:{}  unique_edges: {}".format(len(edge_index),num_
  #########################################

    return num_edges

if 'IS_GRADESCOPE_ENV' not in os.environ:
    idx = 200
    num_edges = get_graph_num_edges(pyg_dataset, idx)
    print('Graph with index {} has {} edges'.format(idx, num_edges))
```

```
original # edges:106  unique_edges: 29
Graph with index 200 has 29 edges
```

2) Open Graph Benchmark (OGB)

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.

Dataset and Data

OGB also supports PyG dataset and data classes. Here we take a look on the `ogbn-arxiv` dataset.

In [24]:
```
#!pip install ogb
!python -V
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
Python 3.8.8
```

In [25]:
```
import torch_geometric.transforms as T
from ogb.nodeproppred import PygNodePropPredDataset

if 'IS_GRADESCOPE_ENV' not in os.environ:
  dataset_name = 'ogbn-arxiv'
  # Load the dataset and transform it to sparse tensor
  dataset = PygNodePropPredDataset(name=dataset_name,
                                  transform=T.ToSparseTensor())
  print('The {} dataset has {} graph'.format(dataset_name, len(dataset)))

  # Extract the graph
  data = dataset[0]
  print(data)
```

```
The ogbn-arxiv dataset has 1 graph
Data(num_nodes=169343, x=[169343, 128], node_year=[169343, 1], y=[169343, 1],
adj_t=[169343, 169343, nnz=1166243])
```

Question 4: How many features are in the ogbn-arxiv graph? (5 points)

In [26]:
```
def graph_num_features(data):
  # TODO: Implement a function that takes a PyG data object,
  # and returns the number of features in the graph (as an integer).

    num_features = 0

  ############# Your code here ############
  ## (~1 line of code)
    num_features = data.num_node_features
  #########################################

    return num_features

if 'IS_GRADESCOPE_ENV' not in os.environ:
    num_features = graph_num_features(data)
    print('The graph has {} features'.format(num_features))
```

```
The graph has 128 features
```

3) GNN: Node Property Prediction

In this section we will build our first graph neural network using PyTorch Geometric. Then we will apply it to the task of node property prediction (node classification).

Specifically, we will use GCN as the foundation for your graph neural network (Kipf et al. (2017)). To do so, we will work with PyG's built-in `GCNConv` layer.

Setup

In [27]:

```
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
```

```
1.10.2
```

Load and Preprocess the Dataset

In [36]:

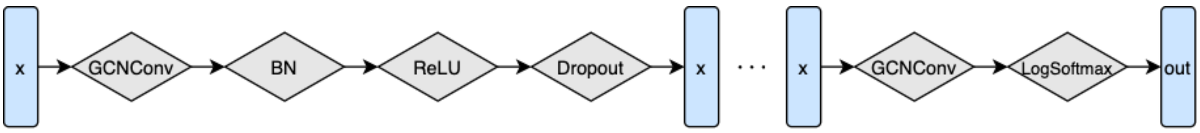
```
if 'IS_GRADESCOPE_ENV' not in os.environ:
    dataset_name = 'ogbn-arxiv'
    dataset = PygNodePropPredDataset(name=dataset_name,
                                  transform=T.ToSparseTensor())
    data = dataset[0]

  # 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)
```

```
Device: cpu
```

GCN Model

Now we will implement our GCN model!

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



In [44]:

```
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

        ############# Your code here ############
        ## 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' ar
        ## 'out_channels'. For more information please refer to the documenta
        ## https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html
        ## 6. The only parameter you need to set for BatchNorm1d is 'num_feat
        ## For more information please refer to the documentation:
        ## https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.htm
        ## (~10 lines of code)
        self.convs = torch.nn.ModuleList()

        self.convs.append(GCNConv(input_dim, hidden_dim))
        for layer in range(num_layers - 2):
            self.convs.append(GCNConv(hidden_dim,hidden_dim))
        self.convs.append(GCNConv(hidden_dim, output_dim))

        self.bns = torch.nn.ModuleList()
        for layer in range(num_layers - 1):
            self.bns.append(torch.nn.BatchNorm1d(hidden_dim))

        self.softmax = torch.nn.LogSoftmax()

        self.relu = torch.nn.ReLU()

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

        # Probability of an element getting zeroed
        self.dropout = dropout

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

    def reset_parameters(self):
        for conv in self.convs:
            conv.reset_parameters()
        for bn in self.bns:
            bn.reset_parameters()

    def forward(self, x, adj_t):
        # TODO: Implement a function that takes the feature tensor x and
        # edge_index tensor adj_t and returns the output tensor as
        # shown in the figure.

        out = None

        ############# Your code here ############
        ## Note:
        ## 1. Construct the network as shown in the figure
        ## 2. torch.nn.functional.relu and torch.nn.functional.dropout are us
        ## For more information please refer to the documentation:
```

```
        ## https://pytorch.org/docs/stable/nn.functional.html
        ## 3. Don't forget to set F.dropout training to self.training
        ## 4. If return_embeds is True, then skip the last softmax layer
        ## (~7 lines of code)

## Some error with the following code!!  Need to debug!
## Used code snippet from https://github.com/luciusssss/CS224W-Colab/blob/mai
#          for layer in range(len(self.convs)-1):
#              x = self.convs[layer](x, adj_t)
#              x = self.bns[layer](x)
#              x = F.relu(x)
#              #x = self.softmax(x)
#              x = F.dropout(x,self.dropout,self.training)
#          x = self.convs

#          out = self.convs[-1](x, adj_t)
#          if not self.return_embed:
#              out=self.softmax(out)

        for conv, bn in zip(self.convs[:-1], self.bns):
            x1 = F.relu(bn(conv(x, adj_t)))
            if self.training:
                x1 = F.dropout(x1, p=self.dropout)
            x = x1
        x = self.convs[-1](x, adj_t)
        out = x if self.return_embeds else self.softmax(x)
        #########################################

        return out
```

In [45]:

```
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()
    output = model(data.x, data.adj_t)
    predictions = output[train_idx]
#     train_label = data.y[train_index,0]
    train_label = data.y[train_idx,0]
    loss = loss_fn(predictions, train_label)

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

    loss.backward()
    optimizer.step()

    return loss.item()
```

In [46]:

```
# 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)
    #########################################

    y_pred = out.argmax(dim=-1, keepdim=True)

    train_acc = evaluator.eval({
        'y_true': data.y[split_idx['train']],
        'y_pred': y_pred[split_idx['train']],
    })['acc']
    valid_acc = evaluator.eval({
        'y_true': data.y[split_idx['valid']],
        'y_pred': y_pred[split_idx['valid']],
    })['acc']
    test_acc = evaluator.eval({
        'y_true': data.y[split_idx['test']],
        'y_pred': y_pred[split_idx['test']],
    })['acc']

    if save_model_results:
      print ("Saving Model Predictions")

      data = {}
      data['y_pred'] = y_pred.view(-1).cpu().detach().numpy()

      df = pd.DataFrame(data=data)
      # Save locally as csv
      df.to_csv('ogbn-arxiv_node.csv', sep=',', index=False)


    return train_acc, valid_acc, test_acc
```

In [47]:

```
# Please do not change the args
if 'IS_GRADESCOPE_ENV' not in os.environ:
  args = {
      'device': device,
      'num_layers': 3,
      'hidden_dim': 256,
      'dropout': 0.5,
      'lr': 0.01,
      'epochs': 100,
  }
  args
```

In [48]:

```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  model = GCN(data.num_features, args['hidden_dim'],
              dataset.num_classes, args['num_layers'],
              args['dropout']).to(device)
  evaluator = Evaluator(name='ogbn-arxiv')
```

In [49]:

```
import copy
```

```python
if 'IS_GRADESCOPE_ENV' not in os.environ:
  # reset the parameters to initial random value
  model.reset_parameters()

  optimizer = torch.optim.Adam(model.parameters(), lr=args['lr'])
  loss_fn = F.nll_loss

  best_model = None
  best_valid_acc = 0

  for epoch in range(1, 1 + args["epochs"]):
    loss = train(model, data, train_idx, optimizer, loss_fn)
    result = test(model, data, split_idx, evaluator)
    train_acc, valid_acc, test_acc = result
    if valid_acc > best_valid_acc:
        best_valid_acc = valid_acc
        best_model = copy.deepcopy(model)
    print(f'Epoch: {epoch:02d}, '
          f'Loss: {loss:.4f}, '
          f'Train: {100 * train_acc:.2f}%, '
          f'Valid: {100 * valid_acc:.2f}% '
          f'Test: {100 * test_acc:.2f}%')
```

```
/tmp/ipykernel_146981/324804549.py:95: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an a
rgument.
  out = x if self.return_embeds else self.softmax(x)
Epoch: 01, Loss: 3.9266, Train: 22.49%, Valid: 27.99% Test: 25.83%
Epoch: 02, Loss: 2.2406, Train: 24.32%, Valid: 22.24% Test: 27.51%
Epoch: 03, Loss: 1.9299, Train: 26.57%, Valid: 25.52% Test: 24.73%
Epoch: 04, Loss: 1.7258, Train: 28.23%, Valid: 24.08% Test: 23.85%
Epoch: 05, Loss: 1.6177, Train: 33.29%, Valid: 20.51% Test: 17.49%
Epoch: 06, Loss: 1.5345, Train: 37.83%, Valid: 23.10% Test: 19.49%
Epoch: 07, Loss: 1.4730, Train: 38.18%, Valid: 22.79% Test: 19.58%
Epoch: 08, Loss: 1.4290, Train: 38.59%, Valid: 22.53% Test: 19.40%
Epoch: 09, Loss: 1.3909, Train: 42.04%, Valid: 28.20% Test: 25.56%
Epoch: 10, Loss: 1.3580, Train: 46.83%, Valid: 36.73% Test: 35.36%
Epoch: 11, Loss: 1.3238, Train: 49.14%, Valid: 40.92% Test: 43.60%
Epoch: 12, Loss: 1.2949, Train: 49.02%, Valid: 40.00% Test: 44.12%
Epoch: 13, Loss: 1.2760, Train: 49.92%, Valid: 42.27% Test: 46.75%
Epoch: 14, Loss: 1.2602, Train: 52.68%, Valid: 48.81% Test: 52.60%
Epoch: 15, Loss: 1.2387, Train: 55.37%, Valid: 53.39% Test: 56.13%
Epoch: 16, Loss: 1.2248, Train: 56.17%, Valid: 54.41% Test: 56.80%
Epoch: 17, Loss: 1.2120, Train: 55.66%, Valid: 53.43% Test: 56.40%
Epoch: 18, Loss: 1.1951, Train: 54.84%, Valid: 51.74% Test: 55.32%
Epoch: 19, Loss: 1.1836, Train: 54.42%, Valid: 50.66% Test: 54.38%
Epoch: 20, Loss: 1.1714, Train: 55.31%, Valid: 51.39% Test: 55.40%
Epoch: 21, Loss: 1.1615, Train: 57.48%, Valid: 54.92% Test: 58.62%
Epoch: 22, Loss: 1.1498, Train: 59.77%, Valid: 58.60% Test: 61.48%
Epoch: 23, Loss: 1.1433, Train: 61.28%, Valid: 60.67% Test: 63.09%
Epoch: 24, Loss: 1.1364, Train: 61.72%, Valid: 61.05% Test: 63.60%
Epoch: 25, Loss: 1.1259, Train: 62.05%, Valid: 61.27% Test: 63.85%
Epoch: 26, Loss: 1.1173, Train: 62.56%, Valid: 61.78% Test: 64.21%
Epoch: 27, Loss: 1.1108, Train: 63.82%, Valid: 63.44% Test: 65.33%
Epoch: 28, Loss: 1.1067, Train: 65.55%, Valid: 65.61% Test: 66.57%
Epoch: 29, Loss: 1.0997, Train: 67.02%, Valid: 67.15% Test: 66.73%
Epoch: 30, Loss: 1.0876, Train: 67.76%, Valid: 67.69% Test: 66.53%
Epoch: 31, Loss: 1.0833, Train: 68.01%, Valid: 68.04% Test: 67.13%
Epoch: 32, Loss: 1.0789, Train: 67.98%, Valid: 68.26% Test: 67.89%
Epoch: 33, Loss: 1.0726, Train: 67.81%, Valid: 67.97% Test: 68.12%
Epoch: 34, Loss: 1.0705, Train: 68.01%, Valid: 68.09% Test: 68.30%
Epoch: 35, Loss: 1.0658, Train: 68.68%, Valid: 68.78% Test: 68.65%
Epoch: 36, Loss: 1.0581, Train: 69.13%, Valid: 69.04% Test: 68.26%
Epoch: 37, Loss: 1.0564, Train: 69.54%, Valid: 69.27% Test: 68.16%
Epoch: 38, Loss: 1.0504, Train: 69.74%, Valid: 69.59% Test: 68.99%
Epoch: 39, Loss: 1.0492, Train: 69.96%, Valid: 69.78% Test: 69.41%
Epoch: 40, Loss: 1.0429, Train: 70.11%, Valid: 69.87% Test: 69.53%
```

```
Epoch: 41, Loss: 1.0399, Train: 70.36%, Valid: 70.03% Test: 69.33%
Epoch: 42, Loss: 1.0383, Train: 70.61%, Valid: 70.07% Test: 69.10%
Epoch: 43, Loss: 1.0323, Train: 70.80%, Valid: 70.21% Test: 69.10%
Epoch: 44, Loss: 1.0265, Train: 71.03%, Valid: 70.47% Test: 69.24%
Epoch: 45, Loss: 1.0267, Train: 71.08%, Valid: 70.53% Test: 69.70%
Epoch: 46, Loss: 1.0206, Train: 71.12%, Valid: 70.66% Test: 69.92%
Epoch: 47, Loss: 1.0198, Train: 71.15%, Valid: 70.64% Test: 70.09%
Epoch: 48, Loss: 1.0134, Train: 71.28%, Valid: 70.67% Test: 70.06%
Epoch: 49, Loss: 1.0091, Train: 71.35%, Valid: 70.55% Test: 69.48%
Epoch: 50, Loss: 1.0069, Train: 71.45%, Valid: 70.72% Test: 69.46%
Epoch: 51, Loss: 1.0026, Train: 71.54%, Valid: 70.98% Test: 70.05%
Epoch: 52, Loss: 1.0039, Train: 71.62%, Valid: 71.00% Test: 70.20%
Epoch: 53, Loss: 0.9997, Train: 71.64%, Valid: 71.00% Test: 70.01%
Epoch: 54, Loss: 0.9993, Train: 71.66%, Valid: 70.88% Test: 69.73%
Epoch: 55, Loss: 0.9954, Train: 71.66%, Valid: 70.83% Test: 69.73%
Epoch: 56, Loss: 0.9919, Train: 71.75%, Valid: 70.84% Test: 69.80%
Epoch: 57, Loss: 0.9912, Train: 71.89%, Valid: 70.89% Test: 69.79%
Epoch: 58, Loss: 0.9848, Train: 72.04%, Valid: 70.97% Test: 69.69%
Epoch: 59, Loss: 0.9840, Train: 72.15%, Valid: 70.86% Test: 69.69%
Epoch: 60, Loss: 0.9824, Train: 72.20%, Valid: 71.10% Test: 69.99%
Epoch: 61, Loss: 0.9796, Train: 72.17%, Valid: 71.27% Test: 70.18%
Epoch: 62, Loss: 0.9783, Train: 72.23%, Valid: 71.39% Test: 70.44%
Epoch: 63, Loss: 0.9740, Train: 72.24%, Valid: 71.39% Test: 70.45%
Epoch: 64, Loss: 0.9719, Train: 72.25%, Valid: 71.25% Test: 70.03%
Epoch: 65, Loss: 0.9719, Train: 72.21%, Valid: 70.94% Test: 69.61%
Epoch: 66, Loss: 0.9678, Train: 72.18%, Valid: 70.88% Test: 69.39%
Epoch: 67, Loss: 0.9661, Train: 72.28%, Valid: 70.97% Test: 69.51%
Epoch: 68, Loss: 0.9653, Train: 72.39%, Valid: 70.98% Test: 69.18%
Epoch: 69, Loss: 0.9688, Train: 72.49%, Valid: 71.20% Test: 69.86%
Epoch: 70, Loss: 0.9628, Train: 72.62%, Valid: 71.36% Test: 70.64%
Epoch: 71, Loss: 0.9572, Train: 72.75%, Valid: 71.29% Test: 70.79%
Epoch: 72, Loss: 0.9560, Train: 72.92%, Valid: 71.31% Test: 70.07%
Epoch: 73, Loss: 0.9551, Train: 72.84%, Valid: 70.72% Test: 68.81%
Epoch: 74, Loss: 0.9556, Train: 72.85%, Valid: 70.62% Test: 68.54%
Epoch: 75, Loss: 0.9528, Train: 72.96%, Valid: 71.17% Test: 69.74%
Epoch: 76, Loss: 0.9496, Train: 73.03%, Valid: 71.52% Test: 70.52%
Epoch: 77, Loss: 0.9471, Train: 73.03%, Valid: 71.57% Test: 70.79%
Epoch: 78, Loss: 0.9463, Train: 72.88%, Valid: 71.52% Test: 70.87%
Epoch: 79, Loss: 0.9418, Train: 72.87%, Valid: 71.43% Test: 70.94%
Epoch: 80, Loss: 0.9427, Train: 72.99%, Valid: 71.50% Test: 70.80%
Epoch: 81, Loss: 0.9404, Train: 73.04%, Valid: 71.54% Test: 70.68%
Epoch: 82, Loss: 0.9403, Train: 73.05%, Valid: 71.30% Test: 70.31%
Epoch: 83, Loss: 0.9377, Train: 73.24%, Valid: 71.27% Test: 70.18%
Epoch: 84, Loss: 0.9346, Train: 73.19%, Valid: 71.22% Test: 69.86%
Epoch: 85, Loss: 0.9334, Train: 73.23%, Valid: 71.32% Test: 70.09%
Epoch: 86, Loss: 0.9340, Train: 73.31%, Valid: 71.61% Test: 70.75%
Epoch: 87, Loss: 0.9361, Train: 73.40%, Valid: 71.70% Test: 71.14%
Epoch: 88, Loss: 0.9289, Train: 73.48%, Valid: 71.69% Test: 70.90%
Epoch: 89, Loss: 0.9275, Train: 73.43%, Valid: 71.55% Test: 70.35%
Epoch: 90, Loss: 0.9249, Train: 73.32%, Valid: 71.38% Test: 70.01%
Epoch: 91, Loss: 0.9263, Train: 73.35%, Valid: 71.63% Test: 70.45%
Epoch: 92, Loss: 0.9254, Train: 73.35%, Valid: 71.69% Test: 70.94%
Epoch: 93, Loss: 0.9239, Train: 73.46%, Valid: 71.75% Test: 70.82%
Epoch: 94, Loss: 0.9195, Train: 73.66%, Valid: 71.63% Test: 70.29%
Epoch: 95, Loss: 0.9191, Train: 73.76%, Valid: 71.51% Test: 70.18%
Epoch: 96, Loss: 0.9147, Train: 73.67%, Valid: 71.60% Test: 70.77%
Epoch: 97, Loss: 0.9145, Train: 73.62%, Valid: 71.81% Test: 71.24%
Epoch: 98, Loss: 0.9145, Train: 73.73%, Valid: 71.77% Test: 70.73%
Epoch: 99, Loss: 0.9120, Train: 73.88%, Valid: 71.79% Test: 70.82%
Epoch: 100, Loss: 0.9116, Train: 73.90%, Valid: 71.94% Test: 71.37%
```

Question 5: What are your `best_model` validation and test accuracies?(20 points)

Run the cell below to see the results of your best of model and save your model's predictions to a file named *ogbn-arxiv_node.csv*.

You can view this file by clicking on the *Folder* icon on the left side pannel. As in Colab 1, when you sumbit your assignment, you will have to download this file and attatch it to your submission.

In [50]:
```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  best_result = test(best_model, data, split_idx, evaluator, save_model_resul
  train_acc, valid_acc, test_acc = best_result
  print(f'Best model: '
        f'Train: {100 * train_acc:.2f}%, '
        f'Valid: {100 * valid_acc:.2f}% '
        f'Test: {100 * test_acc:.2f}%')
```

```
Saving Model Predictions
Best model: Train: 73.90%, Valid: 71.94% Test: 71.37%
/tmp/ipykernel_146981/324804549.py:95: UserWarning: Implicit dimension choice
for log_softmax has been deprecated. Change the call to include dim=X as an a
rgument.
  out = x if self.return_embeds else self.softmax(x)
```

4) GNN: Graph Property Prediction

In this section we will create a graph neural network for graph property prediction (graph classification).

Load and preprocess the dataset

In [51]:
```
from ogb.graphproppred import PygGraphPropPredDataset, Evaluator
from torch_geometric.data import DataLoader
from tqdm.notebook import tqdm

if 'IS_GRADESCOPE_ENV' not in os.environ:
  # Load the dataset
  dataset = PygGraphPropPredDataset(name='ogbg-molhiv')

  device = 'cuda' if torch.cuda.is_available() else 'cpu'
  print('Device: {}'.format(device))

  split_idx = dataset.get_idx_split()

  # Check task type
  print('Task type: {}'.format(dataset.task_type))
```

```
Device: cpu
Task type: binary classification
```

In [52]:
```
# Load the dataset splits into corresponding dataloaders
# We will train the graph classification task on a batch of 32 graphs
# Shuffle the order of graphs for training set
if 'IS_GRADESCOPE_ENV' not in os.environ:
  train_loader = DataLoader(dataset[split_idx["train"]], batch_size=32, shuf
  valid_loader = DataLoader(dataset[split_idx["valid"]], batch_size=32, shuf
  test_loader = DataLoader(dataset[split_idx["test"]], batch_size=32, shuffle
```

```
/home/arch/anaconda3/envs/GNN_env/lib/python3.8/site-packages/torch_geometri
c/deprecation.py:13: UserWarning: 'data.DataLoader' is deprecated, use 'loade
r.DataLoader' instead
  warnings.warn(out)
```

In [ ]:
```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  # Please do not change the args
  args = {
      'device': device,
      'num_layers': 5,
      'hidden_dim': 256,
      'dropout': 0.5,
      'lr': 0.001,
      'epochs': 30,
  }
  args
```

Graph Prediction Model

Graph Mini-Batching

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`.

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]
```

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.

Implemention

Now, we have all of the tools to implement a GCN Graph Prediction model!

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 the each graph. Remeber that the `batch` attribute will be essential for performining Global Pooling over our mini-batch of graphs.

In [54]:
```
# 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.ht

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

#         # 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, bat
#         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 i
#         ## For more information please refer to the documentation:
#         ## https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.ht
#         ## 3. Use a linear layer to predict each graph's property
#         ## (~3 lines of code)


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

#         return out
```

In [90]:

```
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 = global_mean_pool

        ############# Your code here ############
        ## Note:
        ## 1. Initialize the self.pool to global mean pooling layer
        ## More information please refer to the documentation:
        ## https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html
        ## (~1 line of code)

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

        # 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 this function that takes the input tensor batched_d
        # returns a batched output tensor for each graph.
        x, edge_index, batch = batched_data.x, batched_data.edge_index, batch
        embed = self.node_encoder(x)

        out = None

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

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

        return out
```

In [93]:

```
def train(model, device, data_loader, optimizer, loss_fn):
    # TODO: Implement this function that trains the 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 might change the type of label
        ## 5. Feed the output and label to loss_fn
        ## (~3 lines of code)
        optimizer.zero_grad()
        out = model(batch)
        loss = loss_fn(out[is_labeled], batch.y[is_labeled].float())

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

        loss.backward()
        optimizer.step()

    return loss.item()
```

In [95]:

```
# The evaluation function
def eval(model, device, loader, evaluator):
    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}

    return evaluator.eval(input_dict)
```

In [96]:

```
# if 'IS_GRADESCOPE_ENV' not in os.environ:
#   model = GCN_Graph(args['hidden_dim'],
#               dataset.num_tasks, args['num_layers'],
#               args['dropout']).to(device)
#   evaluator = Evaluator(name='ogbg-molhiv')

model = GCN_Graph(args['hidden_dim'],
            dataset.num_tasks, args['num_layers'],
            args['dropout']).to(device)
evaluator = Evaluator(name='ogbg-molhiv')
```

In [99]:

```
import copy

if 'IS_GRADESCOPE_ENV' not in os.environ:
  model.reset_parameters()

  optimizer = torch.optim.Adam(model.parameters(), lr=args['lr'])
  loss_fn = torch.nn.BCEWithLogitsLoss()
```

```
  best_model = None
  best_valid_acc = 0

  for epoch in range(1, 1 + args["epochs"]):
#  for epoch in range(1, 3):
    print('Training...')
    loss = train(model, device, train_loader, optimizer, loss_fn)

    print('Evaluating...')
    train_result = eval(model, device, train_loader, evaluator)
    val_result = eval(model, device, valid_loader, evaluator)
    test_result = eval(model, device, test_loader, evaluator)

    train_acc, valid_acc, test_acc = train_result[dataset.eval_metric], val_r
    if valid_acc > best_valid_acc:
        best_valid_acc = valid_acc
        best_model = copy.deepcopy(model)
    print(f'Epoch: {epoch:02d}, '
          f'Loss: {loss:.4f}, '
          f'Train: {100 * train_acc:.2f}%, '
          f'Valid: {100 * valid_acc:.2f}% '
          f'Test: {100 * test_acc:.2f}%')
```

```
Training...

Evaluating...


Epoch: 01, Loss: 0.0299, Train: 75.79%, Valid: 76.40% Test: 74.17%
Training...

Evaluating...


Epoch: 02, Loss: 0.0313, Train: 76.11%, Valid: 75.82% Test: 74.24%
Training...

Evaluating...


Epoch: 03, Loss: 0.0301, Train: 77.71%, Valid: 75.42% Test: 71.16%
Training...

Evaluating...


Epoch: 04, Loss: 0.0433, Train: 76.92%, Valid: 76.72% Test: 72.83%
Training...

Evaluating...


Epoch: 05, Loss: 0.0225, Train: 78.40%, Valid: 76.94% Test: 73.23%
Training...

Evaluating...


Epoch: 06, Loss: 0.0301, Train: 77.21%, Valid: 77.50% Test: 68.12%
Training...

Evaluating...


Epoch: 07, Loss: 0.0351, Train: 78.76%, Valid: 78.16% Test: 72.67%
```

```
Training...

Evaluating...


Epoch: 08, Loss: 0.0222, Train: 78.83%, Valid: 77.43% Test: 73.15%
Training...

Evaluating...


Epoch: 09, Loss: 0.3799, Train: 79.69%, Valid: 75.49% Test: 73.28%
Training...

Evaluating...


Epoch: 10, Loss: 0.0271, Train: 80.14%, Valid: 76.33% Test: 73.96%
Training...

Evaluating...


Epoch: 11, Loss: 0.0294, Train: 80.24%, Valid: 80.19% Test: 72.46%
Training...

Evaluating...


Epoch: 12, Loss: 0.3881, Train: 80.80%, Valid: 74.69% Test: 72.35%
Training...

Evaluating...


Epoch: 13, Loss: 0.0268, Train: 81.18%, Valid: 77.23% Test: 74.98%
Training...

Evaluating...


Epoch: 14, Loss: 0.5622, Train: 81.30%, Valid: 78.21% Test: 74.52%
Training...

Evaluating...


Epoch: 15, Loss: 0.1473, Train: 81.35%, Valid: 78.76% Test: 73.32%
Training...

Evaluating...


Epoch: 16, Loss: 0.0315, Train: 81.31%, Valid: 78.45% Test: 73.72%
Training...

Evaluating...


Epoch: 17, Loss: 0.0292, Train: 81.99%, Valid: 77.45% Test: 73.57%
Training...

Evaluating...


Epoch: 18, Loss: 0.0126, Train: 81.94%, Valid: 78.74% Test: 73.81%
Training...
```

```
Evaluating...


Epoch: 19, Loss: 0.0234, Train: 81.66%, Valid: 79.12% Test: 72.27%
Training...

Evaluating...


Epoch: 20, Loss: 0.0391, Train: 82.35%, Valid: 79.91% Test: 72.54%
Training...

Evaluating...


Epoch: 21, Loss: 0.0162, Train: 82.35%, Valid: 77.31% Test: 74.49%
Training...

Evaluating...


Epoch: 22, Loss: 0.5076, Train: 81.98%, Valid: 76.23% Test: 74.46%
Training...

Evaluating...


Epoch: 23, Loss: 0.0365, Train: 82.94%, Valid: 76.92% Test: 74.79%
Training...

Evaluating...


Epoch: 24, Loss: 0.0418, Train: 83.13%, Valid: 78.11% Test: 74.82%
Training...

Evaluating...


Epoch: 25, Loss: 0.0188, Train: 83.91%, Valid: 79.33% Test: 75.28%
Training...

Evaluating...


Epoch: 26, Loss: 0.0194, Train: 83.55%, Valid: 74.70% Test: 74.50%
Training...

Evaluating...


Epoch: 27, Loss: 0.0196, Train: 84.22%, Valid: 78.28% Test: 76.51%
Training...

Evaluating...


Epoch: 28, Loss: 0.7886, Train: 83.86%, Valid: 79.00% Test: 76.67%
Training...

Evaluating...


Epoch: 29, Loss: 0.0205, Train: 83.95%, Valid: 77.13% Test: 76.89%
Training...

Evaluating...
```

```
Epoch: 30, Loss: 0.0150, Train: 84.17%, Valid: 79.30% Test: 76.38%
```

Question 6: What are your `best_model` validation and test ROC-AUC scores? (20 points)

Run the cell below to see the results of your best of model and save your model's predictions over the validation and test datasets. The resulting files are named *ogbn-arxiv_graph_valid.csv* and *ogbn-arxiv_graph_test.csv*.

Again, you can view these files by clicking on the *Folder* icon on the left side pannel. As in Colab 1, when you sumbit your assignment, you will have to download these files and attatch them to your submission.

In [101…

```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  train_acc = eval(best_model, device, train_loader, evaluator)[dataset.eval_
#  valid_acc = eval(best_model, device, valid_loader, evaluator, save_model_r
  valid_acc = eval(best_model, device, valid_loader, evaluator)[dataset.eval_
  test_acc  = eval(best_model, device, test_loader, evaluator)[dataset.eval_m

  print(f'Best model: '
      f'Train: {100 * train_acc:.2f}%, '
      f'Valid: {100 * valid_acc:.2f}% '
      f'Test: {100 * test_acc:.2f}%')
```

```
Best model: Train: 80.24%, Valid: 80.19% Test: 72.46%
```

Question 7 (Optional): Experiment with the two other global pooling layers in Pytorch Geometric.

Submission

You will need to submit four files on Gradescope to complete this notebook.

1. Your completed *CS224W_Colab2.ipynb*. From the "File" menu select "Download .ipynb" to save a local copy of your completed Colab. **PLEASE DO NOT CHANGE THE NAME!** The autograder depends on the .ipynb file being called "CS224W_Colab2.ipynb".
2. *ogbn-arxiv_node.csv*
3. *ogbg-molhiv_graph_valid.csv*
4. *ogbg-molhiv_graph_test.csv*

Download the csv files by selecting the *Folder* icon on the left panel.

To submit your work, zip the files downloaded in steps 1-4 above and submit to gradescope. **NOTE:** DO NOT rename any of the downloaded files.