Task2

March 25, 2025

1 Task 2:Classical GNN

Graph Neural Network are used to handle the data in the form of vertices and edges unline CNN,RNN etc. It helps to establish relationship between entities

We will first start by importing all the required libraries and will be using **pytorch** for implementation

[1]:
```
import os
import glob
import numpy as np
import torch
import torch.nn.functional as F

from torch_geometric.data import InMemoryDataset, Data
from torch_geometric.loader import DataLoader
from torch_geometric.nn import GCNConv, global_mean_pool
from torch_cluster import knn_graph
from sklearn.model_selection import train_test_split
from tqdm import tqdm
from sklearn.metrics import accuracy_score
import torch.optim as optim
from torch_geometric.nn import GCNConv, GATConv, GINConv, BatchNorm,␣
 ↪global_mean_pool
from torch.nn import BatchNorm1d
```

1.1 Data Preprocessing

We will create a class for QGDataset.

The dataset is in .npz files. Since there are many files we will join all the files and then store the processed data in data.pt

The dataset has 2 labels X and Y ### X field X contain a 2D matrix that is collection of particles and each particles has 4 field

X.shape= (number of jets,number of particle,features)

The number of jets are the total jets in the data set

The number of particles is total particles in each jet

The features =4 ie pT, eta, phi and PDG ID

1.1.1 Y field

Y.shape= (num of jets)

It is a 1D matrix that contain only 0 and 1 for Gluon and Quark

For X we are using float32 as datatype because of memory constraint on the system

We will be using only first 3 features of X ie pT, phi and eta and we will not use PDGID beacuse it is just the id of particle and it does no define any spatial behaviour

1.1.2 Graph

We will be using KNN to create a graph

Node/Vertices= Particle in a Jet

Edges= Relationship between particles based on pT, phi, eta

```
[ ]: 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):
             # All .npz files in the raw directory
             return glob.glob(os.path.join(self.raw_dir, '*.npz'))

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

         def process(self):
             data_list = []
             for fpath in tqdm(self.raw_file_names, desc="Processing .npz Files"):
                 npz = np.load(fpath, mmap_mode='r')
                 # Use the keys 'X' and 'y' as found in your files
                 features = npz['X'].astype(np.float32)  # shape: (num_jets,␣
     ↪num_particles, num_features)
                 labels = npz['y'].astype(np.int64)      # shape: (num_jets,)
                 num_jets, num_particles, num_feats = features.shape

                 for i in range(num_jets):
```

```
                x_np = features[i]  # shape: (num_particles, num_feats)
                mean = np.mean(x_np, axis=0, keepdims=True)
                std = np.std(x_np, axis=0, keepdims=True) + 1e-6
                x_np_norm = (x_np - mean) / std

                x = torch.tensor(x_np_norm, dtype=torch.float32)
                y = torch.tensor([labels[i]], dtype=torch.long)

                # Build graph: use first three features (pT,  ,  ) for KNN
                coords = x[:, :3]
                edge_index = knn_graph(coords, k=10, loop=False)
                data_obj = Data(x=x, edge_index=edge_index, y=y)
                data_list.append(data_obj)

        self.data, self.slices = self.collate(data_list)
        torch.save((self.data, self.slices), self.processed_paths[0])
```

1.2 First Model : Simple GCN

1.2.1 1. Model Components

def **init**(self, in_channels, hidden_dim=64, out_dim=2):
The `in_channels` is the number of input features per node = 3 (`pT,  ,  `).
`hidden_dim=64`: Hidden layer size.
`out_dim = 2`: We have only 2 outputs (0 for Gluon, 1 for Quark).

1.2.2 2. Layers in the Model

self.conv1 = GCNConv(in_channels, hidden_dim)
Applies the first Graph Convolutional Layer, which extracts meaningful features from input nodes.

self.conv2 = GCNConv(hidden_dim, hidden_dim)
A second Graph Convolutional Layer for refining learned representations.

self.lin = torch.nn.Linear(hidden_dim, out_dim)
A fully connected layer that converts the graph representation into logits for classification.

1.2.3 3. Forward Pass

def forward(self, x, edge_index, batch):
Defines how input data passes through the model.

1st GCN Layer- Non-Linearity (ReLU) x = self.conv1(x, edge_index)
Applies the first graph convolution (message passing).
x = F.relu(x)
Uses ReLU activation to introduce non-linearity.

2nd GCN Layer - Non-Linearity (ReLU) x = self.conv2(x, edge_index)
Refines the node features based on graph connections.
x = F.relu(x)
Applies another non-linearity.

Global Pooling x = global_mean_pool(x, batch)
Aggregates all node features to form a single graph-level feature vector.

Fully Connected Output x = self.lin(x)
Final classification layer for binary output (Quark vs Gluon).
return x
Returns logits for classification.

---

[3]:
```
class SimpleGCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_dim=64, out_dim=2):
        super(SimpleGCN, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.lin = torch.nn.Linear(hidden_dim, out_dim)

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

1.3 Second Model: GCNModel

1.3.1 1. Model Components

def **init**(self, in_channels, hidden_dim=256, out_dim=2, dropout=0.3):
- `in_channels`: Number of input features per node = 3 (`pT, , `).
- `hidden_dim=256`: Size of the hidden layers for feature extraction.
- `out_dim=2`: We have only 2 output classes (0 for Gluon, 1 for Quark).
- `dropout=0.3`: Probability of randomly dropping neurons to prevent overfitting.

1.3.2 2. Layers in the Model

self.conv1 = GCNConv(in_channels, hidden_dim)
First Graph Convolutional Layer to process input node features.

self.bn1 = BatchNorm(hidden_dim)
Batch Normalization to stabilize training and speed up convergence.

self.conv2 = GCNConv(hidden_dim, hidden_dim)
Second Graph Convolutional Layer for deeper feature learning.

self.bn2 = BatchNorm(hidden_dim)
Another Batch Normalization for improved stability.

self.dropout = dropout
Dropout to prevent overfitting during training.

self.lin = torch.nn.Linear(hidden_dim, out_dim)
Fully connected layer for final classification.

1.3.3 3. Forward Pass

def forward(self, x, edge_index, batch):
Defines how data moves through the model.

1st GCN Layer → BatchNorm → ReLU → Dropout x = self.conv1(x, edge_index)
Applies the first graph convolution layer.

x = self.bn1(x)
Normalizes feature distribution.

x = F.relu(x)
Applies ReLU activation for non-linearity.

x = F.dropout(x, p=self.dropout, training=self.training)
Randomly drops some neurons for regularization.

2nd GCN Layer → BatchNorm → ReLU x = self.conv2(x, edge_index)
Extracts higher-level features from the graph.

x = self.bn2(x)
Batch normalization to maintain stable distributions.

x = F.relu(x)
Applies another ReLU activation.

Global Pooling x = global_mean_pool(x, batch)
Aggregates features across all nodes in the graph.

Fully Connected Output x = self.lin(x)
Final classification layer for predicting Quark (1) or Gluon (0).

return x
Returns the logits for classification.

---

[4]:
```
class GCNModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_dim=256, out_dim=2, dropout=0.3):
        super(GCNModel, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_dim)
        self.bn1 = BatchNorm1d(hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.bn2 = BatchNorm1d(hidden_dim)
        self.dropout = dropout
        self.lin = torch.nn.Linear(hidden_dim, out_dim)

    def forward(self, x, edge_index, batch):
        x = F.relu(self.bn1(self.conv1(x, edge_index)))
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = F.relu(self.bn2(self.conv2(x, edge_index)))
        x = global_mean_pool(x, batch)
        return self.lin(x)
```

1.4 Third Model : GATModel

1.4.1 1. Model Components

def **init**(self, in_channels, hidden_dim=256, out_dim=2, heads=4, dropout=0.3):
- `in_channels`: Number of input features per node = 3 (`pT,  ,  `).
- `hidden_dim=256`: Size of the hidden layers for feature extraction.
- `out_dim=2`: We have only 2 output classes (0 for Gluon, 1 for Quark).
- `heads=4`: Number of attention heads in the first GAT layer.
- `dropout=0.3`: Dropout rate to prevent overfitting.

1.4.2 2. Layers in the Model

self.conv1 = GATConv(in_channels, hidden_dim, heads=heads, concat=True)
First Graph Attention Layer (GAT) with multi-head attention (concatenated output).

self.bn1 = BatchNorm(hidden_dim * heads)
Batch Normalization to stabilize training and speed up convergence.

self.conv2 = GATConv(hidden_dim * heads, hidden_dim, heads=1, concat=False)
Second GAT Layer (single-head) to refine learned features.

self.bn2 = BatchNorm(hidden_dim)
Another Batch Normalization for improved stability.

self.dropout = dropout
Dropout to prevent overfitting during training.

self.lin = torch.nn.Linear(hidden_dim, out_dim)
Fully connected layer for final classification.

1.4.3 3. Forward Pass

def forward(self, x, edge_index, batch):
Defines how data moves through the model.

1st GAT Layer → BatchNorm → ELU → Dropout x = self.conv1(x, edge_index)
Applies the first graph attention convolution.

x = self.bn1(x)
Normalizes feature distribution.

x = F.elu(x)
Applies ELU activation for smooth feature transformation.

x = F.dropout(x, p=self.dropout, training=self.training)
Randomly drops some neurons for regularization.

2nd GAT Layer → BatchNorm → ELU x = self.conv2(x, edge_index)
Extracts refined features from the graph.

x = self.bn2(x)
Batch normalization to maintain stable distributions.

x = F.elu(x)
Applies another ELU activation.

Global Pooling x = global_mean_pool(x, batch)
Aggregates features across all nodes in the graph.

Fully Connected Output x = self.lin(x)
Final classification layer for predicting Quark (1) or Gluon (0).

return x
Returns the logits for classification.

---

[5]:
```
class GATModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_dim=256, out_dim=2, heads=4,␣
 ↪dropout=0.3):
        super(GATModel, self).__init__()
        self.conv1 = GATConv(in_channels, hidden_dim, heads=heads, concat=True)
        self.bn1 = BatchNorm1d(hidden_dim * heads)
        self.conv2 = GATConv(hidden_dim * heads, hidden_dim, heads=1,␣
 ↪concat=False)
        self.bn2 = BatchNorm1d(hidden_dim)
        self.dropout = dropout
        self.lin = torch.nn.Linear(hidden_dim, out_dim)

    def forward(self, x, edge_index, batch):
        x = F.elu(self.bn1(self.conv1(x, edge_index)))
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = F.elu(self.bn2(self.conv2(x, edge_index)))
        x = global_mean_pool(x, batch)
        return self.lin(x)
```

2 Fourth Model: Residual GCN

2.0.1 1. Model Components

def **init**(self, in_channels, hidden_dim=256, out_dim=2, dropout=0.3): - **in_channels**: Number of input features per node (e.g., 3 for pT, ,).
- **hidden_dim = 256**: Size of the hidden layers used for feature extraction.
- **out_dim = 2**: Number of output classes (e.g., 0 for Gluon, 1 for Quark).
- **dropout = 0.3**: Dropout rate to help prevent overfitting.

2.0.2 2. Layers in the Model

- **self.conv1 = GCNConv(in_channels, hidden_dim)**
  The first graph convolutional layer, which transforms input features into a hidden representation.
- **self.bn1 = BatchNorm1d(hidden_dim)**
  Batch normalization for the output of the first GCN layer to stabilize and speed up training.
- **self.conv2 = GCNConv(hidden_dim, hidden_dim)**
  The second GCN layer that further processes the features.
- **self.bn2 = BatchNorm1d(hidden_dim)**
  Batch normalization applied after the second GCN layer.
- **self.conv3 = GCNConv(hidden_dim, hidden_dim)**
  The third GCN layer which extracts deeper features from the graph.
- **self.bn3 = BatchNorm1d(hidden_dim)**
  Batch normalization applied to the output of the third GCN layer.
- **self.lin = torch.nn.Linear(hidden_dim, out_dim)**
  A fully connected layer that maps the final hidden representation to the output classes.

2.0.3 3. Forward Pass

def forward(self, x, edge_index, batch):

1. **First GCN Layer:**
   x1 = F.relu(self.bn1(self.conv1(x, edge_index)))
   - Applies the first GCN layer, followed by batch normalization and ReLU activation.
2. **Second GCN Layer:**
   x2 = F.relu(self.bn2(self.conv2(x1, edge_index)))
   - Applies the second GCN layer on the output of the first layer, followed by batch normalization and ReLU activation.
3. **Third GCN Layer with Residual Connection:**
   x3 = F.relu(self.bn3(self.conv3(x2, edge_index)) + x1)
   - The output of the third GCN layer is added to the output of the first layer (residual connection) to help preserve initial features, then batch normalization and ReLU activation are applied.

4. **Global Pooling:**
   x = global_mean_pool(x3, batch)
   - Aggregates node features into a single graph-level representation by taking the mean of all node features.
5. **Final Classification:**
   x = self.lin(x)
   - The pooled features are passed through a linear layer to produce the final logits for classification.
6. **Return:**
   return x
   - Returns the output logits.

---

[6]:
```
class ResidualGCNModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_dim=256, out_dim=2, dropout=0.3):
        super(ResidualGCNModel, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_dim)
        self.bn1 = BatchNorm1d(hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.bn2 = BatchNorm1d(hidden_dim)
        self.conv3 = GCNConv(hidden_dim, hidden_dim)
        self.bn3 = BatchNorm1d(hidden_dim)
        self.dropout = dropout
        self.lin = torch.nn.Linear(hidden_dim, out_dim)

    def forward(self, x, edge_index, batch):
        x1 = F.relu(self.bn1(self.conv1(x, edge_index)))
        x2 = F.relu(self.bn2(self.conv2(x1, edge_index)))
        # Residual connection: add output of first layer (x1) to third layer's␣
↪output
        x3 = F.relu(self.bn3(self.conv3(x2, edge_index)) + x1)
        x = global_mean_pool(x3, batch)
        x = self.lin(x)
        return x
```

2.1 Training

2.1.1 1. Cross-Entropy Loss

(`F.cross_entropy(out, data.y)`) It measures how bad model predictions are.

- The model gives a **score** for each class (e.g., **quark** or **gluon**), but these scores might not be correct.

- The **cross-entropy loss** tells us **how far off** the model's predictions are from the actual answers.

- The goal is to **reduce this loss** so the model makes better predictions over time.

Example:
If the actual label is **1 (quark)** and the model predicts [**0.1, 0.9**], meaning it's **90% sure it's a quark**, the loss is **small** (good!).
If the model predicts [**0.9, 0.1**], meaning it's **90% sure it's a gluon**, the loss is **large** (bad!).

2.1.2 2. Optimizer

(`optimizer.step()`) - It helps to change parameters after mistakes - It looks at the **cross-entropy loss** and decides how to **adjust the model's settings (weights)** to improve its predictions.
- Popular optimizers:
- **SGD (Slow Learner)** – Adjusts weights little by little.
- **Adam (Smart Learner)** – Learns **faster** by adjusting weights differently for each part of the model.

2.1.3 3. `loss.backward()`

- This tells the model **which parts of itself are responsible for the mistake** (computing gradients).
- It helps figure out which **weights** should be changed and by how much.

2.1.4 4. `model.train()`

- **`model.train()`** – The model is **actively learning**. It allows techniques like **dropout** (randomly ignoring some neurons) to **prevent overfitting**. ### 5. `model.eval()`
- **`model.eval()`** – The model is in **testing mode**, meaning no tricks like dropout—it just makes predictions **as accurately as possible**.

2.2 How It All Works Together

1. The model makes a **guess** (prediction).
2. The **cross-entropy loss** checks how wrong the guess is.
3. **Backpropagation (`loss.backward()`)** finds out **what to fix**.
4. The **optimizer (`optimizer.step()`)** updates the model to **reduce errors** next time.
5. Over time, the model **gets better at classifying quarks and gluons!**

[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)
```

[9]:
```
def main():
    root_dir = 'qg_data'
    dataset = QGDataset(root=root_dir)
    train_size = int(0.8 * len(dataset))
    train_dataset, test_dataset = dataset[:train_size], dataset[train_size:]
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
    in_channels = dataset[0].x.shape[1]
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    models = {
        "SimpleGCN": SimpleGCN(in_channels),
        "GCNModel": GCNModel(in_channels),
        "GATModel": GATModel(in_channels),
        "ResidualGCNModel": ResidualGCNModel(in_channels)
    }

    for model_name, model in models.items():
        print(f"\nTraining {model_name}...")
        model.to(device)
        optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
        best_acc=0
        for epoch in range(1, 21):
            loss = train(model, train_loader, optimizer, device)
            acc = test(model, test_loader, device)
            if(acc>best_acc):
                best_acc=acc
            print(f"Epoch {epoch:02d}, Loss: {loss:.4f}, Test Accuracy: {acc:.
↪4f}")
        print(f"Best accuracy {best_acc}")

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

```
C:\Users\arnav\AppData\Local\Temp\ipykernel_1468\2633872568.py:12:
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)
```

```
Training SimpleGCN…

Training: 100%|      | 125/125 [00:01<00:00, 87.22it/s]

Epoch 01, Loss: 0.6852, Test Accuracy: 0.6830

Training: 100%|      | 125/125 [00:00<00:00, 162.12it/s]

Epoch 02, Loss: 0.6056, Test Accuracy: 0.6760

Training: 100%|      | 125/125 [00:00<00:00, 164.78it/s]

Epoch 03, Loss: 0.5649, Test Accuracy: 0.7460

Training: 100%|      | 125/125 [00:00<00:00, 161.66it/s]

Epoch 04, Loss: 0.5452, Test Accuracy: 0.7450

Training: 100%|      | 125/125 [00:00<00:00, 163.15it/s]

Epoch 05, Loss: 0.5377, Test Accuracy: 0.7500

Training: 100%|      | 125/125 [00:00<00:00, 160.84it/s]

Epoch 06, Loss: 0.5303, Test Accuracy: 0.7580

Training: 100%|      | 125/125 [00:00<00:00, 161.22it/s]
```

```
Epoch 07, Loss: 0.5316, Test Accuracy: 0.7570

Training: 100%|      | 125/125 [00:00<00:00, 161.94it/s]

Epoch 08, Loss: 0.5314, Test Accuracy: 0.7520

Training: 100%|      | 125/125 [00:00<00:00, 160.22it/s]

Epoch 09, Loss: 0.5313, Test Accuracy: 0.7580

Training: 100%|      | 125/125 [00:00<00:00, 162.16it/s]

Epoch 10, Loss: 0.5274, Test Accuracy: 0.7440

Training: 100%|      | 125/125 [00:00<00:00, 158.18it/s]

Epoch 11, Loss: 0.5289, Test Accuracy: 0.7530

Training: 100%|      | 125/125 [00:00<00:00, 161.96it/s]

Epoch 12, Loss: 0.5261, Test Accuracy: 0.7490

Training: 100%|      | 125/125 [00:00<00:00, 157.78it/s]

Epoch 13, Loss: 0.5254, Test Accuracy: 0.7490

Training: 100%|      | 125/125 [00:00<00:00, 145.35it/s]

Epoch 14, Loss: 0.5253, Test Accuracy: 0.7450

Training: 100%|      | 125/125 [00:00<00:00, 160.66it/s]

Epoch 15, Loss: 0.5235, Test Accuracy: 0.7640

Training: 100%|      | 125/125 [00:00<00:00, 146.40it/s]

Epoch 16, Loss: 0.5279, Test Accuracy: 0.7580

Training: 100%|      | 125/125 [00:00<00:00, 153.79it/s]

Epoch 17, Loss: 0.5248, Test Accuracy: 0.7590

Training: 100%|      | 125/125 [00:00<00:00, 163.33it/s]

Epoch 18, Loss: 0.5283, Test Accuracy: 0.7650

Training: 100%|      | 125/125 [00:00<00:00, 166.32it/s]

Epoch 19, Loss: 0.5240, Test Accuracy: 0.7630

Training: 100%|      | 125/125 [00:00<00:00, 166.18it/s]

Epoch 20, Loss: 0.5233, Test Accuracy: 0.7420
Best accuracy 0.765

Training GCNModel…

Training: 100%|      | 125/125 [00:01<00:00, 75.82it/s]

Epoch 01, Loss: 0.5664, Test Accuracy: 0.6950

Training: 100%|      | 125/125 [00:01<00:00, 76.09it/s]
```

```
Epoch 02, Loss: 0.5379, Test Accuracy: 0.7010
Training: 100%|      | 125/125 [00:01<00:00, 75.82it/s]
Epoch 03, Loss: 0.5312, Test Accuracy: 0.7330
Training: 100%|      | 125/125 [00:01<00:00, 75.99it/s]
Epoch 04, Loss: 0.5338, Test Accuracy: 0.7240
Training: 100%|      | 125/125 [00:01<00:00, 75.55it/s]
Epoch 05, Loss: 0.5276, Test Accuracy: 0.7730
Training: 100%|      | 125/125 [00:01<00:00, 75.55it/s]
Epoch 06, Loss: 0.5300, Test Accuracy: 0.7570
Training: 100%|      | 125/125 [00:01<00:00, 76.20it/s]
Epoch 07, Loss: 0.5258, Test Accuracy: 0.7580
Training: 100%|      | 125/125 [00:01<00:00, 75.08it/s]
Epoch 08, Loss: 0.5243, Test Accuracy: 0.7690
Training: 100%|      | 125/125 [00:01<00:00, 76.56it/s]
Epoch 09, Loss: 0.5253, Test Accuracy: 0.7520
Training: 100%|      | 125/125 [00:01<00:00, 76.14it/s]
Epoch 10, Loss: 0.5228, Test Accuracy: 0.7670
Training: 100%|      | 125/125 [00:01<00:00, 76.40it/s]
Epoch 11, Loss: 0.5249, Test Accuracy: 0.7620
Training: 100%|      | 125/125 [00:01<00:00, 76.22it/s]
Epoch 12, Loss: 0.5307, Test Accuracy: 0.7330
Training: 100%|      | 125/125 [00:01<00:00, 75.50it/s]
Epoch 13, Loss: 0.5232, Test Accuracy: 0.7810
Training: 100%|      | 125/125 [00:01<00:00, 76.07it/s]
Epoch 14, Loss: 0.5218, Test Accuracy: 0.7740
Training: 100%|      | 125/125 [00:01<00:00, 75.91it/s]
Epoch 15, Loss: 0.5242, Test Accuracy: 0.7620
Training: 100%|      | 125/125 [00:01<00:00, 75.57it/s]
Epoch 16, Loss: 0.5253, Test Accuracy: 0.7640
Training: 100%|      | 125/125 [00:01<00:00, 76.44it/s]
Epoch 17, Loss: 0.5241, Test Accuracy: 0.7340
Training: 100%|      | 125/125 [00:01<00:00, 75.92it/s]
```

```
Epoch 18, Loss: 0.5222, Test Accuracy: 0.7720

Training: 100%|     | 125/125 [00:01<00:00, 76.14it/s]

Epoch 19, Loss: 0.5230, Test Accuracy: 0.7790

Training: 100%|     | 125/125 [00:01<00:00, 76.16it/s]

Epoch 20, Loss: 0.5236, Test Accuracy: 0.7790
Best accuracy 0.781

Training GATModel…

Training: 100%|     | 125/125 [00:04<00:00, 26.33it/s]

Epoch 01, Loss: 0.5487, Test Accuracy: 0.7170

Training: 100%|     | 125/125 [00:04<00:00, 26.42it/s]

Epoch 02, Loss: 0.5261, Test Accuracy: 0.7690

Training: 100%|     | 125/125 [00:04<00:00, 26.25it/s]

Epoch 03, Loss: 0.5224, Test Accuracy: 0.7740

Training: 100%|     | 125/125 [00:04<00:00, 26.26it/s]

Epoch 04, Loss: 0.5157, Test Accuracy: 0.7710

Training: 100%|     | 125/125 [00:04<00:00, 26.47it/s]

Epoch 05, Loss: 0.5276, Test Accuracy: 0.7460

Training: 100%|     | 125/125 [00:04<00:00, 26.37it/s]

Epoch 06, Loss: 0.5213, Test Accuracy: 0.7690

Training: 100%|     | 125/125 [00:04<00:00, 26.43it/s]

Epoch 07, Loss: 0.5208, Test Accuracy: 0.7260

Training: 100%|     | 125/125 [00:04<00:00, 26.42it/s]

Epoch 08, Loss: 0.5191, Test Accuracy: 0.7600

Training: 100%|     | 125/125 [00:04<00:00, 26.40it/s]

Epoch 09, Loss: 0.5252, Test Accuracy: 0.7590

Training: 100%|     | 125/125 [00:04<00:00, 26.42it/s]

Epoch 10, Loss: 0.5200, Test Accuracy: 0.7610

Training: 100%|     | 125/125 [00:04<00:00, 26.47it/s]

Epoch 11, Loss: 0.5241, Test Accuracy: 0.7400

Training: 100%|     | 125/125 [00:04<00:00, 26.46it/s]

Epoch 12, Loss: 0.5194, Test Accuracy: 0.7580

Training: 100%|     | 125/125 [00:04<00:00, 26.39it/s]
```

```
Epoch 13, Loss: 0.5157, Test Accuracy: 0.7520

Training: 100%|     | 125/125 [00:04<00:00, 26.46it/s]

Epoch 14, Loss: 0.5180, Test Accuracy: 0.7770

Training: 100%|     | 125/125 [00:04<00:00, 26.47it/s]

Epoch 15, Loss: 0.5192, Test Accuracy: 0.7610

Training: 100%|     | 125/125 [00:04<00:00, 26.50it/s]

Epoch 16, Loss: 0.5194, Test Accuracy: 0.7700

Training: 100%|     | 125/125 [00:04<00:00, 26.46it/s]

Epoch 17, Loss: 0.5200, Test Accuracy: 0.7720

Training: 100%|     | 125/125 [00:04<00:00, 26.43it/s]

Epoch 18, Loss: 0.5210, Test Accuracy: 0.7720

Training: 100%|     | 125/125 [00:04<00:00, 26.47it/s]

Epoch 19, Loss: 0.5216, Test Accuracy: 0.7710

Training: 100%|     | 125/125 [00:04<00:00, 26.37it/s]

Epoch 20, Loss: 0.5234, Test Accuracy: 0.7680
Best accuracy 0.777

Training ResidualGCNModel…

Training: 100%|     | 125/125 [00:02<00:00, 54.51it/s]

Epoch 01, Loss: 0.5456, Test Accuracy: 0.7620

Training: 100%|     | 125/125 [00:02<00:00, 54.75it/s]

Epoch 02, Loss: 0.5303, Test Accuracy: 0.7250

Training: 100%|     | 125/125 [00:02<00:00, 54.55it/s]

Epoch 03, Loss: 0.5317, Test Accuracy: 0.7710

Training: 100%|     | 125/125 [00:02<00:00, 54.70it/s]

Epoch 04, Loss: 0.5214, Test Accuracy: 0.7440

Training: 100%|     | 125/125 [00:02<00:00, 54.04it/s]

Epoch 05, Loss: 0.5295, Test Accuracy: 0.7790

Training: 100%|     | 125/125 [00:02<00:00, 54.69it/s]

Epoch 06, Loss: 0.5256, Test Accuracy: 0.7770

Training: 100%|     | 125/125 [00:02<00:00, 54.87it/s]

Epoch 07, Loss: 0.5184, Test Accuracy: 0.7780

Training: 100%|     | 125/125 [00:02<00:00, 54.77it/s]
```

```
Epoch 08, Loss: 0.5175, Test Accuracy: 0.7560

Training: 100%|      | 125/125 [00:02<00:00, 54.76it/s]

Epoch 09, Loss: 0.5210, Test Accuracy: 0.7530

Training: 100%|      | 125/125 [00:02<00:00, 54.89it/s]

Epoch 10, Loss: 0.5224, Test Accuracy: 0.7840

Training: 100%|      | 125/125 [00:02<00:00, 54.15it/s]

Epoch 11, Loss: 0.5161, Test Accuracy: 0.7810

Training: 100%|      | 125/125 [00:02<00:00, 54.56it/s]

Epoch 12, Loss: 0.5238, Test Accuracy: 0.7360

Training: 100%|      | 125/125 [00:02<00:00, 54.76it/s]

Epoch 13, Loss: 0.5233, Test Accuracy: 0.7770

Training: 100%|      | 125/125 [00:02<00:00, 54.57it/s]

Epoch 14, Loss: 0.5144, Test Accuracy: 0.7370

Training: 100%|      | 125/125 [00:02<00:00, 54.85it/s]

Epoch 15, Loss: 0.5201, Test Accuracy: 0.7390

Training: 100%|      | 125/125 [00:02<00:00, 54.76it/s]

Epoch 16, Loss: 0.5182, Test Accuracy: 0.7650

Training: 100%|      | 125/125 [00:02<00:00, 54.61it/s]

Epoch 17, Loss: 0.5215, Test Accuracy: 0.7440

Training: 100%|      | 125/125 [00:02<00:00, 53.93it/s]

Epoch 18, Loss: 0.5148, Test Accuracy: 0.7650

Training: 100%|      | 125/125 [00:02<00:00, 54.72it/s]

Epoch 19, Loss: 0.5171, Test Accuracy: 0.7840

Training: 100%|      | 125/125 [00:02<00:00, 54.68it/s]

Epoch 20, Loss: 0.5157, Test Accuracy: 0.7730
Best accuracy 0.784
```

3 Accuracy

Simple GCN —-76.5% GCN Model —-78.1% GAT Model —-77.7% Residual GCN —-78.4%

3.0.1 Simple GCN

It is a simple model with only 2 layers and it cannot capture complex relations

3.0.2 GCN Model

The GCN model adds batch normalization that helps to stabalize the learning rates

3.0.3 GAT Model

Graph Attention Network applies attention mechanism that weigh the importance of neighbour nodes

3.0.4 Residual GCN

It adds recidual connection which helps to preserve low level features and improves gradient flow

4 Future Improvements

We can do hyperparameter tuning and increase the epochs and try new and complex models that can increase the accuracy

Due to memory and computational issues I have to preprocess the data in batch and we have to convert float64 to float32 I have implemented four simple models that take less computation as fine tuning took a lot of time and increasing layer or hidden dimension had both time and memory issues, so I implemented 4 models.