```
# Install required packages.
import os
import torch
os.environ['TORCH'] = torch.__version__
print(torch.__version__)

!pip install -q torch-scatter -f https://data.pyg.org/whl/torch-${TORCH}.html
!pip install -q torch-sparse -f https://data.pyg.org/whl/torch-${TORCH}.html
!pip install -q git+https://github.com/pyg-team/pytorch_geometric.git

# Helper function for visualization.
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

def visualize(h, color):
    z = TSNE(n_components=2).fit_transform(h.detach().cpu().numpy())

    plt.figure(figsize=(10,10))
    plt.xticks([])
    plt.yticks([])

    plt.scatter(z[:, 0], z[:, 1], s=70, c=color, cmap="Set2")
    plt.show()
```

```
 2.2.1+cu121
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
```

Node Classification with Graph Neural Networks

Previous: Introduction: Hands-on Graph Neural Networks

This tutorial will teach you how to apply **Graph Neural Networks (GNNs) to the task of node classification**. Here, we are given the ground-truth labels of only a small subset of nodes, and want to infer the labels for all the remaining nodes (*transductive learning*).

To demonstrate, we make use of the `Cora` dataset, which is a **citation network** where nodes represent documents. Each node is described by a 1433-dimensional bag-of-words feature vector. Two documents are connected if there exists a citation link between them. The task is to infer the category of each document (7 in total).

This dataset was first introduced by Yang et al. (2016) as one of the datasets of the `Planetoid` benchmark suite. We again can make use PyTorch Geometric for an easy access to this dataset via `torch_geometric.datasets.Planetoid`:

```
from torch_geometric.datasets import Planetoid
from torch_geometric.transforms import NormalizeFeatures

dataset = Planetoid(root='data/Planetoid', name='Cora', transform=NormalizeFeatures())

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 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'Number of training nodes: {data.train_mask.sum()}')
print(f'Number of valdiation nodes: {data.val_mask.sum()}')
print(f'Number of testing nodes: {data.test_mask.sum()}')
print(f'Training node label rate: {int(data.train_mask.sum()) / data.num_nodes:.2f}')
print(f'Has isolated nodes: {data.has_isolated_nodes()}')
print(f'Has self-loops: {data.has_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
Number of training nodes: 140
Number of valdiation nodes: 500
Number of testing nodes: 1000
Training node label rate: 0.05
Has isolated nodes: False
Has self-loops: False
Is undirected: True
```

Overall, this dataset is quite similar to the previously used `KarateClub` network. We can see that the `Cora` network holds 2,708 nodes and 10,556 edges, resulting in an average node degree of 3.9. For training this dataset, we are given the ground-truth categories of 140 nodes (20 for each class). This results in a training node label rate of only 5%.

In contrast to `KarateClub`, this graph holds the additional attributes `val_mask` and `test_mask`, which denotes which nodes should be used for validation and testing. Furthermore, we make use of **data transformations** **via `transform=NormalizeFeatures()`**. Transforms can be used to modify your input data before inputting them into a neural network, *e.g.*, for normalization or data augmentation. Here, we row-normalize the bag-of-words input feature vectors.

We can further see that this network is undirected, and that there exists no isolated nodes (each document has at least one citation).

Comment FELIXWAL

a.)

- Number of nodes in training set:
  - 140
- Number of nodes in validation set:
  - 500
- Number of nodes in testing set:
  - 1000

Supervised learning gives the model accurate information on the classification of the processed data. But this also means that this classification needs to be done in advance, which can be a fairly time consuming process. With semi supervised learning, we have more data to classify, which counters the problem of overfitting, but in the case of anonamlies in this data, they would most likely not be properly classified and therefore lead to errors in the training.

Training a Multi-layer Perception Network (MLP)

In theory, we should be able to infer the category of a document solely based on its content, *i.e.* its bag-of-words feature representation, without taking any relational information into account.

Let's verify that by constructing a simple MLP that solely operates on input node features (using shared weights across all nodes):

```
import torch
from torch.nn import Linear
import torch.nn.functional as F


class MLP(torch.nn.Module):
    def __init__(self, hidden_channels):
        super().__init__()
        torch.manual_seed(12345)
        self.lin1 = Linear(dataset.num_features, hidden_channels)
        self.lin2 = Linear(hidden_channels, dataset.num_classes)

    def forward(self, x):
        x = self.lin1(x)
        x = x.relu()
        # To have a dense network, we delete the dropout
        # x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin2(x)
        return x

model = MLP(hidden_channels=16)
print(model)
```

```
MLP(
  (lin1): Linear(in_features=1433, out_features=50, bias=True)
  (lin2): Linear(in_features=50, out_features=7, bias=True)
)
```

Our MLP is defined by two linear layers and enhanced by ReLU non-linearity and dropout. Here, we first reduce the 1433-dimensional feature vector to a low-dimensional embedding (`hidden_channels=16`), while the second linear layer acts as a classifier that should map each low-dimensional node embedding to one of the 7 classes.

Let's train our simple MLP by following a similar procedure as described in the first part of this tutorial. We again make use of the **cross entropy loss** and **Adam optimizer**. This time, we also define a **`test` function** to evaluate how well our final model performs on the test node set (which labels have not been observed during training).

```
from IPython.display import Javascript  # Restrict height of output cell.
display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 300})'''))

model = MLP(hidden_channels=16)
criterion = torch.nn.CrossEntropyLoss()  # Define loss criterion.
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)  # Define optimizer.

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

def test():
      model.eval()
      out = model(data.x)
      pred = out.argmax(dim=1)  # Use the class with highest probability.
      test_correct = pred[data.test_mask] == data.y[data.test_mask]  # Check against ground-truth labels.
      test_acc = int(test_correct.sum()) / int(data.test_mask.sum())  # Derive ratio of correct predictions.
      return test_acc

for epoch in range(1, 201):
    loss = train()
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}')
```

```
Epoch: 001, Loss: 1.9615
Epoch: 002, Loss: 1.9538
Epoch: 003, Loss: 1.9452
Epoch: 004, Loss: 1.9357
Epoch: 005, Loss: 1.9250
Epoch: 006, Loss: 1.9136
Epoch: 007, Loss: 1.9011
Epoch: 008, Loss: 1.8877
Epoch: 009, Loss: 1.8735
Epoch: 010, Loss: 1.8583
Epoch: 011, Loss: 1.8421
Epoch: 012, Loss: 1.8250
Epoch: 013, Loss: 1.8068
Epoch: 014, Loss: 1.7875
Epoch: 015, Loss: 1.7672
Epoch: 016, Loss: 1.7457
Epoch: 017, Loss: 1.7232
```

After training the model, we can call the `test` function to see how well our model performs on unseen labels. Here, we are interested in the accuracy of the model, *i.e.*, the ratio of correctly classified nodes:

```
test_acc = test()
print(f'Test Accuracy: {test_acc:.4f}')
```

```
Test Accuracy: 0.5940
```

As one can see, our MLP performs rather bad with only about 59% test accuracy. But why does the MLP do not perform better? The main reason for that is that this model suffers from heavy overfitting due to only having access to a **small amount of training nodes**, and therefore generalizes poorly to unseen node representations.

It also fails to incorporate an important bias into the model: **Cited papers are very likely related to the category of a document**. That is exactly where Graph Neural Networks come into play and can help to boost the performance of our model.

This is the standard dense model

Training a Graph Neural Network (GNN)

We can easily convert our MLP to a GNN by swapping the `torch.nn.Linear` layers with PyG's GNN operators.

Following-up on the first part of this tutorial, we replace the linear layers by the `GCNConv` module. To recap, the **GCN layer** (Kipf et al. (2017)) is defined as

$$\mathbf{x}_v^{(\ell+1)} = \mathbf{W}^{(\ell+1)} \sum_{w \in \square(v) \cup \{v\}} \frac{1}{c_{w,v}} \cdot \mathbf{x}_w^{(\ell)}$$

where $\mathbf{W}^{(\ell+1)}$ denotes a trainable weight matrix of shape `[num_output_features, num_input_features]` and $c_{w,v}$ refers to a fixed normalization coefficient for each edge. In contrast, a single `Linear` layer is defined as

$$\mathbf{x}_v^{(\ell+1)} = \mathbf{W}^{(\ell+1)} \mathbf{x}_v^{(\ell)}$$

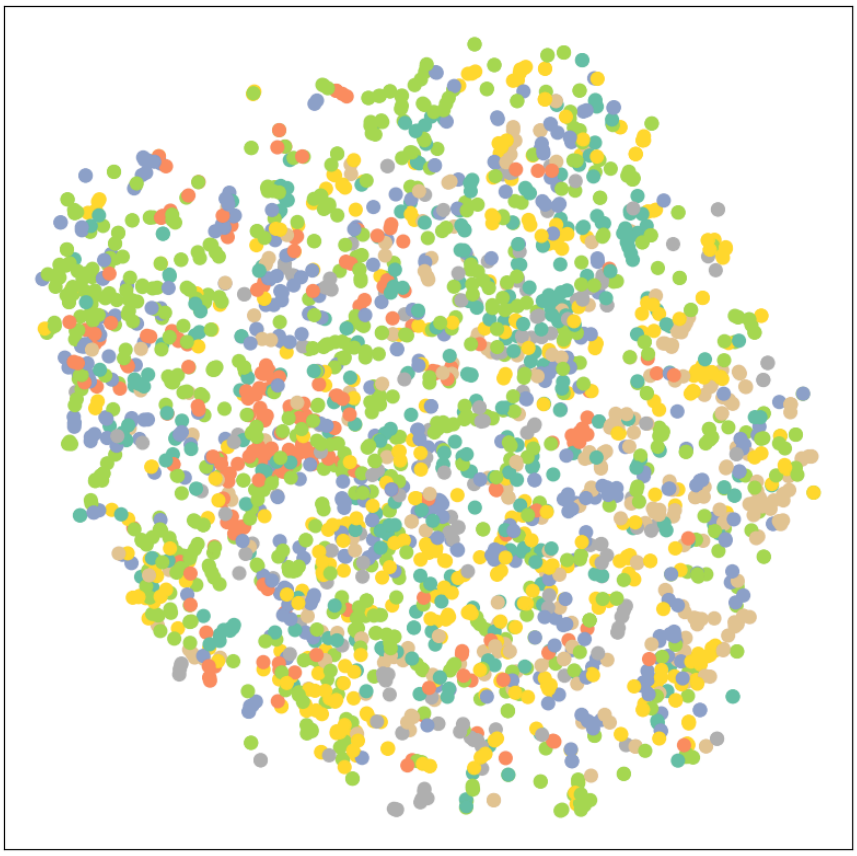
which does not make use of neighboring node information.

```
from torch_geometric.nn import GCNConv

class GCN(torch.nn.Module):
    def __init__(self, hidden_channels):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GCNConv(dataset.num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, dataset.num_classes)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv2(x, edge_index)
        return x

model = GCN(hidden_channels=16)
print(model)
```

```
GCN(
  (conv1): GCNConv(1433, 16)
  (conv2): GCNConv(16, 7)
)
```

Let's visualize the node embeddings of our **untrained** GCN network. For visualization, we make use of **TSNE** to embed our 7-dimensional node embeddings onto a 2D plane.

```
model = GCN(hidden_channels=16)
model.eval()

out = model(data.x, data.edge_index)
visualize(out, color=data.y)
```



We certainly can do better by training our model. The training and testing procedure is once again the same, but this time we make use of the node features `x` **and** the graph connectivity `edge_index` as input to our GCN model.

$$X^{l+1} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{l}W^{l})$$

```
from IPython.display import Javascript  # Restrict height of output cell.
display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 300})'''))

model = GCN(hidden_channels=16)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)
criterion = torch.nn.CrossEntropyLoss()

def train():
      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 based on the training nodes
      loss.backward()  # Derive gradients.
      optimizer.step()  # Update parameters based on gradients.
      return loss

def test():
      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 against ground-truth labels.
      test_acc = int(test_correct.sum()) / int(data.test_mask.sum())  # Derive ratio of correct predictions.
      return test_acc


for epoch in range(1, 101):
    loss = train()
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}')
```