

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**

---

**SINGAPORE**

CZ4071 Network Science

Assignment 2 - Final Report

**Survey and Implementation of Paper:**

[7] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanzhan Gu.  
“LayerDependent Importance Sampling for Training Deep and Large Graph Convolutional  
Networks”. In NeurIPS 2019

Submitted by: Team 4

Eddy Lim Qing Yang	U1620115B	eddy0006@e.ntu.edu.sg
Shawn Kan Jung Tze	U1721495G	skan003@e.ntu.edu.sg
Leung Cheuk Yui	U1622880A	cleung002@e.ntu.edu.sg
Joshua Lim Ting Wei	U1622602G	jlim222@e.ntu.edu.sg
Visco Christienne Grace Regodon	U1722616L	christie001@e.ntu.edu.sg
Li Yuting	U1720729F	b170024@e.ntu.edu.sg
Nguyen Ngoc Khanh	U1720451D	C170030@e.ntu.edu.sg
Xing Wanting	U1721577A	wxing003@e.ntu.edu.sg

# Chapter 1

## Introduction

Graph Convolutional Networks (GCNs) applies convolution filter into graphs. Each GNC layer aggregate the embeddings of a node's neighbours from the previous layer and transforms it in a non-linear fashion to obtain the new contextualised node representation. Through stacking multiple GCN layers, each node representation is thus able to make use of a wide receptive field from both immediate and distant node neighbours, which intuitively increases model capacity **assigned\_paper\_zou2019layer**.

An experiment will be conducted in relation to the paper in **assigned\_paper\_zou2019layer**. The experiment will include the implementation and adaptation of algorithms developed from the paper and the replication of experimental results from the algorithms implemented in the paper.

## Chapter 2

### Background

#### 2.1 Problem Statement and Related Works

GCNs are rising in popularity due to favourable implementations in various areas. However, training GCNs is difficult in practice due to the fact that graph data can be extremely large, unlike tokens in a paragraph or pixels in an image that normally have a fixed size. A node's embedding in GCNs depend recursively on all its neighbours' embeddings, causing an exponential growth in computation dependency depending on the number of layers. Thus, GCNs are still inapplicable in large-scale graphs.

Classical solutions to the size problem includes sampling-based methods such as the "node-wise neighbour-sampling" and "layer-wise importance sampling" that train GCNs on a subset of nodes. The "node-wise neighbour-sampling", which recursively samples a fixed number of neighbour nodes and calculates each node's embedding **assigned\_paper\_zou2019layer**. However, when nodes share the same sampled neighbour, the neighbour's embedding has to be reevaluated and the redundancy increases exponentially with more layers. This results in an exponentially increasing computation cost with a node's neighbour size.

An the other hand, the "layer-wise importance sampling" calculates the sampling probability based on the node's degree and a fixed number of nodes is sampled for each layer. The sampled nodes are used to build a smaller, now sampled adjacency matrix, which reduces computation costs. A problem, however, is that sampled nodes from adjacent layers may not be connected due to the fact that sampling probability is independent for each layer. This may cause the matrix to be extremely sparse and so, the sampled nodes may suffer from sparse connectivity problems.

Due to the limitations of the classical methods, alternative methods such as the VR-GCN **chen2017stochastic** FastGCN **chen2018fastgcn** and Cluster-GCN **chiang2019cluster** have been proposed in the recent years. The FastGCN reduces the computation cost by using only sampled nodes to build a sampled adjacency matrix **chen2018fastgcn**, while the VR-GCN uses variance reduction techniques to improve sample complexity to reduce computation redundancy **chen2017stochastic** and the Cluster-GCN does preprocessing before training the GCN by restricting the sampled neighbors within some dense subgraphs through a graph clustering algorithm **chiang2019cluster**. However, it is argued in **assigned\_paper\_zou2019layer** that those proposed methods still cannot resolve the issue of redundant computations completely.

## 2.2 Real-World Applications

Due to its flexibility, GCNs can be applied in various domains in the real-world. Examples of such domains are in the application of GCNs in Computer Vision (CV), Natural Language Processing (NLP) and other sciences.

CV has been a hot research area in the past decades. Although classic convolutional neural networks (CNN) have achieved great successes in CV, it is not feasible to encode the intrinsic graph structures in the specific learning tasks. On the other hand, GCNs have been applied and achieved a comparable or even better performance in some CV problems. For example, **cui2018context** proposes a graph CNN to leverage both the semantic graphs of words and spatial scene graph for visual relationship detection. Furthermore, in **chen2017photographic**, it is shown that a GCN model can be used to process the input scene graph and generate the images by a cascaded refinement network for photographic image synthesis. Moreover, GCNs can also be applied in videos. **yan2018spatial** applied GCN to perform action recognition by proposing a spatial-temporal graph convolutional model to eliminate the need of hand-crafted part assignment which achieved a greater expressive power.

GCNs also have applications in NLP. For text classification, citation network can be constructed with the documents as nodes and the citation relationships among them as edges. And node classification can be a straightforward way to classify documents into different categories. In **yao2019graph**, TextGCN performs text classification by modeling a whole corpus to a heterogeneous graph and learn word embedding and document embedding simultaneously, followed by a softmax classifier for text classification. In addition, a syntactic GCN model is developed and it can be used on top of syntactic dependence trees, which is suitable for semantic role labelling **marcheggiani2017encoding** and neural machine translation **bastings2017graph**.

GCNs are also applied in other domains outside of Computer Science.

In chemistry, **zitnik2018modeling** first models drug-protein target interactions and protein-protein interactions into a multimodal graph, and then graph convolutions is applied to predict polypharmacy side effects.

In material science, **xie2018crystal** proposes a crystal GCN to directly learn material properties from the connections of atoms in the crystal.

In social sciences, GCNs have been widely used for social recommendation to improve the recommendation performances based on user-item interactions and/or user-user interactions. **wu2019neural** proposes a neural influence diffusion model for better social recommendation by considering the influence of trusted friends.

## Chapter 3

### The Challenges of the Problem

The goal of the LADIES sampling method is to improve on the previously implemented sampling methods. The issue with the other methods is that it requires high computation and memory cost. The weakness of node-wise neighbour-sampling method is that it recursively samples a fixed number of neighbours which leads to high computation cost. Furthermore, layer-wise importance-sampling method discards the neighbour-dependent constraints and thus the nodes sampled will lead to high memory cost.

A good sampling method should satisfy 2 criteria. The sampling method has to be layer-wise, neighbour-dependent and must use importance-sampling. The LADIES sampling method solves these issues by mini-batch processing. Every iteration, it chooses a mini-batch nodes then

dependent-sampling of neighbours is applied with a fixed size of sampled nodes that limits the space complexity.

## Chapter 4

### Previous Work: LADIES

#### 4.1 Independent Layer-wise Sampling Framework

Let  $S_l = \{i_k^{(l)}\}_{k=1,2,\dots,s_l}$  where  $s_l = |S_l|$  and  $i_k^{(l)} \in V$  be the set of sampled nodes at the  $l$ -th layer. Hence, The value of matrix production  $PH^{(l)}$  can be estimated in equation ??:

$$PH^{(l)} \approx PS^{(l)}H^{(l)} \quad (4.1)$$

where  $S^{(l)} = \mathbb{R}^{|V| \times |V|}$  is a diagonal matrix defined by the sampling distribution adopted  $p_1^{(l)}, p_2^{(l)}, \dots, p_{|V|}^{(l)}$  to all nodes in  $V$ , which is defined as

$$S_{i,i}^{(l)} = \begin{cases} \frac{1}{s_l p_i^{(l)}}, & i \in S_l \\ 0, & \text{otherwise} \end{cases} \quad (4.2)$$

Define the row selection matrix  $Q^{(l)} \in \mathbb{R}^{s_l \times |V|}$  as:

$$Q_{r,c}^{(l)} = \begin{cases} 1, & (r,c) = (k, i_k^{(l)}) \\ 0, & \text{otherwise} \end{cases} \quad (4.3)$$

Then the forward process of GCN with layer-wise sampling can be approximated by:

$$\tilde{Z}^{(l+1)} = \tilde{P}^{(l)} \tilde{H}^{(l)} W^{(l)}, \quad \tilde{H}^{(l)} = \sigma \tilde{Z}^{(l)} \quad (4.4)$$

where  $\tilde{Z}^{(l)} \in \mathbb{R}^{s_l \times d^{(l)}}$  denotes the approximated intermediate embedding for sampled nodes at  $l$ -layer and equation ?? serves as a modified Laplacian matrix.

$$\tilde{P}^{(l)} = Q^{(l+1)} P S^{(l)} Q^{(l)T} \in \mathbb{R}^{s_{l+1} \times s_l} \quad (4.5)$$

#### 4.2 Layer-dependent Importance Sampling

**Sampling Distribution:** The sampling distribution at  $l$ -th layer is defined in equation ??:

$$p_i^{(l)} = \frac{\|Q^{(l+1)} P_{*,i}\|_2^2}{\|Q^{(l+1)} P\|_F^2} \quad (4.6)$$

where  $\|\mathbf{M}\|_2$  and  $\|\mathbf{M}\|_F$  are respectively L2 Norm and Frobenius Norm.

**Normalisation:** Normalised  $\tilde{P}^{(l)}$

Normalisation operation is essential since it maintains the scale of embedding in the forward process and avoid exploding and vanishing gradient. The proposed normalisation of  $\tilde{P}^{(l)}$  is stated in equation ??:

$$\tilde{P}^{(l)} \leftarrow \mathbf{D}_{\tilde{P}^{(l)}}^{-1} \tilde{P}^{(l)} \quad (4.7)$$

---

**Algorithm 1** Sampling Procedure of LADIES

---

- 1: **Require:** Normalised Laplacian Matrix  $P$ ; Batch Size  $b$ , Sample Number  $n$ . Randomly sample a batch of  $b$  output nodes as  $Q^L$
  - 2: **for**  $l = L - 1$  to  $0$  **do**
  - 3:   Get layer-dependent laplacian matrix  $Q^{(l+1)}P$ . Calculate sampling probability for each node using  $p_i^{(l)} = \frac{\|Q^{(l+1)}P_{*,i}\|_2^2}{\|Q^{(l+1)}P\|_F^2}$ , and organize them into a random diagonal matrix  $S^{(l)}$
  - 4:   Sample  $n$  nodes in  $l$  layer using  $p^{(l)}$ . The sampled nodes formulate  $Q^{(l)}$
  - 5:   Reconstruct sampled laplacian matrix between sampled nodes in layer  $l$  and  $l+1$  by  $\tilde{P}^{(l)} = Q^{(l+1)}PS^{(l)}Q^{(l)T}$ , then normalise it by  $\tilde{P}^{(l)} \leftarrow \mathbf{D}_{\tilde{P}^{(l)}}^{-1} \tilde{P}^{(l)}$
  - 6: **end for**
  - 7: **return** Modified Laplacian Matrices  $\{\tilde{P}^{(l)}\}_{l=1,\dots,L}$  and Sample Node at Input Layer  $Q^0$
- 

## Chapter 5

### Experiment

#### 5.1 Implementation

We managed to fix many bugs in the original code and finally produced the correct version according to the paper. Sampling Procedure is described as follow in algorithm ??, and the code can be found in listing ??.

---

**Algorithm 2** LADIES Pseudo-code

---

- 1: **for** For layer from last to first **do**
  - 2:   Calculate L2 Norm of  $Q(l)P^*$ ,  $i$  and Frobenius Norm of  $Q(l)P$  from squared normalized laplacian matrix using row-selection (Note that QM is equivalent to row-selection operator)
  - 3:   Calculate sampling probabilities  $p$
  - 4:   Fix number of sampled nodes in case  $p$  is negative. (This has no effect on the code since all norms are non-negative but it is included in the original code since they wrongly used laplacian matrix instead of the square.)
  - 5:   Random sample nodes and add the nodes of next layers.
  - 6:   Conduct the unbiased sampling by dividing the to sampling probabilities
  - 7:   Row-normalization to avoid explosion of feature vectors.
  - 8: **end for**
- 

```

1 def sampler(batch_nodes: np.ndarray, samp_num_list: np.ndarray, num_nodes: int,
2             lap_matrix: sparse.csr_matrix, lap2_matrix: sparse.csr_matrix, num_layers: int)
3     -> SimpleNamespace:
4     ##### INPUT #####
5     F = \text{row-F} = \text{row-permutation}(I_N)
6     permutation}(I_N)
7     # samp_num_list: array of number of sampled nodes at all layers
8     # num_nodes : number of graph nodes
9     # lap_matrix : row-normalized laplacian matrix
10    # lap2_matrix : squared lap_matrix (precomputed)
11    # num_layers : len(samp_num_list)
12    ##### OUTPUT #####
13    # adjs : P matrix

```

```

13 # input_nodes: sampled nodes at input
14 # output_nodes : batch_nodes
15 #####
16 def full_sampler(batch_nodes: np.ndarray, samp_num_list: np.ndarray, num_nodes:
    int, lap_matrix: sparse.csr_matrix, lap2_matrix: sparse.csr_matrix, num_layers:
    int) -> SimpleNamespace:
17 # simply sample the full lap_matrix for every layers
18 sample = SimpleNamespace(
19     adjs= [lap_matrix for _ in range(num_layers)],
20     input_nodes= np.arange(num_nodes),
21     output_nodes= batch_nodes,
22 )
23 return s
24 def ladies_sampler(batch_nodes: np.ndarray, samp_num_list: np.ndarray, num_nodes:
    int, lap_matrix: sparse.csr_matrix, lap2_matrix: sparse.csr_matrix, num_layers:
    int) -> SimpleNamespace:
25 '''
26     LADIES_Sampler: Sample a fixed number of nodes per layer. The sampling
    probability (importance)
27         is computed adaptively according to the nodes sampled in
    the upper layer.
28 '''
29 previous_nodes = batch_nodes
30 adjs = []
31 '''
32     Sample nodes from top to bottom, based on the probability computed
    adaptively (layer-dependent).
33 '''
34 for d in range(num_layers):
35     # row-select the lap2_matrix (U2) by previously sampled nodes
36     U2 = lap2_matrix[previous_nodes , :]
37     # calculate sampling probabilities
38     pi = np.sum(U2, axis=0)
39     p = pi / np.sum(pi)
40     # get number of sampled nodes
41     s_num = np.min([np.sum(p > 0), samp_num_list[d]])
42     p = p.view(np.ndarray).flatten()
43     # sample the next layer's nodes based on the adaptively probability (p).
44     after_nodes = np.random.choice(num_nodes, s_num, p = p, replace = False)
45     # Add output nodes for self-loop
46     after_nodes = np.unique(np.concatenate((after_nodes, batch_nodes)))
47     # row-select and col-select the lap_matrix (U), and then divided by the
    sampled probability for unbiased-sampling.
48     adj = lap_matrix[previous_nodes, :][:, after_nodes]
49     adj = adj.multiply(1/ p[after_nodes])
50     # conduct row-normalization to avoid value explosion.
51     adj = row_normalize(adj)
52     # fill the sub-matrix into the original
53     adj = sparse_fill(lap_matrix.shape, adj, previous_nodes, after_nodes)
54     adjs.append(adj)
55     # turn the sampled nodes as previous_nodes, recursively conduct sampling.
56     previous_nodes = after_nodes
57 # Reverse the sampled probability from bottom to top. Only require input how
    the lastly sampled nodes.
58 adjs.reverse()
59
60 sample = SimpleNamespace(
61     adjs= adjs,
62     input_nodes= previous_nodes,
63     output_nodes= batch_nodes,
64 )
65 return sample

```

Listing 5.1: LADIES code

## 5.2 Experiment

Due to the limitation of computing power, we decided to use random block model with two same-sized clusters with connection probability as follow in equation ??.

$$P = \begin{bmatrix} \frac{\log(N)}{N}, \frac{\log(N)}{N^2} \\ \frac{\log(N)}{N^2}, \frac{\log(N)}{N} \end{bmatrix} \quad (5.1)$$

where  $P_{i,j}$  is the connection probability between nodes in cluster  $i$  and cluster  $j$ .

The feature vector is taken as an random row-permutation of Identity matrix where feature vector is represented in equation ??.

$$F = \text{row-permutation}(I_N) \quad (5.2)$$

The neural network consists of 5 GCN layers of 64 hidden layer dimensions and a Linear layer at the end works as a classifier. All layers were implemented with a dropout probability of 0.5

We used cross-entropy loss with log\_softmax with Adam Optimizer, the learning rate was set to  $1e-3$ .

For each run, the model was trained in 80 epochs with two sampling modes: Full-batch and ladies. The execution time, loss and f1 score was recorded in table ?. The plots obtained by the experiment can be referred to in in Appendix ?.

Number of Nodes	Runtime for 80 epochs	
	Full-Batch	LADIES
64	2.6	8.82
128	6.63	30.71
256	22.80	101.91
512	98.51	315.95
1024	384.68	1020.65

Table 5.1: Model runtime for 80 epochs

Number of Nodes	Convergence Epochs	
	Full-Batch	LADIES
64	35	50
128	40	36
256	40	20
512	30	18
1024	16	15

Table 5.2: Number of Epochs for convergence

Number of Nodes	Updates per epoch	
	Full-Batch	LADIES
64	1	1
128	1	2
256	1	4
512	1	8
1024	1	16

Table 5.3: Number of updates per epoch

Number of Nodes	Updates for convergence	
	Full-Batch	LADIES
64	35	50
128	40	72
256	40	80
512	30	144
1024	16	240

Table 5.4: Number of updates to convergence

Number of Nodes	Convergence time	
	Full-Batch	LADIES
64	1.3	5.8
128	3.5	15
256	12.5	25
512	38	65
1024	85	140

Table 5.5: Convergence time

As such, the observation from the results in table ?? is that LADIES converges much faster than Full-Batch w.r.t epochs especially for large graphs due to the nature of stochastic gradient descent. LADIES sparse matrix implementation can be improved further.

## Chapter 6

### Conclusion

The difference between Layer-Dependent Importance Sampling (LADIES) method and the Full-Batch sampling method is that the LADIES method does not sample the whole graph. It samples the important nodes that will be processed in the neural network. However, the Full-Batch sampling method uses the whole graph as training samples. The methods can be considered as batch gradient descent and stochastic gradient descent. Throughout the experiments, it is found that Full-Batch sampling has a better execution time as compared to LADIES sampling. However, due to space complexity, Full-Batch sampling cannot be scaled up for large-size data. On the other hand, by the nature of stochastic gradient descent, LADIES method takes the advantage of conducting many updates in a batch of data, it converges faster than Full-Batch sampling.



## **Chapter 7**

### **Further Research**

It has been observed that the execution time of LADIES method is at least 1.5 times slower than Full-Batch method caused by the inefficiency in sparse matrix implementation. A better implementation is possible to yield better results.

# Appendix A

## Experiment Code

```
1 import pdb
2 import argparse
3 from typing import List, Dict
4 from types import SimpleNamespace
5 import multiprocessing as mp
6 import time
7
8 import numpy as np
9 import scipy as sp
10 import torch
11 import torch.nn as nn
12 import torch.nn.functional as F
13 import torch.optim as optim
14 from sklearn.metrics import f1_score
15 from sklearn.linear_model import LinearRegression
16 from torchviz import make_dot
17
18 from module import GCN, GCNLinear
19 from load_data import load_random_block
20 from sampler import full_sampler, ladies_sampler
21 from utils import adj_to_lap_matrix, row_normalize,
    sparse_mx_to_torch_sparse_tensor, sparse_fill
22
23
24 #np.seterr(all="raise")
25
26 def random_sampling_train(args: SimpleNamespace, model: SimpleNamespace, data:
    SimpleNamespace) -> SimpleNamespace:
27     if args.batch_size <= 0 or args.batch_size >= len(data.train_nodes):
28         batch_nodes = data.train_nodes
29     else:
30         batch_nodes = np.random.choice(data.train_nodes, size= args.batch_size,
            replace= True)
31     sample = model.sampler(
32         batch_nodes= batch_nodes,
33         samp_num_list= [len(batch_nodes) for _ in range(args.num_layers)],
34         num_nodes= data.num_nodes,
35         lap_matrix= data.lap_matrix,
36         lap2_matrix= data.lap2_matrix,
37         num_layers= args.num_layers,
38     )
39     return sample
40
41 def sampling_valid(args: SimpleNamespace, model: SimpleNamespace, data:
    SimpleNamespace) -> SimpleNamespace:
42     sample = full_sampler(
43         #batch_nodes= data.valid_nodes,
44         batch_nodes= np.arange(data.num_nodes),
45         samp_num_list= None,
46         num_nodes= data.num_nodes,
47         lap_matrix= data.lap_matrix,
48         lap2_matrix= None,
49         num_layers= args.num_layers,
50     )
51     return sample
52
53
54
55 if __name__ == "__main__":
56     # load args
57     parser = argparse.ArgumentParser(description='Training GCN')
58     parser.add_argument('--hidden_features', type=int, default=64,
```

```

59         help='hidden layer embedding dimension')
60     parser.add_argument('--num_epochs', type=int, default=10,
61         help='Number of epochs')
62     parser.add_argument('--batch_size', type=int, default=64,
63         help='batch_size: number of sampled nodes at output layer')
64     parser.add_argument('--num_layers', type=int, default=5,
65         help='Number of GCN layers')
66     parser.add_argument('--sampling_method', type=str, default='full',
67         help='sampling algorithms: full/ladies')
68     parser.add_argument('--cuda', type=int, default=-1,
69         help='GPU ID, (-1) for CPU')
70     parser.add_argument('--dropout', type=float, default=0.5,
71         help='dropout probability')
72     parser.add_argument('--learning_rate', type=float, default=1e-3,
73         help='learning rate')
74     parser.add_argument('--num_nodes', type=int, default=100,
75         help='number of network nodes in random block model')
76
77     args = parser.parse_args()
78     print(args)
79     # set up device
80     if args.cuda != -1:
81         device = torch.device("cuda:" + str(args.cuda))
82     else:
83         device = torch.device("cpu")
84     # load data
85     all = args.num_nodes
86     half = int(all/2)
87     p1 = np.log(all) / all
88     p2 = p1 / all
89     data = load_random_block(
90         [half, all-half],
91         [[p1, p2], [p2, p1]],
92     )
93     data.num_nodes = data.features.shape[0]
94     data.in_features = data.features.shape[1]
95     data.out_features = len(np.unique(data.labels))
96     data.lap_matrix = row_normalize(adj_to_lap_matrix(data.adj_matrix))
97     data.lap2_matrix = data.lap_matrix.multiply(data.lap_matrix)
98
99     if args.sampling_method == "full":
100         args.batch_size = len(data.train_nodes)
101     # create pool
102     pool = mp.Pool(processes=1)
103     # create model
104     model: SimpleNamespace = SimpleNamespace()
105     model.sampler = full_sampler
106     if args.sampling_method == "ladies":
107         model.sampler = ladies_sampler
108     model.module = GCNLinear(
109         encoder=GCN(
110             in_features=data.in_features,
111             hidden_features=args.hidden_features,
112             out_features=args.hidden_features,
113             num_layers=args.num_layers,
114             dropout=args.dropout
115         ),
116         in_features=args.hidden_features,
117         out_features=data.out_features,
118         dropout=args.dropout,
119     )
120
121
122     model.module.to(device)
123     optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.module.parameters
    ()), lr=args.learning_rate)

```

```

124 criterion = nn.CrossEntropyLoss()
125 times = []
126 losses = []
127 f1s = []
128 next_sample_async = None
129 sample = None
130
131 #START TRAINING
132 start = time.time()
133 try:
134     for epoch in range(args.num_epochs):
135         # train
136         model.module.train() # train mode
137         print(f"Epoch {epoch}: ", flush= True)
138         num_iterations = int(data.num_nodes / args.batch_size)
139         for iter in range(num_iterations):
140             print(f"\tIteration {iter}: ", end= "", flush= True)
141             if next_sample_async is None:
142                 sample = random_sampling_train(args, model, data)
143             else:
144                 sample = next_sample_async.get()
145                 next_sample_async = pool.apply_async(random_sampling_train, args= (args,
146 model, data))
147                 optimizer.zero_grad()
148
149                 output = model.module(
150                     x= sparse_mx_to_torch_sparse_tensor(sparse_fill(shape= data.features.
151 shape, mx= data.features[sample.input_nodes], row= sample.input_nodes)),
152                     adjs= list(map(lambda adj: sparse_mx_to_torch_sparse_tensor(adj).to(
153 device), sample.adjs)),
154                 )
155
156                 loss = criterion(
157                     output[sample.output_nodes],
158                     torch.from_numpy(data.labels[sample.output_nodes]).long(),
159                 )
160                 loss.backward()
161                 torch.nn.utils.clip_grad_norm_(model.module.parameters(), 0.2)
162                 optimizer.step()
163
164                 #if epoch == 0 and iter == 0:
165                 # dot = make_dot(loss.mean(), params= dict(model.module.named_parameters
166 ()))
167                 # dot.render("test.gv", view= True)
168
169                 loss = loss.detach().cpu()
170                 print(f"Loss {loss}", flush= True)
171             # eval
172             model.module.eval() # eval mode
173             sample = sampling_valid(args, model, data)
174             output = model.module(
175                 x= sparse_mx_to_torch_sparse_tensor(data.features[sample.input_nodes]),
176                 adjs= list(map(lambda adj: sparse_mx_to_torch_sparse_tensor(adj).to(device
177 ), sample.adjs)),
178             )
179             loss = criterion(
180                 output[sample.output_nodes],
181                 torch.from_numpy(data.labels[sample.output_nodes]).long(),
182             )
183
184             output = output.detach().cpu()
185             loss = loss.detach().cpu()
186             f1 = f1_score(
187                 output[sample.output_nodes].argmax(dim=1),
188                 data.labels[sample.output_nodes],

```

```

185         average= "micro",
186     )
187     times.append(time.time() - start)
188     losses.append(loss)
189     f1s.append(f1)
190
191     epochs = np.arange(len(times)).reshape(-1, 1)
192     reg = LinearRegression().fit(epochs, times)
193     eta = reg.predict(np.array(args.num_epochs - 1).reshape(-1, 1))[0]
194
195     print(f"Epoch {epoch}: Loss {loss} F1 {f1} ETA {eta - time.time() + start}s"
196           , flush= True)
197
198 except KeyboardInterrupt:
199     pass
200     print(f"Elapsed Time: {time.time() - start}")
201
202 import matplotlib.pyplot as plt
203 fig, axs = plt.subplots(nrows= 2, ncols= 2, constrained_layout=True)
204 fig.suptitle(f"Sampling method: {args.sampling_method}, Number of nodes {args.
205              num_nodes}")
206
207 axs[0][0].plot(np.arange(len(times)), losses)
208 axs[0][0].set_xlabel("Epoch")
209 axs[0][0].set_ylabel("Loss")
210
211 axs[0][1].plot(times, losses)
212 axs[0][1].set_xlabel("Time")
213 axs[0][1].set_ylabel("Loss")
214
215 axs[1][0].plot(np.arange(len(times)), f1s)
216 axs[1][0].set_xlabel("Epoch")
217 axs[1][0].set_ylabel("F1")
218
219 axs[1][1].plot(times, f1s)
220 axs[1][1].set_xlabel("Time")
221 axs[1][1].set_ylabel("F1")
222
223 plt.show()

```

Listing A.1: Training code

## Appendix B

### Experiment Plots

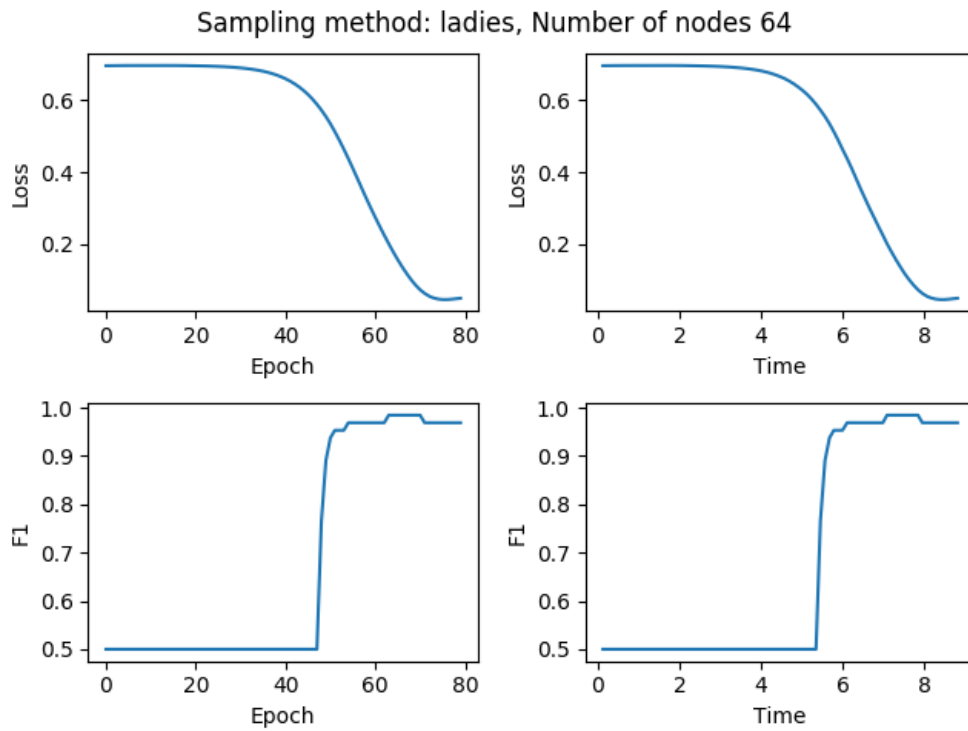


Figure B.1: Ladies with 64 nodes

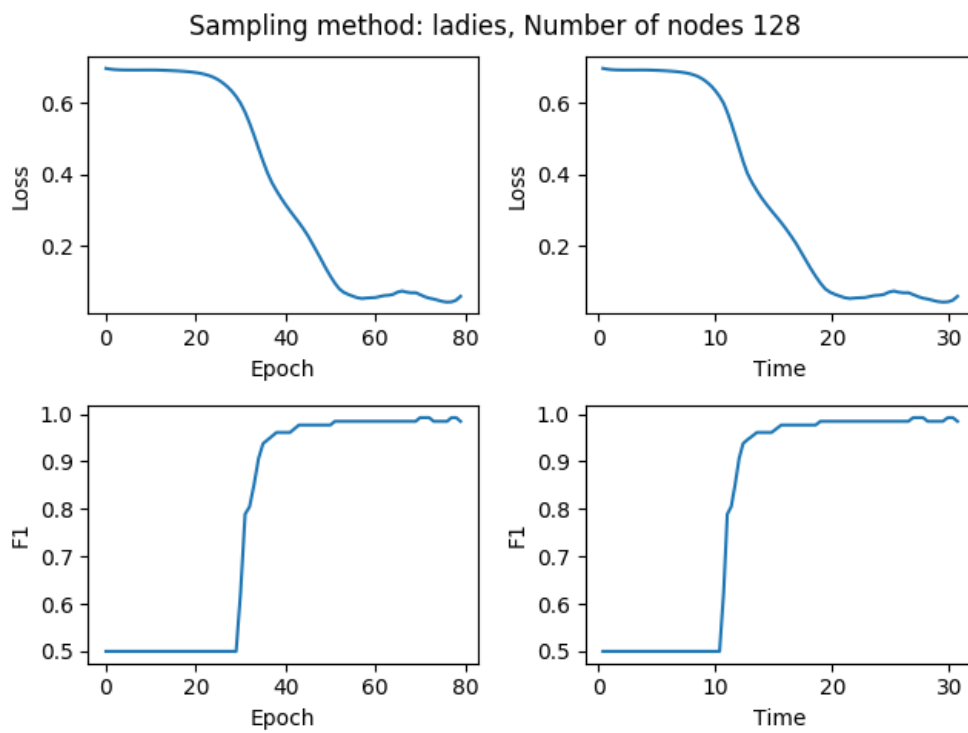


Figure B.2: Ladies with 128 nodes

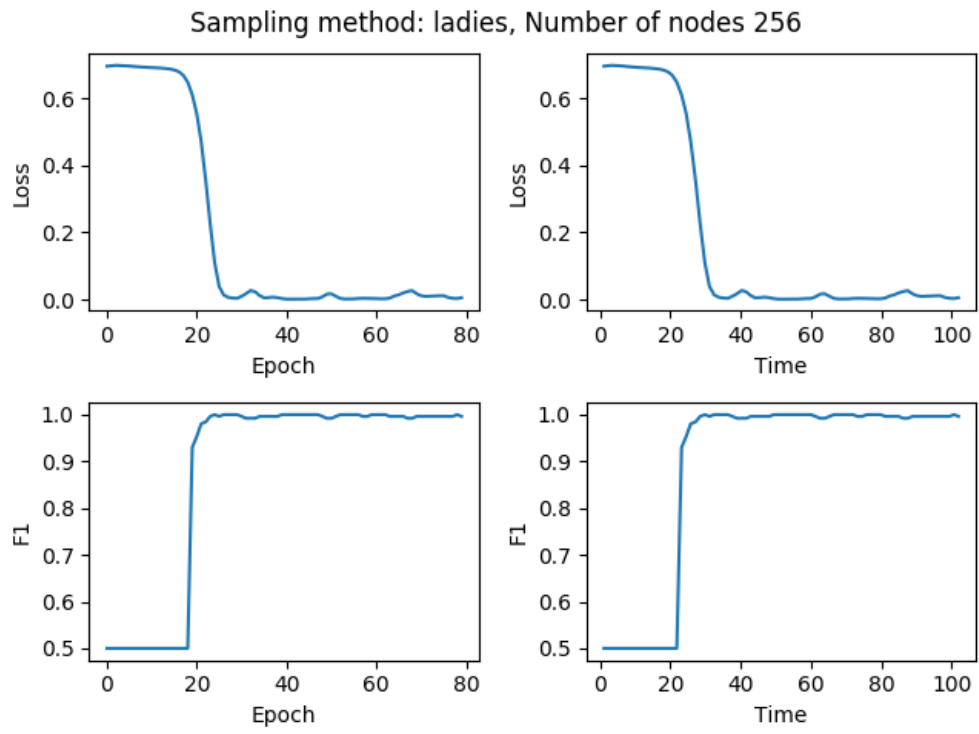


Figure B.3: Ladies with 256 nodes

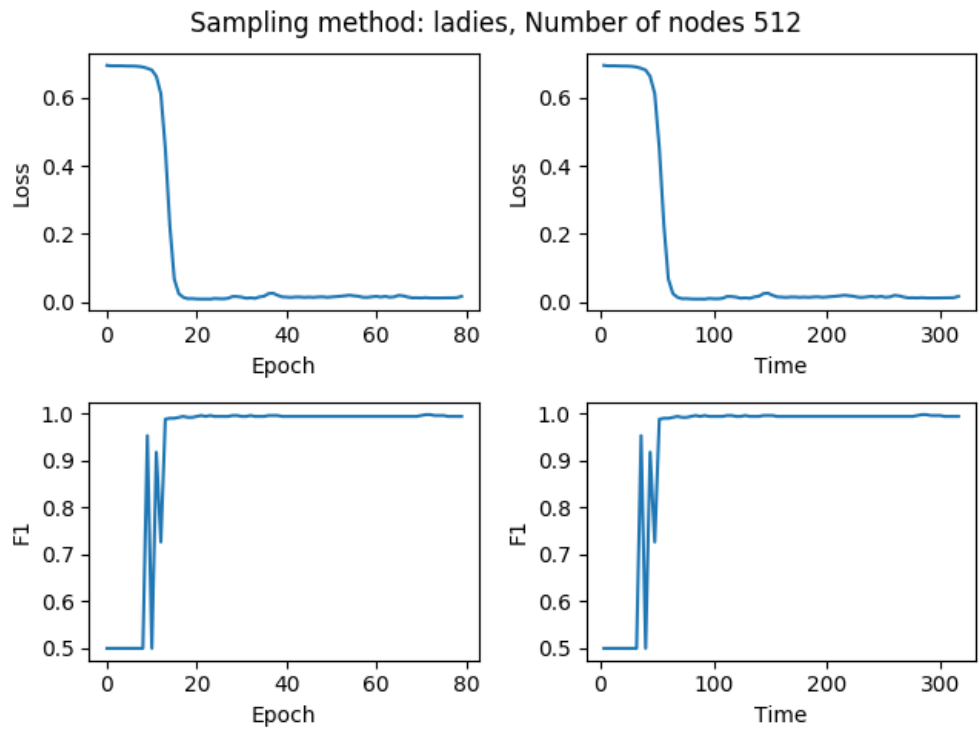


Figure B.4: Ladies with 512 nodes

Sampling method: ladies, Number of nodes 1024

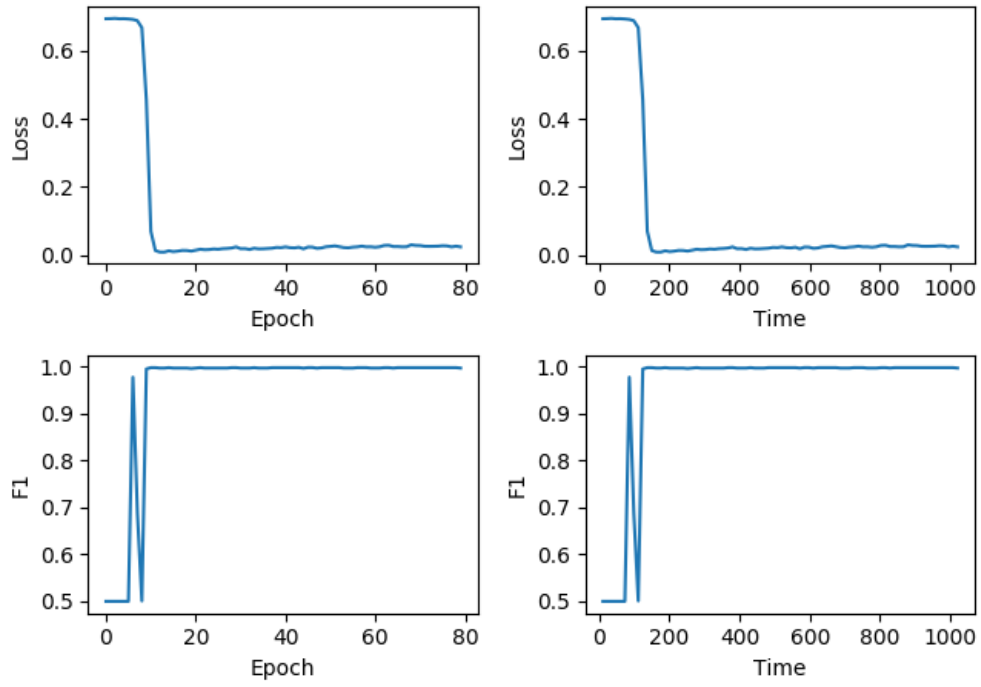


Figure B.5: Ladies with 1024 nodes

Sampling method: full, Number of nodes 64

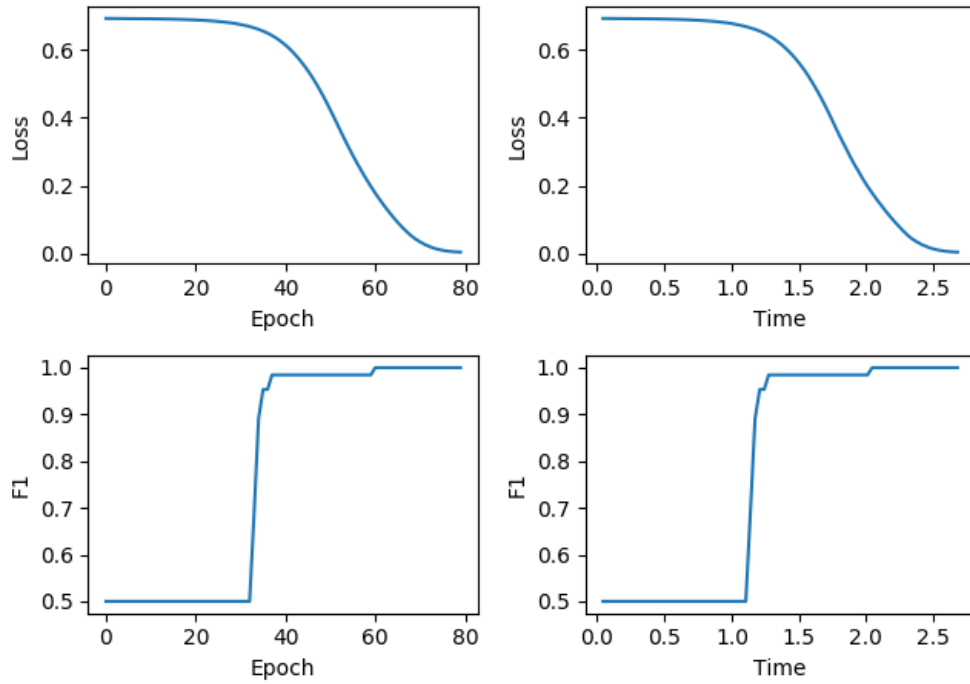


Figure B.6: Full GCN with 64 nodes



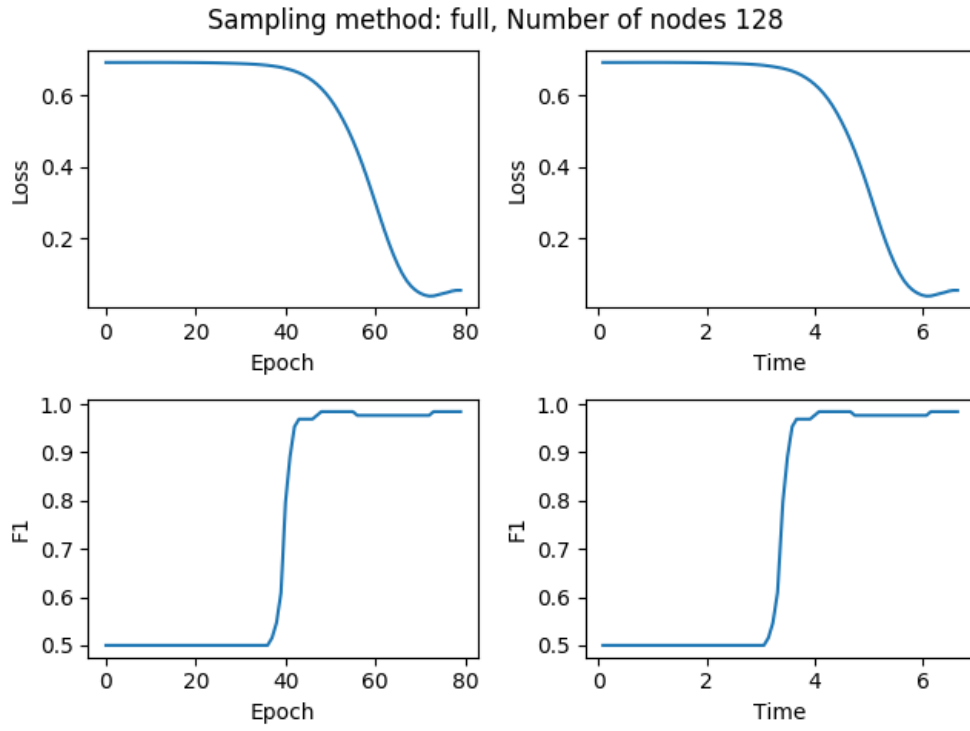


Figure B.7: Full with 128 nodes

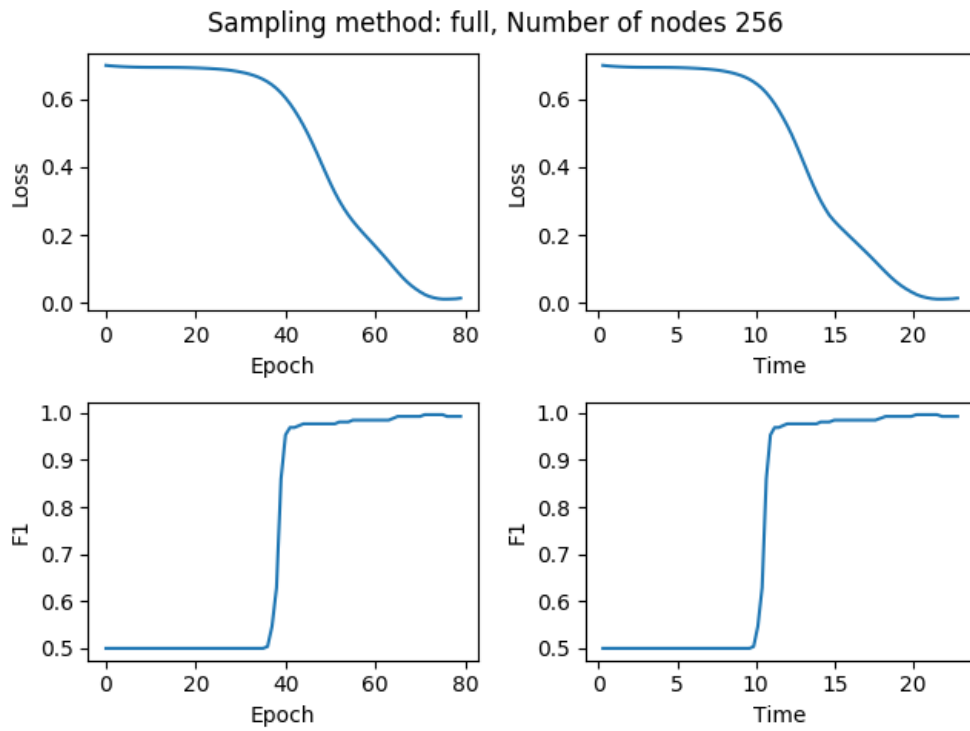


Figure B.8: Full GCN with 256 nodes

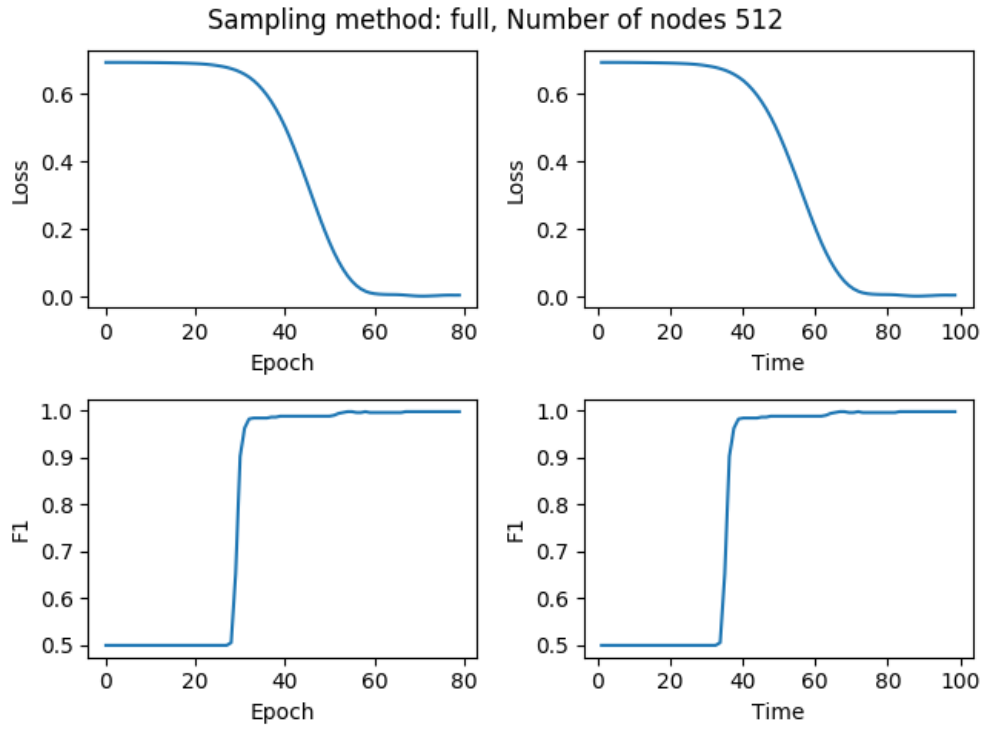


Figure B.9: Full GCN with 512 nodes

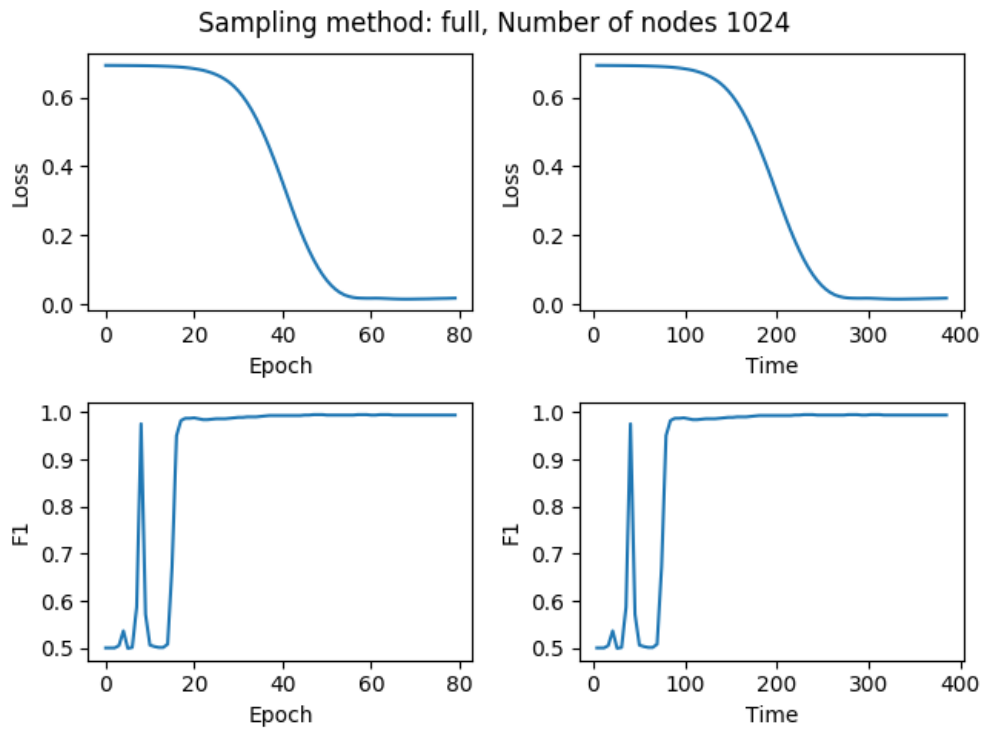


Figure B.10: Full GCN with 1024 nodes