Author = Song Jian

#### 阅读论文： SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS

|  |  |  |
| --- | --- | --- |
| @article{Kipf2017,  author = {Kipf, Thomas N and Welling, Max},  pages = {1--14},  title = {{SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS}},  year = {2017}  } | [1] T. N. Kipf and M. Welling, “SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS,” pp. 1–14, 2017.  参考：[**http://tkipf.github.io/graph-convolutional-networks/**](http://tkipf.github.io/graph-convolutional-networks/)  源码：[<https://github.com/tkipf/gcn>](https://github.com/tkipf/gcn) | 阿姆斯特丹大学  CIFAR实验室 |

#### 文档由以下几个部分组成：

* 文章简述
* 理论背景
* 论文实验
* 代码部分

1文章简述

本篇文章的主要工作是将卷积扩展到图结构的数据中，能够得到比较好的数据表示，并且在半监督任务中也取得了不错的效果。

1.1半监督学习

比如对图（例如引文网络）中的节点（如引文网络中的文章）进行分类的问题，其中仅有一小部分节点有标签（即明确知道该节点属于哪一类），这个问题可以被定义为基于图的半监督学习。

1.2经典的方法

1. 基于平滑正则，即假设相邻的节点具有相似特征的(Zhu et al., 2003; Zhou et al., 2004; Belkin et al., 2006; Weston et al., 2012)，但是这种方法可能会限制建模能力，因为图的边不一定需要编码节点相似性，而可能包含其他信息
2. 基于embedding的方法，在embedding的结果上训练一个分类器，如DeepWalk，Node2Vec。这种方法由于两者分开，也就是学得的embedding针对该分类任务不一定是最优的。

因此本文：

1. 提出了一种卷积神经网络的变种，即提出了一种新的图卷积方法。
2. 使用谱图卷积（spectral graph convolution）的局部一阶近似，来确定卷积结构。
3. 所提出的的网络可以学习图上局部结构的特征，并进行编码。

2理论背景

2.1 GCN的符号定义

对于𝐺=(𝑉,𝐸)，我们有输入𝑋是一个𝑁 ×𝐷的矩阵(N表示节点的数量，D表示输入的特征数量)，表示节点的特征，同时有图的邻接矩阵 A。我们希望得到一个 𝑁 ×F的特征矩阵 Z，表示我们学得的每个节点的特征表示，其中 F是我们希望得到的表示的维度。对于每一个神经网络层来说，可以表示为：

其中

一个比较简单的例子，

其中σ(⋅)是一个非线性的激活函数如ReLU， 是第l层的权重矩阵。这个例子虽然简单但是却很强大，不过依然有缺陷。

对于，不难看出乘以邻接矩阵A就相当于对每个节点都加上了邻居节点的特征，但是没有增加自身的特征，因此需要闭环，即

由于A是没有正则化的，因此需要正则化，使得每一行的和为1。例如，D为度矩阵，文章中使用的是理论可以由谱图卷积以及经典的WL算法进行解释。

因此公式变成：

上述的变换，可以由谱图卷积以及经典的WL算法进行解释。

* 1. Weisfeiler-Lehman和WL Test

一维的Weisfeiler-Lehman如下所示：

对于任意的节点：

* 获取节点所有的邻居节点的特征
* 更新该节点的特征，hash是一个单射函数

重复以上步骤K次直到收敛。

事实上，Weisfeiler-Lehman算法在大多数图上会得到一个独一无二的特征集合，这意味着图上的每一个节点都有着独一无二的角色定位（例外在于网格，链式结构等等）。因此，对于大多数非规则的图结构，得到的特征可以作为图是否同构的判别依据，也就是WL Test。

#### 3论文实验

3.1

数据集：karate club

网络层数：3层，也就是卷积到三阶邻域(因为一层可以收集一阶邻居的特征，第二层再次收集一阶邻居特征的时候，此时的一阶邻居已经包含其第二阶邻居的特征，类推)

输入：邻接矩阵，随机的权重矩阵，特征矩阵(为单位矩阵，也就是没有输入任何节点的特征)

可以观察到在没有输入任何节点的特征以及没有进行任何更新的情况下的情况下，三层网络的embedding已经与社区网络结构相似了。

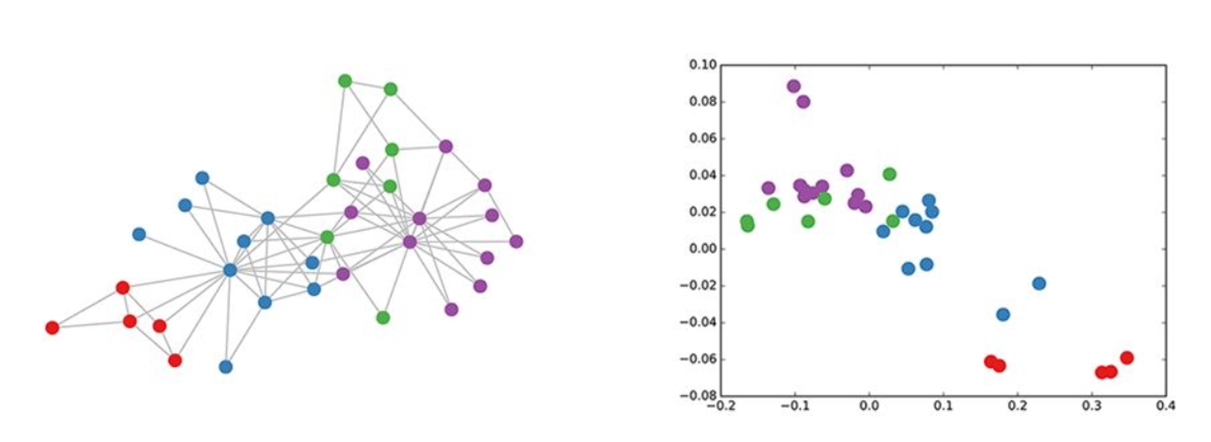
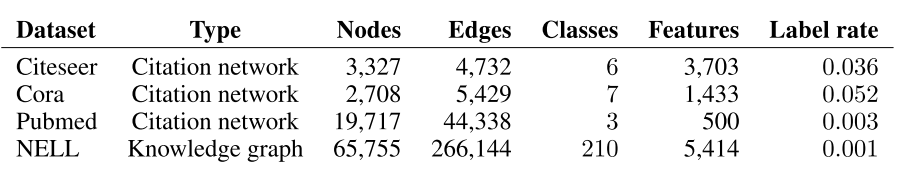


Fig1 原文实验

3.2半监督节点分类

数据集： Citeseer, Cora,Pubmed and NELL



Dataset statistics, as reported in Yang 1

本文使用了一个两层的GCN进行节点分类，模型构造如图。

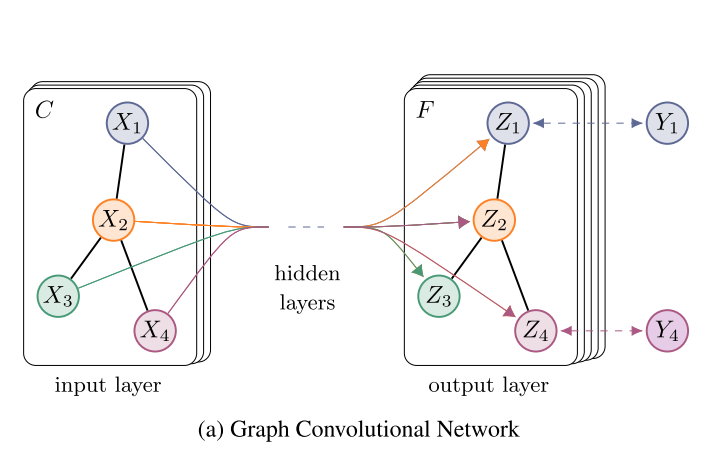


Fig2 原文实验

具体的流程为：

1. 首先获得节点的特征表示并计算邻接矩阵
2. 将其输入到一个两层的GCN中，得到每个标签的预测结果

其中为第1层的权值矩阵，用于将特征表示映射为相应的隐层状态，为第2层的权值矩阵，最后将每个节点的表示通过一个softmax，即可得到每个标签的预测结果。

效果：

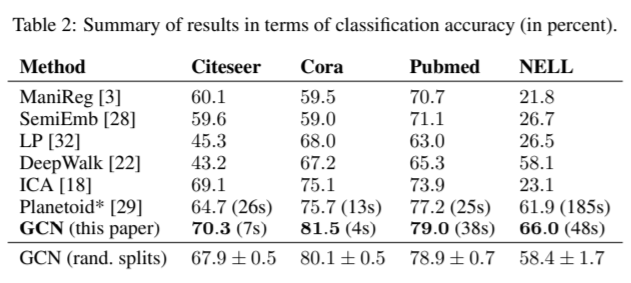


Fig3 原文实验结果

下面为文章后附带实验：

数据集：karate club

网络迭代次数：300

特征：未输入任何特征

效果视频：<http://tkipf.github.io/graph-convolutional-networks/images/video.mp4>

#### 代码部分

非源代码，仅为理解。

环境：Ubuntu，pgl，paddlepaddle

<https://github.com/PaddlePaddle/PGL>

导入所需要的包

**import** pgl

**from** pgl **import** data\_loader

**from** pgl.utils.logger **import** log

**import** paddle.fluid **as** fluid

**import** numpy **as** np

**import** time

**import** argparse

构建一个简单的GCN层

**import** paddle.fluid **as** fluid

**def** **gcn\_layer**(graph\_wrapper, node\_feature, hidden\_size, act):

**def** **send\_func**(src\_feat, dst\_feat, edge\_feat):

**return** src\_feat["h"]

**def** **recv\_func**(msg):

**return** fluid.layers.sequence\_pool(msg, "sum")

message = graph\_wrapper.send(send\_func, nfeat\_list=[("h", node\_feature)])

output = graph\_wrapper.recv(recv\_func, message)

output = fluid.layers.fc(output, size=hidden\_size, act=act)

**return** output

数据集

**def** **load**(name):

**if** name == 'cora':

dataset = data\_loader.CoraDataset()

**elif** name == "pubmed":

dataset = data\_loader.CitationDataset("pubmed", symmetry\_edges=False)

**elif** name == "citeseer":

dataset = data\_loader.CitationDataset("citeseer", symmetry\_edges=False)

**else**:

**raise** ValueError(name + " dataset doesn't exists")

**return** dataset

定义训练超参数

parser = argparse.ArgumentParser(description='GCN')

parser.add\_argument(

"--dataset", type=str, default="cora", help="dataset (cora, pubmed)")

parser.add\_argument("--use\_cuda", action='store\_true', help="use\_cuda")

args = parser.parse\_args(args=[])

log.info(args)

调用数据集

dataset = load(args.dataset)

归一化，-0.5为度矩阵的-0.5次方，对称归一化拉普拉斯矩阵的计算公式

indegree = dataset.graph.indegree()

norm = np.zeros\_like(indegree, dtype="float32")

norm[indegree > 0] = np.power(indegree[indegree > 0], -0.5)

dataset.graph.node\_feat["norm"] = np.expand\_dims(norm, -1)

定义GCN网络

place = fluid.CUDAPlace(0) **if** args.use\_cuda **else** fluid.CPUPlace()

train\_program = fluid.Program()

startup\_program = fluid.Program()

test\_program = fluid.Program()

hidden\_size = 16

**with** fluid.program\_guard(train\_program, startup\_program):

gw = pgl.graph\_wrapper.GraphWrapper(

name="graph",

place=place,

node\_feat=dataset.graph.node\_feat\_info())

output = pgl.layers.gcn(gw,

gw.node\_feat["words"],

hidden\_size,

activation="relu",

norm=gw.node\_feat['norm'],

name="gcn\_layer\_1")

output = fluid.layers.dropout(

output, 0.5, dropout\_implementation='upscale\_in\_train')

output = pgl.layers.gcn(gw,

output,

dataset.num\_classes,

activation=None,

norm=gw.node\_feat['norm'],

name="gcn\_layer\_2")

node\_index = fluid.layers.data(

"node\_index",

shape=[None, 1],

dtype="int64",

append\_batch\_size=False)

node\_label = fluid.layers.data(

"node\_label",

shape=[None, 1],

dtype="int64",

append\_batch\_size=False)

pred = fluid.layers.gather(output, node\_index)

loss, pred = fluid.layers.softmax\_with\_cross\_entropy(

logits=pred, label=node\_label, return\_softmax=True)

acc = fluid.layers.accuracy(input=pred, label=node\_label, k=1)

loss = fluid.layers.mean(loss)

test\_program = train\_program.clone(for\_test=True)

**with** fluid.program\_guard(train\_program, startup\_program):

adam = fluid.optimizer.Adam(

learning\_rate=1e-2,

regularization=fluid.regularizer.L2DecayRegularizer(

regularization\_coeff=0.0005))

adam.minimize(loss)

开始训练

exe = fluid.Executor(place)

exe.run(startup\_program)

feed\_dict = gw.to\_feed(dataset.graph)

train\_index = dataset.train\_index

train\_label = np.expand\_dims(dataset.y[train\_index], -1)

train\_index = np.expand\_dims(train\_index, -1)

val\_index = dataset.val\_index

val\_label = np.expand\_dims(dataset.y[val\_index], -1)

val\_index = np.expand\_dims(val\_index, -1)

test\_index = dataset.test\_index

test\_label = np.expand\_dims(dataset.y[test\_index], -1)

test\_index = np.expand\_dims(test\_index, -1)

dur = []

**for** epoch **in** range(200):

**if** epoch >= 3:

t0 = time.time()

feed\_dict["node\_index"] = np.array(train\_index, dtype="int64")

feed\_dict["node\_label"] = np.array(train\_label, dtype="int64")

train\_loss, train\_acc = exe.run(train\_program,

feed=feed\_dict,

fetch\_list=[loss, acc],

return\_numpy=True)

**if** epoch >= 3:

time\_per\_epoch = 1.0 \* (time.time() - t0)

dur.append(time\_per\_epoch)

feed\_dict["node\_index"] = np.array(val\_index, dtype="int64")

feed\_dict["node\_label"] = np.array(val\_label, dtype="int64")

val\_loss, val\_acc = exe.run(test\_program,

feed=feed\_dict,

fetch\_list=[loss, acc],

return\_numpy=True)

log.info("Epoch %d " % epoch + "(%.5lf sec) " % np.mean(dur) +

"Train Loss: %f " % train\_loss + "Train Acc: %f " % train\_acc

+ "Val Loss: %f " % val\_loss + "Val Acc: %f " % val\_acc)

feed\_dict["node\_index"] = np.array(test\_index, dtype="int64")

feed\_dict["node\_label"] = np.array(test\_label, dtype="int64")

test\_loss, test\_acc = exe.run(test\_program,

feed=feed\_dict,

fetch\_list=[loss, acc],

return\_numpy=True)

log.info("Accuracy: %f" % test\_acc)