CS224W_Colab4

May 25, 2022

1 CS224W - Colab 4

In Colab 2 we constructed GNN models by using PyTorch Geometric's built in GCN layer, `GCNConv`. In Colab 3 we implemented the **GraphSAGE** (Hamilton et al. (2017)) layer. In this colab you'll use what you've learned and implement a more powerful layer: **GAT** (Veličković et al. (2018)). Then we will run our models on the CORA dataset, which is a standard citation network benchmark dataset.

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

Have fun and good luck on Colab 4 :)

2 Device

We recommend using a GPU for this Colab.

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

2.0.1 Acknowledgement

Much assistance from the following website: https://github.com/luciusssss/CS224W-Colab/blob/main/CS224W-Colab%203.ipynb

2.1 Installation

[1]:
```
# Install torch geometric
import os
if 'IS_GRADESCOPE_ENV' not in os.environ:
  !pip install torch-scatter -f https://data.pyg.org/whl/torch-1.10.0+cu113.html
  !pip install torch-sparse -f https://data.pyg.org/whl/torch-1.10.0+cu113.html
  !pip install torch-geometric
  !pip install -q git+https://github.com/snap-stanford/deepsnap.git
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no
version information available (required by /bin/bash)
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Looking in links: https://data.pyg.org/whl/torch-1.10.0+cu113.html
Requirement already satisfied: torch-scatter in
```

```
/home/arch/anaconda3/lib/python3.8/site-packages (2.0.7)
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no
version information available (required by /bin/bash)
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Looking in links: https://data.pyg.org/whl/torch-1.10.0+cu113.html
Requirement already satisfied: torch-sparse in
/home/arch/anaconda3/lib/python3.8/site-packages (0.6.9)
Requirement already satisfied: scipy in /home/arch/anaconda3/lib/python3.8/site-
packages (from torch-sparse) (1.6.2)
Requirement already satisfied: numpy<1.23.0,>=1.16.5 in
/home/arch/anaconda3/lib/python3.8/site-packages (from scipy->torch-sparse)
(1.20.1)
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no
version information available (required by /bin/bash)
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Requirement already satisfied: torch-geometric in
/home/arch/anaconda3/lib/python3.8/site-packages (2.0.3)
Requirement already satisfied: scikit-learn in
/home/arch/anaconda3/lib/python3.8/site-packages (from torch-geometric) (0.24.1)
Requirement already satisfied: numpy in /home/arch/anaconda3/lib/python3.8/site-
packages (from torch-geometric) (1.20.1)
Requirement already satisfied: tqdm in /home/arch/anaconda3/lib/python3.8/site-
packages (from torch-geometric) (4.59.0)
Requirement already satisfied: jinja2 in /home/arch/.local/lib/python3.8/site-
packages (from torch-geometric) (3.0.3)
Requirement already satisfied: networkx in
/home/arch/anaconda3/lib/python3.8/site-packages (from torch-geometric) (2.5)
Requirement already satisfied: pandas in /home/arch/.local/lib/python3.8/site-
packages (from torch-geometric) (1.3.5)
Requirement already satisfied: requests in
/home/arch/anaconda3/lib/python3.8/site-packages (from torch-geometric) (2.25.1)
Requirement already satisfied: scipy in /home/arch/anaconda3/lib/python3.8/site-
packages (from torch-geometric) (1.6.2)
Requirement already satisfied: PyYAML in /home/arch/.local/lib/python3.8/site-
packages (from torch-geometric) (6.0)
Requirement already satisfied: pyparsing in
/home/arch/anaconda3/lib/python3.8/site-packages (from torch-geometric) (2.4.7)
Requirement already satisfied: googledrivedownloader in
/home/arch/anaconda3/lib/python3.8/site-packages (from torch-geometric) (0.4)
Requirement already satisfied: yacs in /home/arch/anaconda3/lib/python3.8/site-
packages (from torch-geometric) (0.1.8)
Requirement already satisfied: rdflib in
/home/arch/anaconda3/lib/python3.8/site-packages (from torch-geometric) (6.1.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/home/arch/anaconda3/lib/python3.8/site-packages (from jinja2->torch-geometric)
(2.1.1)
Requirement already satisfied: decorator>=4.3.0 in
/home/arch/.local/lib/python3.8/site-packages (from networkx->torch-geometric)
```

```
(5.1.1)
Requirement already satisfied: pytz>=2017.3 in
/home/arch/anaconda3/lib/python3.8/site-packages (from pandas->torch-geometric)
(2021.1)
Requirement already satisfied: python-dateutil>=2.7.3 in
/home/arch/.local/lib/python3.8/site-packages (from pandas->torch-geometric)
(2.8.2)
Requirement already satisfied: six>=1.5 in
/home/arch/anaconda3/lib/python3.8/site-packages (from python-
dateutil>=2.7.3->pandas->torch-geometric) (1.15.0)
Requirement already satisfied: setuptools in
/home/arch/anaconda3/lib/python3.8/site-packages (from rdflib->torch-geometric)
(52.0.0.post20210125)
Requirement already satisfied: isodate in
/home/arch/anaconda3/lib/python3.8/site-packages (from rdflib->torch-geometric)
(0.6.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/home/arch/anaconda3/lib/python3.8/site-packages (from requests->torch-
geometric) (1.26.4)
Requirement already satisfied: idna<3,>=2.5 in
/home/arch/anaconda3/lib/python3.8/site-packages (from requests->torch-
geometric) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in
/home/arch/anaconda3/lib/python3.8/site-packages (from requests->torch-
geometric) (4.0.0)
Requirement already satisfied: certifi>=2017.4.17 in
/home/arch/anaconda3/lib/python3.8/site-packages (from requests->torch-
geometric) (2020.12.5)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/home/arch/anaconda3/lib/python3.8/site-packages (from scikit-learn->torch-
geometric) (2.1.0)
Requirement already satisfied: joblib>=0.11 in
/home/arch/anaconda3/lib/python3.8/site-packages (from scikit-learn->torch-
geometric) (1.1.0)
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no
version information available (required by /bin/bash)
```

[2]:
```
import torch_geometric
torch_geometric.__version__
```

```
/home/arch/anaconda3/envs/GNN_env/lib/python3.8/site-
packages/torch/cuda/__init__.py:80: UserWarning: CUDA initialization: CUDA
unknown error - this may be due to an incorrectly set up environment, e.g.
changing env variable CUDA_VISIBLE_DEVICES after program start. Setting the
available devices to be zero. (Triggered internally at  /opt/conda/conda-
bld/pytorch_1640811757556/work/c10/cuda/CUDAFunctions.cpp:112.)
  return torch._C._cuda_getDeviceCount() > 0
```

[2]: '2.0.3'

3 1) GNN Layers

3.1 Implementing Layer Modules

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 Colabs 3 and 4, we provide a build upon a general Graph Neural Network Stack, into which we will be able to plugin our own module implementations: GraphSAGE and GAT.

We will then use our layer implemenations 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.

3.2 GNN Stack Module

Below is the implementation of a general GNN stack, where we can plugin any GNN layer, such as **GraphSage**, **GAT**, etc. This module is provided for you. Your implementations of the **GraphSage** and **GAT** layers will function as components in the GNNStack Module.

[4]:
```python
import torch
import torch_scatter
import torch.nn as nn
import torch.nn.functional as F

import torch_geometric.nn as pyg_nn
import torch_geometric.utils as pyg_utils

from torch import Tensor
from typing import Union, Tuple, Optional
from torch_geometric.typing import (OptPairTensor, Adj, Size, NoneType,
                                    OptTensor)

from torch.nn import Parameter, Linear
from torch_sparse import SparseTensor, set_diag
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.utils import remove_self_loops, add_self_loops, softmax,␣
 ↪degree

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.convs to be
            # 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)
```

3.3 Creating Our Own Message Passing Layer

Now let's start implementing our own message passing layers! 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 will work with and implement 3 critcal functions needed to define a PyG Message Passing Layer: `forward`, `message`, and `aggregate`.

Before diving head first into the coding details, let us quickly review the key components of the message passing process. To do so, we will focus on a single round of messsage passing with respect to a single central node x. Before message passing, x is associated with a feature vector x^{l-1}, and the goal of message passing is to update this feature vector as x^l. To do so, we implement the following steps: 1) each neighboring node v passes its current message v^{l-1} across the edge (x, v) - 2) for the node x, we aggregate all of 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.

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

The `forward` fuction that we will 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.

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. To be more specific, after information is propagated (message passing), we can further transform the node embeddings outputed by `propagate`. Therefore, the output of `forward` is exactly the node embeddings after one GNN layer.

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

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 ($x_{neighbor}$).

  Taking the example of message passing above, for a central node u we will aggregate and transform all of the messages associated with the nodes v s.t. $(u, v) \in \mathcal{E}$ (i.e. $v \in \mathcal{N}_u$). Thus we see, the subscripts `_i` and `_j` allow us to specifcally differenciate features associated with central nodes (i.e. nodes recieving message information) and neighboring nodes (i.e. nodes passing messages).

  This is definitely a somewhat confusing concept; however, one key thing to remember / wrap your head around is that depending on the perspective, a node x acts as a central node or a neighboring node. In fact, in undirected graphs we store both edge directions (i.e. (i, j) and (j, i)). From the central node perspective, `x_i`, x is collecting neighboring information to update its embedding. From a neighboring node perspective, `x_j`, x is passing its message information along the edge connecting it to a different central node.

- `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.

The output of the `propagate` function is a matrix of node embeddings after the message passing process and has shape $[N, d]$.

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:

- `x_j` represents a matrix of feature embeddings for all neighboring nodes passing their messages along their respective edge (i.e. all nodes j for edges $(i, j) \in \mathcal{E}$). Thus, its shape is $[|\mathcal{E}|, d]$!
- In implementing GAT we will see how to access additional variables passed to propagate

Critically, we see that the output of the `message` function is a matrix of neighboring node embeddings ready to be aggregated, having shape $[|\mathcal{E}|, d]$.

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 $[N, d]$.

For additional resources refer to the PyG documentation for implementing custom message passing layers: https://pytorch-geometric.readthedocs.io/en/latest/notes/create_gnn.html

3.4 GAT Implementation

Attention mechanisms have become the 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) (Veličković et al. (2018)).

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: $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, \dots, \overrightarrow{h_N}\}$, $\overrightarrow{h_i} \in R^F$. The output of each graph attentional layer is a new set of node features, which may have a new dimension F': $\mathbf{h'} = \{\overrightarrow{h'_1}, \overrightarrow{h'_2}, \dots, \overrightarrow{h'_N}\}$, with $\overrightarrow{h'_i} \in \mathbb{R}^{F'}$.

We will now describe how this transformation is performed for each graph attention layer. First, a shared linear transformation parametrized by the weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is applied to every node.

Next, we perform self-attention on the nodes. We use a shared attention function a:

$$a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}. \tag{1}$$

that computes the attention coefficients capturing the importance of node j's features to node i:

$$e_{ij} = a(\mathbf{W_l}\overrightarrow{h_i}, \mathbf{W_r}\overrightarrow{h_j}) \tag{2}$$

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 e_{ij} for nodes $j \in \mathcal{N}_i$ where $\mathcal{N}_i$ is some neighborhood of node i in the graph.

To easily compare coefficients across different nodes, we normalize the coefficients across j using a softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{3}$$

For this problem, our attention mechanism a will be a single-layer feedforward neural network parametrized by a weight vectors $\overrightarrow{a_l} \in \mathbb{R}^{F'}$ and $\overrightarrow{a_r} \in \mathbb{R}^{F'}$, 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:

$$\alpha_{ij} = \frac{\exp\Big(\text{LeakyReLU}\Big(\overrightarrow{a_l}^T \mathbf{W_l}\overrightarrow{h_i} + \overrightarrow{a_r}^T \mathbf{W_r}\overrightarrow{h_j}\Big)\Big)}{\sum_{k \in \mathcal{N}_i} \exp\Big(\text{LeakyReLU}\Big(\overrightarrow{a_l}^T \mathbf{W_l}\overrightarrow{h_i} + \overrightarrow{a_r}^T \mathbf{W_r}\overrightarrow{h_k}\Big)\Big)} \tag{4}$$

For the following questions, we denote `alpha_l` $= \alpha_l = [..., \overrightarrow{a_l}^T \mathbf{W_l} \overrightarrow{h_i}, ...] \in \mathcal{R}^n$ and `alpha_r` $= \alpha_r = [..., \overrightarrow{a_r}^T \mathbf{W_r} \overrightarrow{h_j}, ...] \in \mathcal{R}^n$.

At every layer of GAT, after the attention coefficients are computed for that layer, the aggregation function can be computed by a weighted sum of neighborhood messages, where weights are specified by α_{ij}.

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.

$$h_i' = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W_r} \overrightarrow{h_j}. \tag{5}$$

At this point, we have covered a lot of information! Before reading further about multi-head attention, we encourage you to go again through the excersize of thinking about what components of the attention mechanism correspond with the different functions: 1) `forward`, 2) `message`, and 3 `aggregate`.

- Hint 1: Our aggregation is very similar to that of GraphSage except now we are using sum aggregation
- Hint 2: The terms we aggregate over again represent the individual message that each neighbor node j sends. Thus, we see that α_{ij} is part of the message each node sends and is thus computed during the message step. This makes sense since an attention weight is associated with each edge in the graph.
- Hint 3: Look at the terms in the definition of α_{ij}. What values do we want to pre-process and pass as parameters to the `propagate` function. The parameters of `message(..., x_j, alpha_j, alpha_i, ...)` should give a good hint.

3.4.1 Multi-Head Attention

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:

$$\overrightarrow{h_i}' = ||_{k=1}^{K} \Big(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} \mathbf{W_r}^{(k)} \overrightarrow{h_j} \Big) \tag{6}$$

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, $\mathbf{h'} \in \mathbb{R}^{KF'}$.

[9]:
```python
class GAT(MessagePassing):

    def __init__(self, in_channels, out_channels, heads = 2,
                 negative_slope = 0.2, dropout = 0., **kwargs):
        super(GAT, self).__init__(node_dim=0, **kwargs)

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.heads = heads
        self.negative_slope = negative_slope
        self.dropout = dropout

        self.lin_l = None
        self.lin_r = None
        self.att_l = None
        self.att_r = None

        ␣
↪############################################################################
        # TODO: Your code here!
        # Define the layers needed for the message functions below.
        # self.lin_l is the linear transformation that you apply to embeddings
        # BEFORE message passing.
        #
        # Pay attention to dimensions of the linear layers, since we're using
        # multi-head attention.
        # Our implementation is ~1 lines, but don't worry if you deviate from␣
↪this.

        self.lin_l = nn.Linear(self.in_channels, self.out_channels * self.heads)
        ␣
↪############################################################################

        self.lin_r = self.lin_l

        ␣
↪############################################################################
        # TODO: Your code here!
        # Define the attention parameters \overrightarrow{a_l/r}^T in the above␣
↪intro.
        # You have to deal with multi-head scenarios.
        # Use nn.Parameter instead of nn.Linear
        # Our implementation is ~2 lines, but don't worry if you deviate from␣
↪this.

        self.att_l = nn.Parameter(torch.zeros(self.heads, self.out_channels))
        self.att_r = nn.Parameter(torch.zeros(self.heads, self.out_channels))

        ␣
↪############################################################################

        self.reset_parameters()
```

```
    def reset_parameters(self):
        nn.init.xavier_uniform_(self.lin_l.weight)
        nn.init.xavier_uniform_(self.lin_r.weight)
        nn.init.xavier_uniform_(self.att_l)
        nn.init.xavier_uniform_(self.att_r)

    def forward(self, x, edge_index, size = None):

        H, C = self.heads, self.out_channels

        ␣
↪############################################################################
        # TODO: Your code here!
        # Implement message passing, as well as any pre- and post-processing␣
↪(our update rule).
        # 1. First apply linear transformation to node embeddings, and split␣
↪that
        #    into multiple heads. We use the same representations for source and
        #    target nodes, but apply different linear weights (W_l and W_r)
        # 2. Calculate alpha vectors for central nodes (alpha_l) and neighbor␣
↪nodes (alpha_r).
        # 3. Call propagate function to conduct the message passing.
        #    3.1 Remember to pass alpha = (alpha_l, alpha_r) as a parameter.
        #    3.2 See there for more information: https://pytorch-geometric.
↪readthedocs.io/en/latest/notes/create_gnn.html
        # 4. Transform the output back to the shape of [N, H * C].
        # Our implementation is ~5 lines, but don't worry if you deviate from␣
↪this.


        x_l = self.lin_l(x).reshape(-1,H,C)
        x_r = self.lin_l(x).reshape(-1,H,C)
        alpha_l = self.att_l*x_l
        alpha_r = self.att_r*x_r
        out=self.propagate(edge_index, x=(x_l, x_r), alpha=(alpha_l, alpha_r),␣
↪size=size)
        out=out.reshape(-1, H*C)



        ␣
↪############################################################################

        return out


    def message(self, x_j, alpha_j, alpha_i, index, ptr, size_i):
     ␣
↪############################################################################
        # TODO: Your code here!
        # Implement your message function. Putting the attention in message
        # instead of in update is a little tricky.
        # 1. Calculate the final attention weights using alpha_i and alpha_j,
        #    and apply leaky Relu.
        # 2. Calculate softmax over the neighbor nodes for all the nodes. Use
        #    torch_geometric.utils.softmax instead of the one in Pytorch.
        # 3. Apply dropout to attention weights (alpha).
        # 4. Multiply embeddings and attention weights. As a sanity check, the␣
↪output
        #    should be of shape [E, H, C].
        # 5. ptr (LongTensor, optional): If given, computes the softmax based on
        #    sorted inputs in CSR representation. You can simply pass it to␣
↪softmax.
        # Our implementation is ~4-5 lines, but don't worry if you deviate from␣
↪this.
        alpha = F.leaky_relu(alpha_i + alpha_j, negative_slope=self.
↪negative_slope)
        if ptr:
            att_weight = F.softmax(alpha_i + alpha_j, ptr)
        else:
            att_weight = torch_geometric.utils.softmax(alpha_i + alpha_j, index)

        att_weight = F.dropout(att_weight, p=self.dropout)
        out = att_weight * x_j

     ␣
↪############################################################################

        return out


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

     ␣
↪############################################################################
        # TODO: Your code here!
        # Implement your aggregate function here.
        # See here as how to use torch_scatter.scatter: https://pytorch-scatter.
↪readthedocs.io/en/latest/_modules/torch_scatter/scatter.html
        # Pay attention to "reduce" parameter is different from that in␣
↪GraphSage.
```

```
        # Our implementation is ~1 lines, but don't worry if you deviate from␣
↪this.
        out = torch_scatter.scatter(inputs, index, self.node_dim, dim_size =␣
↪dim_size, reduce='sum')
       ␣
↪############################################################################

        return out
```

3.5 Building Optimizers

This function has been implemented for you. **For grading purposes please use the default Adam optimizer**, but feel free to play with other types of optimizers on your own.

[10]:
```
import torch.optim as optim

def build_optimizer(args, params):
    weight_decay = args.weight_decay
    filter_fn = filter(lambda p : p.requires_grad, params)
    if args.opt == 'adam':
        optimizer = optim.Adam(filter_fn, lr=args.lr, weight_decay=weight_decay)
    elif args.opt == 'sgd':
        optimizer = optim.SGD(filter_fn, lr=args.lr, momentum=0.95,␣
↪weight_decay=weight_decay)
    elif args.opt == 'rmsprop':
        optimizer = optim.RMSprop(filter_fn, lr=args.lr,␣
↪weight_decay=weight_decay)
    elif args.opt == 'adagrad':
        optimizer = optim.Adagrad(filter_fn, lr=args.lr,␣
↪weight_decay=weight_decay)
    if args.opt_scheduler == 'none':
        return None, optimizer
    elif args.opt_scheduler == 'step':
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=args.
↪opt_decay_step, gamma=args.opt_decay_rate)
    elif args.opt_scheduler == 'cos':
        scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=args.
↪opt_restart)
    return scheduler, optimizer
```

3.6 Training and Testing

Here we provide you with the functions to train and test. **Please do not modify this part for grading purposes.**

[11]:
```
import time
import networkx as nx
import numpy as np
import torch
import torch.optim as optim
from tqdm import trange
import pandas as pd
import copy

from torch_geometric.datasets import TUDataset
from torch_geometric.datasets import Planetoid
from torch_geometric.data import DataLoader

import torch_geometric.nn as pyg_nn

import matplotlib.pyplot as plt


def train(dataset, args):

    print("Node task. test set size:", np.sum(dataset[0]['test_mask'].numpy()))
    print()
    test_loader = loader = DataLoader(dataset, batch_size=args.batch_size,␣
 ↪shuffle=False)

    # build model
    model = GNNStack(dataset.num_node_features, args.hidden_dim, dataset.
 ↪num_classes,
                            args)
    scheduler, opt = build_optimizer(args, model.parameters())

    # train
    losses = []
    test_accs = []
    best_acc = 0
    best_model = None
    for epoch in trange(args.epochs, desc="Training", unit="Epochs"):
        total_loss = 0
        model.train()
        for batch in loader:
            opt.zero_grad()
            pred = model(batch)
            label = batch.y
            pred = pred[batch.train_mask]
            label = label[batch.train_mask]
            loss = model.loss(pred, label)
            loss.backward()
            opt.step()
            total_loss += loss.item() * batch.num_graphs
        total_loss /= len(loader.dataset)
        losses.append(total_loss)

        if epoch % 10 == 0:
          test_acc = test(test_loader, model)
          test_accs.append(test_acc)
          if test_acc > best_acc:
            best_acc = test_acc
            best_model = copy.deepcopy(model)
        else:
          test_accs.append(test_accs[-1])

    return test_accs, losses, best_model, best_acc, test_loader

def test(loader, test_model, is_validation=False, save_model_preds=False,␣
 ↪model_type=None):
    test_model.eval()

    correct = 0
    # Note that Cora is only one graph!
    for data in loader:
        with torch.no_grad():
            # max(dim=1) returns values, indices tuple; only need indices
            pred = test_model(data).max(dim=1)[1]
            label = data.y

        mask = data.val_mask if is_validation else data.test_mask
        # node classification: only evaluate on nodes in test set
        pred = pred[mask]
        label = label[mask]

        if save_model_preds:
          print ("Saving Model Predictions for Model Type", model_type)

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

          df = pd.DataFrame(data=data)
          # Save locally as csv
          df.to_csv('CORA-Node-' + model_type + '.csv', sep=',', index=False)

        correct += pred.eq(label).sum().item()

    total = 0
    for data in loader.dataset:
        total += torch.sum(data.val_mask if is_validation else data.test_mask).
↪item()

    return correct / total

class objectview(object):
    def __init__(self, d):
        self.__dict__ = d
```

3.7 Let's Start the Training!

We will be working on the CORA dataset on node-level classification.

This part is implemented for you. **For grading purposes, please do not modify the default parameters.** However, feel free to play with different configurations just for fun!

Submit your best accuracy and loss on Gradescope.

[12]:
```
if 'IS_GRADESCOPE_ENV' not in os.environ:
    for args in [
        {'model_type': 'GAT', 'dataset': 'cora', 'num_layers': 2, 'heads': 1,␣
↪'batch_size': 32, 'hidden_dim': 32, 'dropout': 0.5, 'epochs': 500, 'opt':␣
↪'adam', 'opt_scheduler': 'none', 'opt_restart': 0, 'weight_decay': 5e-3,␣
↪'lr': 0.01},
    ]:
        args = objectview(args)
        for model in ['GAT']:
            args.model_type = model

            # Match the dimension.
            if model == 'GAT':
              args.heads = 2
            else:
              args.heads = 1

            if args.dataset == 'cora':
                dataset = Planetoid(root='/tmp/cora', name='Cora')
            else:
                raise NotImplementedError("Unknown dataset")
            test_accs, losses, best_model, best_acc, test_loader =␣
↪train(dataset, args)

            print("Maximum test set accuracy: {0}".format(max(test_accs)))
            print("Minimum loss: {0}".format(min(losses)))

            # Run test for our best model to save the predictions!
            test(test_loader, best_model, is_validation=False,␣
↪save_model_preds=True, model_type=model)
```

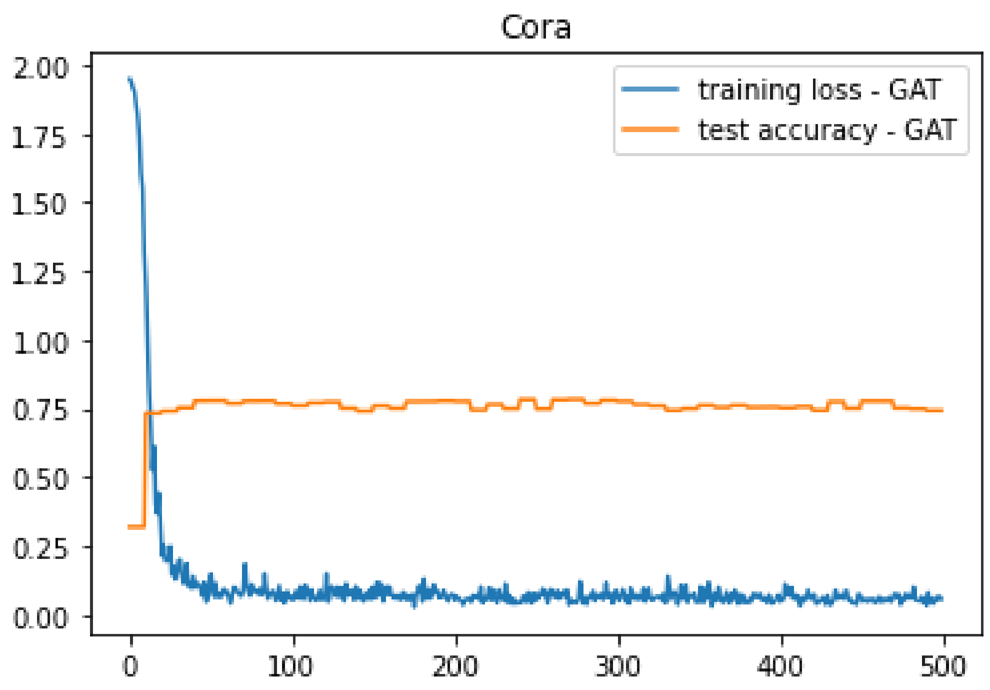
```
            print()

            plt.title(dataset.name)
            plt.plot(losses, label="training loss" + " - " + args.model_type)
            plt.plot(test_accs, label="test accuracy" + " - " + args.model_type)
        plt.legend()
        plt.show()
```

```
Node task. test set size: 1000


Training: 100%|               | 500/500 [00:50<00:00,  9.99Epochs/s]

Maximum test set accuracy: 0.784
Minimum loss: 0.027421951293945312
Saving Model Predictions for Model Type GAT
```



3.8 Question 1: What is the maximum accuracy obtained on test set for GAT? (10 points)

Running the training cell above will also save your best GAT model predictions as *CORA-Node-GAT.csv.*

When you sumbit your assignment, you will have to download this file and attatch it to your submission. As with the other colabs, please zip this file (DON'T CHANGE ITS NAME) and the

.csv file that's generated!

The maximum test set accuracy for GAT is 78.4%. This compares with 79.8% for GraphSAGE.