

《图神经网络导论》 实验报告

报告人 ： _____ 练炳诚 _____

实验指导教师： _____ 郑龙 _____

报告批阅教师： _____

计算机科学与技术学院

2020 年 12 月 6 日

华中科技大学课程实验报告

目 录

1 课程实验概述.....	1
1.1 实验目的.....	1
1.2 实验要求.....	1
1.3 实验任务.....	2
1.4 实验平台.....	2
2 实验内容实践.....	4
2.1 实验（1） 图神经网络环境的搭建.....	4
2.2 实验（2） DGL 编程入门.....	5
2.3 实验（3） GNN 实战.....	9
2.3.1 CORA 数据集介绍.....	9
2.3.2 实现 GCN.....	9
2.3.3 实现 GraphSAGE.....	12
2.3.4 实现 GAT.....	15
3 总结与心得.....	20
3.1 实验总结.....	20
3.2 实验心得.....	20

1 课程实验概述

1.1 实验目的

以深度神经网络形式表达的深度学习是一类机器学习算法，它用级联的多层非线性处理单元来进行特征的提取与转换，每个后续的层都使用前一层的输出作为输入。尽管传统的深度学习方法被应用在提取欧氏空间数据的特征方面取得了巨大的成功，但许多实际应用场景中的数据（例如，图）是从非欧式空间生成的，传统的深度学习方法在处理非欧式空间数据上的表现却仍难以使人满意，图神经网络应运而生。

图神经网络是基于深度学习的方法，在图结构数据上设计的神经网络模型，并且由于其令人信服的模型精度，广泛应用于很多领域。本实验基于单机平台构建单节点的图神经网络，基于 DGL 图神经网络编程库，从不同图数据集任务的角度了解图神经网络的数据处理、并行编程、模型训练、以及模型推理等，加深对典型图神经网络模型性能参数的理解，以达到如下目的：

1. 了解图神经网络编程和运行环境的搭建，熟悉典型编程框架的操作和使用，包括前端编程库的使用、后端编程接口的了解等；
2. 了解图神经网络的基本操作，包括图的创建、特征赋值、图神经网络模型定义、数据准备和初始化、以及模型训练与可视化；
3. 结合节点分类或图分类等现实需求，深入体会图神经网络的基本流程和具体步骤，了解解决实际问题的基本过程；
4. 针对若干典型图神经网络，对比分析不同图神经网络模型在训练精度和性能的差异，探讨潜在的改进措施。

1.2 实验要求

本实验以解决实际问题为导向，注重对图神经网络模型训练过程和推理效率的观

华中科技大学课程实验报告

察分析。实验评分以满分 100 分计算，完成步骤（1）和（2）的基本实验要求为 60 分，后续步骤中，每完成一个实验步骤增加 20 分。

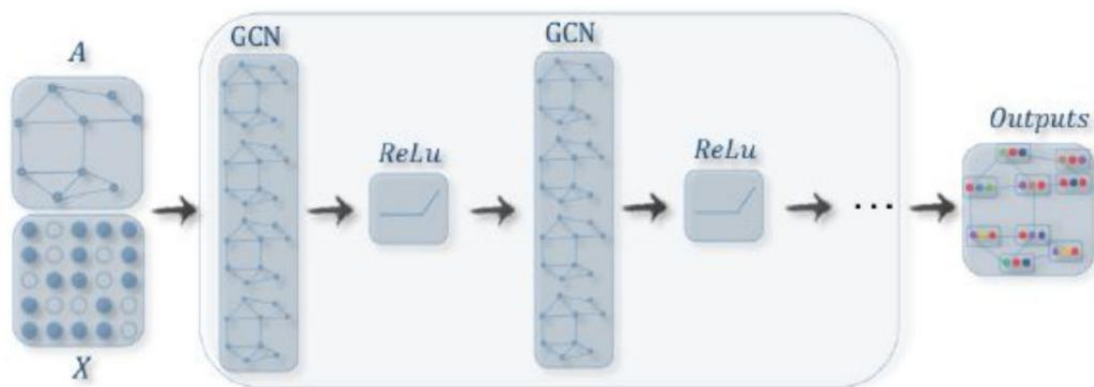


图 1-1 GCN 模型框架

1.3 实验任务

（1）图神经网络环境的搭建；

（2）DGL 编程入门：本实验主要实践 DGL 教程中的【扎卡里的空手道俱乐部问题】，了解 DGL 编程库的基本流程，熟悉 DGL 图神经网络的基本操作。；

（3）GNN 实战：利用 DGL 实现 GCN、GraphSAGE、GAT 三种图神经网络模型，基于 CORA 数据集实现复杂节点的分类；

（4）典型图神经网络的理解分析：

任务一：描述每个模型对应任务的验证精度与训练时间的关系；

任务二：探讨并回答以下问题：

1) 哪种模型执行效果最佳，解释相应原因？

2) 根据分析原因，修改模型，可否得到更好的结果？

1.4 实验平台

本实验使用的图神经网络编程库 Deep Graph Library（简称 DGL）。该框架组

华中科技大学课程实验报告

约大学和亚马逊公司主动开发，旨在为开发者提供一个基于现有深度学习张量计算框架开发 GNN 算法的统一计算平台。

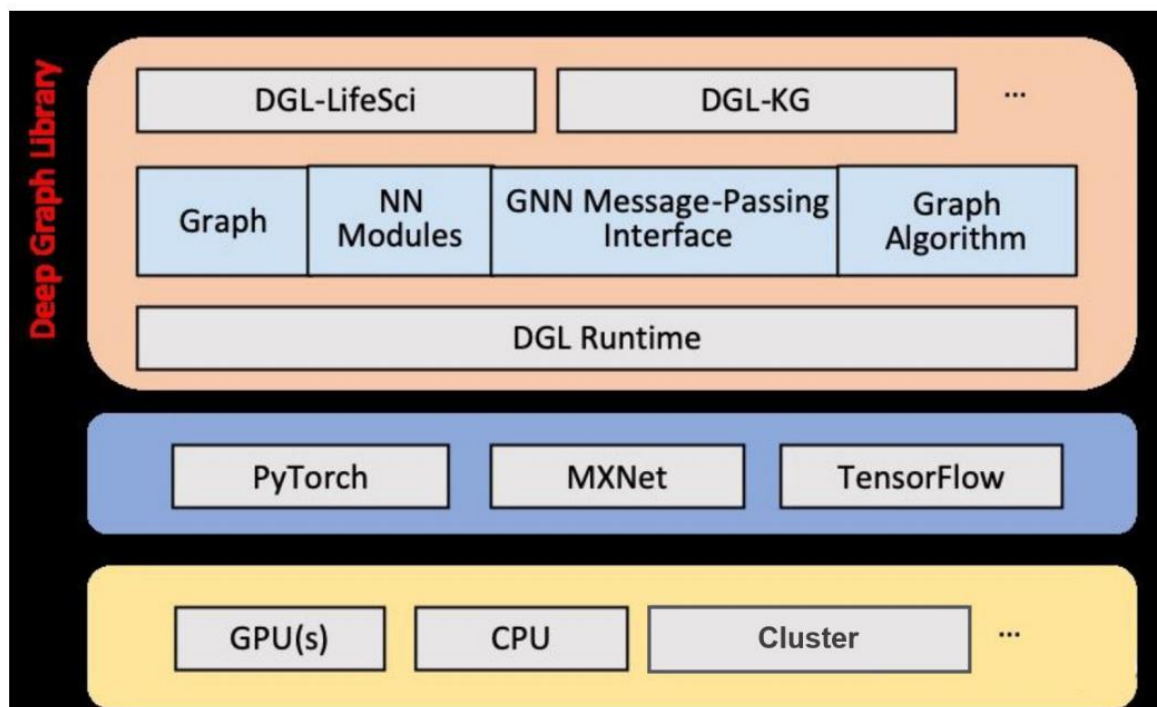


图 1-2 DGL 框架

图 1-2 展示了 DGL 的基础架构，DGL 最上层是应用层，包含生命科学和知识图谱两类，代码路径在 `/dgl/apps` 目录下；中间层有 DGL 定义的图、神经网络模块、图上的消息传递接口、图算法等；最下层是 DGL 的运行结构，与底层深度学习执行引擎对接（这部分对用户透明）。目前 DGL 支持的深度学习引擎有 PyTorch、MXnet、TensorFlow。

本次实验使用的 DGL 底层深度学习执行引擎是 PyTorch，DGL 系统环境是 Ubuntu20.04，python 版本是 3.8.6。

华中科技大学课程实验报告

2 实验内容实践

2.1 实验（1） 图神经网络环境的搭建

为方便，实验在 VMWare 虚拟机下进行，下载 Ubuntu20.10 镜像文件并安装系统，查看 python 版本为 3.8.6，如图 2-1 所示。

```
bingchlian@bingchlian-virtual-machine:~/桌面$ python3 --version
Python 3.8.6
```

图 2-1 python 版本

输入以下三条指令更新软件包并安装 pip3，查看 pip3 版本确认是基于 python3.8 的，如图 2-2 所示。

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install python3-pip

bingchlian@bingchlian-virtual-machine:~/桌面$ pip3 --version
pip 20.1.1 from /usr/lib/python3/dist-packages/pip (python 3.8)
```

图 2-2 pip3 版本

输入以下指令安装 pytorch 引擎、dgl，并配置 dgl 后端引擎为 pytorch

```
pip3 install torch==1.5.1+cpu torchvision==0.6.1+cpu torchaudio==0.5.1 -f http
s://download.pytorch.org/whl/torch_stable.html
pip3 install dgl
python3 -m dgl.backend.set_default_backend pytorch
```

运行 python3，尝试 import torch，结果成功，表明 pytorch 引擎安装成功，如图 2-3 所示。

```
bingchlian@bingchlian-virtual-machine:~/桌面$ python3
Python 3.8.6 (default, Sep 25 2020, 09:36:53)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>>
```

华中科技大学课程实验报告

图 2-3 pytorch 安装测试结果

至此环境搭配成功。

2.2 实验（2） DGL 编程入门

首先手动创建一个图结构，代码如下

```
import dgl
import numpy as np

# describe the graph with DGL
def build_karate_club_graph():
    # 所有 78 条边都存储在两个 numpy 数组中，一个用于源端点而另一个用于目标端点
    src = np.array([1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 7, 7, 7, 8, 8, 9, 10, 10,
10, 11, 12, 12, 13, 13, 13, 13, 16, 16, 17, 17, 19, 19, 21, 21,
25, 25, 27, 27, 27, 28, 29, 29, 30, 30, 31, 31, 31, 31, 32, 32,
32, 32, 32, 32, 32, 32, 32, 32, 33, 33, 33, 33, 33, 33, 33,
33, 33, 33, 33, 33, 33, 33, 33, 33])
    dst = np.array([0, 0, 1, 0, 1, 2, 0, 0, 0, 4, 5, 0, 1, 2, 3, 0, 2, 2, 0, 4,
5, 0, 0, 3, 0, 1, 2, 3, 5, 6, 0, 1, 0, 1, 0, 1, 23, 24, 2, 23,
24, 2, 23, 26, 1, 8, 0, 24, 25, 28, 2, 8, 14, 15, 18, 20, 22, 23,
29, 30, 31, 8, 9, 13, 14, 15, 18, 19, 20, 22, 23, 26, 27, 28, 29, 30,
31, 32])

    # 边在 DGL 中是有方向的；使它们双向
    u = np.concatenate([src, dst])
    v = np.concatenate([dst, src])

    # 构建图
    return dgl.DGLGraph((u, v))

G = build_karate_club_graph()

print('We have %d nodes.' % G.number_of_nodes())
```

华中科技大学课程实验报告

```
print('We have %d edges.' % G . number_of_nodes())
```

使用 `networks` 对图进行可视化，代码如下，建立的图结构如图 2-4 所示。

```
import networkx as nx
import matplotlib.pyplot as plt

# Since the actual graph is undirected, we convert it for visualization
# purpose.
nx_G = G.to_networkx().to_undirected()

# Kamada-Kawai layout usually looks pretty for arbitrary graphs
pos = nx.kamada_kawai_layout(nx_G)

nx.draw(nx_G, pos, with_labels=True, node_color=[[.7, .7, .7]])
plt.show()
```

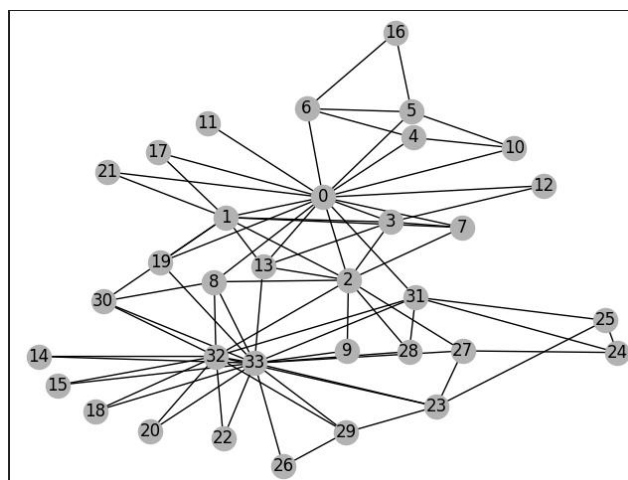


图 2-4 初始化图结构可视化

然后对节点特征赋值，使用 `torch.nn.Embedding` 为每个节点分配一个 5 维向量，代码如下

```
# assign features to nodes and edges
import torch
import torch.nn as nn
import torch.nn.functional as F

embed = nn.Embedding(34, 5) # 34 nodes with embedding dim equal to 5
```


华中科技大学课程实验报告

```
G.ndata['feat'] = embed.weight
```

接着定义一个 GCN 模型,这里使用了 dgl 包中提供的一层 GCN 实现 GraphConv,使用它定义了一个包含两层 GCN 的模型,代码如下

```
class GCN (nn.Module):  
    def __init__(self, in_feats, hidden_size, num_classes):  
        super(GCN, self).__init__()  
        self.conv1 = GraphConv(in_feats, hidden_size)  
        self.conv2 = GraphConv(hidden_size, num_classes)  
    def forward(self, g, inputs):  
        h = self.conv1(g, inputs)  
        h = torch.relu(h)  
        h = self.conv2(g, h)  
        return h
```

其中, forward()函数执行两层 GCN 之间的消息传递和计算工作,在 GCN 中,将上一层的输入聚合后进行输出即可,其中用到的激活函数为 ReLU,即聚合操作。

然后实例化网络,将 5 维特征转换到隐藏层也是 5 维,输出大小为 2 的特征表示两种俱乐部

```
net = GCN(5, 5, 2)
```

问题是将除了 0 和 33 号的节点进行分类,即除了 0 号和 33 号节点有标签外,其余节点均无标签,这是一种半监督学习的设置,故需先为 0 号节点和 33 号节点分配标签

```
inputs = embed.weight  
labeled_nodes = torch.tensor([0,33])  
labels = torch.tensor([0,1])
```

最后就可以训练模型了,首先创建一个优化器,然后将输入输入模型,接着计算损失,然后使用自动分级优化模型,代码如下

```
# train and visualization
```

华中科技大学课程实验报告

```
import itertools

optimizer = torch.optim.Adam(itertools.chain(net.parameters(), embed.parameters()),
                               lr=0.01)

all_logits = []

for epoch in range(50):

    logits = net(G, inputs)

    # 保存 logits 主要为了后续进行可视化实现
    all_logits.append(logits.detach())

    logp = F.log_softmax(logits, 1)

    # 只计算标签节点的损失
    loss = F.nll_loss(logp[labeled_nodes], labels)

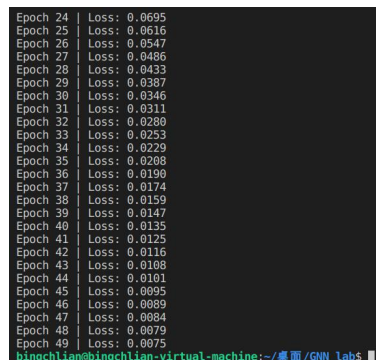
    optimizer.zero_grad()

    loss.backward()

    optimizer.step()

    print('Epoch %d | Loss: %.4f' % (epoch, loss.item()))
```

代码运行结果如图 2-5，可以看到在经过 50 次迭代后，已经达到一个很好的收敛效果。



Epoch	Loss
24	0.0695
25	0.0616
26	0.0547
27	0.0486
28	0.0433
29	0.0387
30	0.0346
31	0.0311
32	0.0280
33	0.0253
34	0.0229
35	0.0208
36	0.0190
37	0.0174
38	0.0159
39	0.0147
40	0.0135
41	0.0125
42	0.0116
43	0.0108
44	0.0101
45	0.0095
46	0.0089
47	0.0084
48	0.0079
49	0.0075

图 2-5 代码运行结果

华中科技大学课程实验报告

2.3 实验（3） GNN 实战

2.3.1 CORA 数据集介绍

共 2708 个样本点，每个样本点都是一篇科学论文，所有样本点被分为 7 个类别，类别分别是 1) 基于案例；2) 遗传算法；3) 神经网络；4) 概率方法；5) 强化学习；6) 规则学习；7) 理论。每篇论文都由一个 1433 维的词向量表示，所以，每个样本点具有 1433 个特征。词向量的每个元素都对应一个词，且该元素只有 0 或 1 两个取值。取 0 表示该元素对应的词不在论文中，取 1 表示在论文中。所有的词来源于一个具有 1433 个词的字典。

每篇论文都至少引用了一篇其他论文，或者被其他论文引用，也就是样本点之间存在联系，没有任何一个样本点与其他样本点完全没联系。如果将样本点看做图中的点，则这是一个连通的图，不存在孤立点。

2.3.2 实现 GCN

这一部分主要参考了文档中的实现，实现步骤跟实验（2）大致相同，不过没使用 dgl 中的 GraphConv，而是自己定义了网络的消息传递机制和节点的操作行为，再利用上述定义的机制操作定义 GCN 的 Embedding 更新层，最后就可定义一个包含两层 GCN 层的图神经网络分类器。

对与消息传递机制，每个节点发送 Embedding 时，直接复制即可，接受 Embedding 时，采用相加的方式进行聚合，代码如下

```
# message passing
gcn_msg = fn.copy_src(src='h', out='m')
gcn_reduce = fn.sum(msg='m', out='h')
```

接着定义节点的操作行为和 GCN 的 Embedding 更新层，即 GCNLayer 模块，本质上是在所有节点上执行消息传递并应用于全连接层。在节点上接收上层节点的输

华中科技大学课程实验报告

出，调用 `torch.nn.linear` 经过线性变换后再经过非线性函数进行计算，代码如下

```
# node UDF
class GCNLayer(nn.Module):
    def __init__(self, in_feats, out_feats):
        super(GCNLayer, self).__init__()
        self.linear = nn.Linear(in_feats, out_feats)

    def forward(self, g, feature):
        with g.local_scope():
            g.ndata['h'] = feature
            g.update_all(gcn_msg, gcn_reduce)
            h = g.ndata['h']
            return self.linear(h)
```

接着定义包含两层 GCN 层的分类器，输入特征维数为 1433 维，共 7 个类别的输出，第一层的激活函数使用默认的 ReLU，隐藏层输出大小为 16，代码如下

```
# net
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = GCNLayer(1433, 16)
        self.layer2 = GCNLayer(16, 7)

    def forward(self, g, features):
        x = F.relu(self.layer1(g, features))
        x = self.layer2(g, x)
        return x

net = Net()
```

然后就是加载 CORA 数据集进行预处理，本过程可适用于三种模型的数据处理，从数据集中提取出特征向量、标签，并划分训练集，创建 `DGLGraph`，代码如下

华中科技大学课程实验报告

```
# load data
from dgl.data import citation_graph as citegrh
import networkx as nx
def load_cora_data():
    data = citegrh.load_cora()
    features = th.FloatTensor(data.features)
    labels = th.LongTensor(data.labels)
    train_mask = th.BoolTensor(data.train_mask)
    g = DGLGraph(data.graph)
    return g, features, labels, train_mask
```

最后就是训练模型了，选择合适的迭代次数，计算 loss 和本次迭代的时间花费，代码如下

```
import time
import numpy as np
g, features, labels, train_mask = load_cora_data()
# Add edges between each node and itself to preserve old node representations
g.add_edges(g.nodes(), g.nodes())
optimizer = th.optim.Adam(net.parameters(), lr=1e-3)
dur = []
for epoch in range(50):
    if epoch >= 3:
        t0 = time.time()

        net.train()
        logits = net(g, features)
        logp = F.log_softmax(logits, 1)
        loss = F.nll_loss(logp[train_mask], labels[train_mask])

        optimizer.zero_grad()
```

华中科技大学课程实验报告

```
loss.backward()

optimizer.step()

if epoch >= 3:
    dur.append(time.time() - t0)

print("Epoch {:05d} | Loss {:.4f} | Time(s) {:.4f}".format( epoch, loss.item(),
np.mean(dur)))
```

选择迭代 50 次，运行结果如图 2-6 所示。

Epoch 00038	Loss 1.2905	Time(s) 0.0353
Epoch 00039	Loss 1.2753	Time(s) 0.0356
Epoch 00040	Loss 1.2603	Time(s) 0.0355
Epoch 00041	Loss 1.2456	Time(s) 0.0354
Epoch 00042	Loss 1.2312	Time(s) 0.0355
Epoch 00043	Loss 1.2170	Time(s) 0.0355
Epoch 00044	Loss 1.2030	Time(s) 0.0356
Epoch 00045	Loss 1.1893	Time(s) 0.0356
Epoch 00046	Loss 1.1758	Time(s) 0.0357
Epoch 00047	Loss 1.1624	Time(s) 0.0362
Epoch 00048	Loss 1.1491	Time(s) 0.0368
Epoch 00049	Loss 1.1360	Time(s) 0.0371

bingchlian@bingchlian-virtual-machine:~/桌面/GNN lab\$

图 2-6 GCN 运行结果

在经过 50 次迭代以后，Loss 降低到 1.136，每次迭代耗时在 0.035~0.037 之间。

2.3.3 实现 GraphSAGE

GraphSAGE(Graph sample and aggregate)网络相较 GCN 网络可以泛化到训练过程中没有出现过的节点，是一种归纳式的学习框架。GraphSAGE 主要流程分为以下三个步骤：

- (1) 对图中每个节点信息进行采样；
- (2) 使用聚合函数聚集邻居节点的蕴含信息；
- (3) 得到图中各节点信息的向量表示供下游任务使用。

其运行流程如图 2-7 所示。

华中科技大学课程实验报告

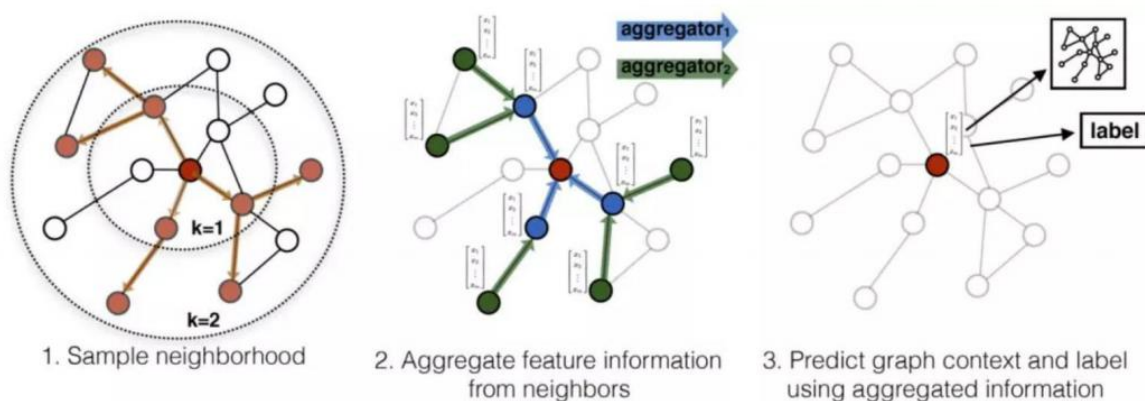


图 2-7 GraphSAGE 运行流程

实现 GraphSAGE 的关键在于定义神经网络的 Embedding 更新层和网络结构，数据处理和模型训练的部分与 GCN 相同。为快速实现，这里使用了 dgl 包的 SAGEConv 快速定义网络层，实现了一个包含输入层、隐藏层和输出层的图神经网络，代码如下

```
class GraphSAGE(nn.Module):  
    def __init__(self,  
                 in_feats,  
                 n_hidden,  
                 n_classes,  
                 n_layers,  
                 activation,  
                 dropout,  
                 aggregator_type):  
        super(GraphSAGE, self).__init__()  
        self.layers = nn.ModuleList()  
        self.dropout = nn.Dropout(dropout)  
        self.activation = activation  
  
        # input layer  
        self.layers.append(SAGEConv(in_feats, n_hidden, aggregator_type))
```

华中科技大学课程实验报告

```
# hidden layers

for i in range(n_layers - 1):
    self.layers.append(SAGEConv(n_hidden, n_hidden, aggregator_type))

# output layer
self.layers.append(SAGEConv(n_hidden, n_classes, aggregator_type))

def forward(self, graph, inputs):
    h = self.dropout(inputs)
    for l, layer in enumerate(self.layers):
        h = layer(graph, h)
        if l != len(self.layers) - 1:
            h = self.activation(h)
            h = self.dropout(h)
    return h
```

其中，使用了 `torch.nn.ModuleList()` 来维护 Layer，参数 `n_layers` 表示总的层数，中间的隐藏层数量为 `n_layers - 1`。此外，除了输出层外，其余层设置了 `torch.nn.Dropout()` 来防止过拟合，参与的神经元数量由 `n_dropout` 决定，默认值是 0.5 表示每次迭代有 50% 的神经元失活。其余的定义与 GCN 类似，此处不再叙述。

采用与 GCN 相同的数据处理函数和训练方式，设定隐藏层数量为 1，选取迭代 50 次，运行结果如图 2-8 所示，经过 50 次迭代后 loss 降低到 1.8758，每次迭代耗时在 0.074~0.076 之间。

Epoch 00038	Loss 1.8947	Time(s) 0.0729
Epoch 00039	Loss 1.9011	Time(s) 0.0731
Epoch 00040	Loss 1.8939	Time(s) 0.0742
Epoch 00041	Loss 1.8940	Time(s) 0.0747
Epoch 00042	Loss 1.8872	Time(s) 0.0747
Epoch 00043	Loss 1.8843	Time(s) 0.0746
Epoch 00044	Loss 1.8877	Time(s) 0.0748
Epoch 00045	Loss 1.8863	Time(s) 0.0750
Epoch 00046	Loss 1.8830	Time(s) 0.0752
Epoch 00047	Loss 1.8812	Time(s) 0.0756
Epoch 00048	Loss 1.8758	Time(s) 0.0760
Epoch 00049	Loss 1.8758	Time(s) 0.0762

bingchlian@bingchlian-virtual-machine:~/桌面/GNN Lab\$

图 2-8 GraphSAGE 运行结果

华中科技大学课程实验报告

2.3.4 实现 GAT

GAT 是一种注意力网络，它具有与 GCN 相似的结构，但是它在收集邻居顶点的特征向量的时候，对每一条边通过计算 Attention 对信息进行加权。与 GCN 平等对待节点的所有邻居相比，注意力机制可以为每个邻居分配不同的注意力得分，从而识别出更重要的邻居。

GAT 的实现参考了 dgl 技术文档中的介绍 [click](#)，GAT 的原理如图 2-9 所示。

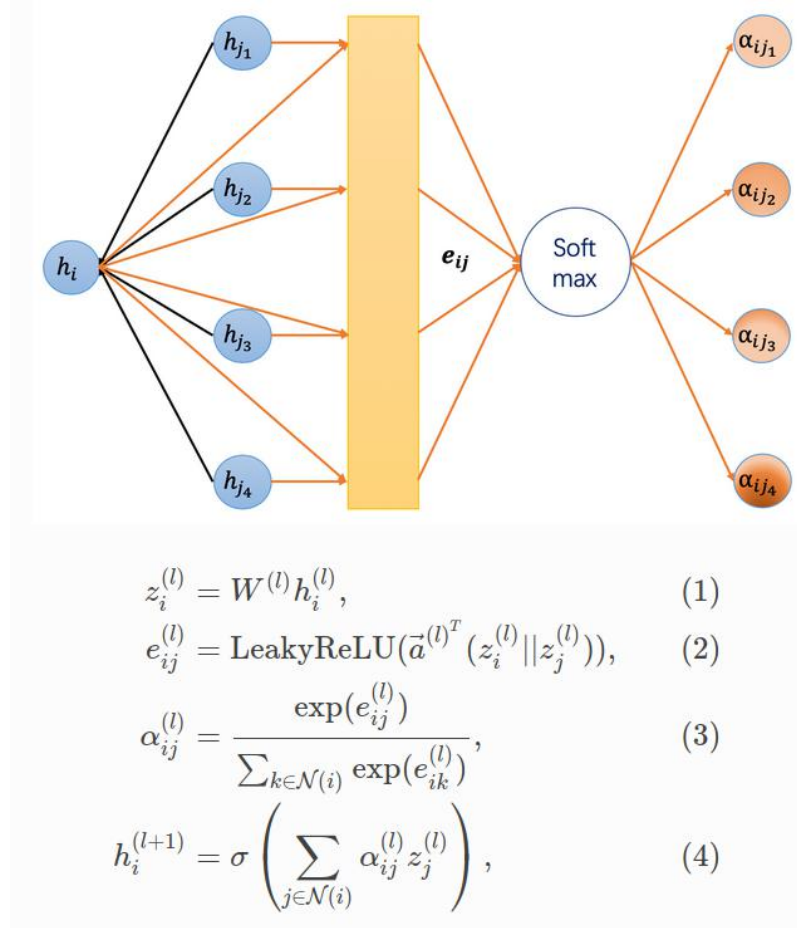


图 2-9 GAT 原理

其中，式（1）将下层特征向量 $h_i^{(l)}$ 进行线性转换， $W^{(l)}$ 为学习权重矩阵；式（2）计算两个邻居之间的一个成对的未归一化的 attention score，首先将两个邻居的经式（1）计算得到的向量进行拼接，并与一个可学习的权值向量 $\vec{a}^{(l)T}$ 进行点积运算，然

华中科技大学课程实验报告

后应用 LeakyReLU 激活函数得到未归一化的 attention score；式（3）应用 softmax 将每个顶点传入的 attention score 进行归一化；式（4）与 GCN 类似，将来自邻居的信息进行聚合，并根据 attention score 进行缩放。

与实现 GraphSAGE 类似，首先是要用 dgl 实现一个 GATLayer，即对图 2-9 中的 4 个式子进行表示，代码如下

```
from dgl.nn.pytorch import GATConv

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

class GATLayer(nn.Module):
    def __init__(self, g, in_dim, out_dim):
        super(GATLayer, self).__init__()

        self.g = g

        # equation (1)
        self.fc = nn.Linear(in_dim, out_dim, bias=False)

        # equation (2)
        self.attn_fc = nn.Linear(2 * out_dim, 1, bias=False)

        self.reset_parameters()

    def reset_parameters(self):
        gain = nn.init.calculate_gain('relu')
        nn.init.xavier_normal_(self.fc.weight, gain=gain)
        nn.init.xavier_normal_(self.attn_fc.weight, gain=gain)

    def edge_attention(self, edges):
        # edge UDF for equation (2)
        z2 = torch.cat([edges.src['z'], edges.dst['z']], dim=1)
```

华中科技大学课程实验报告

```
a = self.attn_fc(z2)

return {'e': F.leaky_relu(a)}

def message_func(self, edges):

    # message UDF for equation (3) & (4)

    return {'z': edges.src['z'], 'e': edges.data['e']}

def reduce_func(self, nodes):

    # reduce UDF for equation (3) & (4)

    # equation (3)

    alpha = F.softmax(nodes.mailbox['e'], dim=1)

    # equation (4)

    h = torch.sum(alpha * nodes.mailbox['z'], dim=1)

    return {'h': h}

def forward(self, h):

    # equation (1)

    z = self.fc(h)

    self.g.ndata['z'] = z

    # equation (2)

    self.g.apply_edges(self.edge_attention)

    # equation (3) & (4)

    self.g.update_all(self.message_func, self.reduce_func)

    return self.g.ndata.pop('h')

def edge_attention(self, edges):

    # edge UDF for equation (2)

    z2 = torch.cat([edges.src['z'], edges.dst['z']], dim=1)

    a = self.attn_fc(z2)

    return {'e' : F.leaky_relu(a)}
```

华中科技大学课程实验报告

```
def reduce_func(self, nodes):  
    # reduce UDF for equation (3) & (4)  
    # equation (3)  
    alpha = F.softmax(nodes.mailbox['e'], dim=1)  
    # equation (4)  
    h = torch.sum(alpha * nodes.mailbox['z'], dim=1)  
    return {'h' : h}
```

其中，式（1）采用 `torch.nn.Linear` 函数实现即可，式（2）中的特征向量是来自于邻居节点，因此可以把它看作边的数据，调用 `apply_edges` 这个 API，参数是一个 Edge UDF，即下面的 `edge_attention` 函数，其中调用 `torch.cat` 将向量进行拼接调用 `attn_fc` 实现点积计算，并按标记'e'进行传递，式（3）将上述标记为'e'的未归一化的向量调用 `softmax` 进行归一化，式（4）将该归一化的向量与权重矩阵进行乘积。

GAT 使用了 `muti-head attention` 来丰富模型的容量和稳定学习过程。依靠实现的 `GATLayer` 可实现 `MutiHeadGATLayer`，实现思路与 `GraphSAGE` 类似，使用 `ModuleList` 来维护一个 head 列表，每个 head 是一个 `GATLayer` 实例，代码如下

```
class MultiHeadGATLayer(nn.Module):  
    def __init__(self, g, in_dim, out_dim, num_heads, merge='cat'):  
        super(MultiHeadGATLayer, self).__init__()  
        self.heads = nn.ModuleList()  
        for i in range(num_heads):  
            self.heads.append(GATLayer(g, in_dim, out_dim))  
        self.merge = merge  
  
    def forward(self, h):  
        head_outs = [attn_head(h) for attn_head in self.heads]  
        if self.merge == 'cat':  
            # concat on the output feature dimension (dim=1)
```

华中科技大学课程实验报告

```
        return torch.cat(head_outs, dim=1)

    else:

        # merge using average

        return torch.mean(torch.stack(head_outs))
```

然后就可以定义包含两层网络的 GAT 模型了，注意隐藏层输入为 head 数，输出为隐藏层数*head 数

```
class GAT(nn.Module):

    def __init__(self, g, in_dim, hidden_dim, out_dim, num_heads):

        super(GAT, self).__init__()

        self.layer1 = MultiHeadGATLayer(g, in_dim, hidden_dim, num_heads)

        # Be aware that the input dimension is hidden_dim*num_heads since

        # multiple head outputs are concatenated together. Also, only

        # one attention head in the output layer.

        self.layer2 = MultiHeadGATLayer(g, hidden_dim * num_heads, out_dim, 1)

    def forward(self, h):

        h = self.layer1(h)

        h = F.elu(h)

        h = self.layer2(h)

        return h
```

数据处理和模型训练的代码与前两个模型相同，设定合适的迭代次数，经过 50 次迭代后，结果如图 2-10 所示，loss 降低到 1.77，每次迭代平均 0.18s。

```
Epoch 00041 | Loss 1.8030 | Time(s) 0.1816
Epoch 00042 | Loss 1.7990 | Time(s) 0.1809
Epoch 00043 | Loss 1.7949 | Time(s) 0.1810
Epoch 00044 | Loss 1.7908 | Time(s) 0.1806
Epoch 00045 | Loss 1.7867 | Time(s) 0.1804
Epoch 00046 | Loss 1.7826 | Time(s) 0.1801
Epoch 00047 | Loss 1.7784 | Time(s) 0.1798
Epoch 00048 | Loss 1.7742 | Time(s) 0.1795
Epoch 00049 | Loss 1.7700 | Time(s) 0.1792
bingchlian@bingchlian-virtual-machine:~/桌面/GNN lab$
```

图 2-10 GAT 运行结果

3 总结与心得

3.1 实验总结

在本课程的实验过程中，作了如下几点工作：

(1) 在 Linux 环境下，搭建了实验环境，学习了使用 pyTorch 为后端引擎的 GDL 的编程。

(2) 学习使用了 DGL 实现了简单的 GCN 网络，以半监督学习的形式解决了二分类问题并对结果进行了可视化。

(3) 参考了实验指导书、DGL 技术文档和数篇博客，用 DGL 实现了 GCN、GraphSAGE、GAT 网络，并用 CORA 数据集进行训练，得到 loss 值。

3.2 实验心得

(1) 了解了当今主流的图神经网络框架 DGL 的基本架构，以及深度学习执行引擎 pyTorch。

(2) 学习了三种经典图神经网络的模型 GCN、GraphSAGE 和 GAT，了解到它们之间的性能和特点差异，初步结论是 GCN 的性能在 CORA 数据集上更好。

(3) 通过实验过程对图神经网络的概念有了一些了解，为以后更深入学习相关知识的敲门砖。