Survey on GCN and GAT with Cora Dataset

Haoyang Ling

Feburary 28, 2022

1 Import

In this report, I use torch , torch_geometric, and torch_sparse to achieve GCN and GAT by simple implementation with torch.nn.Module and MessagePassing on **Cora** default benchmark.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch_geometric
import torch_sparse

from torch_geometric.datasets import Planetoid
from torch_geometric.loader import DataLoader

import numpy as np
from tqdm import trange, tqdm
import pandas as pd
import copy

import matplotlib.pyplot as plt
```

2 Basic Layer

2.1 GCNLayer

1. __init__ initializes GCNLayer with in_channels and out_channels
2. reset_parameters: to initialize the layer to avoid the bad performance during training with xavier_normal_, which balances the forward and backward propagation
3. forward: based on Github: tkipf/pygcn
4. reg_loss: involve the L2 regularization

```
class GCNLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels, bias=True):
        super(GCNLayer, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.weight = nn.Parameter(torch.Tensor(in_channels, out_channels))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_channels))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.xavier_normal_(self.weight.data)
        if self.bias is not None:
            nn.init.zeros_(self.bias.data)

    def forward(self, inputs, adj):
        out = torch_sparse.matmul(adj, inputs, reduce='add')
        return torch.mm(out, self.weight) + self.bias

    def reg_loss(self):
```

```
        return torch.sum(self.weight**2)

    def __repr__(self):
        return self.__class__.__name__ + '_from_' + \
            str(self.in_features) + '_to_' + str(self.out_features)
```

2.2 GAT

1. self.lin_l and self.lin_r to get the embedding of the node features
2. self.attr helps computes the attention weights of x_i and x_j
3. self.propagate will call the self-defined self.message and self.aggregate to update embedding
4. self.heads : multi-heads attention
5. **Some changes:** include eps in the message function
   a. **Change:** $x = x\_j + (1 + self.eps) * x\_i$
   b. **Reason:** this formula comes from the idea of GIN

```
class GAT(MessagePassing):
    def __init__(self, in_channels, out_channels, heads = 2,
                 negative_slope = 0.2, dropout = 0., eps=1e-2, **kwargs):
        super(GAT, self).__init__(node_dim=0, **kwargs)
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.heads = heads
        self.negative_slope = negative_slope
        self.dropout = dropout
        self.eps = eps

        self.lin_l = nn.Linear(self.in_channels, self.heads*self.out_channels)
        self.lin_r = self.lin_l

        self.attr = nn.Parameter(torch.Tensor(1, heads, out_channels))

        self.reset_parameters()

    def reset_parameters(self):
        nn.init.xavier_normal_(self.lin_l.weight)
        nn.init.xavier_normal_(self.lin_r.weight)
        nn.init.xavier_uniform_(self.attr)

    def forward(self, x, edge_index, size = None):

        H, C = self.heads, self.out_channels

        x_l = self.lin_l(x).view(-1, H, C)
        x_r = self.lin_r(x).view(-1, H, C)

        out = self.propagate(edge_index=edge_index, size=size, x=(x_l, x_r))
        out = out.view(-1, H*C)

        return out

    def message(self, x_j, x_i, index):
        x = x_j + (1+self.eps) * x_i
        x = F.leaky_relu(x, negative_slope=self.negative_slope)
        alpha_ij = (x * self.attr).sum(dim=-1)
        alpha_ij = F.dropout(softmax(alpha_ij, index), p=self.dropout, inplace=True, training=self.
            training)
        out = x_j * alpha_ij.unsqueeze_(-1)
        return out

    def aggregate(self, inputs, index):
        node_dim = self.node_dim
        out = torch_scatter.scatter(inputs, index, node_dim, reduce='sum')
        return out
```

3 Model

3.1 GCN Model with GCNLayer

1. num_layers is the number of layers in the model
2. dropout and reg to regularize the training
3. reg is included in the loss
4. self.pos_mp: the output layer for classification

```
class GCN(torch.nn.Module):
    def __init__(self, in_features, hidden_dims,
            num_layers, out_features, dropout, reg=0):
        super(GCN, self).__init__()
        assert num_layers >= 2

        self.in_features = in_features
        self.hidden_dims = hidden_dims
        self.num_layers = num_layers
        self.out_features = out_features
        self.dropout = dropout
        self.reg = reg

        self.layers = nn.ModuleList([GCNLayer(in_features, hidden_dims)])
        self.layers.extend([
            GCNLayer(hidden_dims, hidden_dims) for x in range(num_layers-1)
        ])
        self.post_mp = torch.nn.Sequential(
            nn.Linear(hidden_dims, hidden_dims), nn.Dropout(args.dropout),
            nn.Linear(hidden_dims, out_features, bias=False))

    def reset_parameters(self):
        for layer in self.layers:
            layer.reset_parameters()
        for layer in self.post_mp:
            if isinstance(layer, nn.Linear):
                nn.init.xavier_normal_(layer.weight.data)
                if self.bias:
                    nn.init.xavier_normal_(layer.bias.data)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        deg = torch_geometric.utils.degree(edge_index[0], data.num_nodes, dtype=x.dtype).to(x.device)
        value = 1/torch.sqrt(deg[edge_index[0]] * deg[edge_index[1]])
        adj = torch_sparse.SparseTensor(row=edge_index[0], col=edge_index[1], value=value,
                    sparse_sizes=(data.num_nodes, data.num_nodes))
        out = x
        for layer in self.layers:
            out = F.relu(layer(out, adj))
            out = F.dropout(out, self.dropout, training=self.training)
        return F.log_softmax(self.post_mp(out), dim=1)

    def loss(self, pred, label):
        loss = F.nll_loss(pred, label)
        if self.reg != 0:
            for layer in self.layers:
                loss += self.reg*layer.reg_loss()
            for layer in self.post_mp:
                if isinstance(layer, nn.Linear):
                    loss += self.reg*torch.sum(layer.weight.data**2)
        return loss
```

GAT model is similar to this, so it is omitted here.

4 Utils

1. Utils part refers to the CS224W Colab Homework 4 with some changes

4.1 Train

1. mini-batch implementation with epochs iteration
2. support choosing device with args.device

```
def train(dataset, args):
    device = args.device
    print("Node task. test set size:", np.sum(dataset[0]['test_mask'].numpy()))
    print()
    test_loader = loader = DataLoader(dataset,
                                      batch_size=args.batch_size, shuffle=True)
    # build model
    model = GCN(dataset.num_node_features, args.hidden_dim,
                args.num_layers, dataset.num_classes,
                dropout=args.dropout, reg=args.reg).to(device)
    opt = optim.Adam(model.parameters(), lr=args.lr,
                weight_decay=args.weight_decay)

    # train
    train_losses = []
    valid_accs = []
    test_accs = []
    best_acc = 0
    best_model = None
    for epoch in trange(args.epochs, desc="Training", unit="Epochs"):
        total_loss = 0
        model.train()
        for batch in loader:
            opt.zero_grad()
            pred = model(batch.to(device))
            label = batch.y.to(device)
            pred = pred[batch.train_mask]
            label = label[batch.train_mask]
            loss = model.loss(pred, label)
            loss.backward()
            opt.step()
            total_loss += loss.item() * batch.num_graphs
        total_loss /= len(loader.dataset)
        train_losses.append(total_loss)

        if epoch % 10 == 0:
            valid_acc = test(loader, model, device, is_validation=True)
            valid_accs.append(valid_acc)
            test_acc = test(test_loader, model, device)
            test_accs.append(test_acc)
            if test_acc > best_acc:
                best_acc = test_acc
                best_model = copy.deepcopy(model)
        else:
            valid_accs.append(valid_accs[-1])
            test_accs.append(test_accs[-1])

    return valid_accs, test_accs, train_losses, best_model, best_acc, test_loader
```

4.2 Test

```
def test(loader, test_model, device, is_validation=False,
        save_model_preds=False, model_type=None):

    test_model.eval()

    correct = 0
    # Note that Cora is only one graph!
    for data in loader:
        with torch.no_grad():
            # max(dim=1) returns values, indices tuple; only need indices
            pred = test_model(data.to(device)).max(dim=1)[1]
            label = data.y.to(device)

        mask = data.val_mask if is_validation else data.test_mask
        # node classification: only evaluate on nodes in test set
```

```
        pred = pred[mask]
        label = label[mask]

        if save_model_preds:
            print ("Saving Model Predictions for Model Type", model_type)

            data = {}
            data['pred'] = pred.view(-1).cpu().detach().numpy()
            data['label'] = label.view(-1).cpu().detach().numpy()

            df = pd.DataFrame(data=data)
            # Save locally as csv
            df.to_csv('CORA-Node-' + model_type + '.csv', sep=',', index=False)

        correct += pred.eq(label).sum().item()

    total = 0
    for data in loader.dataset:
        total += torch.sum(data.val_mask if is_validation else data.test_mask).item()

    return correct / total
```

4.3 Setup Seed

1. assure the replication of the result 2. **issue**: small differences in each training but acceptable

```
def setup_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

4.4 Objectview

```
class objectview(object):
    def __init__(self, d):
        self.__dict__ = d
```

4.5 Main function

```
if __name__=="__main__":
    # setup the deterministic operation
    setup_seed(22)

    model_type = args.model_type
    plt.style.use('seaborn-paper')

    # load the cora dataset
    if args.dataset == 'cora':
        dataset = Planetoid(root='/tmp/cora', name='Cora')
    else:
        raise NotImplementedError("Unknown dataset")
    valid_accs, test_accs, losses, best_model, best_acc, test_loader = train(dataset, args)

    print("Maximum valid set accuracy: {0}".format(max(valid_accs)))
    print("Maximum test set accuracy: {0}".format(max(test_accs)))
    print("Minimum loss: {0}".format(min(losses)))

    # Run test for our best model to save the predictions!
    test(test_loader, best_model, args.device,
         is_validation=False, save_model_preds=True, model_type=model_type)
    print()
    plt.title(dataset.name)
    plt.plot(losses, label="training loss" + " - " + args.model_type)
    plt.plot(valid_accs, label="valid accuracy" + " - " + args.model_type)
```

```
plt.plot(test_accs, label="test_accuracy" + " - " + args.model_type)
plt.legend()
plt.savefig(f"result_{model_type}.jpg", dpi=300)
plt.close()

torch.save(best_model, f"best_model_{model_type}.pth")
```

5 Environment Setup

Device: GeForce RTX 2080 Ti
Dataset: Cora from Planetoid

CLASS Planetoid (**root**: str, **name**: str, **split**: str = `'public'`, **num_train_per_class**: int = `20`, **num_val**: int = `500`, **num_test**: int = `1000`, **transform**: Optional[Callable] = `None`, **pre_transform**: Optional[Callable] = `None`) [source]

The citation network datasets "Cora", "CiteSeer" and "PubMed" from the "Revisiting Semi-Supervised Learning with Graph Embeddings" paper. Nodes represent documents and edges represent citation links. Training, validation and test splits are given by binary masks.

5.1 GCN

```
args = {
    'device': 'cuda:0',
    'model_type': 'GCN',
    'dataset': 'cora',
    'num_layers': 2,
    'batch_size': 32,
    'hidden_dim': 32,
    'dropout': 0.5,
    'epochs': 500,
    'weight_decay': 5e-3,
    'lr': 1e-2,
    'reg': 5e-4
}
```

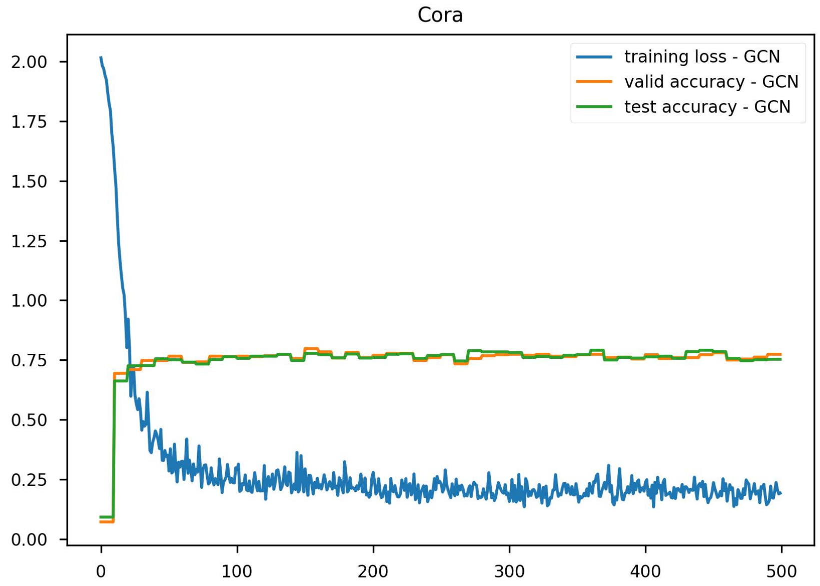
5.2 GAT

```
args = {
    'device': 'cuda:0',
    'model_type': 'GAT',
    'dataset': 'cora',
    'num_layers': 2,
    'batch_size': 64,
    'hidden_dim': 32,
    'dropout': 0.6,
    'epochs': 500,
    'opt': 'adam',
    'opt_scheduler': 'none',
    'opt_restart': 0,
    'weight_decay': 5e-3,
    'lr': 1e-3,
    'heads': 8,
    'reg': 5e-4
},
```

6 Results

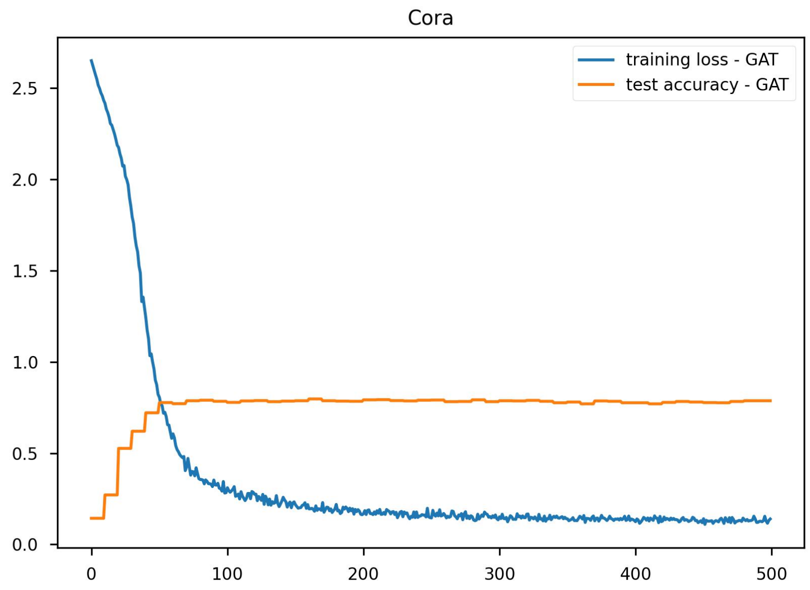
6.1 GCN

```
Node task. test set size: 1000

Training: 100%|████████████████████████████████████████| 500/500 [00:05<00:00, 95.92Epochs/s]
Best model train accuracy: 1.0
Maximum valid set accuracy: 0.798
Maximum test set accuracy: 0.791
Minimum loss: 0.13484381139278412
Saving Model Predictions for Model Type GCN
```



6.2 GAT

```
Multi-heads:  8
Node task. test set size: 1000

Training: 100%|████████████████████████████████████████| 500/500 [00:07<00:00, 68.49Epochs/s]
Maximum valid set accuracy: 0.796
Maximum test set accuracy: 0.812
Minimum loss: 0.10460884124040604
Saving Model Predictions for Model Type GAT
```



6.3 Comparison

In this part, the result above will be compared with the results summarized in the paper "A Comprehensive Survey on Graph Neural Networks" by Wu, et al shown below

Method	Cora	Citeseer	Pubmed	PPI	Reddit
SSE (2018)	-	-	-	83.60	-
GCN (2016)	81.50	70.30	79.00	-	-
Cayleynets (2017)	81.90	-	-	-	-
DualGCN (2018)	83.50	72.60	80.00	-	-
GraphSage (2017)	-	-	-	61.20	95.40
GAT (2017)	83.00	72.50	79.00	97.30	-

6.4 Reflection

1. The layer normalization seems to have few impacts on the results.
2. Initialization may affect the result about 2-3 %.
3. Setting the last layer without bias may help to improve the performance.

6.5 Conclusions

By implementing it, the code almost achieves the original performance within 2-3%. And, it helps me understand the mechanics of torch_geometric.nn.MessagePassing and torch.nn.Module.