[Code系列 01] GraphSage 学习笔记

原创 勃艮第红 贤成法狮 2018-11-03

Mr. Curiosity

Jason Mraz - Mr. A-Z

Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." ***Advances in Neural Information Processing Systems*** **. 2017.**

https://github.com/williamleif/graphsage-simple

> 碎片前言

动手能力不行，故新开 Code 系列，多练一下。

1 GraphSage

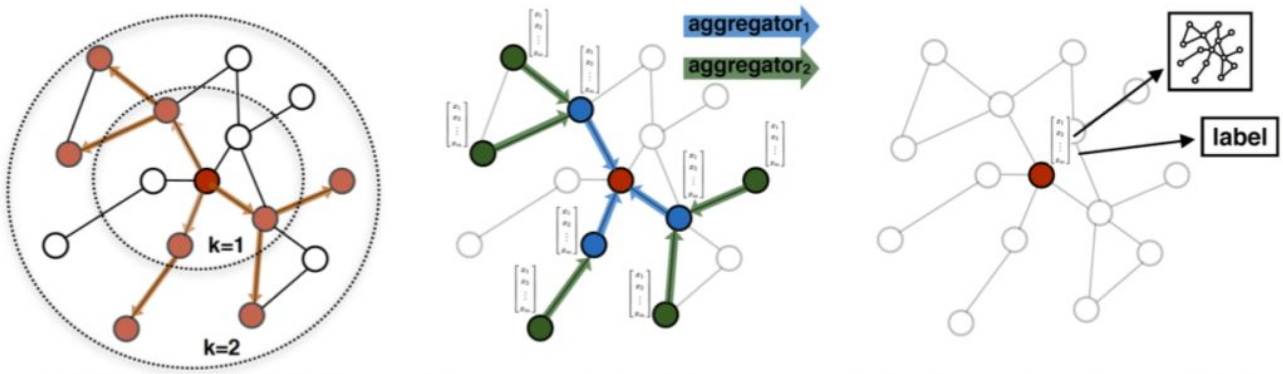

Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

2 Code

2.1 Aggregator

把节点 *u* 邻居节点的特征 aggregate 得到 neigh_feats:

$$\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v))$$

```
num_samples = 5
batch_nodes = [1, 2, 3, 4, 5, 6, 7, 8, 9]

agg =  MeanAggregator(features, cuda=False)

neigh_feats = agg.forward(nodes=nodes,
                          to_neighs=[adj_lists[node] for node in batch_nodes],
                          num_sample=num_samples)
```

给定 num_samples ， 对每个节点 aggregator 得到相应 samp_neighs，如果 gcn 则加上 self-loop。

MeanAggregator 根据 node 的 samp_neigh 得到最简单的 mean features。

```
import torch
import torch.nn as nn
from torch.autograd import Variable

import random

class MeanAggregator(nn.Module):
    def __init__(self, features, cuda=False, gcn=False):
        super(MeanAggregator, self).__init__()
        self.features = features
```

```
        self.cuda = cuda
        self.gcn = gcn

    def forward(self, nodes, to_neighs, num_sample=10):
        """
        nodes -- list of nodes in a batch
        to_neighs -- list of sets, each set is the set of neighbors for node in batch
        """
        if num_sample is not None:
            samp_neighs = [set(random.sample(to_neigh, num_sample)) if len(to_neigh) >= num_sar
        else:
            samp_neighs = to_neighs

        # self-loop for gcn
        if self.gcn:
            samp_neighs = [samp_neigh + set([nodes[i]]) for i, samp_neigh in enumerate(samp_ne:

        unique_nodes_list = list(set.union(*samp_neighs))
        unique_nodes = {n:i for i,n in enumerate(unique_nodes_list)}
        mask = Variable(torch.zeros(len(samp_neighs), len(unique_nodes)))
        column_indices = [unique_nodes[n] for samp_neigh in samp_neighs for n in samp_neigh]
        row_indices = [i for i in range(len(samp_neighs)) for j in range(len(samp_neighs[i]))]
        mask[row_indices, column_indices] = 1
        if self.cuda:
            mask = mask.cuda()
        num_neigh = mask.sum(1, keepdim=True)
        mask = mask.div(num_neigh)
        if self.cuda:
            embed_matrix = self.features(torch.LongTensor(unique_nodes_list).cuda())
        else:
            embed_matrix = self.features(torch.LongTensor(unique_nodes_list))
        to_feats = mask.mm(embed_matrix)
        return to_feats
```

2.2 Encoder

把自身和邻居的 features 拼起来，得到新的特征：

```
weight = nn.Parameter(torch.FloatTensor(embed_dim, feat_dim))

neigh_feats = agg.forward(nodes, to_neighs, num_samples)
self_feats = features(torch.LongTensor(nodes))
combined = torch.cat([self_feats, neigh_feats], dim=1)
```

```
combined = F.relu(weight.mm(combined.t()))
```

$$\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \mathrm{CONCAT}\left(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k\right)\right)$$

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

from torch.nn import init

class Encoder(nn.Module):
    def __init__(self, features, feature_dim, embed_dim, adj_lists,
                 aggregator, num_sample=10, base_model=None, gcn=False,
                 cuda=False, feature_transform=False):
        super(Encoder, self).__init__()

        self.features = features
        self.feat_dim = feature_dim
        self.adj_lists = adj_lists
        self.aggregator = aggregator
        self.num_sample = num_sample
        if base_model != None:
            self.base_model = base_model

        self.gcn = gcn
        self.embed_dim = embed_dim
        self.cuda = cuda
        self.aggregator.cuda = cuda
        self.weight = nn.Parameter(
            torch.FloatTensor(embed_dim, self.feat_dim if self.gcn else 2 * self.featdim))
        init.xavier_uniform_(self.weight)

    def forward(self, nodes):
        """
        Generate embeddings for a batch of nodes.
        """
        neigh_feats = self.aggregator.forward(nodes, [self.adj_lists[int(node)] for node in noc
        if not self.gcn:
            if self.cuda:
                self_feats = self.features(torch.LongTensor(nodes).cuda())
            else:
                self_feats = self.features(torch.LongTensor(nodes))
            combined = torch.cat([self_feats, neigh_feats], dim=1)
        else:
            combined = neigh_feats
```

```
        combined = F.relu(self.weight.mm(combined.t()))
        return combined
```

2.3 GraphSage

等于经过 encoder 后，通过一个 linear 层得到输出，CrossEntropy 作为 loss。

```
import torch.nn as nn
from torch.nn import init

class SupervisedGraphSage(nn.Module):
    def __init__(self, num_classes, enc):
        super(SupervisedGraphSage, self).__init__()
        self.enc = enc
        self.xent = nn.CrossEntropyLoss()

        self.weight = nn.Parameter(torch.FloatTensor(num_classes, enc.embed_dim))
        init.xavier_uniform_(self.weight)

    def forward(self, nodes):
        embeds = self.enc(nodes)
        scores = self.weight.mm(embeds)
        return scores.t()

    def loss(self, nodes, labels):
        scores = self.forward(nodes)
        return self.xent(scores, labels.squeeze())
```

2.4 Load data

```
import numpy as np
from collections import defaultdict

def load_cora():
    nb_nodes = 2708
    nb_feats = 1433
    feat_data = np.zeros((nb_nodes, nb_feats))
    labels = np.empty((nb_nodes, 1), dtype=np.int64)
    node_map = {}  # paper dict
    label_map = {}  # label dict
    with open("cora/cora.content") as fp:
        for i, line in enumerate(fp):
            info = line.strip().split()
            feat_data[i, :] = list(map(np.float32, info[1:-1]))
            node_map[info[0]] = i
            if info[-1] not in label_map:
                label_map[info[-1]] = len(label_map)
```

```
            labels[i] = label_map[info[-1]]

    adj_lists = defaultdict(set)
    with open('cora/cora.cites') as fp:
        for i, line in enumerate(fp):
            info = line.strip().split()
            paper1 = node_map[info[0]]
            paper2 = node_map[info[1]]
            adj_lists[paper1].add(paper2)
            adj_lists[paper2].add(paper1)
    return feat_data, labels, adj_lists
```

2.5 Run Cora

两层的 aggregator，每层 aggregator 用到 1-hop neighs，最后等价于用到 2-hop neighs。

```
def run_cora():
    np.random.seed(1)
    random.seed(1)
    nb_nodes = 2708
    feat_data, labels, adj_lists = load_cora()

    # fixed embedding
    features = nn.Embedding(2708, 1433)
    features.weight = nn.Parameter(torch.FloatTensor(feat_data), requires_grad=False)

    agg1 = MeanAggregator(features, cuda=False)
    enc1 = Encoder(features, 1433, 128, adj_lists, agg1, gcn=True, cuda=False)
    agg2 = MeanAggregator(lambda nodes : enc1(nodes).t(), cuda=False)
    enc2 = Encoder(lambda nodes : enc1(nodes).t(), enc1.embed_dim, 128, adj_lists, agg2, base_r
    enc1.num_samples = 5
    enc2.num_samples = 5

    graphsage = SupervisedGraphSage(7, enc2)

    rand_indices = np.random.permutation(nb_nodes)
    test = rand_indices[:1000]
    val = rand_indices[1000:1500]
    train = list(rand_indices[1500:])

    optimizer = torch.optim.SGD(filter(lambda p : p.requires_grad, graphsage.parameters()), lr=

    times = []
    for batch in range(100):
        batch_nodes = train[:256]
        random.shuffle(train)
        start_time = time.time()
```

```
        optimizer.zero_grad()
        loss = graphsage.loss(batch_nodes, Variable(torch.LongTensor(labels[np.array(batch_node
        loss.backward()
        optimizer.step()
        end_time = time.time()
        times.append(end_time - start_time)
        print(batch, loss.data[0])

    val_output = graphsage.forward(val)
    print('Validation F1:', f1_score(labels[val], val_output.data.numpy().argmax(axis=1), avera
    print('Average batch time:', np.mean(times))
```