Supernode
Advanced Topics in Machine Learning and Optimization

Samuele Angheben, 240268

1 Introduction

2 Data

graph classification

2.1 Synthetic Data Generation

For the first experiments we decided to build our own datasets, in this way we are in control of the full details and understanding of the data. In particular we build ad-hoc dataset with particular patterns that represent some concepts.

2.1.1 Dataset_tree_cycle

This is the first dataset that we synthesized, it is composed of graphs with and without cycles, in particular the class of each graph is given by this property.
Considering the fact that the concept `cycle_basis` for each cycle will create a supernode connected to each node of the corrispettive cycle, the idea is to check whether this preprocessing phase will actually increase the performance of the classification.

Remark. In this case the preprocessing computation alone will be able to correctly classify the dataset, since to add a supernode we need to find the cycles, but the goal is understand if the model will benefit from this preprocessing on the graph to then apply this techniques in more complex settings.

To synthesize this dataset we exploit the fact that a graph without cycles is a tree and therefore we first constructed random trees and then to half of them we added `cycle_level`-times random edges. In this way we can generate this type of dataset quickly and flexibly in respect to number of graphs, graphs size, cycle level and proportions.

Remark. We set each node features to [1], in this way we force the model to reason more on the structure of the graph.

After the dataset generation we implemented the `torch.geometric.data.Dataset` class for this dataset to integrate it to the torch ecosystem. At a later time we implemented the class `torch.geometric.data.InMemoryDataset` to have better performance during the training phase. Then we also create a `dataloader` to split our dataset in training, validation and test set according to our needs.

Remark. Code in the folder `supernode/dataset`

2.2 Concepts and Supernodes

Definition 2.1 (Concept)**.** A concept is a pattern in a graph. Given a graph $G = (V, E)$ we can extract each realization of it as a list of subsets of nodes $[C_1, \dots, C_n]$ where $C_i \subseteq V$.

Example. `max_cliques`, `cycle_basis`, `line_paths`, `star` ◇

Remark. Check the file `ConceptsVisualization.pdf` for the code and visualization of each concept on a simple graph.

For each concepts in the example we built a function to extract the corrispettive list of subnodes with the help of the `networkx` library.

Definition 2.2 (Supernodes)**.** Given a graph $G = (V, E)$ and a list $[C_1, \ldots, C_n]$ where $C_i \subseteq V$ identify a concept in the graph, we define for each C_i a supernode a node S_i.
We then can add the supernodes S_i to the graph G in this way:

$$G' = (V + S_i, E + E_i) \qquad \forall i$$

where $E_i = \{(S_i, v_j) | v_j \in C_i\}$

Remark. A single concept create multiple supernodes if the pattern that it represent can be found in different location of the graph, so if we decide to transforms our data with n concepts we will get n set of supernodes (n types of supernodes), each set (type) corresponds to one concept and will contains one supernode for each realization of the particular pattern.

2.3 AddSupernode Transformation

To integrate this techniques with the torch geometric ecosystem we implemented the transformation `AddSupernode` from the class `torch_geometric.transforms.BaseTransform`.
In summary given a list of concepts we convert the graph to networkx to extract each concepts, for each of them we add the corrispettive supernode and edges, initialize its features to [1] and convert it back to the torch geometric representation. This is very useful because now we can easily transform the dataset:

Code (Transform - AddSupernode)**.**

```
concepts_list_ex = [
        {"name": "GCB", "fun": cycle_basis, "args": []},
        {"name": "GMC", "fun": max_cliques, "args": []},
        {"name": "GLP2", "fun": line_paths, "args": [2]}
    ]

dataset.transform = AddSupernodes(concepts_list_ex)
```

And since we have implemented also the `Dataset` class we can pretransform the dataset:

Code (Pretransform - AddSupernode)**.**

```
dataset = Dataset_tree_cycle_Memory(root="./dataset/d1MemT",
                                    dataset_path="./project/dataset/d1",
                                    pre_transform=AddSupernodes(concepts_list_ex))
```

In this way we compute the preprocessing of the dataset only one time, then is loaded from the disk if the transformation is not changed.

Remark. The graphs both before and after the `AddSupernode` transformation are homogeneous graph, this means that all nodes are of the same type.

Remark. Code in the folder `supernode/concepts`

3 Models

In this section we present the different models architecture that we used to analyze the supernodes transformation.
To recap, Graph neural network for graph classification is usually composed of three parts:

1. Node embeddings: a sequence of graph convolutional layers that compute node embeddings.
2. Readout: a function that aggregates node embeddings into a graph embedding.
3. Classifier: a function that takes the graph embedding and computes the output.

We follow this structure for all the models that we implemented, but we added a new step at the beginning which works only if the data has been transformed, in other words if the graph contains the supernodes:

0. Supernode embeddings: a graph convolutional layers that compute supernode embeddings.

To perform the Supernode embeddings operation on homogeneous graph we:

1. During the `AddSupernode` transformation set each supernode feature to [1], save a mask that tells which nodes are supernode S and a edge mask that tells if the edge contains one supernode $edge\_S$
2. During the forward pass of the model if the graph contains supernodes:
   (a) Execute the supernode convolution only with the edges $edge\_S$ and save it to X_2
   (b) Update the original data using the mask: $X[S] = X_2[S]$

This is the code in the `forward` method that we added to the model to perform the supernode embeddings operation:

Code (Model - Supernode Embeddings).

```
if supernode_mask is not None:
    # 0. compute supernode initial features
    x2 = self.supconv(x, edge_index, edge_mask)
    x[supernode_mask] = x2[supernode_mask]
```

Remark. Code of the various models is in the folder `supernode/models`

3.1 GCN

Definition 3.1 (Model - GCN). The GCN model is built in the following way:

0. Supernode embeddings: SimpleConv(add)
1. Node embeddings:
   (a) GCNConv + relu
   (b) GCNConv + relu
   (c) GCNConv + relu
2. Readout: global_mean_pool
3. Classifier: Multi-Layer Perception

Code (Model - GCN).

```
class GCN(torch.nn.Module):
    def __init__(self, num_node_features, hidden_channels, num_classes):
        super(GCN, self).__init__()

        self.supconv = SimpleConv("add")
        self.conv1 = GCNConv(num_node_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, hidden_channels)
        self.mlp = MLP([hidden_channels, hidden_channels, num_classes],
                       norm=None, dropout=0.5)

    def forward(self, x, edge_index, supernode_mask, edge_mask, batch):
        if supernode_mask is not None:
            # 0. compute supernode initial features
            x2 = self.supconv(x, edge_index, edge_mask)
            x[supernode_mask] = x2[supernode_mask]

        # 1. Obtain node embeddings
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = self.conv3(x, edge_index)

        # 2. Readout layer
        x = global_add_pool(x, batch)

        # 3. Apply a final classifier
```

```
        return self.mlp(x)
```

Definition 3.2 (MessagePassing - GCNConv)**.** The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" paper. Its node-wise formulation is given by:

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j.$$

With $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$, where $e_{j,i}$ denotes the edge weight from source node j to the target node i.

3.2 GIN

Definition 3.3 (Model - GIN)**.** The GIN model is built in the following way:

0. Supernode embeddings: SimpleConv(add)
1. Node embeddings:
   (a) GINConv(MLP) + relu
   (b) GINConv(MLP) + relu
   (c) GINConv(MLP) + relu
2. Readout: global_add_pool
3. Classifier: Multi-Layer Perception

Code (Model - GCN)**.**

```
class GIN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, num_classes, num_layers):
        super().__init__()

        self.supconv = SimpleConv("add")
        self.convs = torch.nn.ModuleList()
        for _ in range(num_layers):
            mlp = MLP([in_channels, hidden_channels, hidden_channels])
            self.convs.append(GINConv(nn=mlp, train_eps=False))
            in_channels = hidden_channels

        self.mlp = MLP([hidden_channels, hidden_channels, num_classes],
                       norm=None, dropout=0.5)

    def forward(self, x, edge_index, supernode_mask, edge_mask, batch):
        if supernode_mask is not None:
            # 0. compute supernode initial features
            x2 = self.supconv(x, edge_index, edge_mask)
            x[supernode_mask] = x2[supernode_mask]

        # 1. Obtain node embeddings
        for conv in self.convs:
            x = conv(x, edge_index).relu()

        # 2. Readout layer
        x = global_add_pool(x, batch)

        # 3. Apply a final classifier
        return self.mlp(x)
```

Definition 3.4 (MessagePassing - GINConv)**.** The graph isomorphism operator from the "How Powerful are Graph Neural Networks?" paper.

$$\mathbf{x}_i' = h_{\mathbf{\Theta}}\left((1+\varepsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right).$$

here $h_{\mathbf{\Theta}}$ denotes a neural network, .i.e. an MLP.

4 Experiments

4.0.1 Cycle Supernode

The goal of this experiment is to use the `Dataset_tree_cycle`, which assigns the label to the graph based on whether it contains a cycle, and test if adding the supernodes in particular the one corresponding to the `cycle_basis` concept help the models.

Dataset				
type	graph number	proportions	node number	cycle level
Dataset_tree_cycle	10000	0.5	40	10

Models				
model name	supernode embeddings	node embeddings	readout	classifier
GCN1	SimpleConv(Add)	3 * (GCNConv(32) + relu)	global_add_pool	MLP(3L,32)
GIN1	SimpleConv(Add)	3 * (GINConv(MLP,32) + relu)	global_add_pool	MLP(3L,32)

For training all the models we used as criterion `CrossEntropyLoss` and as optimizer `Adam`, furthermore the dataloader uses a `batch_size` of 60 graphs.

Results				
model	concepts	number of epoch	vanilla test accuracy	supernodes test accuracy
GCN1	cycle_basis, max_cliques, line_path(2)	10	0.5000	1.0000
GIN1	cycle_basis, max_cliques, line_path(2)	10	1.0000	0.9995

5 Definitions

In this section there are some useful definitions that we used previously.

5.0.1 MessagePassing

Definition 5.1 (MessagePassing)**.** Message passing layers follow the form

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}}\left(\mathbf{x}_i, \bigoplus_{j\in\mathcal{N}(i)}\phi_{\mathbf{\Theta}}\left(\mathbf{x}_i,\mathbf{x}_j,\mathbf{e}_{j,i}\right)\right),$$

where $\bigoplus$ denotes a differentiable, permutation invariant function, e.g., sum, mean, min, max or mul, and $\gamma_{\mathbf{\Theta}}$ and $\phi_{\mathbf{\Theta}}$ denote differentiable functions such as MLPs

Definition 5.2 (MessagePassing - SimpleConv)**.** A simple message passing operator that performs (non-trainable) propagation.

$$\mathbf{x}'_i = \bigoplus_{j \in \mathcal{N}(i)} e_{ji} \cdot \mathbf{x}_j.$$

where $\bigoplus$ defines a custom aggregation scheme (eg: `add`, `sum`, `mean`, `min`, `max`, `mul`)

5.0.2 Activation

Definition 5.3 (Activation - relu)**.** Applies the rectified linear unit function element-wise:

$$ReLU(x) = (x)^+ = max(0, x).$$

5.0.3 Aggregation / Pooling

Definition 5.4 (Aggregation - global_mean_pool)**.** Returns batch-wise graph-level-outputs by averaging node features across the node dimension. For a single graph, its output is computed by

$$\mathbf{r}_i = \frac{1}{N_i} \sum_{n=1}^{N_i} \mathbf{x}_n.$$

Definition 5.5 (Aggregation - global_add_pool)**.** Returns batch-wise graph-level-outputs by averaging node features across the node dimension. For a single graph, its output is computed by

$$\mathbf{r}_i = \sum_{n=1}^{N_i} \mathbf{x}_n.$$