**Birla Institute of Technology & Science, Pilani**
**First Semester 2023-2024**
**Graph Mining (CS F426)**
**Comprehensive Exam-Part A**

| | | |
|---|---|---|
| Date | : Dec 13, 2023 | Duration: 90 minutes |
| Nature of Exam | : Closed Book | Maximum Marks: 60 |

Answer all parts together, separate them by line.

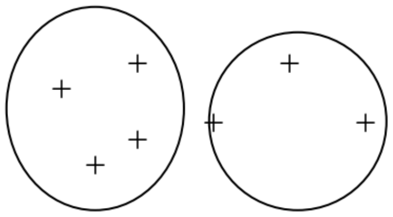
---

**Q1.** a) Outline a high-level algorithm for spectral graph partitioning using the Laplacian matrix and its eigenvectors. **[4 marks]**

b) Let us consider an example graph with six nodes 1, 2, 3, 4, 5 and 6, with the corresponding eigenvalues and eigenvectors as shown in the table. Explain the possible graph partitions can be obtained using second and third eigenvectors alone and in combination. **[4.5 marks]**

| **Eigenvalue** | **0** | **1** | **3** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **Eigenvector** | 1 | 1 | -5 | -1 | -1 | -1 |
| | 1 | 2 | 4 | -2 | 1 | 0 |
| | 1 | 1 | 1 | 3 | -1 | 1 |
| | 1 | -1 | -5 | -1 | 1 | 1 |
| | 1 | -2 | 4 | -2 | -1 | 0 |
| | 1 | -1 | 1 | 3 | 1 | -1 |

**Q2.** Are the two clusters shown below well separated? Yes or No? Justify your answer. **[3 marks]**



**Q3.** Suppose you have a multi-relational knowledge graph with 1000 nodes and 200 relation types. The objective is to learn the embedding of nodes to solve final downstream tasks such as node classification, link prediction, etc. Answer the following: **[6+6=12 marks]**

a) Propose a shallow and deep learning architecture that are suitable for solving the tasks. Mention input, output and objective functions clearly.

b) Identify the number of parameters needed to learn for each of these models separately.

**Q 4.** Suppose we wish to store an n × n boolean matrix (0 and 1 elements only). We could represent it by the bits themselves, or we could represent the matrix by listing the positions of the 1's as pairs of integers, each integer requiring [log2 *n*] bits. The former is suitable for dense matrices; the latter is suitable for sparse matrices. Based on this answer following: **[2+2+4=8 marks]**

a) What is the total number of bits required to store an n × n matrix in dense representation?

b) How many bits are required to save each 1 in sparse representation?

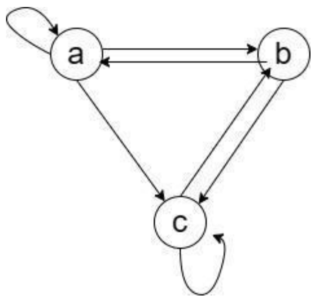
c) How sparse must the matrix be (i.e., what fraction of the elements should be 1's) for the sparse representation to save space?

**Q 5.** For the given graph with three nodes a, b and c, compute the following: **[2+3+4=9 marks]**

a. Adjacency matrix, initial PageRank vector

b. Pagerank of each page assuming no taxation

c. Pagerank of each page assuming $\beta = 0.8$



**Q.6** For the given data, if the graph structure is not known in advance, which of the GNN architecture is advisable to solve the node classification task? Also explain why the proposed architecture is suitable? **[2+2 =4 marks]**

**Q7.** Assume that in a Graph attention network with two layers, input at the first layer is a set of node features h = {$h_1, h_2, \ldots, h_N$}, $h_i \in R^F$, where N is the number of nodes, and F is the number of features in each node. The second layer produces a transformed set of node features h` = {$h`_1, h`_2, \ldots, h`_N$ }, $h`_i \in R^{F`}$, as its output. After that, to perform self-attention on each pair of nodes (as shown in Fig below), we first compute attention coefficients $e_{ij}$ , which indicates the importance of the jth neighbor of node i. Then $e_{ij}$ passes through the softmax function to obtain final self-attention weights $\alpha_{ij}$ between every pair of nodes $i$ and $j$ using the below equations:

$$e_{ij} = a(Wh_i, Wh_j) \quad \text{and} \quad \alpha_{ij} = softmax(e_{ij})$$

The final output feature h` for every node is computed as below:

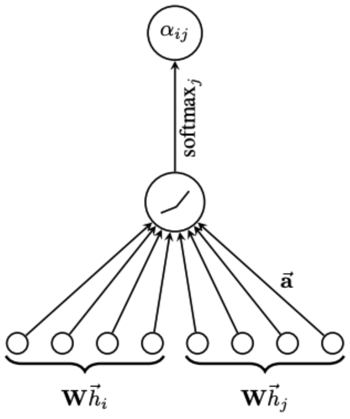
$$h`_i = \sigma(\sum_{j=1}^{N_i} \alpha_{ij} W h_j)$$

Answer the following questions: **[2+2+4 =8 marks]**

a) What will be the size of the weight-parameterized matrix W used for feature transformation from h to h`?

b) What will be the size of the self-attention vector $a$ used to compute attention coefficients $e_{ij}$.

c) If the graph is large, computing $e_{ij}$ for every pair of nodes will be computationally expensive. What are the possible optimal ways you suggest learning self-attention so that a desirable graph structure is obtained?



**Q8.** Suppose we have a model that classifies images into three classes: "Cat," "Dog," and "Bird" and the following results were obtained from the model on the test dataset.

| Cat: | Dog: | Bird: |
|---|---|---|
| TP_Cat = 90<br>FP_Cat = 10<br>TN_Cat = 800<br>FN_Cat = 20 | TP_Dog = 80<br>FP_Dog = 20<br>TN_Dog = 900<br>FN_Dog = 10 | TP_Bird = 70<br>FP_Bird = 30<br>TN_Bird = 850<br>FN_Bird = 50 |

Fill in the blanks provided in the table. **[7.5 marks]**

| | **Precision** | **Recall** | **F1-score** |
|---|---|---|---|
| ***cat*** | __ | __ | __ |
| ***dog*** | __ | __ | __ |
| **bird** | __ | __ | __ |
| **Macro avg** | __ | __ | __ |
| **Micro avg** | __ | __ | __ |

*******************************END***********************************

## Graph Mining - CSF426

Comprehensive Exam 2023: Part-B.1

Duration: 45 mins

Total Marks: 20

Open book

Instructions:

1. Rename code file with your "firstname and student id".
2. Students are required to fill the blanks with code syntax in the sections provided below TO-DO comments.

Objective: This programming exercise is designed to perform node classification task on cora dataset.

```
import networkx as nx
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

## 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".

## (TO-DO) (Load dataset after changing the file location) [1 mark]

```
cora_citations = pd.read_csv(
    "/content/cora.cites",
    sep="\t",
    header=None,
    names=["target", "source"],
)
```

```
cora_citations
```

```
      target   source
0         35     1033
1         35   103482
2         35   103515
3         35  1050679
4         35  1103960
...      ...      ...
5424  853116    19621
5425  853116   853155
5426  853118  1140289
5427  853155   853118
5428  954315  1155073
5429 rows × 2 columns
```

```
node_list = list(set(cora_citations['source'].unique()) | set(cora_citations['target'].unique()))
# node_list contains the uniques ids of all papers
```

```
# Create a directed graph using NetworkX
G = nx.from_pandas_edgelist(cora_citations, 'source', 'target', create_using=nx.DiGraph())
```

## (TO-DO) Create adjacency matrix using the node order provided in node_list [2 marks]

```
Adj_mat = ______________________ #write your code here
Adj_mat.todense()
```

## Extract paper id and labels from cora_content into cora_labels with the same order in node_list

```
column_names = ["paper_id"] + [f"term_{idx}" for idx in range(1433)] + ["label"]
# cora_content = pd.read_csv("/content/cora.content",sep="\t", header=None, names = column_names)
cora_content = pd.read_csv("/content/drive/MyDrive/graph mining/Exam paper 2023/cora/cora.content",sep="\t", header=None, na
cora_labels = cora_content.iloc[:,[0, -1]]
cora_labels
```

```
      paper_id                   label
0        31336          Neural_Networks
1      1061127            Rule_Learning
2      1106406  Reinforcement_Learning
3        13195  Reinforcement_Learning
4        37879    Probabilistic_Methods
...        ...                     ...
2703   1128975       Genetic_Algorithms
2704   1128977       Genetic_Algorithms
2705   1128978       Genetic_Algorithms
2706    117328               Case_Based
2707     24043          Neural_Networks
2708 rows × 2 columns
```

## Map the class labels from "cora_labels" in the same sequence as in "node_list" to get Y

```
id_label_mapping = dict(zip(cora_labels['paper_id'], cora_labels['label']))
# Map IDs from the node list to labels using the dictionary
labels_for_nodes = [id_label_mapping.get(node_id) for node_id in node_list]
# Create a new DataFrame with 'ID' and 'Label' columns
```

```
Y = pd.DataFrame({'Paper_id': node_list, 'Label': labels_for_nodes})
Y
```

## TO-DO: - Print the count of the papers in each class [2 marks]

```
#display the count of the papers in each subject.
print(__________________)
```

```
Neural_Networks           818
Probabilistic_Methods     426
Genetic_Algorithms        418
Theory                    351
Case_Based                298
Reinforcement_Learning    217
Rule_Learning             180
Name: label, dtype: int64
```

## (TO-DO) - Print the top five most cited papers along with their citation count [2.5 marks]

```
#Find the top 5 most cited papers
Top5_paperID =_________________________________________________

# Print the top five paper ids and their citation count
for ______________:
    print(f"Paper with id {_______} has {__________} citations.")
```

```
Paper with id 35 has 166 citations.
Paper with id 6213 has 76 citations.
Paper with id 1365 has 74 citations.
Paper with id 3229 has 61 citations.
Paper with id 114 has 42 citations.
```

## (TO-DO) - Compute the node homophily for top 5 most cited papers [5 marks]

```
def node_homophily(adjacency_matrix, Top5_paperID):

    return homophily
```

## (TO-DO) - Convert labels into 1-hot encoding and store in y [1.5 marks]

```
#write your code
```

## Split the dataset into train, val and test sets in 60:20:20 ratio

```
num_nodes = cora_labels.shape[0]
train=0.15
val=0.15
idx_train = np.random.choice(range(num_nodes), int(train * num_nodes), replace=False)
idx_vt = list(set(range(num_nodes)) - set(idx_train))
idx_val = np.random.choice(idx_vt, int(val * num_nodes), replace=False)
idx_test = np.array(list(set(idx_vt) - set(idx_val)))
print(idx_train.shape, idx_val.shape,idx_test.shape)
```

Expected output:

(406,) (406,) (1896,)

## (TO-DO) [6 marks]

Perform the following steps for Label Prediction:

1. Apply iterative label spreding on train data labels (hide the labels of validation and test data) and use normalized adjacency
$$S = D^{-1/2}.W.D^{-1/2}$$
2. Tune alpha hyperparameter using validation data (by computing accuracy on validation data for each alpha).
3. Finally compute accuracy on the test data using the optimal alpha.

```
alpha_values =[0.001,0.01,0.02,0.05, 0.1, 0.2, 0.5]
iter = 50
# Normalize adjacency matrix using S = D-1/2AD-1/2

def LSA(y_train, Norm_Adj, alpha, iter)



  return y_hat

def alpha_tuning(  ):
# Iterate over possible alpha values to find the best one
    for alpha in alpha_values:
        ##write your code here

        # Compute accuracy on validation set

        ## write your code here

        print(f"alpha {alpha},validation_acc {validation_acc}")



    return optimum_alpha

#Compute accuracy on test data on optimal alpha value
# *******write your code here******
```

# Graph Mining - CSF426

## Comprehensive Exam 2023: Part-B.2

Duration: 45 mins

Total Marks: 20

Open book

Instructions:

1. Rename code file with your "firstname and student id".
2. Students are required to fill the blanks with code syntax in the sections provided below TO-DO comments.

### Objective:

This programming exercise is designed to perform node classification task using Graph Convolution Networks on cora dataset.

## Graph Covolutional Neural Network using PyTorch

## Load Packages:

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

numpy; scipy.sparse; torch

```
# sparse matrices have a different encoding that saves space (doesn't store lots
import numpy as np
import scipy.sparse as sp
import torch
torch.__version__
import math
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
%matplotlib inline
```

```
    '2.1.0+cu118'
```

## One hot encoding

Converting all the class labels into one hot encoding

```
# 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(class
  labels_oh = np.array(list(map(classes_dict.get, labels)), dtype=np.int32)
  return labels_oh
```

## The Dataset (Cora)

For dataset description, follow Part B.1

## Loading graph data in Tensors; Nodes, Nodes features, Node labels

```
def load_nodes(path, dataset):
  nodes = np.genfromtxt(f'{path}{dataset}.content', dtype=np.dtype(str)) # index,
  features = sp.csr_matrix(nodes[:, 1:-1], dtype=np.float32) # one-hot dictionary
  labels = encode_onehot(nodes[:, -1]) # the target to predict
  idxs = np.array(nodes[:, 0], dtype=np.int32) # the index of the papers (UID)
  return idxs, features, labels
```

## Compute Adjacency matrix

```
def load_adj(path, dataset, idxs_map, labels):
  edges_unordered = np.genfromtxt(f'{path}{dataset}.cites', dtype=np.int32)
  edges = np.array(list(map(idxs_map.get, edges_unordered.flatten())), dtype=np.i
  edges = edges.reshape(edges_unordered.shape)
  edges_t = (np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1]))
  adj = sp.coo_matrix(edges_t, shape=(labels.shape[0], labels.shape[0]), dtype=np
  adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
  return adj
```

## Normalize Adjacency Matrix

```
# normalize a matrix row-wise
def normalize(mx):
  rowsum = np.array(mx.sum(1))
  r_inv  = np.power(rowsum, -1).flatten()
  r_inv[np.isinf(r_inv)] = 0.
  r_mat_inv = sp.diags(r_inv)
  mx = r_mat_inv.dot(mx)
  return mx
```

## 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.
  values = torch.from_numpy(sparse_mx.data)
  shape = torch.Size(sparse_mx.shape)
  return torch.sparse.FloatTensor(indices, values, shape)
```

## Dataset split : Train, Test and Validation set

```
def train_val_test_split(num_nodes, train=0.15, val=0.15):
  idx_train = np.random.choice(range(num_nodes), int(train * num_nodes), replace=
  idx_vt = list(set(range(num_nodes)) - set(idx_train))
  idx_val = np.random.choice(idx_vt, int(val * num_nodes), replace=False)
  idx_test = list(set(idx_vt) - set(idx_val))
  idx_train = torch.LongTensor(idx_train)
  idx_val = torch.LongTensor(idx_val)
  idx_test = torch.LongTensor(idx_test)
  return idx_train, idx_val, idx_test
```

## (TO-DO): Setup the location to load data [1 mark]

```
def dataloader(path='####path of file', dataset='cora'):
  idxs, features, labels = load_nodes(path, dataset)  # Use load_nodes fuctions
  idxs_map = {j : i for i, j in enumerate(idxs)}
  adj = load_adj(path, dataset, idxs_map, labels)
  features = normalize(features)
  adj = normalize(adj + sp.eye(adj.shape[0]))                          # add self
  idx_train, idx_val, idx_test = train_val_test_split(adj.shape[0])
  features = torch.FloatTensor(np.array(features.todense()))
  labels = torch.LongTensor(np.where(labels)[1])
  adj = sparse_mx_to_torch_sparse_tensor(adj)
  return adj, features, labels, idx_train, idx_val, idx_test
```

## (TO-DO) Print the following parameters: [2 marks]

```
adj, features, labels, idx_train, idx_val, idx_test = dataloader()
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])
```

## (TO-DO)Print the size of Train, Test and Validation [1.5 marks] set

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

## GRAPH CONVOLUTIONAL NETWORK (GCN) for Node Classification

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 OUTPUT: node-level matrix Z.

## 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, $\sigma$ 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, \ie, $l = L$, 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 $\sigma$ is a non-linear function

## Second order GCN model,

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

image.png

$H^{(0)} = X$ **( 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 :** $(n * n)(n * n_h)(n_h * n_l) + (n_l * 1) = (n * n_l)$

## 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.

```
class GraphConv(nn.Module):

  def __init__(self, in_features, out_features):
    super(GraphConv, self).__init__()
    self.in_features  = in_features
    self.out_features = out_features
    self.weight = nn.Parameter(torch.Tensor(in_features, out_features))  # passin
    self.bias = nn.Parameter(torch.Tensor(out_features))  # passing number of bia
    self.reset_parameters()

    # WEIGHTS and BIAS initialization
  def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    self.weight.data.uniform_(-stdv, stdv)
    self.bias.data.uniform_(-stdv, stdv)

  def forward(self, input, adj):
    support = torch.mm(input, self.weight)  # input matrix is multiplied with wei
    output = torch.spmm(adj, support)     # permutation inv sum of all neighbor f
    return output + self.bias

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

## (TO-DO)Test parameter setting for GraphConv Layer [1.5 marks]

**GraphConv(5, 4)**

```
SEED = 42
torch.manual_seed(SEED)
np.random.seed(SEED)

test_layer = GraphConv(5, 4)
```

```
print("Weight Matrix", _____)
print("Bias", _____ )
```

Expected Output:**

Weight Matrix Parameter containing:

```
tensor([[ 0.3823, 0.4150, -0.1171, 0.4593],
        [-0.1096, 0.1009, -0.2434, 0.2936],
        [0.4408, -0.3668, 0.4346, 0.0936],
        [0.3694, 0.0677, 0.2411, -0.0706],
        [0.3854, 0.0739, -0.2334, 0.1274]], requires_grad=True)
```

Bias Parameter containing:

```
        tensor([ -0.2304, -0.0586, -0.2031, 0.3317], requires_grad=True)
```

## (TO-DO) Design two-layer GCN

The follwing GCN model is created with two hidden layers of same size $n\_h$. You are required to change the model with two hidden layers of different size $n\_h1 = 64$ and $n\_h2 = 16$.

## [6 marks]

```
class VanillaGCN(nn.Module):

  def __init__(self, n_f, n_h, n_l):
    super(VanillaGCN, self).__init__()
    self.gc1 = GraphConv(n_f, n_h)
    self.gc2 = GraphConv(___, ___)
    self.gc3 = GraphConv(___, ___)


  def forward(self, x, adj):
    H_1 = F.relu(self.gc1(x, adj))
    H_2 = F.relu(____________)
    Z = F.log_softmax(_________, dim=1)

    return Z
```

## (TO-DO):Parameter setting and Model(VanillaGCN) Initialization [3 marks]

```
lr = 0.01  # Learning rate
epochs = 300
```

```
wd = 5e-4
hidden = 16  # Size of hiddel layer
fastmode = False


model = VanillaGCN(n_f = features.shape[1], n_h = hidden, n_l = labels.max().item
optimizer = optim.Adam(model.parameters(), lr = lr, weight_decay = wd)   # initia

# 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])
```

Expected Output::

Model's state_dict:

```
gc1.weight      torch.Size([1433, 16])
gc1.bias      torch.Size([16])
gc2.weight      torch.Size([16, 16])
gc2.bias      torch.Size([16])
gc3.weight      torch.Size([16, 7])
gc3.bias      torch.Size([7])
```

Optimizer's state_dict:

```
state {}
param_groups [{'lr': 0.01, 'betas': (0.9, 0.999), 'eps': 1e-08, 'weight_decay': 0.0005, 'amsgrad': False,
'params': [0, 1, 2, 3, 4, 5]}]
```

## 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)
```

## (TO-DO) [3 marks]

```
def train(epoch):
  t = time.time()
  model.train()      # invoke training of model
  optimizer.zero_grad()
  output = model(features, adj)  # Sending input for Model training
  loss_train = F.nll_loss(output[idx_train], labels[idx_train])  # Loss on train
  acc_train = accuracy(_____ , ______)   # Accuracy on train data
  loss_train.backward()   # Gradient compute:  d(loss)/d(w),  d(loss)/d(b)
  optimizer.step()        # update weights

  # Model validation
  model.eval()
  output = model(features, adj)
  loss_val = F.nll_loss(output[idx_val], labels[idx_val])   # Loss on validation
  acc_val = accuracy(_____ , _______)      # Accuracy on validation data

  if epoch % 10 == 0:

      print('Epoch: {:04d}'.format(epoch),
        'loss_train: {:.4f}'.format(loss_train.item()),
        'loss_val: {:.4f}'.format(loss_val.item()),
        'acc_val: {:.4f}'.format(acc_val.item()))
  return loss_train.item(), loss_val.item()

def test():
  model.eval()    # invoke evaluation of model
  output = model(features, adj)
  loss_test = F.nll_loss(output[idx_test], labels[idx_test])
  acc_test = accuracy(_____ , _______)

  print("Test set results:",
        "loss= {:.4f}".format(loss_test.item()),
        "accuracy= {:.4f}".format(acc_test.item()))
```

## Running GCN model

### (TO-DO) Append train loss and validation loss for each epoch [2 marks]

```
import time
t_total = time.time()
train_losses, val_losses = [], []

print("Training GCN \n")

for epoch in range(epochs):
  loss_train, loss_val = train(epoch)
  train_losses.append(______)
```

```
    val_losses.append(       )
```