Emerging Directions in Deep Learning

Laboratory 3

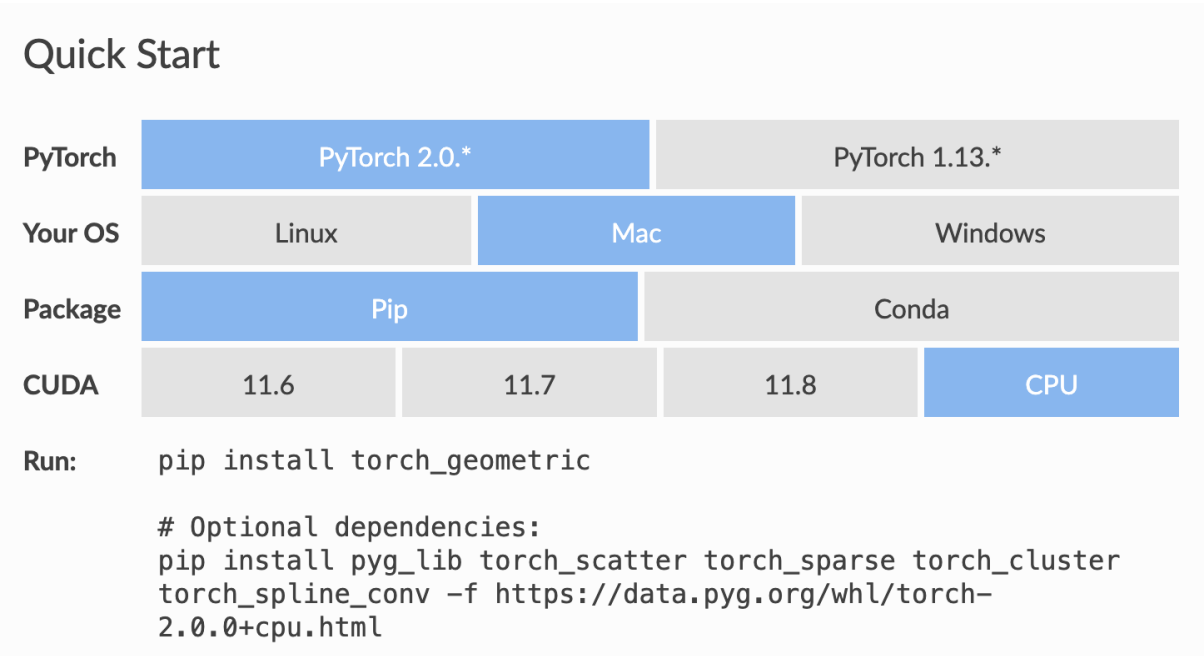
Dana Dăscălescu

April 26, 2023

Exercise 1

Task: Install the *PyTorch Geometric* package [1]. Run the notebooks L_03_1, L_03_2, L_03_3 provided in the laboratory class.

The official documentation of PyTorch Geometric contains instructions for installation:

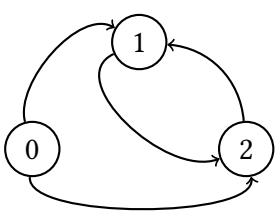


Exercise 2

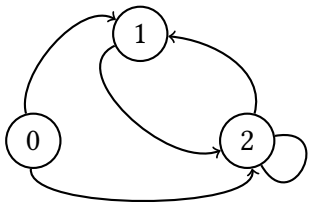
Task: Make the changes to each notebook, as follows:

- `L_03_01` - describe another graph;
- `L_03_02` - load another dataset;
- `L_03_03` - improve the accuracy.

Task 2.1: The following graphs will be created



(a) First graph



(b) Second graph

(a)

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 0, 1, 2],
                           [1, 2, 2, 1]])
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)
data3 = Data(x=x, edge_index=edge_index.contiguous())
```

The provided code snippet is creating a graph using PyTorch Geometric library. The graph has four edges connecting various nodes, specifically nodes 0 and 1, 0 and 2, 1 and 2, and 2 and 1. The first row of the 'edge_index' tensor represents the source nodes of each edge, whereas the second row denotes the target nodes of each edge.

The 'Data' object is being used to create the graph with the given edge_index and node feature tensor x. The node feature tensor 'x' has three nodes, and each node has one feature represented by a scalar value.

The 'edge_index' tensor is a two-dimensional tensor comprising of two rows and four columns, where each column represents an edge in the graph. The first row of the tensor contains the source nodes of each edge, and the second row contains the target nodes of each edge. For example, the first edge connects node 0 (source) and node 1 (target), and the second edge connects node 0 (source) and node 2 (target).

The 'contiguous()' function is being used to ensure that the edge_index tensor is contiguous in memory.

(b)

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 0, 1, 2, 2],
                           [1, 2, 2, 1, 2]])
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)
data3 = Data(x=x, edge_index=edge_index.contiguous())
```

The following code snippet represents the same graph as the previous example, but with an additional self-loop.

Task 2.2: For selecting a dataset, I consulted the Data Cheatsheet available in the official documentation of PyTorch Geometric. I decided to use the PubMed subset from PlaNetoid.

```
# Load dataset
from torch_geometric.datasets import TUDataset, Planetoid
dataset = Planetoid(root='/tmp/PubMed', name='PubMed')
data=dataset[0]
```

Task 2.3: To further improve the accuracy of the model and increase the training procedure, I developed a new neural network architecture using PyTorch and PyTorch Geometric libraries. The code snippet below shows the implementation of the new architecture:

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, GATConv

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = GATConv(dataset.num_node_features, 64, heads=8,
   dropout=0.5)
        self.conv2 = GATConv(64 * 8, 32, heads=1, concat=True)
        self.conv3 = GCNConv(32, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv3(x, edge_index)

        return F.log_softmax(x, dim=1)
```

Accuracy obtained:

```
_, pred = model(data).max(dim=1)
correct = int(pred[data.test_mask].eq(data.y[data.test_mask]).sum().item())
acc = correct / int(data.test_mask.sum())
print('Accuracy: {:.4f}'.format(acc))
```

```
Accuracy: 0.7900
```

The architecture consists of three layers of graph convolutional networks (GCN) and graph attention networks (GAT) that take the feature matrix and adjacency matrix of a graph as input. The first layer is a GAT layer with 64 output features and 8 attention heads, followed by a ReLU activation function and a 50% dropout rate. The second layer is also a GAT layer, but with 32 output features and a single attention head, which receives as input the concatenation of the output of the previous layer's heads. A 50% dropout rate and another ReLU activation

function follow this. Finally, the third layer is a GCN layer with 32 input features and a number of output features equal to the number of classes in the dataset.

The forward function defines the forward pass of the network. It first applies the first GAT layer to the input features and adjacency matrix, followed by a ReLU activation function and a dropout operation. Then, it applies the second GAT layer, another ReLU activation function, and dropout. Finally, it applies the GCN layer and returns the output after a logarithmic softmax function is applied to it.

Overall, the new architecture increases the depth of the network and the number of trainable parameters, which can help improve the accuracy of the model. Additionally, the 50% dropout rate helps prevent overfitting during training, which can result in more robust and generalizable results.

Exercise 3

Task: Run the Colab Notebooks 1, 2, 3 from

Notebook 1: This notebook provides an overview of a hands-on introduction to deep learning on graphs using graph neural networks based on PyTorch Geometric (PyG) library [1]. The notebook's exercises concentrate on a simple graph-structure example, the Zachary's Karate Club network [4], which describes a social network of 34 karate club members and records links between members who interacted outside of the club. The task at hand is to identify communities that forms from member interactions.

The first section of this notebook introduces data manipulation with PyG. Each graph in PyTorch Geometric is represented by a singular Data object that contains all of the relevant data. The properties of the Data object, which is analogous to a dictionary, are discussed, and utility functions for inferring the fundamental properties are presented. Several data visualisation tools are also introduced to assist in data comprehension.

The second section of the notebook concentrates on Graph Neural Networks (GNNs) implementation. Based on the research of [2], a concise introduction to the underlying theory is provided. Additionally, the benefits of GNNs are discussed. Even before training the model's weights, GNNs produce an embedding of nodes that closely resembles the community structure of the graph. This implies that nodes belonging to the same community are already densely clustered, introducing a strong inductive bias that results in similar embeddings for nodes that are near to one another in the input graph.

A training procedure for the Karate Club Network is outlined in the final section of the notebook. Also provided are the results of the training, which demonstrate the efficacy of the GNN approach in identifying communities in the graph.

Notebook 2: The focus of this notebook is Node Classification with Graph Neural Networks. The notebook utilises the Cora dataset introduced by Yang et al. (2016) [3] as part of the Planetoid benchmark suite. Cora is a citation network in which nodes represent documents and edges represent citation connections between two documents. The task at hand is to categorise each document.

A Multilayer Perceptron (MLP) is initially trained, and its efficacy is compared to that of a Graph Neural Network (GNN) trained on the same data. On the other hand, the MLP performs poorly on the limited data, resulting in overfitting. In contrast, the GNN incorporates an essential bias into the model, namely that cited papers are highly likely to be associated with a document's category. The GNN improves classification performance by propagating information from citation network neighbours.

I will present my solution to the set of optional exercises below.

1. The code needs to be modified to select and test the best model using the validation node set from the Cora dataset. This will help improve model performance and prevent overfitting.

```
# Initialize the best validation accuracy to be 0.
best_val_acc = 0.0

for epoch in range(1, 101):
    loss = train()
    # Evaluate on the validation set.
    val_acc = test()
    # Update the best validation accuracy.
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        # Save the best model so far.
        torch.save(model.state_dict(), 'best_model.pth')
```

2. What is the impact of increasing the hidden feature dimensionality or adding layers on the behavior of GCN? Does adding more layers result in any improvement?

   In the experiments, it was observed that increasing the hidden feature dimensionality from 16 to 32 and 64 did not have any significant impact on the behaviour of GCN. However, adding more layers resulted in a slight improvement in accuracy. This suggests that deeper models with more layers could potentially learn more complex patterns and representations, leading to better performance.

3. You can experiment with various GNN layers to observe how model performance varies. What happens if all GCNConv instances are replaced with GATConv layers that utilise attention? Attempt to create a 2-layer GAT model with 8 attention heads in the first layer and 1 attention head in the second layer, a dropout ratio of 0.6 inside and outside each GATConv call, and 8 hidden_channels per head.

```
class GAT(torch.nn.Module):
    def __init__(self, hidden_channels, heads):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GATConv(dataset.num_features, hidden_channels,
   heads=heads)
        self.conv2 = GATConv(hidden_channels * heads, dataset.
   num_classes)
```

```
    def forward(self, x, edge_index):
        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv1(x, edge_index)
        x = F.elu(x)
        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv2(x, edge_index)
        return x

model = GAT(hidden_channels=32, heads=8)
```

Notebook 3: This tutorial notebook centred on graph classification with Graph Neural Networks (GNNs). The main task is to predict molecular properties, where molecules are represented as graphs, and the objective may be to infer whether a molecule inhibits HIV virus replication or not. The MUTAG dataset from TUDatasets collected by TU Dortmund University is used in this tutorial. The notebook presents an approach for mini-batching of graphs that is more practical and less memory efficient than the traditional methods. Specifically, the adjacency matrices are arranged diagonally to generate an immense graph containing numerous isolated subgraphs. Nodes and target features are then simply concatenated in the node dimension. The readout layer is implemented in this tutorial by simply averaging the node embeddings.

I will present my solution to the optional exercises listed below:

As per recent research, applying neighbourhood normalisation can decrease the ability of GNNs to distinguish certain graph structures. To overcome this, an alternative formulation has been proposed that involves skipping neighbourhood normalisation and adding a simple skip-connection to the GNN layer. This formulation is implemented as GraphConv in the PyTorch Geometric library. To explore this approach, the task is to modify the provided code and replace GCNConv with GraphConv.

Solution:

```
from torch_geometric.nn import GraphConv


class GNN(torch.nn.Module):
    def __init__(self, hidden_channels):
        super(GNN, self).__init__()
        torch.manual_seed(12345)
        self.conv1 = GraphConv(dataset.num_node_features, hidden_channels
   )
        self.conv2 = GraphConv(hidden_channels, hidden_channels)
        self.conv3 = GraphConv(hidden_channels, hidden_channels)
        self.lin = Linear(hidden_channels, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = self.conv3(x, edge_index)
```

```
        x = global_mean_pool(x, batch)

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

        return x

model = GNN(hidden_channels=64)
print(model)
```

References

[1] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. arXiv: 1903.02428 [cs.LG].

[2] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG].

[3] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. *Revisiting Semi-Supervised Learning with Graph Embeddings*. 2016. arXiv: 1603.08861 [cs.LG].

[4] Wayne W Zachary. "An information flow model for conflict and fission in small groups." In: *Journal of anthropological research* 33.4 (1977), pp. 452–473.