Installing and Importing the neccessary dependencies

```
import torch
import os
os.environ['TORCH'] = torch.__version__
print(torch.__version__)

!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
```

```
2.1.0+cu121
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.8/10.8 MB 59.2 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.0/5.0 MB 27.8 MB/s eta
0:00:00
ents to build wheel ... etadata (pyproject.toml) ... etric
(pyproject.toml) ...
```

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

import torch.nn.functional as F
from torch.nn import Linear

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

Mounting and Loading the data

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

```
Mounted at /content/drive
```

```
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'][0:5000] # Working with only a subset of data
due to computational limits
  y = f['y'][0:5000]
  print(f"X_jets shape : {X_jets.shape}, y : {y.shape}") # printing
the shape of the images and amount
```

```
X_jets shape : (5000, 125, 125, 3), y : (5000,)
```

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

Creating the point cloud from the dataset

```
size = X_jets.shape[0]
cloud = []
for i in range(size):
  #Selecting only the nonzero points
  nonzero_Track = np.nonzero(X_jets[i,:,:,0])
  nonzero_ECAL = np.nonzero(X_jets[i,:,:,1])
  nonzero_HCAL = np.nonzero(X_jets[i,:,:,2])

  #Getting the values of the respective channels
  valuesTrack = X_jets[i,nonzero_Track[0],nonzero_Track[1],0]
  valuesECAL = X_jets[i,nonzero_ECAL[0],nonzero_ECAL[1],1]
  valuesHCAL = X_jets[i,nonzero_HCAL[0],nonzero_HCAL[1],2]

  #Getting the co-ordinates of the respective channels
  coord_Track = np.hstack((np.column_stack(nonzero_Track),
np.zeros((np.column_stack(nonzero_Track).shape[0],1))))
  coord_ECAL = np.hstack((np.column_stack(nonzero_ECAL),
np.zeros((np.column_stack(nonzero_ECAL).shape[0],1))))
  coord_HCAL = np.hstack((np.column_stack(nonzero_HCAL),
np.zeros((np.column_stack(nonzero_HCAL).shape[0],1))))


  cloud.append({"Track":(coord_Track, valuesTrack), "ECAL":
(coord_ECAL, valuesECAL), "HCAL":(coord_HCAL, valuesHCAL)})


sample_cloud = cloud[0]

ax = plt.figure(figsize=(15,15)).add_subplot(111,projection='3d')

ax.scatter(sample_cloud["Track"][0][:,0],sample_cloud["Track"][0]
[:,1],sample_cloud["Track"][0][:,2],c=sample_cloud["Track"][1])
ax.scatter(sample_cloud["ECAL"][0][:,0],sample_cloud["ECAL"][0]
[:,1],sample_cloud["ECAL"][0][:,2],c=sample_cloud["ECAL"][1])
ax.scatter(sample_cloud["HCAL"][0][:,0],sample_cloud["HCAL"][0]
```

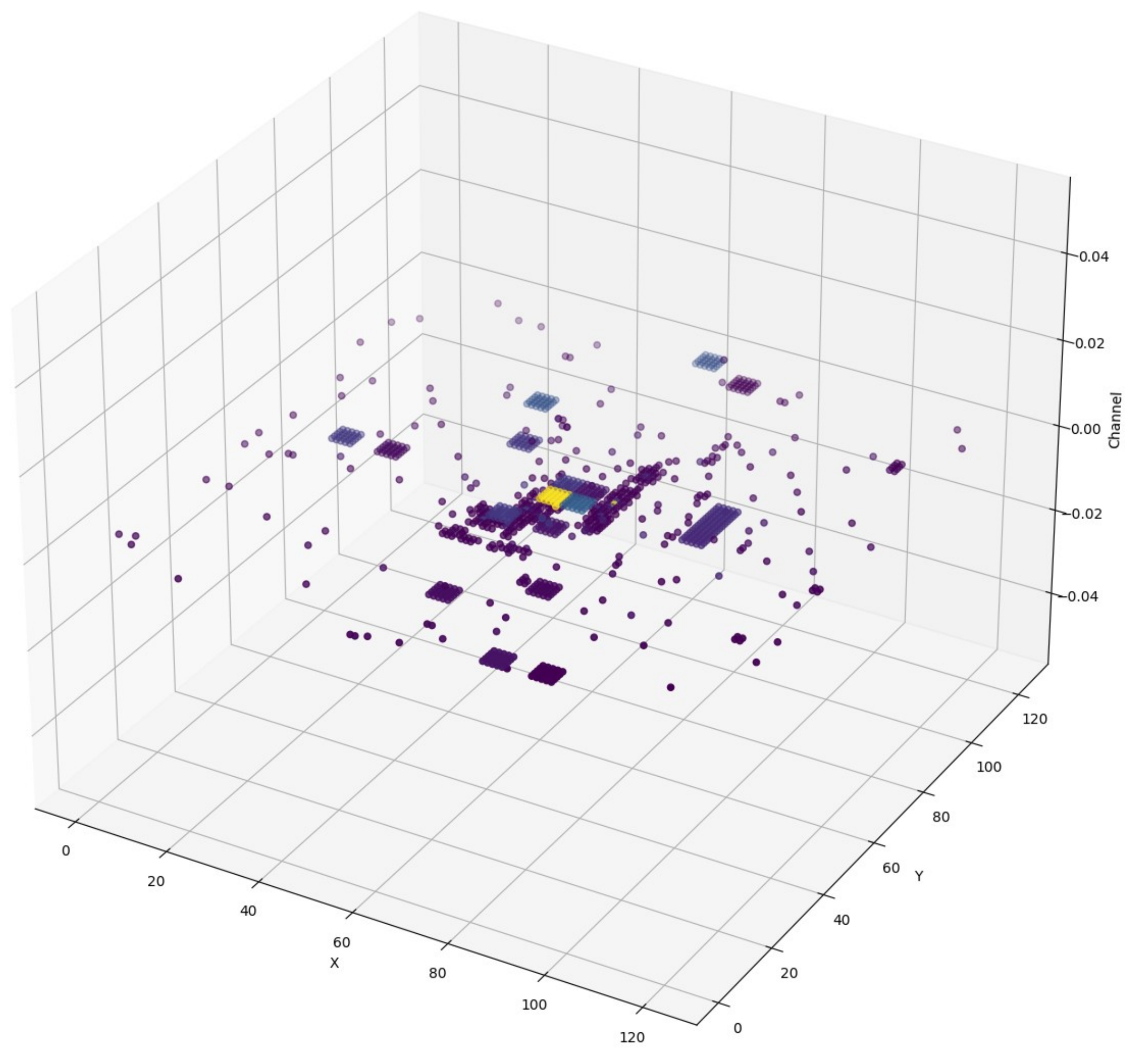
```
[:,1],sample_cloud["HCAL"][0][:,2],c=sample_cloud["HCAL"][1])

# sample_cloud['Track'

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel("Channel")
ax.set_title("Point cloud for the first X_jet Image")

plt.show()
```

Point cloud for the first X_jet Image



Converting the dataset to a graph format

```
dataset = []

for i, x in enumerate(X_jets):
  flattened = x.reshape(-1,3)
  non_zero = np.any(flattened != (0,0,0), axis = -1)
  node = flattened[non_zero]
  edges = kneighbors_graph(node, 4, 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(tor
ch.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: 5000
Number of nodes: 884
Number of edges: 3536
Number of node features: 3
Number of edges features: 1
Data(x=[884, 3], edge_index=[2, 3536], edge_attr=[3536, 1], y=[1])
```

Splitting the dataset to train, test and validation split

```
train_loader = DataLoader(dataset[:3000], batch_size=32, shuffle=True)
test_loader = DataLoader(dataset[3000:4000], batch_size=32,
shuffle=False)
val_loader = DataLoader(dataset[4000:], batch_size = 32, shuffle =
False)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Defining and traing Model 1

This model is based on Graph Convolution.

```python
class GCN(torch.nn.Module):
  def __init__(self, hidden_channels):
    super(GCN, self).__init__()
    torch.manual_seed(42)
    self.conv1 = GCNConv(dataset[0].num_node_features,
hidden_channels)
    self.conv2 = GCNConv(hidden_channels, 2*hidden_channels)
    self.lin = Linear(2*hidden_channels, 2)

  def forward(self, x, edge_index, batch):
    x = self.conv1(x, edge_index)
    x = x.relu()
    x = self.conv2(x, edge_index)
    x = global_mean_pool(x, batch)

    x = F.dropout(x, p = 0.5, training = self.training)
    x = self.lin(x)

    return x

model = GCN(hidden_channels = 32).to(device)
print(model)
```

```
GCN(
  (conv1): GCNConv(3, 32)
  (conv2): GCNConv(32, 64)
  (lin): Linear(in_features=64, out_features=2, bias=True)
)
```

```python
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = torch.nn.CrossEntropyLoss().to(device)

def train():
    model.train()

    for data in train_loader:  # Iterate in batches over the training
dataset.

         data = data.to(device)
         out = model(data.x, data.edge_index, data.batch)  # Perform a
single forward pass.
         loss = criterion(out, data.y) # Compute the loss.
         loss.backward()  # Derive gradients.
         optimizer.step()  # Update parameters based on gradients.
         optimizer.zero_grad()  # Clear gradients.

def test(loader):
     model.eval()

     correct = 0
     for data in loader:  # Iterate in batches over the training/test
```

```
dataset.
        data = data.to(device)
        out = model(data.x, data.edge_index, data.batch)
        pred = out.argmax(dim=1)  # Use the class with highest
probability.
        correct += int((pred == data.y).sum())  # Check against
ground-truth labels.
     return correct / len(loader.dataset)  # Derive ratio of correct
predictions.

def val(loader):
  model.eval()
  correct = 0
  with torch.no_grad():
    for data in loader:
      data = data.to(device)
      out = model(data.x, data.edge_index, data.batch)
      pred = out.argmax(dim=1)  # Use the class with highest
probability.
      correct += int((pred == data.y).sum())  # Check against ground-
truth labels.
    return correct / len(loader.dataset)

for epoch in range(1, 31):
    train()
    train_acc = test(train_loader)
    test_acc = test(test_loader)
    val_acc = val(val_loader)
    print(f'Epoch: {epoch:03d}, Train Acc: {train_acc:.4f}, Test Acc:
{test_acc:.4f}, Val Acc : {val_acc:.4f}')
```

```
Epoch: 001, Train Acc: 0.6767, Test Acc: 0.6830, Val Acc : 0.7060
Epoch: 002, Train Acc: 0.6783, Test Acc: 0.6810, Val Acc : 0.7100
Epoch: 003, Train Acc: 0.6797, Test Acc: 0.6850, Val Acc : 0.7080
Epoch: 004, Train Acc: 0.6753, Test Acc: 0.6770, Val Acc : 0.7030
Epoch: 005, Train Acc: 0.6707, Test Acc: 0.6760, Val Acc : 0.7010
Epoch: 006, Train Acc: 0.6777, Test Acc: 0.6820, Val Acc : 0.7110
Epoch: 007, Train Acc: 0.6760, Test Acc: 0.6830, Val Acc : 0.7030
Epoch: 008, Train Acc: 0.6787, Test Acc: 0.6810, Val Acc : 0.7100
Epoch: 009, Train Acc: 0.6733, Test Acc: 0.6810, Val Acc : 0.7030
Epoch: 010, Train Acc: 0.6757, Test Acc: 0.6730, Val Acc : 0.7040
Epoch: 011, Train Acc: 0.6740, Test Acc: 0.6820, Val Acc : 0.7030
Epoch: 012, Train Acc: 0.6793, Test Acc: 0.6850, Val Acc : 0.7090
Epoch: 013, Train Acc: 0.6800, Test Acc: 0.6830, Val Acc : 0.7080
Epoch: 014, Train Acc: 0.6800, Test Acc: 0.6820, Val Acc : 0.7080
Epoch: 015, Train Acc: 0.6750, Test Acc: 0.6820, Val Acc : 0.7030
Epoch: 016, Train Acc: 0.6807, Test Acc: 0.6820, Val Acc : 0.7080
Epoch: 017, Train Acc: 0.6740, Test Acc: 0.6810, Val Acc : 0.7010
Epoch: 018, Train Acc: 0.6763, Test Acc: 0.6830, Val Acc : 0.7060
Epoch: 019, Train Acc: 0.6790, Test Acc: 0.6810, Val Acc : 0.7090
```

```
Epoch: 020, Train Acc: 0.6783, Test Acc: 0.6810, Val Acc : 0.7100
Epoch: 021, Train Acc: 0.6790, Test Acc: 0.6810, Val Acc : 0.7090
Epoch: 022, Train Acc: 0.6740, Test Acc: 0.6820, Val Acc : 0.7040
Epoch: 023, Train Acc: 0.6710, Test Acc: 0.6750, Val Acc : 0.6990
Epoch: 024, Train Acc: 0.6800, Test Acc: 0.6820, Val Acc : 0.7080
Epoch: 025, Train Acc: 0.6793, Test Acc: 0.6840, Val Acc : 0.7080
Epoch: 026, Train Acc: 0.6750, Test Acc: 0.6730, Val Acc : 0.7030
Epoch: 027, Train Acc: 0.6787, Test Acc: 0.6820, Val Acc : 0.7100
Epoch: 028, Train Acc: 0.6770, Test Acc: 0.6750, Val Acc : 0.7020
Epoch: 029, Train Acc: 0.6763, Test Acc: 0.6750, Val Acc : 0.7030
Epoch: 030, Train Acc: 0.6790, Test Acc: 0.6830, Val Acc : 0.7080
```

Defining and traing Model 2

This model is based on Graph Attention.

```
class GAT(torch.nn.Module):
  def __init__(self, hidden_channels):
    super(GAT, self).__init__()
    torch.manual_seed(42)
    self.conv1 = GATConv(dataset[0].num_node_features,
hidden_channels)
    self.conv2 = GATConv(hidden_channels, 2*hidden_channels)
    self.lin = Linear(2*hidden_channels, 2)

  def forward(self, x, edge_index, batch):
    x = self.conv1(x, edge_index)
    x = F.elu(x)
    x = self.conv2(x, edge_index)
    x = global_mean_pool(x, batch)

    x = F.dropout(x, p = 0.5, training = self.training)
    x = self.lin(x)

    return F.log_softmax(x, dim = 1)

model_2 = GAT(hidden_channels = 32).to(device)
print(model_2)
```

```
GAT(
  (conv1): GATConv(3, 32, heads=1)
  (conv2): GATConv(32, 64, heads=1)
  (lin): Linear(in_features=64, out_features=2, bias=True)
)
```

```
optimizer = torch.optim.Adam(model_2.parameters(), lr=0.001)
criterion = torch.nn.CrossEntropyLoss().to(device)
```

```python
def train():
    model_2.train()

    for data in train_loader:  # Iterate in batches over the training
dataset.

         data = data.to(device)
         out = model_2(data.x, data.edge_index, data.batch)  # Perform
a single forward pass.
         loss = criterion(out, data.y) # Compute the loss.
         loss.backward()  # Derive gradients.
         optimizer.step()  # Update parameters based on gradients.
         optimizer.zero_grad()  # Clear gradients.

def test(loader):
     model_2.eval()

     correct = 0
     for data in loader:  # Iterate in batches over the training/test
dataset.
         data = data.to(device)
         out = model_2(data.x, data.edge_index, data.batch)
         pred = out.argmax(dim=1)  # Use the class with highest
probability.
         correct += int((pred == data.y).sum())  # Check against
ground-truth labels.
     return correct / len(loader.dataset)  # Derive ratio of correct
predictions.

def val(loader):
  model_2.eval()
  correct = 0
  with torch.no_grad():
    for data in loader:
      data = data.to(device)
      out = model_2(data.x, data.edge_index, data.batch)
      pred = out.argmax(dim=1)  # Use the class with highest
probability.
      correct += int((pred == data.y).sum())  # Check against ground-
truth labels.
    return correct / len(loader.dataset)

for epoch in range(1, 31):
    train()
    train_acc = test(train_loader)
    test_acc = test(test_loader)
    val_acc = val(val_loader)
    print(f'Epoch: {epoch:03d}, Train Acc: {train_acc:.4f}, Test Acc:
{test_acc:.4f}, Val Acc : {val_acc:.4f}')
```

```
Epoch: 001, Train Acc: 0.5080, Test Acc: 0.4970, Val Acc : 0.4880
Epoch: 002, Train Acc: 0.5080, Test Acc: 0.4970, Val Acc : 0.4880
Epoch: 003, Train Acc: 0.5080, Test Acc: 0.4970, Val Acc : 0.4880
Epoch: 004, Train Acc: 0.5167, Test Acc: 0.5270, Val Acc : 0.5360
Epoch: 005, Train Acc: 0.6113, Test Acc: 0.6230, Val Acc : 0.6320
Epoch: 006, Train Acc: 0.5803, Test Acc: 0.5920, Val Acc : 0.5880
Epoch: 007, Train Acc: 0.5620, Test Acc: 0.5720, Val Acc : 0.5700
Epoch: 008, Train Acc: 0.5890, Test Acc: 0.5890, Val Acc : 0.5910
Epoch: 009, Train Acc: 0.6287, Test Acc: 0.6400, Val Acc : 0.6480
Epoch: 010, Train Acc: 0.5850, Test Acc: 0.5780, Val Acc : 0.5780
Epoch: 011, Train Acc: 0.6767, Test Acc: 0.6750, Val Acc : 0.7000
Epoch: 012, Train Acc: 0.6763, Test Acc: 0.6750, Val Acc : 0.7040
Epoch: 013, Train Acc: 0.6697, Test Acc: 0.6690, Val Acc : 0.6930
Epoch: 014, Train Acc: 0.6693, Test Acc: 0.6720, Val Acc : 0.6940
Epoch: 015, Train Acc: 0.6767, Test Acc: 0.6750, Val Acc : 0.7040
Epoch: 016, Train Acc: 0.6327, Test Acc: 0.6410, Val Acc : 0.6470
Epoch: 017, Train Acc: 0.6770, Test Acc: 0.6750, Val Acc : 0.7040
Epoch: 018, Train Acc: 0.6647, Test Acc: 0.6700, Val Acc : 0.6890
Epoch: 019, Train Acc: 0.6767, Test Acc: 0.6790, Val Acc : 0.7000
Epoch: 020, Train Acc: 0.6760, Test Acc: 0.6730, Val Acc : 0.7060
Epoch: 021, Train Acc: 0.6663, Test Acc: 0.6740, Val Acc : 0.6870
Epoch: 022, Train Acc: 0.6677, Test Acc: 0.6730, Val Acc : 0.6870
Epoch: 023, Train Acc: 0.6740, Test Acc: 0.6720, Val Acc : 0.7050
Epoch: 024, Train Acc: 0.6660, Test Acc: 0.6740, Val Acc : 0.6870
Epoch: 025, Train Acc: 0.6680, Test Acc: 0.6740, Val Acc : 0.6880
Epoch: 026, Train Acc: 0.6450, Test Acc: 0.6670, Val Acc : 0.6650
Epoch: 027, Train Acc: 0.6650, Test Acc: 0.6630, Val Acc : 0.6740
Epoch: 028, Train Acc: 0.6777, Test Acc: 0.6710, Val Acc : 0.7060
Epoch: 029, Train Acc: 0.6773, Test Acc: 0.6740, Val Acc : 0.7060
Epoch: 030, Train Acc: 0.6723, Test Acc: 0.6750, Val Acc : 0.6970
```

Defining and traing Model 3 (Selected model)

This model is based on Graph Sage.

```
class GSage(torch.nn.Module):
  def __init__(self, hidden_channels):
    super(GSage, self).__init__()
    torch.manual_seed(42)
    self.conv1 = SAGEConv(3, hidden_channels)
    self.conv2 = SAGEConv(hidden_channels, 2*hidden_channels)
    self.lin = Linear(2*hidden_channels, 2)

  def forward(self, x, edge_index, batch):
    x = self.conv1(x, edge_index)
    x = F.relu(x)
    x = self.conv2(x, edge_index)
```

```python
    x = global_mean_pool(x, batch)

    x = F.dropout(x, p = 0.5, training = self.training)
    x = self.lin(x)

    return x
model_3 = GSage(hidden_channels = 32).to(device)
print(model_3)
```

```
GSage(
  (conv1): SAGEConv(3, 32, aggr=max)
  (conv2): SAGEConv(32, 64, aggr=max)
  (lin): Linear(in_features=64, out_features=2, bias=True)
)
```

```python
optimizer = torch.optim.Adam(model_3.parameters(), lr=0.001)
criterion = torch.nn.CrossEntropyLoss().to(device)

def train():
    model_3.train()

    for data in train_loader:  # Iterate in batches over the training
dataset.

         data = data.to(device)
         out = model_3(data.x, data.edge_index, data.batch)  # Perform
a single forward pass.
         loss = criterion(out, data.y) # Compute the loss.
         loss.backward()  # Derive gradients.
         optimizer.step()  # Update parameters based on gradients.
         optimizer.zero_grad()  # Clear gradients.

def test(loader):
     model_3.eval()

     correct = 0
     for data in loader:  # Iterate in batches over the training/test
dataset.
         data = data.to(device)
         out = model_3(data.x, data.edge_index, data.batch)
         pred = out.argmax(dim=1)  # Use the class with highest
probability.
         correct += int((pred == data.y).sum())  # Check against
ground-truth labels.
     return correct / len(loader.dataset)  # Derive ratio of correct
predictions.

def val(loader):
  model_3.eval()
  correct = 0
```

```
  with torch.no_grad():
    for data in loader:
      data = data.to(device)
      out = model_3(data.x, data.edge_index, data.batch)
      pred = out.argmax(dim=1)  # Use the class with highest
probability.
      correct += int((pred == data.y).sum())  # Check against ground-
truth labels.
    return correct / len(loader.dataset)

for epoch in range(1, 31):
    train()
    train_acc = test(train_loader)
    test_acc = test(test_loader)
    val_acc = val(val_loader)
    print(f'Epoch: {epoch:03d}, Train Acc: {train_acc:.4f}, Test Acc:
{test_acc:.4f}, Val Acc : {val_acc:.4f}')
```

```
Epoch: 001, Train Acc: 0.5620, Test Acc: 0.5810, Val Acc : 0.5720
Epoch: 002, Train Acc: 0.5400, Test Acc: 0.5420, Val Acc : 0.5220
Epoch: 003, Train Acc: 0.5203, Test Acc: 0.5090, Val Acc : 0.5070
Epoch: 004, Train Acc: 0.6170, Test Acc: 0.6370, Val Acc : 0.6270
Epoch: 005, Train Acc: 0.6650, Test Acc: 0.6660, Val Acc : 0.6720
Epoch: 006, Train Acc: 0.6467, Test Acc: 0.6450, Val Acc : 0.6580
Epoch: 007, Train Acc: 0.6730, Test Acc: 0.6790, Val Acc : 0.7000
Epoch: 008, Train Acc: 0.6670, Test Acc: 0.6760, Val Acc : 0.6780
Epoch: 009, Train Acc: 0.6683, Test Acc: 0.6780, Val Acc : 0.6870
Epoch: 010, Train Acc: 0.6810, Test Acc: 0.6830, Val Acc : 0.7050
Epoch: 011, Train Acc: 0.6843, Test Acc: 0.6980, Val Acc : 0.7150
Epoch: 012, Train Acc: 0.6793, Test Acc: 0.6890, Val Acc : 0.7080
Epoch: 013, Train Acc: 0.6690, Test Acc: 0.6800, Val Acc : 0.6940
Epoch: 014, Train Acc: 0.6850, Test Acc: 0.7020, Val Acc : 0.7150
Epoch: 015, Train Acc: 0.6870, Test Acc: 0.6910, Val Acc : 0.7070
Epoch: 016, Train Acc: 0.6873, Test Acc: 0.6950, Val Acc : 0.7160
Epoch: 017, Train Acc: 0.6893, Test Acc: 0.6970, Val Acc : 0.7210
Epoch: 018, Train Acc: 0.6853, Test Acc: 0.6990, Val Acc : 0.7180
Epoch: 019, Train Acc: 0.6860, Test Acc: 0.7000, Val Acc : 0.7170
Epoch: 020, Train Acc: 0.6780, Test Acc: 0.6900, Val Acc : 0.7070
Epoch: 021, Train Acc: 0.6837, Test Acc: 0.7010, Val Acc : 0.7180
Epoch: 022, Train Acc: 0.6877, Test Acc: 0.7000, Val Acc : 0.7190
Epoch: 023, Train Acc: 0.6777, Test Acc: 0.6870, Val Acc : 0.7040
Epoch: 024, Train Acc: 0.6877, Test Acc: 0.6940, Val Acc : 0.7200
Epoch: 025, Train Acc: 0.6863, Test Acc: 0.6990, Val Acc : 0.7200
Epoch: 026, Train Acc: 0.6843, Test Acc: 0.7050, Val Acc : 0.7160
Epoch: 027, Train Acc: 0.6840, Test Acc: 0.6980, Val Acc : 0.7130
Epoch: 028, Train Acc: 0.6857, Test Acc: 0.7000, Val Acc : 0.7200
Epoch: 029, Train Acc: 0.6823, Test Acc: 0.7040, Val Acc : 0.7170
Epoch: 030, Train Acc: 0.6857, Test Acc: 0.7000, Val Acc : 0.7200
```

Discussion

The final architecture choosen here is the Graph Sage architecture.

- Firstly, among all three architecture we experimented on Graph Sage performed just slightly better.
- From a computational point of view Graph Sage is much more efficient than the other two architecture.
- Although the performance between the models is more or less same here it can change when we use large batches or use a bigger subset of data. In such cases Graph Attention Networks are better as they will capture the deeper relation between the nodes, due to the attention mechanism, much better as compared to Graph Sage but at a trade off of performance.
- Another factor to consider here is how we are representing the Images as graph. We can choose different node features such as the Values of the channels or momentum etc or use euclidian metric instead of adjacency to be in the edge embedding to enrich the graph more. This may help in capturing a better representation of the graph overall.