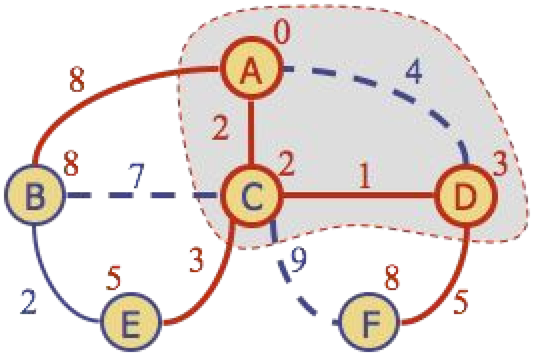
Neural Network for solving Shortest Path Problem

Shizhao Wang
Zhenye Na

Overview

- Introduction & Background
- Implementation (DNN)
- Implementation (GCN)
- Summary of results



Introduction & Background

The shortest path problem is concerned with finding the shortest path from a specified starting node (origin) to a specified ending node (destination) in a given network while minimizing the total cost associated with the path.

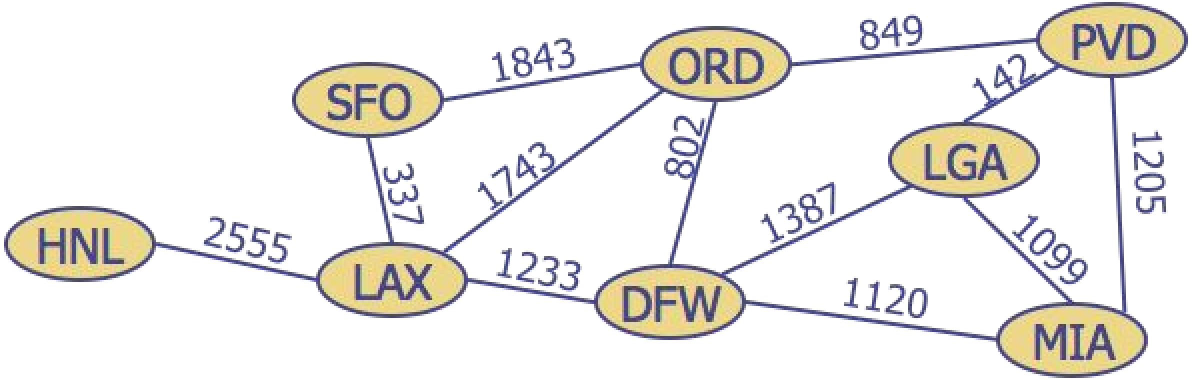
The applications of the shortest path problem include

- Vehicle routing in transportation systems.
- Traffic routing in telecommunication networks.
- Path planning in robotic systems.

Furthermore, the shortest path problem also has numerous variations such as the minimum weight problem, the quickest path problem, the most reliable path problem.

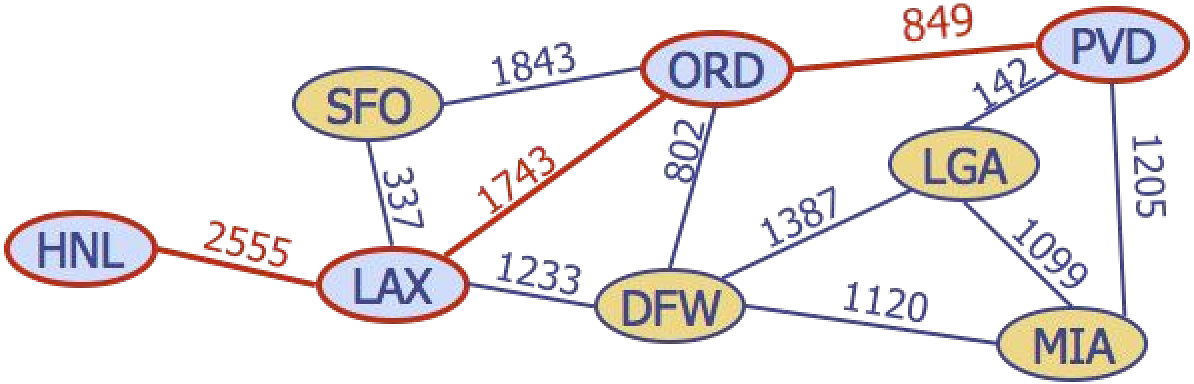
Weighted Graph

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Path Problem

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
    - Length of a path is the sum of the weights of its edges.



Existing Algorithms

Dijkstra's Algorithm

Assumptions

- The graph is connected.
- The edges are undirected.
- The edge weights are nonnegative.

At each step:

- Add to the cloud the vertex u outside the cloud with the smallest distance label, d(u).
- Update the labels of the vertices adjacent to u.

Bellman-Ford Algorithm

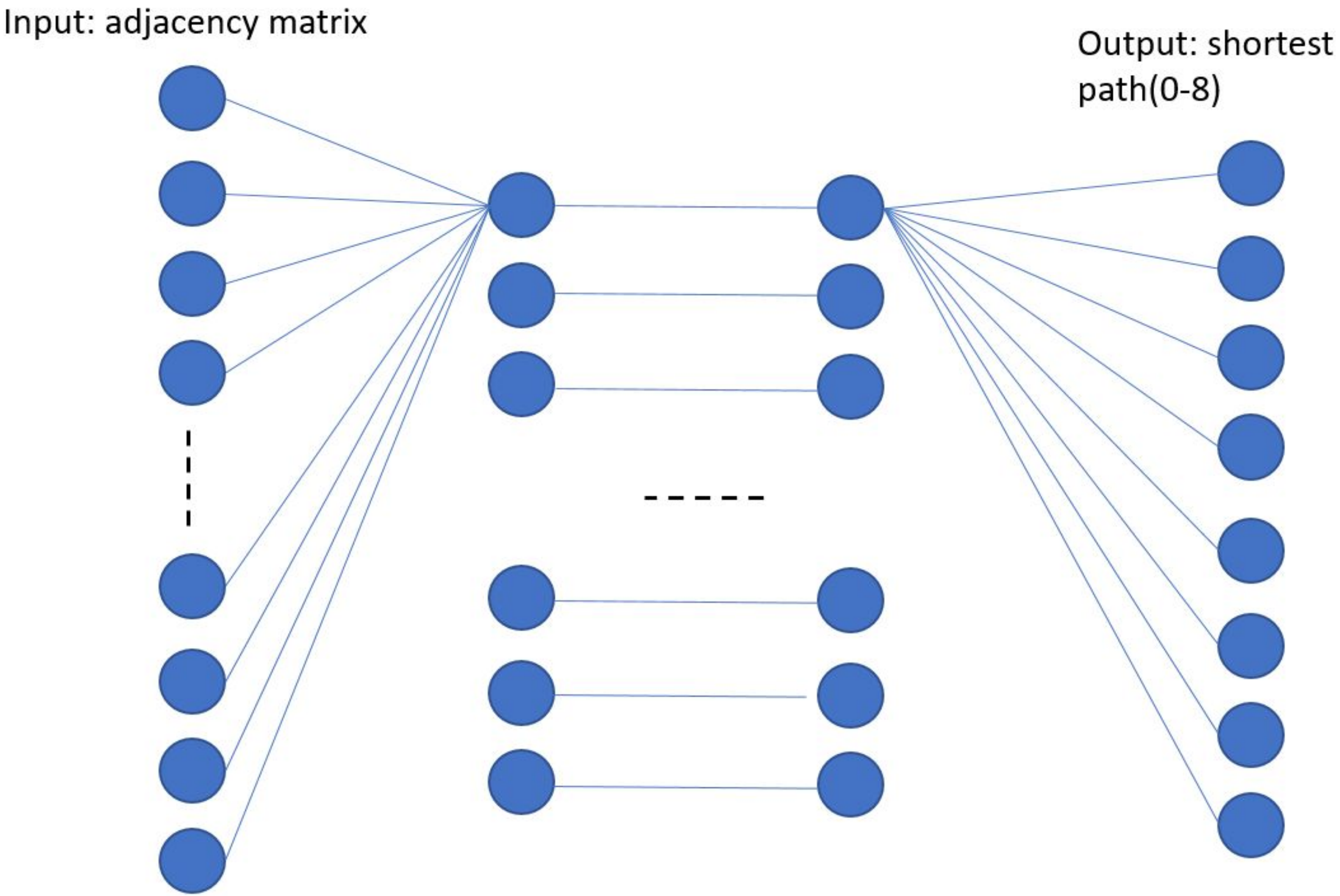
- Works even with negative-weight edges.
- Must assume directed edges.
- Iteration i finds all shortest paths that use i edges.

Implementation via DNN

- Input data: 20000 randomly generated graph with 100 nodes and 200 edges. Each graph has an adjacency matrix of size 100 x 100.
- Input matrix: we flatten each adjacency matrix to a 1x10000 array and append all 20000 to a single matrix with size 20000 x 10000
- Training data : testing data = 8:2

- Architecture:
  - Input layer: 10000
  - Hidden layers
  - Out layer: 9 (Possible shortest length from 0 to 8)

- Parameters:
  - Learning rate
  - Batch size
  - Number of steps


Input: adjacency matrix
Output: shortest
path(0-8)

Choosing the architecture

Hidden layer 1	Hidden layer 2	Hidden layer 3	Hidden layer 4	Accuracy
1024	512	256	0	0.375
256	128	256	0	0.371
256	256	0	0	0.340
256	512	256	128	0.374

Choosing the learning rate by fixing the architecture

Learning rate	Accuracy
0.1	0.33
0.01	0.37
0.001	0.31

The best architecture we get so far

Hidden layer 1: 256

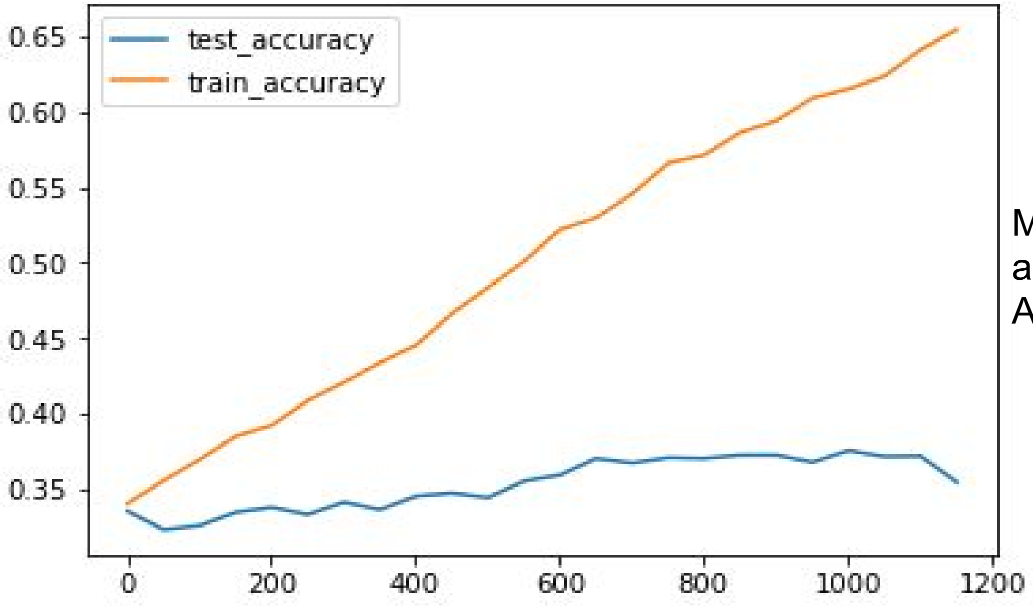
Hidden layer 2: 256

Hidden layer 3: 128

Learning rate: 0.01

Train/Test accuracy vs. training steps

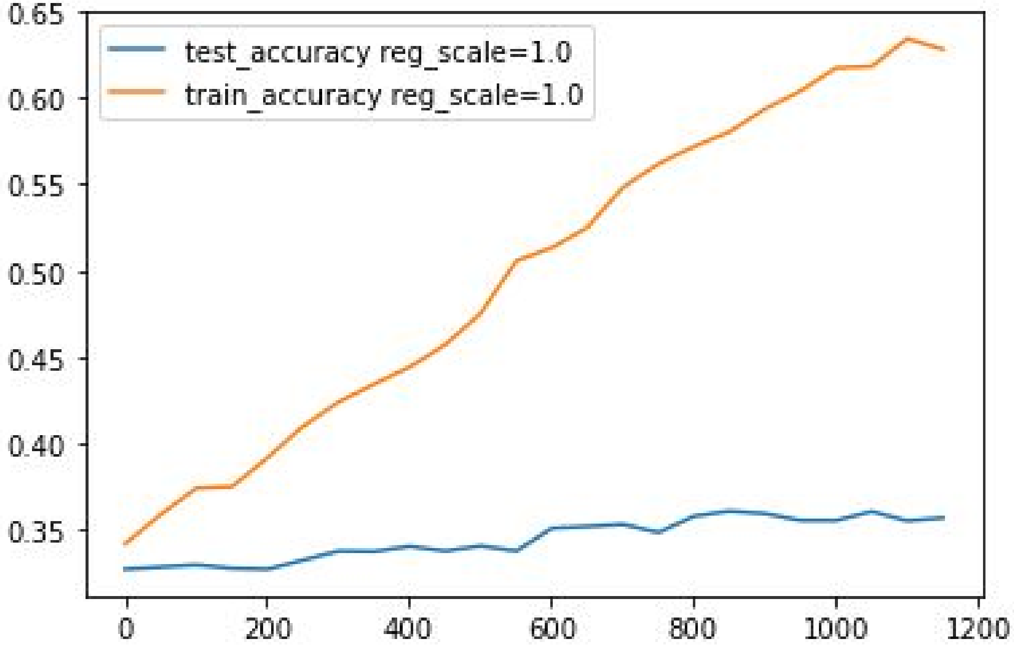
Without regulation or dropout



Max. test acc=0.3705 At 1000 steps

Train/Test accuracy vs. training steps

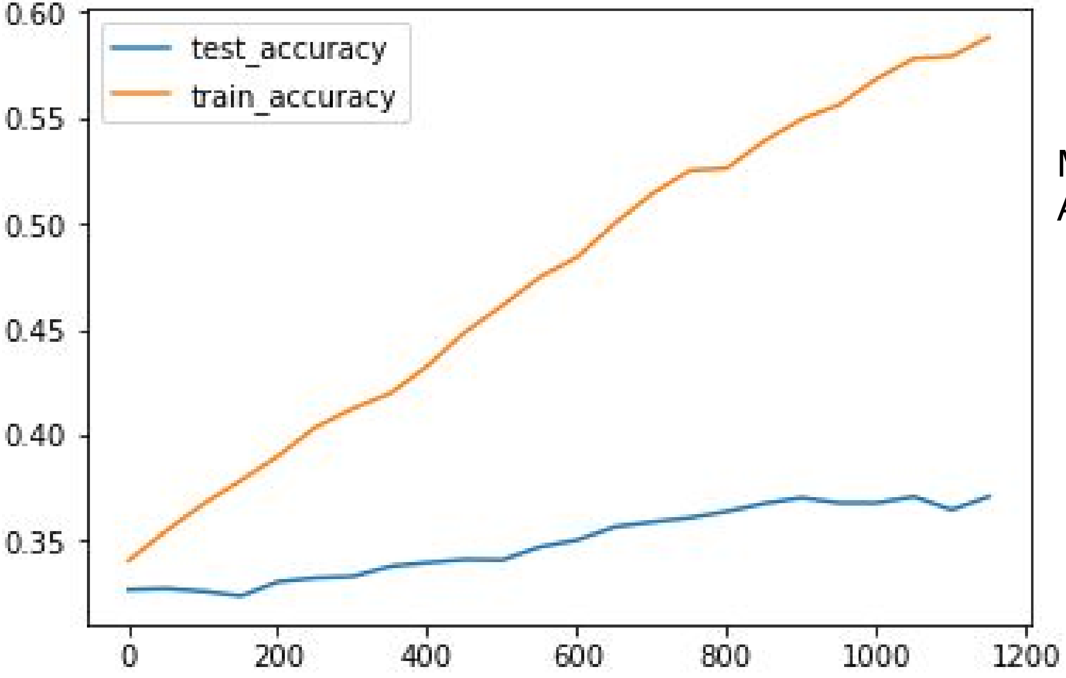
With regulation



Max. test acc=0.364
At 800 steps

Train/Test accuracy vs. training steps

With dropout (rate=0.1)



Max. test acc=0.372
At 1100 steps

Some predictions

```
Model prediction: 3
true value: 4
Model prediction: 4
true value: 2
Model prediction: 4
true value: 4
Model prediction: 4
true value: 4
Model prediction: 3
true value: 3
Model prediction: 4
true value: 5
Model prediction: 4
true value: 4
Model prediction: 3
```

```
Model prediction: 3
true value: 4
Model prediction: 4
true value: 3
Model prediction: 3
true value: 3
Model prediction: 3
true value: 3
Model prediction: 3
true value: 3
Model prediction: 3
true value: 4
Model prediction: 4
true value: 3
Model prediction: 4
true value: 3
Model prediction: 4
true value: 3
```

```
Model prediction: 3
true value: 3
Model prediction: 4
true value: 4
Model prediction: 3
true value: 5
Model prediction: 3
true value: 2
Model prediction: 4
true value: 0
Model prediction: 1
true value: 1
Model prediction: 4
true value: 2
Model prediction: 4
true value: 4
Model prediction: 3
true value: 4
```

Example 5.2.

$$\text{minimize } 2x_{12} - x_{13} - 4x_{23} + 3x_{24} - 6x_{34}$$

$$\text{subject to} \begin{cases} x_{12} + x_{13} = 1, \\ x_{23} + x_{24} - x_{12} = 0, \\ x_{34} - x_{13} - x_{23} = 0, \\ -x_{24} - x_{34} = -1. \end{cases}$$

For simplicity, let assume $x_1 = x_{12}$, $x_2 = x_{13}$, $x_3 = x_{23}$, $x_4 = x_{24}$, $x_5 = x_{34}$. According to the simplified edge path representation, the equivalent LP problem can be described as follows:

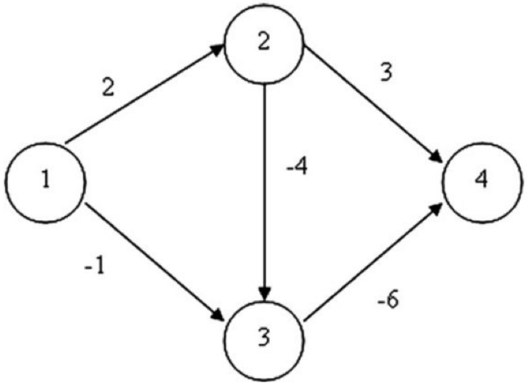


Fig. 4. The graph for Example 5.2.

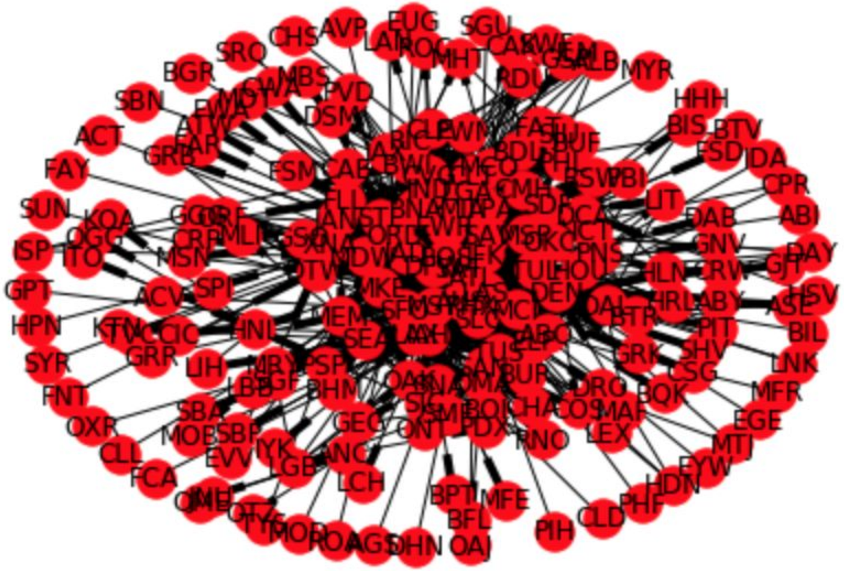
Background Info:

The U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS) tracks the on-time performance of domestic flights operated by large air carriers. Summary information on the number of on-time, delayed, canceled and diverted flights appears in DOT's monthly Air Travel Consumer Report, published about 30 days after the month's end, as well as in summary tables posted on this website. BTS began collecting details on the causes of flight delays in June 2003. Summary statistics and raw data are made available to the public at the time the Air Travel Consumer Report is released.

Dataset Descriptions:

This dataset is free in Kaggle and is once used for Kaggle Competition - Airlines Delay. The dataset covered a long period of time which continued from 1987 to 2008. We selected the latest dataset and used only three columns of the raw dataset for Shortest Path Problem.

Variables	Descriptions
Origin	origin: Departure Airport
Dest	destination: Landing Airport
Distance	distance: distance in miles between Departure Airport and Landing Airport



Train-test-splitting is random.

Cannot make sure the exists a path from the first node to the last node

Furthermore, the number of airport in splitting is not fixed

Brief intro to GCN

Definitions:

Currently, most graph neural network models have a somewhat universal architecture in common.

Graph Convolutional Networks (GCNs); convolutional, because filter parameters are typically shared over all locations in the graph (or a subset thereof as in Duvenaud et al., NIPS 2015).

Input: based on a graph $G = (V,E)$

- A feature description x_i 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 (or some function thereof)

Node-level output:

- Z (an N×F feature matrix, where F is the number of output features per node).

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

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

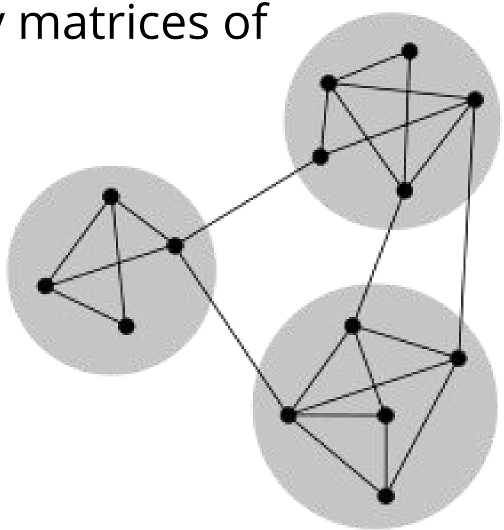
with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\cdot, \cdot)$ is chosen and parameterized.

Implementation via GCN

Alternatively, we created a big-graph which contains 10,000 nodes. For each ten nodes, they consist a ‘community’ so that we get 1,000 ‘communities’.

So the adjacency matrix input of our implementation is the adjacency matrix of the 10,000-node graph; the features will be the adjacency matrices of 10-node subgraph, which are flattened.

All of the above are sparse matrices.



```
num_community = 1000
node = [10 for i in range(num_community)]
GG = nx.random_partition_graph(node,.3,.01,seed=66)
adj_GG = np.zeros((num_community, num_community))
for edge in GG.edges():
    row = edge[0] // num_community
    col = edge[1] // num_community
    if row != col:
        adj_GG[row][col]=1
        adj_GG[col][row]=1

adj_sparse = sparse.csr_matrix(adj_GG)
```

```
adj_sparse
```

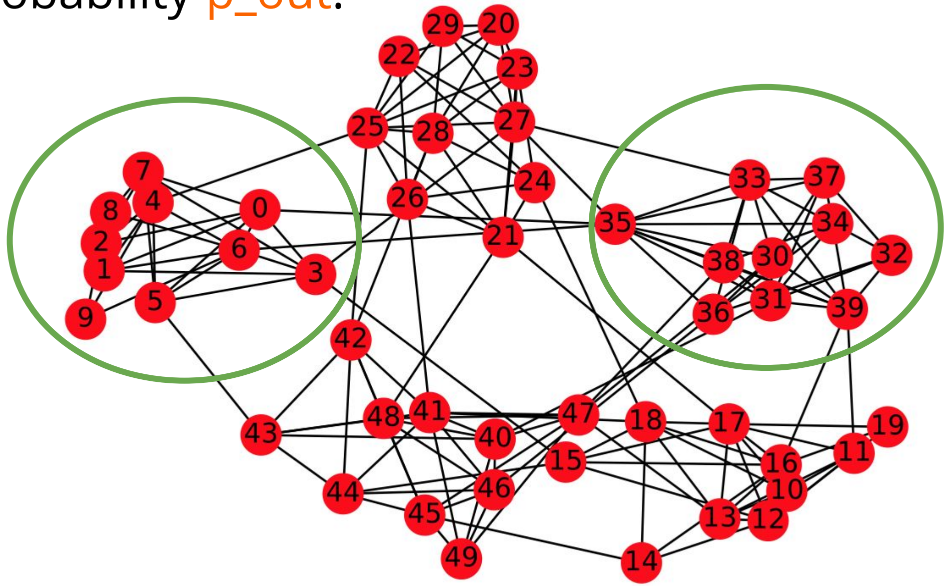
```
<1000x1000 sparse matrix of type '<class 'numpy.float64'>'
	with 90 stored elements in Compressed Sparse Row format>
```

Adj input:

the adjacency matrix of the 1,000-big-node graph.

So 1000 x 1000 sparse matrix.

A partition graph is a graph of communities with sizes defined by s in sizes. Nodes in the same group are connected with probability p_in and nodes of different groups are connected with probability p_out.



```
partition = GG.graph['partition']
```

```
adjlist = [[]]
for i in range(len(partition)):
    H = GG.subgraph(partition[i])
    adj = nx.adjacency_matrix(H).todense().tolist()
    for element in adj:
        adjlist[i].extend(element)
    adjlist.append([])

adjlist = adjlist[:-1]
adj_input = np.array(adjlist)
features_sparse = sparse.csr_matrix(adj_input)
```

```
features_sparse
```

```
<1000x100 sparse matrix of type '<class 'numpy.int64'>'
	with 27008 stored elements in Compressed Sparse Row format>
```

features:

adjacency matrices of 10-node subgraph, which are flattened.

Leads to a 1,000 x 100 sparse matrix to feed the model

labels

```
# labels
Label_Train = np.zeros((int(num_community*0.6),7))
Label_Test = np.zeros((int(num_community*0.2),7))
Label_Val = np.zeros((int(num_community*0.2),7))


for j in range(0,len(Train_path)):
    i=Train_path[j][0]
    Label_Train[j][i-1] = 1

for j in range(0,len(Validation_path)):
    i=Validation_path[j][0]
    Label_Val[j][i-1] = 1

for j in range(0,len(Test_path)):
    i=Test_path[j][0]
    Label_Test[j][i-1] = 1

label_tv = np.concatenate((Label_Train, Label_Val))
labels = np.concatenate((label_tv, Label_Test))

label_tv = np.concatenate((Label_Train, Label_Val))
labels = np.concatenate((label_tv, Label_Test))
```

```
labels.shape
```

```
(1000, 7)
```

Labels:

Labels we used are the shortest path length for each subgraph.

one-hot encoding

```python
def sample_mask(idx, l):
    """Create mask."""
    mask = np.zeros(l)
    mask[idx] = 1
    return np.array(mask, dtype=np.bool)


# Settings
train_size = int(num_community*0.6)
val_size = int(num_community*0.2)
test_size = int(num_community*0.2)


idx_train = range(train_size)
idx_val = range(train_size, train_size+val_size)
idx_test = range(len(idx_val), len(idx_val)+test_size)


train_mask = sample_mask(idx_train, labels.shape[0])
val_mask = sample_mask(idx_val, labels.shape[0])
test_mask = sample_mask(idx_test, labels.shape[0])

y_train = np.zeros(labels.shape)
y_val = np.zeros(labels.shape)
y_test = np.zeros(labels.shape)
y_train[train_mask, :] = labels[train_mask, :]
y_val[val_mask, :] = labels[val_mask, :]
y_test[test_mask, :] = labels[test_mask, :]
```

y_train, y_test, train_mask, test_mask

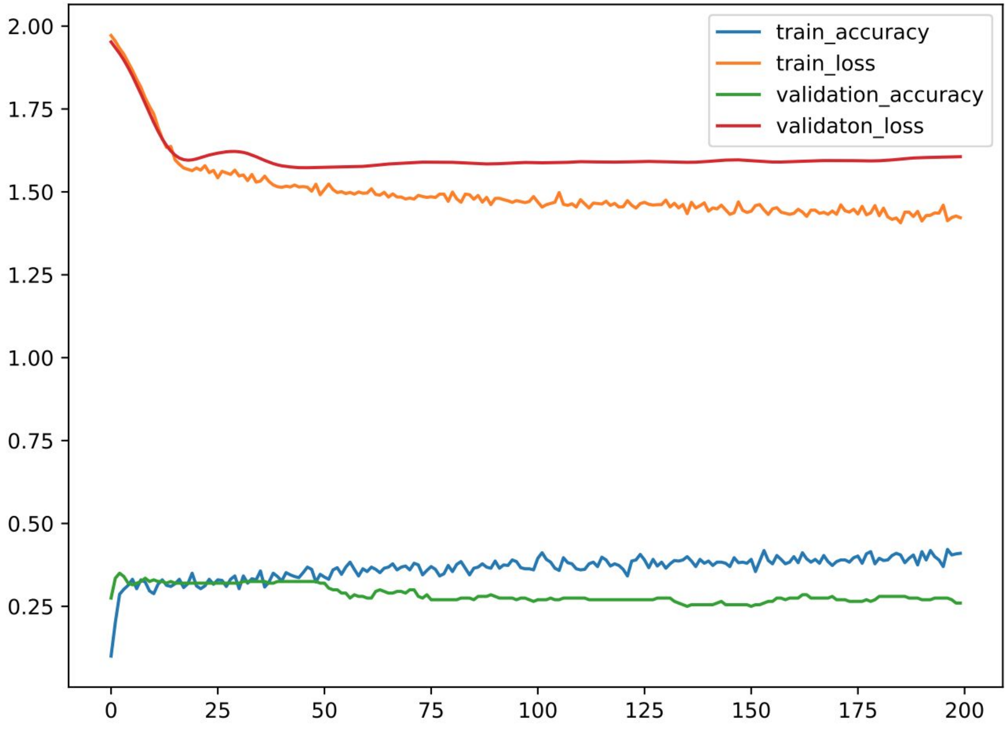
Based on the architecture of GCN and fit our input

- Labels: length of the shortest path in subgraphs. *(via Dijkstra's Algorithm)*
    - (fixed seed -> num_classes fixed)
- Train-validation-test: 60%-20%-20%
- Parameters initial setting:
    - learning_rate: Initial learning rate. learning_rate=0.01，0.05 or 0.1
    - epochs: Number of epochs to train. epochs=200
    - hidden1: Number of units in hidden layer 1. hidden1=8 or 16
    - # weight_decay: Weight for L2 loss on embedding matrix. weight_decay=5e-4
    - dropout: dropout rate (1 - keep probability). dropout=0.5 or 0

```python
def sparse_to_tuple(sparse_mx):
    """Convert sparse matrix to tuple representation."""
    def to_tuple(mx):
        if not sp.sparse.isspmatrix_coo(mx):
            mx = mx.tocoo()
        coords = np.vstack((mx.row, mx.col)).transpose()
        values = mx.data
        shape = mx.shape
        return coords, values, shape

    if isinstance(sparse_mx, list):
        for i in range(len(sparse_mx)):
            sparse_mx[i] = to_tuple(sparse_mx[i])
    else:
        sparse_mx = to_tuple(sparse_mx)

    return sparse_mx


def preprocess_features(features):
    """Row-normalize feature matrix and convert to tuple representation"""
    rowsum = np.array(features.sum(1))
    r_inv1 = np.power(rowsum, -0.5).flatten()
    r_inv2 = np.power(rowsum, -0.5).flatten()
    r_inv = np.multiply(r_inv1, r_inv2)
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.sparse.diags(r_inv, 0)
    features = r_mat_inv.dot(features)
    return sparse_to_tuple(features)
```

Correct the functions to preprocess the features to feed our input features to the gcn model

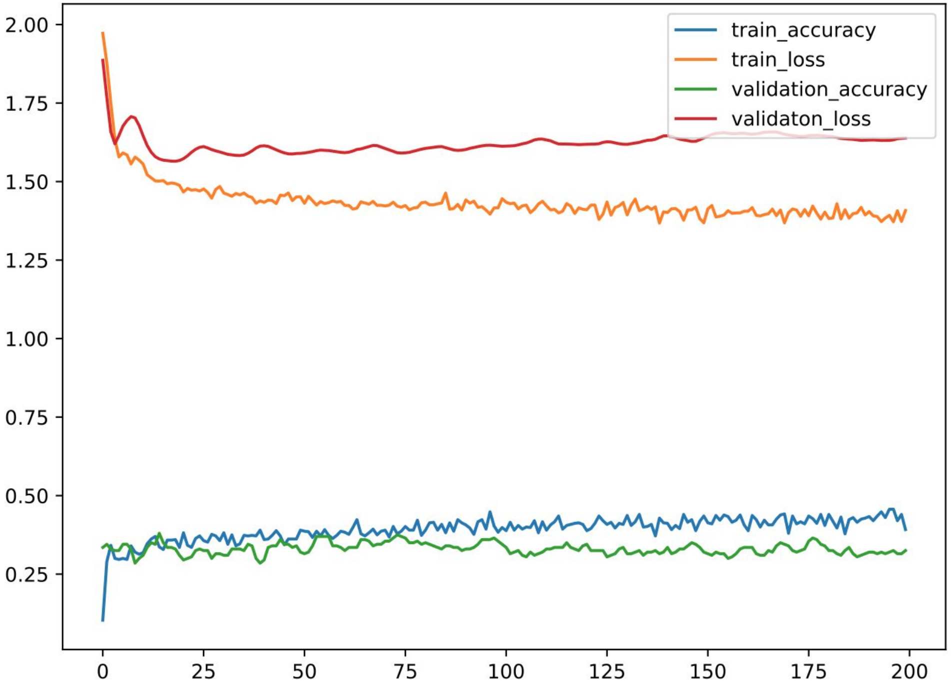
First run:



- learning rate = 0.01
- Dropout = 0.5
- hidden unit = 64

Test accuracy = 0.495

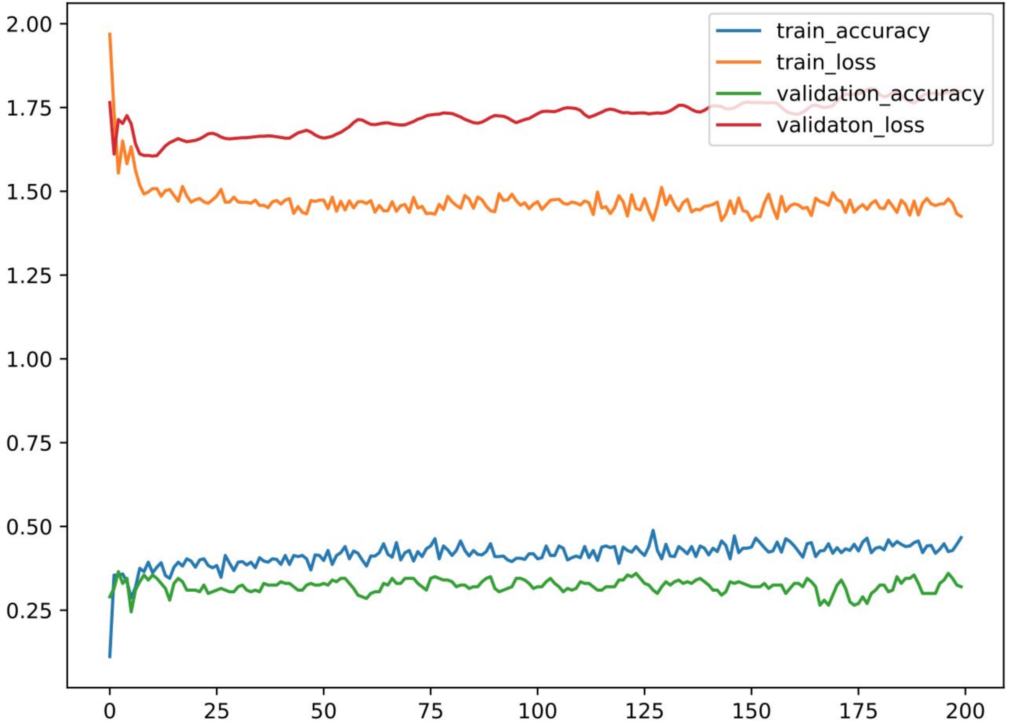
First run:



- Learning rate = 0.05
- Hidden units = 64
- Dropout = 0.5

Test accuracy = **0.6**

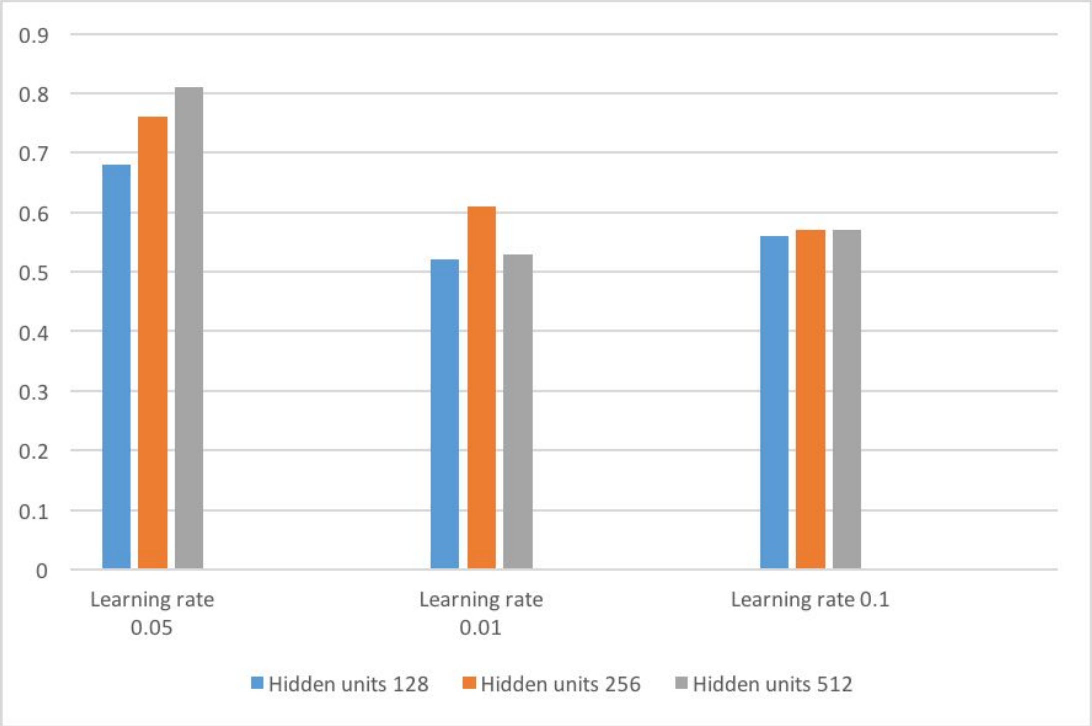
First run:



- learning rate 0.1
- dropout = 0.5
- Hidden units 64

Test accuracy = 0.54000

Fixed dropout = 0.5



Series of run

Fixed dropout = 0.5

test_acc	Hidden units 128	Hidden units 256	Hidden units 512	Hidden units 1024	Hidden units 2048
Learning rate 0.05	0.68	0.76	0.81	0.84	0.79

References

- Kipf & Welling (ICLR 2017), *Semi-Supervised Classification with Graph Convolutional Networks*.
- Defferrard et al. (NIPS 2016), *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering.*
- Alireza Nazemi et al. *An efficient dynamic model for solving the shortest path problem*
- Filipe Araújo et al. *A Neural Network for Shortest Path Computation*