Note

Click here to download the full example code

Relational Graph Convolutional Network

Author: Lingfan Yu, Mufei Li, Zheng Zhang

Warning

The tutorial aims at gaining insights into the paper, with code as a mean of explanation. The implementation thus is NOT optimized for running efficiency. For recommended implementation, please refer to the official examples.

In this tutorial, you learn how to implement a relational graph convolutional network (R-GCN). This type of network is one effort to generalize GCN to handle different relationships between entities in a knowledge base. To learn more about the research behind R-GCN, see Modeling Relational Data with Graph Convolutional Networks

The straightforward graph convolutional network (GCN) exploits structural information of a dataset (that is, the graph connectivity) in order to improve the extraction of node representations. Graph edges are left as untyped.

A knowledge graph is made up of a collection of triples in the form subject, relation, object. Edges thus encode important information and have their own embeddings to be learned. Furthermore, there may exist multiple edges among any given pair.

A brief introduction to R-GCN

In *statistical relational learning* (SRL), there are two fundamental tasks:

- **Entity classification** - Where you assign types and categorical properties to entities.
- **Link prediction** - Where you recover missing triples.

In both cases, missing information is expected to be recovered from the neighborhood structure of the graph. For example, the R-GCN paper cited earlier provides the following example. Knowing that Mikhail Baryshnikov was educated at the Vaganova Academy implies both that Mikhail Baryshnikov should have the label person, and that the triple (Mikhail Baryshnikov, lived in, Russia) must belong to the knowledge graph.

R-GCN solves these two problems using a common graph convolutional network. It's extended with multi-edge encoding to compute embedding of the entities, but with different downstream processing.

- Entity classification is done by attaching a softmax classifier at the final embedding of an entity (node). Training is through loss of standard cross-entropy.
- Link prediction is done by reconstructing an edge with an autoencoder architecture, using a parameterized score function. Training uses negative sampling.

This tutorial focuses on the first task, entity classification, to show how to generate entity representation. Complete code for both tasks is found in the DGL Github repository.

Key ideas of R-GCN

Recall that in GCN, the hidden representation for each node i at $(l+1)^{th}$ layer is computed by:

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

where c_i is a normalization constant.

The key difference between R-GCN and GCN is that in R-GCN, edges can represent different relations. In GCN, weight $W^{(l)}$ in equation (1) is shared by all edges in layer l. In contrast, in R-GCN, different edge types use different weights and only edges of the same relation type r are associated with the same projection weight $W_r^{(l)}$.

So the hidden representation of entities in $(l+1)^{th}$ layer in R-GCN can be formulated as the following equation:

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

where N_i^r denotes the set of neighbor indices of node i under relation $r \in R$ and $c_{i,r}$ is a normalization constant. In entity classification, the R-GCN paper uses $c_{i,r} = |N_i^r|$.

The problem of applying the above equation directly is the rapid growth of the number of parameters, especially with highly multi-relational data. In order to reduce model parameter size and prevent overfitting, the original paper proposes to use basis decomposition.

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

Therefore, the weight $W_r^{(l)}$ is a linear combination of basis transformation $V_b^{(l)}$ with coefficients $a_{rb}^{(l)}$. The number of bases B is much smaller than the number of relations in the knowledge base.

Note

Another weight regularization, block-decomposition, is implemented in the link prediction.

Implement R-GCN in DGL

An R-GCN model is composed of several R-GCN layers. The first R-GCN layer also serves as input layer and takes in features (for example, description texts) that are associated with node entity and project to hidden space. In this tutorial, we only use the entity ID as an entity feature.

R-GCN layers

For each node, an R-GCN layer performs the following steps:

- Compute outgoing message using node representation and weight matrix associated with the edge type (message function)
- Aggregate incoming messages and generate new node representations (reduce and apply function)

The following code is the definition of an R-GCN hidden layer.

Note

Each relation type is associated with a different weight. Therefore, the full weight matrix has three dimensions: relation, input_feature, output_feature.

Note

This is showing how to implement an R-GCN from scratch. DGL provides a more efficient `builtin R-GCN layer module`.

```python
import os

os.environ["DGLBACKEND"] = "pytorch"
from functools import partial

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


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[dgl.ETYPE] * self.in_feat + edges.src["id"]
            return {"msg": embed[index] * edges.data["norm"]}

    else:

        def message_func(edges):
            w = weight[edges.data[dgl.ETYPE]]
            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)
```

Full R-GCN model defined

```
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")
```

Handle dataset

This tutorial uses Institute for Applied Informatics and Formal Description Methods (AIFB) dataset from R-GCN paper.

```
# load graph data
dataset = dgl.data.rdf.AIFBDataset()
g = dataset[0]
category = dataset.predict_category
train_mask = g.nodes[category].data.pop("train_mask")
test_mask = g.nodes[category].data.pop("test_mask")
train_idx = torch.nonzero(train_mask, as_tuple=False).squeeze()
test_idx = torch.nonzero(test_mask, as_tuple=False).squeeze()
labels = g.nodes[category].data.pop("label")
num_rels = len(g.canonical_etypes)
num_classes = dataset.num_classes
# normalization factor
for cetype in g.canonical_etypes:
    g.edges[cetype].data["norm"] = dgl.norm_by_dst(g, cetype).unsqueeze(1)
category_id = g.ntypes.index(category)
```

Out:

```
Done loading data from cached files.
```

Create graph and model

```
# 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 = dgl.to_homogeneous(g, edata=["norm"])
node_ids = torch.arange(g.num_nodes())
target_idx = node_ids[g.ndata[dgl.NTYPE] == category_id]

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

Training loop

```
# 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)
    logits = logits[target_idx]
    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[test_idx], labels[test_idx])
    val_acc = torch.sum(logits[test_idx].argmax(dim=1) == labels[test_idx])
    val_acc = val_acc.item() / len(test_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()
        )
    )
```

Out:

```
start training...
Epoch 00000 | Train Accuracy: 0.2214 | Train Loss: 1.3864 | Validation Accuracy: 0.2500 |
Validation loss: 1.3865
Epoch 00001 | Train Accuracy: 0.9714 | Train Loss: 1.3568 | Validation Accuracy: 0.9167 |
Validation loss: 1.3636
Epoch 00002 | Train Accuracy: 0.9714 | Train Loss: 1.3125 | Validation Accuracy: 0.9444 |
Validation loss: 1.3287
Epoch 00003 | Train Accuracy: 0.9714 | Train Loss: 1.2511 | Validation Accuracy: 0.9444 |
Validation loss: 1.2796
Epoch 00004 | Train Accuracy: 0.9714 | Train Loss: 1.1775 | Validation Accuracy: 0.9444 |
Validation loss: 1.2192
Epoch 00005 | Train Accuracy: 0.9643 | Train Loss: 1.1028 | Validation Accuracy: 0.9444 |
Validation loss: 1.1558
Epoch 00006 | Train Accuracy: 0.9643 | Train Loss: 1.0381 | Validation Accuracy: 0.9444 |
Validation loss: 1.0989
Epoch 00007 | Train Accuracy: 0.9714 | Train Loss: 0.9876 | Validation Accuracy: 0.9444 |
Validation loss: 1.0523
Epoch 00008 | Train Accuracy: 0.9714 | Train Loss: 0.9497 | Validation Accuracy: 0.9444 |
Validation loss: 1.0153
Epoch 00009 | Train Accuracy: 0.9714 | Train Loss: 0.9210 | Validation Accuracy: 0.9444 |
Validation loss: 0.9859
```

The second task, link prediction

So far, you have seen how to use DGL to implement entity classification with an R-GCN model. In the knowledge base setting, representation generated by R-GCN can be used to uncover potential relationships between nodes. In the R-GCN paper, the authors feed the entity representations generated by R-GCN into the DistMult prediction model to predict possible relationships.

The implementation is similar to that presented here, but with an extra DistMult layer stacked on top of the R-GCN layers. You can find the complete implementation of link prediction with R-GCN in our Github Python code example.

Total running time of the script: (0 minutes 3.672 seconds)

Download Python source code: 4_rgcn.py

Download Jupyter notebook: 4_rgcn.ipynb