Graph Mining - CSF426

Lab Session - 11

Date - 26/11/2024

Marks - 10

Instructor IC - Vinti Agarwal

Lab_starter

Graph Covolutional Neural Network using PyTorch

```
!pip install torch_geometric --quiet
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 63.1/63.1 kB 581.4 kB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.1/1.1 MB 9.1 MB/s eta 0:00:00
```

+ Code + Text

1. Packages:

Let's import all the packages we would need during programming

```
import numpy as np
import scipy.sparse as sp
import torch
torch.__version__
```

```
'2.5.1+cu121'
```

```
import math
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from torch_geometric.datasets import Planetoid
%matplotlib inline
```

2. One hot encoding

Converting all the class labels into one hot encoding

e.g. Three class Problem: labels = {"apple", "mango", "tomato"}

1-hot encoding will be:

apple =[1, 0, 0] mango =[0, 1, 0] tomato =[0, 0, 1]

```
# one-hot encode a list
def encode_onehot(labels):
  classes = set(labels)
  classes_dict = {c : np.identity(len(classes))[i, :] for i, c in enumerate(classes)}
  labels_oh = np.array(list(map(classes_dict.get, labels)), dtype=np.int32)
  return labels_oh
```

```
# an example
encode_onehot(["apple", "mango", "tomato"])
```

```
array([[0, 1, 0],
       [0, 0, 1],
       [1, 0, 0]], dtype=int32)
```

3. The Dataset (Cora)

The Cora dataset consists of ML-based research papers, classified into one of the following seven classes:

- Case_Based
- Genetic_Algorithms
- Neural_Networks
- Probabilistic_Methods
- Reinforcement_Learning
- Rule_Learning
- Theory

The papers were selected in a way such that in the final corpus every paper cites or is cited by atleast one other paper. There are 2708 papers in the whole corpus.

(This means the graph is a fully spanning graph with 2708 reachable nodes)

After stemming and removing stopwords we were left with a vocabulary of size 1433 unique words.

1. `cora.content` contains descriptions of the papers in the following format:

```
<paper_id> <word_attributes> <class_label>
e.g. array(['1061127', '0', '0', ..., '0', '0', 'Rule_Learning'])
```

The first entry in each line contains the unique string ID of the paper followed by binary values indicating whether each word in the vocabulary is present (indicated by 1) or absent (indicated by 0) in the paper. Finally, the last entry in the line contains the class label of the paper.

`cora.cites` contains the citation graph of the corpus. Each line describes a link in the following format:

```
<ID of cited paper> <ID of citing paper>
 e.g. array([ 35, 103482]
```

Each line contains two paper IDs. The first entry is the ID of the paper being cited and the second ID stands for the paper which contains the citation. The direction of the link is from right to left. If a line is represented by "paper1 paper2" then the link is "paper2->paper1".

4. Preprocess adjacency matrix

4.1 Normalize Adjacency Matrix - (TODO)

Implement $D^{-1}.A$ normalization

```
def normalize1(adj):
```

Implement $D^{-1/2}.A.D^{-1/2}$ normalization

```
def normalize2(adj):
```

4.2 Transform Adjacecy matrix: from scipy.sparse to torch.sparse

```
def sparse_mx_to_torch_sparse_tensor(sparse_mx):
  sparse_mx = sparse_mx.tocoo().astype(np.float32)
  indices = torch.from_numpy(np.vstack((sparse_mx.row, sparse_mx.col)).astype(np.int64))
  values = torch.from_numpy(sparse_mx.data)
  shape = torch.Size(sparse_mx.shape)
  return torch.sparse.FloatTensor(indices, values, shape)
```

5. Load Data (TODO)

```
def load_cora_data():
  dataset = Planetoid(root='/tmp/Cora', name='Cora')
  data = dataset[0]
  adj = sp.coo_matrix((torch.ones(data.edge_index.size(1)), (data.edge_index[0], data.edge_index[1])), shape=(data.x.shape[0], data.x.sh
  adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)  # Make it undirected and remove duplicate edges
  adj= ____________________________    # add self loop
  adj =_____________________________ # normalize adjacency matrix
  adj = sparse_mx_to_torch_sparse_tensor(adj)
```

```
  features = data.x
  # features = torch.FloatTensor(np.array(features.todense()))
  labels = data.y
  # labels = torch.LongTensor(np.where(labels)[1])
  idx_train = np.where(data.train_mask.numpy())[0]
  idx_val = np.where(data.val_mask.numpy())[0]
  idx_test = np.where(data.test_mask.numpy())[0]

  return adj, features, labels, idx_train, idx_val, idx_test
```

5.1 Check the size of input and output (TO-DO)

```
adj, features, labels, idx_train, idx_val, idx_test = load_cora_data()
print ("adjacency matrix size =", _______________ )
print ("The number of nodes n =", _______________ )
print ("The number of features n_f =", _________________)
print ("The list of node labels n_l=", __________________ )
```

Expected output:

adjacency matrix size = torch.Size([2708, 2708])
The number of nodes n = 2708
The number of features n_f = 1433
The list of node labels n_l= tensor([0, 1, 2, 3, 4, 5, 6])

5.2 Check the size of Train, Test and Validation set (TO-DO)

```
print ("Training set size =", _______________)
print ("Testing set size =", _______________)
print ("Validation set size =", _____________)
```

```
Training set size = 140
Testing set size = 1000
Validation set size = 500
```

Expected output:

Training set size = 140
Testing set size = 1000
Validation set size = 500

6. GRAPH CONVOLUTIONAL NETWORK (GCN) for Node Classification

6.1 Notations:

For a graph G = (V, E), Where V is set of vertices and E is set of edges

n is the number of nodes in the graph.

n_f is the number of features for each node.

X is the Node-Feature matrix with size $n \times n_f$, where each row corresponds to a feature vector of a node.

A is the Adjacency matrix of size $n \times n$.

Z is the Node-Label matrix of size $n \times n_l$, where each row is a one-hot encoded label vector for each node.

For GCN, the goal is to learn a function $f : X \rightarrow Z$ on graph G, which takes :

(1) INPUT: node-feature matrix X

(2) and produces OUTPUT: node-level matrix Z.

6.2 Layer-wise propagation rule in GCN

For GCN, the output for $l + 1^{th}$ layer is defined as:

$H^{(l+1)} = g(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)} + b^{(l)})$

where W^l and $b^{(l)}$ are weight and bias matrices, σ is a non-linear function. For the input layer, i.e., $l = 0$, we have $H^{(0)} = X$ and the output from the last layer, i.e., $l = L$, we have $H^{(L)} = Z$.

In this assignment, let's consider the following form of a **layer-wise propagation rule**:

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

where σ is a non-linear function

6.3 Second order GCN model,

ONE input layer, TWO hidden layer, ONE output layer are defined :

$H^{(0)} = X$ **(Output Size =** $n * n_f$ **)**

$H^{(1)} = f(H^{(0)}, A) = ReLu(AH^{(0)}W^{(0)} + b^0)$ **#Output** $H^{(1)}$ **Size :** $(n * n)(n * n_f)(n_f * n_h) + (n_h * 1) = (n * n_h)$

$H^{(2)} = f(H^{(l)}, A) = ReLu(AH^{(1)}W^{(1)} + b^1)$ **# Output** $H^{(2)}$ **Size :** $(n * n)(n * n_h)(n_h * n_h) + (n_h * 1) = (n_h * n_h)$

$Z = f(H^{(2)}, A) = softmax(AH^{(2)}W^{(2)} + b^2)$ **#Output** Z **Size :** $(n * n)(n * n_h)(n_h * n_l) + (n_l * 1) = (n * n_l)$

6.4 Write code to define GraphConv layer of GCN from torch.nn.module

STEPS INVOLVED ARE:

1. Define the size of weight matrices $W^{(0)}, W^{(1)}, W^{(2)}$ and biases $b^{(0)}, b^{(1)}, b^{(2)}$.
2. Initialize weights and bias values.
3. Define forward propagation rule.

(TODO)

```
class GraphConvolution(nn.Module):

    def __init__(self, in_features, out_features, bias=True):
        super(GraphConvolution, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = nn.Parameter(torch.FloatTensor(in_features, out_features))
        if bias:
            self.bias = nn.Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, input, adj):
        support = torch.mm(_____________, ___________)
        output = torch.spmm(_________________, _______________)
        if self.bias is not None:
            return output + self.bias
        else:
            return output

    def __repr__(self):
        return self.__class__.__name__ + ' (' \
               + str(self.in_features) + ' -> ' \
               + str(self.out_features) + ')'
```

6.5 Test parameter setting for GraphConv Layer (TO-DO)

GraphConv(5, 4)

```
SEED = 123
torch.manual_seed(SEED)
np.random.seed(SEED)
```

```
test_layer = GraphConvolution(5, 4)
print("Weight Matrix", _________________)
print("Bias", _____________________)
print(test_layer.in_features,test_layer.out_features)
```

```
Weight Matrix Parameter containing:
tensor([[-0.2039,  0.0166, -0.2483,  0.1886],
        [-0.4260,  0.3665, -0.3634, -0.3975],
        [-0.3159,  0.2264, -0.1847,  0.1871],
        [-0.4244, -0.3034, -0.1836, -0.0983],
        [-0.3814,  0.3274, -0.1179,  0.1605]], requires_grad=True)
Bias Parameter containing:
tensor([0.3536, 0.0932, 0.1367, 0.4826], requires_grad=True)
5 4
```

6.6 Design two-layer GCN (TO-DO)

```
class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout):
        super(GCN, self).__init__()

        self.gc1 = GraphConvolution(__________, ________)
        self.gc2 = GraphConvolution(__________, ________)
        self.dropout = dropout

    def forward(self, x, adj):
        x1 = F.relu(self.gc1(_________, __________))
        x2 = F.dropout(_____________, self.dropout, training=self.training)
        x3 = self.gc2(_____________, _____________)
        return F.log_softmax(_____________, dim=1)
```

9. Parameter setting and Model(VanillaGCN) Initialization

```
lr = [0.001,0.01]  # Learning rate
epochs = 200
wd = [5e-4,5e-03]
hidden = [8,16]  # Size of hidden layer
dropout = [0.1,0.3,0.5]
fastmode = False
```

(TODO)

```
def intialize_model(nfeat, hidden, c, dropout, lr, wd):
  # Model and optimizer
  model = GCN(nfeat=____________________,
              nhid=____________________,
              nclass=_________________,
              dropout=_________________)
  optimizer = optim.Adam(model.parameters(),
                         lr=____________, weight_decay=_____________)


  # # Print model's state_dict
  # print("Model's state_dict:")
  # for param_tensor in model.state_dict():
  #     print(param_tensor, "\t", model.state_dict()[param_tensor].size())

  # # Print optimizer's state_dict
  # print("Optimizer's state_dict:")
  # for var_name in optimizer.state_dict():
  #     print(var_name, "\t", optimizer.state_dict()[var_name])

  return model, optimizer
```

10. Accuracy, Training, Testing Functions

```
def accuracy(output, labels):
  preds   = output.max(1)[1].type_as(labels)
  correct = preds.eq(labels).double()
  correct = correct.sum()
  return correct / len(labels)
```

(TODO)

```
def train(epoch, final_train=False):

    best_acc = 0
    t = time.time()

  ## write code to train the model and compute train-loss and train-accuracy
  ## Then write code to evaluate the model and compute val-loss and val-accuracy

    if acc_val > best_acc:
      best_acc = acc_val
    if final_train:
      if epoch % 10==0:
        print('Epoch: {:04d}'.format(epoch+1),
              'loss_train: {:.4f}'.format(loss_train.item()),
              'acc_train: {:.4f}'.format(acc_train.item()),
              'loss_val: {:.4f}'.format(loss_val.item()),
              'acc_val: {:.4f}'.format(acc_val.item()),
              'time: {:.4f}s'.format(time.time() - t))

    return loss_train.item(), loss_val.item(), best_acc.item()


def test():
    ## write code to check the performance of model on test data. Compute test-loss and test accuracy
    print("Test set results:",
          "loss= {:.4f}".format(loss_test.item()),
          "accuracy= {:.4f}".format(acc_test.item()))
```

11. Running GCN model (TODO)

Use validation data to find the best set of hyperparameters for training the model.

```
import time

# Initialize tracking variables
t_total = time.time()
max_val_acc = 0.0
best_hyperparameters = None

print("Training GCN\n")

# Iterate over all hyperparameter combinations and store best hyperparameters




print(f"Best val_acc for this set of hyperparameters: {best_val_acc_this_set}\n")

print("Optimization Finished!")
print(f"Best Validation Accuracy over all hyperparameters: {max_val_acc}")
print(f"Best Hyperparameters: LR={best_hyperparameters[0]}, Weight Decay={best_hyperparameters[1]}, Hidden Size={best_hyperparameters[2]
print(f"Total time elapsed: {time.time() - t_total:.4f}s")
```

Re-run GCN on best hyperparameters (TODO)

```
lr , wd, hidden, dropout = best_hyperparameters
model, optimizer = intialize_model(features.shape[1], hidden, labels.max().item() + 1, dropout, lr, wd)

### Retrain on best hyperparameters

    # Update the best accuracy
    if val_acc > best_val_acc_this_set:
        best_val_acc_this_set = val_acc

print(f"Best val_acc for this set: {best_val_acc_this_set}\n")
```

```
test()
```

```
Test set results: loss= 0.6173 accuracy= 0.8100
```

Expected output

Test set results: loss= 0.6173 accuracy= 0.8100

```
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.grid()
plt.xlabel('Epochs')
plt.ylabel('NLLLoss')
plt.legend()
```

(TODO)

Plot test accuracy achieved by $D^{-1}.A$ and $D^{-1/2}.A.D^{-1/2}$ adjacency normalization

```
Start coding or generate with AI.
```

```
#I hope you enjoyed the assignment 😊 Thank you!!
```