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. The main idea behind the paper is that since graph structures do not exhibit euclidean geometry, standard convoultions 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 stochasticiy 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, the authors of the paper Graphical Attention Networks 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 stochasticiy 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, the authors compile and outline a whole host of sampling methods along with their respective algorithms that offer increases in 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 GNCs, 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:

2 Testing and Evaluation

2.1 Testing Setup

The sampling methods were tested on three citation networks: Cora, CiteSeer, and PubMed.

2.2 Results

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

3 Conclusion

	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%

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()
```