模型教程

目录

1. 总览

1.1 神经网络和它的变体

- **Graph convolutional network (GCN)** [research paper] [tutorial] [Pytorch code] [MXNet code]: 这是最基本的 GCN。 本教程介绍了 DGL API 的基本用法。
- **Graph attention network (GAT)** [research paper] [tutorial] [Pytorch code] [MXNet code]: GAT 通过在节点附近部署多头注意力来扩展 GCN 功能。 这大大提高了模型的容量和表达能力。
- **Relational-GCN** [research paper] [tutorial] [Pytorch code] [MXNet code]: Relational-GCN 允许图的两个实体之间有多个边。 具有不同关系的边的编码方式不同。
- **Line graph neural network (LGNN)** [research paper] [tutorial] [Pytorch code]: 该网络通过检测图的结构来实现社区检测。它同时使用原始图和它的线形图同伴的表示。除了演示一个算法如何利用多个图之外，这个实现还展示了如何明智地将简单张量操作和稀疏矩阵张量操作以及 DGL 中的消息传递混合在一起。
- **Stochastic steady-state embedding (SSE)** [research paper] [tutorial] [MXNet code]: SSE 是一个示例，用于说明算法和系统的协同设计。 采样可确保渐进收敛，同时降低复杂度并跨样本进行批处理，以实现最大的并行度。 这里的重点是一张巨大的图无法舒适地适合一张 GPU 卡。

1.2 批处理很多小图

- **Tree-LSTM** [paper] [tutorial] [PyTorch code]: 句子具有固有的结构，只需将它们简单地视为序列即可将其丢弃。 Tree-LSTM 是一个功能强大的模型，它通过使用诸如语法分析树之类的现有语法结构来学习表示形式。训练中的挑战在于，仅通过将句子填充到最大长度就不再起作用。 不同句子的树具有不同的大小和拓扑。 DGL 通过将树添加到更大的容器图中，然后使用消息传递来探索最大的并行性来解决此问题。 批处理是实现此目的的关键 API。

1.3 生成模型

- **DGMG** [paper] [tutorial] [PyTorch code]: 该模型属于处理结构生成的家族。 图的深度生成模型（DGMG）使用状态机方法。 这也是非常具有挑战性的，因为与 Tree-LSTM 不同，每个样本都具有动态的，由概率驱动的结构，该结构在训练之前无法获得。 您可以逐步利用图内和图间并

行性来稳定地提高性能。

- **JTNN** [paper] [PyTorch code]: 该网络使用变分自动编码器的框架生成分子图。 联结树神经网络（JTNN）分层构建结构。 对于分子图，它使用连接树作为中间支架。

1.4 从图角度重新审视经典模型

- **Capsule** [paper] [tutorial] [PyTorch code]: 这个新的计算机视觉模型具有两个关键思想。 首先，以称为胶囊的矢量形式（而不是标量）增强特征表示。 第二，用动态路由替换最大池化。 动态路由的想法是通过非参数消息传递将较低级别的容器集成到一个或几个较高级别的容器中。 教程显示了如何使用 DGL API 实现后者。

  Transformer [paper] [tutorial] [PyTorch code] and **Universal Transformer** [paper] [tutorial][PyTorch code]: 这两个模型用多层多头注意力代替了递归神经网络（RNN），以编码和发现句子标记之间的结构。 这些注意机制类似地表示为带有消息传递的图操作。

2. 图卷积网络

Author: Qi Huang, Minjie Wang, Yu Gai, Quan Gan, Zheng Zhang

这是使用 DGL 实现图卷积网络的简短介绍（Kipf & Welling 等人，Semi-Supervised Classification with Graph Convolutional Networks）。 我们将解释 GraphConv 模块的内容。 希望读者学习如何使用 DGL 的消息传递 API 定义新的 GNN 层。

我们以 DGLGraph 上较早的教程为基础，并演示 DGL 如何将图与深度神经网络相结合并学习结构表示。

2.1 模型总览

2.1.1 从消息传递的角度看 GCN

我们从消息传递的角度描述了图卷积神经网络的一层。对于每个节点 u，它可以归纳为以下步骤：

1）汇总邻居的表示 h_v 以产生中间表示$\hat{h}_u$。

2）用线性投影和随后的非线性变换聚合表示$\hat{h}_u$：$h_u = f(W\hat{h}_u)$。

我们将通过 DGL 消息传递实现第 1 步，并通过 PyTorch nn.Module 实现第 2 步。

2.1.2 用 DGL 实现 GCN

我们首先向往常一样定义消息归约函数。 由于节点 u 上的聚合仅涉及对邻居的表示 h_v 的求和，因此我们可以简单地使用内置函数：

```
import dgl
```

```
import dgl.function as fn
import torch as th
import torch.nn as nn
import torch.nn.functional as F
from dgl import DGLGraph

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

然后，我们继续定义 GCNLayer 模块。 GCNLayer 本质上在所有节点上执行消息传递，然后应用全连接层。

```
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):
        # Creating a local scope so that all the stored ndata and edata
        # (such as the `'h'` ndata below) are automatically popped out
        # when the scope exits.
        with g.local_scope():
            g.ndata['h'] = feature
            g.update_all(gcn_msg, gcn_reduce)
            h = g.ndata['h']
            return self.linear(h)
```

前向函数与 PyTorch 中任何其他常见的 NN 模型基本相同。 我们可以像任何 nn.Module 一样初始化 GCN。 例如，让我们定义一个由两个 GCN 层组成的简单神经网络。 假设我们正在训练 cora 数据集的分类器（输入要素大小为 1433，类别数为 7）。 最后的 GCN 层计算节点嵌入，因此最后一层通常不应用激活。

```
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()
print(net)
```

Out:

```
Net(
  (layer1): GCNLayer(
    (linear): Linear(in_features=1433, out_features=16, bias=True)
```

```
  )
  (layer2): GCNLayer(
    (linear): Linear(in_features=16, out_features=7, bias=True)
  )
)
```

我们使用 DGL 的内置数据模块加载 cora 数据集。

```
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)
    test_mask = th.BoolTensor(data.test_mask)
    g = DGLGraph(data.graph)
    return g, features, labels, train_mask, test_mask
```

训练模型后，我们可以使用以下方法评估模型在测试数据集上的性能：

```
def evaluate(model, g, features, labels, mask):
    model.eval()
    with th.no_grad():
        logits = model(g, features)
        logits = logits[mask]
        labels = labels[mask]
        _, indices = th.max(logits, dim=1)
        correct = th.sum(indices == labels)
        return correct.item() * 1.0 / len(labels)
```

然后，我们按如下方式训练网络：

```
import time
import numpy as np
g, features, labels, train_mask, test_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-2)
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()
```

```
    loss.backward()
    optimizer.step()

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

    acc = evaluate(net, g, features, labels, test_mask)
    print("Epoch {:05d} | Loss {:.4f} | Test Acc {:.4f} | Time(s) {:.4f}".format(
            epoch, loss.item(), acc, np.mean(dur)))
```

结果：

```
Finished data loading and preprocessing.
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done saving data into cached files.
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.feat will be deprecated, please use
g.ndata['feat'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.label will be deprecated, please use
g.ndata['label'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.train_mask will be deprecated, please
use g.ndata['train_mask'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.test_mask will be deprecated, please use
g.ndata['test_mask'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.graph will be deprecated, please use
dataset.g instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
```

```
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/base.py:45: DGLWarning: Recommend creating graphs by `dgl.graph(data)` instead of `dgl.DGLGraph(data)`.
  return warnings.warn(message, category=category, stacklevel=1)
/home/ubuntu/.pyenv/versions/miniconda3-latest/lib/python3.7/site-packages/numpy/core/fromnumeric.py:3257: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/home/ubuntu/.pyenv/versions/miniconda3-latest/lib/python3.7/site-packages/numpy/core/_methods.py:161: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
Epoch 00000 | Loss 1.9543 | Test Acc 0.2160 | Time(s) nan
Epoch 00001 | Loss 1.8303 | Test Acc 0.3620 | Time(s) nan
Epoch 00002 | Loss 1.6874 | Test Acc 0.5060 | Time(s) nan
Epoch 00003 | Loss 1.5428 | Test Acc 0.6280 | Time(s) 0.0271
Epoch 00004 | Loss 1.4042 | Test Acc 0.6980 | Time(s) 0.0269
Epoch 00005 | Loss 1.2841 | Test Acc 0.7120 | Time(s) 0.0268
Epoch 00006 | Loss 1.1710 | Test Acc 0.7050 | Time(s) 0.0267
Epoch 00007 | Loss 1.0704 | Test Acc 0.7010 | Time(s) 0.0254
Epoch 00008 | Loss 0.9797 | Test Acc 0.6920 | Time(s) 0.0256
Epoch 00009 | Loss 0.8972 | Test Acc 0.6960 | Time(s) 0.0255
Epoch 00010 | Loss 0.8208 | Test Acc 0.7080 | Time(s) 0.0249
Epoch 00011 | Loss 0.7496 | Test Acc 0.7110 | Time(s) 0.0249
Epoch 00012 | Loss 0.6837 | Test Acc 0.7240 | Time(s) 0.0245
Epoch 00013 | Loss 0.6234 | Test Acc 0.7310 | Time(s) 0.0248
Epoch 00014 | Loss 0.5684 | Test Acc 0.7380 | Time(s) 0.0248
Epoch 00015 | Loss 0.5182 | Test Acc 0.7400 | Time(s) 0.0251
Epoch 00016 | Loss 0.4721 | Test Acc 0.7430 | Time(s) 0.0252
Epoch 00017 | Loss 0.4301 | Test Acc 0.7410 | Time(s) 0.0251
Epoch 00018 | Loss 0.3921 | Test Acc 0.7360 | Time(s) 0.0250
Epoch 00019 | Loss 0.3577 | Test Acc 0.7310 | Time(s) 0.0247
Epoch 00020 | Loss 0.3265 | Test Acc 0.7300 | Time(s) 0.0248
Epoch 00021 | Loss 0.2981 | Test Acc 0.7390 | Time(s) 0.0248
Epoch 00022 | Loss 0.2724 | Test Acc 0.7430 | Time(s) 0.0250
Epoch 00023 | Loss 0.2491 | Test Acc 0.7430 | Time(s) 0.0247
Epoch 00024 | Loss 0.2279 | Test Acc 0.7440 | Time(s) 0.0245
Epoch 00025 | Loss 0.2085 | Test Acc 0.7520 | Time(s) 0.0246
Epoch 00026 | Loss 0.1908 | Test Acc 0.7520 | Time(s) 0.0247
Epoch 00027 | Loss 0.1745 | Test Acc 0.7530 | Time(s) 0.0245
Epoch 00028 | Loss 0.1595 | Test Acc 0.7550 | Time(s) 0.0245
Epoch 00029 | Loss 0.1458 | Test Acc 0.7550 | Time(s) 0.0246
Epoch 00030 | Loss 0.1333 | Test Acc 0.7550 | Time(s) 0.0246
Epoch 00031 | Loss 0.1218 | Test Acc 0.7560 | Time(s) 0.0244
Epoch 00032 | Loss 0.1114 | Test Acc 0.7530 | Time(s) 0.0246
Epoch 00033 | Loss 0.1019 | Test Acc 0.7520 | Time(s) 0.0246
```

```
Epoch 00034 | Loss 0.0933 | Test Acc 0.7480 | Time(s) 0.0247
Epoch 00035 | Loss 0.0855 | Test Acc 0.7450 | Time(s) 0.0247
Epoch 00036 | Loss 0.0785 | Test Acc 0.7450 | Time(s) 0.0246
Epoch 00037 | Loss 0.0720 | Test Acc 0.7450 | Time(s) 0.0246
Epoch 00038 | Loss 0.0662 | Test Acc 0.7460 | Time(s) 0.0246
Epoch 00039 | Loss 0.0609 | Test Acc 0.7460 | Time(s) 0.0246
Epoch 00040 | Loss 0.0561 | Test Acc 0.7420 | Time(s) 0.0247
Epoch 00041 | Loss 0.0518 | Test Acc 0.7420 | Time(s) 0.0247
Epoch 00042 | Loss 0.0478 | Test Acc 0.7400 | Time(s) 0.0247
Epoch 00043 | Loss 0.0443 | Test Acc 0.7390 | Time(s) 0.0247
Epoch 00044 | Loss 0.0410 | Test Acc 0.7390 | Time(s) 0.0248
Epoch 00045 | Loss 0.0380 | Test Acc 0.7380 | Time(s) 0.0248
Epoch 00046 | Loss 0.0353 | Test Acc 0.7390 | Time(s) 0.0249
Epoch 00047 | Loss 0.0329 | Test Acc 0.7360 | Time(s) 0.0249
Epoch 00048 | Loss 0.0307 | Test Acc 0.7350 | Time(s) 0.0248
Epoch 00049 | Loss 0.0287 | Test Acc 0.7350 | Time(s) 0.0248
```

2.1.3 GCN 中的公式

在数学上，GCN 模型遵循以下公式：

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)})$$

此处，$H^{(l)}$表示网络中的第 l^{th} 层，σ是非线性，W 是该层的权重矩阵。$\tilde{D}$和$\tilde{A}$分别是图的度和邻接矩阵。对于上标～，我们指的是在每个节点与其自身之间添加其他边以保留其在图卷积中的旧表示形式的变体。输入 $H^{(0)}$的形状为 N×D，其中 N 是节点数，D 是输入特征的数量。我们可以将多层链接起来，以生成具有 N×F 形状的节点级表示输出，其中 F 是输出节点特征向量的维度。

可以使用稀疏矩阵乘法内核（例如 Kipf 的 pygcn 代码）有效地实现该方程式。实际上，由于使用内置函数，上述 DGL 实现实际上已经使用了该技巧。要了解幕后花絮，请阅读我们在 PageRank 上的教程。

请注意，本教程的代码实现了 GCN 的简化版本，其中我们用$\tilde{A}$替换了$\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$。有关完整的实现，请参见此处的示例:
https://github.com/dmlc/dgl/tree/master/examples/pytorch/gcn

2.2 关系图卷积网络

Author: Lingfan Yu, Mufei Li, Zheng Zhang

在本教程中，您将学习如何实现关系图卷积网络(R-GCN)。这种类型的网络是对 GCN 进行一般化以处理知识库中实体之间的不同关系的一种努力。要了解更多关于 R-GCN 背后的研究，请参见使用 Modeling Relational Data with Graph Convolutional Networks

图卷积网络(GCN)和 DGL 教程直接利用数据集的结构信息(即图的连通性)来改进节点表示的提取。图的边保留为无类型的。

知识图由主体、关系、对象的三元组集合构成。边因此编码了重要的信息，并有自己的嵌入需要学习。此外，任意给定的对之间可能存在多条边。

2.2.1 R-GCN 简介

在统计关系学习（SRL）中，有两个基本任务：

- 实体分类-在其中为实体分配类型和分类属性。
- 链接预测-在其中恢复丢失的三元组。

在这两种情况下，期望从图的邻域结构中恢复丢失的信息。 例如，前面引用的 R-GCN 论文提供了以下示例。 知道米哈伊尔·巴里什尼科夫在瓦加诺瓦学院受过教育，就意味着米哈伊尔·巴里什尼科夫应该有个人标签，也就是三元组（米哈伊尔·巴里什尼科夫，居住，俄罗斯）必须属于知识图谱。

R-GCN 使用通用图卷积网络解决了这两个问题。 它对多边编码进行了扩展，以计算实体的嵌入，但具有不同的下游处理。

- 实体分类是通过最后一个实体(节点) 嵌入后附加一个 softmax 分类器。训练是通过标准交叉熵损失进行的。
- 链接预测是通过使用参数化评分函数，利用 autoencoder 架构重建边来完成的。训练使用负抽样。

本教程重点介绍第一个任务，即实体分类，以展示如何生成实体表示。 在 DGL Github 存储库中可以找到这两项任务的完整代码。

https://github.com/dmlc/dgl/tree/rgcn/examples/pytorch/rgcn

2.2.2 R-GCN 的关键思想

回想一下，在 GCN 中，第(l+1)层中每个节点 i 的隐藏表示由以下公式计算：

$$h_i^{l+1} = \sigma\left(\sum_{j \in N_i} \frac{1}{c_i} W^{(l)} h_j^{(l)}\right) \qquad (1)$$

其中 c_i 是归一化常数。

R-GCN 和 GCN 之间的关键区别在于，在 R-GCN 中，边可以表示不同的关系。在 GCN 中，等式（1）中的权重 $W^{(l)}$由层 l 中的所有边共享。 相反，在 R-GCN 中，不同的边类型使用不同的权重，并且只有相同关系类型 r 的边与相同的投影权重 $W_r^{(l)}$相关联。

因此，R-GCN 中第(l+1)中实体的隐藏表示可以表示为以下方程式：

$$h_i^{l+1} = \sigma\left(W_0^{(l)} h_i^{(l)} + \sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)}\right) \qquad (2)$$

其中N_i^r表示关系 r∈R 下节点 i 的邻居索引集，而 $c_{i,r}$ 是归一化常数。 在实体分类中，R-GCN 论文使用 $c_{i,r}=|N_i^r|$。

直接应用上述方程的问题是参数数量的快速增长，特别是在高度多关系数据的情况下。为了减小模型参数的大小，防止模型过拟合，该文提出使用基分解。

$$W_r^{(l)} = \sum_{b=1}^{B} a_{rb}^{(l)} V_b^{(l)} \qquad (3)$$

因此，权重$W_r^{(l)}$是基础变换$V_b^{(l)}$与系数$a_{rb}^{(l)}$的线性组合。 基数 B 的数量比知识库中的关系数小得多。

注意：在链接预测中实现了另一个权重正则化，即块分解。

2.2.3 在 DGL 中实现 R-GCN

R-GCN 模型由几个 R-GCN 层组成。 第一 R-GCN 层还用作输入层，并接受与节点实体相关联并投影到隐藏空间的特征（例如，描述文本）。 在本教程中，我们仅将实体 ID 用作实体特征。

2.2.3.1 R-GCN 层

对于每个节点，R-GCN 层执行以下步骤:

- 使用与边类型相关的节点表示和权值矩阵计算传出消息(消息函数)
- 聚合传入消息并生成新的节点表示(reduce 和 apply 函数)

以下代码是 R-GCN 隐藏层的定义。

注意：每种关系类型都具有不同的权重。 因此，权重矩阵具有三个维度：relation，input_feature，output_feature。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from dgl import DGLGraph
import dgl.function as fn
from functools import partial

class RGCNLayer(nn.Module):
    def __init__(self, in_feat, out_feat, num_rels, num_bases=-1, bias=None,
                 activation=None, is_input_layer=False):
        super(RGCNLayer, self).__init__()
        self.in_feat = in_feat
        self.out_feat = out_feat
        self.num_rels = num_rels
        self.num_bases = num_bases
        self.bias = bias
        self.activation = activation
        self.is_input_layer = is_input_layer

        # sanity check
        if self.num_bases <= 0 or self.num_bases > self.num_rels:
            self.num_bases = self.num_rels
```

```
        # weight bases in equation (3)
        self.weight = nn.Parameter(torch.Tensor(self.num_bases, self.in_feat,
                                                self.out_feat))
        if self.num_bases < self.num_rels:
            # linear combination coefficients in equation (3)
            self.w_comp = nn.Parameter(torch.Tensor(self.num_rels, self.num_bases))

        # add bias
        if self.bias:
            self.bias = nn.Parameter(torch.Tensor(out_feat))

        # init trainable parameters
        nn.init.xavier_uniform_(self.weight,
                                gain=nn.init.calculate_gain('relu'))
        if self.num_bases < self.num_rels:
            nn.init.xavier_uniform_(self.w_comp,
                                    gain=nn.init.calculate_gain('relu'))
        if self.bias:
            nn.init.xavier_uniform_(self.bias,
                                    gain=nn.init.calculate_gain('relu'))

    def forward(self, g):
        if self.num_bases < self.num_rels:
            # generate all weights from bases (equation (3))
            weight = self.weight.view(self.in_feat, self.num_bases, self.out_feat)
            weight = torch.matmul(self.w_comp, weight).view(self.num_rels,
                                                            self.in_feat, self.out_feat)
        else:
            weight = self.weight

        if self.is_input_layer:
            def message_func(edges):
                # for input layer, matrix multiply can be converted to be
                # an embedding lookup using source node id
                embed = weight.view(-1, self.out_feat)
                index = edges.data['rel_type'] * self.in_feat + edges.src['id']
                return {'msg': embed[index] * edges.data['norm']}
        else:
            def message_func(edges):
                w = weight[edges.data['rel_type']]
                msg = torch.bmm(edges.src['h'].unsqueeze(1), w).squeeze()
                msg = msg * edges.data['norm']
                return {'msg': msg}
```

```
        def apply_func(nodes):
            h = nodes.data['h']
            if self.bias:
                h = h + self.bias
            if self.activation:
                h = self.activation(h)
            return {'h': h}

        g.update_all(message_func, fn.sum(msg='msg', out='h'), apply_func)
```

2.2.3.2 完整的 R-GCN 模型定义

```
class Model(nn.Module):
    def __init__(self, num_nodes, h_dim, out_dim, num_rels,
                 num_bases=-1, num_hidden_layers=1):
        super(Model, self).__init__()
        self.num_nodes = num_nodes
        self.h_dim = h_dim
        self.out_dim = out_dim
        self.num_rels = num_rels
        self.num_bases = num_bases
        self.num_hidden_layers = num_hidden_layers

        # create rgcn layers
        self.build_model()

        # create initial features
        self.features = self.create_features()

    def build_model(self):
        self.layers = nn.ModuleList()
        # input to hidden
        i2h = self.build_input_layer()
        self.layers.append(i2h)
        # hidden to hidden
        for _ in range(self.num_hidden_layers):
            h2h = self.build_hidden_layer()
            self.layers.append(h2h)
        # hidden to output
        h2o = self.build_output_layer()
        self.layers.append(h2o)

    # initialize feature for each node
    def create_features(self):
        features = torch.arange(self.num_nodes)
```

```
        return features

    def build_input_layer(self):
        return RGCNLayer(self.num_nodes, self.h_dim, self.num_rels, self.num_bases,
                         activation=F.relu, is_input_layer=True)

    def build_hidden_layer(self):
        return RGCNLayer(self.h_dim, self.h_dim, self.num_rels, self.num_bases,
                         activation=F.relu)

    def build_output_layer(self):
        return RGCNLayer(self.h_dim, self.out_dim, self.num_rels, self.num_bases,
                         activation=partial(F.softmax, dim=1))

    def forward(self, g):
        if self.features is not None:
            g.ndata['id'] = self.features
        for layer in self.layers:
            layer(g)
        return g.ndata.pop('h')
```

2.2.3.3 处理数据集

本教程使用来自 R-GCN 论文的应用信息学和形式描述方法研究所(AIFB)数据集。

```
# load graph data
from dgl.contrib.data import load_data
data = load_data(dataset='aifb')
num_nodes = data.num_nodes
num_rels = data.num_rels
num_classes = data.num_classes
labels = data.labels
train_idx = data.train_idx
# split training and validation set
val_idx = train_idx[:len(train_idx) // 5]
train_idx = train_idx[len(train_idx) // 5:]

# edge type and normalization factor
edge_type = torch.from_numpy(data.edge_type)
edge_norm = torch.from_numpy(data.edge_norm).unsqueeze(1)

labels = torch.from_numpy(labels).view(-1)
```

输出：

```
Downloading /home/ubuntu/.dgl/aifb.tgz from https://data.dgl.ai/dataset/aifb.tgz...
Loading dataset aifb
```

```
Number of nodes:  8285
Number of edges:  66371
Number of relations:  91
Number of classes:  4
removing nodes that are more than 3 hops away
```

2.2.3.4 创建图和模型

```
# configurations
n_hidden = 16 # number of hidden units
n_bases = -1 # use number of relations as number of bases
n_hidden_layers = 0 # use 1 input layer, 1 output layer, no hidden layer
n_epochs = 25 # epochs to train
lr = 0.01 # learning rate
l2norm = 0 # L2 norm coefficient

# create graph
g = DGLGraph((data.edge_src, data.edge_dst))
g.edata.update({'rel_type': edge_type, 'norm': edge_norm})

# create model
model = Model(len(g),
              n_hidden,
              num_classes,
              num_rels,
              num_bases=n_bases,
              num_hidden_layers=n_hidden_layers)
```

输出：

```
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/base.py:45: DGLWarning: Recommend creating graphs by `dgl.graph(data)` instead of `dgl.DGLGraph(data)`.
  return warnings.warn(message, category=category, stacklevel=1)
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/base.py:45: DGLWarning: DGLGraph.__len__ is deprecated.Please directly call DGLGraph.number_of_nodes.
  return warnings.warn(message, category=category, stacklevel=1)
```

2.2.3.5 训练循环

```
# optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=l2norm)

print("start training...")
model.train()
for epoch in range(n_epochs):
    optimizer.zero_grad()
```

```
    logits = model.forward(g)
    loss = F.cross_entropy(logits[train_idx], labels[train_idx])
    loss.backward()

    optimizer.step()

    train_acc = torch.sum(logits[train_idx].argmax(dim=1) == labels[train_idx])
    train_acc = train_acc.item() / len(train_idx)
    val_loss = F.cross_entropy(logits[val_idx], labels[val_idx])
    val_acc = torch.sum(logits[val_idx].argmax(dim=1) == labels[val_idx])
    val_acc = val_acc.item() / len(val_idx)
    print("Epoch {:05d} | ".format(epoch) +
          "Train Accuracy: {:.4f} | Train Loss: {:.4f} | ".format(
              train_acc, loss.item()) +
          "Validation Accuracy: {:.4f} | Validation loss: {:.4f}".format(
              val_acc, val_loss.item()))
```

输出：

```
start training...
Epoch 00000 | Train Accuracy: 0.2679 | Train Loss: 1.3859 | Validation Accuracy: 0.2500
| Validation loss: 1.3856
Epoch 00001 | Train Accuracy: 0.9554 | Train Loss: 1.3381 | Validation Accuracy: 1.0000
| Validation loss: 1.3518
Epoch 00002 | Train Accuracy: 0.9554 | Train Loss: 1.2653 | Validation Accuracy: 1.0000
| Validation loss: 1.3006
Epoch 00003 | Train Accuracy: 0.9196 | Train Loss: 1.1749 | Validation Accuracy: 1.0000
| Validation loss: 1.2337
Epoch 00004 | Train Accuracy: 0.9196 | Train Loss: 1.0853 | Validation Accuracy: 1.0000
| Validation loss: 1.1581
Epoch 00005 | Train Accuracy: 0.9554 | Train Loss: 1.0091 | Validation Accuracy: 1.0000
| Validation loss: 1.0819
Epoch 00006 | Train Accuracy: 0.9554 | Train Loss: 0.9485 | Validation Accuracy: 1.0000
| Validation loss: 1.0116
Epoch 00007 | Train Accuracy: 0.9554 | Train Loss: 0.9012 | Validation Accuracy: 0.9643
| Validation loss: 0.9516
Epoch 00008 | Train Accuracy: 0.9643 | Train Loss: 0.8651 | Validation Accuracy: 0.9643
| Validation loss: 0.9043
Epoch 00009 | Train Accuracy: 0.9643 | Train Loss: 0.8379 | Validation Accuracy: 0.9643
| Validation loss: 0.8687
Epoch 00010 | Train Accuracy: 0.9732 | Train Loss: 0.8177 | Validation Accuracy: 0.9643
| Validation loss: 0.8426
Epoch 00011 | Train Accuracy: 0.9732 | Train Loss: 0.8028 | Validation Accuracy: 0.9643
| Validation loss: 0.8240
Epoch 00012 | Train Accuracy: 0.9732 | Train Loss: 0.7918 | Validation Accuracy: 0.9643
| Validation loss: 0.8112
```

```
Epoch 00013 | Train Accuracy: 0.9732 | Train Loss: 0.7836 | Validation Accuracy: 0.9643
| Validation loss: 0.8027
Epoch 00014 | Train Accuracy: 0.9732 | Train Loss: 0.7774 | Validation Accuracy: 0.9643
| Validation loss: 0.7973
Epoch 00015 | Train Accuracy: 0.9732 | Train Loss: 0.7727 | Validation Accuracy: 0.9643
| Validation loss: 0.7942
Epoch 00016 | Train Accuracy: 0.9821 | Train Loss: 0.7690 | Validation Accuracy: 0.9643
| Validation loss: 0.7927
Epoch 00017 | Train Accuracy: 0.9821 | Train Loss: 0.7660 | Validation Accuracy: 0.9643
| Validation loss: 0.7924
Epoch 00018 | Train Accuracy: 0.9821 | Train Loss: 0.7635 | Validation Accuracy: 0.9643
| Validation loss: 0.7930
Epoch 00019 | Train Accuracy: 0.9821 | Train Loss: 0.7611 | Validation Accuracy: 0.9643
| Validation loss: 0.7943
Epoch 00020 | Train Accuracy: 0.9821 | Train Loss: 0.7589 | Validation Accuracy: 0.9286
| Validation loss: 0.7961
Epoch 00021 | Train Accuracy: 0.9821 | Train Loss: 0.7566 | Validation Accuracy: 0.9286
| Validation loss: 0.7984
Epoch 00022 | Train Accuracy: 0.9911 | Train Loss: 0.7544 | Validation Accuracy: 0.9286
| Validation loss: 0.8009
Epoch 00023 | Train Accuracy: 1.0000 | Train Loss: 0.7523 | Validation Accuracy: 0.9286
| Validation loss: 0.8037
Epoch 00024 | Train Accuracy: 1.0000 | Train Loss: 0.7504 | Validation Accuracy: 0.9286
| Validation loss: 0.8066
```

2.2.4 第二项任务：链接预测

到目前为止，您已经了解了如何使用 DGL 通过 R-GCN 模型实现实体分类。在知识库设置中，R-GCN 生成的表示可用于发现节点之间的潜在关系。 在 R-GCN 论文中，作者将 R-GCN 生成的实体表示提供给 DistMult 预测模型，以预测可能的关系。

该实现与此处介绍的实现类似，但在 R-GCN 层之上堆叠了一个额外的 DistMult 层。 您可以在我们的`Github Python 代码示例中找到使用 R-GCN 进行链接预测的完整实现。
https://github.com/dmlc/dgl/blob/master/examples/pytorch/rgcn/link_predict.py

3. 线性图神经网络

Author: Qi Huang, Yu Gai, Minjie Wang, Zheng Zhang

在本教程中，您将学习如何通过实现线性图神经网络（LGNN）解决社区检测任务。社区检测或图聚类包括将图中的顶点划分为群集，在群集中节点之间更加相似。

在“图卷积网络”教程中，您学习了如何在半监督设置下对输入图的节点进行分类。您使用图卷积神经网络（GCN）作为图特征的嵌入机制。

为了将图神经网络（GNN）推广到有监督的社区检测中，在研究论文 Supervised Community Detection with Line Graph Neural Networks 中引入

了基于线性图的 GNN 变体。该模型的亮点之一是增强了直接的 GNN 架构，使其可以在由非回溯运算符定义的边邻接的线性图上进行操作。

线性图神经网络（LGNN）显示了 DGL 如何通过混合基本张量运算，稀疏矩阵乘法和消息传递 API 来实现高级图算法。

在以下各节中，您将了解社区检测，线性图，LGNN 及其实现。

3.1 使用 Cora 数据集进行监督的社区检测任务

3.1.1 社区检测

在社区检测任务中，您将相似的节点聚类而不是对其进行标记。通常将节点相似性描述为在每个群集中具有较高的内部密度。

社区检测和节点分类有什么区别?与节点分类相比，社区检测侧重于检索图中的集群信息，而不是给节点分配特定的标签。例如，只要一个节点与其社区成员聚集在一起，无论该节点被分配为“社区 A”，还是“社区 B”，在电影网络分类任务中，将所有“优秀电影”分配为“坏电影”都是一种灾难。

那么，社区检测算法和其他聚类算法（例如 k-means）有什么区别？社区检测算法对图结构数据进行操作。与 k 均值相比，社区检测利用图结构，而不是根据节点的特征简单地对节点进行聚类。

3.1.2 Cora 数据集

为了与 GCN 教程保持一致，您使用 Cora dataset 来演示一个简单的社区检测任务。Cora 是一个科学出版物数据库，有 2708 篇论文，属于七个不同的机器学习领域。在这里，您将 Cora 表示为一个有向图，每个节点都是一篇论文，每个边都是一个引文链接(A ->B 表示 A 引用了 B)。

Cora 自然包含 7 个类，下面的统计表明，每个类都满足我们对 community 的假设，即相同类的节点之间的连接概率高于不同类的节点之间的连接概率。下面的代码片段验证了类内边比类间边多。

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

import dgl
from dgl.data import citation_graph as citegrh

data = citegrh.load_cora()

G = dgl.DGLGraph(data.graph)
labels = th.tensor(data.labels)

# find all the nodes labeled with class 0
label0_nodes = th.nonzero(labels == 0).squeeze()
# find all the edges pointing to class 0 nodes
src, _ = G.in_edges(label0_nodes)
```

```
src_labels = labels[src]
# find all the edges whose both endpoints are in class 0
intra_src = th.nonzero(src_labels == 0)
print('Intra-class edges percent: %.4f' % (len(intra_src) / len(src_labels)))
```

输出：

```
Loading from cache failed, re-processing.
Finished data loading and preprocessing.
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done saving data into cached files.
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.graph will be deprecated, please use
dataset.g instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/ba
se.py:45: DGLWarning: Recommend creating graphs by `dgl.graph(data)` instead of
`dgl.DGLGraph(data)`.
  return warnings.warn(message, category=category, stacklevel=1)
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.label will be deprecated, please use
g.ndata['label'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old,
new))
Intra-class edges percent: 0.6994
```

3.1.3 来自 Cora 的二元社区子图和测试数据集

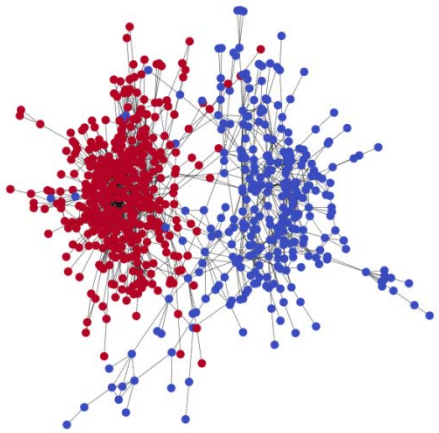
不失一般性，在本教程中，您将该任务的范围限制为二进制社区检测。

注意：要从 Cora 中创建一个实践的二进制社区数据集，首先从原来的 Cora 七个类中提取所有的两类对。对于每一对，您将每个类视为一个社区，并找到至少包含一个跨社区边的最大子图作为训练示例。因此，在这个小数据集中总共有 21 个训练样本。

使用以下代码，您可以可视化其中一个训练样本及其社区结构。

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

train_set = dgl.data.CoraBinary()
G1, pmpd1, label1 = train_set[1]
nx_G1 = G1.to_networkx()
```

```
def visualize(labels, g):
    pos = nx.spring_layout(g, seed=1)
    plt.figure(figsize=(8, 8))
    plt.axis('off')
    nx.draw_networkx(g, pos=pos, node_size=50, cmap=plt.get_cmap('coolwarm'),
                     node_color=labels, edge_color='k',
                     arrows=False, width=0.5, style='dotted', with_labels=False)
visualize(label1, nx_G1)
```



输出：

```
Done saving data into cached files.
Done saving data into cached files.
```

要了解更多信息，请访问原始研究论文以了解如何将其推广到多个社区的案例。

3.1.4 在有监督的环境中进行社区检测

社区检测问题可以通过有监督和无监督的方法来解决。 您可以在受监管的环境中制定社区检测，如下所示：

- Each training example consists of (G, L), where G is a directed graph (V, E). For each node v in V, we assign a ground truth community label $z_v \in \{0, 1\}$.
- The parameterized model $f(G, \theta)$ predicts a label set $\tilde{Z} = f(G)$ for nodes V.
- For each example (G, L), the model learns to minimize a specially designed loss function (equivariant loss) $L_{equivariant} = (\tilde{Z}\ ,\ Z)$

注意：在这种监督下，模型自然可以为每个社区预测标签。 但是，社区分配应与标签排列等价。 为了实现这一目标，在每个正向过程中，我们都采用从标签的所有可能排列计算得出的损失中的最小值。

Mathematically, this means $L_{equivariant} = \min_{\pi \in S_c} -\log(\hat{\pi}, \pi)$, where S_c is the set of all permutations of labels, and $\hat{\pi}$ is the set of predicted labels, $-\log(\hat{\pi}, \pi)$ denotes negative log likelihood.

For instance, for a sample graph with node $\{1, 2, 3, 4\}$ and community assignment $\{A, A, A, B\}$, with each node's label $l \in \{0, 1\}$,The group of all possible permutations $S_c = \{\{0, 0, 0, 1\}, \{1, 1, 1, 0\}\}$.

3.2 线性图神经网络的关键思想

该主题的一项关键创新是线性图的使用。 与以前的教程中的模型不同，消息传递不仅发生在原始图上，例如 来自 Cora 的二元社区子图，而且位于与原始图相关联的线性图上。

3.2.1 什么是线性图?

在图论中，线性图是对原始图的边邻接结构进行编码的图表示。

具体地，线性图 L(G)将原始图 G 的边变成节点。 下面的图表对此进行了说明（摘自研究论文）。

在这里，eA：=（i→j）和 eB：=（j→k）是原始图 G 中的两个边。 在线性图 G_L 中，它们对应于节点v_A^l和v_B^l。

下一个自然的问题是，如何连接线性图中的节点？如何连接两个边？ 在这里，我们使用以下连接规则：

如果 g 中对应的两个边 e_A，e_B 共享一个且仅一个节点，则 lg 中的两个节点 v_A^l，v_B^l连接在一起：eA 的目标节点是 eB 的源节点（j）。

注意：

Mathematically, this definition corresponds to a notion called non-backtracking operator: $B_{(i\to j),(\hat{i}\to\hat{j})} = \begin{cases} 1 \text{ if } j = \hat{i}, \hat{j} \neq i \\ 0 \text{ otherwise} \end{cases}$ where an edge is formed if $B_{node1,node2} = 1$.

3.2.2 LGNN 中的一层，算法结构

LGNN 将一系列线形图神经网络层链接在一起。图表示 x 和它的线图伴侣 y 随数据流的变化如下。（图片失效了。。。）

在 $k-$第层 i的第神经元 l频道更新其嵌入 $x_{i,l}^{(k+1)}$与：

$$
\begin{aligned}
x_{i,l}^{(k+1)} = \rho[&x_i^{(k)}\theta_{1,l}^{(k)} + (Dx^{(k)})_i\theta_{2,l}^{(k)} \\
&+ \sum_{j=0}^{J-1}(A^{2^j}x^k)_i\theta_{3+j,l}^{(k)} \\
&+ [\{\mathrm{Pm}, \mathrm{Pd}\}y^{(k)}]_i\theta_{3+J,l}^{(k)}] \\
&+ \text{skip-connection} \qquad i \in V, l = 1, 2, 3, \ldots b_{k+1}/2
\end{aligned}
$$

然后，线图表示$y_{i,l}^{(k+1)}$与，

$$
\begin{aligned}
y_{i',l'}^{(k+1)} = \rho[&y_{i'}^{(k)}\gamma_{1,l'}^{(k)} + (D_{L(G)}y^{(k)})_{i'}\gamma_{2,l'}^{(k)} \\
&+ \sum_{j=0}^{J-1}(A_{L(G)}^{2^j}y^{k})_i\gamma_{3+j,l'}^{(k)} \\
&+ [\{\mathrm{Pm},\mathrm{Pd}\}^T x^{(k+1)}]_{i'}\gamma_{3+J,l'}^{(k)}] \\
&+ \text{skip-connection} \qquad i' \in V_l, l' = 1,2,3,\dots b_{k+1}'/2
\end{aligned}
$$

Where **skip-connection** refers to performing the same operation without the non-linearity ρ, and with linear projection $\theta_{\{\frac{b_{k+1}}{2}+1,\dots,b_{k+1}-1,b_{k+1}\}}$ and $\gamma_{\{\frac{b_{k+1}}{2}+1,\dots,b_{k+1}-1,b_{k+1}\}}$.

3.3 在 DGL 中实现 LGNN

即使上一节中的方程式可能看起来令人生畏，但在实现 LGNN 之前有助于理解以下信息。

这两个方程是对称的，可以实现为具有不同参数的同一类的两个实例。 第一个方程对图表示 x 进行运算，第二个方程对线图表示 y 进行运算。 让我们将此抽象表示为 f。 那么第一个是 f（x，y; θ x），第二个是 f（y，x， θ y）。 即，将它们参数化以分别计算原始图及其伴随线图的表示形式。

每个方程式由四个项组成。下面以第一个为例。

- $x^{(k)}\theta_{1,l}^{(k)}$，前一层输出的线性投影 $x^{(k)}$，表示为 $\mathrm{prev}(x)$。
- $(Dx^{(k)})\theta_{2,l}^{(k)}$，度算子在上的线性投影$x^{(k)}$，表示为 $\deg(x)$。
- $\sum_{j=0}^{J-1}(A^{2^j}x^{(k)})\theta_{3+j,l}^{(k)}$，是 2^j 邻接运算符$x^{(k)}$，表示为$\mathrm{radius}(x)$.
  -$[\{Pm,Pd\}y^{(k)}]\theta_{3+J,l}^{(k)}$，使用关联矩阵融合另一个图的嵌入信息$\{Pm,Pd\}$，然后是线性投影，表示为$\mathrm{fuse}(y)$。

再次使用不同的参数执行每个项，并且求和后没有非线性。因此，f 可以写成：

$$
\begin{aligned}
f(x^{(k)},y^{(k)}) = \rho[&\mathrm{prev}(x^{(k-1)}) + \deg(x^{(k-1)}) + \mathrm{radius}(x^{k-1}) + \mathrm{fuse}(y^{(k)})] \\
&+\mathrm{prev}(x^{(k-1)}) + \deg(x^{(k-1)}) + \mathrm{radius}(x^{k-1}) + \mathrm{fuse}(y^{(k)})
\end{aligned}
$$

两个方程按以下顺序链接起来：

$$
\begin{aligned}
x^{(k+1)} &= f(x^{(k)},y^{(k)}) \\
y^{(k+1)} &= f(y^{(k)},x^{(k+1)})
\end{aligned}
$$

请记住本概述中列出的意见，然后继续执行。重要的一点是，您对提到的术语使用不同的策略。

注意：通过此说明，您可以更全面地了解{Pm，Pd}。 粗略地说，g 和 lg（线性图）如何与循环简短传播一起工作。 在这里，您将{Pm，Pd}实现为数据集中的 SciPy COO 稀疏矩阵，并在批处理时将它们堆叠为张量。 另一种批处理解决方案是将{Pm，Pd}视为二部图的邻接矩阵，该图将线性图的特征映射到图的特征，反之亦然。

3.3.1 实现 prev 和 deg 作为张量操作

线性投影和度运算都是简单的矩阵乘法。将它们编写为 PyTorch 张量操作。

在中__init__，您可以定义投影变量。

```
self.linear_prev = nn.Linear(in_feats, out_feats)
self.linear_deg = nn.Linear(in_feats, out_feats)
```

在 forward()，prev 和 deg 与任何其他 PyTorch 张量操作相同。

```
prev_proj = self.linear_prev(feat_a)
deg_proj = self.linear_deg(deg * feat_a)
```

3.3.2 在 DGL 中将 radiusradius 实现为消息传递

正如 GCN 教程中讨论的那样，您可以将一个邻接运算符表述为一步传递消息。作为概括，可以将 2^j 邻接操作表示为执行消息传递的 2^j 步骤。 因此，求和等于对节点的 2^j，j = 0,1,2..步消息传递进行表示，即在每个节点的 2^j 邻域中收集信息。

在__init__中，定义在消息传递的每个 2^j 步骤中使用的投影变量。。

```
self.linear_radius = nn.ModuleList(
        [nn.Linear(in_feats, out_feats) for i in range(radius)])
```

在__forward__中，使用以下函数 aggregate_radius()收集来自多个跃点的数据。在以下代码中可以看到。 注意，update_all 被多次调用。

```
# Return a list containing features gathered from multiple radius.
import dgl.function as fn
def aggregate_radius(radius, g, z):
    # initializing list to collect message passing result
    z_list = []
    g.ndata['z'] = z
    # pulling message from 1-hop neighbourhood
    g.update_all(fn.copy_src(src='z', out='m'), fn.sum(msg='m', out='z'))
    z_list.append(g.ndata['z'])
    for i in range(radius - 1):
        for j in range(2 ** i):
            #pulling message from 2^j neighborhood
            g.update_all(fn.copy_src(src='z', out='m'), fn.sum(msg='m', out='z'))
        z_list.append(g.ndata['z'])
    return z_list
```

3.3.3 实现 fuse 为稀疏矩阵乘法实

{Pm，Pd}是一个稀疏矩阵，每列上只有两个非零条目。 因此，您可以将其构造为数据集中的稀疏矩阵，并实现 fuse 作为稀疏矩阵乘法。

在__forward__中：

```
fuse = self.linear_fuse(th.mm(pm_pd, feat_b))
```

3.3.4 完成 f(x, y)

最后，以下内容显示了如何将所有项汇总在一起，将其传递给 skip connection 以及 batch norm。

```
result = prev_proj + deg_proj + radius_proj + fuse
```

传递结果给 skip connection。

```
result = th.cat([result[:, :n], F.relu(result[:, n:])], 1)
```

传递结果给 batch norm。

```
result = self.bn(result) #Batch Normalization.
```

这是一个 LGNN 层抽象 f（x，y）的完整代码：

```
class LGNNCore(nn.Module):
    def __init__(self, in_feats, out_feats, radius):
        super(LGNNCore, self).__init__()
        self.out_feats = out_feats
        self.radius = radius

        self.linear_prev = nn.Linear(in_feats, out_feats)
        self.linear_deg = nn.Linear(in_feats, out_feats)
        self.linear_radius = nn.ModuleList(
                [nn.Linear(in_feats, out_feats) for i in range(radius)])
        self.linear_fuse = nn.Linear(in_feats, out_feats)
        self.bn = nn.BatchNorm1d(out_feats)

    def forward(self, g, feat_a, feat_b, deg, pm_pd):
        # term "prev"
        prev_proj = self.linear_prev(feat_a)
        # term "deg"
        deg_proj = self.linear_deg(deg * feat_a)

        # term "radius"
        # aggregate 2^j-hop features
        hop2j_list = aggregate_radius(self.radius, g, feat_a)
        # apply linear transformation
        hop2j_list = [linear(x) for linear, x in zip(self.linear_radius, hop2j_list)]
        radius_proj = sum(hop2j_list)

        # term "fuse"
        fuse = self.linear_fuse(th.mm(pm_pd, feat_b))

        # sum them together
        result = prev_proj + deg_proj + radius_proj + fuse

        # skip connection and batch norm
```

```
        n = self.out_feats // 2
        result = th.cat([result[:, :n], F.relu(result[:, n:])], 1)
        result = self.bn(result)

        return result
```

3.3.5 将 LGNN 抽象链接为 LGNN 层

为了实现:

$$x^{(k+1)} = f(x^{(k)}, y^{(k)})$$
$$y^{(k+1)} = f(y^{(k)}, x^{(k+1)})$$

如示例代码中所示，将两个 LGNNCore 实例链接在一起，并在前向传递中使用不同的参数。

```
class LGNNLayer(nn.Module):
    def __init__(self, in_feats, out_feats, radius):
        super(LGNNLayer, self).__init__()
        self.g_layer = LGNNCore(in_feats, out_feats, radius)
        self.lg_layer = LGNNCore(in_feats, out_feats, radius)

    def forward(self, g, lg, x, lg_x, deg_g, deg_lg, pm_pd):
        next_x = self.g_layer(g, x, lg_x, deg_g, pm_pd)
        pm_pd_y = th.transpose(pm_pd, 0, 1)
        next_lg_x = self.lg_layer(lg, lg_x, x, deg_lg, pm_pd_y)
        return next_x, next_lg_x
```

3.3.6 链接 LGNN 层

定义一个具有三个隐藏层的 LGNN，如以下示例所示。

```
class LGNN(nn.Module):
    def __init__(self, radius):
        super(LGNN, self).__init__()
        self.layer1 = LGNNLayer(1, 16, radius)  # input is scalar feature
        self.layer2 = LGNNLayer(16, 16, radius)  # hidden size is 16
        self.layer3 = LGNNLayer(16, 16, radius)
        self.linear = nn.Linear(16, 2)  # predice two classes

    def forward(self, g, lg, pm_pd):
        # compute the degrees
        deg_g = g.in_degrees().float().unsqueeze(1)
        deg_lg = lg.in_degrees().float().unsqueeze(1)
        # use degree as the input feature
        x, lg_x = deg_g, deg_lg
        x, lg_x = self.layer1(g, lg, x, lg_x, deg_g, deg_lg, pm_pd)
        x, lg_x = self.layer2(g, lg, x, lg_x, deg_g, deg_lg, pm_pd)
```

```
        x, lg_x = self.layer3(g, lg, x, lg_x, deg_g, deg_lg, pm_pd)
        return self.linear(x)
```

3.4 训练与推断

首先加载数据：

```
from torch.utils.data import DataLoader
training_loader = DataLoader(train_set,
                             batch_size=1,
                             collate_fn=train_set.collate_fn,
                             drop_last=True)
```

接下来，定义主要的训练循环。 请注意，每个训练样本均包含三个对象：DGLGraph，SciPy 稀疏矩阵 pmpd 和 numpy.ndarray 中的标签数组。 使用以下命令生成线性图：

```
lg = g.line_graph(backtracking=False)
```

请注意，需要 backtracking = False 才能正确模拟非回溯操作。 我们还定义了一个实用函数，将 SciPy 稀疏矩阵转换为 torch 稀疏张量。

```
# Create the model
model = LGNN(radius=3)
# define the optimizer
optimizer = th.optim.Adam(model.parameters(), lr=1e-2)

# A utility function to convert a scipy.coo_matrix to torch.SparseFloat
def sparse2th(mat):
    value = mat.data
    indices = th.LongTensor([mat.row, mat.col])
    tensor = th.sparse.FloatTensor(indices, th.from_numpy(value).float(), mat.shape)
    return tensor

# Train for 20 epochs
for i in range(20):
    all_loss = []
    all_acc = []
    for [g, pmpd, label] in training_loader:
        # Generate the line graph.
        lg = g.line_graph(backtracking=False)
        # Create torch tensors
        pmpd = sparse2th(pmpd)
        label = th.from_numpy(label)

        # Forward
        z = model(g, lg, pmpd)

        # Calculate loss:
        # Since there are only two communities, there are only two permutations
```

```
        #  of the community labels.
        loss_perm1 = F.cross_entropy(z, label)
        loss_perm2 = F.cross_entropy(z, 1 - label)
        loss = th.min(loss_perm1, loss_perm2)

        # Calculate accuracy:
        _, pred = th.max(z, 1)
        acc_perm1 = (pred == label).float().mean()
        acc_perm2 = (pred == 1 - label).float().mean()
        acc = th.max(acc_perm1, acc_perm2)
        all_loss.append(loss.item())
        all_acc.append(acc.item())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    niters = len(all_loss)
    print("Epoch %d | loss %.4f | accuracy %.4f" % (i,
        sum(all_loss) / niters, sum(all_acc) / niters))
```

输出：

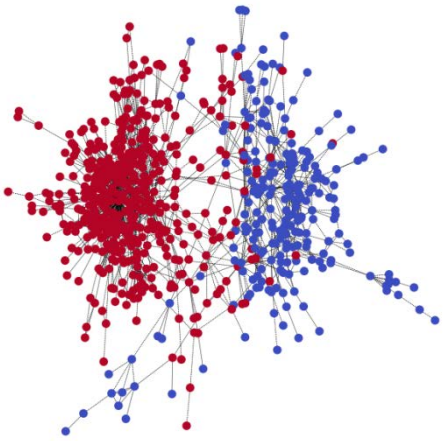
```
Epoch 0 | loss 0.5494 | accuracy 0.7177
Epoch 1 | loss 0.5023 | accuracy 0.7595
Epoch 2 | loss 0.4787 | accuracy 0.7847
Epoch 3 | loss 0.4638 | accuracy 0.7731
Epoch 4 | loss 0.4444 | accuracy 0.8015
Epoch 5 | loss 0.4378 | accuracy 0.8014
Epoch 6 | loss 0.4354 | accuracy 0.7910
Epoch 7 | loss 0.4339 | accuracy 0.8086
Epoch 8 | loss 0.4253 | accuracy 0.8014
Epoch 9 | loss 0.4193 | accuracy 0.7998
Epoch 10 | loss 0.4770 | accuracy 0.7755
Epoch 11 | loss 0.4332 | accuracy 0.8055
Epoch 12 | loss 0.4333 | accuracy 0.7965
Epoch 13 | loss 0.4334 | accuracy 0.7924
Epoch 14 | loss 0.4506 | accuracy 0.7956
Epoch 15 | loss 0.4474 | accuracy 0.7964
Epoch 16 | loss 0.4319 | accuracy 0.8075
Epoch 17 | loss 0.4215 | accuracy 0.8174
Epoch 18 | loss 0.4328 | accuracy 0.8049
Epoch 19 | loss 0.4099 | accuracy 0.8213
```

3.5 可视化训练进程

您可以通过一个训练示例将网络的社区预测以及 ground truth 可视化。 从以

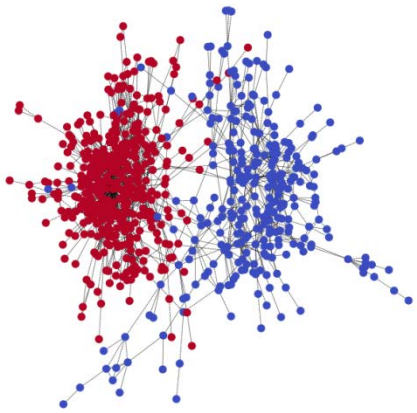
下代码示例开始。

```
pmpd1 = sparse2th(pmpd1)
LG1 = G1.line_graph(backtracking=False)
z = model(G1, LG1, pmpd1)
_, pred = th.max(z, 1)
visualize(pred, nx_G1)
```



与 ground truth 相比。 请注意，这两个社区的颜色可能相反，因为该模型用于正确预测分区。

```
visualize(label1, nx_G1)
```



3.6 批处理并行图

LGNN 收集了一系列不同的图。 您可能会考虑批处理是否可用于并行性。

批处理已进入数据加载器本身。 在 PyTorch 数据加载器的 collate_fn 中，使用 DGL 的 batched_graph API 对图进行批处理。 DGL 通过将它们合并成一个大图来批量处理图，每个小图的邻接矩阵都是沿着大图邻接矩阵对角线的一个块。将

{math，{Pm，Pd}}连接为块对角线矩阵，对应于 DGL 批处理图 API。

```
def collate_fn(batch):
    graphs, pmpds, labels = zip(*batch)
    batched_graphs = dgl.batch(graphs)
    batched_pmpds = sp.block_diag(pmpds)
    batched_labels = np.concatenate(labels, axis=0)
    return batched_graphs, batched_pmpds, batched_labels
```

您可以在 Github 上的图神经网络社区检测（CDGNN）上找到完整的代码。
https://github.com/dmlc/dgl/tree/master/examples/pytorch/line_graph

4. 图注意力网络

Authors: Hao Zhang, Mufei Li, Minjie Wang Zheng Zhang

在本教程中，您将了解图注意力网络（GAT）以及如何在 PyTorch 中实现它。您还可以学习可视化并了解注意力机制所学的内容。

Graph Convolutional Network (GCN),中描述的研究表明，结合局部图结构和节点级特征可以在节点分类任务上产生良好的性能。 但是，GCN 聚合的方式取决于结构，这可能会损害其通用性。

一种解决方法是按研究论文 GraphSAGE.中所述简单地平均所有邻居节点特征。 但是， Graph Attention Network 提出了另一种类型的聚合。 GAN 以关注的方式使用特征依赖和无结构规格化的加权邻居特征。

4.1 将注意力引入到 GCN

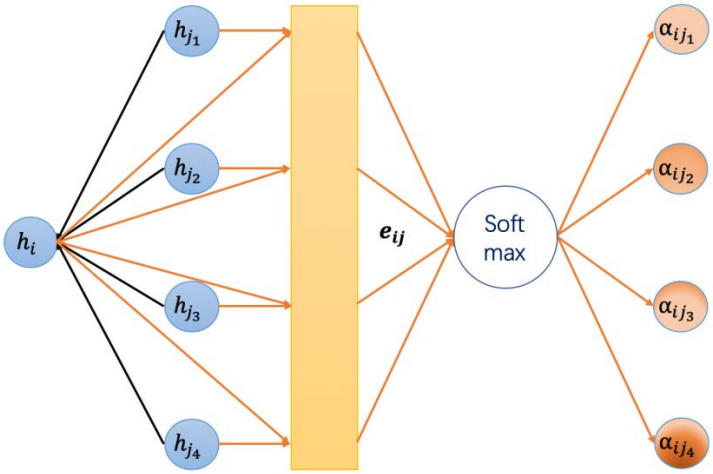
GAT 和 GCN 之间的主要区别在于如何汇总来自一簇社区的信息。

对于 GCN，图卷积运算会生成邻居节点特征的归一化总和。

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right)$$

其中 N(i)是其 one-hop 邻居的集合（要在集合中包含 v_i，只需向每个节点添加一个自环），$c_{ij} = \sqrt{|N(i)|}\sqrt{|N(j)|}$是基于图结构的归一化常数， σ 是激活函数（GCN 使用 ReLU），而 $W^{(l)}$是用于节点特征转换的共享权重矩阵。 GraphSAGE 中提出的另一种模型采用相同的更新规则，不同之处在于它们设置了 $c_{ij} = |N(i)|$。

GAT 引入了注意力机制，以替代静态归一化卷积操作。 下面是根据层 l 的嵌入计算层 l + 1 的节点嵌入$h_i^{(l+1)}$的等式。



$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \mathrm{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

说明：

- 等式（1）是嵌入$h_i^{(l)}$的下层的线性变换，而 W$^{(l)}$是其可学习的权重矩阵。
- 公式（2）计算两个邻居之间的成对非标准化注意力得分。在这里，它首先拼接两个节点的 zz 嵌入，其中||表示拼接，然后取其点积与可学习的权重向量$\vec{a}^{l()}$，最后应用 LeakyReLU。这种注意形式通常称为加性注意，与 Transformer 模型中的点积注意相反。
- 公式（3）应用 softmax 对每个节点的传入边的注意分数进行归一化。
- 等式（4）与 GCN 相似。来自邻居的嵌入信息被聚合在一起，根据注意力分数进行缩放。

论文中还有其他详细信息，例如 dropout 和 skip connection。为了简单起见，本教程中省略了这些详细信息。要查看更多详细信息，请下载完整示例。

https://github.com/dmlc/dgl/blob/master/examples/pytorch/gat/gat.py

本质上，GAT 只是一种不同的聚合函数，它关注邻居的特征，而不是简单的均值聚合。

4.2 在 DGL 中的 GAT

DGL 提供了 DGL.nn.<backend>下的 GAT 层的现成实现。只需导入 GATConv，如下所示。

```
from dgl.nn.pytorch import GATConv
```

读者可以跳过下面对实现的分步解释，直接跳到“把所有东西放在一起以获得训练和可视化结果”。

https://docs.dgl.ai/tutorials/models/1_gnn/9_gat.html#put-everything-together

首先，您将对如何在 DGL 中实现 GATLayer 模块有一个总体印象。在本节中，上面的四个方程式一次分解为一个。

```
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):
        """Reinitialize learnable parameters."""
        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)
        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')
```

4.2.1 公式（1）

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, (1)$$

第一个是线性变换。它很常见，可以很容易地在 Pytorch 中使用 torch.nn. linear 实现。

4.2.2 公式（2）

$$e_{ij}^{(l)} = \mathrm{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)}|z_j^{(l)})), (2)$$

使用相邻节点 i 和 j 的嵌入来计算未归一化的注意力得分 e_{ij}。 这表明可以将注意力得分视为边数据，可以通过 apply_edges API 计算得出。 apply_edges 的参数是 Edge UDF，其定义如下：

```
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)}
```

这里再次使用 PyTorch 的线性变换 attn_fc 实现了与可学习权向量$\overrightarrow{a^{(l)}}$的点积。注意，apply_edges 将在一个张量中批处理所有的边数据，因此 cat，这里的 attn_fc 将并行应用于所有的边。

4.2.3 公式（3）和（4）

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in\mathcal{N}(i)}\exp(e_{ik}^{(l)})}, \quad (3)$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}^{(l)} z_j^{(l)}\right), \quad (4)$$

与 GCN 相似，update_all API 用于触发所有节点上的消息传递。 消息函数发出两个张量：源节点的变换 z 嵌入和每个边上的非标准化注意力得分 e。 然后 reduce 函数执行两项任务：

- 使用 softmax 对注意分数进行规范化(式(3))。
- 用注意力得分加权来聚合邻居嵌入(公式(4))。

这两个任务都首先从邮箱中获取数据，然后在批量处理邮件的第二个维度（dim = 1）上对其进行操作。

```
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}
```

4.2.4 多头注意力

与 ConvNet 中的多通道相似，GAT 引入了多头注意，以加强模型容能力，稳定学习过程。每个注意头都有自己的参数，其输出可以通过两种方式合并:

$$\text{concatenation}: h_i^{(l+1)} = ||_{k=1}^{K}\sigma\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}^{k} W^k h_j^{(l)}\right)$$

或者

$$\text{average}: h_i^{(l+1)} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)}\alpha_{ij}^{k} W^k h_j^{(l)}\right)$$

其中 K 是头的次数。可以对中间层进行拼接，对最后一层取平均值。

使用上面定义的 single-head GATLayer 作为下面 MultiHeadGATLayer 的构建块:

```
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)
            return torch.cat(head_outs, dim=1)
        else:
            # merge using average
            return torch.mean(torch.stack(head_outs))
```

4.2.5 将所有放在一起

现在，您可以定义一个两层的 GAT 模型。

```
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
```

然后，我们使用 DGL 的内置数据模块加载 Cora 数据集。

```
from dgl import DGLGraph
from dgl.data import citation_graph as citegrh
import networkx as nx

def load_cora_data():
    data = citegrh.load_cora()
    features = torch.FloatTensor(data.features)
    labels = torch.LongTensor(data.labels)
    mask = torch.BoolTensor(data.train_mask)
    g = DGLGraph(data.graph)
    return g, features, labels, mask
```

训练循环与 GCN 教程中的完全相同。

```
import time
import numpy as np

g, features, labels, mask = load_cora_data()
```

```
# create the model, 2 heads, each head has hidden size 8
net = GAT(g,
          in_dim=features.size()[1],
          hidden_dim=8,
          out_dim=7,
          num_heads=2)

# create optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)

# main loop
dur = []
for epoch in range(30):
    if epoch >= 3:
        t0 = time.time()

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

    optimizer.zero_grad()
    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)))
```

输出：

```
Loading from cache failed, re-processing.
Finished data loading and preprocessing.
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done saving data into cached files.
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/da
ta/utils.py:285: UserWarning: Property dataset.feat will be deprecated, please use
g.ndata['feat'] instead.
```

```
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old, new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/data/utils.py:285: UserWarning: Property dataset.label will be deprecated, please use g.ndata['label'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old, new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/data/utils.py:285: UserWarning: Property dataset.train_mask will be deprecated, please use g.ndata['train_mask'] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old, new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/data/utils.py:285: UserWarning: Property dataset.graph will be deprecated, please use dataset.g instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old, new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/base.py:45: DGLWarning: Recommend creating graphs by `dgl.graph(data)` instead of `dgl.DGLGraph(data)`.
  return warnings.warn(message, category=category, stacklevel=1)
/home/ubuntu/.pyenv/versions/miniconda3-latest/lib/python3.7/site-packages/numpy/core/fromnumeric.py:3257: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/home/ubuntu/.pyenv/versions/miniconda3-latest/lib/python3.7/site-packages/numpy/core/_methods.py:161: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
Epoch 00000 | Loss 1.9450 | Time(s) nan
Epoch 00001 | Loss 1.9428 | Time(s) nan
Epoch 00002 | Loss 1.9405 | Time(s) nan
Epoch 00003 | Loss 1.9383 | Time(s) 0.2641
Epoch 00004 | Loss 1.9360 | Time(s) 0.2665
Epoch 00005 | Loss 1.9338 | Time(s) 0.2710
Epoch 00006 | Loss 1.9315 | Time(s) 0.2704
Epoch 00007 | Loss 1.9293 | Time(s) 0.2695
Epoch 00008 | Loss 1.9270 | Time(s) 0.2690
Epoch 00009 | Loss 1.9247 | Time(s) 0.2691
Epoch 00010 | Loss 1.9225 | Time(s) 0.2688
Epoch 00011 | Loss 1.9202 | Time(s) 0.2686
Epoch 00012 | Loss 1.9179 | Time(s) 0.2684
Epoch 00013 | Loss 1.9156 | Time(s) 0.2692
Epoch 00014 | Loss 1.9133 | Time(s) 0.2688
Epoch 00015 | Loss 1.9110 | Time(s) 0.2685
Epoch 00016 | Loss 1.9087 | Time(s) 0.2682
```

```
Epoch 00017 | Loss 1.9063 | Time(s) 0.2681
Epoch 00018 | Loss 1.9040 | Time(s) 0.2680
Epoch 00019 | Loss 1.9016 | Time(s) 0.2680
Epoch 00020 | Loss 1.8993 | Time(s) 0.2692
Epoch 00021 | Loss 1.8969 | Time(s) 0.2691
Epoch 00022 | Loss 1.8945 | Time(s) 0.2688
Epoch 00023 | Loss 1.8921 | Time(s) 0.2687
Epoch 00024 | Loss 1.8897 | Time(s) 0.2691
Epoch 00025 | Loss 1.8873 | Time(s) 0.2689
Epoch 00026 | Loss 1.8848 | Time(s) 0.2688
Epoch 00027 | Loss 1.8824 | Time(s) 0.2688
Epoch 00028 | Loss 1.8799 | Time(s) 0.2692
Epoch 00029 | Loss 1.8774 | Time(s) 0.2690
```

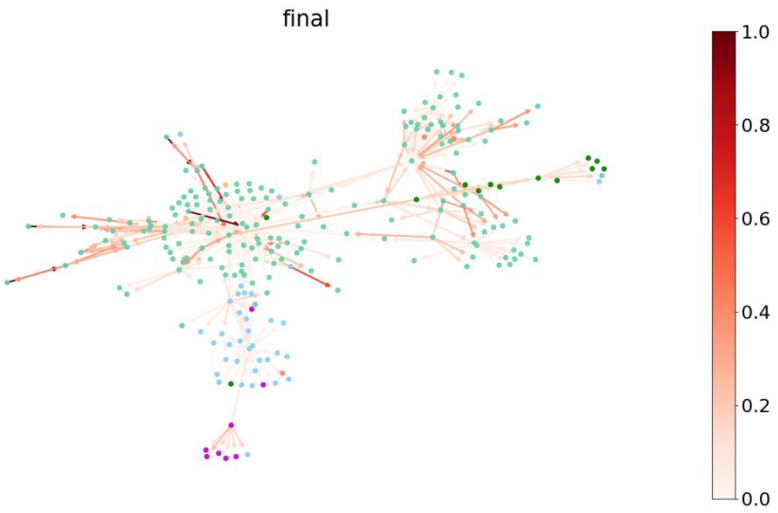
4.3 可视化和理解所学注意力

下表总结了在 GAT 论文中报告并通过 DGL 实现获得的 Cora 模型性能。

Model	Accuracy
GCN (paper)	81.4 ± 0.5
GCN (dgl)	82.05 ± 0.33
GAT (paper)	83.0 ± 0.7
GAT (dgl)	83.69 ± 0.529

我们的模型学到了什么样的注意力分布？

由于注意权重 a_{ij} 与边相关联，因此可以通过为边着色来形象化它。在下面，您可以选择 Cora 的一个子图，并绘制最后一个 GATLayer 的注意力权重。 节点根据其标签进行着色，而边根据注意权重的大小进行着色，这可以通过右侧的色条来引用。

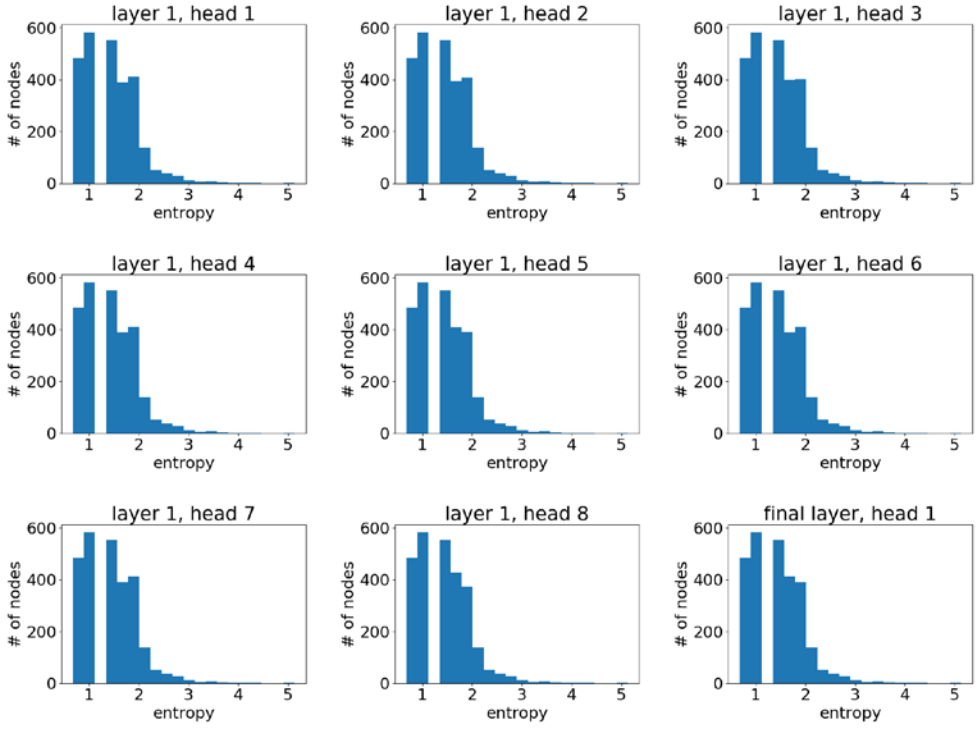


您可以看到该模型似乎学习了不同的注意力权重。 若要更全面地了解分布，请测量注意力分布的熵。 对于任何节点 i，$\{\alpha_{ij}\}_{j\in N(i)}$在其所有邻居上形成一个离散的概率分布，其熵为:

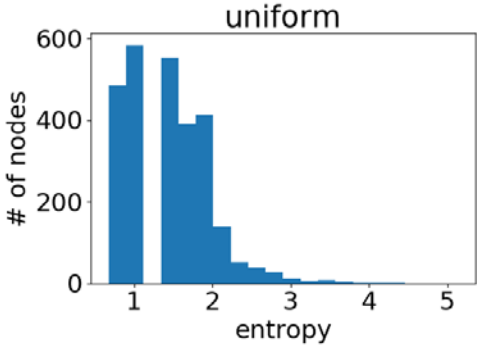
$$H(\alpha_{ij_{j\in\mathcal{N}(i)}}) = -\sum_{j\in\mathcal{N}(i)} \alpha_{ij}\log\alpha_{ij}$$

熵低意味着集中度高，反之亦然。 熵为 0 表示所有注意力都集中在一个源节点上。 均匀分布具有 log(N(i))的最高熵。 理想情况下，您希望看到模型学习较低的熵分布（即，一个或两个邻居比其他邻居重要得多）。

注意，由于节点可以具有不同的度数，因此最大熵也将不同。 因此，您可以绘制整个图中所有节点的熵值的聚合直方图。 以下是每个注意头学习到的注意直方图。



作为参考，以下是所有节点均具有统一注意力权重分布的直方图。



可以看到，学习到的注意力值与均匀分布非常相似（即所有邻居都同样重要）。这部分解释了为什么 GAT 在 Cora 上的性能接近 GCN 的性能（根据作者的报告结果，100 次运行的平均准确度差异小于 2%）。 注意并不重要，因为它的区别不大。

这是否意味着注意力机制没有用？ 没有！ 另一个不同的数据集表现出完全不同的模式，如下所示。

4.4 蛋白质相互作用（PPI）网络

此处使用的 PPI 数据集由对应于不同人体组织的 2424 个图组成。 节点最多可以具有 121121 种标签，因此节点的标签表示为大小为 121121 的二进制张量。任务是预测节点标签。

使用 2020 图进行训练，使用 22 用于验证，使用 22 用于测试。 每个图的平均节点数为 23722372。每个节点具有 5050 个特征，这些特征由位置基因集，基序基因集和免疫学特征组成。至关重要的是，在训练过程中完全看不见测试图，这种设置称为“归纳学习”。

比较此任务上 1010 次随机运行的 GAT 和 GCN 的性能，并在验证集上使用超参数搜索来找到最佳模型。

Model	F1 Score(micro)
GAT	0.975 ± 0.006
GCN	0.509 ± 0.025
Paper	0.973 ± 0.002

上表是该实验的结果，您可以使用 micro F1 分数评估模型性能。

Note

Below is the calculation process of F1 score:

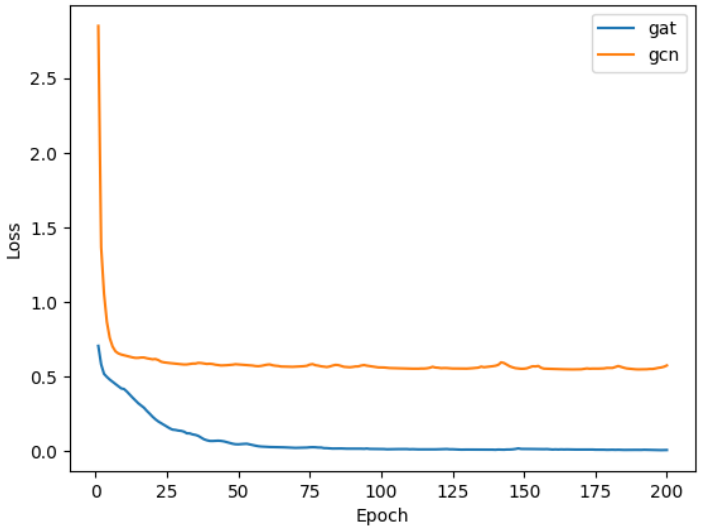
$$precision = \frac{\sum_{t=1}^{n} TP_t}{\sum_{t=1}^{n}(TP_t + FP_t)}$$

$$recall = \frac{\sum_{t=1}^{n} TP_t}{\sum_{t=1}^{n}(TP_t + FN_t)}$$

$$F1_{micro} = 2\frac{precision * recall}{precision + recall}$$

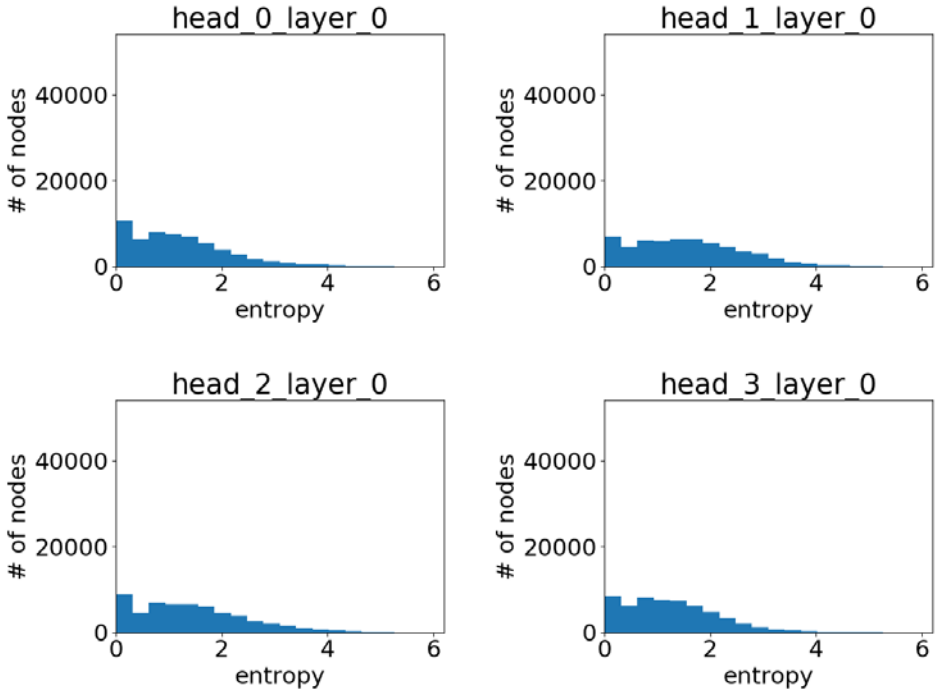
- TP_t represents for number of nodes that both have and are predicted to have label t
- FP_t represents for number of nodes that do not have but are predicted to have label t
- FN_t represents for number of output classes labeled as t but predicted as others.
- n is the number of labels, i.e. 121 in our case.

在训练期间，使用 BCEWithLogitsLoss 作为损失函数。 GAT 和 GCN 的学习曲线如下所示； 显而易见的是，与 GCN 相比，GAT 的显着性能优势。

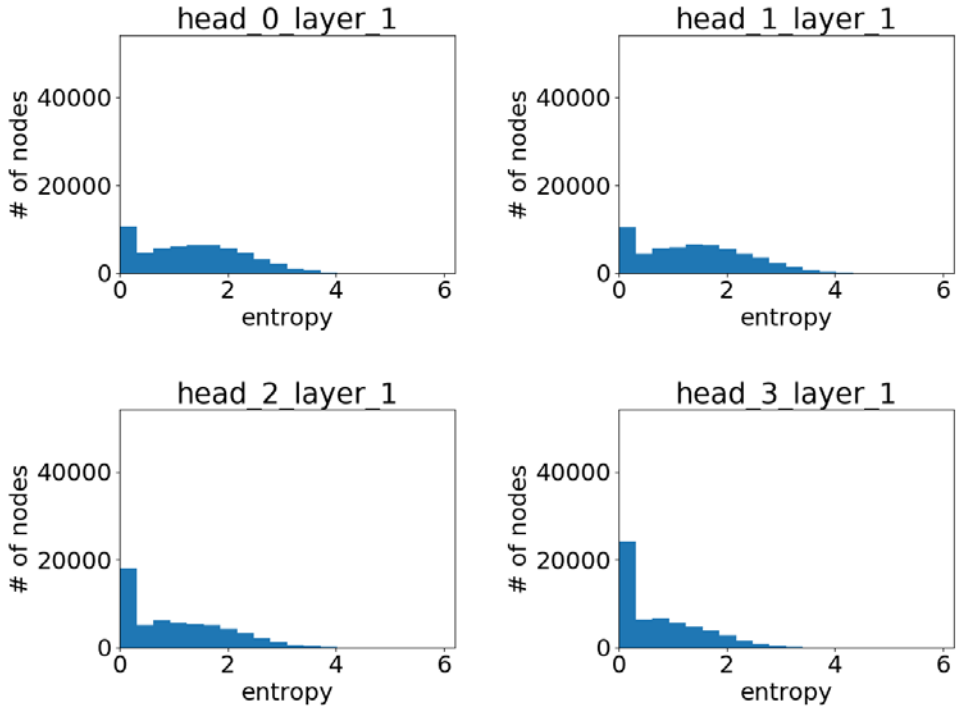


像以前一样，您可以通过显示节点式注意熵的直方图来对学习到的注意事项进行统计理解。 以下是不同注意力层学习的注意力直方图。

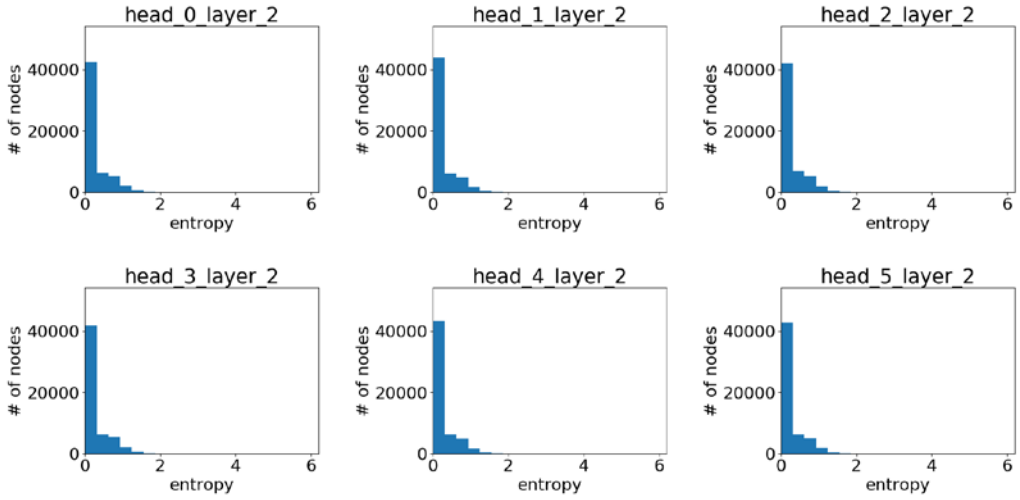
在第 1 层中学习到的注意力：



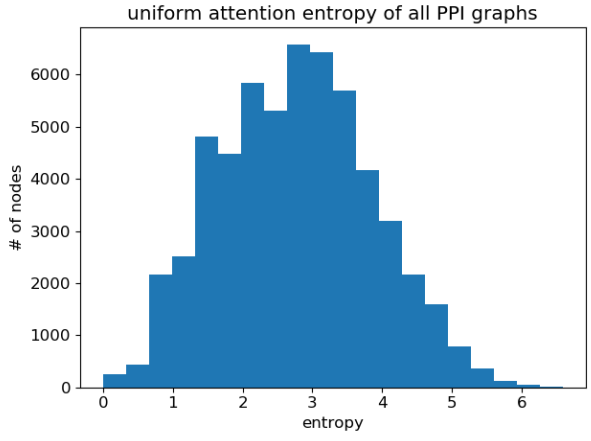
在第 2 层中学习到的注意力：



在最终层中学习到的注意力：



再次，与均匀分布比较：



显然，GAT 确实获得了敏锐的关注权重！ 各层上也有清晰的图案：层越深，注意力越清晰。

与在 Cora 数据集中 GAT 的收益最少不同，对于 PPI 而言，GAT 和其他 GAT 论文中比较的 GNN 变量之间存在显著的性能差距(至少 20%)，而且两者之间的注意分布也明显不同。虽然这值得进一步研究，但一个直接的结论是，GAT 的优势可能更多地在于它处理具有更复杂邻域结构的图的能力。

4.5 下一个是什么？

到目前为止，您已经了解了如何使用 DGL 来实现 GAT。 我们还缺少一些遗漏的详细信息，例如，dropout，skip connection 和超参数调整，这些实践不涉及 DGL 相关概念。 有关更多信息，请查看完整示例。
https://github.com/dmlc/dgl/blob/master/examples/pytorch/gat/gat.py
下一个教程介绍了如何通过并行化多个关注头和 SPMV 优化来加速 GAT 模型。

5. DGL 中的 Tree-LSTM

- **Tree-LSTM** [paper] [tutorial] [PyTorch code]: 句子具有固有的结构，只需将它们简单地视为序列即可将其丢弃。 Tree-LSTM 是一个功能强大的模型，它通过使用诸如语法分析树之类的现有语法结构来学习表示形式。训练中的挑战在于，仅通过将句子填充到最大长度就不再起作用。 不同句子的树具有不同的大小和拓扑。 DGL 通过将树添加到更大的容器图中，然后使用消息传递来探索最大的并行性来解决此问题。 批处理是实现此目的的关键 API。

Author: Zihao Ye, Qipeng Guo, Minjie Wang, Jake Zhao, Zheng Zhang

在本教程中，您将学习使用 Tree-LSTM 网络进行情感分析。 Tree-LSTM 是长短时记忆（LSTM）网络到树结构网络拓扑的概括。

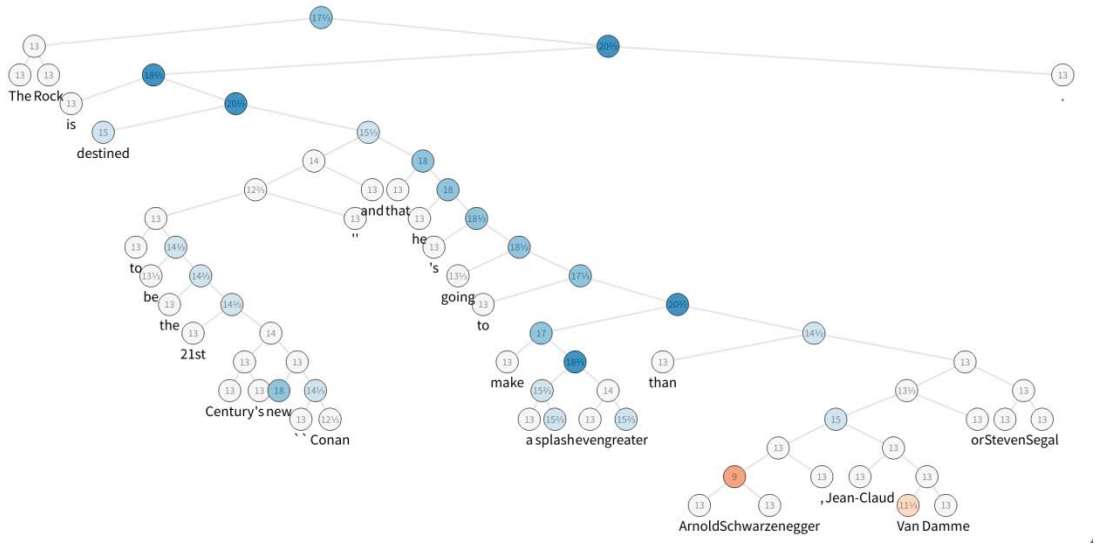
Tree-LSTM 结构首先由 Kai 等人在 ACL2015 论文中引入：

Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks 核心思想是通过将链结构 LSTM 扩展为树结构 LSTM，为语言任务引入语法信息。 The dependency tree and constituency tree techniques are leveraged to obtain a ‘‘latent tree’’.

训练 Tree-LSTM 的挑战是分批处理-----这是机器学习中加速优化的标准技术。但是，由于树木本质上通常具有不同的形状，因此并行化是不平凡的。 DGL 提供了一种替代方法。 将所有树合并为一个图，然后在每棵树的结构的引导下，诱导消息通过它们。

5.1 任务和数据集

此处的步骤使用 dgl.data 中的 Stanford Sentiment Treebank 。 数据集提供了细粒度的树级情感注释。 有五类：非常消极，消极，中立，积极和非常积极，它们表示当前子树中的情绪。 选区树中的非叶子节点不包含单词，因此请使用特殊的 PAD_WORD 标记来表示它们。 在训练和推理期间，它们的嵌入将被屏蔽为全零。



该图显示了 SST 数据集的一个样本，这是一个选区分析树，其节点标记有情感。 为了加快处理速度，请构建一个包含五个句子的小集合，然后看看第一个句子。

```
from collections import namedtuple

import dgl
from dgl.data.tree import SSTDataset


SSTBatch = namedtuple('SSTBatch', ['graph', 'mask', 'wordid', 'label'])

# Each sample in the dataset is a constituency tree. The leaf nodes
# represent words. The word is an int value stored in the "x" field.
# The non-leaf nodes have a special word PAD_WORD. The sentiment
# label is stored in the "y" feature field.
trainset = SSTDataset(mode='tiny')  # the "tiny" set has only five trees
tiny_sst = trainset.trees
num_vocabs = trainset.num_vocabs
num_classes = trainset.num_classes

vocab = trainset.vocab # vocabulary dict: key -> id
inv_vocab = {v: k for k, v in vocab.items()} # inverted vocabulary dict: id -> word

a_tree = tiny_sst[0]
for token in a_tree.ndata['x'].tolist():
    if token != trainset.PAD_WORD:
        print(inv_vocab[token], end=" ")
```

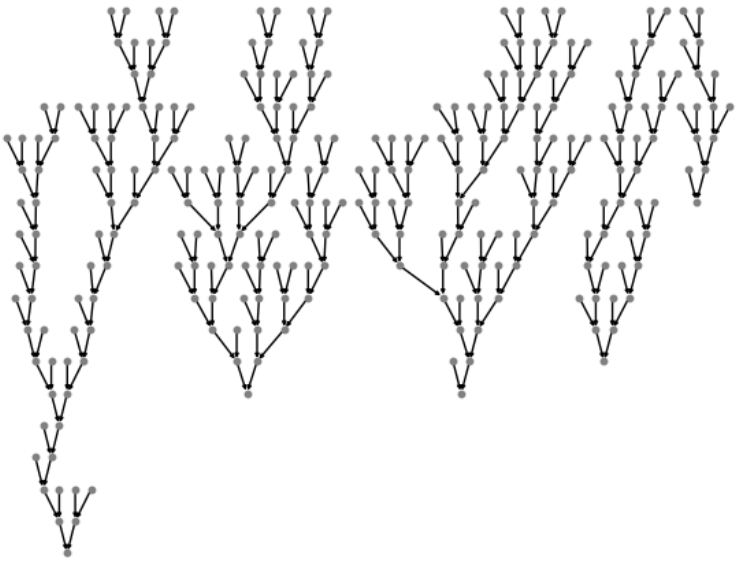
输出：

```
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/data/utils.py:285: UserWarning: Property dataset.trees will be deprecated, please use [dataset[i] for i in len(dataset)] instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old, new))
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/data/utils.py:285: UserWarning: Property dataset.num_vocabs will be deprecated, please use dataset.vocab_size instead.
  warnings.warn('Property {} will be deprecated, please use {} instead.'.format(old, new))
the rock is destined to be the 21st century 's new `` conan '' and that he 's going to make a splash even greater than arnold schwarzenegger , jean-claud van damme or steven segal .
```

5.2 步骤 1：批量化

使用 batch() API 将所有树添加到一个图中。

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

graph = dgl.batch(tiny_sst)
def plot_tree(g):
    # this plot requires pygraphviz package
    pos = nx.nx_agraph.graphviz_layout(g, prog='dot')
    nx.draw(g, pos, with_labels=False, node_size=10,
            node_color=[[.5, .5, .5]], arrowsize=4)
    plt.show()

plot_tree(graph.to_networkx())
```



您可以阅读有关 batch()定义的更多信息，或者直接跳到下一个步骤:..注意:

```
**Definition**: :func:`~dgl.batch` unions a list of :math:`B`
  :class:`~dgl.DGLGraph`\ s and returns a :class:`~dgl.DGLGraph` of batch
  size :math:`B`.

- The union includes all the nodes,
  edges, and their features. The order of nodes, edges, and features are
  preserved.

    - Given that you have :math:`V_i` nodes for graph
      :math:`\mathcal{G}_i`, the node ID :math:`j` in graph
      :math:`\mathcal{G}_i` correspond to node ID
      :math:`j + \sum_{k=1}^{i-1} V_k` in the batched graph.

    - Therefore, performing feature transformation and message passing on
      the batched graph is equivalent to doing those
      on all ``DGLGraph`` constituents in parallel.

- Duplicate references to the same graph are
  treated as deep copies; the nodes, edges, and features are duplicated,
  and mutation on one reference does not affect the other.
- The batched graph keeps track of the meta
  information of the constituents so it can be
  :func:`~dgl.batched_graph.unbatch`\ ed to list of ``DGLGraph``\ s.
```

5.3 步骤 2：具有消息传递 API 的 Tree-LSTM 单元

研究人员提出了两种类型的 Tree-LSTM：Child-Sum Tree-LSTM 和 N-ary Tree-LSTM。在本教程中，您专注于将 Binary Tree-LSTM 应用于二值化的选区树。此应用程序也称为选区树 LSTM。 使用 PyTorch 作为建立网络的后端框架。

在 N 元树 LSTM 中，节点 j 的每个单元都维护一个隐藏的表示 h_j 和一个存储单元 c_j。 单元 j 将输入向量 x_j 和子单元的隐藏表示：h_{jl}，$1 \leqslant l \leqslant N$ 作为输入，然后通过以下方式更新其新的隐藏表示 h_j 和存储单元 c_j：

$$i_j = \sigma\left(W^{(i)}x_j + \sum_{l=1}^{N} U_l^{(i)} h_{jl} + b^{(i)}\right), \quad (1)$$

$$f_{jk} = \sigma\left(W^{(f)}x_j + \sum_{l=1}^{N} U_{kl}^{(f)} h_{jl} + b^{(f)}\right), \quad (2)$$

$$o_j = \sigma\left(W^{(o)}x_j + \sum_{l=1}^{N} U_l^{(o)} h_{jl} + b^{(o)}\right), \quad (3)$$

$$u_j = \tanh\left(W^{(u)}x_j + \sum_{l=1}^{N} U_l^{(u)} h_{jl} + b^{(u)}\right), \quad (4)$$

$$c_j = i_j \odot u_j + \sum_{l=1}^{N} f_{jl} \odot c_{jl}, \quad (5)$$

$$h_j = o_j \cdot \tanh(c_j), \quad (6)$$

它可以分解为三个阶段:message_func、reduce_func 和 apply_node_func。

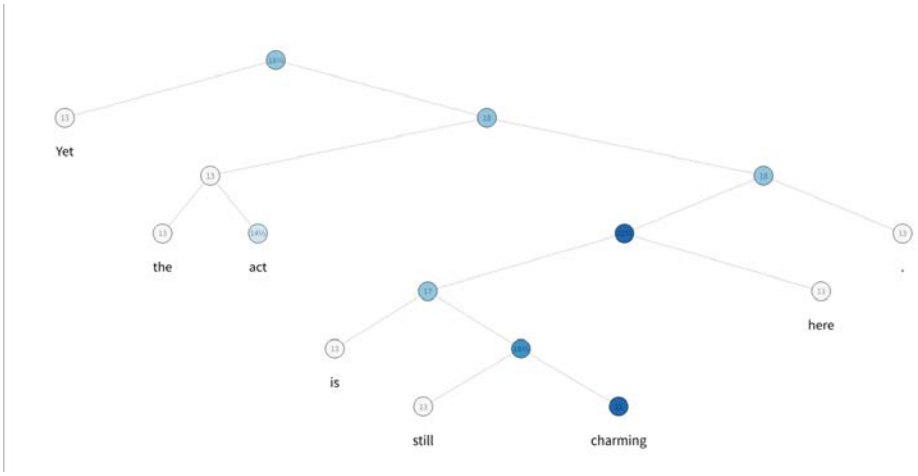
注意：apply_node_func 是以前尚未引入的新节点 UDF。 在 apply_node_func 中，用户指定如何处理节点特征，而不考虑边特征和消息。 在 Tree-LSTM 情况下，必须使用 apply_node_func，因为存在（叶）个节点的传入边为 0，而不会使用 reduce_func 进行更新。

```
import torch as th
import torch.nn as nn

class TreeLSTMCell(nn.Module):
    def __init__(self, x_size, h_size):
        super(TreeLSTMCell, self).__init__()
        self.W_iou = nn.Linear(x_size, 3 * h_size, bias=False)
        self.U_iou = nn.Linear(2 * h_size, 3 * h_size, bias=False)
        self.b_iou = nn.Parameter(th.zeros(1, 3 * h_size))
        self.U_f = nn.Linear(2 * h_size, 2 * h_size)

    def message_func(self, edges):
        return {'h': edges.src['h'], 'c': edges.src['c']}

    def reduce_func(self, nodes):
        # concatenate h_jl for equation (1), (2), (3), (4)
        h_cat = nodes.mailbox['h'].view(nodes.mailbox['h'].size(0), -1)
        # equation (2)
        f = th.sigmoid(self.U_f(h_cat)).view(*nodes.mailbox['h'].size())
        # second term of equation (5)
        c = th.sum(f * nodes.mailbox['c'], 1)
        return {'iou': self.U_iou(h_cat), 'c': c}

    def apply_node_func(self, nodes):
        # equation (1), (3), (4)
        iou = nodes.data['iou'] + self.b_iou
        i, o, u = th.chunk(iou, 3, 1)
        i, o, u = th.sigmoid(i), th.sigmoid(o), th.tanh(u)
        # equation (5)
        c = i * u + nodes.data['c']
        # equation (6)
        h = o * th.tanh(c)
        return {'h' : h, 'c' : c}
```

5.4 步骤 3：定义遍历

定义消息传递函数之后，引导触发它们的正确顺序。这与 GCN 等模型有很大的不同，在 GCN 中，所有节点同时从上游节点提取消息。

对于 tree - lstm，消息从树的叶子开始，向上传播/处理，直到到达根。可视化如下:（这是个动图）



DGL 定义了一个生成器来执行拓扑排序，每一项都是一个张量，用于记录从底层到根的节点。 通过检查以下差异，可以了解并行度：

```
# to heterogenous graph
trv_a_tree = dgl.graph(a_tree.edges())
print('Traversing one tree:')
print(dgl.topological_nodes_generator(trv_a_tree))

# to heterogenous graph
trv_graph = dgl.graph(graph.edges())
print('Traversing many trees at the same time:')
```

输出：

```
Traversing one tree:
(tensor([ 2,  3,  6,  8, 13, 15, 17, 19, 22, 23, 25, 27, 28, 29, 30, 32, 34, 36,
        38, 40, 43, 46, 47, 49, 50, 52, 58, 59, 60, 62, 64, 65, 66, 68, 69, 70]), tensor([ 1,
21, 26, 45, 48, 57, 63, 67]), tensor([24, 44, 56, 61]), tensor([20, 42, 55]), tensor([18,
54]), tensor([16, 53]), tensor([14, 51]), tensor([12, 41]), tensor([11, 39]), tensor([10,
37]), tensor([35]), tensor([33]), tensor([31]), tensor([9]), tensor([7]), tensor([5]),
tensor([4]), tensor([0]))
Traversing many trees at the same time:
(tensor([  2,   3,   6,   8,  13,  15,  17,  19,  22,  23,  25,  27,  28,  29,
         30,  32,  34,  36,  38,  40,  43,  46,  47,  49,  50,  52,  58,  59,
         60,  62,  64,  65,  66,  68,  69,  70,  74,  76,  78,  79,  82,  83,
         85,  88,  90,  92,  93,  95,  96, 100, 102, 103, 105, 109, 110, 112,
        113, 117, 118, 119, 121, 125, 127, 129, 130, 132, 133, 135, 138, 140,
        141, 142, 143, 150, 152, 153, 155, 158, 159, 161, 162, 164, 168, 170,
        171, 174, 175, 178, 179, 182, 184, 185, 187, 189, 190, 191, 192, 195,
        197, 198, 200, 202, 205, 208, 210, 212, 213, 214, 216, 218, 219, 220,
        223, 225, 227, 229, 230, 232, 235, 237, 240, 242, 244, 246, 248, 249,
        251, 253, 255, 256, 257, 259, 262, 263, 267, 269, 270, 271, 272]), tensor([  1,
21,  26,  45,  48,  57,  63,  67,  77,  81,  91,  94, 101, 108,
        111, 116, 128, 131, 139, 151, 157, 160, 169, 173, 177, 183, 188, 196,
```

```
       211, 217, 228, 247, 254, 261, 268]), tensor([ 24,  44,  56,  61,  75,  89,  99,
107, 115, 126, 137, 149, 156, 167,
       181, 186, 194, 209, 215, 226, 245, 252, 266]), tensor([ 20,  42,  55,  73,  87,
124, 136, 154, 180, 207, 224, 243, 250, 265]), tensor([ 18,  54,  86, 123, 134, 148, 176,
206, 222, 241, 264]), tensor([ 16,  53,  84, 122, 172, 204, 239, 260]), tensor([ 14,  51,
80, 120, 166, 203, 238, 258]), tensor([ 12,  41,  72, 114, 165, 201, 236]), tensor([ 11,
39, 106, 163, 199, 234]), tensor([ 10,  37, 104, 147, 193, 233]), tensor([ 35,  98, 146,
231]), tensor([ 33,  97, 145, 221]), tensor([ 31,  71, 144]), tensor([9]), tensor([7]),
tensor([5]), tensor([4]), tensor([0]))
```

调用 prop_nodes()触发消息传递:

```
import dgl.function as fn
import torch as th

trv_graph.ndata['a'] = th.ones(graph.number_of_nodes(), 1)
traversal_order = dgl.topological_nodes_generator(trv_graph)
trv_graph.prop_nodes(traversal_order,
                     message_func=fn.copy_src('a', 'a'),
                     reduce_func=fn.sum('a', 'a'))

# the following is a syntax sugar that does the same
# dgl.prop_nodes_topo(graph)
```

注意：在调用 prop_nodes()之前，预先指定 message_func 和 reduce_func。在本例中，您可以看到内置的从源复制和 sum 函数作为消息函数，以及用于演示的 reduce 函数。

5.5 把它们放在一起

下面是指定 Tree-LSTM 类的完整代码。

```
class TreeLSTM(nn.Module):
    def __init__(self,
                 num_vocabs,
                 x_size,
                 h_size,
                 num_classes,
                 dropout,
                 pretrained_emb=None):
        super(TreeLSTM, self).__init__()
        self.x_size = x_size
        self.embedding = nn.Embedding(num_vocabs, x_size)
        if pretrained_emb is not None:
            print('Using glove')
            self.embedding.weight.data.copy_(pretrained_emb)
            self.embedding.weight.requires_grad = True
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(h_size, num_classes)
```

```
        self.cell = TreeLSTMCell(x_size, h_size)

    def forward(self, batch, h, c):
        """Compute tree-lstm prediction given a batch.

        Parameters
        ----------
        batch : dgl.data.SSTBatch
            The data batch.
        h : Tensor
            Initial hidden state.
        c : Tensor
            Initial cell state.

        Returns
        -------
        logits : Tensor
            The prediction of each node.
        """
        g = batch.graph
        # to heterogenous graph
        g = dgl.graph(g.edges())
        # feed embedding
        embeds = self.embedding(batch.wordid * batch.mask)
        g.ndata['iou'] = self.cell.W_iou(self.dropout(embeds)) *
batch.mask.float().unsqueeze(-1)
        g.ndata['h'] = h
        g.ndata['c'] = c
        # propagate
        dgl.prop_nodes_topo(g,
                            message_func=self.cell.message_func,
                            reduce_func=self.cell.reduce_func,
                            apply_node_func=self.cell.apply_node_func)
        # compute logits
        h = self.dropout(g.ndata.pop('h'))
        logits = self.linear(h)
        return logits
```

5.6 主循环

最后，您可以在 PyTorch 中编写训练范例。

```
from torch.utils.data import DataLoader
import torch.nn.functional as F

device = th.device('cpu')
```

```
# hyper parameters
x_size = 256
h_size = 256
dropout = 0.5
lr = 0.05
weight_decay = 1e-4
epochs = 10

# create the model
model = TreeLSTM(trainset.num_vocabs,
                 x_size,
                 h_size,
                 trainset.num_classes,
                 dropout)
print(model)

# create the optimizer
optimizer = th.optim.Adagrad(model.parameters(),
                             lr=lr,
                             weight_decay=weight_decay)

def batcher(dev):
    def batcher_dev(batch):
        batch_trees = dgl.batch(batch)
        return SSTBatch(graph=batch_trees,
                        mask=batch_trees.ndata['mask'].to(device),
                        wordid=batch_trees.ndata['x'].to(device),
                        label=batch_trees.ndata['y'].to(device))
    return batcher_dev

train_loader = DataLoader(dataset=tiny_sst,
                          batch_size=5,
                          collate_fn=batcher(device),
                          shuffle=False,
                          num_workers=0)

# training loop
for epoch in range(epochs):
    for step, batch in enumerate(train_loader):
        g = batch.graph
        n = g.number_of_nodes()
        h = th.zeros((n, h_size))
        c = th.zeros((n, h_size))
        logits = model(batch, h, c)
```

```
        logp = F.log_softmax(logits, 1)
        loss = F.nll_loss(logp, batch.label, reduction='sum')
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        pred = th.argmax(logits, 1)
        acc = float(th.sum(th.eq(batch.label, pred))) / len(batch.label)
        print("Epoch {:05d} | Step {:05d} | Loss {:.4f} | Acc {:.4f} |".format(
            epoch, step, loss.item(), acc))
```

输出：

```
TreeLSTM(
  (embedding): Embedding(19536, 256)
  (dropout): Dropout(p=0.5, inplace=False)
  (linear): Linear(in_features=256, out_features=5, bias=True)
  (cell): TreeLSTMCell(
    (W_iou): Linear(in_features=256, out_features=768, bias=False)
    (U_iou): Linear(in_features=512, out_features=768, bias=False)
    (U_f): Linear(in_features=512, out_features=512, bias=True)
  )
)
Epoch 00000 | Step 00000 | Loss 428.8259 | Acc 0.3370 |
Epoch 00001 | Step 00000 | Loss 250.2482 | Acc 0.7289 |
Epoch 00002 | Step 00000 | Loss 480.4140 | Acc 0.5751 |
Epoch 00003 | Step 00000 | Loss 427.4128 | Acc 0.8059 |
Epoch 00004 | Step 00000 | Loss 448.3498 | Acc 0.6190 |
Epoch 00005 | Step 00000 | Loss 179.1408 | Acc 0.8425 |
Epoch 00006 | Step 00000 | Loss 99.7904 | Acc 0.8755 |
Epoch 00007 | Step 00000 | Loss 81.6280 | Acc 0.9121 |
Epoch 00008 | Step 00000 | Loss 56.6379 | Acc 0.9414 |
Epoch 00009 | Step 00000 | Loss 98.5619 | Acc 0.8974 |
```

要在使用不同设置(比如 CPU 或 GPU)的完整数据集上训练模型，请参考 PyTorch 示例。

https://github.com/dmlc/dgl/tree/master/examples/pytorch/tree_lstm

还有 Child-Sum Tree-LSTM 的实现。

6. 图生成模型

Author: Mufei Li, Lingfan Yu, Zheng Zhang

在本教程中，您将学习如何一次训练和生成一个图。您还将探索图嵌入操作中的并行性，这是一个基本的构建模块。本教程以一个简单的优化结束，该优化通过对图进行批处理实现了两倍的速度。

较早的教程显示了如何嵌入图或节点使您能够完成诸如 semi-supervised classification for nodes or sentiment analysis 之类的任务。 预测图的未来演变并迭代执行分析是否会很有趣？

为了解决图的演变，您可以生成各种图样本。 换句话说，您需要图的生成

模型。 除学习节点和边特征外，您还需要对任意图的分布进行建模。 虽然一般的生成模型可以显式和隐式地对密度函数进行建模，并且可以一次或顺序生成样本，但是在这里，您只关注用于顺序生成的显式生成模型。 典型的应用包括药物或材料的发现，化学过程或蛋白质组成。

6.1 介绍

在 Deep graph Library (DGL)中，改变一个图的基本动作就是 add_nodes 和 add_edges。也就是说，如果你画一个有三个节点的圆，

您可以按照以下方式编写代码。

```
import dgl


g = dgl.DGLGraph()
g.add_nodes(1)              # Add node 0
g.add_nodes(1)              # Add node 1

# Edges in DGLGraph are directed by default.
# For undirected edges, add edges for both directions.
g.add_edges([1, 0], [0, 1]) # Add edges (1, 0), (0, 1)
g.add_nodes(1)              # Add node 2
g.add_edges([2, 1], [1, 2]) # Add edges (2, 1), (1, 2)
g.add_edges([2, 0], [0, 2]) # Add edges (2, 0), (0, 2)
```

输出：

```
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/base.py:45: DGLWarning: Recommend creating graphs by `dgl.graph(data)` instead of `dgl.DGLGraph(data)`.
  return warnings.warn(message, category=category, stacklevel=1)
```

真实世界的图表要复杂得多。图有许多族，具有不同的大小、拓扑、节点类型、边类型和多图的可能性。此外，同样的图可以以许多不同的顺序生成。无论如何，生成过程需要几个步骤。

- 编码一个变化的图。
- 随机执行操作。
- 如果你正在训练，收集错误信号并优化模型参数。

在实现方面，另一个重要的方面是速度。如果生成一个图基本上是一个顺序的过程，那么如何并行化计算呢?

注意：当然，这并不一定是一个硬性约束。子图可以并行构建，然后进行组装。但在本教程中，我们将限制自己的顺序过程。

6.2 DGMG：主要流程

在本教程中，您将使用 Deep Generative Models of Graphs (DGMG)来使用 DGL 实现图生成模型。它的算法框架很通用，但并行化也很有挑战性。

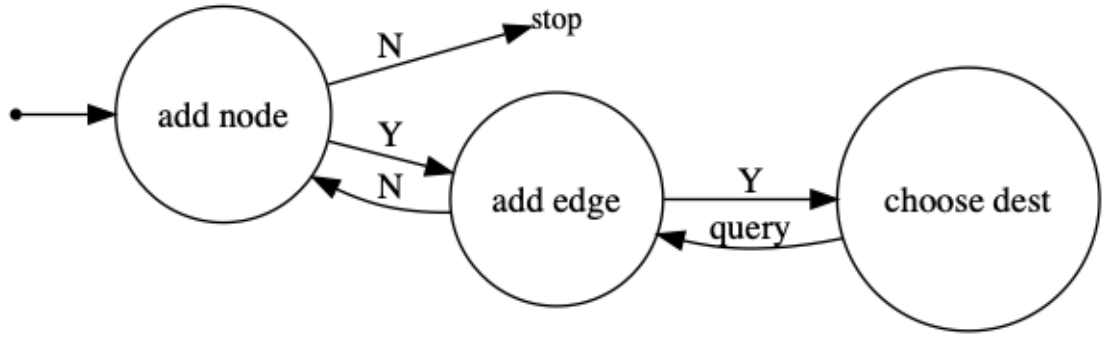
注意：虽然 DGMG 可以处理具有类型化节点、类型化边和多图的复杂图，但这里使用它的简化版本来生成图拓扑。

DGMG 通过遵循状态机来生成图，该状态机基本上是一个两级循环。 一次生成一个节点，然后一次将其连接到现有节点的子集。 这类似于语言建模。 生

成过程是一种迭代过程，它以到目前为止生成的序列为条件，一次生成一个单词，一个字符或一个句子。

在每一个时间步骤，你要么：

- 向图中添加一个新节点
- 选择两个现有节点并在它们之间添加一条边



Python 代码如下所示。 实际上，这正是在 DGL 中实现 DGMG 推理的方式。

```
def forward_inference(self):
    stop = self.add_node_and_update()
    while (not stop) and (self.g.number_of_nodes() < self.v_max + 1):
        num_trials = 0
        to_add_edge = self.add_edge_or_not()
        while to_add_edge and (num_trials < self.g.number_of_nodes() - 1):
            self.choose_dest_and_update()
            num_trials += 1
            to_add_edge = self.add_edge_or_not()
        stop = self.add_node_and_update()

    return self.g
```

假设您具有用于生成节点 10-20 的循环的预训练模型。它如何在推理过程中即时生成一个循环？ 使用下面的代码使用自己的模型创建动画。

```
import torch
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import networkx as nx
from copy import deepcopy

if __name__ == '__main__':
    # pre-trained model saved with path ./model.pth
    model = torch.load('./model.pth')
    model.eval()
    g = model()

    src_list = g.edges()[1]
    dest_list = g.edges()[0]
```

```
    evolution = []

    nx_g = nx.Graph()
    evolution.append(deepcopy(nx_g))

    for i in range(0, len(src_list), 2):
        src = src_list[i].item()
        dest = dest_list[i].item()
        if src not in nx_g.nodes():
            nx_g.add_node(src)
            evolution.append(deepcopy(nx_g))
        if dest not in nx_g.nodes():
            nx_g.add_node(dest)
            evolution.append(deepcopy(nx_g))
        nx_g.add_edges_from([(src, dest), (dest, src)])
        evolution.append(deepcopy(nx_g))

    def animate(i):
        ax.cla()
        g_t = evolution[i]
        nx.draw_circular(g_t, with_labels=True, ax=ax,
                         node_color=['#FEBD69'] * g_t.number_of_nodes())

    fig, ax = plt.subplots()
    ani = animation.FuncAnimation(fig, animate,
                                  frames=len(evolution),
                                  interval=600)
```

6.3 DGMG：优化目标

与语言建模类似，DGMG 通过行为克隆或教师强迫来训练模型。 假设每个图都有一个生成它的预言动作 a_1，…，a_T 序列。 模型要做的是遵循这些动作，计算这些动作序列的联合概率，并使它们最大化。

By chain rule, the probability of taking $a_1, \cdots, a_T$ is:

$$p(a_1, \cdots, a_T) = p(a_1)p(a_2|a_1)\cdots p(a_T|a_1, \cdots, a_{T-1}).$$

The optimization objective is then simply the typical MLE loss:

$$-\log p(a_1, \cdots, a_T) = -\sum_{t=1}^{T} \log p(a_t|a_1, \cdots, a_{t-1}).$$

```
def forward_train(self, actions):
    """
    - actions: list
        - Contains a_1, ..., a_T described above
```

```
    - self.prepare_for_train()
        - Initializes self.action_step to be 0, which will get
          incremented by 1 every time it is called.
        - Initializes objects recording log p(a_t|a_1,...a_{t-1})

    Returns
    -------
    - self.get_log_prob(): log p(a_1, ..., a_T)
    """
    self.prepare_for_train()

    stop = self.add_node_and_update(a=actions[self.action_step])
    while not stop:
        to_add_edge = self.add_edge_or_not(a=actions[self.action_step])
        while to_add_edge:
            self.choose_dest_and_update(a=actions[self.action_step])
            to_add_edge = self.add_edge_or_not(a=actions[self.action_step])
        stop = self.add_node_and_update(a=actions[self.action_step])

    return self.get_log_prob()
```

forward_train 和 forward_inference 的关键区别在于，训练过程将 oracle 动作作为输入，并返回 log probability 来评估损失。

6.4 DGMG：实现

6.4.1 DGMG 类

在下面，您可以找到该模型的框架代码。 您逐步浏览每个功能的详细信息。

```
import torch.nn as nn


class DGMGSkeleton(nn.Module):
    def __init__(self, v_max):
        """
        Parameters
        ----------
        v_max: int
            Max number of nodes considered
        """
        super(DGMGSkeleton, self).__init__()

        # Graph configuration
        self.v_max = v_max

    def add_node_and_update(self, a=None):
```

```
        """Decide if to add a new node.
        If a new node should be added, update the graph."""
        return NotImplementedError

    def add_edge_or_not(self, a=None):
        """Decide if a new edge should be added."""
        return NotImplementedError

    def choose_dest_and_update(self, a=None):
        """Choose destination and connect it to the latest node.
        Add edges for both directions and update the graph."""
        return NotImplementedError

    def forward_train(self, actions):
        """Forward at training time. It records the probability
        of generating a ground truth graph following the actions."""
        return NotImplementedError

    def forward_inference(self):
        """Forward at inference time.
        It generates graphs on the fly."""
        return NotImplementedError

    def forward(self, actions=None):
        # The graph you will work on
        self.g = dgl.DGLGraph()

        # If there are some features for nodes and edges,
        # zero tensors will be set for those of new nodes and edges.
        self.g.set_n_initializer(dgl.frame.zero_initializer)
        self.g.set_e_initializer(dgl.frame.zero_initializer)

        if self.training:
            return self.forward_train(actions=actions)
        else:
            return self.forward_inference()
```

6.4.2 编码动态图

从概率分布中采样所有生成图的动作。 为此，可以将结构化数据（即图）投影到欧几里得空间上。 挑战在于，随着图的变化，需要重复这种称为嵌入的过程。

图嵌入：

令 G =（V，E）是任意图。 每个节点 v 具有嵌入向量 $h_v \in R^n$。 类似地，该图具有嵌入矢量 $h_G \in R^k$。 通常，k> n，因为图比单个节点包含更多的信息。

图嵌入是线性变换下节点嵌入的加权总和：

$$\mathbf{h}_G = \sum_{v \in V} \text{Sigmoid}(g_m(\mathbf{h}_v)) f_m(\mathbf{h}_v),$$

第一项，Sigmoid(gm(hv))，计算一个门控函数，可以被认为是整体图嵌入在每个节点的参与量。第二项 $f_m:R^n \rightarrow R^k$ 将节点嵌入映射到图嵌入空间。

将图嵌入实现为 GraphEmbed 类。

```
mport torch


class GraphEmbed(nn.Module):
    def __init__(self, node_hidden_size):
        super(GraphEmbed, self).__init__()

        # Setting from the paper
        self.graph_hidden_size = 2 * node_hidden_size

        # Embed graphs
        self.node_gating = nn.Sequential(
            nn.Linear(node_hidden_size, 1),
            nn.Sigmoid()
        )
        self.node_to_graph = nn.Linear(node_hidden_size,
                                       self.graph_hidden_size)

    def forward(self, g):
        if g.number_of_nodes() == 0:
            return torch.zeros(1, self.graph_hidden_size)
        else:
            # Node features are stored as hv in ndata.
            hvs = g.ndata['hv']
            return (self.node_gating(hvs) *
                    self.node_to_graph(hvs)).sum(0, keepdim=True)
```

通过图传播更新节点嵌入：

DGMG 中更新节点嵌入的机制与图卷积网络类似。对于图中的节点 v，它的邻居 u 向它发送一条消息

$$\mathbf{m}_{u \rightarrow v} = \mathbf{W}_m \text{concat}([\mathbf{h}_v, \mathbf{h}_u, \mathbf{x}_{u,v}]) + \mathbf{b}_m,$$

其中 $x_{u,v}$ 是 u 与 v 之间嵌入的边。

在接收到来自所有邻居的消息后，v 使用节点激活向量对它们进行汇总：

$$\mathbf{a}_v = \sum_{u:(u,v) \in E} \mathbf{m}_{u \rightarrow v}$$

并利用这些信息更新自身特征:

$$\mathbf{h}_v' = \mathbf{GRU}(\mathbf{h}_v, \mathbf{a}_v).$$

对所有节点同步执行上述所有操作，称为一轮图传播。执行的图传播轮数越多，消息在图中传播的距离就越长。

使用 DGL，可以使用 g.update_all 实现图传播。这里的消息符号可能有点令人困惑。研究者可以将 $m_{u\to v}$ 称为 message，但是下面的 message 功能仅通过 concat($[h_u,x_{u,v}]$)。

为了提高效率，在所有边同时执行操作 $W_m$concat($[h_v,h_u,x_{u,v}]$)+b_m。

```
import torch.nn.functional as F
from torch.distributions import Bernoulli

def bernoulli_action_log_prob(logit, action):
    """Calculate the log p of an action with respect to a Bernoulli
    distribution. Use logit rather than prob for numerical stability."""
    if action == 0:
        return F.logsigmoid(-logit)
    else:
        return F.logsigmoid(logit)

class AddNode(nn.Module):
    def __init__(self, graph_embed_func, node_hidden_size):
        super(AddNode, self).__init__()

        self.graph_op = {'embed': graph_embed_func}

        self.stop = 1
        self.add_node = nn.Linear(graph_embed_func.graph_hidden_size, 1)

        # If to add a node, initialize its hv
        self.node_type_embed = nn.Embedding(1, node_hidden_size)
        self.initialize_hv = nn.Linear(node_hidden_size + \
                                       graph_embed_func.graph_hidden_size,
                                       node_hidden_size)

        self.init_node_activation = torch.zeros(1, 2 * node_hidden_size)

    def _initialize_node_repr(self, g, node_type, graph_embed):
        """Whenver a node is added, initialize its representation."""
        num_nodes = g.number_of_nodes()
        hv_init = self.initialize_hv(
            torch.cat([
                self.node_type_embed(torch.LongTensor([node_type])),
```

```
                graph_embed], dim=1))
        g.nodes[num_nodes - 1].data['hv'] = hv_init
        g.nodes[num_nodes - 1].data['a'] = self.init_node_activation

    def prepare_training(self):
        self.log_prob = []

    def forward(self, g, action=None):
        graph_embed = self.graph_op['embed'](g)

        logit = self.add_node(graph_embed)
        prob = torch.sigmoid(logit)

        if not self.training:
            action = Bernoulli(prob).sample().item()
        stop = bool(action == self.stop)

        if not stop:
            g.add_nodes(1)
            self._initialize_node_repr(g, action, graph_embed)

        if self.training:
            sample_log_prob = bernoulli_action_log_prob(logit, action)

            self.log_prob.append(sample_log_prob)

        return stop
```

6.4.3 动作

从使用神经网络参数化的分布中采样所有动作，然后依次进行。

动作 1:添加节点:

给定图嵌入向量 h_G，请评估:

$$\text{Sigmoid}(\mathbf{W}_{\text{add node}}\mathbf{h}_G + b_{\text{add node}}),$$

然后用于参数化伯努利分布，以决定是否添加新节点。

如果要添加一个新节点，初始化其特征

$$\mathbf{W}_{\text{init}}\,\text{concat}([\mathbf{h}_{\text{init}}, \mathbf{h}_G]) + \mathbf{b}_{\text{init}},$$

h_{init} 是用于无类型节点的可学习嵌入模块。

```
mport torch.nn.functional as F
from torch.distributions import Bernoulli
```

```
def bernoulli_action_log_prob(logit, action):
    """Calculate the log p of an action with respect to a Bernoulli
    distribution. Use logit rather than prob for numerical stability."""
    if action == 0:
        return F.logsigmoid(-logit)
    else:
        return F.logsigmoid(logit)

class AddNode(nn.Module):
    def __init__(self, graph_embed_func, node_hidden_size):
        super(AddNode, self).__init__()

        self.graph_op = {'embed': graph_embed_func}

        self.stop = 1
        self.add_node = nn.Linear(graph_embed_func.graph_hidden_size, 1)

        # If to add a node, initialize its hv
        self.node_type_embed = nn.Embedding(1, node_hidden_size)
        self.initialize_hv = nn.Linear(node_hidden_size + \
                                       graph_embed_func.graph_hidden_size,
                                       node_hidden_size)

        self.init_node_activation = torch.zeros(1, 2 * node_hidden_size)

    def _initialize_node_repr(self, g, node_type, graph_embed):
        """Whenver a node is added, initialize its representation."""
        num_nodes = g.number_of_nodes()
        hv_init = self.initialize_hv(
            torch.cat([
                self.node_type_embed(torch.LongTensor([node_type])),
                graph_embed], dim=1))
        g.nodes[num_nodes - 1].data['hv'] = hv_init
        g.nodes[num_nodes - 1].data['a'] = self.init_node_activation

    def prepare_training(self):
        self.log_prob = []

    def forward(self, g, action=None):
        graph_embed = self.graph_op['embed'](g)

        logit = self.add_node(graph_embed)
        prob = torch.sigmoid(logit)
```

```
        if not self.training:
            action = Bernoulli(prob).sample().item()
        stop = bool(action == self.stop)

        if not stop:
            g.add_nodes(1)
            self._initialize_node_repr(g, action, graph_embed)

        if self.training:
            sample_log_prob = bernoulli_action_log_prob(logit, action)

            self.log_prob.append(sample_log_prob)

        return stop
```

动作 2:添加边:

给定图嵌入向量 h_G 和最新节点 v 的节点嵌入向量 h_v，您可以评估:

$$\text{Sigmoid}(\mathbf{W}_{\text{add edge}}\text{concat}([\mathbf{h}_G, \mathbf{h}_v]) + b_{\text{add edge}}),$$

然后将其用于参数化 Bernoulli 分布，以决定是否从 v 开始添加新边。

```
class AddEdge(nn.Module):
    def __init__(self, graph_embed_func, node_hidden_size):
        super(AddEdge, self).__init__()

        self.graph_op = {'embed': graph_embed_func}
        self.add_edge = nn.Linear(graph_embed_func.graph_hidden_size + \
                                  node_hidden_size, 1)

    def prepare_training(self):
        self.log_prob = []

    def forward(self, g, action=None):
        graph_embed = self.graph_op['embed'](g)
        src_embed = g.nodes[g.number_of_nodes() - 1].data['hv']

        logit = self.add_edge(torch.cat(
            [graph_embed, src_embed], dim=1))
        prob = torch.sigmoid(logit)

        if self.training:
            sample_log_prob = bernoulli_action_log_prob(logit, action)
            self.log_prob.append(sample_log_prob)
        else:
            action = Bernoulli(prob).sample().item()
```

```
        to_add_edge = bool(action == 0)
        return to_add_edge
```

动作 3 选择目的地：

当操作 2 返回 True 时，请为最新节点 v 选择目标。

对于每个可能的目标 u∈{0，…，v-1}，选择它的概率为：

$$\frac{\exp(\mathbf{W}_{\text{dest}}\,\text{concat}([\mathbf{h}_u, \mathbf{h}_v]) + \mathbf{b}_{\text{dest}})}{\sum_{i=0}^{v-1} \exp(\mathbf{W}_{\text{dest}}\,\text{concat}([\mathbf{h}_i, \mathbf{h}_v]) + \mathbf{b}_{\text{dest}})}$$

```
from torch.distributions import Categorical

class ChooseDestAndUpdate(nn.Module):
    def __init__(self, graph_prop_func, node_hidden_size):
        super(ChooseDestAndUpdate, self).__init__()

        self.graph_op = {'prop': graph_prop_func}
        self.choose_dest = nn.Linear(2 * node_hidden_size, 1)

    def _initialize_edge_repr(self, g, src_list, dest_list):
        # For untyped edges, only add 1 to indicate its existence.
        # For multiple edge types, use a one-hot representation
        # or an embedding module.
        edge_repr = torch.ones(len(src_list), 1)
        g.edges[src_list, dest_list].data['he'] = edge_repr

    def prepare_training(self):
        self.log_prob = []

    def forward(self, g, dest):
        src = g.number_of_nodes() - 1
        possible_dests = range(src)

        src_embed_expand = g.nodes[src].data['hv'].expand(src, -1)
        possible_dests_embed = g.nodes[possible_dests].data['hv']

        dests_scores = self.choose_dest(
            torch.cat([possible_dests_embed,
                       src_embed_expand], dim=1)).view(1, -1)
        dests_probs = F.softmax(dests_scores, dim=1)

        if not self.training:
            dest = Categorical(dests_probs).sample().item()
```

```
        if not g.has_edge_between(src, dest):
            # For undirected graphs, add edges for both directions
            # so that you can perform graph propagation.
            src_list = [src, dest]
            dest_list = [dest, src]

            g.add_edges(src_list, dest_list)
            self._initialize_edge_repr(g, src_list, dest_list)

            self.graph_op['prop'](g)

        if self.training:
            if dests_probs.nelement() > 1:
                self.log_prob.append(
                    F.log_softmax(dests_scores, dim=1)[:, dest: dest + 1])
```

6.4.4 将它们放在一起

现在，您可以准备完整的模型类实现了。

```
class DGMG(DGMGSkeleton):
    def __init__(self, v_max, node_hidden_size,
                 num_prop_rounds):
        super(DGMG, self).__init__(v_max)

        # Graph embedding module
        self.graph_embed = GraphEmbed(node_hidden_size)

        # Graph propagation module
        self.graph_prop = GraphProp(num_prop_rounds,
                                    node_hidden_size)

        # Actions
        self.add_node_agent = AddNode(
            self.graph_embed, node_hidden_size)
        self.add_edge_agent = AddEdge(
            self.graph_embed, node_hidden_size)
        self.choose_dest_agent = ChooseDestAndUpdate(
            self.graph_prop, node_hidden_size)

        # Forward functions
        self.forward_train = partial(forward_train, self=self)
        self.forward_inference = partial(forward_inference, self=self)

    @property
    def action_step(self):
```

```
        old_step_count = self.step_count
        self.step_count += 1

        return old_step_count

    def prepare_for_train(self):
        self.step_count = 0

        self.add_node_agent.prepare_training()
        self.add_edge_agent.prepare_training()
        self.choose_dest_agent.prepare_training()

    def add_node_and_update(self, a=None):
        """Decide if to add a new node.
        If a new node should be added, update the graph."""

        return self.add_node_agent(self.g, a)

    def add_edge_or_not(self, a=None):
        """Decide if a new edge should be added."""

        return self.add_edge_agent(self.g, a)

    def choose_dest_and_update(self, a=None):
        """Choose destination and connect it to the latest node.
        Add edges for both directions and update the graph."""

        self.choose_dest_agent(self.g, a)

    def get_log_prob(self):
        add_node_log_p = torch.cat(self.add_node_agent.log_prob).sum()
        add_edge_log_p = torch.cat(self.add_edge_agent.log_prob).sum()
        choose_dest_log_p = torch.cat(self.choose_dest_agent.log_prob).sum()
        return add_node_log_p + add_edge_log_p + choose_dest_log_p
```

下面是一个动画，其中在前 400 批次的每 10 批次训练之后，动态生成图。 您可以看到模型随着时间的推移如何改进并开始生成周期。

https://user-images.githubusercontent.com/19576924/48929291-60fe3880-ef22-11e8-832a-fbe56656559a.gif

对于生成模型，可以通过在运行中生成的图中检查有效图的百分比来评估性能。

```
import torch.utils.model_zoo as model_zoo

# Download a pre-trained model state dict for generating cycles with 10-20 nodes.
state_dict = model_zoo.load_url('https://data.dgl.ai/model/dgmg_cycles-5a0c40be.pth')
```

```
model = DGMG(v_max=20, node_hidden_size=16, num_prop_rounds=2)
model.load_state_dict(state_dict)
model.eval()

def is_valid(g):
    # Check if g is a cycle having 10-20 nodes.
    def _get_previous(i, v_max):
        if i == 0:
            return v_max
        else:
            return i - 1

    def _get_next(i, v_max):
        if i == v_max:
            return 0
        else:
            return i + 1

    size = g.number_of_nodes()

    if size < 10 or size > 20:
        return False

    for node in range(size):
        neighbors = g.successors(node)

        if len(neighbors) != 2:
            return False

        if _get_previous(node, size - 1) not in neighbors:
            return False

        if _get_next(node, size - 1) not in neighbors:
            return False

    return True

num_valid = 0
for i in range(100):
    g = model()
    num_valid += is_valid(g)

del model
print('Among 100 graphs generated, {}% are valid.'.format(num_valid))
```

输出：

```
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/base.py:45: DGLWarning: DGLGraph.has_edge_between is deprecated. Please use DGLGraph.has_edges_between
  return warnings.warn(message, category=category, stacklevel=1)
Among 100 graphs generated, 97% are valid.
```

有关完整的实现，请参见 DGL DGMG 示例。

https://github.com/dmlc/dgl/tree/master/examples/pytorch/dgmg

7. 胶囊网络

Author: Jinjing Zhou, Jake Zhao, Zheng Zhang, Jinyang Li

在本教程中，您将学习如何从图的角度描述较为经典的模型之一。这种方法提供了一个不同的视角。本教程描述了如何为胶囊网络实现胶囊模型。

http://xxx.itp.ac.cn/pdf/1710.09829.pdf

7.1 胶囊的主要思想

胶囊模型提供了两个关键思想：丰富的表示形式和动态路由。

丰富的表示形式 - 在经典的卷积网络中，标量值表示给定特征的激活。 相比之下，胶囊输出矢量。 向量的长度代表特征存在的可能性。 向量的方向代表特征的各种属性（例如姿势，变形，纹理等）。

动态路由 - 一个胶囊的输出被发送到上层的某些父包，这是基于胶囊的预测与父包的一致程度。这种按协议动态路由概括了 max-pooling 的静态路由。

在训练期间，路由是迭代地完成的。每次迭代都根据观察到的协议调整胶囊之间的路由权重。这是一种类似于 k 均值算法或竞争学习的方式。

在本教程中，您将看到如何将胶囊的动态路由算法自然地表达为图算法。该实现改编自 Cedric Chee，只替换了路由层。这个版本达到了类似的速度和精度。

7.2 模型实现

7.2.1 步骤 1：设定和图初始化

胶囊的两层之间的连通性形成有向的二分图，如下图所示。（图片加载不出来）。

每个节点 j 与特征 v_j 相关联，代表其胶囊的输出。 每个边与特征 b_{ij} 和$\hat{u}_{j|i}$相关联。 b_{ij} 确定路由权重，并且$\hat{u}_{j|i}$表示胶囊 i 的预测 j。

这是我们设置图并初始化节点和边特征的方法。

```
import torch.nn as nn
import torch as th
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
import dgl
```

```
def init_graph(in_nodes, out_nodes, f_size):
    u = np.repeat(np.arange(in_nodes), out_nodes)
    v = np.tile(np.arange(in_nodes, in_nodes + out_nodes), in_nodes)
    g = dgl.DGLGraph((u, v))
    # init states
    g.ndata['v'] = th.zeros(in_nodes + out_nodes, f_size)
    g.edata['b'] = th.zeros(in_nodes * out_nodes, 1)
    return g
```

7.2.2 步骤 2：定义消息传递功能

这是 Capsule 路由算法的伪代码。（图片又挂了。。）

通过以下步骤 在class *DGLRoutingLayer*中实现伪代码行4-7 ：

1. 计算Capsule系数。
   Coefficients are the softmax over all out-edge of in-capsules.$\mathbf{c}_{i,j} = \mathrm{softmax}(\mathbf{b}_{i,j})$
2. 计算所有Capsule中的加权总和。
   Output of a capsule is equal to the weighted sum of its in-capsules $s_j = \sum_i c_{ij}\hat{u}_{j|i}$
3. 壁球输出。
   Squash the length of a Capsule’ s output vector to range (0,1), so it can represent the probability (of some feature being present).
   $v_j = \mathrm{squash}(s_j) = \frac{||s_j||^2}{1+||s_j||^2}\frac{s_j}{||s_j||}$
4. 通过协议量更新权重。
   The scalar product $\hat{u}_{j|i} \cdot v_j$ can be considered as how well capsule i agrees with j. It is used to update $b_{ij} = b_{ij} + \hat{u}_{j|i} \cdot v_j$

```
import dgl.function as fn

class DGLRoutingLayer(nn.Module):
    def __init__(self, in_nodes, out_nodes, f_size):
        super(DGLRoutingLayer, self).__init__()
        self.g = init_graph(in_nodes, out_nodes, f_size)
        self.in_nodes = in_nodes
        self.out_nodes = out_nodes
        self.in_indx = list(range(in_nodes))
        self.out_indx = list(range(in_nodes, in_nodes + out_nodes))

    def forward(self, u_hat, routing_num=1):
        self.g.edata['u_hat'] = u_hat

        for r in range(routing_num):
            # step 1 (line 4): normalize over out edges
            edges_b = self.g.edata['b'].view(self.in_nodes, self.out_nodes)
            self.g.edata['c'] = F.softmax(edges_b, dim=1).view(-1, 1)
            self.g.edata['c u_hat'] = self.g.edata['c'] * self.g.edata['u_hat']
```

```
            # Execute step 1 & 2
            self.g.update_all(fn.copy_e('c u_hat', 'm'), fn.sum('m', 's'))

            # step 3 (line 6)
            self.g.nodes[self.out_indx].data['v'] =
self.squash(self.g.nodes[self.out_indx].data['s'], dim=1)

            # step 4 (line 7)
            v = th.cat([self.g.nodes[self.out_indx].data['v']] * self.in_nodes, dim=0)
            self.g.edata['b'] = self.g.edata['b'] + (self.g.edata['u_hat'] *
v).sum(dim=1, keepdim=True)

    @staticmethod
    def squash(s, dim=1):
        sq = th.sum(s ** 2, dim=dim, keepdim=True)
        s_norm = th.sqrt(sq)
        s = (sq / (1.0 + sq)) * (s / s_norm)
        return s
```

7.2.3 步骤 3：测试

制作一个简单的 20x10 胶囊层。

```
in_nodes = 20
out_nodes = 10
f_size = 4
u_hat = th.randn(in_nodes * out_nodes, f_size)
routing = DGLRoutingLayer(in_nodes, out_nodes, f_size)
```

输出：

```
/home/ubuntu/prod-doc/readthedocs.org/user_builds/dgl/checkouts/0.5.x/python/dgl/ba
se.py:45: DGLWarning: Recommend creating graphs by `dgl.graph(data)` instead of
`dgl.DGLGraph(data)`.
  return warnings.warn(message, category=category, stacklevel=1)
```

您可以通过监视耦合系数的熵来可视化胶囊网络的行为。 它们应该从高处开始然后下降，因为权重逐渐集中在较少的边上。

```
entropy_list = []
dist_list = []

for i in range(10):
    routing(u_hat)
    dist_matrix = routing.g.edata['c'].view(in_nodes, out_nodes)
    entropy = (-dist_matrix * th.log(dist_matrix)).sum(dim=1)
    entropy_list.append(entropy.data.numpy())
    dist_list.append(dist_matrix.data.numpy())

stds = np.std(entropy_list, axis=1)
```

```
means = np.mean(entropy_list, axis=1)
plt.errorbar(np.arange(len(entropy_list)), means, stds, marker='o')
plt.ylabel("Entropy of Weight Distribution")
plt.xlabel("Number of Routing")
plt.xticks(np.arange(len(entropy_list)))
plt.close()
```

另外，我们还可以观察直方图的演变。

```
import seaborn as sns
import matplotlib.animation as animation

fig = plt.figure(dpi=150)
fig.clf()
ax = fig.subplots()


def dist_animate(i):
    ax.cla()
    sns.distplot(dist_list[i].reshape(-1), kde=False, ax=ax)
    ax.set_xlabel("Weight Distribution Histogram")
    ax.set_title("Routing: %d" % (i))


ani = animation.FuncAnimation(fig, dist_animate, frames=len(entropy_list),
interval=500)
plt.close()
```

图片地址：动图

https://github.com/VoVAllen/DGL_Capsule/raw/master/routing_dist.gif

你可以监视低能级的胶囊是如何逐渐附着到高能级的胶囊上的。

```
import networkx as nx
from networkx.algorithms import bipartite

g = routing.g.to_networkx()
X, Y = bipartite.sets(g)
height_in = 10
height_out = height_in * 0.8
height_in_y = np.linspace(0, height_in, in_nodes)
height_out_y = np.linspace((height_in - height_out) / 2, height_out, out_nodes)
pos = dict()

fig2 = plt.figure(figsize=(8, 3), dpi=150)
fig2.clf()
ax = fig2.subplots()
pos.update((n, (i, 1)) for i, n in zip(height_in_y, X))  # put nodes from X at x=1
pos.update((n, (i, 2)) for i, n in zip(height_out_y, Y))  # put nodes from Y at x=2
```

```
def weight_animate(i):
    ax.cla()
    ax.axis('off')
    ax.set_title("Routing: %d  " % i)
    dm = dist_list[i]
    nx.draw_networkx_nodes(g, pos, nodelist=range(in_nodes), node_color='r',
node_size=100, ax=ax)
    nx.draw_networkx_nodes(g, pos, nodelist=range(in_nodes, in_nodes + out_nodes),
node_color='b', node_size=100, ax=ax)
    for edge in g.edges():
        nx.draw_networkx_edges(g, pos, edgelist=[edge], width=dm[edge[0], edge[1] -
in_nodes] * 1.5, ax=ax)


ani2 = animation.FuncAnimation(fig2, weight_animate, frames=len(dist_list),
interval=500)
plt.close()
```

图片地址：动图

https://github.com/VoVAllen/DGL_Capsule/raw/master/routing_vis.gif

该可视化的完整代码在 GitHub 上提供。 在 MNIST 上训练的完整代码也在 GitHub 上。

https://github.com/dmlc/dgl/blob/master/examples/pytorch/capsule/simple_routing.py

https://github.com/dmlc/dgl/tree/tutorial/examples/pytorch/capsule

8. Transformer

Author: Zihao Ye, Jinjing Zhou, Qipeng Guo, Quan Gan, Zheng Zhang

在本教程中，您将学习 Transformer 模型的简化实现。 您可以看到最重要的设计要点的亮点。 例如，只有单头注意力。 完整的代码可以在这里找到。

https://github.com/dmlc/dgl/tree/master/examples/pytorch/transformer

总体结构与研究论文“Annotated Transformer.”中的结构相似。

在研究论文中介绍了 Transformer 模型，以替代 CNN / RNN 体系结构进行序列建模：Attention is All You Need。 它改善了机器翻译以及自然语言推理任务(GPT)的技术水平。 带有大型语料库(BERT)的预训练 Transformer 的最新工作支持它能够学习高质量的语义表示。

Transformer 有趣的部分是其广泛的关注度。 注意力的经典用法来自机器翻译模型，其中输出标记处理所有输入标记。

Transformer 还会在解码器和编码器中应用自注意力。 不论单词在序列中的位置如何，此过程都会迫使彼此相关的单词组合在一起。 这与基于 RNN 的模型不同，在 RNN 的模型中，单词（在源句子中）沿着链条组合在一起，这被认为过于局限。

8.1 Transformer 中的注意力层

在 Transformer 的注意力层中，对于每一个节点，模块将学习为其边分配权重。 对于节点 x_i，$x_j \in R^n$ 的节点对（i，j）（从 i 到 j），它们的连接分数定义如下：

$$
\begin{aligned}
q_j &= W_q \cdot x_j \\
k_i &= W_k \cdot x_i \\
v_i &= W_v \cdot x_i \\
\text{score} &= q_j^T k_i
\end{aligned}
$$

where $W_q, W_k, W_v \in \mathbb{R}^{n \times d_k}$ map the representations x to "query", "key", and "value" space respectively.

这是实现评分功能的其他可能性。点积度量给定查询 q_j 和密钥 k_i 的相似性：如果 j 需要 i 中存储的信息，则假定位置 j（q_j）处的查询向量与位置 ii（$_k$i）处的密钥向量接近。

然后将分数被用于计算经过边权重标准化的传入值的总和，该总和存储在 wv 中的。然后将仿射层应用于 wv 以获取输出 o：

$$
\begin{aligned}
w_{ji} &= \frac{\exp\{\text{score}_{ji}\}}{\sum_{(k,i)\in E} \exp\{\text{score}_{ki}\}} \\
\text{wv}_i &= \sum_{(k,i)\in E} w_{ki} v_k \\
o &= W_o \cdot \text{wv}
\end{aligned}
$$

8.2 多头注意力机制

在 Transformer 中，注意力是多头的。一个头很像卷积网络中的通道。多头注意力由多个注意力头组成，其中每个注意力头指的是单个注意力模块。所有头的 $\text{wv}^{(i)}$用仿射层连接并映射到输出 o:

$$
o = W_o \cdot \text{concat}\left(\left[\text{wv}^{(0)}, \text{wv}^{(1)}, \cdots, \text{wv}^{(h)}\right]\right)
$$

下面的代码包装了用于多头注意的必要组件，并提供了两个接口。

- 获取 maps 状态“ x”，以 query，key 和 value，这是以下步骤所必需的（propagate_attention）。
- get_o 将更新后的值映射到输出 o 以进行后处理。

```
class MultiHeadAttention(nn.Module):
    "Multi-Head Attention"
    def __init__(self, h, dim_model):
        "h: number of heads; dim_model: hidden dimension"
        super(MultiHeadAttention, self).__init__()
        self.d_k = dim_model // h
```

```
        self.h = h
        # W_q, W_k, W_v, W_o
        self.linears = clones(nn.Linear(dim_model, dim_model), 4)

    def get(self, x, fields='qkv'):
        "Return a dict of queries / keys / values."
        batch_size = x.shape[0]
        ret = {}
        if 'q' in fields:
            ret['q'] = self.linears[0](x).view(batch_size, self.h, self.d_k)
        if 'k' in fields:
            ret['k'] = self.linears[1](x).view(batch_size, self.h, self.d_k)
        if 'v' in fields:
            ret['v'] = self.linears[2](x).view(batch_size, self.h, self.d_k)
        return ret

    def get_o(self, x):
        "get output of the multi-head attention"
        batch_size = x.shape[0]
        return self.linears[3](x.view(batch_size, -1))
```

8.3 DGL 如何通过图神经网络实现 Transformer

通过将注意力视为图中的边，并采用在边上传递的消息来引发适当的处理，您将从不同角度了解 Transformer。

8.3.1 图结构

通过将源语句和目标语句的标记映射到节点来构造图。完整的 Transformer 图由三个子图组成:

源语言图。这是一个完整的图，每个令牌 s_i 都可以处理任何其他令牌 s_j(包括自循环)。**目标语言图**。这个图是半完整的，因为如果 i>j(一个输出标记不能依赖于将来的单词)，t_i 只关心 t_j。**跨语言图**。这是一个双向图，其中有一条边从每个源令牌 s_i 到每个目标令牌 t_j，这意味着每个目标令牌都可以参与到源令牌上。

【这里的图在原英文文档中都打不开】

在数据集准备阶段预构建图。

8.3.2 消息传递

定义了图结构之后，继续定义消息传递的计算。

假设您已经计算了所有的查询 q_i，键 k_i 和值 v_i。对于每个节点 i(无论它是源令牌还是目标令牌)，可以将注意力计算分解为两个步骤:

- 消息计算:取 q_i 与 k_j 的 scaled-dot 积，计算 i 与所有连接的节点 j 之间的注意力分值 $score_{ij}$。j 发送给 i 的消息将由分数 $score_{ij}$ 和值 v_j 组成。
- 消息聚合:根据 $score_{ij}$ 从所有 j 中聚合值 v_j。

简单实现：

消息计算：

计算分数并将源节点的 v 发送到目标邮箱：

```
def message_func(edges):
    return {'score': ((edges.src['k'] * edges.dst['q'])
                      .sum(-1, keepdim=True)),
            'v': edges.src['v']}
```

消息聚集：

对所有边进行归一化并加权求和以获得输出：

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


def reduce_func(nodes, d_k=64):
    v = nodes.mailbox['v']
    att = F.softmax(nodes.mailbox['score'] / th.sqrt(d_k), 1)
    return {'dx': (att * v).sum(1)}
```

在特定的边执行：

```
import functools.partial as partial
def naive_propagate_attention(self, g, eids):
    g.send_and_recv(eids, message_func, partial(reduce_func, d_k=self.d_k))
```

内置功能加速：

使用 DGL 内建的功能，可加快讯息传递过程，包括:

- fn.src_mul_egdes(src_field、edges_field、out_field)乘源属性和 edges 属性，并将结果发送到以 out_field 为键的目标节点的邮箱。
- fn.copy_edge(edges_field, out_field)将 edge 的属性复制到目标节点的邮箱。
- fn.sum(edges_field, out_field)对边的属性进行求和，并将聚合结果发送到目标节点的邮箱。

在这里，您将这些内置函数组装到 property_attention 中，后者也是最终实现中的主要图操作函数。 要使其加速，请将 softmax 操作分为以下步骤。 回想一下，每个 head 都有两个阶段。

这里直接用英文了：

1. Compute attention score by multiply src node's `k` and dst node's `q`
   - `g.apply_edges(src_dot_dst('k', 'q', 'score'), eids)`
2. Scaled Softmax over all dst nodes' in-coming edges
   - Step 1: Exponentialize score with scale normalize constant
     - `g.apply_edges(scaled_exp('score', np.sqrt(self.d_k)))`

$$\text{score}_{ij} \leftarrow \exp\left(\frac{\text{score}_{ij}}{\sqrt{d_k}}\right)$$

   - Step 2: Get the "values" on associated nodes weighted by "scores" on in-coming edges of each node; get the sum of "scores" on in-coming edges of each node for normalization. Note that here **wv** is not normalized.
     - `msg: fn.src_mul_edge('v', 'score', 'v'), reduce: fn.sum('v', 'wv')`

$$\text{wv}_j = \sum_{i=1}^{N} \text{score}_{ij} \cdot v_i$$

     - `msg: fn.copy_edge('score', 'score'), reduce: fn.sum('score', 'z')`

$$\text{z}_j = \sum_{i=1}^{N} \text{score}_{ij}$$

wv 的规范化工作留待后期处理。

```
def src_dot_dst(src_field, dst_field, out_field):
    def func(edges):
        return {out_field: (edges.src[src_field] * edges.dst[dst_field]).sum(-1,
keepdim=True)}

    return func

def scaled_exp(field, scale_constant):
    def func(edges):
        # clamp for softmax numerical stability
        return {field: th.exp((edges.data[field] / scale_constant).clamp(-5, 5))}

    return func


def propagate_attention(self, g, eids):
    # Compute attention score
    g.apply_edges(src_dot_dst('k', 'q', 'score'), eids)
    g.apply_edges(scaled_exp('score', np.sqrt(self.d_k)))
    # Update node state
    g.send_and_recv(eids,
```

```
                [fn.src_mul_edge('v', 'score', 'v'), fn.copy_edge('score',
'score')],
                [fn.sum('v', 'wv'), fn.sum('score', 'z')])
```

8.3.3 预处理和后处理

在 Transformer 中，需要在 propagate_attention 函数之前和之后对数据进行预处理。

预处理：预处理函数 pre_func 首先将节点表示标准化，然后将它们映射到一组查询，键和值，并以自注意力为例：

$$x \leftarrow \text{LayerNorm}(x)$$
$$[q, k, v] \leftarrow [W_q, W_k, W_v] \cdot x$$

后处理：后处理函数 post_funcs 完成与 Transformer 的一层相对应的整个计算：

（1） 规范化 wv 并获取多头注意层 o 的输出。

$$\text{wv} \leftarrow \frac{\text{wv}}{z}$$
$$o \leftarrow W_o \cdot \text{wv} + b_o$$

添加跳跃连接：

$$x \leftarrow x + o$$

（2） 在 x 上应用两层位置前馈层，然后添加残差连接：

$$x \leftarrow x + \text{LayerNorm}(\text{FFN}(x))$$

FFN 此处指前馈函数。

```
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.N = N
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def pre_func(self, i, fields='qkv'):
        layer = self.layers[i]
        def func(nodes):
            x = nodes.data['x']
            norm_x = layer.sublayer[0].norm(x)
            return layer.self_attn.get(norm_x, fields=fields)
        return func
    def post_func(self, i):
```

```
        layer = self.layers[i]
        def func(nodes):
            x, wv, z = nodes.data['x'], nodes.data['wv'], nodes.data['z']
            o = layer.self_attn.get_o(wv / z)
            x = x + layer.sublayer[0].dropout(o)
            x = layer.sublayer[1](x, layer.feed_forward)
            return {'x': x if i < self.N - 1 else self.norm(x)}
        return func

class Decoder(nn.Module):
    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.N = N
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def pre_func(self, i, fields='qkv', l=0):
        layer = self.layers[i]
        def func(nodes):
            x = nodes.data['x']
            if fields == 'kv':
                norm_x = x # In enc-dec attention, x has already been normalized.
            else:
                norm_x = layer.sublayer[l].norm(x)
            return layer.self_attn.get(norm_x, fields)
        return func

    def post_func(self, i, l=0):
        layer = self.layers[i]
        def func(nodes):
            x, wv, z = nodes.data['x'], nodes.data['wv'], nodes.data['z']
            o = layer.self_attn.get_o(wv / z)
            x = x + layer.sublayer[l].dropout(o)
            if l == 1:
                x = layer.sublayer[2](x, layer.feed_forward)
            return {'x': x if i < self.N - 1 else self.norm(x)}
        return func
```

这样就完成了 Transformer 中一层编码器和解码器的所有过程。

注意：子层连接部分与原论文略有不同。但是，这个实现与 The Annotated Transformer and OpenNMT 相同。

8.4 Transformer graph 的主要类

可以将 Transformer 的处理流程视为完整图中的两阶段消息传递（适当地添加预处理和后处理）：1）编码器中的自注意力，2）解码器中的自注意力，然后

接一个跨编码器和解码器之间的注意力，如下所示。

```
class Transformer(nn.Module):
    def __init__(self, encoder, decoder, src_embed, tgt_embed, pos_enc, generator, h,
d_k):
        super(Transformer, self).__init__()
        self.encoder, self.decoder = encoder, decoder
        self.src_embed, self.tgt_embed = src_embed, tgt_embed
        self.pos_enc = pos_enc
        self.generator = generator
        self.h, self.d_k = h, d_k

    def propagate_attention(self, g, eids):
        # Compute attention score
        g.apply_edges(src_dot_dst('k', 'q', 'score'), eids)
        g.apply_edges(scaled_exp('score', np.sqrt(self.d_k)))
        # Send weighted values to target nodes
        g.send_and_recv(eids,
                        [fn.src_mul_edge('v', 'score', 'v'), fn.copy_edge('score',
'score')],
                        [fn.sum('v', 'wv'), fn.sum('score', 'z')])

    def update_graph(self, g, eids, pre_pairs, post_pairs):
        "Update the node states and edge states of the graph."

        # Pre-compute queries and key-value pairs.
        for pre_func, nids in pre_pairs:
            g.apply_nodes(pre_func, nids)
        self.propagate_attention(g, eids)
        # Further calculation after attention mechanism
        for post_func, nids in post_pairs:
            g.apply_nodes(post_func, nids)

    def forward(self, graph):
        g = graph.g
        nids, eids = graph.nids, graph.eids

        # Word Embedding and Position Embedding
        src_embed, src_pos = self.src_embed(graph.src[0]), self.pos_enc(graph.src[1])
        tgt_embed, tgt_pos = self.tgt_embed(graph.tgt[0]), self.pos_enc(graph.tgt[1])
        g.nodes[nids['enc']].data['x'] = self.pos_enc.dropout(src_embed + src_pos)
        g.nodes[nids['dec']].data['x'] = self.pos_enc.dropout(tgt_embed + tgt_pos)

        for i in range(self.encoder.N):
            # Step 1: Encoder Self-attention
```

```
            pre_func = self.encoder.pre_func(i, 'qkv')
            post_func = self.encoder.post_func(i)
            nodes, edges = nids['enc'], eids['ee']
            self.update_graph(g, edges, [(pre_func, nodes)], [(post_func, nodes)])

        for i in range(self.decoder.N):
            # Step 2: Dncoder Self-attention
            pre_func = self.decoder.pre_func(i, 'qkv')
            post_func = self.decoder.post_func(i)
            nodes, edges = nids['dec'], eids['dd']
            self.update_graph(g, edges, [(pre_func, nodes)], [(post_func, nodes)])
            # Step 3: Encoder-Decoder attention
            pre_q = self.decoder.pre_func(i, 'q', 1)
            pre_kv = self.decoder.pre_func(i, 'kv', 1)
            post_func = self.decoder.post_func(i, 1)
            nodes_e, nodes_d, edges = nids['enc'], nids['dec'], eids['ed']
            self.update_graph(g, edges, [(pre_q, nodes_d), (pre_kv, nodes_e)],
[(post_func, nodes_d)])

        return self.generator(g.ndata['x'][nids['dec']])
```

注意：通过调用 update_graph 函数，您可以用几乎相同的代码在任何子图上创建自己的 Transformer。这种灵活性使我们能够发现新的稀疏结构(这里提到的 c.f. local attention)。注意，在这个实现中，您没有使用掩码或填充，这使得逻辑更加清晰并节省内存。这样做的代价是实现速度较慢。

8.5 训练

本教程不介绍其他技术，例如原始论文中提到的标签平滑和 Noam 优化。有关这些模块的详细说明，请阅读由哈佛 NLP 团队编写的 The Annotated Transformer。

8.5.1 任务和数据集

Transformer 是用于各种 NLP 任务的通用框架。本教程重点介绍 sequence to sequence：这是说明其工作原理的典型案例。

对于数据集，有两个示例任务：复制和排序，以及两个实际翻译任务：multi30k en-de 任务和 wmt14 en-de 任务。

- copy dataset: copy input sequences to output. (train/valid/test: 9000, 1000, 1000)
- sort dataset: sort input sequences as output. (train/valid/test: 9000, 1000, 1000)
- Multi30k en-de, translate sentences from En to De. (train/valid/test: 29000, 1000, 1000)
- WMT14 en-de, translate sentences from En to De. (Train/Valid/Test: 4500966/3000/3003)

注意：使用 wmt14 进行训练需要多 GPU 支持，并且不可用。

8.5.2 构造图

批处理：这类似于您处理 Tree-LSTM 的方式。 预先构建一个图池化，包括输入长度和输出长度的所有可能组合。 然后，对于批次中的每个样本，调用 dgl.batch 将其大小的图一起批处理成一个大图。

您可以将创建图池化和构建 BatchedGraph 的过程包装在 dataset.GraphPool 和 dataset.TranslationDataset 中。

```
graph_pool = GraphPool()

data_iter = dataset(graph_pool, mode='train', batch_size=1, devices=devices)
for graph in data_iter:
    print(graph.nids['enc']) # encoder node ids
    print(graph.nids['dec']) # decoder node ids
    print(graph.eids['ee']) # encoder-encoder edge ids
    print(graph.eids['ed']) # encoder-decoder edge ids
    print(graph.eids['dd']) # decoder-decoder edge ids
    print(graph.src[0]) # Input word index list
    print(graph.src[1]) # Input positions
    print(graph.tgt[0]) # Output word index list
    print(graph.tgt[1]) # Ouptut positions
    break
```

输出：

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8], device='cuda:0')
tensor([ 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], device='cuda:0')
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
        18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
        54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
        72, 73, 74, 75, 76, 77, 78, 79, 80], device='cuda:0')
tensor([ 81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,  92,  93,  94,
         95,  96,  97,  98,  99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
        109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122,
        123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136,
        137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150,
        151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164,
        165, 166, 167, 168, 169, 170], device='cuda:0')
tensor([171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184,
        185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198,
        199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212,
        213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225],
       device='cuda:0')
tensor([28, 25,  7, 26,  6,  4,  5,  9, 18], device='cuda:0')
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8], device='cuda:0')
tensor([ 0, 28, 25,  7, 26,  6,  4,  5,  9, 18], device='cuda:0')
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], device='cuda:0')
```

8.6 将它们放在一起

在 copy 任务上训练一层 128 维度的单头 Transformer。 将其他参数设置为默认值。

推理模块未包含在本教程中。 它需要 beam 搜索。 有关完整的实现，请参阅 GitHub 存储库。

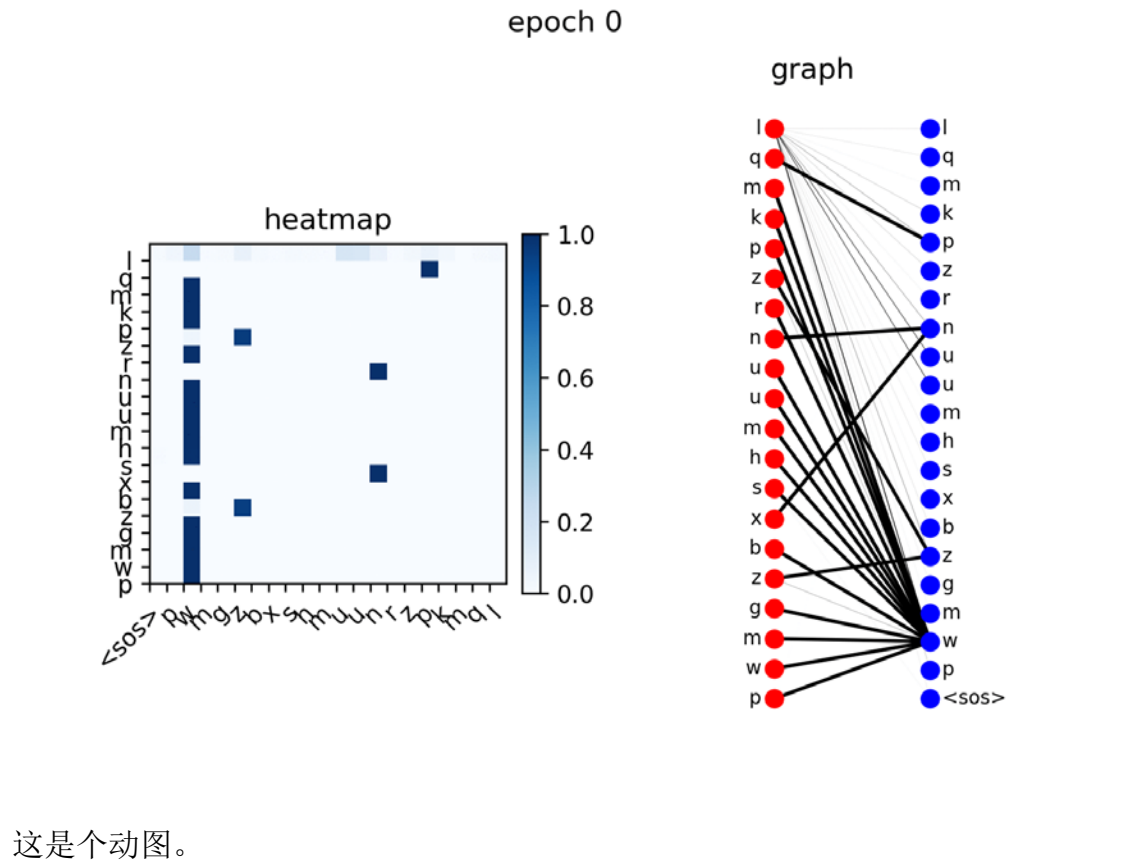
https://github.com/dmlc/dgl/tree/master/examples/pytorch/transformer

```
from tqdm import tqdm
import torch as th
import numpy as np

from loss import LabelSmoothing, SimpleLossCompute
from modules import make_model
from optims import NoamOpt
from dgl.contrib.transformer import get_dataset, GraphPool

def run_epoch(data_iter, model, loss_compute, is_train=True):
    for i, g in tqdm(enumerate(data_iter)):
        with th.set_grad_enabled(is_train):
            output = model(g)
            loss = loss_compute(output, g.tgt_y, g.n_tokens)
    print('average loss: {}'.format(loss_compute.avg_loss))
    print('accuracy: {}'.format(loss_compute.accuracy))

N = 1
batch_size = 128
devices = ['cuda' if th.cuda.is_available() else 'cpu']

dataset = get_dataset("copy")
V = dataset.vocab_size
criterion = LabelSmoothing(V, padding_idx=dataset.pad_id, smoothing=0.1)
dim_model = 128

# Create model
model = make_model(V, V, N=N, dim_model=128, dim_ff=128, h=1)

# Sharing weights between Encoder & Decoder
model.src_embed.lut.weight = model.tgt_embed.lut.weight
model.generator.proj.weight = model.tgt_embed.lut.weight

model, criterion = model.to(devices[0]), criterion.to(devices[0])
model_opt = NoamOpt(dim_model, 1, 400,
```

```
                th.optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.98),
eps=1e-9))
loss_compute = SimpleLossCompute

att_maps = []
for epoch in range(4):
    train_iter = dataset(graph_pool, mode='train', batch_size=batch_size,
devices=devices)
    valid_iter = dataset(graph_pool, mode='valid', batch_size=batch_size,
devices=devices)
    print('Epoch: {} Training...'.format(epoch))
    model.train(True)
    run_epoch(train_iter, model,
              loss_compute(criterion, model_opt), is_train=True)
    print('Epoch: {} Evaluating...'.format(epoch))
    model.att_weight_map = None
    model.eval()
    run_epoch(valid_iter, model,
              loss_compute(criterion, None), is_train=False)
    att_maps.append(model.att_weight_map)
```

8.7 可视化

经过训练后，您可以可视化 Transformer 在 copy 任务上生成的注意力。



这是个动图。

从图中可以看到，解码器节点逐渐学会处理输入序列中对应的节点，这是预期的行为。

8.7.1 多头注意力

除了在 toy 任务上训练单头注意力。我们还可视化了在 multi-30k 数据集上训练的单层 Transformer 网络的编码器的自注意力、解码器的自注意力和编码器-解码器注意力的注意力得分。

从想象中你可以看到不同的头，这就是你所期望的。不同的头学习不同的词组之间的关系。

- 编码器自注意力
- 编码器-解码器自注意力：目标序列中的大多数单词都与源序列中的相关单词相关联，例如：生成“ See”（在 De 中）时，多个头部都附着“ lake”上； 生成“Eisfischerhütte”时，几个头都附着“ ice”。
- 解码器自注意力：大多数单词都在其前几个单词的后面。

【同样的，图失效了。】

8.8 自适应通用 Transformer

Google 最近的研究论文 Universal Transformer 就是一个例子，展示了 update_graph 如何适应更复杂的更新规则。

提出了 The Universal Transformer，通过在 Transformer 中引入递归来解决 vanilla Transformer 在计算上不通用的问题：

- Universal Transformer 的基本思想是在每个重复步骤中，通过在表示形式上施加一个 Transformer 层，反复修改其在序列中所有符号的表示形式。
- 与普通 Transformer 相比，Universal Transformer 在其各层之间共享权重，并且它不固定循环时间（这意味着 Transformer 中的层数）。

进一步的优化采用 adaptive computation time (ACT) (ACT)机制，允许模型动态调整序列中每个位置的表示被修改的次数(以下称为 step)。该模型也称为 Adaptive Universal Transformer (AUT)。

在 AUT 中，您维护一个活动节点列表。 在每个步骤 t 中，我们通过以下公式计算此列表中所有节点的停止概率：h（0 <h <1）：

$$h_i^t = \sigma(W_h x_i^t + b_h)$$

然后动态地确定哪些节点仍然是活动的。当且仅当$\sum_{t=1}^{T-1} h_t < 1 - \varepsilon \leq \sum_{t=1}^{T-1} h_t$时，节点停止运行。停止的节点将从列表中删除。过程继续进行，直到列表为空或达到预定义的最大步长。从 DGL 的角度来看，这意味着“活动”图会随着时间的推移变得更加稀疏。

节点 s_i 的最终状态是h_i^t对x_i^t的加权平均值：

$$s_i = \sum_{t=1}^{T} h_i^t \cdot x_i^t$$

在 DGL 中，通过在仍处于活动状态的节点以及与此节点关联的边上调用 update_graph 来实现算法。 以下代码显示了 DGL 中的 Universal Transformer 类：

```
class UTransformer(nn.Module):
    "Universal Transformer(https://arxiv.org/pdf/1807.03819.pdf) with
ACT(https://arxiv.org/pdf/1603.08983.pdf)."
    MAX_DEPTH = 8
    thres = 0.99
    act_loss_weight = 0.01
    def __init__(self, encoder, decoder, src_embed, tgt_embed, pos_enc, time_enc,
generator, h, d_k):
        super(UTransformer, self).__init__()
        self.encoder,  self.decoder = encoder, decoder
        self.src_embed, self.tgt_embed = src_embed, tgt_embed
        self.pos_enc, self.time_enc = pos_enc, time_enc
        self.halt_enc = HaltingUnit(h * d_k)
        self.halt_dec = HaltingUnit(h * d_k)
        self.generator = generator
        self.h, self.d_k = h, d_k

    def step_forward(self, nodes):
        # add positional encoding and time encoding, increment step by one
        x = nodes.data['x']
        step = nodes.data['step']
        pos = nodes.data['pos']
        return {'x': self.pos_enc.dropout(x + self.pos_enc(pos.view(-1)) +
self.time_enc(step.view(-1))),
                'step': step + 1}

    def halt_and_accum(self, name, end=False):
        "field: 'enc' or 'dec'"
        halt = self.halt_enc if name == 'enc' else self.halt_dec
        thres = self.thres
        def func(nodes):
            p = halt(nodes.data['x'])
            sum_p = nodes.data['sum_p'] + p
            active = (sum_p < thres) & (1 - end)
            _continue = active.float()
            r = nodes.data['r'] * (1 - _continue) + (1 - sum_p) * _continue
            s = nodes.data['s'] + ((1 - _continue) * r + _continue * p) * nodes.data['x']
            return {'p': p, 'sum_p': sum_p, 'r': r, 's': s, 'active': active}
        return func

    def propagate_attention(self, g, eids):
        # Compute attention score
        g.apply_edges(src_dot_dst('k', 'q', 'score'), eids)
        g.apply_edges(scaled_exp('score', np.sqrt(self.d_k)), eids)
```

```
        # Send weighted values to target nodes
        g.send_and_recv(eids,
                        [fn.src_mul_edge('v', 'score', 'v'), fn.copy_edge('score',
'score')],
                        [fn.sum('v', 'wv'), fn.sum('score', 'z')])

    def update_graph(self, g, eids, pre_pairs, post_pairs):
        "Update the node states and edge states of the graph."
        # Pre-compute queries and key-value pairs.
        for pre_func, nids in pre_pairs:
            g.apply_nodes(pre_func, nids)
        self.propagate_attention(g, eids)
        # Further calculation after attention mechanism
        for post_func, nids in post_pairs:
            g.apply_nodes(post_func, nids)

    def forward(self, graph):
        g = graph.g
        N, E = graph.n_nodes, graph.n_edges
        nids, eids = graph.nids, graph.eids

        # embed & pos
        g.nodes[nids['enc']].data['x'] = self.src_embed(graph.src[0])
        g.nodes[nids['dec']].data['x'] = self.tgt_embed(graph.tgt[0])
        g.nodes[nids['enc']].data['pos'] = graph.src[1]
        g.nodes[nids['dec']].data['pos'] = graph.tgt[1]

        # init step
        device = next(self.parameters()).device
        g.ndata['s'] = th.zeros(N, self.h * self.d_k, dtype=th.float, device=device)
# accumulated state
        g.ndata['p'] = th.zeros(N, 1, dtype=th.float, device=device)                  #
halting prob
        g.ndata['r'] = th.ones(N, 1, dtype=th.float, device=device)                   #
remainder
        g.ndata['sum_p'] = th.zeros(N, 1, dtype=th.float, device=device)
# sum of pondering values
        g.ndata['step'] = th.zeros(N, 1, dtype=th.long, device=device)                #
step
        g.ndata['active'] = th.ones(N, 1, dtype=th.uint8, device=device)
# active

        for step in range(self.MAX_DEPTH):
            pre_func = self.encoder.pre_func('qkv')
```

```
            post_func = self.encoder.post_func()
            nodes = g.filter_nodes(lambda v: v.data['active'].view(-1), nids['enc'])
            if len(nodes) == 0: break
            edges = g.filter_edges(lambda e: e.dst['active'].view(-1), eids['ee'])
            end = step == self.MAX_DEPTH - 1
            self.update_graph(g, edges,
                              [(self.step_forward, nodes), (pre_func, nodes)],
                              [(post_func, nodes), (self.halt_and_accum('enc', end),
nodes)])

        g.nodes[nids['enc']].data['x'] =
self.encoder.norm(g.nodes[nids['enc']].data['s'])

        for step in range(self.MAX_DEPTH):
            pre_func = self.decoder.pre_func('qkv')
            post_func = self.decoder.post_func()
            nodes = g.filter_nodes(lambda v: v.data['active'].view(-1), nids['dec'])
            if len(nodes) == 0: break
            edges = g.filter_edges(lambda e: e.dst['active'].view(-1), eids['dd'])
            self.update_graph(g, edges,
                              [(self.step_forward, nodes), (pre_func, nodes)],
                              [(post_func, nodes)])

            pre_q = self.decoder.pre_func('q', 1)
            pre_kv = self.decoder.pre_func('kv', 1)
            post_func = self.decoder.post_func(1)
            nodes_e = nids['enc']
            edges = g.filter_edges(lambda e: e.dst['active'].view(-1), eids['ed'])
            end = step == self.MAX_DEPTH - 1
            self.update_graph(g, edges,
                              [(pre_q, nodes), (pre_kv, nodes_e)],
                              [(post_func, nodes), (self.halt_and_accum('dec', end),
nodes)])

        g.nodes[nids['dec']].data['x'] =
self.decoder.norm(g.nodes[nids['dec']].data['s'])
        act_loss = th.mean(g.ndata['r']) # ACT loss

        return self.generator(g.ndata['x'][nids['dec']]), act_loss *
self.act_loss_weight
```

调用 filter_nodes 和 filter_edge 来查找仍然活动的节点/边:

注意：

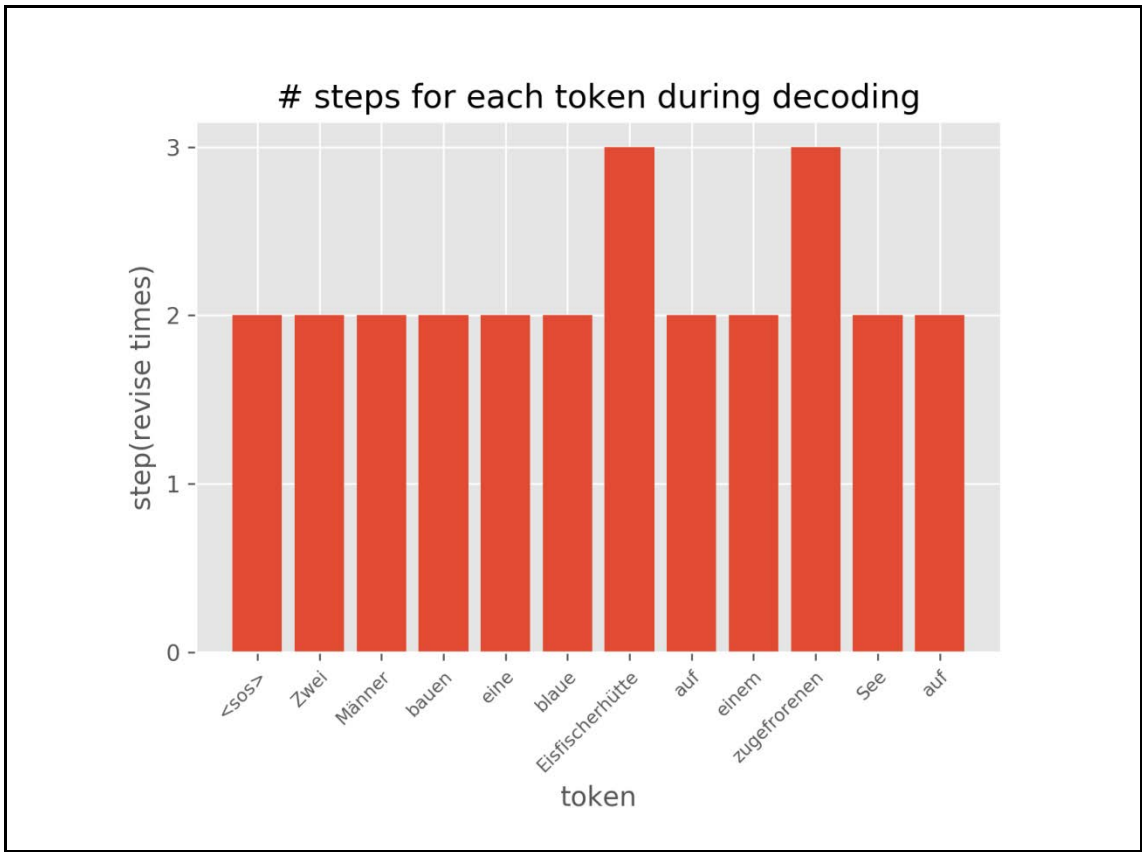
- filter_nodes()接受一个谓词和一个节点 ID 列表/张量作为输入，然后返回一个满足给定谓词的节点 ID 张量。

- filter_edges()接受一个谓词和一个边 ID 列表/张量作为输入，然后返回一个满足给定谓词的边 ID 张量。

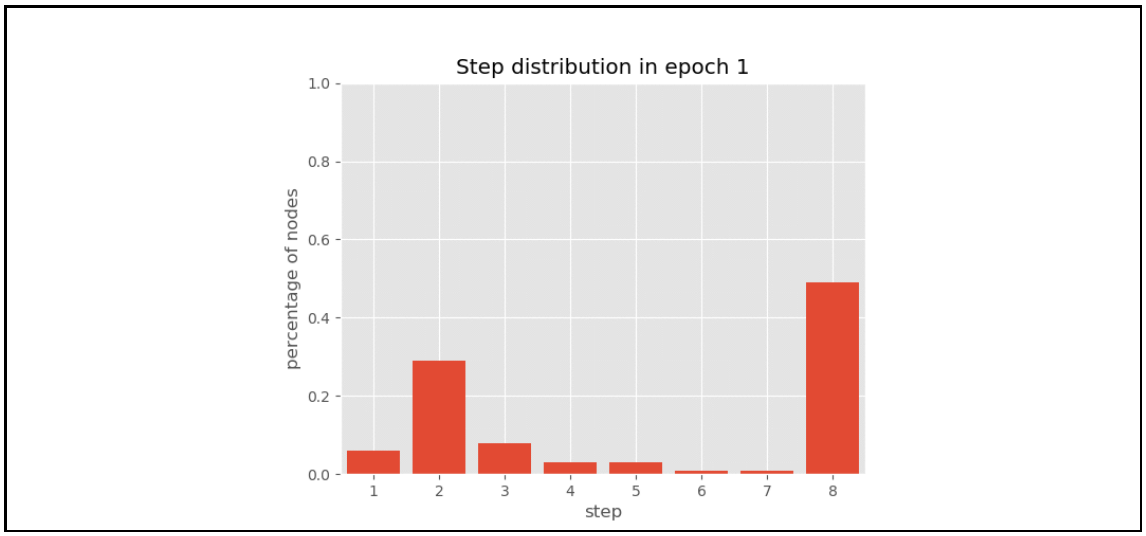
有关完整的实现，请参见 GitHub 存储库。

https://github.com/dmlc/dgl/tree/master/examples/pytorch/transformer/modules/act.py

下图显示了自适应计算时间的影响。 句子的不同位置在不同的时间进行了修改。



您还可以在排序任务的 AUT 训练过程中可视化节点上步长分布的动态（达到 99.7%的准确度），这演示了 AUT 如何学习减少训练中的重复步骤。



【这是个动图】

注意：由于存在许多依赖性，笔记本本身无法执行。 下载 7_transformer.py，

然后将 python 脚本复制到 examples / pytorch / transformer 目录，然后运行 python 7_transformer.py 以查看其工作方式。