Evaluating Graph Sampling Methods for Graph Attention Networks on Citation Networks

Ayush Kumar, Nick Puglisi

December 17, 2021

Contents

1 Introduction

Graph Convolutional Network (GCNs) have been a great contribution to the field of network data ever since Thomas Kipf and Max Welling published the paper: Semi-Supervised Classification With Graph Convolutional Networks [4]. The main idea behind the paper is that since graph structures do not exhibit euclidean geometry, standard convolutions that might be used for image recognition will not translate onto graph structures as well. So, through the use of Laplacian re-normalization trick presented in the GCN paper, classification accuracy has increased in comparison to other similar methods. However, a major tenet in the field of machine learning is that there is no one best method to employ for each and every problem. So, we present an investigation into both sampling techniques combined with Graphical Attention Networks (GATs).

1.1 Motivation

What sparked the investigation presented here was an observation made from the methodology in the original GCNs paper. Throughout model training, Kipf and Welling employed random dropout of nodes to introduce stochasticity during gradient descent. However, the use of random dropout only allows for updates to occur once per epoch while requiring the full data set to be loaded for every training iteration. So, some questions arose on if sampling methods could be employed over random dropout to increase efficiency when combined with the Graphical Attention Networks

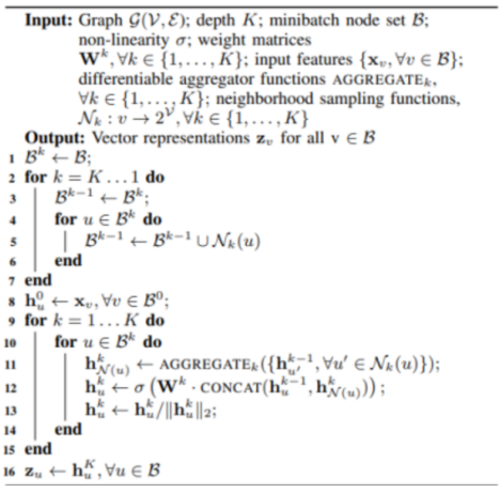
1.2 Overview of Graph Attention Networks

One shortcoming of the GCNs methodology is that it assumes equal importance of neighboring nodes. While some network structures might allow for an assumption like this to be made, other network structures might not allow for this assumption. So in the paper, Graphical Attention Networks [**?**], the authors seek to address this by leveraging self-attentional layers to enable different weights to be assigned across a given cluster of nodes. The GATs method also used dropout to introduce stochasticity and pushed results that successfully achieved or beat other methods of node classification, all while removing the need for equal importance.

1.3 Overview of Graph Sampling Methods

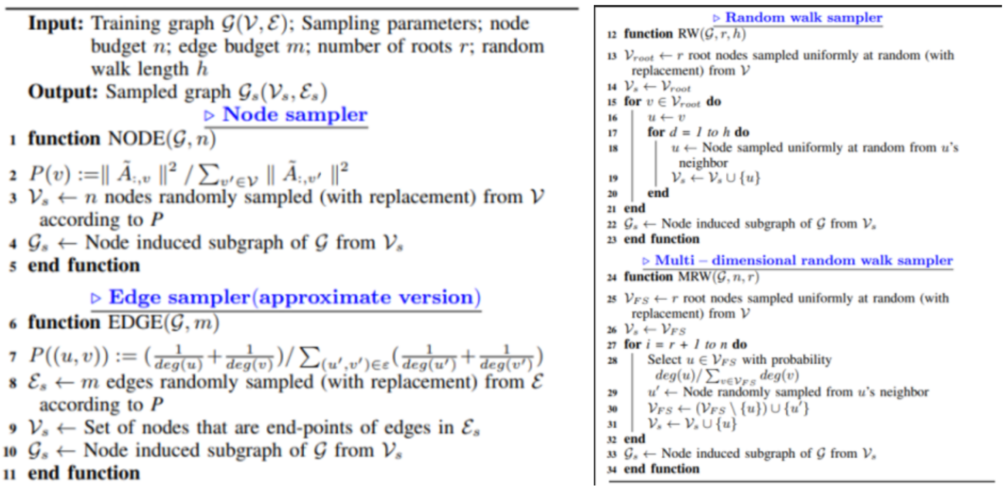
Both GCNs and GATs methodology employed the use of dropout during model training. However, as the size of a network increases, it becomes much more computationally expensive to train a model, for full-batch training only allows for parameters to update once per epoch. This sharp increase in power needed for model training has thus created a need for a way to minimize both storage costs and time spent. In the paper Sampling Methods for Efficient Training of Graph Convolutional Networks: A Survey [5], the authors survey and outline a whole host of sampling methods along with their respective algorithms that promise an increases in training efficiency. The methods outlined fall into two categories, namely: Layer-wise and subgraph-based sampling. One of the downsides of using sampling methods is that a bias-variance trade-off will be introduced. Yet, motivated by both GATs and the sampling methods applied to GCNs, we ask: is there any gain to applying sampling to the Graphical Attention Networks?

We will begin by selecting three different sampling methods to apply to the GATs, namely: GraphSAGE, GraphSAINT, and ClusterGCN. The first, GraphSAGE, trains the model through an inductive process in which neighborhood sampling is undertaken. That is, sampling is carried out by selecting neighboring nodes for each node in the graph followed by aggregation. This greatly reduces the computational costs since only the sampled nodes need be loaded. The Figure below outlines the algorithm for GraphSAGE:



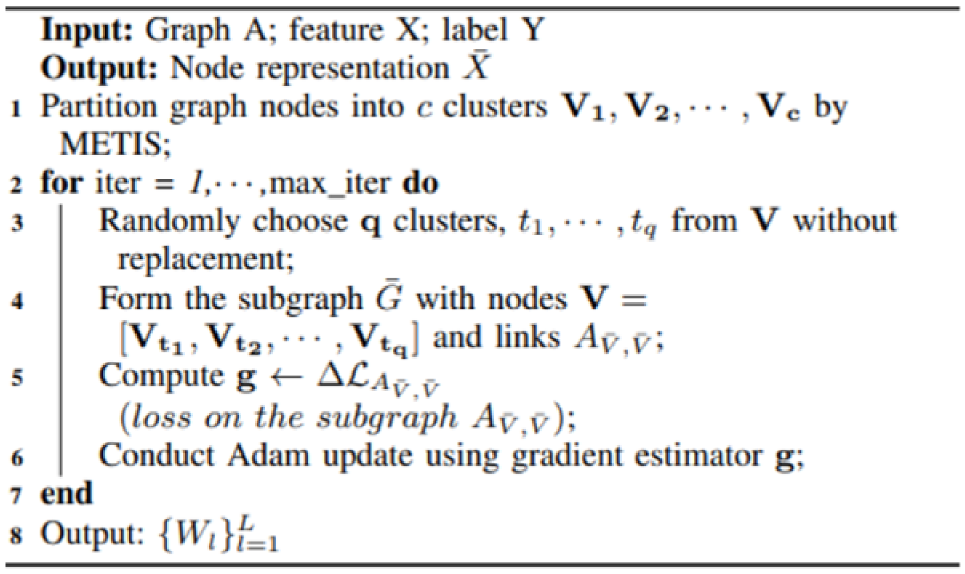
```
Input: Graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; depth $K$; minibatch node set $\mathcal{B}$;
       non-linearity $\sigma$; weight matrices
       $\mathbf{W}^k, \forall k \in \{1,\dots,K\}$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
       differentiable aggregator functions $\text{AGGREGATE}_k$,
       $\forall k \in \{1,\dots,K\}$; neighborhood sampling functions,
       $\mathcal{N}_k : v \to 2^{\mathcal{V}}, \forall k \in \{1,\dots,K\}$
Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{B}$
$\mathcal{B}^k \leftarrow \mathcal{B}$;
for $k = K \dots 1$ do
    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$;
    for $u \in \mathcal{B}^k$ do
        $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$
    end
end
$\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$;
for $k = 1 \dots K$ do
    for $u \in \mathcal{B}^k$ do
        $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\})$;
        $\mathbf{h}_u^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k)\right)$;
        $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$;
    end
end
$\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$
```

The next sampling method selected for application is GraphSAINT [8]. GraphSAINT deploys a sampler used to estimate the probability of nodes and edges being sampled. Now, in each batch, subgraphs deemed to be appropriate according to the sampler are selected and a full GCN is then built and trained. The figure below outlines the sampling algorithm eployed by GraphSAINT:



```
Input: Training graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; Sampling parameters; node
       budget $n$; edge budget $m$; number of roots $r$; random
       walk length $h$
Output: Sampled graph $\mathcal{G}_s(\mathcal{V}_s,\mathcal{E}_s)$
                    ▷ Node sampler
function NODE($\mathcal{G}, n$)
$P(v) := \| \tilde{A}_{:,v} \|^2 / \sum_{v' \in \mathcal{V}} \| \tilde{A}_{:,v'} \|^2$
$\mathcal{V}_s \leftarrow n$ nodes randomly sampled (with replacement) from $\mathcal{V}$
    according to $P$
$\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
end function
                    ▷ Edge sampler(approximate version)
function EDGE($\mathcal{G}, m$)
$P((u,v)) := (\frac{1}{deg(u)} + \frac{1}{deg(v)}) / \sum_{(u',v') \in \mathcal{E}} (\frac{1}{deg(u')} + \frac{1}{deg(v')})$
$\mathcal{E}_s \leftarrow m$ edges randomly sampled (with replacement) from $\mathcal{E}$
    according to $P$
$\mathcal{V}_s \leftarrow$ Set of nodes that are end-points of edges in $\mathcal{E}_s$
$\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
end function
                    ▷ Random walk sampler
function RW($\mathcal{G}, r, h$)
$\mathcal{V}_{root} \leftarrow r$ root nodes sampled uniformly at random (with
    replacement) from $\mathcal{V}$
$\mathcal{V}_s \leftarrow \mathcal{V}_{root}$
for $v \in \mathcal{V}_{root}$ do
    $u \leftarrow v$
    for $d = 1$ to $h$ do
        $u \leftarrow$ Node sampled uniformly at random from $u$'s
            neighbor
        $\mathcal{V}_s \leftarrow \mathcal{V}_s \cup \{u\}$
    end
end
$\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
end function
                    ▷ Multi – dimensional random walk sampler
function MRW($\mathcal{G}, n, r$)
$\mathcal{V}_{FS} \leftarrow r$ root nodes sampled uniformly at random (with
    replacement) from $\mathcal{V}$
$\mathcal{V}_s \leftarrow \mathcal{V}_{FS}$
for $i = r + 1$ to $n$ do
    Select $u \in \mathcal{V}_{FS}$ with probability
        $deg(u) / \sum_{v \in \mathcal{V}_{FS}} deg(v)$
    $u' \leftarrow$ Node randomly sampled from $u$'s neighbor
    $\mathcal{V}_{FS} \leftarrow (\mathcal{V}_{FS} \setminus \{u\}) \cup \{u'\}$
    $\mathcal{V}_s \leftarrow \mathcal{V}_s \cup \{u\}$
end
$\mathcal{G}_s \leftarrow$ Node induced subgraph of $\mathcal{G}$ from $\mathcal{V}_s$
end function
```

The final sampling method employed for this analysis is ClusterGCN [1]. As the name of the method suggests, ClusterGCN employs the use of sampling onto subgraphs and clusters. A given graph is first partitioned into multiple clusters that then get randomly sampled as a batch to form a subgraph. Finally, each iteration of training is carried out on a subgraph previously generated. Since only a subgraph needs to be loaded for each training iteration, the overall memory requirement scales well! The figure below outlines the training algorithm employed by the ClusterGCN sampling method:



```
Input: Graph A; feature X; label Y
Output: Node representation $\bar{X}$
Partition graph nodes into $c$ clusters $\mathbf{V_1}, \mathbf{V_2}, \cdots, \mathbf{V_c}$ by
  METIS;
for iter = $1, \cdots,$max_iter do
    Randomly choose q clusters, $t_1, \cdots, t_q$ from V without
      replacement;
    Form the subgraph $\bar{G}$ with nodes $\mathbf{V} =$
      $[\mathbf{V_{t_1}}, \mathbf{V_{t_2}}, \cdots, \mathbf{V_{t_q}}]$ and links $A_{\bar{V},\bar{V}}$;
    Compute $\mathbf{g} \leftarrow \Delta \mathcal{L}_{A_{\bar{V},\bar{V}}}$
      (loss on the subgraph $A_{\bar{V},\bar{V}}$);
    Conduct Adam update using gradient estimator g;
end
Output: $\{W_l\}_{l=1}^{L}$
```

2 Testing and Evaluation

2.1 Testing Setup

The sampling methods were tested on three citation networks originally featured in the original Graph Attention Networks paper: Cora, Citeseer, and PubMed. The following table summarizes the key characteristics of the datasets.

	Cora	Citeseer	PubMed
# Nodes	2708	3327	19717
# Edges	5429	4732	44338
# Features	1433	3703	500
# Classes	7	6	3
# Training Nodes	140	120	60
# Validation Nodes	500	500	500
# Test Nodes	1000	1000	1000

The same architecture was used for all the datasets, and they were tested using the following sampling methods described in the literature review: Random Node Sampler, ClusterGCN, GraphSAGE Mini-Batch Sampling, GraphSAINT Node Sampling, GraphSAINT Edge Sampling, GraphSAINT Random Walk Sampling.

2.2 Results

Below are the results of the sampling methods, based on average testing accuracy over four trials. Comparable results have been bolded.

	Cora	CiteSeer	PubMed
Baseline (No Sampling)	**82.0%**	**69.9%**	**77.0%**
Random Node Sampler	77.6%	65.2%	73.4%
ClusterGCN	**80.8%**	**70.7%**	**77.1%**
GraphSAGE	59.3%	**70.9%**	**76.3%**
GraphSAINT Node Sampler	60.9%	34.2%	26.4%
GraphSAINT Edge Sampler	70.7%	50.5%	55.7%
GraphSAINT Random Walk Sampler	**79.9%**	68.6%	75.3%

3 Conclusion

Based on experimental results we observe that sampling methods can be effectively utilized to match or even surpass performance of baseline models. We also observe that while some sampling methods are all around excellent like ClusterGCN, others performance varies greatly with the dataset, like GraphSAGE or the GraphSAINT Random Walk Sampler. Since Graph Attention Networks depend on embedding based on neighbors it is not surprising that subgraph based methods that preserve neighborhood outperformed stochastic edge or node-based methods. There is no singular best sampling method in any case, and the best method may vary by dataset size and structure. There is no free lunch when it comes to sampling for Graph Attention Networks.

Effective sampling for GATs opens the door for future research in efficient computation on much larger datasets without exploding training times, and allowing for multiple gradient updates every epoch. Matching the performance of the baseline method is a massive victory for computational efficiency. Baseline methods were trained for 1000 epochs while the sampling methods were only trained for 100 epochs, and many converged even sooner.

References

[1] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, Jul 2019.

[2] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[3] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[4] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[5] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. Sampling methods for efficient training of graph convolutional networks: A survey, 2021.

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[7] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. accepted as poster.

[8] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020.

A Code

This is the code used to run the Cora sampling procedures. Other datasets were used by simply changing the dataset name in the initial definition. For this reason, only the Cora sampling script is included in this document.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

from torch_geometric.data import Data
from torch_geometric.nn import GATConv
from torch_geometric.datasets import Planetoid
import torch_geometric.transforms as T
from torch_geometric.loader import ClusterData, ClusterLoader
from torch_geometric.loader import NeighborLoader
from torch_geometric.loader import GraphSAINTNodeSampler, GraphSAINTEdgeSampler,
    GraphSAINTRandomWalkSampler
from torch_geometric.loader import RandomNodeSampler

import tqdm

def main():
    baseline()

    # graphsaint_RandomNodeSampler()
    dataset = Planetoid(root='../data/', name='Cora', transform=T.NormalizeFeatures
        ())
    data = dataset[0]

    random_loader = RandomNodeSampler(data, num_parts=10)
    sampling(random_loader, "RandomNodeSampler")

    cluster_data = ClusterData(data, num_parts=32)
    cluster_loader = ClusterLoader(cluster_data, batch_size=128, shuffle=True,
        num_workers=4)
    sampling(cluster_loader, "ClusterGCN")

    neighbor_loader = NeighborLoader(data, num_neighbors=[20]*2, batch_size=128,
        input_nodes = data.train_mask)
    sampling(neighbor_loader, "GraphSAGE")

    gsaint_node_sampler = GraphSAINTNodeSampler(data, batch_size=32, num_steps=100)
    sampling(gsaint_node_sampler, "GraphSAINTNodeSampler")

    gsaint_edge_sampler = GraphSAINTEdgeSampler(data, batch_size=32, num_steps=100)
    sampling(gsaint_edge_sampler, "GraphSAINTEdgeSampler")

    gsaint_random_walk_sampler = GraphSAINTRandomWalkSampler(data, batch_size=32,
        num_steps=100, walk_length=32)
    sampling(gsaint_random_walk_sampler, "GraphSAINTRandomWalkSampler")

class GAT(torch.nn.Module):
    def __init__(self, dataset):
        super(GAT, self).__init__()
        self.hid = 8
        self.in_head = 8
        self.out_head = 1

        self.conv1 = GATConv(dataset.num_features, self.hid, heads=self.in_head,
            dropout=0.6)
        self.conv2 = GATConv(self.hid*self.in_head, dataset.num_classes, concat=
            False,
                             heads=self.out_head, dropout=0.6)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv1(x, edge_index)
        x = F.elu(x)
        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)

def baseline():
    dataset = Planetoid(root='../data/', name='Cora', transform=T.NormalizeFeatures
        ())
    data = dataset[0]
    #for reproducibility
    #torch.manual_seed(12345)
    #np.random.seed(12345)

    device = "cpu"
    model = GAT(dataset).to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=0.005, weight_decay=5e 4)
    criterion = torch.nn.CrossEntropyLoss()

    def test():
        model.eval()
        out = model(data)
        pred = out.argmax(dim=1)  # Use the class with highest probability.

        accs = []
        for mask in [data.train_mask, data.val_mask, data.test_mask]:
            correct = pred[mask] == data.y[mask]  # Check against ground truth
                labels.
            accs.append(int(correct.sum()) / int(mask.sum()))  # Derive ratio of
                correct predictions.
        return accs

    print("Baseline␣(No␣Sampling):")
    model.train()
    t = tqdm.trange(1, 1000, desc='Epoch␣1')
    for epoch in t:
        model.train()
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        train_acc, val_acc, test_acc = test()
        t.set_description(f'Epoch:␣{epoch:03d},␣Train:␣{train_acc:.4f},␣Val␣Acc:␣{
            val_acc:.4f},␣Test␣Acc:␣{test_acc:.4f}')

        loss.backward()
        optimizer.step()

def sampling(sampling_method, method_name):
    dataset = Planetoid(root='../data/', name='Cora', transform=T.NormalizeFeatures
        ())
    data = dataset[0]
    #for reproducibility
    #torch.manual_seed(12345)
    #np.random.seed(12345)

    device = "cpu"
    model = GAT(dataset).to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=0.005, weight_decay=5e 4)
    criterion = torch.nn.CrossEntropyLoss()

    def train():
        model.train()

        for sub_data in sampling_method:
            out = model(sub_data)
            loss = criterion(out[sub_data.train_mask], sub_data.y[sub_data.
                train_mask])
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

    def test():
        model.eval()
        out = model(data)
        pred = out.argmax(dim=1)  # Use the class with highest probability.

        accs = []
        for mask in [data.train_mask, data.val_mask, data.test_mask]:
            correct = pred[mask] == data.y[mask]  # Check against ground truth
                labels.
            accs.append(int(correct.sum()) / int(mask.sum()))  # Derive ratio of
                correct predictions.
        return accs

    print(f"{method_name}␣(Sampling):")
    t = tqdm.trange(1, 100, desc='Epoch␣1')
    for epoch in t:
        loss = train()
        train_acc, val_acc, test_acc = test()
        t.set_description(f'Epoch:␣{epoch:03d},␣Train:␣{train_acc:.4f},␣Val␣Acc:␣{
           val_acc:.4f},␣Test␣Acc:␣{test_acc:.4f}')

if __name__ == '__main__':
    main()
```