```
#%% md

#%% md
Please choose a graph-based GNN model of your choice to classify (quark/gluon) jets. Proceed as follows:

* Convert the images into a point cloud dataset by only considering the non-zero pixels for every event.

* Cast the point cloud data into a graph representation by coming up with suitable representations for nodes and edges.

* Train your model on the obtained graph representations of the jet events.

Discuss the resulting performance of the chosen architecture.

#%% md
## Loading Quark-Gluon Data :

Loading the hdf5 file using `h5py`'s `File` object with the Standard Driver appropriate for the current platform.

#%%
from google.colab import drive
drive.mount('/content/drive')
#%%

#%%
import numpy as np
import h5py

def load_h5(file_name, size):
    # Load the dataset from the HDF5 file
    with h5py.File(file_name, 'r') as f:
        X = np.array(f['X_jets'][:size])
        y = np.array(f['y'][:size])
    return X, y
```

```
34
35 #%%
36
37 #%%
38 file = '/content/drive/MyDrive/quark-gluon_data-
   set_n139306.hdf5'
39 size = 16000
40 X, y = load_h5(file, size)
41 #%%
42 # Count occurrences of each value (0s and 1s)
43 y_int = y.astype(np.int64)
44 counts = np.bincount(y_int)
45
46 # Create a dictionary to store counts
47 counts_dict = {'0': counts[0], '1': counts[1]}
48
49 print(counts_dict)
50 del y_int
51 del counts
52 del counts_dict
53 #%%
54 !pip install torch-geometric
55 #%% md
56
57 Importing necessary libaries and modules as required
   for the immediate tasks ->
58 #%%
59 !pip install pytorch-lightning
60 #%%
61 import matplotlib.pyplot as plt
62 from sklearn.model_selection import train_test_split
63 from torch_geometric.data import Data, Batch
64 from torch_geometric.loader import DataLoader
65 from torch.optim import Adam
66 import torch.optim as optim
67 import pytorch_lightning as pl
68 import torch
69 import torch.nn as nn
70 import torch.nn.functional as F
71 from torch_geometric.nn import SAGEConv
72 from torch_geometric.nn import global_mean_pool
```

```
 73 from torch.nn import Linear
 74 #%%
 75 from skimage.transform import resize
 76 from sklearn.preprocessing import normalize
 77 
 78 def data_preprocess(X_jets):
 79     #Normalizing the images
 80 
 81     # Resizing images from (125, 125, 3) to (128,
    128, 3)
 82     resized_images = np.zeros((X_jets.shape[0], 128
    , 128, 3), dtype=np.float32)
 83     for i in range(X_jets.shape[0]):
 84         resized_images[i] = resize(X_jets[i], (128,
    128), anti_aliasing=True)
 85 
 86     X_jets = resized_images
 87     del resized_images
 88 
 89     # Normalizing the entire image across all
    channels
 90     mean = np.mean(X_jets)
 91     std = np.std(X_jets)
 92     X_jets = (X_jets - mean) / std
 93 
 94     # Assuming X_jets is your image array
 95     X_jets = np.clip(X_jets, 0, None)
 96     return X_jets
 97 #%%
 98 X = data_preprocess(X)
 99 #%%
100 #combined.shape
101 #%%
102 import numpy as np
103 import matplotlib.pyplot as plt
104 from mpl_toolkits.mplot3d import Axes3D
105 
106 def plot_point_clouds(X, y):
107     # Store the point clouds for all images in a
    list
108     point_clouds = []
```

```
109
110     for i in range(5): # Limit to the first 5 images
111         # Extract the non-zero pixel coordinates and
     values for each channel
112         non_zero_Tracks = np.nonzero(X[i, :, :, 0])
113         non_zero_ECAL = np.nonzero(X[i, :, :, 1])
114         non_zero_HCAL = np.nonzero(X[i, :, :, 2])
115
116         # Create a 2D array, where each row
    represents the coordinate of the non-zero entity.
117         coords_Tracks = np.column_stack(
    non_zero_Tracks)
118         coords_ECAL = np.column_stack(non_zero_ECAL)
119         coords_HCAL = np.column_stack(non_zero_HCAL)
120
121         # For visualization placing Tracks, ECAL,
    and HCAL on z = 0,1,2 respectively.
122         values_Tracks = X[i, non_zero_Tracks[0],
    non_zero_Tracks[1], 0]
123         values_ECAL = X[i, non_zero_ECAL[0],
    non_zero_ECAL[1], 1]
124         values_HCAL = X[i, non_zero_HCAL[0],
    non_zero_HCAL[1], 2]
125
126         coords_Tracks = np.hstack((coords_Tracks, np
    .zeros((coords_Tracks.shape[0], 1))))
127         coords_ECAL = np.hstack((coords_ECAL, np.
    zeros((coords_ECAL.shape[0], 1))))
128         coords_HCAL = np.hstack((coords_HCAL, np.
    zeros((coords_HCAL.shape[0], 1))))
129
130         # Store the point cloud for this image in
    the list
131         point_clouds.append({'tracks': (
    coords_Tracks, values_Tracks), 'ECAL': (coords_ECAL
    , values_ECAL), 'HCAL': (coords_HCAL, values_HCAL)})
132
133         # Plot the point cloud for this image
134         fig = plt.figure(figsize=(10, 10))
135         ax = fig.add_subplot(111, projection='3d')
136
```

```
137         ax.scatter(coords_Tracks[:, 0],
    coords_Tracks[:, 1], coords_Tracks[:, 2], c=
    values_Tracks, cmap='viridis', alpha=0.5)
138         ax.scatter(coords_ECAL[:, 0], coords_ECAL
    [:, 1], coords_ECAL[:, 2] + 1, c=values_ECAL, cmap='
    viridis', alpha=0.5)
139         ax.scatter(coords_HCAL[:, 0], coords_HCAL
    [:, 1], coords_HCAL[:, 2] + 2, c=values_HCAL, cmap='
    viridis', alpha=0.5)
140
141         ax.set_xlabel('X')
142         ax.set_ylabel('Y')
143         ax.set_zlabel('Channel')
144         ax.set_title('Point Cloud Visualization for
    Image {}'.format(i))
145
146         plt.show()
147
148 #
149 # del non_zero_Tracks
150 # del non_zero_ECAL
151 # del non_zero_HCAL
152 # del coords_Tracks
153 # del coords_ECAL
154 # del coords_HCAL
155 # del values_Tracks
156 # del values_ECAL
157 # del values_HCAL
158 # #del point_clouds
159 # del point_cloud
160 #%%
161 plot_point_clouds(X, y)
162 #%% md
163 Now, the question comes -> how to suitably cast this
     point cloud data as a suitable graph data structure
    .
164
165 * We need a `node` construct, and an `edge`
    construct to atleast start off.
166 * Either make each pixel a `node`, or construct some
     cluster to acquire a node.
```

```
167 * Initially, we make each pixel a node, and invoke the `kneighbors_graph` function to construct the edges around these nodes, joining `n=2` nearest neighbors with each node.
168 #%% md
169 `dataset` entity contains a graph for each of the loaded images, which we further split and load in train, test and validation loaders.
170 #%% md
171 Instead of working on individual points like PointNet, exploit local geometric structures by constructing a local neighborhood graph and applying convolution-like operations on the edges connecting neighboring pairs of points, in the spirit of GNNs.
172
173 This gives us the added advantage of exploiting the nonlocal information that diffuses through the point cloud's updated feature spaces in subsequent layers.
174 #%%
175 def return_nonzero(X):
176     # Reshape the data to be compatible with torch_geometric
177     orig_len = X.shape[1]
178     orig_wid = X.shape[2]
179     data = X.reshape((-1, orig_len* orig_wid, 3))
180
181     #Making a bool array to find which indices have non-zero values
182     non_black_pixels_mask = np.any(data != [0., 0., 0.], axis=-1)
183
184     mask_reshaped = non_black_pixels_mask.reshape((-1, orig_len, orig_wid))
185     return mask_reshaped
186 #%%
187 mask_reshaped = return_nonzero(X)
188 #%%
189 def create_graph_feats(mask_reshaped):
190     indices_list = []
191     features_list = []
```

```
192     for i, x in enumerate(mask_reshaped):
193         true_indices = np.where(x)
194         feature_vec = X[i, true_indices[0],
    true_indices[1], :]
195         indices_array = np.column_stack((
    true_indices[0], true_indices[1]))
196         indices_list.append(indices_array)
197         features_list.append(feature_vec)
198     return indices_list, features_list
199 
200 #%%
201 indices_list, features_list = create_graph_feats(
    mask_reshaped)
202 #%%
203 print(features_list[0].shape)
204 #%%
205 import argparse
206 import numpy as np
207 import scipy.sparse as sp
208 
209 
210 def import_class(name):
211     try:
212         components = name.split('.')
213         module = __import__(components[0])
214         for c in components[1:]:
215             module = getattr(module, c)
216     except AttributeError:
217         module = None
218     return module
219 
220 def str2bool(v):
221     if v.lower() in ('yes', 'true', 't', 'y', '1'):
222         return True
223     elif v.lower() in ('no', 'false', 'f', 'n', '0'
    ):
224         return False
225     else:
226         raise argparse.ArgumentTypeError('Boolean
    value expected.')
227 
```

```
228
229 def get_total_parameters(model):
230     total = sum(p.numel() for p in model.parameters
    ())
231     trainable = sum(p.numel() for p in model.
    parameters() if p.requires_grad)
232     return {'Total': total, 'Trainable': trainable}
233
234
235 def _get_weights(dist, indices):
236     num_nodes, k = dist.shape
237     assert num_nodes, k == indices.shape
238     assert dist.min() >= 0
239     # weight matrix
240     sigma2 = np.mean(dist[:, -1])**2
241     dist = np.exp(-dist**2 / sigma2)
242     i = np.arange(0, num_nodes).repeat(k)
243     j = indices.reshape(num_nodes * k)
244     v = dist.reshape(num_nodes * k)
245     weights = sp.coo_matrix((v, (i, j)), shape=(
    num_nodes, num_nodes))
246     # no self-loop
247     weights.setdiag(0)
248     # undirected graph
249     bigger = weights.T > weights
250     weights = weights - weights.multiply(bigger) +
    weights.T.multiply(bigger)
251     return weights
252
253
254 def _get_normalize_adj(dist, indices):
255     adj = _get_weights(dist, indices)
256     adj = sp.coo_matrix(adj)
257     row_sum = np.array(adj.sum(1))
258     row_sum_nonzero = np.where(row_sum != 0, row_sum
    , 1)  # Avoid division by zero
259     d_inv = np.power(row_sum_nonzero, -0.5).flatten
    ()
260     d_inv[np.isinf(d_inv)] = 0.0
261     d_mat_inv_sqrt = sp.diags(d_inv)
262     return adj.dot(d_mat_inv_sqrt).transpose().dot(
```

```
d_mat_inv_sqrt).tocoo()


def build_graph(coordinates, k=4):
    """
    :param coordinates: positions for 3D point cloud (N * 3)
    :param k: number of nearest neighbors
    :return: adjacency matrix for 3D point cloud
    """
    from scipy.spatial import cKDTree
    tree = cKDTree(coordinates)
    dist, indices = tree.query(coordinates, k=k)
    return _get_normalize_adj(dist, indices)

#%%
def create_graph_dataset(indices_list, y, connectivity = 10):
    graph_dataset = []
    for i, points in enumerate(indices_list):
        c = build_graph(points, k=connectivity)
        #edge_index = build_graph(points, k = connectivity)
        edge_list = torch.from_numpy(np.vstack((c.row, c.col))).type(torch.long)
#         edge_index = torch.from_numpy(edge_index).type(torch.long)
        edge_weight = torch.from_numpy(c.data.reshape(-1, 1))
        labels = torch.tensor([int(y[i])], dtype=torch.long)

        # Convert COO matrix to tensor
    #    adj_tensor = torch.sparse_coo_tensor((c.row, c.col), c.data, c.shape).to_dense()
        data = Data(x=torch.from_numpy(features_list[i]), edge_index=edge_list, y=labels, edge_attr = edge_weight)
    # edge_attr=edge_weight
        graph_dataset.append(data)

```

```
293     return graph_dataset
294 
295 #%%
296 graph_dataset = create_graph_dataset(indices_list, y
    , connectivity = 8)
297 #%% md
298 Having a scoop at the obtained list of `
    torch_geometric.data.data.Data` objects, with each
    Data object being populated with the features tensor
     as `data.x`, Adjacency list as `data.edge_index`,
    Edge Attributes, containing the distance metric as `
    data.edge_attr`, and the sparsely constructed
    adjacency matrix as `data.adj`.
299 #%%
300 import networkx as nx
301 import matplotlib.pyplot as plt
302 
303 G = nx.Graph()
304 data = graph_dataset[0]
305 edge_tensor = data.edge_index
306 # Convert the edge tensor to a list of tuples
307 edge_list = [(edge_tensor[0, i].item(), edge_tensor[
    1, i].item()) for i in range(edge_tensor.shape[1])]
308 
309 
310 #print(data.edge_index.shape)
311 G.add_edges_from(edge_list)
312 pos = nx.spring_layout(G, iterations=15, seed=1721)
313 fig, ax = plt.subplots(figsize=(15, 9))
314 ax.axis("off")
315 plot_opt = {"node_size" : 10, "with_labels" : False
    , "width" : 0.05}
316 nx.draw_networkx(G, pos=pos, ax=ax, **plot_opt)
317 plt.show()
318 
319 print(f'Number of graphs to work upon : {len(
    graph_dataset)}')
320 
321 print(f'For the FIRST graph in the  graph dataset
     : ')
322 print(f'Type of each graph entity data object: {type
```

```
(data)}')
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Number of node features: {data.
num_node_features}')
print(f'Number of edges features: {data.
num_edge_features}')
#print(f'Number of edges features: {data.num_classes
}')

#%% md
## Defining a custom Graph Classification Scheme :


### Earlier Attempt :

GraphSAGE is one of the most popular GNN
architectures, utilizing neighborhood aggregations
via the use of a more expressive convolutional
operator.

However, the resultant model could not exceed the
baseline (51% accuracy), possibly due to the OVER-
QUASHING PROBLEM and the shallowness of the jet
image embeddings.

### Random Subset Conv :

I had earlier constructed a graph structure with
high connectivity (8), hence I thought of using
random neighborhood aggregation for message passing
in the GNN, which is what I have implemented in the
following :
#%%
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops
class RandomSubsetConv(MessagePassing):
    def __init__(self, in_channels, out_channels,
subset_size):
        super(RandomSubsetConv, self).__init__(aggr=
```

```
348 'add') # "add" aggregation.
349         self.lin = torch.nn.Linear(in_channels,
    out_channels)
350         self.subset_size = subset_size
351 
352     def forward(self, x, edge_index):
353         # x has shape [N, in_channels]
354         # edge_index has shape [2, E]
355 
356         # Step 1: Add self-loops to the adjacency
    matrix
357         edge_index, _ = add_self_loops(edge_index,
    num_nodes=x.size(0))
358 
359         # Step 2: Sample a subset of neighbors for
    each node
360         row, col = edge_index
361         subset_indices = torch.randperm(row.size(0
    ))[:self.subset_size]
362         subset_edge_index = edge_index[:,
    subset_indices]
363 
364         # Step 3: Perform message passing
365         return self.propagate(subset_edge_index,
    size=(x.size(0), x.size(0)), x=x)
366 
367     def message(self, x_j, edge_index, size):
368         # x_j has shape [E, in_channels]
369         # Step 4: Aggregate node features
370         return self.lin(x_j)
371 
372     def update(self, aggr_out):
373         # aggr_out has shape [N, out_channels]
374         # Step 5: Update node features
375         return aggr_out
376 
377 #%%
378 import torch
379 import torch.utils.data as data
380 from torchvision import transforms
381 from torch_geometric.nn import GCNConv, GATConv,
```

```
GATv2Conv, TransformerConv, global_mean_pool
from torch_geometric.nn.norm import BatchNorm
import torch.nn as nn


class GraphSAGE(torch.nn.Module):
    def __init__(self, c_in, c_hidden, c_out, dp_rate_linear=0.3, dp_rate_dropout=0.5, subset_size=10):
        super().__init__()
        torch.manual_seed(17)

        self.conv1 = RandomSubsetConv(c_in, c_hidden, subset_size = 4)
        self.norm1 = nn.BatchNorm1d(c_hidden)
#         self.conv2 = RandomSubsetConv(c_hidden, 2 * c_hidden, subset_size = 4)
#         self.norm2 = nn.BatchNorm1d(2 * c_hidden)
        self.conv2 = RandomSubsetConv(c_hidden, c_hidden, subset_size = 4)
        self.norm2 = nn.BatchNorm1d(c_hidden)

        self.lin1 = nn.Linear(c_hidden, c_hidden // 2)
        self.lin2 = nn.Linear(c_hidden //2, c_out)

        self.dp_rate_linear = dp_rate_linear
        self.dp_rate_dropout = dp_rate_dropout
        self.dropout = nn.Dropout(dp_rate_dropout)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x = self.norm1(x)
        x = x.relu()
        x = self.dropout(x)

        x = self.conv2(x, edge_index)
        x = self.norm2(x)
        x = x.relu()
        x = self.dropout(x)

```

```
416  #         x = self.conv3(x, edge_index)
417  #         x = self.norm3(x)
418  #         x = x.relu()
419  #         x = self.dropout(x)
420
421          x = global_mean_pool(x, batch)
422          x = self.dropout(x)
423          x = self.lin1(x)
424          x = x.relu()
425          x = self.lin2(x)
426
427          return x
428
429
430 #%%
431 class GraphLevelGNN(pl.LightningModule):
432     def __init__(self, **model_kwargs):
433         super().__init__()
434         # Saving hyperparameters
435         self.save_hyperparameters()
436  #         self.transform = transforms.Compose([
    transforms.ToTensor()])
437         self.model = GraphSAGE(**model_kwargs)
438         self.loss_module = nn.BCEWithLogitsLoss() if
     self.hparams.c_out == 1 else nn.CrossEntropyLoss()
439
440 #     def __getitem__(self, data, index):
441 #         return self.transform(self.data.x[index]),
    self.transform(self.data.edge_index[index]), self.
    tranform(self.data.batch[index])
442
443     def forward(self, data, mode="train"):
444 #         x, edge_index, batch_idx = data.x, data.
    edge_index, data.batch
445         # print(data.x.shape, data.edge_index.shape
    , data.batch.shape)
446         x, edge_index, batch_idx = data.x, data.
    edge_index, data.batch
447 #         print(x.shape)
448 #         print(edge_index.shape)
449 #         print(batch_idx.shape)
```

```
450         x = self.model(x, edge_index, batch_idx)
451         x = x.squeeze(dim=-1)
452 
453         if self.hparams.c_out == 1:
454             preds = (x > 0.7).float()
455             data.y = data.y.float()
456         else:
457             preds = x.argmax(dim=-1)
458 
459         loss = self.loss_module(x, data.y)
460         acc = (preds == data.y).sum().float() /
    preds.shape[0]
461 
462         return loss, acc
463 
464     def configure_optimizers(self):
465         optimizer = optim.Adam(self.parameters(), lr
    =0.01, weight_decay=0.1)
466         return optimizer
467 
468     def training_step(self, batch, batch_idx):
469         loss, acc = self.forward(batch, mode="train"
    )
470         self.log('train_loss', loss, prog_bar=True)
471         self.log('train_acc', acc, prog_bar=True)
472         return loss
473 
474     def validation_step(self, batch, batch_idx):
475         loss, acc = self.forward(batch, mode="val")
476         self.log('val_loss', loss, on_epoch=True,
    prog_bar=False)
477         self.log('val_acc', acc, on_epoch=True,
    prog_bar=False)
478 
479     def test_step(self, batch, batch_idx):
480         loss, acc = self.forward(batch, mode="test")
481         self.log('test_loss', loss, on_epoch=True,
    prog_bar=False)
482         self.log('test_acc', acc, on_epoch=True,
    prog_bar=False)
483 #%%
```

```
484 import torch
485 
486 
487 def train_graph_classifier(model_name, **
    model_kwargs):
488     pl.seed_everything(17)
489 
490     trainer = pl.Trainer(max_epochs=50)
491 
492     # Check whether pretrained model exists. If yes
    , load it and skip training
493     model = GraphLevelGNN(**model_kwargs)
494     print(model)
495     trainer.fit(model, train_loader, val_loader)
496     model = GraphLevelGNN.load_from_checkpoint(
    trainer.checkpoint_callback.best_model_path)
497 
498     # Test best model on validation and test set
499     # train_result = trainer.test(model, dataloaders
    =train_loader, verbose=False)
500     val_result = trainer.test(model, dataloaders=
    val_loader, verbose=False)
501     test_result = trainer.test(model, dataloaders=
    test_loader, verbose=False)
502     result = {"test": test_result[0]['test_acc'], "
    valid": val_result[0]['test_acc']}
503     return trainer, model, result
504 
505 #%%
506 from torch_geometric.loader import DataLoader
507 from sklearn.model_selection import train_test_split
508 # Step 1: Data Splitting
509 train_dataset, test_dataset = train_test_split(
    graph_dataset, test_size=0.2, random_state=14)
510 train_dataset, val_dataset = train_test_split(
    train_dataset, test_size=0.1, random_state=14)
511 # # Step 2: Define custom collate function
512 def collate(data_list):
513     return Batch.from_data_list(data_list)
514 
515 # Step 2: Data Loaders
```

```
516 train_loader = DataLoader(train_dataset, batch_size
     = 128, shuffle=False)
517 test_loader = DataLoader(test_dataset, batch_size =
    128, shuffle=False)
518 val_loader = DataLoader(val_dataset, batch_size= 128
    , shuffle=False)
519 #%%
520 device = torch.device('cuda' if torch.cuda.
    is_available() else 'cpu')
521 trainer, model, result = train_graph_classifier("
    GraphSAGE", c_in=3, c_hidden = 64, c_out=2)
522
523
524 # THIS MODEL IS GIVING A VERY POOR SCORE. There is
    some intermittent issue in exceeding the baseline
    performance of error = 0.693
525
526 # Possibilities -> Cycles are being generated in the
     graph structures, which are deviating the learned
    representations to a certain constant vector
    embedding
527 # FOR THIS REASON, RATHER THAN CREATING A COMPLEX
    GRAPH CLASSIFICATON MODEL THAT MIGHT BE EXPERIENCING
     Over-Quashing problem, I create and train a very
    shallow neural network IN THE FOLLOWING SECTION :
528 #%%
529 from torch.nn import Linear
530 import torch.nn.functional as F
531 from torch_geometric.nn import GCNConv
532 from torch_geometric.nn import global_mean_pool
533
534
535 class GCN(torch.nn.Module):
536     def __init__(self, num_in_features,
    hidden_channels, num_classes):
537         super(GCN, self).__init__()
538         torch.manual_seed(12345)
539         # self.conv1 = RandomSubsetConv(
    num_in_features, hidden_channels, subset_size = 4)
540         # self.conv2 = RandomSubsetConv(
    hidden_channels, 2*hidden_channels, subset_size = 4)
```

```
541         # self.conv3 = RandomSubsetConv(2*
    hidden_channels, 4*hidden_channels, subset_size = 4)
542         # self.conv4 = RandomSubsetConv(4*
    hidden_channels, 2*hidden_channels, subset_size = 4)
543         # self.conv5 = RandomSubsetConv(2*
    hidden_channels, hidden_channels, subset_size = 4)
544         self.conv1 = GCNConv(num_in_features,
    hidden_channels)
545         self.conv2 = GCNConv(hidden_channels, 2 *
    hidden_channels)
546         self.conv3 = GCNConv(2 * hidden_channels, 4
     * hidden_channels)
547         self.conv4 = GCNConv(4 * hidden_channels, 2
     * hidden_channels)
548         self.conv5 = GCNConv(2 * hidden_channels,
    hidden_channels)
549         self.lin = nn.Sequential(Linear(
    hidden_channels, hidden_channels//4, bias = False),
550                                  nn.LeakyReLU(
    negative_slope= 0.2),
551                                  Linear(
    hidden_channels//4, hidden_channels //8, bias =
    False),
552                                  nn.LeakyReLU(
    negative_slope=0.2),
553                                  Linear(
    hidden_channels//8, num_classes, bias = False),
554                                  )
555
556
557     def forward(self, x, edge_index, batch):
558         # 1. Obtain node embeddings
559         x = F.leaky_relu(self.conv1(x, edge_index),
    negative_slope=0.2)
560 #         x = x.relu()
561         x = F.leaky_relu(self.conv2(x, edge_index),
    negative_slope=0.2)
562 #         x = x.relu()
563         x = F.leaky_relu(self.conv3(x, edge_index),
    negative_slope= 0.2)
564
```

```
565         x = F.leaky_relu(self.conv4(x, edge_index),
    negative_slope=0.2)
566 #         x = x.relu()
567         x = F.leaky_relu(self.conv5(x, edge_index),
    negative_slope=0.2)
568 
569         # 2. Readout layer
570         x = global_mean_pool(x, batch)  # [
    batch_size, hidden_channels]
571 
572         # 3. Apply a final classifier
573         x = F.dropout(x, p=0.5, training=self.
    training)
574         x = self.lin(x)
575 
576         return x
577 
578 model = GCN(num_in_features = 3, hidden_channels=64
    , num_classes = 2)
579 print(model)
580 #%%
581 import torch
582 from torch_geometric.data import DataLoader
583 from IPython.display import Javascript
584 
585 # Assuming you've defined `GCN` class, `train_loader
    `, and `test_loader` DataLoader instances
586 
587 # Set device to CUDA if available, otherwise
    fallback to CPU
588 device = torch.device('cuda' if torch.cuda.
    is_available() else 'cpu')
589 
590 # Instantiate the model and move it to the
    appropriate device
591 model = GCN(num_in_features=3, hidden_channels=64,
    num_classes=2).to(device)
592 
593 # Move optimizer to the same device as the model
594 optimizer = torch.optim.Adam(model.parameters(), lr=
    0.001)
```

```
595 
596 # Define the loss function
597 criterion = torch.nn.CrossEntropyLoss()
598 
599 def train():
600     model.train()
601 
602     for data in train_loader:
603         # Move data to the device
604         data = data.to(device)
605 
606         out = model(data.x, data.edge_index, data.
    batch)
607         loss = criterion(out, data.y)
608         loss.backward()
609         optimizer.step()
610         optimizer.zero_grad()
611 
612 def test(loader):
613     model.eval()
614 
615     correct = 0
616     for data in loader:
617         data = data.to(device)
618         out = model(data.x, data.edge_index, data.
    batch)
619         pred = out.argmax(dim=1)
620         correct += int((pred == data.y).sum())
621     return correct / len(loader.dataset)
622 
623 # Training loop
624 for epoch in range(1, 40):
625     train()
626     train_acc = test(train_loader)
627     test_acc = test(test_loader)
628     print(f'Epoch: {epoch:03d}, Train Acc: {
    train_acc:.4f}, Test Acc: {test_acc:.4f}')
629 #%% md
630 The following contains some other templates that I
    tried/wanted to try :
631 Similar results of
```

```
632
633 `Train_acc = 71 %`  & `Test_Acc = 69%` were achieved
    .
634 #%%
635 import torch
636 import torch.nn as nn
637
638
639
640 class GraphConvolution(nn.Module):
641     def __init__(self, in_features, out_features,
    bias=True):
642         super(GraphConvolution, self).__init__()
643         self.conv = nn.Conv1d(
644             in_features, out_features, kernel_size=1
    , bias=bias
645         )
646
647     def forward(self, adj, x):
648
649 #         x = x.unsqueeze(0)  # Add a batch dimension
     at the beginning
650 #         adj = adj.unsqueeze(0)
651         # print("Inside bmm", x.shape)
652         # print("Inside bmm, adj", adj.shape)
653 #         x = torch.bmm(x, adj)
654         x = self.conv(x)
655         return x
656
657
658 class GlobalPooling(nn.Module):
659     def __init__(self):
660         super(GlobalPooling, self).__init__()
661         self.max_pool = nn.AdaptiveMaxPool1d(
    output_size=1)
662         self.avg_pool = nn.AdaptiveAvgPool1d(
    output_size=1)
663
664     def forward(self, x):
665         batch_size = x.size(0)
666 #         print('Batch Size', batch_size)
```

```
667         x0 = self.max_pool(x).view(batch_size, -1)
668 #         x1 = self.avg_pool(x).view(batch_size, -1)
669 #         x = torch.cat((x0, x1), dim=-1)
670         return torch.transpose(x0, 0, 1)
671 
672 #%%
673 import torch
674 import torch.nn as nn
675 import torch.nn.functional as F
676 from torch_geometric.nn import SAGEConv
677 from torch_geometric.nn import global_mean_pool
678 from torch.nn import Linear
679 from collections import OrderedDict
680 
681 
682 
683 
684 class MultiLayerGCN(nn.Module):
685     def __init__(self, dropout=0.5, num_classes=1):
686         super(MultiLayerGCN, self).__init__()
687         self.conv0 = GraphConvolution(3, 64, bias=
    False)
688         self.conv1 = GraphConvolution(64, 64, bias=
    False)
689         self.conv2 = GraphConvolution(64, 128, bias=
    False)
690         self.conv3 = GraphConvolution(128, 256, bias
    =False)
691         self.conv4 = GraphConvolution(256, 512, bias
    =False)
692 #         self.bn0 = nn.BatchNorm1d(64)
693 #         self.bn1 = nn.BatchNorm1d(64)
694 #         self.bn2 = nn.BatchNorm1d(128)
695 #         self.bn3 = nn.BatchNorm1d(256)
696 #         self.bn4 = nn.BatchNorm1d(1024)
697         self.pool = GlobalPooling()
698         # self.classifier = nn.Sequential(
    OrderedDict([
699         #     ('fc0', nn.Linear(512, 256, bias=False
    )),
700         #     ('relu0', nn.LeakyReLU(negative_slope=
```

```
700 0.2)),
701         #     ('bn0', nn.BatchNorm1d(256)),
702         #     ('drop0', nn.Dropout(p=dropout)),
703         #     ('fc1', nn.Linear(256, 128, bias=False
    )),
704         #     ('relu1', nn.LeakyReLU(negative_slope=
    0.2)),
705         #     ('bn1', nn.BatchNorm1d(128)),
706         #     ('drop1', nn.Dropout(p=dropout)),
707         #     ('fc2', nn.Linear(128, num_classes)),
708         # ]))
709         self.classifier = nn.Sequential(nn.Linear(
    512, 256, bias = False),
710                                         nn.LeakyReLU
    (negative_slope=0.2),
711                                         nn.Linear(
    256, 128, bias = False),
712                                         nn.LeakyReLU
    (negative_slope=0.2),
713                                         nn.Linear(
    128, 64, bias = False),
714                                         nn.LeakyReLU
    (negative_slope = 0.2),
715                                         nn.Linear(64
    , 2, bias = False))
716
717     def forward(self, adj, x):
718         # Convert the adjacency list to tensor
719  #      adj_tensor = torch.stack(adj, dim=0)
720 #         print("Shape of x ", x.shape)
721         x0 = F.leaky_relu(self.conv0(adj, x),
    negative_slope=0.2)
722 #         print("Shape of x0", x0.shape)
723         x1 = F.leaky_relu(self.conv1(adj, x0),
    negative_slope=0.2)
724 #         print("Shape of x1", x1.shape)
725         x2 = F.leaky_relu(self.conv2(adj, x1),
    negative_slope=0.2)
726 #         print("Shape of x2", x2.shape)
727         x3 = F.leaky_relu(self.conv3(adj, x2),
    negative_slope=0.2)
```

```
728         # x0 = F.leaky_relu(self.bn0(self.conv0(adj
    , x)), negative_slope=0.2)
729         # x1 = F.leaky_relu(self.bn1(self.conv1(adj
    , x0)), negative_slope=0.2)
730         # x2 = F.leaky_relu(self.bn2(self.conv2(adj
    , x1)), negative_slope=0.2)
731         # x3 = F.leaky_relu(self.bn3(self.conv3(adj
    , x2)), negative_slope=0.2)
732  #       x = torch.cat((x0, x1, x2, x3), dim=-1)
733 #        print("Shape of x3", x3.shape)
734         x = F.leaky_relu(self.conv4(adj, x3),
    negative_slope=0.2)
735
736  #       print("Again, shape of x", x.shape)
737         x = self.pool(x)
738  #       print("Shape of my heart", x.shape)
739         x = self.classifier(x)
740         return x
741
742
```