Bipartite Graph Conventional Network for Streamflow Forecasting

Elham Nasarian

Independent Study ISE5974

(ML on Graphs)

Professor Peter Beling

Abstract

Nowadays, the use of Graph Neural Network highly increased and it has been applied in many different applications for solving problems and decision making in complex system. Graph Conventional Network is the most popular algorithm between Graph Neural Network methods, is an effective way to work with node information based on a graph structure. Graph Conventional Network is indicated as one of the basic Graph Neural Networks types.

Keyword: Graph Neural Network (GNN), Graph Conventional Network (GCN), Node, Classification, Forecasting, Bipartite Graph.

1. Introduction

GNN is a machine learning algorithm demonstrated for graph structural data such as drug graphs, networks in social media, or DNA and RNA graphs. It has evolved rapidly over the last few years and is used in many different applications. GNN can be used to apply a several of real-world applications, and it can be applied in these graphs structural based:

1.1. Node-level Prediction

Predicting the classes or labels of nodes. For example, detecting fraudulent entities in the network in cybersecurity can be a node classification problem. **Figure 1** shows the architecture of node prediction [1].

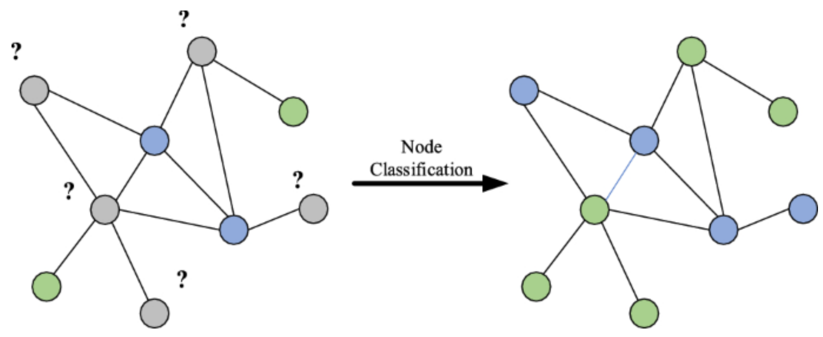


Figure 1. Node Classification Architecture.

1.2. Link-level Prediction

Predicting if there are potential linkages (edges) between nodes. For example, a social networking service suggests possible friend connections based on network data. **Figure 2** shows the architecture of link prediction [2].

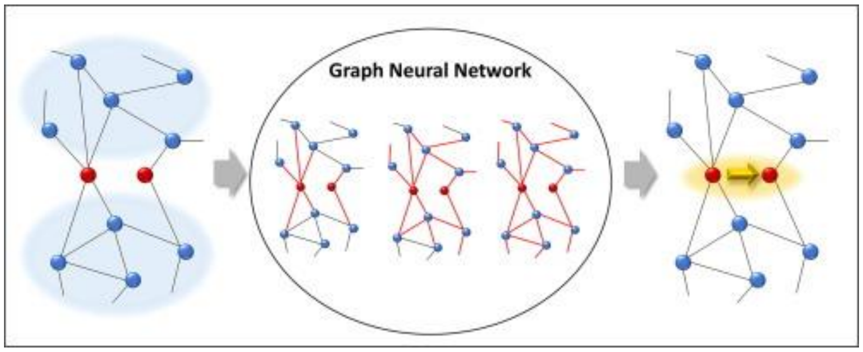


Figure 2. Link Prediction Architecture.

1.3. Graph-level Prediction

Classifying a graph itself into different categories. An example is determining if a chemical compound is toxic or non-toxic by looking at its graph structure. **Figure 3** shows the architecture of graph prediction [3].

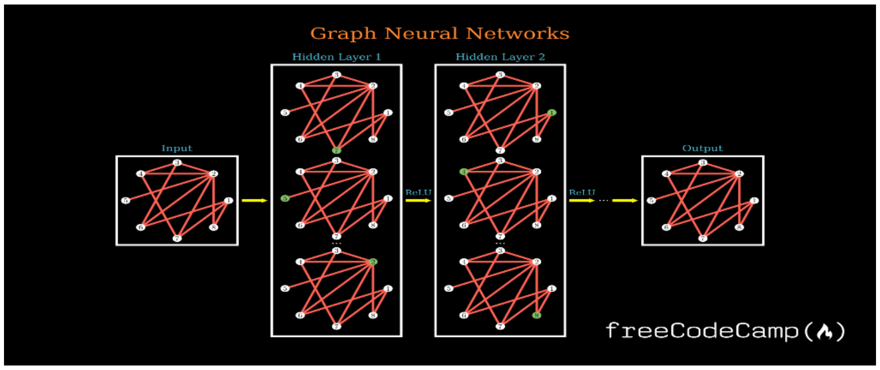


Figure 3. Graph Prediction Architecture.

1.4. Graph Conventional Network (GCN)

Graph Conventional Network (GCN) is the most popular algorithm between Graph Neural Network (GNN) methods, is an effective way to work with node information based on a graph structure. Graph Conventional Network (GCN) is indicated as one of the basic Graph Neural Network (GNN) types. In graphs, nodes have nonstructural orders and the size of neighborhood differs across nodes. Graph Convolution Network takes the average of the node features of a given node and its neighbors to calculate updated node representation values of the node. Through this convolution operation, the node representation captures localized graph information. **Figure 4** shows the architecture of GCN [4].

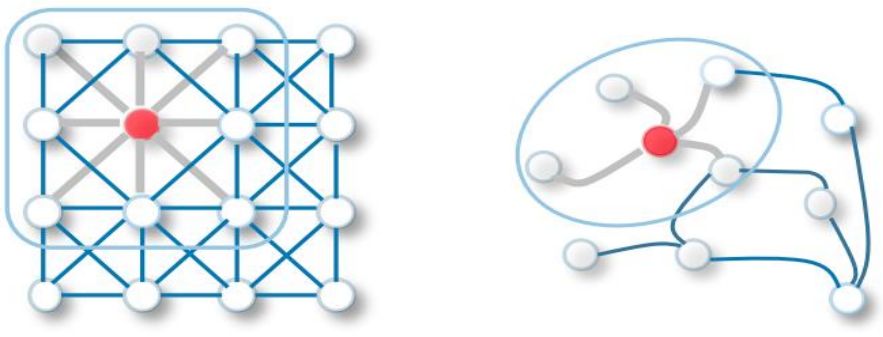


Figure 4. Graph Conventional Network (GCN) Architecture.

GCN performs similar operations as Conventional Neural Network (CNN) where the model learns the features by inspecting neighboring nodes. The major difference between CNN and GNN is that CNN are specially built to operate on regular (Euclidean) structured data, while GNN are the generalized version of CNNs where the numbers of nodes connections vary and the nodes are unordered (irregular on non-Euclidean structured data) [5].

In this approach, we will take into account the Adjacency Matrix (A) in the forward propagation equation in addition to the node features (or so-called input features). A is a matrix that represents the edges or connection between the nodes in the forward propagation equation. The insertion of A in the forward pass equation enables the model to learn the feature representations based on nodes connectivity. For the sake of simplicity, the bias b is omitted. The resulting GCN can be seen as the first-order approximation of Spectral Graph Convolution in the form of a message passing network where the information is propagated along the neighboring nodes within the graph [6].

By adding the adjacency matrix as an additional element, the forward pass equation will then be:

$$H^{[i+1]} = \sigma(W^{[i]} H^{[i]} A^{*})$$

A* is the normalized version of A. To get better understanding on why we need to normalize A and what happens during forward pass in GCNs, let's do an experiment [7].

2. Dataset

For this study I have selected some part of the Global Streamflow Index Metadata (GSIM) [8] which contains North America (987 catchments), South America (813 catchments), and Western Europe (457 catchments). It has three climatology features in 12 months (one feature as a target), **Table 1** shows these features.

Features	Description
P	Precipitation
T	Temperature
Q	Streamflow

Table 1. Features Description

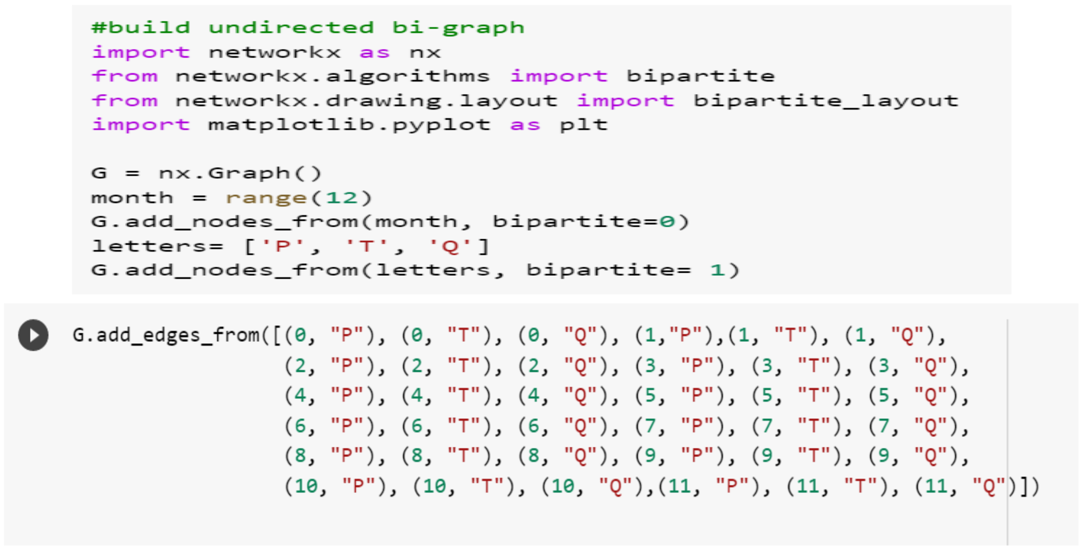
3. Methodology

3.1. Implementing Streamflow Undirected Bipartite Graph in Python

A Bipartite Graph is a graph whose vertices can be divided into two independent sets – A and B. Every (a, b) means a connection between a node from set A and a node from set B. Here “a” belongs to A and “b” belongs to B. The nodes in one set cannot be connected to one another; they can only be connected to nodes in the other set.

In this dataset, for one catchment, I have selected 12 months as a first group (A) with 12 nodes, and three climatology features as a second group (B) with three nodes. Every node in group A has connection with each node in group B, and their connection is indicated as edge. Therefore, each node in group A has three edges with nodes in group B, and our Bipartite graph has 15 nodes and 36 edges.

For building this graph in python, I have used these libraries:



```
#build undirected bi-graph
import networkx as nx
from networkx.algorithms import bipartite
from networkx.drawing.layout import bipartite_layout
import matplotlib.pyplot as plt

G = nx.Graph()
month = range(12)
G.add_nodes_from(month, bipartite=0)
letters= ['P', 'T', 'Q']
G.add_nodes_from(letters, bipartite= 1)
```

```
G.add_edges_from([(0, "P"), (0, "T"), (0, "Q"), (1,"P"),(1, "T"), (1, "Q"),
                  (2, "P"), (2, "T"), (2, "Q"), (3, "P"), (3, "T"), (3, "Q"),
                  (4, "P"), (4, "T"), (4, "Q"), (5, "P"), (5, "T"), (5, "Q"),
                  (6, "P"), (6, "T"), (6, "Q"), (7, "P"), (7, "T"), (7, "Q"),
                  (8, "P"), (8, "T"), (8, "Q"), (9, "P"), (9, "T"), (9, "Q"),
                  (10, "P"), (10, "T"), (10, "Q"),(11, "P"), (11, "T"), (11, "Q")])
```

Figure 5. Defining Nodes & Edges for Streamflow Bipartite Graph in Python

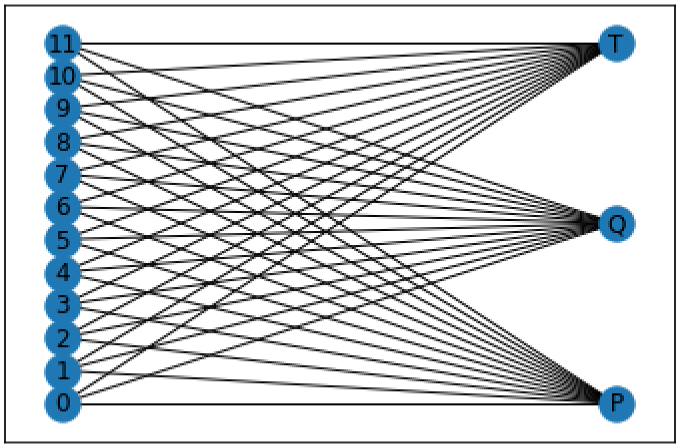


Figure 6. Streamflow Bipartite Graph

3.2. Splitting Train and Test Streamflow Bipartite Graph

For this study and making undirected Bipartite graph, I only have used one catchment with 12 months features, and for this reason (small dataset), I divided train and test to 50%. **Figure 7** shows train graph and test graph.

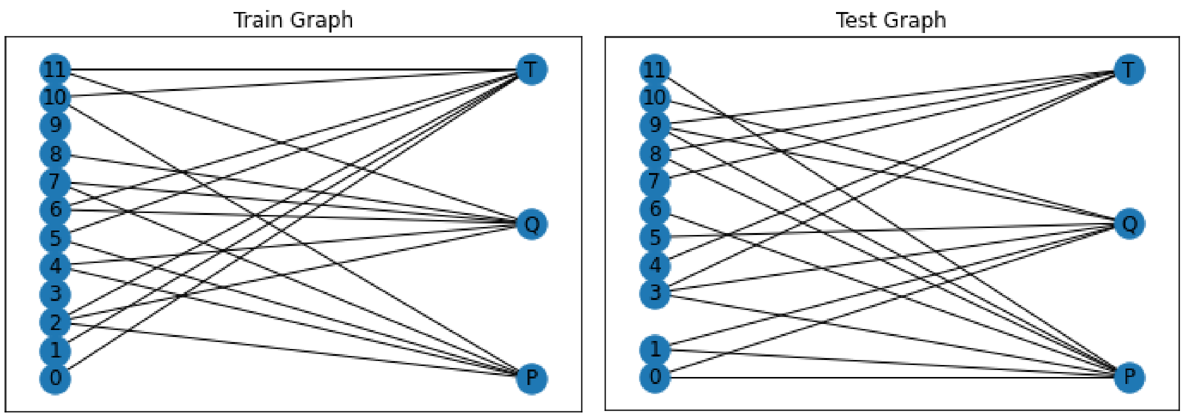
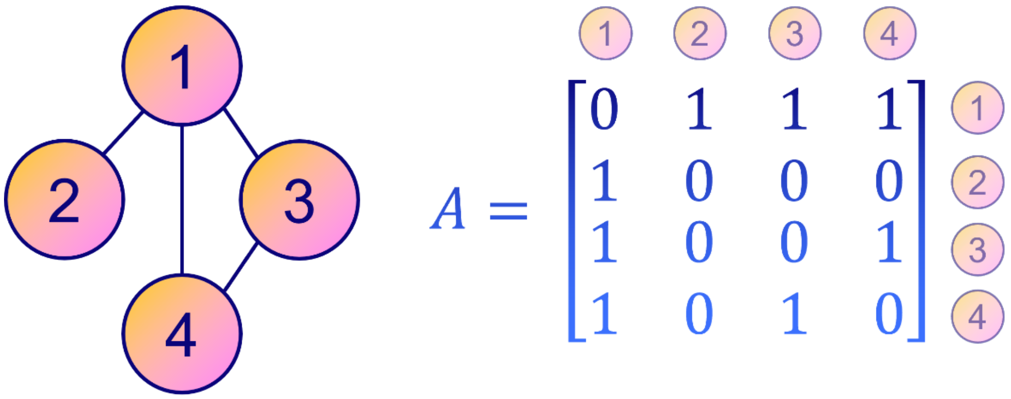


Figure 7. Streamflow Train and Test Graph.

3.3. Building Adjacency Matrix

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix are either 0 or 1, indicating whether there is an edge between two vertices or not. If there is an edge between two vertices, then the corresponding entry in the matrix is 1, and if there is no edge, the entry is 0. The size of the matrix is equal to the number of vertices in the graph, and the rows and columns of the matrix correspond to the vertices. An adjacency matrix is often used to represent undirected graphs, but it can also be used to represent directed graphs by having the elements be either 0 or 1 or a weight representing the strength of the edge [9].

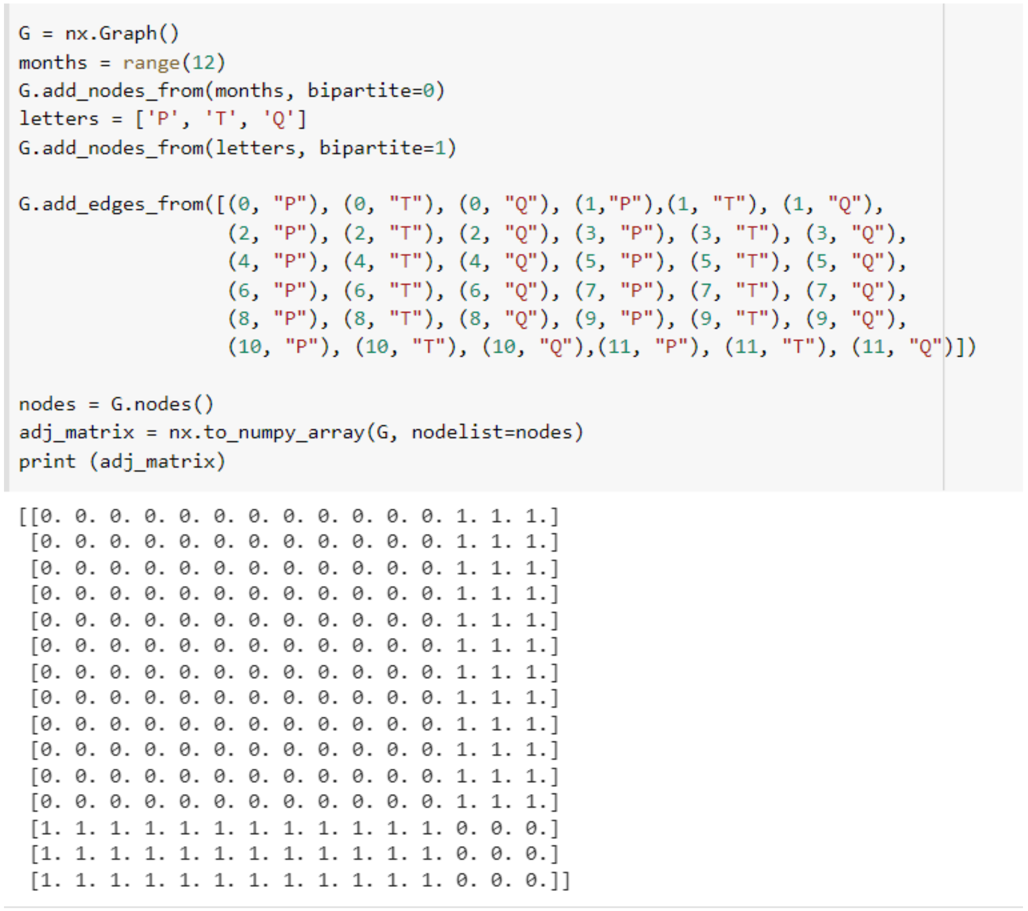
A more intuitive way to represent the graph connectivity would be a simple adjacency matrix *A*, where a non-zero element *Aij* indicates a connection from *i* to *j*. **Figure 8** shows an example of adjacency matrix architecture.



$$A = \begin{array}{cccc|c} 1 & 2 & 3 & 4 & \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 1 & 3 \\ 1 & 0 & 1 & 0 & 4 \end{array}$$

Figure 8. An example of Adjacency Matrix Architecture

I have made adjacency matrix in TensorFlow which can see in following **Figure 8**.



```
G = nx.Graph()
months = range(12)
G.add_nodes_from(months, bipartite=0)
letters = ['P', 'T', 'Q']
G.add_nodes_from(letters, bipartite=1)

G.add_edges_from([(0, "P"), (0, "T"), (0, "Q"), (1,"P"),(1, "T"), (1, "Q"),
                  (2, "P"), (2, "T"), (2, "Q"), (3, "P"), (3, "T"), (3, "Q"),
                  (4, "P"), (4, "T"), (4, "Q"), (5, "P"), (5, "T"), (5, "Q"),
                  (6, "P"), (6, "T"), (6, "Q"), (7, "P"), (7, "T"), (7, "Q"),
                  (8, "P"), (8, "T"), (8, "Q"), (9, "P"), (9, "T"), (9, "Q"),
                  (10, "P"), (10, "T"), (10, "Q"),(11, "P"), (11, "T"), (11, "Q")])

nodes = G.nodes()
adj_matrix = nx.to_numpy_array(G, nodelist=nodes)
print (adj_matrix)
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0.]]
```

Figure 8. Building Adjacency Matrix in TensorFlow

3.4. Applying Bipartite Graph Conventional Network for Streamflow Forecasting

A Graph Convolutional Network (GCN) is a type of deep learning model that can be applied to graph-structured data. In TensorFlow, we can build a GCN by defining a custom layer that performs the graph convolution operation, and then stacking multiple such layers to form a full GCN model.

As we said Bipartite graphs are graphs that consist of two sets of nodes with no edges between nodes within the same set. These graphs are commonly used to model relationships between two different types of entities, such as users and items in recommendation systems, or authors and papers in co-authorship networks [10].

In a GCN (Graph Convolutional Network), the adjacency matrix is used to encode the graph structure, and the node features are updated using convolutional operations over the graph. When working with a bipartite graph, a common approach is to first project the graph onto a lower-dimensional space and then apply the GCN on the resulting graph. This projection can be performed using an adjacency matrix that represents the graph, or using an incidence matrix that encodes the connections between nodes and edges [11].

Once the graph has been projected, the GCN can be applied in the usual way. The GCN operates by updating the node features by aggregating information from the features of their neighbors in the graph. This is done by performing matrix multiplications between the node features, the adjacency matrix or incidence matrix, and the weight matrices of the GCN. The weight matrices are learned during the training process, allowing the GCN to capture the underlying patterns in the graph structure and node features.

In the case of a bipartite graph, the resulting representations produced by the GCN will capture the relationships between the two sets of nodes, making it possible to perform tasks such as link prediction or node classification. It is important to note that the choice of the projection matrix and the architecture of the GCN can greatly impact the performance of the model, so careful consideration should be given to these aspects when working with bipartite graphs [11].

In this study, I first built Adjacency Matrix and then applied GNC for forecasting Streamflow. For this reason, I used this hyperparameters in TensorFlow in **Table 2.**

Hyper Parameters	Values
Loss function	MSE
Optimizer	Adam
Graph structure	Adjacency Matrix
Batch	32
Number of epochs	50
Input Shape	5
Metrics	Accuracy
Number of output units or Neurons	8*3
Initializer	'uniform' and 'zeros'
Trainable	True

Table 2: GCN Hyperparameters.

4. Result

Table 3 demonstrates results for GCN in this dataset.

Number	Epoch	Time	Loss	Accuracy
1	Epoch 1/50	1s 916ms/step	1.2241	0.3333
2	Epoch 2/50	0s 13ms/step	1.2395	0.1333
3	Epoch 3/50	0s 12ms/step	1.2280	0.4000
4	Epoch 4/50	0s 12ms/step	1.2228	0.4667
5	Epoch 5/50	0s 11ms/step	1.2210	0.4000
6	Epoch 6/50	0s 11ms/step	1.2204	0.4000
7	Epoch 7/50	0s 13ms/step	1.2221	0.4000
8	Epoch 8/50	0s 12ms/step	1.2143	0.4000
9	Epoch 9/50	0s 11ms/step	1.2382	0.3333
10	Epoch 10/50	0s 11ms/step	1.2072	0.4667
11	Epoch 11/50	0s 10ms/step	1.2172	0.4000

12	Epoch 12/50	0s 10ms/step	1.2062	0.4000
13	Epoch 13/50	0s 10ms/step	1.2224	0.2667
14	Epoch 14/50	0s 11ms/step	1.2145	0.4000
15	Epoch 15/50	0s 11ms/step	1.2097	0.4000
16	Epoch 16/50	0s 11ms/step	1.2040	0.2667
17	Epoch 17/50	0s 10ms/step	1.1931	0.4000
18	Epoch 18/50	0s 11ms/step	1.1985	0.4000
19	Epoch 19/50	0s 13ms/step	1.2200	0.4000
20	Epoch 20/50	0s 11ms/step	1.1949	0.4000
21	Epoch 21/50	0s 11ms/step	1.2028	0.4000
22	Epoch 22/50	0s 11ms/step	1.2031	0.4000
23	Epoch 23/50	0s 11ms/step	1.1866	0.4000
24	Epoch 24/50	0s 11ms/step	1.2096	0.4000
25	Epoch 25/50	0s 12ms/step	1.2180	0.4000
26	Epoch 26/50	0s 11ms/step	1.2323	0.4000
27	Epoch 27/50	0s 14ms/step	1.1926	0.4000
28	Epoch 28/50	0s 10ms/step	1.1969	0.4000
29	Epoch 29/50	0s 10ms/step	1.2125	0.4000
30	Epoch 30/50	0s 10ms/step	1.2245	0.4000
31	Epoch 31/50	0s 10ms/step	1.2149	0.4000
32	Epoch 32/50	0s 10ms/step	1.2101	0.4000
33	Epoch 33/50	0s 16ms/step	1.2104	0.4000
34	Epoch 34/50	0s 17ms/step	1.1980	0.4000
35	Epoch 35/50	0s 14ms/step	1.2150	0.4000
36	Epoch 36/50	0s 9ms/step	1.1991	0.4000
37	Epoch 37/50	0s 14ms/step	1.1814	0.4000
38	Epoch 38/50	0s 12ms/step	1.1780	0.4000
39	Epoch 39/50	0s 10ms/step	1.1940	0.4000
40	Epoch 40/50	0s 11ms/step	1.2053	0.4000
41	Epoch 41/50	0s 12ms/step	1.1985	0.4000
42	Epoch 42/50	0s 11ms/step	1.2076	0.4000
43	Epoch 43/50	0s 12ms/step	1.2196	0.4000
44	Epoch 44/50	0s 13ms/step	1.2194	0.4000
45	Epoch 45/50	0s 10ms/step	1.2080	0.4000
46	Epoch 46/50	0s 12ms/step	1.1883	0.4000
47	Epoch 47/50	0s 12ms/step	1.1844	0.4000
48	Epoch 48/50	0s 10ms/step	1.2161	0.4000
49	Epoch 49/50	0s 13ms/step	1.1925	0.4000
50	Epoch 50/50	0s 11ms/step	1.2062	0.4000

Table 3: GCN Streamflow Forecasting Results.

As **Table 3** indicates, in the Epoch 4/50 and Epoch 10/50 GCN model had higher accuracy (46.64%). Also, in the Epoch 38/50 GCN method had smaller MSE (1.1780).

Also, these following charts (**Figure 9** and **Figure 10**) visualize the GCN results in 50 epochs.

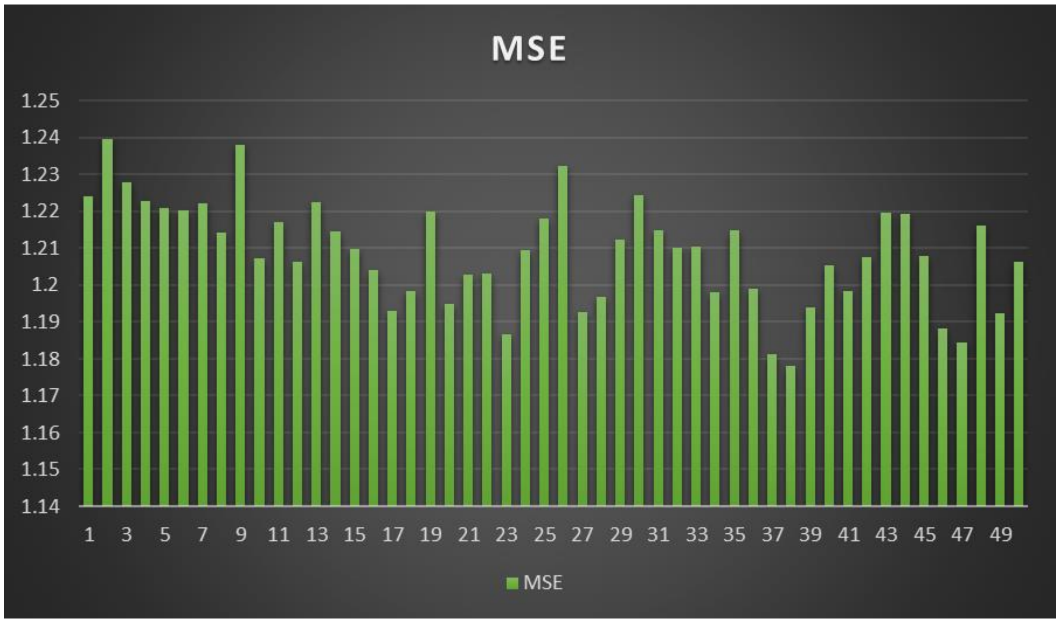


Figure 9. MSE in 50 Epochs.

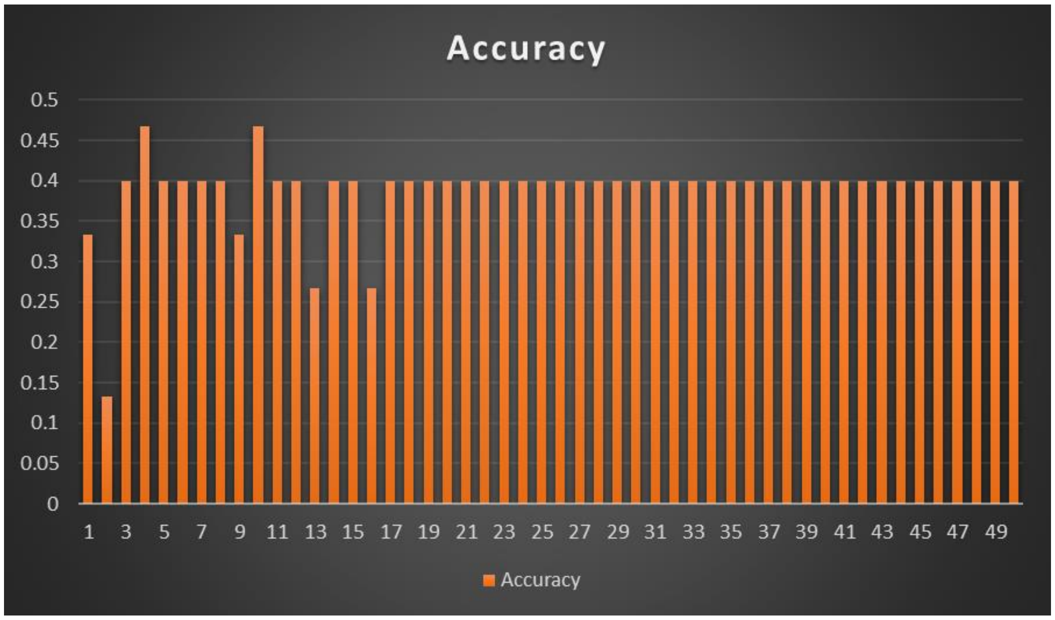


Figure 10. Accuracy in 50 Epochs.

5. Conclusion and future works

The result of a GCN (Graph Convolutional Network) depends on various factors such as the quality of the data, the number of layers in the GCN, the size of the filters used, the activation functions used, the choice of optimizer and the number of epochs run during training, etc.

Typically, GCNs are used for graph-structured data, such as social network graphs, protein interaction networks, or knowledge graphs, among others. The goal is to learn a new representation of the nodes in the graph that captures their local and global relationships. The final result of a

GCN can be used for various downstream tasks such as node classification, graph classification, or link prediction, among others [12].

In general, the accuracy of the GCN model can be evaluated by comparing the predicted labels or values with the ground-truth labels or values using metrics such as accuracy, precision, recall, F1 score, etc. The final result will depend on the specific problem being solved and the quality of the GCN architecture and training process.

In the future and for the next steps, I will use bigger data set and try building different graphs instead of Bipartite graph. Moreover, I will apply different types of GNN with different type of hyperparameters and compare them together.

References:

[1]https://www.google.com/url?sa=i&url=https%3A%2F%2Flink.springer.com%2Farticle%2F10.1007%2Fs00138-021-01251-0&psig=AOvVaw1omnXuUDbFFfMOCiIFMWV8&ust=1675869188441000&source=images&cd=vfe&ved=0CBAQjhxqFwoTCKD1yMLZg_0CFQAAAAAdAAAAABAX

[2]https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.sciencedirect.com%2Fscience%2Farticle%2Fpii%2FS200103702030444X&psig=AOvVaw0jqOEgiOO08PCJhBIQQugu&ust=1675869584488000&source=images&cd=vfe&ved=0CBAQjhxqFwoTCMjil_Xag_0CFQAAAAAdAAAAABBW

[3]https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.freecodecamp.org%2Fnews%2Fgraph-neural-networks-explained-with-examples%2F&psig=AOvVaw2sQPBqDDeIW-pUxREO2Rkx&ust=1675869827968000&source=images&cd=vfe&ved=0CBAQjhxqFwoTCPDR67Pcg_0CFQAAAAAdAAAAABAQ

[4]https://www.google.com/url?sa=i&url=https%3A%2F%2Ftowardsdatascience.com%2Fgraph-convolutional-networks-explained-d88566682b8f&psig=AOvVaw2DWd5FVYQRqw97qjv9Bil_&ust=1675877108906000&source=images&cd=vfe&ved=0CBAQjhxqFwoTCKiwwvb2g_0CFQAAAAAdAAAAABAY

[5] T. Kipf and M. Welling, Semi-Supervised Classification with Graph Convolutional Networks (2017). arXiv preprint arXiv:1609.02907. ICLR 2017.

[6] T. S. Jepsen, https://towardsdatascience.com/how-to-do-deep-learning-on-graphs-with-graph-convolutional-networks-7d2250723780.

[7] Thomas N. Kipf, Max Welling, Semi-Supervised Classification with Graph Convolutional Networks (2017), ICLR 2017.

[8] Do, H. X., Gudmundsson, L., Leonard, M., and Westra, S.: The Global Streamflow Indices and Metadata Archive (GSIM) – Part 1: The production of a daily streamflow archive and metadata, Earth Syst. Sci. Data, 10, 765–785, https://doi.org/10.5194/essd-10-765-2018, 2018.

[9] https://mlabonne.github.io/blog/introgrnn/

[10] Zhezhe Xing, Rui Song, Yun Teng, Hao Xu,DynHEN: A heterogeneous network model for dynamic bipartite graph representation learning, Neurocomputing, Volume 508, 2022.

[11] Wang Z, Zhou M, Arnold C. Toward heterogeneous information fusion: bipartite graph convolutional networks for in silico drug repurposing. Bioinformatics. 2020 Jul 1;36(Suppl_1): i525-i533. doi: 10.1093/bioinformatics/btaa437. PMID: 32657387; PMCID: PMC7355266.

[12]https://www.youtube.com/watch?v=JAB_plj2rbA&list=PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn