Task 2

1. Create a point cloud representation
2. Create a graph representation from the given point cloud
3. Train a GNN model from Quark/Gluon classification

Issue

- A point cloud is nothing but a 3d representation of a point but the give data only contains 2d data i.e x, y coordinate and value of the particular channel at that point.

In [ ]:
```
from torch.utils.data import Dataset as TorchDataset
import h5py
import torch
from torch_geometric.data import Dataset as PygDataset, Data
from torch_geometric.loader import DataLoader
import numpy as np
import os.path as osp
from torchmetrics import Accuracy, Precision, Recall
import os
import multiprocessing as mp
import torch.nn.functional as F
from multiprocessing import Pool
from torch_geometric.nn import GATConv, Linear, TopKPooling, global_max_poo
from torchvision import transforms
from torch_geometric.datasets import Planetoid
from tqdm import tqdm
cpu_count = mp.cpu_count()
device = "cpu"
```

In [ ]:
```
def subset_dataset(raw_path, processed_path, subset_len = 6000, starter = 0
    with h5py.File(raw_path, 'r') as f, h5py.File(processed_path, 'w') as p
        keys = list(f.keys())
        total_events = f[keys[1]].shape[0]
        for key in keys:
            shape = (subset_len,)
            if len(f[key].shape) > 1:
                shape = (subset_len, 125, 125, 3)
            p.create_dataset(key, shape=shape)
        quark_count = 0
        gluon_count = 0
        idx = 0
        for i in range(starter, starter + subset_len):
            if quark_count < subset_len // 2:
                for key in keys:
                    p[key][idx] = f[key][i]
                quark_count += 1
                idx += 1
            elif gluon_count < subset_len // 2:
                for key in keys:
                    p[key][idx] = f[key][i]
                gluon_count += 1
                idx+=1
```

The given dataset is too large so instead a small subset is
is used as a POC for Quark/Gluon classification using Graph Neural Networks.

In [ ]:
```
train_path = "../Data/hdf5/processed/train.hdf5"
val_path = "../Data/hdf5/processed/val.hdf5"
test_path = "../Data/hdf5/processed/test.hdf5"
quark_gluon_path = "../Data/hdf5/processed/quark-gluon-dataset.hdf5"
```

In [ ]:
```
subset_dataset(quark_gluon_path, train_path, 600)
subset_dataset(quark_gluon_path, val_path, 120, 600)
subset_dataset(quark_gluon_path, test_path, 120, 720)
```

To create a graph representation we treat all non-zero positions of any channel as nodes and these non zero points will have the features as the channel values i.e [ecal, hcal, tracks]
at that particular position. Edges are formed between nodes by calculating the k-nearest neighbours using euclidean distance.

In [ ]:
```
def get_pillow(x):
    return x.transpose((2,1,0))
def get_k_nearest(indices, k = 10):
    edges = None
    for i in range(indices.shape[0]):
        k_nearest = np.sum((indices - indices[i])**2, axis=1).argsort()
        k_nearest_edges = np.array([[i, j] for j in k_nearest[1:k]])
        if edges is None:
            edges = k_nearest_edges
        else:
            edges = np.vstack((edges, k_nearest_edges))
    return edges
def create_graph(idx,quark_gluon_path ,outpath ):
    data = Data()
    with h5py.File(quark_gluon_path, 'r') as f:
        y = f['y'][idx]
        x = f['X_jets'][idx]
        non_zero_indices = np.argwhere(np.sum(x, axis=2))
        non_zero_fetures = x[non_zero_indices[:, 0], non_zero_indices[:, 1]
        data.x = torch.from_numpy(non_zero_fetures)
        edges = get_k_nearest(non_zero_indices)
        data.edge_index = torch.from_numpy(edges).t().contiguous().to(torch
        data.y = torch.from_numpy(np.asarray([y]))
        data.pos = torch.from_numpy(non_zero_indices)
        torch.save(data, osp.join(outpath, f"{idx}.pt"))
```

In [ ]:
```
def grapher(root_dir = "../Data/hdf5/processed"):
    files = ["train.hdf5", "val.hdf5", "test.hdf5"]
    for file in files:
        path = osp.join(root_dir , file)
        with h5py.File(path, 'r') as f:
            event_count = len(f["X_jets"])
        data = file.split(".")[0]
        for i in range(event_count):
            create_graph(i, path , "../Data/Graphs/{}/raw".format(data))
```

In [ ]:
```
grapher()
```

In [ ]:
```
class QuarkGluonGraphs(PygDataset):
    def __init__(self, root = None, transform = None, pre_transform = None,
        super().__init__(root, transform, pre_transform, pre_filter, log)

    @property
    def raw_file_names(self):
```

```
        return os.listdir(osp.join(self.root, "raw"))

    @property
    def processed_file_names(self):
        return os.listdir(osp.join(self.root, "raw"))
    def download(self):
        pass

    def process(self):
        for raw_path in self.raw_file_names:
            data = torch.load(osp.join(self.raw_dir, raw_path))
            data.y = data.y.to(torch.int64)
            torch.save(data, osp.join(self.processed_dir, raw_path))
    def len(self):
        return len(self.processed_file_names)
    def get(self, idx):
        data = torch.load(osp.join(self.processed_dir, f"{idx}.pt"))
        if self.transform is not None:
            data.x = self.transform(data.x)
        return data
```

In [ ]:
```
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
transform = None
train_data = QuarkGluonGraphs("../Data/Graphs/train/", transform=transform)
val_data = QuarkGluonGraphs("../Data/Graphs/val/", transform=transform)
test_data = QuarkGluonGraphs("../Data/Graphs/test/", transform=transform)
```

GCN model in the given github repository is used as a base to complete the task.

In [ ]:
```
class GCN(torch.nn.Module):
    def __init__(self, feature_size):
        super().__init__()
        num_classes = 2
        embedding_size = 256

        # GNN Layers
        self.conv1 = GATConv(feature_size, embedding_size, heads=3, dropout
        self.head_transform1 = Linear(embedding_size*3, embedding_size)
        self.pool1 = TopKPooling(embedding_size, ratio=0.8)
        self.conv2 = GATConv(embedding_size, embedding_size, heads=3, dropo
        self.head_transform2 = Linear(embedding_size*3, embedding_size)
        self.pool2 = TopKPooling(embedding_size, ratio=0.5)
        self.conv3 = GATConv(embedding_size, embedding_size, heads=3, dropo
        self.head_transform3 = Linear(embedding_size*3, embedding_size)
        self.pool3 = TopKPooling(embedding_size, ratio=0.2)

        # Linear Layers
        self.linear1 = Linear(embedding_size*2, embedding_size)
        self.linear2 = Linear(embedding_size, num_classes)
        self.softmax = torch.nn.Softmax(dim = -1)

    def forward(self, x, edge_index, batch_index):
        # first block
        x = self.conv1(x, edge_index)
        x = self.head_transform1(x)

        x, edge_index, _, batch_index, _, _ = self.pool1(x, edge_index, Non
        x1 = torch.cat([gmp(x, batch_index), gap(x, batch_index)], dim=1)

        # second block
        x = self.conv2(x, edge_index)
        x = self.head_transform2(x)
```

```
        x, edge_index, _, batch_index, _, _ = self.pool2(x, edge_index, Non
        x2 = torch.cat([gmp(x, batch_index), gap(x, batch_index)], dim=1)

        # Third block
        x = self.conv3(x, edge_index)
        x = self.head_transform3(x)

        x, edge_index, _, batch_index, _, _ = self.pool3(x, edge_index, Non
        x3 = torch.cat([gmp(x, batch_index), gap(x, batch_index)], dim=1)

        # concat pooled vectors
        x = x1 + x2 + x3

        # output block
        x = self.linear1(x).relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.linear2(x)

        return self.softmax(x)
```

In [ ]:
```
model = GCN(feature_size=train_data[0].x.shape[1])
```

In [ ]:
```
model = model.to(device)
```

In [ ]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

In [ ]:
```
count_parameters(model)
```

Out[ ]:
```
1125634
```

In [ ]:
```
model
```

Out[ ]:
```
GCN(
  (conv1): GATConv(3, 256, heads=3)
  (head_transform1): Linear(768, 256, bias=True)
  (pool1): TopKPooling(256, ratio=0.8, multiplier=1.0)
  (conv2): GATConv(256, 256, heads=3)
  (head_transform2): Linear(768, 256, bias=True)
  (pool2): TopKPooling(256, ratio=0.5, multiplier=1.0)
  (conv3): GATConv(256, 256, heads=3)
  (head_transform3): Linear(768, 256, bias=True)
  (pool3): TopKPooling(256, ratio=0.2, multiplier=1.0)
  (linear1): Linear(512, 256, bias=True)
  (linear2): Linear(256, 2, bias=True)
  (softmax): Softmax(dim=-1)
)
```

In [ ]:
```
# weights = torch.tensor([0, 1], dtype=torch.float32).to(device)
loss_fn = torch.nn.CrossEntropyLoss()#(weight=weights)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
```

In [ ]:
```
NUM_GRAPHS_PER_BATCH = 8
train_loader = DataLoader(train_data,
                          batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True
                          )
test_loader = DataLoader(test_data,
                         batch_size=NUM_GRAPHS_PER_BATCH,
                         )
```

In [ ]:
```
accuracy = Accuracy("binary", num_classes=2)
precision = Precision("binary", num_classes=2)
recall = Recall("binary", num_classes=2)
```

In [ ]:
```
def train(epochs, model, train_loader, loss_fn):
    # Enumerate over the data
    all_preds = []
    all_labels = []
    for _, batch in enumerate((train_loader)):
        # Reset gradients
        optimizer.zero_grad()
        # passing the node features and the connection info
        pred = model(batch.x,
                     batch.edge_index.to(torch.int64),
                     batch.batch
                     )
        # Calculate the loss and the gradient
        # print(pred.shape)
        loss = torch.sqrt(loss_fn(pred, batch.y.float()))
        loss.backward()
        # Update using the gradients
        optimizer.step()

        all_preds.append(np.argmax(pred.cpu().detach().numpy(), axis=1))
        all_labels.append(batch.y.cpu().detach().numpy())
    all_preds = np.concatenate(all_preds).ravel()
    all_labels = np.concatenate(all_labels).ravel()
    return loss

def test(epoch, model, test_loader, loss_fn):
    all_preds = []
    all_preds_raw = []
    all_labels = []
    running_loss = 0.0
    step = 0
    for batch in test_loader:
        # batch.to(device)
        batch.edge_index = batch.edge_index.to(torch.int64)
        pred = model(batch.x,
                        batch.edge_index,
                        batch.batch)
        pred_ = torch.argmax(pred, dim = 1)
        y = torch.argmax(batch.y, dim  = 1)
        # print(pred.shape, pred_.shape, y.shape)
        loss = torch.sqrt(loss_fn(pred, batch.y.float()))

        prec = precision(pred_, y)
        rec = recall(pred_, y)
        acc = accuracy(pred_, y)
         # Update tracking
        running_loss += loss.item()
        step += 1
        all_preds.append(pred_)
        all_labels.append(y)

    all_preds = torch.cat(all_preds)
    all_labels = torch.cat(all_labels)
    # print(all_preds.shape, all_labels.shape)
    prec = precision(all_preds, all_labels)
    # print(all_preds)
    # print(all_labels)
    rec = recall(all_preds, all_labels)
    acc = accuracy(all_preds, all_labels)
```

```
    # print(all_preds_raw[0][:10])
    # print(all_preds[:10])
    # print(all_labels[:10])
    return running_loss, prec, rec, acc
```

In [ ]:
```
for epoch in range(10):
    model.train()
    running_loss = train(epoch, model, train_loader, loss_fn)
    running_loss = running_loss.detach().cpu().numpy()
    print(" Epoch {} | training loss {}".format(epoch, running_loss))
    scheduler.step()
    with torch.no_grad():
        running_loss, prec, rec, acc = test(epoch, model, test_loader, loss_
    print(" Epoch {} | testing loss {} | precision {} | recall {} | accurac
```

```
 Epoch 0 | training loss 0.835361897945404
 Epoch 0 | testing loss 124.92844700813293 | precision 0.4866666793823242 |
recall 1.0 | accuracy 0.4866666793823242
 Epoch 1 | training loss 0.8361220359802246
 Epoch 1 | testing loss 125.12552988529205 | precision 0.0 | recall 0.0 | a
ccuracy 0.5133333206176758
 Epoch 2 | training loss 0.8330801725387573
 Epoch 2 | testing loss 124.88511437177658 | precision 0.4866666793823242 |
recall 1.0 | accuracy 0.4866666793823242
 Epoch 3 | training loss 0.8235042095184326
 Epoch 3 | testing loss 125.25484645366669 | precision 0.4866666793823242 |
recall 1.0 | accuracy 0.4866666793823242
 Epoch 4 | training loss 0.83642578125
 Epoch 4 | testing loss 124.90925747156143 | precision 0.4866666793823242 |
recall 1.0 | accuracy 0.4866666793823242
 Epoch 5 | training loss 0.8318431377410889
 Epoch 5 | testing loss 124.88898611068726 | precision 0.4866666793823242 |
recall 1.0 | accuracy 0.4866666793823242
 Epoch 6 | training loss 0.8323637247085571
 Epoch 6 | testing loss 124.87893605232239 | precision 0.0 | recall 0.0 | a
ccuracy 0.5133333206176758
 Epoch 7 | training loss 0.8286725282669067
 Epoch 7 | testing loss 124.84818017482758 | precision 0.0 | recall 0.0 | a
ccuracy 0.5133333206176758
 Epoch 8 | training loss 0.8458583354949951
 Epoch 8 | testing loss 125.36048144102097 | precision 0.4866666793823242 |
recall 1.0 | accuracy 0.4866666793823242
 Epoch 9 | training loss 0.824140191078186
 Epoch 9 | testing loss 124.91393315792084 | precision 0.0 | recall 0.0 | a
ccuracy 0.5133333206176758
```

The given GCN model uses node features {ecal, hcal, tracks} and non-weighted edges for the given classification.

Possible improvements:

- Utilise edge attribute as distance between the nodes
- Utilise positional(geometric) information.
- Try and check different pooling layers and select the best for our use case.

In [ ]: