

Installing all the necessary dependencies

```
import torch
import os
os.environ['TORCH'] = torch.__version__
print(torch.__version__)
!pip install h5py
!pip install -q torch-scatter -f https://data.pyg.org/whl/torch-$
{TORCH}.html
!pip install -q torch-sparse -f https://data.pyg.org/whl/torch-$
{TORCH}.html
!pip install -q git+https://github.com/pyg-team/pytorch_geometric.git
!pip install PyGCL
!pip install dgl
!pip install pytorch_metric_learning
```

Importing all the necessary dependencies

Note : This project uses PyTorch Geometric Contrastive Learning(PyGCL), a PyTorch-based, library for all the Contrastive learning task.

```
import numpy as np
import h5py
import tqdm
import matplotlib.pyplot as plt
from sklearn.neighbors import kneighbors_graph

import torch.nn as nn
import torch.nn.functional as F
from torch.nn import Linear, ReLU
from torch.optim import Adam

from torch_geometric.nn import GCNConv, global_mean_pool
from torch_geometric.data import Data
from torch_geometric.loader import DataLoader

import GCL.augmentors as A
import GCL.losses as L
from GCL.models import DualBranchContrast
```

Mounting and Loading the data from Drive

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

```

Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).

path = "/content/drive/MyDrive/quark-gluon_data-set_n139306.hdf5"
#Path to the dataset on my google drive

with h5py.File(path, 'r') as f:
    X_jets = f['X_jets'][0:4000] # Working with only a subset of data
    # due to computational limits
    y = f['y'][0:4000]
    print(f"X_jets shape : {X_jets.shape}, y : {y.shape}") # printing
    # the shape of the images and amount

X_jets shape : (4000, 125, 125, 3), y : (4000,)

X_jets = np.array(X_jets)
y = np.array(y)

```

Converting the data to Graph format and doing preprocessing

```

dataset = []

for i, x in enumerate(X_jets):
    flattened = x.reshape(-1,3)
    non_zero = np.any(flattened != (0,0,0), axis = -1) # Removing any
    # zero element by considering only non zero ones
    node = flattened[non_zero]
    edges = kneighbors_graph(node, 2, mode = 'connectivity', include_self
    = True)
    edges = edges.tocoo()
    data = Data(x=torch.from_numpy(node),
    edge_index=torch.from_numpy(np.vstack((edges.row,edges.col))).type(torch.long),
    edge_attr=torch.from_numpy(edges.data.reshape(-1,1)),
    y=torch.tensor([int(y[i])]))
    dataset.append(data)

print(f'Number of graphs: {len(dataset)}')
print(f'Number of nodes: {dataset[0].num_nodes}')
print(f'Number of edges: {dataset[0].num_edges}')
print(f'Number of node features: {dataset[0].num_node_features}')
print(f'Number of edges features: {dataset[0].num_edge_features}')
print(dataset[0])

Number of graphs: 4000
Number of nodes: 884
Number of edges: 1768
Number of node features: 3

```

```

Number of edges features: 1
Data(x=[884, 3], edge_index=[2, 1768], edge_attr=[1768, 1], y=[1])

train_loader = DataLoader(dataset[:1000], batch_size=8, shuffle=True)
#Creating the train loader with batch = 8
test_loader = DataLoader(dataset[1000:2000], batch_size=8,
shuffle=False) # Creating the test loader with batch = 8

aug = A.Compose([A.EdgeRemoving(pe=0.3), A.FeatureMasking(pf=0.3)]) #
Selecting the graph augmentations

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

```

Creating the Contrastive model

```

class GCN(nn.Module):
    def __init__(self):
        super(GCN, self).__init__()

        self.conv1 = GCNConv(3, 32)
        self.conv2 = GCNConv(32, 32)
        self.fc1 = Linear(32, 32)
        self.fc2 = Linear(32, 32)
        self.act = ReLU()

    def forward(self, data):
        # Performing the augmentaion twice as we use dual branch
        contrastive learning
        augm_1 = aug(data.x, data.edge_index)
        augm_2 = aug(data.x, data.edge_index)

        x1 = self.conv1(augm_1[0], augm_1[1])
        x1 = self.act(x1)
        x2 = self.conv2(x1, augm_1[1])
        z1 = self.act(x2)

        x1 = self.conv1(augm_2[0], augm_2[1])
        x1 = self.act(x1)
        x2 = self.conv2(x1, augm_2[1])
        z2 = self.act(x2)

        x1 = self.conv1(data.x, data.edge_index)
        x1 = self.act(x1)
        x2 = self.conv2(x1, data.edge_index)
        z = self.act(x2)

        return z, z1, z2

    def project(self, z: torch.Tensor) -> torch.Tensor:

```

```

#Projection head to reduce the size of the embeddings
z = F.elu(self.fc1(z))
return self.fc2(z)

```

Training the contrastive learning model

```

def train(encoder_model, contrast_model, data, optimizer):
    encoder_model.train()
    optimizer.zero_grad()
    z, z1, z2 = encoder_model(data)
    h1, h2 = [encoder_model.project(x) for x in [z1, z2]] # Creating
the reduced embeddings for the contrastive learning
    loss = contrast_model(h1, h2)
    loss.backward()
    optimizer.step()
    return loss.item()

encoder_model = GCN().to(device)
#Using Dual Branch Contrastive Learning with InfoNCE loss and using
local-to-local mode[to learn local representation]
contrast_model = DualBranchContrast(loss=L.InfoNCE(tau=0.2),
mode='L2L').to(device)

optimizer = Adam(encoder_model.parameters(), lr=0.01)

for epoch in range(30):
    total_loss = 0
    for _, data in enumerate(tqdm.tqdm(train_loader)):
        data = data.to(device)
        loss = train(encoder_model, contrast_model, data, optimizer)
        total_loss += loss * data.num_graphs
    print(f'Epoch {epoch:03d}, Loss:
{total_loss/len(train_loader.dataset):.4f}')

0%|          | 0/125 [00:00<?,
?it/s]/usr/local/lib/python3.10/dist-packages/torch_geometric/deprecat
ion.py:26: UserWarning: 'dropout_adj' is deprecated, use
'dropout_edge' instead
  warnings.warn(out)
100%|██████████| 125/125 [00:07<00:00, 17.47it/s]

Epoch 000, Loss: 8.1647

100%|██████████| 125/125 [00:06<00:00, 19.28it/s]

Epoch 001, Loss: 7.9752

100%|██████████| 125/125 [00:06<00:00, 19.85it/s]

```

Epoch 002, Loss: 7.9306
100%|██████████| 125/125 [00:06<00:00, 19.20it/s]
Epoch 003, Loss: 7.8618
100%|██████████| 125/125 [00:06<00:00, 19.81it/s]
Epoch 004, Loss: 7.6426
100%|██████████| 125/125 [00:06<00:00, 19.26it/s]
Epoch 005, Loss: 7.6531
100%|██████████| 125/125 [00:06<00:00, 19.90it/s]
Epoch 006, Loss: 7.5355
100%|██████████| 125/125 [00:06<00:00, 19.19it/s]
Epoch 007, Loss: 7.4389
100%|██████████| 125/125 [00:06<00:00, 19.87it/s]
Epoch 008, Loss: 7.3446
100%|██████████| 125/125 [00:06<00:00, 19.20it/s]
Epoch 009, Loss: 7.3140
100%|██████████| 125/125 [00:06<00:00, 19.65it/s]
Epoch 010, Loss: 7.2173
100%|██████████| 125/125 [00:06<00:00, 19.06it/s]
Epoch 011, Loss: 7.1093
100%|██████████| 125/125 [00:06<00:00, 18.66it/s]
Epoch 012, Loss: 7.2032
100%|██████████| 125/125 [00:06<00:00, 18.90it/s]
Epoch 013, Loss: 7.1579
100%|██████████| 125/125 [00:06<00:00, 19.67it/s]
Epoch 014, Loss: 7.1036
100%|██████████| 125/125 [00:06<00:00, 19.09it/s]
Epoch 015, Loss: 7.1141
100%|██████████| 125/125 [00:06<00:00, 19.69it/s]

Epoch 016, Loss: 7.1741
100%|██████████| 125/125 [00:06<00:00, 18.59it/s]
Epoch 017, Loss: 7.0800
100%|██████████| 125/125 [00:06<00:00, 19.55it/s]
Epoch 018, Loss: 7.0694
100%|██████████| 125/125 [00:06<00:00, 19.00it/s]
Epoch 019, Loss: 7.0422
100%|██████████| 125/125 [00:06<00:00, 19.59it/s]
Epoch 020, Loss: 6.9561
100%|██████████| 125/125 [00:06<00:00, 19.13it/s]
Epoch 021, Loss: 7.0563
100%|██████████| 125/125 [00:06<00:00, 19.63it/s]
Epoch 022, Loss: 6.9543
100%|██████████| 125/125 [00:06<00:00, 19.06it/s]
Epoch 023, Loss: 6.8769
100%|██████████| 125/125 [00:06<00:00, 19.69it/s]
Epoch 024, Loss: 6.8457
100%|██████████| 125/125 [00:06<00:00, 19.02it/s]
Epoch 025, Loss: 6.7956
100%|██████████| 125/125 [00:06<00:00, 19.55it/s]
Epoch 026, Loss: 6.8292
100%|██████████| 125/125 [00:06<00:00, 19.14it/s]
Epoch 027, Loss: 6.7919
100%|██████████| 125/125 [00:06<00:00, 19.59it/s]
Epoch 028, Loss: 6.8415
100%|██████████| 125/125 [00:06<00:00, 19.04it/s]
Epoch 029, Loss: 6.8689

Defining the classification model

Here we use the model defined for learning representation before but without the projection head as we only need the learned representation

```
class GCNWithClassifier(nn.Module):
    def __init__(self):
        super(GCNWithClassifier, self).__init__()
        self.gcn = GCN()
        self.classifier = Linear(32, 2)

    def forward(self, data):
        z, _, _ = self.gcn(data) # Using GCN model to get embeddings
        x = global_mean_pool(z, data.batch) # Using global pooling to
get a global representation of a graph
        out = self.classifier(x)
        return out
```

Training and Testing of the Classification model

```
def train_classification(model, loader, optimizer, criterion):
    model.train()
    total_loss = 0
    for data in loader:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out, data.y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * data.num_graphs

    return total_loss / len(loader.dataset)

def test_classification(model, loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in loader:
            data = data.to(device)
            out = model(data)
            pred = out.argmax(dim=1)
            correct += (pred == data.y).sum().item() #Calculating the
correct predictions
            total += data.num_graphs
```

```

        accuracy = correct / total
    return accuracy

model = GCNWithClassifier().to(device)
optimizer_2 = Adam(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()

for epoch in range(20):
    loss = train_classification(model, train_loader, optimizer_2,
criterion)
    print(f'Epoch {epoch+1}, Loss: {loss:.4f}')
    train_acc = test_classification(model, train_loader)
    print(f'Train Accuracy: {train_acc:.4f}')

```

```

Epoch 1, Loss: 0.6936
Train Accuracy: 0.5180
Epoch 2, Loss: 0.6609
Train Accuracy: 0.6790
Epoch 3, Loss: 0.6271
Train Accuracy: 0.6890
Epoch 4, Loss: 0.6187
Train Accuracy: 0.6660
Epoch 5, Loss: 0.6182
Train Accuracy: 0.6550
Epoch 6, Loss: 0.6166
Train Accuracy: 0.6870
Epoch 7, Loss: 0.6172
Train Accuracy: 0.6800
Epoch 8, Loss: 0.6132
Train Accuracy: 0.6630
Epoch 9, Loss: 0.6204
Train Accuracy: 0.6720
Epoch 10, Loss: 0.6158
Train Accuracy: 0.6650
Epoch 11, Loss: 0.6159
Train Accuracy: 0.6700
Epoch 12, Loss: 0.6163
Train Accuracy: 0.6760
Epoch 13, Loss: 0.6181
Train Accuracy: 0.6540
Epoch 14, Loss: 0.6149
Train Accuracy: 0.6830
Epoch 15, Loss: 0.6177
Train Accuracy: 0.6670
Epoch 16, Loss: 0.6199
Train Accuracy: 0.6740
Epoch 17, Loss: 0.6173
Train Accuracy: 0.6810
Epoch 18, Loss: 0.6164
Train Accuracy: 0.6460

```



```
Epoch 19, Loss: 0.6171  
Train Accuracy: 0.6700  
Epoch 20, Loss: 0.6165  
Train Accuracy: 0.6850
```

```
test_accuracy = test_classification(model, test_loader) #Calculating  
the testing accuracy  
print(f'Test Accuracy: {test_accuracy:.4f}')
```

Test Accuracy: 0.6800

Conclusion

The model's accuracy is 68% which is not the best. There are a multitude of reasons for that.

- One big problem is graph-level representation. Although, I have used global pooling to get a graph-level representation that is not the best way.
- We only consider an extremely small subset of the actual data due to memory issues which may cause data imbalance which stops the model from learning properly.
- Another problem is the graph representation isn't being learned well. Many possible reasons can be for this such as the architecture may not be right, the parameter tuning needs to be done well, etc. Further Research into this is required.
- When constructing the contrastive learning architecture other Graph models may be used such as GAT, GraphSage, etc to learn the representation. Each of these models will learn a different representation for the node which may be better or worse but may increase the complexity of the model which may be computationally inefficient for larger datasets and graphs or also decrease.