Graph Neural Networks for Particle Momentum Estimation in the CMS Trigger System Task-2

Installing Requirements

```
pip install energyflow
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/cola
Collecting energyflow
  Downloading EnergyFlow-1.3.2-py2.py3-none-any.whl (700 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 700.5/700.5 KB 11.7 MB/s eta 0:
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.8/dist
Collecting wasserstein>=0.3.1
  Downloading Wasserstein-1.1.0-cp38-cp38-manylinux_2_17_x86_64.manylinux20
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 503.0/503.0 KB 29.5 MB/s eta 0:
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.8/dist
Requirement already satisfied: numpy>=1.16.0 in /usr/local/lib/python3.8/di
Collecting wurlitzer>=2.0.0
  Downloading wurlitzer-3.0.3-py3-none-any.whl (7.3 kB)
Installing collected packages: wurlitzer, wasserstein, energyflow
Successfully installed energyflow-1.3.2 wasserstein-1.1.0 wurlitzer-3.0.3
```

```
!pip install pyg_lib torch_scatter torch_sparse -f https://data.pyg.org/whl/torc
!pip install torch-geometric
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/cola
Looking in links: https://data.pyg.org/whl/torch-1.13.1+cu116.html
Collecting pyg_lib
  Downloading https://data.pyg.org/whl/torch-1.13.0%2Bcu116/pyg_lib-0.1.0%2
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 22.9 MB/s eta 0:00
Collecting torch_scatter
  Downloading https://data.pyg.org/whl/torch-1.13.0%2Bcu116/torch_scatter-2
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.4/9.4 MB 50.0 MB/s eta 0:00
Collecting torch_sparse
  Downloading https://data.pyg.org/whl/torch-1.13.0%2Bcu116/torch_sparse-0.
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.5/4.5 MB 43.6 MB/s eta 0:00
Requirement already satisfied: scipy in /usr/local/lib/python3.8/dist-packa
Requirement already satisfied: numpy<1.27.0,>=1.19.5 in /usr/local/lib/pyth
Installing collected packages: torch_scatter, pyg_lib, torch_sparse
Successfully installed pyg_lib-0.1.0+pt113cu116 torch_scatter-2.1.0+pt113cu
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/cola
Collecting torch-geometric
  Downloading torch_geometric-2.2.0.tar.gz (564 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 565.0/565.0 KB 11.6 MB/s eta 0:
```

```
  Preparing metadata (setup.py) ... done
Requirement already satisfied: tqdm in /usr/local/lib/python3.8/dist-packag
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packa
Requirement already satisfied: scipy in /usr/local/lib/python3.8/dist-packa
Requirement already satisfied: jinja2 in /usr/local/lib/python3.8/dist-pack
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-pa
Requirement already satisfied: pyparsing in /usr/local/lib/python3.8/dist-p
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.8/dis
Collecting psutil>=5.8.0
  Downloading psutil-5.9.4-cp36-abi3-manylinux_2_12_x86_64.manylinux2010_x8
                                                 280.2/280.2 KB 24.6 MB/s eta 0:
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.8/
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/pyth
Requirement already satisfied: chardet<5,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dis
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/pytho
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.8/di
Building wheels for collected packages: torch-geometric
  Building wheel for torch-geometric (setup.py) ... done
  Created wheel for torch-geometric: filename=torch_geometric-2.2.0-py3-non
  Stored in directory: /root/.cache/pip/wheels/59/a3/20/198928106d3169865ae
Successfully built torch-geometric
Installing collected packages: psutil, torch-geometric
  Attempting uninstall: psutil
    Found existing installation: psutil 5.4.8
    Uninstalling psutil-5.4.8:
      Successfully uninstalled psutil-5.4.8
Successfully installed psutil-5.9.4 torch-geometric-2.2.0
WARNING: The following packages were previously imported in this runtime:
  [psutil]
You must restart the runtime in order to use newly installed versions.
```

RESTART RUNTIME

Importing Libraries

```
import energyflow
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

from sklearn.model_selection import train_test_split
from sklearn.metrics import auc, f1_score, accuracy_score, roc_curve
import torch
from torch_geometric.data import Data, Dataset
from torch_geometric.loader import DataLoader
from torch_geometric.nn import GCNConv, Linear, global_mean_pool, GraphConv, glo
import torch.nn.functional as F
```

Extracting Data

Downloading data using energyflow library

```
data = energyflow.qg_jets.load(num_data=100000, pad=True, ncol=4, generator='pyt
                        with_bc=False, cache_dir='~/.energyflow')
```

```
    Downloading QG_jets.npz from https://www.dropbox.com/s/fclsl7pukcpobsb/QG_j
```

```
# Tuple unpacking
X, y = data
```

Getting the shape of the dataset. X is a matrix of size (N, M, 4) where,

1. N is the number of jets
2. M is the maximum particles in a jet
3. 4 are the features of each particles denoting (pt, eta, phi, pid) values.

```
print(f'X Shape : {X.shape}')
```

```
    X Shape : (100000, 139, 4)
```

```
# Checking Distribution of Labels
print(f'Division of labels : {np.unique(y, return_counts = True)}')
```

```
    Division of labels : (array([0., 1.]), array([50000, 50000]))
```

Constructing Graphs

The graphs for ech jet is constructed in the following way:

1. First for each jet we remove all the particles having all features values were zero i.e., remove paddings.
2. Next we calculate the pair-wise euclidean distance between the nodes using this metric,

$$R = \sqrt{\Delta\eta^2 + \Delta\phi^2}$$

I have set a threshold for R above which we do not consider an edge between the nodes. The chosen value after experimenting is **0.05**.

3. Next, edge-index are formed and the reverse edges are also concatenated.
4. R values are set as the edge-weights and all the 4 feature are set as the node-features.

```
class JetGraphDataset(Dataset):

    def __init__(self, jet_graphs, jet_labels, split = 'train', seed = 1, split_

        jet_graphs = torch.from_numpy(jet_graphs).float()
        jet_labels = torch.from_numpy(jet_labels).long()

        self.edge_threshold = edge_threshold

        # Splitting the data into train-val-test graphs
        X_train, X_rem, Y_train, Y_rem = train_test_split(jet_graphs, jet_labels
        X_val, X_test, Y_val, Y_test = train_test_split(X_rem, Y_rem, train_size

        if split == 'train':
          self.graphs = X_train
          self.labels = Y_train

        elif split == 'test':
          self.graphs = X_test
          self.labels = Y_test

        else:
          self.graphs = X_val
          self.labels = Y_val

    def __len__(self):
```

```
        return len(self.labels)

    def __getitem__(self, idx):

        jet_graph = self.construct_jet_graph(self.graphs[idx])
        jet_graph.y = self.labels[idx]

        #print('final : ', jet_graph.x.shape, jet_graph.edge_index.shape, jet_gr

        return jet_graph

    def construct_jet_graph(self, jet_matrix):

        # Discard rows with all values as 0
        jet_matrix = jet_matrix[~(jet_matrix == 0).all(1)]

        # Calculate pairwise euclidean distances between rows
        euclidean_distances = torch.cdist(jet_matrix[:, 1:3], jet_matrix[:, 1:3]

        # Create edge indices based on the distance threshold
        edge_indices = torch.nonzero(euclidean_distances <= self.edge_threshold)
        #print('indices', edge_indices.shape)
        edge_indices = edge_indices.t().contiguous()
        edge_indices = torch.cat((edge_indices, edge_indices[[1, 0]]), dim=1)

        # Create node features and edge features
        x = jet_matrix[:, [0, 1, 2, 3]]
        edge_attr = euclidean_distances[edge_indices[0], edge_indices[1]].unsque

        # Create PyTorch Geometric Data object
        data = Data(x=x, edge_index=edge_indices, edge_attr=edge_attr)

        return data


# Creating the train-val-test datasets

train_dataset = JetGraphDataset(X, y, split = 'train', edge_threshold = 0.05)
val_dataset = JetGraphDataset(X, y, split = 'val', edge_threshold = 0.05)
test_dataset = JetGraphDataset(X, y, split = 'test', edge_threshold = 0.05)
```

DataLoaders

```
def get_data_loaders(train_dataset, val_dataset, test_dataset, batch_size=32):

    """
    Function to create the DataLoaders for train-val-test data.
    Can specify batch size. Default value is set to 32.
    """

    # Shuffle=True for training data to get diversity in batches at each trainin
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, val_loader, test_loader

train_loader, val_loader, test_loader = get_data_loaders(train_dataset, val_data
```

Set Device

```
def get_device():
    return torch.device('cuda' if torch.cuda.is_available() else 'cpu')


device·=·get_device()
print(device)
```

```
cpu
```

Model

Model Architecture

I haved used 2 GNN architectures

Architecture 1

1. 3-layer GCN network with relu for aggregation of node-level features.
2. Readput layer as global mean pooling for graph-level embedding.
3. A dropout layer followed by linear layer.

```
class GCN(torch.nn.Module):
    def __init__(self, node_features = 2, hidden_channels = 16, num_classes = 2)

        super(GCN, self).__init__()
        torch.manual_seed(1)

        self.conv1 = GCNConv(node_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, hidden_channels)

        self.lin = torch.nn.Linear(hidden_channels, num_classes)

    def forward(self, data):

        x, edge_index, edge_attr, batch = data.x, data.edge_index, data.edge_att

        x = self.conv1(x, edge_index, edge_attr)
        x = F.relu(x)
        x = self.conv2(x, edge_index, edge_attr)
        x = F.relu(x)
        x = self.conv3(x, edge_index, edge_attr)

        x = global_mean_pool(x, batch)

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

        return x
```

Architecture 2

1. Use of GraphConv layer in-place of GCN layer. It adds skip connections in the network tp preserve central node information and omits neighborhood normalization completely.
2. The same readout layer with gloab_mean_pooling is used. I also tried using a combination of global_mean and global_max pool but it lead to decrease in performance.
3. This is followed by an additional linear layer with relu. Then a dropout and final linear layer.

```
class GraphNN(torch.nn.Module):
    def __init__(self, node_features = 2, hidden_channels = 16, num_classes = 2)

        super(GraphNN, self).__init__()
        torch.manual_seed(1)

        self.conv1 = GraphConv(node_features, hidden_channels)
        self.conv2 = GraphConv(hidden_channels, hidden_channels)
        self.conv3 = GraphConv(hidden_channels, hidden_channels)

        self.lin1 = torch.nn.Linear(hidden_channels, hidden_channels)
        self.lin2 = torch.nn.Linear(hidden_channels, num_classes)


    def forward(self, data):

        x, edge_index, edge_attr, batch = data.x, data.edge_index, data.edge_att

        # Neighbourhood node-feature aggregation
        x = self.conv1(x, edge_index, edge_attr)
        x = F.relu(x)
        x = self.conv2(x, edge_index, edge_attr)
        x = F.relu(x)
        x = self.conv3(x, edge_index, edge_attr)

        #x_max = global_max_pool(x, batch)
        #x_mean = global_mean_pool(x, batch)
        #x = torch.cat([x_max, x_mean], dim=1)

        # Batch-wise node-level aggreation
        x = global_mean_pool(x, batch)  # (batch, features)

        x = F.relu(self.lin1(x))
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin2(x)

        return x


model_GCN = GCN(node_features = 4, hidden_channels = 16, num_classes = 2)
model_GraphNN = GraphNN(node_features = 4, hidden_channels = 16, num_classes = 2

model_GCN = model_GCN.to(device)
model_GraphNN = model_GraphNN.to(device)
```

```
print(model_GCN)
```

```
GCN(
  (conv1): GCNConv(4, 16)
  (conv2): GCNConv(16, 16)
  (conv3): GCNConv(16, 16)
  (lin): Linear(in_features=16, out_features=2, bias=True)
)
```

```
print(model_GraphNN)
```

```
GraphNN(
  (conv1): GraphConv(4, 16)
  (conv2): GraphConv(16, 16)
  (conv3): GraphConv(16, 16)
  (lin1): Linear(in_features=16, out_features=16, bias=True)
  (lin2): Linear(in_features=16, out_features=2, bias=True)
)
```

```
# Defining the optimizer and loss function

optimizer_GCN = torch.optim.Adam(model_GCN.parameters(), lr=0.001)
optimizer_GraphNN = torch.optim.Adam(model_GraphNN.parameters(), lr=0.001)

criterion = torch.nn.CrossEntropyLoss()
```

Defining Train-Val-Test Functions

```
def train(model, device, loader, optimizer, criterion):

    model.train()
    correct = 0
    total_loss = 0
    for data in tqdm(loader):  # Iterate in batches over the training dataset.

        data = data.to(device)

        out = model(data)  # 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.

        pred = out.argmax(dim=1)

        correct += int((pred == data.y).sum())
        total_loss += loss.item()

    print(f'Train Acc: {correct/len(loader.dataset):.3f}, Train Loss: {total_los

    return model, total_loss


def evaluate(model, device, loader):

    model.eval()
    correct = 0

    with torch.no_grad():
        for data in tqdm(loader):

            data = data.to(device)
            out = model(data)

            # Calculation of correctly classified edges
            pred = out.argmax(dim=1)
            correct += (pred == data.y).sum().item()

        print(f'Val Acc : {correct/len(loader.dataset):.3f}\n')
```

Training

```
# Training Loop for Architecture 1
```

```
#·Training·Loop·for·Architecture·1

epochs·=·10

for·epoch·in·range(epochs):

··print(f'Epoch·:·{epoch+1}·\n')

··model_GCN,·loss·=·train(model_GCN,·device,·train_loader,·optimizer_GCN,·criter
··evaluate(model_GCN,·device,·val_loader)
```

```
100%|██████████| 313/313 [00:09<00:00, 34.11it/s]
Val Acc : 0.764

Epoch : 3

100%|██████████| 2500/2500 [01:40<00:00, 24.92it/s]
Train Acc: 0.755, Train Loss: 1288.9206
100%|██████████| 313/313 [00:10<00:00, 30.16it/s]
Val Acc : 0.767

Epoch : 4

100%|██████████| 2500/2500 [01:43<00:00, 24.17it/s]
Train Acc: 0.760, Train Loss: 1278.3362
100%|██████████| 313/313 [00:08<00:00, 35.57it/s]
Val Acc : 0.773

Epoch : 5

100%|██████████| 2500/2500 [01:42<00:00, 24.38it/s]
Train Acc: 0.760, Train Loss: 1272.5268
100%|██████████| 313/313 [00:09<00:00, 34.19it/s]
Val Acc : 0.768

Epoch : 6

100%|██████████| 2500/2500 [01:39<00:00, 25.11it/s]
Train Acc: 0.762, Train Loss: 1267.4016
100%|██████████| 313/313 [00:09<00:00, 34.50it/s]
Val Acc : 0.773

Epoch : 7

100%|██████████| 2500/2500 [01:45<00:00, 23.66it/s]
Train Acc: 0.764, Train Loss: 1266.0319
100%|██████████| 313/313 [00:09<00:00, 34.40it/s]
Val Acc : 0.770

Epoch : 8

100%|██████████| 2500/2500 [02:06<00:00, 19.75it/s]
Train Acc: 0.765, Train Loss: 1261.8620
100%|██████████| 313/313 [00:08<00:00, 35.62it/s]
```

```
100%|██████████| 313/313 [00:08<00:00, 35.62it/s]
Val Acc : 0.771

Epoch : 9

100%|██████████| 2500/2500 [01:50<00:00, 22.65it/s]
Train Acc: 0.767, Train Loss: 1258.8274
100%|██████████| 313/313 [00:09<00:00, 33.47it/s]
Val Acc : 0.771

Epoch : 10

100%|██████████| 2500/2500 [01:47<00:00, 23.23it/s]
Train Acc: 0.767, Train Loss: 1258.1549
100%|██████████| 313/313 [00:09<00:00, 32.50it/s]Val Acc : 0.772
```

```
# Training Loop for Architecture 2

epochs = 10

for epoch in range(epochs):

  print(f'Epoch : {epoch+1} \n')

  model_GraphNN, loss = train(model_GraphNN, device, train_loader, optimizer_Gra
  evaluate(model_GraphNN, device, val_loader)
```

```
100%|██████████| 2500/2500 [01:22<00:00, 30.13it/s]
Train Acc: 0.769, Train Loss: 1262.0039
100%|██████████| 313/313 [00:06<00:00, 48.15it/s]
Val Acc : 0.785

Epoch : 3

100%|██████████| 2500/2500 [01:23<00:00, 29.97it/s]
Train Acc: 0.779, Train Loss: 1232.8763
100%|██████████| 313/313 [00:07<00:00, 39.37it/s]
Val Acc : 0.787

Epoch : 4

100%|██████████| 2500/2500 [01:24<00:00, 29.44it/s]
Train Acc: 0.784, Train Loss: 1215.3112
100%|██████████| 313/313 [00:07<00:00, 39.81it/s]
Val Acc : 0.791

Epoch : 5

100%|██████████| 2500/2500 [01:37<00:00, 25.55it/s]
Train Acc: 0.785, Train Loss: 1210.4889
100%|██████████| 313/313 [00:07<00:00, 39.73it/s]
```

```
Val Acc : 0.786

Epoch : 6

100%|██████████| 2500/2500 [01:28<00:00, 28.14it/s]
Train Acc: 0.786, Train Loss: 1203.9970
100%|██████████| 313/313 [00:07<00:00, 39.99it/s]
Val Acc : 0.791

Epoch : 7

100%|██████████| 2500/2500 [01:25<00:00, 29.19it/s]
Train Acc: 0.786, Train Loss: 1200.0282
100%|██████████| 313/313 [00:07<00:00, 42.83it/s]
Val Acc : 0.788

Epoch : 8

100%|██████████| 2500/2500 [01:27<00:00, 28.73it/s]
Train Acc: 0.788, Train Loss: 1193.1311
100%|██████████| 313/313 [00:07<00:00, 44.28it/s]
Val Acc : 0.788

Epoch : 9

100%|██████████| 2500/2500 [01:25<00:00, 29.10it/s]
Train Acc: 0.788, Train Loss: 1197.1163
100%|██████████| 313/313 [00:06<00:00, 47.42it/s]
Val Acc : 0.790

Epoch : 10

100%|██████████| 2500/2500 [01:28<00:00, 28.31it/s]
Train Acc: 0.789, Train Loss: 1196.4977
100%|██████████| 313/313 [00:07<00:00, 41.29it/s]Val Acc : 0.795
```

Testing

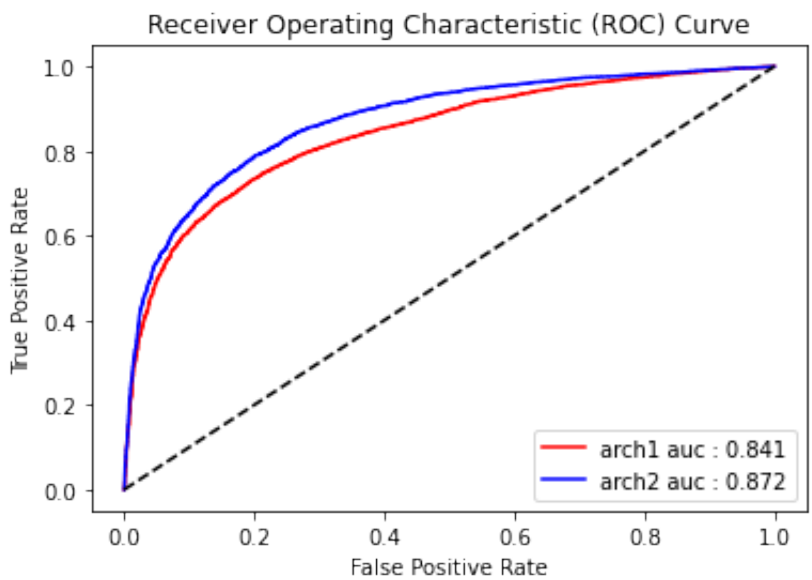
```
def test(model, device, loader):

    model.eval()
    y_true = []
    y_probas = []
    y_pred = []

    with torch.no_grad():

        for data in tqdm(loader):

            data = data.to(device)
            out = model(data)

            y_true += data.y.cpu().numpy().tolist()
            y_pred += out.argmax(dim=1).cpu().numpy().tolist()  # absoulte predi
            y_probas += out[:, 1].cpu().numpy().tolist()  # probability of class

    # Calculating few metrics

    acc = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    fpr, tpr, thresholds = roc_curve(y_true, y_probas)
    roc_auc = auc(fpr, tpr)

    print('\nResults\n')
    print(f'Testing Accuracy {acc:.3f}')
    print(f'F1 score: {f1:.3f}')
    print(f'ROC-AUC: {roc_auc:.3f}\n')

    return acc, f1, fpr, tpr, roc_auc


#·Testing·Architecture·1

acc,·f1,·fpr1,·tpr1,·area1·=·test(model_GCN,·device,·test_loader)
```

```
100%|██████████| 313/313 [00:16<00:00, 18.49it/s]
Results

Testing Accuracy 0.768
F1 score: 0.762
ROC-AUC: 0.841
```

```
# Testing Architecture 2

acc, f1, fpr2, tpr2, area2 = test(model_GraphNN, device, test_loader)
```

```
100%|██████████| 313/313 [00:15<00:00, 19.59it/s]
Results

Testing Accuracy 0.792
F1 score: 0.788
ROC-AUC: 0.872
```

```
plt.plot(fpr1, tpr1, color = 'red', label = f'arch1 auc : {area1:.3f}')
plt.plot(fpr2, tpr2, color = 'blue', label = f'arch2 auc : {area2:.3f}')
plt.plot([0, 1], [0, 1], 'k--')  # diagonal line representing random guessing
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()

plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.savefig(f'roc_auc.png')
plt.show()
```



0s completed at 10:38 PM