In [ ]:
```
# This data handling code is adapted from the PyTorch geometric collection c
import torch
from torch_geometric.datasets import Planetoid
from torch_geometric.transforms import NormalizeFeatures
from torch_geometric.data import DataLoader
# import the graph classifier you built in the last step
from GCN_03 import NodeClassifier, NodeClassifierWelling
```

In [ ]:
```
# - - - DATA PREPARATIONS - - -

dataset = Planetoid(root='data/Planetoid', name='Cora', transform=NormalizeF
print()
print(f'Dataset: {dataset}:')
print('====================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
data = dataset[0]  # Get the first graph object.
print()
print(data)
print('=============================================================')
# Gather some statistics about the first graph.
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f}')
print(f'Contains isolated nodes: {data.contains_isolated_nodes()}')
print(f'Contains self-loops: {data.contains_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
```

```
Dataset: Cora():
====================
Number of graphs: 1
Number of features: 1433
Number of classes: 7

Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val
_mask=[2708], test_mask=[2708])
=============================================================
Number of nodes: 2708
Number of edges: 10556
Average node degree: 3.90
Contains isolated nodes: False
Contains self-loops: False
Is undirected: True
```

The size of tensor a (2708) must match the size of tensor b (13264) at non-singleton dimension 0

In [ ]:
```
def train(model):
    model.train()
    optimizer.zero_grad()  # Clear gradients.
    out = model(data.x, data.edge_index)  # Perform a single forward pass.
    loss = criterion(out[data.train_mask],
                     data.y[data.train_mask])  # Compute the loss solely bas
    loss.backward()  # Derive gradients.
    optimizer.step()  # Update parameters based on gradients.
    return loss
```

In [ ]:
```
def test(model):
    model.eval()
    out = model(data.x, data.edge_index)
```

```
    pred = out.argmax(dim=1)  # Use the class with highest probability.
    test_correct = pred[data.test_mask] == data.y[data.test_mask]  # Check a
    test_acc = int(test_correct.sum()) / int(data.test_mask.sum())  # Derive
    return test_acc
```

New Model

In [ ]:
```
model_new = NodeClassifier(num_node_features=1433, hidden_features=16, num_c
optimizer = torch.optim.Adam(model_new.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

In [ ]:
```
for epoch in range(1, 201):
    loss = train(model_new)
    if epoch % 10 == 0:
        test_acc = test(model_new)
        print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Test Accuracy: {test_a
```

```
Epoch: 010, Loss: 1.9207, Test Accuracy: 0.138
Epoch: 020, Loss: 1.7281, Test Accuracy: 0.391
Epoch: 030, Loss: 1.4046, Test Accuracy: 0.47
Epoch: 040, Loss: 1.1288, Test Accuracy: 0.516
Epoch: 050, Loss: 0.8131, Test Accuracy: 0.576
Epoch: 060, Loss: 0.4540, Test Accuracy: 0.688
Epoch: 070, Loss: 0.3599, Test Accuracy: 0.701
Epoch: 080, Loss: 0.2191, Test Accuracy: 0.696
Epoch: 090, Loss: 0.1600, Test Accuracy: 0.697
Epoch: 100, Loss: 0.1501, Test Accuracy: 0.682
Epoch: 110, Loss: 0.0879, Test Accuracy: 0.686
Epoch: 120, Loss: 0.1622, Test Accuracy: 0.679
Epoch: 130, Loss: 0.0675, Test Accuracy: 0.652
Epoch: 140, Loss: 0.0855, Test Accuracy: 0.693
Epoch: 150, Loss: 0.0419, Test Accuracy: 0.683
Epoch: 160, Loss: 0.0650, Test Accuracy: 0.676
Epoch: 170, Loss: 0.0268, Test Accuracy: 0.684
Epoch: 180, Loss: 0.0425, Test Accuracy: 0.66
Epoch: 190, Loss: 0.0555, Test Accuracy: 0.691
Epoch: 200, Loss: 0.0210, Test Accuracy: 0.67
```

Old model (Welling et al., 2011) for node classification

In [ ]:
```
model_welling = NodeClassifierWelling(num_node_features=1433, hidden_feature
optimizer = torch.optim.Adam(model_welling.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

In [ ]:
```
for epoch in range(1, 201):
    loss = train(model_welling)
    if epoch % 10 == 0:
        test_acc = test(model_welling)
        print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Test Accuracy: {test_a
```

```
Epoch: 010, Loss: 1.9051, Test Accuracy: 0.26
Epoch: 020, Loss: 1.7933, Test Accuracy: 0.397
Epoch: 030, Loss: 1.6306, Test Accuracy: 0.487
Epoch: 040, Loss: 1.3842, Test Accuracy: 0.574
Epoch: 050, Loss: 1.1442, Test Accuracy: 0.642
Epoch: 060, Loss: 0.9415, Test Accuracy: 0.699
Epoch: 070, Loss: 0.7291, Test Accuracy: 0.732
Epoch: 080, Loss: 0.5568, Test Accuracy: 0.759
Epoch: 090, Loss: 0.4906, Test Accuracy: 0.767
Epoch: 100, Loss: 0.3739, Test Accuracy: 0.772
Epoch: 110, Loss: 0.2850, Test Accuracy: 0.771
Epoch: 120, Loss: 0.2750, Test Accuracy: 0.773
Epoch: 130, Loss: 0.2425, Test Accuracy: 0.774
Epoch: 140, Loss: 0.1538, Test Accuracy: 0.773
Epoch: 150, Loss: 0.1688, Test Accuracy: 0.78
Epoch: 160, Loss: 0.1452, Test Accuracy: 0.774
Epoch: 170, Loss: 0.1105, Test Accuracy: 0.778
Epoch: 180, Loss: 0.1324, Test Accuracy: 0.774
Epoch: 190, Loss: 0.1507, Test Accuracy: 0.773
Epoch: 200, Loss: 0.1040, Test Accuracy: 0.772
```

The new model model has lower loss, but also lower accuracy. Increasing the size of Multilayer perceptron might increase the expressive power of the model and improve the accuracy.