Task5

March 25, 2025

1 Task 5: Quantum Graph Neural Network

We will be extending the idea of Task 2 to classify quark/gluon using GNN, but this time we do this using quantum circuits

The preprocessing part will be almost same.

Let us first import all the required libraries

[1]:
```
import os
import glob
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch_geometric.data import InMemoryDataset, Data, DataLoader
from torch_geometric.nn import TransformerConv, global_mean_pool
from torch_cluster import knn_graph
from sklearn.metrics import accuracy_score
from tqdm import tqdm
from torch_geometric.nn import TransformerConv
import pennylane as qml
```

1.1 Load and Preprocess the dataset

As done in task 2, since the data set is very large so we will load the dataset in batches

For preprocessing , this time I will be deleting all the particle data that has all 0 values and rest of the data is normalized

X : a 3D array of shape (num_jets, num_particles, num_features) where features are [pT, , , PDG]. We keep only the first three columns.

y: a 1D array of jet-level labels

We will build a KNN graph

Stores: x: tensor of normalized continuous features (shape: [num_nodes, 3]) edge_index: graph connectivity y: jet-level label

[ ]:
```python
class QGDataset(InMemoryDataset):
    def __init__(self, root, transform=None, pre_transform=None):
        super(QGDataset, self).__init__(root, transform, pre_transform)
        path = self.processed_paths[0]
        if os.path.exists(path):
            self.data, self.slices = torch.load(path)
        else:
            self.process()
            self.data, self.slices = torch.load(path)

    @property
    def raw_file_names(self):
        return glob.glob(os.path.join(self.raw_dir, '*.npz'))

    @property
    def processed_file_names(self):
        return ['data.pt']

    def process(self):
        data_list = []
        batch_size = 5000  # Process in chunks to avoid memory overload
        processed_path = self.processed_paths[0]

        for fpath in tqdm(self.raw_file_names, desc="Processing NPZ Files"):
            npz = np.load(fpath, mmap_mode='r')
            features = npz['X'].astype(np.float32)
            labels = npz['y'].astype(np.int64)
            num_jets, num_particles, num_feats = features.shape

            for i in range(num_jets):
                x_np = features[i]
                mask = ~np.all(x_np == 0, axis=1)
                x_np = x_np[mask]
                if x_np.size == 0:
                    continue
                x_cont_np = x_np[:, :3]
                # Normalize features per jet
                mean = np.mean(x_cont_np, axis=0, keepdims=True)
                std = np.std(x_cont_np, axis=0, keepdims=True) + 1e-6
                x_cont_np = (x_cont_np - mean) / std

                x_cont = torch.tensor(x_cont_np, dtype=torch.float32)
                y = torch.tensor([labels[i]], dtype=torch.long)
                edge_index = knn_graph(x_cont, k=10, loop=False)

                data_obj = Data(x=x_cont, edge_index=edge_index, y=y)
                data_list.append(data_obj)
```

```
                if len(data_list) >= batch_size:
                    self._save_partial(data_list, processed_path)
                    data_list = []
        if data_list:
            self._save_partial(data_list, processed_path)

    def _save_partial(self, data_list, path):
        if os.path.exists(path):
            old_data, old_slices = torch.load(path)
            if isinstance(old_data, Data):
                old_data = [old_data]
            data_list = list(old_data) + data_list
        self.data, self.slices = self.collate(data_list)
        torch.save((self.data, self.slices), path)
```

1.2 Quantum Circuit

We will use 3 qubits to represent pT, ,

`qml.AngleEmbedding`: Encodes input features as quantum angles.

`qml.BasicEntanglerLayers`: Applies trainable entangling layers.

`qml.expval(qml.PauliZ(i))`: Returns quantum measurement outputs.

Wrap the quantum circuit in a PyTorch Layer

[3]:
```
n_qubits = 3
dev = qml.device("default.qubit", wires=n_qubits)

@qml.qnode(dev, interface="torch")
def quantum_circuit(inputs, weights):
    qml.AngleEmbedding(inputs, wires=range(n_qubits))
    qml.BasicEntanglerLayers(weights, wires=range(n_qubits))
    return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]

class QuantumLayer(nn.Module):
    def __init__(self, n_qubits, n_layers):
        super(QuantumLayer, self).__init__()
        weight_shapes = {"weights": (n_layers, n_qubits)}
        self.q_layer = qml.qnn.TorchLayer(quantum_circuit, weight_shapes)

    def forward(self, x):
        return self.q_layer(x)
```

1.3 Hybrid Model

1. Graph Convolution (TransformerConv) Two layers: First layer: TransformerConv with multi-head attention (4 heads). Second layer: TransformerConv with single-head aggregation.

These layers extract graph-based features from the jet data.

2. Global Pooling Applies `global_mean_pool()` to aggregate node-level features into jet-level representations.
3. Quantum Feature Projection Projects features to a 3-dimensional space (matching the quantum circuit). Passes the features through the QuantumLayer.
4. Final Classification Layer A linear layer maps quantum outputs to class logits.

[6]:
```
class HybridTransformerQuantumModel(nn.Module):
    def __init__(self, in_channels=3, hidden_dim=64, out_dim=2, n_layers=3,␣
↪n_qubits=3):
        super(HybridTransformerQuantumModel, self).__init__()
        # Layer 1
        self.conv1 = TransformerConv(in_channels, hidden_dim, heads=4,␣
↪dropout=0.1)
        # Layer 2
        self.conv2 = TransformerConv(hidden_dim * 4, hidden_dim, heads=1,␣
↪dropout=0.1)
        # Projection layer to make output equal to qubits
        self.feature_proj = nn.Linear(hidden_dim, n_qubits)
        self.q_layer = QuantumLayer(n_qubits, n_layers)
        self.lin = nn.Linear(n_qubits, out_dim)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index).relu()    # Output shape: (num_nodes,␣
↪hidden_dim*4)
        x = self.conv2(x, edge_index).relu()    # Output shape: (num_nodes,␣
↪hidden_dim)
        x = global_mean_pool(x, batch)          # Graph-level pooling -> shape:␣
↪(num_graphs, hidden_dim)
        x = self.feature_proj(x)                # Project to (num_graphs,␣
↪n_qubits=3)
        x = self.q_layer(x)                     # Quantum layer -> shape:␣
↪(num_graphs, n_qubits)
        x = self.lin(x)                         # Final classification layer
        return x
```

1.4 Train and Test Functions

Same as Task 2

[7]:
```
def train(model, loader, optimizer, device):
    model.train()
    total_loss = 0
    for data in tqdm(loader, desc="Training"):
        data = data.to(device)
        optimizer.zero_grad()
```

```
        out = model(data.x, data.edge_index, data.batch)
        loss = F.cross_entropy(out, data.y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * data.num_graphs
    return total_loss / len(loader.dataset)

def test(model, loader, device):
    model.eval()
    y_true, y_pred = [], []
    for data in loader:
        data = data.to(device)
        with torch.no_grad():
            out = model(data.x, data.edge_index, data.batch)
            pred = out.argmax(dim=1)
            y_true.extend(data.y.cpu().numpy())
            y_pred.extend(pred.cpu().numpy())
    return accuracy_score(y_true, y_pred)
```

1.5 Main fxn

[ ]:
```
def main():
    root_dir = 'qg_data'
    dataset = QGDataset(root=root_dir)
    print(f"Processed {len(dataset)} jets.")

    train_size = int(0.8 * len(dataset))
    train_dataset = dataset[:train_size]
    test_dataset = dataset[train_size:]

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    test_loader  = DataLoader(test_dataset, batch_size=32, shuffle=False)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = QGNN(in_channels=3, hidden_dim=128, out_dim=2, n_layers=3,␣
 ↪n_qubits=3).to(device)
    optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)

    best_acc = 0
    for epoch in range(1, 51):
        loss = train(model, train_loader, optimizer, device)
        acc = test(model, test_loader, device)
        print(f"Epoch {epoch:02d}, Loss: {loss:.4f}, Test Accuracy: {acc:.4f}")
        if acc > best_acc:
            best_acc = acc
            torch.save(model.state_dict(), "qgnn_improved_best.pth")
```

```
    print("Best Test Accuracy:", best_acc)

if __name__ == "__main__":
    main()
```

```
C:\Users\arnav\AppData\Local\Temp\ipykernel_22224\1247508449.py:66:
FutureWarning: You are using `torch.load` with `weights_only=False` (the current
default value), which uses the default pickle module implicitly. It is possible
to construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  self.data, self.slices = torch.load(path)
C:\Users\arnav\anaconda3\envs\qml\lib\site-
packages\torch_geometric\deprecation.py:26: UserWarning: 'data.DataLoader' is
deprecated, use 'loader.DataLoader' instead
  warnings.warn(out)

Processed 5000 jets.

Training: 100%|      | 125/125 [00:10<00:00, 11.94it/s]

Epoch 01, Loss: 0.6887, Test Accuracy: 0.6750

Training: 100%|      | 125/125 [00:09<00:00, 12.83it/s]

Epoch 02, Loss: 0.6152, Test Accuracy: 0.6940

Training: 100%|      | 125/125 [00:09<00:00, 13.09it/s]

Epoch 03, Loss: 0.6001, Test Accuracy: 0.6910

Training: 100%|      | 125/125 [00:09<00:00, 12.77it/s]

Epoch 04, Loss: 0.5925, Test Accuracy: 0.7070

Training: 100%|      | 125/125 [00:09<00:00, 12.79it/s]

Epoch 05, Loss: 0.5853, Test Accuracy: 0.7180

Training: 100%|      | 125/125 [00:10<00:00, 12.29it/s]

Epoch 06, Loss: 0.5811, Test Accuracy: 0.7110

Training: 100%|      | 125/125 [00:09<00:00, 12.95it/s]

Epoch 07, Loss: 0.5787, Test Accuracy: 0.7270

Training: 100%|      | 125/125 [00:09<00:00, 12.86it/s]
```

```
Epoch 08, Loss: 0.5737, Test Accuracy: 0.7240
Training: 100%|      | 125/125 [00:10<00:00, 12.40it/s]
Epoch 09, Loss: 0.5699, Test Accuracy: 0.7140
Training: 100%|      | 125/125 [00:10<00:00, 12.32it/s]
Epoch 10, Loss: 0.5672, Test Accuracy: 0.7240
Training: 100%|      | 125/125 [00:09<00:00, 12.95it/s]
Epoch 11, Loss: 0.5664, Test Accuracy: 0.7170
Training: 100%|      | 125/125 [00:09<00:00, 13.21it/s]
Epoch 12, Loss: 0.5673, Test Accuracy: 0.7060
Training: 100%|      | 125/125 [00:09<00:00, 13.20it/s]
Epoch 13, Loss: 0.5665, Test Accuracy: 0.7000
Training: 100%|      | 125/125 [00:10<00:00, 12.49it/s]
Epoch 14, Loss: 0.5878, Test Accuracy: 0.7210
Training: 100%|      | 125/125 [00:09<00:00, 12.70it/s]
Epoch 15, Loss: 0.5630, Test Accuracy: 0.7220
Training: 100%|      | 125/125 [00:10<00:00, 12.35it/s]
Epoch 16, Loss: 0.5715, Test Accuracy: 0.7180
Training: 100%|      | 125/125 [00:09<00:00, 12.53it/s]
Epoch 17, Loss: 0.5619, Test Accuracy: 0.7310
Training: 100%|      | 125/125 [00:10<00:00, 12.39it/s]
Epoch 18, Loss: 0.5619, Test Accuracy: 0.7140
Training: 100%|      | 125/125 [00:09<00:00, 12.72it/s]
Epoch 19, Loss: 0.5572, Test Accuracy: 0.7280
Training: 100%|      | 125/125 [00:10<00:00, 12.25it/s]
Epoch 20, Loss: 0.5584, Test Accuracy: 0.7270
Training: 100%|      | 125/125 [00:09<00:00, 12.90it/s]
Epoch 21, Loss: 0.5606, Test Accuracy: 0.7280
Training: 100%|      | 125/125 [00:09<00:00, 13.10it/s]
Epoch 22, Loss: 0.5604, Test Accuracy: 0.7240
Training: 100%|      | 125/125 [00:09<00:00, 13.19it/s]
Epoch 23, Loss: 0.5630, Test Accuracy: 0.7210
Training: 100%|      | 125/125 [00:09<00:00, 12.62it/s]
```

```
Epoch 24, Loss: 0.5602, Test Accuracy: 0.7130
Training: 100%|      | 125/125 [00:10<00:00, 12.23it/s]
Epoch 25, Loss: 0.5569, Test Accuracy: 0.7300
Training: 100%|      | 125/125 [00:10<00:00, 12.09it/s]
Epoch 26, Loss: 0.5574, Test Accuracy: 0.7130
Training: 100%|      | 125/125 [00:09<00:00, 12.85it/s]
Epoch 27, Loss: 0.5540, Test Accuracy: 0.7250
Training: 100%|      | 125/125 [00:09<00:00, 12.76it/s]
Epoch 28, Loss: 0.5571, Test Accuracy: 0.7320
Training: 100%|      | 125/125 [00:09<00:00, 12.64it/s]
Epoch 29, Loss: 0.5619, Test Accuracy: 0.7270
Training: 100%|      | 125/125 [00:09<00:00, 12.95it/s]
Epoch 30, Loss: 0.5574, Test Accuracy: 0.7340
Training: 100%|      | 125/125 [00:09<00:00, 13.35it/s]
Epoch 31, Loss: 0.5667, Test Accuracy: 0.7260
Training: 100%|      | 125/125 [00:09<00:00, 12.71it/s]
Epoch 32, Loss: 0.5892, Test Accuracy: 0.6680
Training: 100%|      | 125/125 [00:09<00:00, 12.65it/s]
Epoch 33, Loss: 0.5902, Test Accuracy: 0.6990
Training: 100%|      | 125/125 [00:10<00:00, 12.38it/s]
Epoch 34, Loss: 0.5727, Test Accuracy: 0.7090
Training: 100%|      | 125/125 [00:09<00:00, 12.56it/s]
Epoch 35, Loss: 0.5787, Test Accuracy: 0.6890
Training: 100%|      | 125/125 [00:09<00:00, 12.58it/s]
Epoch 36, Loss: 0.5541, Test Accuracy: 0.7220
Training: 100%|      | 125/125 [00:09<00:00, 12.93it/s]
Epoch 37, Loss: 0.5494, Test Accuracy: 0.7240
Training: 100%|      | 125/125 [00:10<00:00, 12.36it/s]
Epoch 38, Loss: 0.5536, Test Accuracy: 0.7120
Training: 100%|      | 125/125 [00:09<00:00, 12.77it/s]
Epoch 39, Loss: 0.5519, Test Accuracy: 0.7210
Training: 100%|      | 125/125 [00:10<00:00, 12.16it/s]
```

```
Epoch 40, Loss: 0.5585, Test Accuracy: 0.7300
Training: 100%|      | 125/125 [00:09<00:00, 12.66it/s]
Epoch 41, Loss: 0.5440, Test Accuracy: 0.7310
Training: 100%|      | 125/125 [00:10<00:00, 12.46it/s]
Epoch 42, Loss: 0.5502, Test Accuracy: 0.7270
Training: 100%|      | 125/125 [00:09<00:00, 12.54it/s]
Epoch 43, Loss: 0.5589, Test Accuracy: 0.7280
Training: 100%|      | 125/125 [00:09<00:00, 13.01it/s]
Epoch 44, Loss: 0.5492, Test Accuracy: 0.7280
Training: 100%|      | 125/125 [00:10<00:00, 12.46it/s]
Epoch 45, Loss: 0.5544, Test Accuracy: 0.7310
Training: 100%|      | 125/125 [00:09<00:00, 12.56it/s]
Epoch 46, Loss: 0.5506, Test Accuracy: 0.7270
Training: 100%|      | 125/125 [00:09<00:00, 13.18it/s]
Epoch 47, Loss: 0.5495, Test Accuracy: 0.7270
Training: 100%|      | 125/125 [00:09<00:00, 13.13it/s]
Epoch 48, Loss: 0.5494, Test Accuracy: 0.7300
Training: 100%|      | 125/125 [00:09<00:00, 12.55it/s]
Epoch 49, Loss: 0.5497, Test Accuracy: 0.7250
Training: 100%|      | 125/125 [00:09<00:00, 13.02it/s]
Epoch 50, Loss: 0.5443, Test Accuracy: 0.7310
Best Test Accuracy: 0.734
```

Due to memory issue I cannot increase its accuracy as by increasing any parameter or fine tuning leads to Out of Memory Error

This model gives accuracy of 73.4% which is less than the models we have done in task 2 because of the issues in Computational Power, maybe if I had more computational power, the quantum model would have performed better.