Graph Convolutional Networks

Intro to DL

SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS

Thomas N. Kipf
University of Amsterdam
T.N.Kipf@uva.nl

Max Welling
University of Amsterdam
Canadian Institute for Advanced Research (CIFAR)
M.Welling@uva.nl

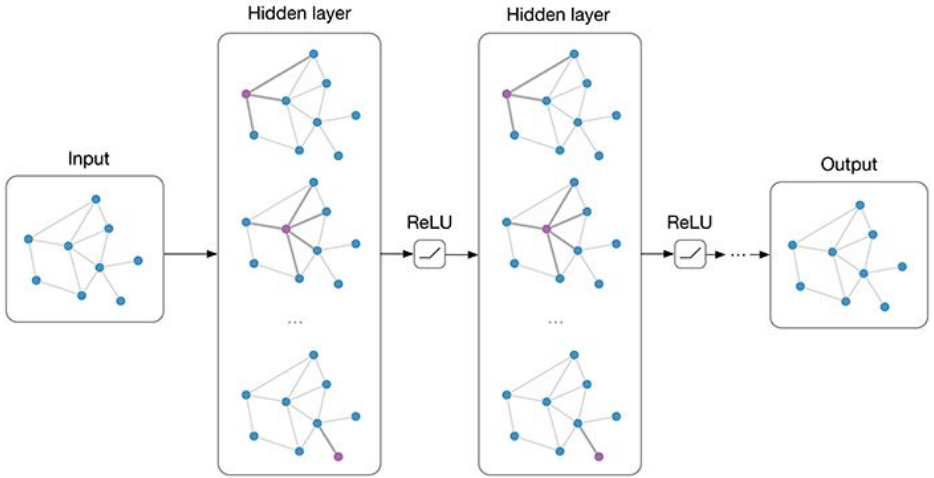
ABSTRACT

We present a scalable approach for semi-supervised learning on graph-structured data that is based on an efficient variant of convolutional neural networks which operate directly on graphs. We motivate the choice of our convolutional architecture via a localized first-order approximation of spectral graph convolutions. Our model scales linearly in the number of graph edges and learns hidden layer representations that encode both local graph structure and features of nodes. In a number of experiments on citation networks and on a knowledge graph dataset we demonstrate that our approach outperforms related methods by a significant margin.

Convolution!!

The core idea of GCN is to generate new node features by combining the features (or signals) of a node with those of its neighboring nodes through a process known as "graph convolution."

This process allows nodes to aggregate information from their neighbors, facilitating learning based on local and global graph structures.



https://tkipf.github.io/graph-convolutional-networks/

Convolution!!

For example, in a social network, when predicting the behavior of a user, GCN not only considers the user's data but also incorporates data from their friends. This approach is utilized in various fields, such as predicting molecular structures, analyzing social networks, and enhancing recommendation systems.

GCN

- The goal is then to learn a function of signals/features on a graph G=(V,E) which takes as input:
  - A feature description xi for every node i; summarized in a N×D feature matrix X (N: number of nodes, D: number of input features)
  - A representative description of the graph structure in matrix form; typically in the form of an adjacency matrix A
- and produces a node-level output Z (an N×F feature matrix, where F is the number of output features per node).

GCN

Every neural network layer can then be written as a non-linear function

$$H^{(l+1)} = f(H^{(l)}, A)\,,$$

with $H^{(0)}$=X and $H^{(L)}$=Z (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\cdot,\cdot)$ is chosen and parameterized.

Example

As an example, let's consider the following very simple form of a layer-wise propagation rule:

$$f(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right)$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU.

Two limitations

1. Self-loop
2. Normalization

But first, let us address two limitations of this simple model: multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by enforcing self-loops in the graph: we simply add the identity matrix to A.

The second major limitation is that A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors (we can understand that by looking at the eigenvalues of A). Normalizing A such that all rows sum to one, i.e. $D^{-1}A$, where D is the diagonal node degree matrix, gets rid of this problem. Multiplying with $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ (as this no longer amounts to mere averaging of neighboring nodes). Combining these two tricks, we essentially arrive at the propagation rule introduced in Kipf & Welling (ICLR 2017):

GCN: Propagation Rule

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

with A^=A+I, where I is the identity matrix and D^ is the diagonal node degree matrix of A^

.

Implementation 1

Node Classification with CiteSeer data

```
import torch
from torch_geometric.datasets import Planetoid
import torch_geometric.transforms as T
from torch_geometric.nn import GCNConv

dataset = Planetoid(root='/tmp/CiteSeer', name='CiteSeer', transform=T.NormalizeFeatures())
data = dataset[0]
```

In PyTorch Geometric, the syntax `dataset[0]` is used to access a dataset when you load it. In this context, the `dataset` object typically contains graph data structures, and `dataset[0]` refers to the first graph in the dataset. For datasets like the `Planetoid` dataset used in PyTorch Geometric, there is actually only one graph, so calling `dataset[0]` retrieves the entire graph data.

Planetoid datasets

https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.datasets.Planetoid.html

Name	#nodes	#edges	#features	#classes
Cora	2,708	10,556	1,433	7
CiteSeer	3,327	9,104	3,703	6
PubMed	19,717	88,648	500	3

Node Classification with CiteSeer data

```
class GCN(torch.nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = torch.dropout(x, p=0.5, train=self.training)
        x = self.conv2(x, edge_index)
        return torch.log_softmax(x, dim=1)
```

Node Classification with CiteSeer data

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN().to(device)
data = data.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = torch.nn.functional.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
```

Node Classification with CiteSeer data

```
model.eval()
pred = model(data.x, data.edge_index).argmax(dim=1)
correct = (pred[data.test_mask] == data.y[data.test_mask]).sum()
acc = int(correct) / int(data.test_mask.sum())
print(f'Accuracy: {acc:.4f}')
```

```
loss = torch.nn.functional.nll_loss(out[data.train_mask], data.y[data.train_mask])
```

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

input = torch.randn(3, 5)  # batch = 3, class = 5
print(input)
target = torch.tensor([1, 0, 4])  # target label
print(target)
# LogSoftmax
log_probs = F.log_softmax(input, dim=1)
print(log_probs)
# NLL Loss
loss = F.nll_loss(log_probs, target)
print(loss)
```

```
tensor([[ 0.0239,  0.8181,  0.1691, -0.0136, -0.9268],
        [ 1.0860, -0.2967,  0.6897, -0.6957,  1.9384],
        [ 0.9760,  0.3759, -1.0396,  0.7030,  0.4365]])
tensor([1, 0, 4])
tensor([[-1.7438, -0.9495, -1.5985, -1.7812, -2.6944],
        [-1.4901, -2.8727, -1.8863, -3.2717, -0.6377],
        [-1.1073, -1.7074, -3.1229, -1.3803, -1.6468]])
tensor(1.3621)
```

Negative Log Likelihood Loss

takes log probabilities and actual class labels as inputs. The log probabilities are typically calculated using the LogSoftmax function. The actual labels are represented as integer indices, each indicating the class of the respective data point.

The loss is calculated using the following formula:

$$\text{Loss} = -\sum_{i=1}^{N} \log(p_{i,y_i})$$

Here, P_{i,y_i}represents the log probability that the model assigns to the true class y_i of the i-th data point.

Implementation 2

Graph Classification with MUTAG data

- Data Description
  - https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.datasets.TUDataset.html
  - The MUTAG dataset consists of **chemical compounds** represented as graphs, where nodes represent atoms and edges represent chemical bonds. Each graph is labeled according to its mutagenic effect on a bacterium. This dataset is commonly used for demonstrating graph classification tasks, where the objective is to predict whether a chemical compound has mutagenic properties.

Name	#graphs	#nodes	#edges	#features	#classes
MUTAG	188	~17.9	~39.6	7	2
ENZYMES	600	~32.6	~124.3	3	6
PROTEINS	1,113	~39.1	~145.6	3	2
COLLAB	5,000	~74.5	~4914.4	0	3
IMDB-BINARY	1,000	~19.8	~193.1	0	2
REDDIT-BINARY	2,000	~429.6	~995.5	0	2

Graph Classification with MUTAG data

```
import torch
import torch.nn.functional as F
from torch_geometric.datasets import TUDataset
from torch_geometric.data import DataLoader
from torch_geometric.nn import GCNConv, global_mean_pool

# Load a graph dataset (e.g., MUTAG) from TUDataset
dataset = TUDataset(root='/tmp/MUTAG', name='MUTAG')
dataset = dataset.shuffle()

# Split the dataset into training and testing sets
train_dataset = dataset[:150]
test_dataset = dataset[150:]

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

```
class GCN(torch.nn.Module):
    def __init__(self, hidden_channels):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(dataset.num_node_features, hidden_channels) #64
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.out = torch.nn.Linear(hidden_channels, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        # First graph convolution layer
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=0.5, training=self.training)

        # Second graph convolution layer
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=0.5, training=self.training)

        # Global mean pooling to aggregate graph-level features
        x = global_mean_pool(x, batch) # pooling to a vector

        # Fully connected layer for class prediction
        x = F.log_softmax(self.out(x), dim=1)
        return x
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN(hidden_channels=64).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

def train():
    model.train()
    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data.x, data.edge_index, data.batch)
        loss = F.nll_loss(out, data.y)
        loss.backward()
        optimizer.step()

def test(loader):
    model.eval()
    correct = 0
    for data in loader:
        data = data.to(device)
        out = model(data.x, data.edge_index, data.batch)
        pred = out.argmax(dim=1)
        correct += int((pred == data.y).sum())
    return correct / len(loader.dataset)
```

Graph Classification with MUTAG data

```
# Training and evaluation
for epoch in range(1, 201):
    train()
    train_acc = test(train_loader)
    test_acc = test(test_loader)
    if epoch % 10 == 0:
        print(f'Epoch: {epoch:03d}, Train Acc: {train_acc:.4f}, Test Acc: {test_acc:.4f}')
```

Graph Pooling

```python
import torch
from torch_geometric.nn import global_mean_pool

# Example node feature matrix
# Assume each node has 3 features and there are 8 nodes in total
x = torch.tensor([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12],
    [13, 14, 15],
    [16, 17, 18],
    [19, 20, 21],
    [22, 23, 24]
], dtype=torch.float)

# batch tensor, indicating the first 4 nodes belong to one graph, and the next 4 to another
batch = torch.tensor([0, 0, 0, 0, 1, 1, 1, 1])

# Apply global_mean_pool
global_features = global_mean_pool(x, batch)

print(global_features)
```

```
tensor([[ 5.5000,  6.5000,  7.5000],
        [17.5000, 18.5000, 19.5000]])
```