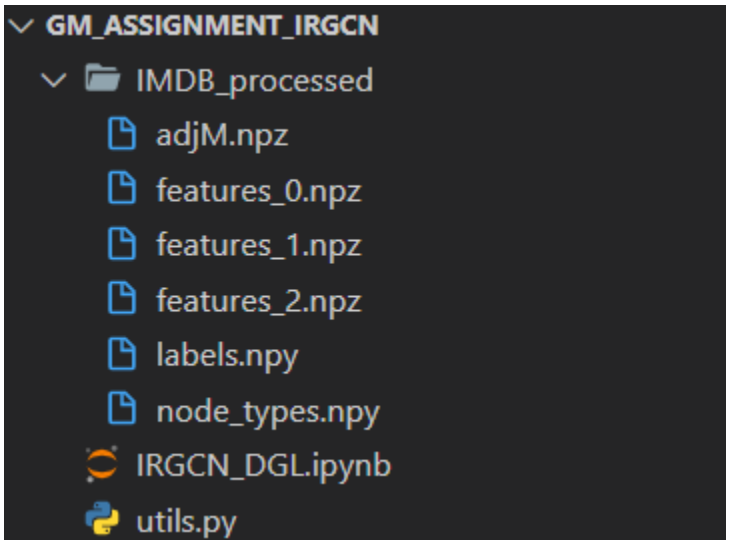
Setup

Create a folder in your drive called **GM_Assignment_IRGCN**
Extract the contents of the zip file in the GM_Assignment_IRGCN folder
The project structure should be as follows:



IMDb Dataset (Preprocessed)

A subset of the IMDb dataset is used, containing **4278 movies, 2081 directors, and 5257 actors** after data preprocessing.

The preprocessed dataset contains **3 node types (0 → movie, 1 → director, 2 → actor)**. The type value for each node is present in **node_types.npy**

The movies are labeled as one of **three classes (0 → Action, 1 → Comedy, 2 → Drama)** based on their genre information. These labels are present in **labels.npy**

The **adjacency matrix** information is present in **adjM.npz**. The first 4278 rows in the matrix contain the adjacency information of movie nodes, the next 2081 for director nodes, and the last 5257 for actor nodes.

The **pretrained initial feature embeddings** for the nodes are also available. The feature embeddings for the nodes of movies, directors and actors are present in **features_0.npz, features_1.npz, and features_2.npz** respectively.

Code Explanation

Basic Setup

Install dgl with cuda build

```
!pip install -q dgl-cu110
```

Required Imports

```
import networkx as nx
import numpy as np
import scipy.sparse
import pickle

from sklearn.model_selection import train_test_split

import time

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

import dgl
from dgl.data.knowledge_graph import load_data
from dgl.nn.pytorch import RelGraphConv

import sys
sys.path.insert(0,'/content/drive/My Drive/GM_Assignment_IRGCN')
import utils
```

utils.py contains helper functions for the evaluation of the model

Data Importing and Preprocessing

Load information from Preprocessed IMDB dataset

```
features_0 = scipy.sparse.load_npz(prefix + '/features_0.npz')
features_1 = scipy.sparse.load_npz(prefix + '/features_1.npz')
features_2 = scipy.sparse.load_npz(prefix + '/features_2.npz')
adjM = scipy.sparse.load_npz(prefix + '/adjM.npz')
type_mask = np.load(prefix + '/node_types.npy')
labels = np.load(prefix + '/labels.npy')
adjM = adjM.toarray()

# Combine the separate feature embeddings
features_list = [features_0, features_1, features_2]
features_ls = [torch.FloatTensor(features.todense()) for features in features_list]
features_ls = torch.cat((features_ls[0], features_ls[1], features_ls[2]))
```

We define **6 relation types**
0 → {(movie, director) | adjM[movie, director] = 1 && label[movie] = 0}
1 → {(movie, director) | adjM[movie, director] = 1 && label[movie] = 1}
2 → {(movie, director) | adjM[movie, director] = 1 && label[movie] = 2}
3 → {(movie, actor) | adjM[movie, actor] = 1 && label[movie] = 0}
4 → {(movie, actor) | adjM[movie, actor] = 1 && label[movie] = 1}
5 → {(movie, actor) | adjM[movie, actor] = 1 && label[movie] = 2}

These relations are constructed as follows:

```
num_nodes = type_mask.size
num_edges = adjM.size
num_rels = 6
g_data=[]
for x in range(labels.size): # Iterating over movie nodes
  for y in range(labels.size, type_mask.size): # Iterating over director and actor nodes
    if adjM[x][y]==1:
        rel = 0
        if type_mask[y]==1:
            rel = labels[x]   # Can be 0, 1, 2 based on label(Action/Comedy/Drama)
        elif type_mask[y]==2:
            rel = 3 + labels[x]   # Can be 3, 4, 5 based on label(Action/Comedy/Drama)
        g_data.append([x, rel, y])
```

g_data is used to construct the **training, test and validation** data sets. Each data point is marked by a **triplet (head node, relation, tail node)**.

Model:

Existing Implementations:

The code for **BaseRGCN** and **LinkPredict** models have been taken from the **official implementation** of RGCN using DGL. BaseRGCN contains abstract functions to be implemented by the RGCN model

```
def build_input_layer(self):
    return None

def build_hidden_layer(self, idx):
    raise NotImplementedError

def build_output_layer(self):
    return None
```

We introduce an **Embedding layer** initially to make use of the pretrained features from the IMDb dataset

```
class EmbeddingLayer(nn.Module):
    def __init__(self, num_nodes, h_dim):
        super(EmbeddingLayer, self).__init__()

        # Construct embeddings from pretrained features
        self.embedding = torch.nn.Embedding.from_pretrained(features_ls)

        self.linear = torch.nn.Linear(features_ls.size()[1], h_dim)

    def forward(self, g, h, r, norm):
        return F.relu(self.linear(self.embedding(h.squeeze())))
```

We set RGCN to use this EmbeddingLayer as its input. For the hidden layers, we use **DGL's existing implementation RelGraphConv** which takes care of the forward and backward propagations for the hidden layers.

```
class RGCN(BaseRGCN):
    def build_input_layer(self):
        return EmbeddingLayer(self.num_nodes, self.h_dim)

    def build_hidden_layer(self, idx):
        act = F.relu if idx < self.num_hidden_layers - 1 else None
        return RelGraphConv(self.h_dim, self.h_dim, self.num_rels, "bdd",
                self.num_bases, activation=act, self_loop=True,
                dropout=self.dropout)
```

The **LinkPredict** model uses this RGCN model to calculate the embeddings

```
class LinkPredict(nn.Module):
    def __init__(self, in_dim, h_dim, num_rels, num_bases=-1,
                 num_hidden_layers=1, dropout=0, use_cuda=False, reg_param=0):
        super(LinkPredict, self).__init__()
        self.rgcn = RGCN(in_dim, h_dim, h_dim, num_rels * 2, num_bases,
                         num_hidden_layers, dropout, use_cuda)
```

In the case of a standard RGCN model, the relation embedding matrix (containing the relation embedding h_r for each relation) is **directly trained from the loss function**. Therefore, the LinkPredict model considers the **relation embedding matrix as a Parameter**.

```
self.w_relation = nn.Parameter(torch.Tensor(num_rels, h_dim))
```

The scores in the LinkPredict model are evaluated by a **DistMult decoder**

```
def calc_score(self, embedding, triplets):
    # DistMult
    s = embedding[triplets[:,0]]
    r = self.w_relation[triplets[:,1]]
    o = embedding[triplets[:,2]]
    score = torch.sum(s * r * o, dim=1)
    return score
```

The Link Predict Model is supervised by **Binary Cross Entropy Loss**

```
def regularization_loss(self, embedding):
    return torch.mean(embedding.pow(2)) + torch.mean(self.w_relation.pow(2))

def get_loss(self, g, embed, triplets, labels):
    # triplets is a list of data samples (positive and negative)
    # each row in the triplets is a 3-tuple of (source, relation, destination)
    score = self.calc_score(embed, triplets)
    predict_loss = F.binary_cross_entropy_with_logits(score, labels)
    reg_loss = self.regularization_loss(embed)
    return predict_loss + self.reg_param * reg_loss
```

IRGCN:

We first **get the node embeddings** from the LinkPredict model

```
self.embedding = LinkPredict(num_nodes,
                             parameter_n_hidden,
                             num_rels,
                             num_bases=parameter_n_bases,
                             num_hidden_layers=parameter_n_layers,
                             dropout=parameter_dropout,
                             use_cuda=use_cuda,
                             reg_param=parameter_regularization)
```

We then **concatenate the head and tail** node embeddings for every triplet in the dataset and **pass them through the MLP**

```
head = self.embed[triplets[:,0]]
tail = self.embed[triplets[:,2]]
x = self.fcc1(torch.cat((head, tail), 1))
x = F.relu(x)
x = self.dropout(x)
x = self.fcc2(x)
```

Finally, we **add the embeddings corresponding to each relation** and **divide by the count of triplets containing that relation**

```
for i in range(self.num_rels):
  self.w_relation[i] = torch.mean(x[((triplets[:, 1] == i )*( labels==1)).nonzero().squeeze(1)], 0)
```

Similar to the LinkPredict model, the IRGCN model scores triplets using a **DistMult decoder**, and is supervised by a **Binary Cross Entropy loss function**.

IRGCN_Complex:

IRGCN_Complex is the same as IRGCN in all aspects, except that it scores triplets using a **ComplEx Decoder**

The embeddings are first split into two halves. The **first half is considered as the real part**, and the **second half is considered as the imaginary part**.

```
def split_node_emb(self, emb):
  # Splits the embedding in half. First half considered as real part, next half as imaginary part
  emb = torch.stack(list(torch.chunk(emb, 2, dim=1)), dim=1)
  emb = emb.permute(1, 0, 2)
  return (emb[0], emb[1])

def calc_score(self, triplets):
  # ComplEx
  s = self.embed[triplets[:,0]]
  r = self.w_relation[triplets[:,1]]
  o = self.embed[triplets[:,2]]

  # Split embeddings into real and imaginary parts
  (re_s, im_s) = self.split_node_emb(s)
  (re_r, im_r) = self.split_node_emb(r)
  (re_o, im_o) = self.split_node_emb(o)
```

These real and imaginary parts are used to evaluate the score.

ComplEx(z_u, r, z_v) = Re(z_u * r * conjugate(z_v)), where * denotes element-wise multiplication

```
# Calculate score
re_score = re_s * re_r - im_s * im_r
im_score = re_s * im_r + re_r * im_s
score = re_score * re_o + im_score * im_o
score = torch.sum(score, dim = 1)
return score
```

Training and Evaluation:

The code for training and evaluation is also taken from the **official implementation of RGCN using DGL**. The steps are the same for all models except for the model creations.

The dataset of triplets is first split into training, test and validation datasets

```
np.random.seed([3,1415])

np.random.shuffle(g_data)
train_data = [x for x in g_data if x[1]!=2]
X = [x for x in g_data if x[1]==2]
irgcn_train_data = X[:100]
for x in X[:100]:
  train_data.append(x)
test_data = X[100:]
valid_data = test_data
```

The required model (in this case IRGCN) is created

```
# create model
model = IRGCN(num_nodes,
                    parameter_n_hidden,
                    num_rels,
                    num_bases=parameter_n_bases,
                    num_hidden_layers=parameter_n_layers,
                    dropout=parameter_dropout,
                    use_cuda=use_cuda,
                    reg_param=parameter_regularization)
```

For each epoch, neighbourhood edge sampling is done to generate the training graph

```
g, node_id, edge_type, node_norm, data, labels = \
    utils.generate_sampled_graph_and_labels(
        train_data, parameter_graph_batch_size, parameter_graph_split_size,
        num_rels, adj_list, degrees, parameter_negative_sample,
        parameter_edge_sampler)
```

The generated data and labels are used to compute the embeddings in the forward propagation step, and to calculate the loss in the backward propagation step

```
model(g, node_id, edge_type, edge_norm, data, labels)

loss = model.get_loss(g, data, labels)
t1 = time.time()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), parameter_grad_norm) # clip gradients
optimizer.step()
t2 = time.time()
```

Validation is performed only after every certain number of epochs as it is time intensive. If the resulting MRR is better than the current MRR, the current state of the model is saved.

```
if epoch % parameter_evaluate_every == 0:
    # perform validation on CPU because full graph is too large
    if use_cuda:
        model.cpu()
    model.eval()
    print("start eval")
    mrr = utils.calc_mrr(model.embedding(test_graph, test_node_id, test_rel, test_norm), model.w_relation.cpu(),
                         torch.LongTensor(train_data).cpu(), valid_data, test_data, hits=[1, 3, 10],
                         eval_bz=parameter_eval_batch_size, eval_p=parameter_eval_protocol)
    # save best model
    if best_mrr < mrr:
        best_mrr = mrr
        torch.save({'state_dict': model.state_dict(), 'epoch': epoch}, model_state_file)
```

For Evaluation, firstly, a test graph is constructed.

```
test_graph, test_rel, test_norm = utils.build_test_graph(
    num_nodes, num_rels, train_data)
test_deg = test_graph.in_degrees(
            range(test_graph.number_of_nodes())).float().view(-1,1)
test_node_id = torch.arange(0, num_nodes, dtype=torch.long).view(-1, 1)
test_rel = torch.from_numpy(test_rel)
test_norm = node_norm_to_edge_norm(test_graph, torch.from_numpy(test_norm).view(-1, 1))
```

The model state having the best MRR is loaded to perform evaluation

```
checkpoint = torch.load(model_state_file)
if use_cuda:
    model.cpu() # test on CPU
model.eval()
model.load_state_dict(checkpoint['state_dict'])
print("Using best epoch: {}".format(checkpoint['epoch']))
utils.calc_mrr(model.embedding(test_graph, test_node_id, test_rel, test_norm), model.w_relation,
               torch.LongTensor(train_data), valid_data,test_data, hits=[1, 3, 10],
               eval_bz=parameter_eval_batch_size, eval_p=parameter_eval_protocol)
```