```
import numpy as np
import random
import torch
from torch.utils.data import Dataset

class GRDataset(Dataset):
    def __init__(self, data, u_items_list, u_users_list, u_users_items_list, i_users_list):
        self.data = data
        self.u_items_list = u_items_list
        self.u_users_list = u_users_list
        self.u_users_items_list = u_users_items_list
        self.i_users_list = i_users_list


    def __getitem__(self, index):
        uid = self.data[index][0]
        iid = self.data[index][1]
        label = self.data[index][2]
        u_items = self.u_items_list[uid]
        u_users = self.u_users_list[uid]
        u_users_items = self.u_users_items_list[uid]
        i_users = self.i_users_list[iid]

        return (uid, iid, label), u_items, u_users, u_users_items, i_users

    def __len__(self):
        return len(self.data)
```

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

class _MultiLayerPercep(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(_MultiLayerPercep, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, input_dim // 2, bias=True),
            nn.LeakyReLU(0.2),
            nn.Linear(input_dim // 2, output_dim, bias=True),
        )

    def forward(self, x):
        return self.mlp(x)


class _Aggregation(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(_Aggregation, self).__init__()
        self.aggre = nn.Sequential(
            nn.Linear(input_dim, output_dim, bias=True),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.aggre(x)


class _UserModel(nn.Module):
    ''' User modeling to learn user latent factors.
    User modeling leverages two types aggregation: item aggregation and social aggregation
    '''
    def __init__(self, emb_dim, user_emb, item_emb, rate_emb):
        super(_UserModel, self).__init__()
        self.user_emb = user_emb
        self.item_emb = item_emb
        self.rate_emb = rate_emb
        self.emb_dim = emb_dim

        self.w1 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w2 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w3 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w4 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w5 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w6 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w7 = nn.Linear(self.emb_dim, self.emb_dim)


        self.g_v = _MultiLayerPercep(2 * self.emb_dim, self.emb_dim)
```

```
        self.user_items_att = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_items = _Aggregation(self.emb_dim, self.emb_dim)

        self.user_items_att_s1 = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_items_s1 = _Aggregation(self.emb_dim, self.emb_dim)
        self.user_users_att_s2 = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_neigbors_s2 = _Aggregation(self.emb_dim, self.emb_dim)

        self.u_user_users_att = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.u_aggre_neigbors = _Aggregation(self.emb_dim, self.emb_dim)



        self.combine_mlp = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(2 * self.emb_dim, 2*self.emb_dim, bias = True),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(2*self.emb_dim, self.emb_dim, bias = True),
            nn.ReLU()
        )

        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        # used for preventing zero div error when calculating softmax score
        self.eps = 1e-10

    def forward(self, uids, u_item_pad, u_user_pad, u_user_item_pad):
        # item aggregation
        q_a = self.item_emb(u_item_pad[:,:,0])   # B x maxi_len x emb_dim
        mask_u = torch.where(u_item_pad[:,:,0] > 0, torch.tensor([1.], device=self.device),
        u_item_er = self.rate_emb(u_item_pad[:, :, 1])  # B x maxi_len x emb_dim
        x_ia = self.g_v(torch.cat([q_a, u_item_er], dim=2).view(-1, 2 * self.emb_dim)).view
        p_i = mask_u.unsqueeze(2).expand_as(q_a) * self.user_emb(uids).unsqueeze(1).expand_

        alpha = self.user_items_att(torch.cat([self.w1(x_ia), self.w1(p_i)], dim = 2).view(
        alpha = torch.exp(alpha) * mask_u
        alpha = alpha / (torch.sum(alpha, 1).unsqueeze(1).expand_as(alpha) + self.eps)

        h_iI = self.aggre_items(torch.sum(alpha.unsqueeze(2).expand_as(x_ia) * x_ia, 1))
        h_iI = F.dropout(h_iI, 0.5, training=self.training)



        # social aggregation
        q_a_s = self.item_emb(u_user_item_pad[:,:,:,0])   # B x maxu_len x maxi_len x emb_d
        mask_s = torch.where(u_user_item_pad[:,:,:,0] > 0, torch.tensor([1.], device=self.d
        p_i_s = mask_s.unsqueeze(3).expand_as(q_a_s) * self.user_emb(u_user_pad).unsqueeze(
        u_user_item_er = self.rate_emb(u_user_item_pad[:, :, :, 1])  # B x maxu_len x maxi_
        x_ia_s = self.g_v(torch.cat([q_a_s, u_user_item_er], dim=3).view(-1, 2 * self.emb_d

        alpha_s = self.user_items_att_s1(torch.cat([self.w4(x_ia_s), self.w4(p_i_s)], dim =
```

```
        alpha_s = torch.exp(alpha_s) * mask_s
        alpha_s = alpha_s / (torch.sum(alpha_s, 2).unsqueeze(2).expand_as(alpha_s) + self.e


        h_oI_temp = torch.sum(alpha_s.unsqueeze(3).expand_as(x_ia_s) * x_ia_s, 2)    # B x 
        h_oI = self.aggre_items_s1(h_oI_temp.view(-1, self.emb_dim)).view(h_oI_temp.size())
        h_oI = F.dropout(h_oI, p=0.5, training=self.training)

        ## calculate attention scores in social aggregation
        mask_su = torch.where(u_user_pad > 0, torch.tensor([1.], device=self.device), torch

        beta = self.user_users_att_s2(torch.cat([self.w5(h_oI), self.w5(self.user_emb(u_use
        beta = torch.exp(beta) * mask_su
        beta = beta / (torch.sum(beta, 1).unsqueeze(1).expand_as(beta) + self.eps)
        h_iS = self.aggre_neigbors_s2(torch.sum(beta.unsqueeze(2).expand_as(h_oI) * h_oI, 1
        h_iS = F.dropout(h_iS, p=0.5, training=self.training)




        ## learning user latent factor
        h =  self.combine_mlp(torch.cat([h_iI, h_iS], dim = 1))

        return h


class _ItemModel(nn.Module):
    '''Item modeling to learn item latent factors.
    '''
    def __init__(self, emb_dim, user_emb, item_emb, rate_emb):
        super(_ItemModel, self).__init__()
        self.emb_dim = emb_dim
        self.user_emb = user_emb
        self.item_emb = item_emb
        self.rate_emb = rate_emb

        self.w1 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w2 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w3 = nn.Linear(self.emb_dim, self.emb_dim)
        self.w4 = nn.Linear(self.emb_dim, self.emb_dim)

        self.g_u = _MultiLayerPercep(2 * self.emb_dim, self.emb_dim)
        self.g_v = _MultiLayerPercep(2 * self.emb_dim, self.emb_dim)

        self.item_users_att_i = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_users_i = _Aggregation(self.emb_dim, self.emb_dim)

        self.item_users_att = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_users = _Aggregation(self.emb_dim, self.emb_dim)
```

```
        self.i_friends_att = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_i_friends = _Aggregation(self.emb_dim, self.emb_dim)

        self.if_friends_att = _MultiLayerPercep(2 * self.emb_dim, 1)
        self.aggre_if_friends = _Aggregation(self.emb_dim, self.emb_dim)

        self.combine_mlp = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(3* self.emb_dim, 2*self.emb_dim, bias = True),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(2*self.emb_dim, self.emb_dim, bias = True),
            nn.ReLU()
        )

        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        # used for preventing zero div error when calculating softmax score
        self.eps = 1e-10

    def forward(self, iids, i_user_pad):
        # user aggregation
        p_t = self.user_emb(i_user_pad[:,:,0])
        mask_i = torch.where(i_user_pad[:,:,0] > 0, torch.tensor([1.], device=self.device),
        i_user_er = self.rate_emb(i_user_pad[:,:,1])
        f_jt = self.g_u(torch.cat([p_t, i_user_er], dim = 2).view(-1, 2 * self.emb_dim)).vi

        # calculate attention scores in user aggregation
        q_j = mask_i.unsqueeze(2).expand_as(f_jt) * self.item_emb(iids).unsqueeze(1).expand_

        miu = self.item_users_att_i(torch.cat([self.w1(f_jt), self.w1(q_j)], dim = 2).view(
        miu = torch.exp(miu) * mask_i
        miu = miu / (torch.sum(miu, 1).unsqueeze(1).expand_as(miu) + self.eps)
        z_j = self.aggre_users_i(torch.sum(miu.unsqueeze(2).expand_as(f_jt) * self.w1(f_jt)
        z_j = F.dropout(z_j, p=0.5, training=self.training)


        return z_j


class GraphRec(nn.Module):
    '''GraphRec model proposed in the paper Graph neural network for social recommendation

    Args:
        number_users: the number of users in the dataset.
        number_items: the number of items in the dataset.
        num_rate_levels: the number of rate levels in the dataset.
        emb_dim: the dimension of user and item embedding (default = 64).

    '''
    def __init__(self, num_users, num_items, num_rate_levels, emb_dim = 64):
```

```
        super(GraphRec, self).__init__()
        self.num_users = num_users
        self.num_items = num_items
        self.num_rate_levels = num_rate_levels
        self.emb_dim = emb_dim
        self.user_emb = nn.Embedding(self.num_users, self.emb_dim, padding_idx = 0)
        self.item_emb = nn.Embedding(self.num_items, self.emb_dim, padding_idx = 0)
        self.rate_emb = nn.Embedding(self.num_rate_levels, self.emb_dim, padding_idx = 0)

        self.user_model = _UserModel(self.emb_dim, self.user_emb, self.item_emb, self.rate_

        self.item_model = _ItemModel(self.emb_dim, self.user_emb, self.item_emb, self.rate_


        self.rate_pred = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(2* self.emb_dim, self.emb_dim, bias = True),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(self.emb_dim, self.emb_dim // 4),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(self.emb_dim // 4, 1)
        )





    def forward(self, uids, iids, u_item_pad, u_user_pad, u_user_item_pad, i_user_pad):
        '''
        Args:
            uids: the user id sequences.
            iids: the item id sequences.
            u_item_pad: the padded user-item graph.
            u_user_pad: the padded user-user graph.
            u_user_item_pad: the padded user-user-item graph.
            i_user_pad: the padded item-user graph.

        Shapes:
            uids: (B).
            iids: (B).
            u_item_pad: (B, ItemSeqMaxLen, 2).
            u_user_pad: (B, UserSeqMaxLen).
            u_user_item_pad: (B, UserSeqMaxLen, ItemSeqMaxLen, 2).
            i_user_pad: (B, UserSeqMaxLen, 2).

        Returns:
            the predicted rate scores of the user to the item.
        '''
```

```
        h = self.user_model(uids, u_item_pad, u_user_pad, u_user_item_pad)
        z = self.item_model(iids, i_user_pad)

        r_ij = self.rate_pred(torch.cat([h,z], dim = 1))

        return r_ij
```

```
import torch
import random

truncate_len = 30
truncate_len_i = 10

"""
Ciao dataset info:
Avg number of items rated per user: 38.3
Avg number of users interacted per user: 2.7
Avg number of users connected per item: 16.4
"""

def collate_fn(batch_data):
    """This function will be used to pad the graph to max length in the batch
       It will be used in the Dataloader
    """
    uids, iids, labels = [], [], []
    u_items, u_users, u_users_items, i_users = [], [], [], []
    u_items_len, u_users_len, i_users_len = [], [], []
    count = 0
    for data, u_items_u, u_users_u, u_users_items_u, i_users_i in batch_data:

        (uid, iid, label) = data
        uids.append(uid)
        iids.append(iid)
        labels.append(label)

        # user-items
        if len(u_items_u) <= truncate_len:
            u_items.append(u_items_u)
        else:
            u_items.append(random.sample(u_items_u, truncate_len))
        u_items_len.append(min(len(u_items_u), truncate_len))


        # user-users and user-users-items
        if len(u_users_u) < truncate_len:
            tmp_users = [item for item in u_users_u]
            tmp_users.append(uid)
            u_users.append(tmp_users)
            u_u_items = []
            for uui in u_users_items_u:
                if len(uui) < truncate_len:
                    u_u_items.append(uui)
                else:
                    u_u_items.append(random.sample(uui, truncate_len))
            # self -loop
            u_u_items.append(u_items[-1])
            u_users_items.append(u_u_items)
```

```
        else:
            sample_index = random.sample(list(range(len(u_users_u))), truncate_len-1)
            tmp_users = [u_users_u[si] for si in sample_index]
            tmp_users.append(uid)
            u_users.append(tmp_users)

            u_users_items_u_tr = [u_users_items_u[si] for si in sample_index]
            u_u_items = []
            for uui in u_users_items_u_tr:
                if len(uui) < truncate_len:
                    u_u_items.append(uui)
                else:
                    u_u_items.append(random.sample(uui, truncate_len))
            u_u_items.append(u_items[-1])
            u_users_items.append(u_u_items)

        u_users_len.append(min(len(u_users_u)+1, truncate_len))

        # item-users
        if len(i_users_i) <= truncate_len:
            i_users.append(i_users_i)
        else:
            i_users.append(random.sample(i_users_i, truncate_len))
        i_users_len.append(min(len(i_users_i), truncate_len))


    batch_size = len(batch_data)

    # padding
    u_items_maxlen = max(u_items_len)
    u_users_maxlen = max(u_users_len)
    i_users_maxlen = max(i_users_len)


    u_item_pad = torch.zeros([batch_size, u_items_maxlen, 2], dtype=torch.long)
    for i, ui in enumerate(u_items):
        u_item_pad[i, :len(ui), :] = torch.LongTensor(ui)

    u_user_pad = torch.zeros([batch_size, u_users_maxlen], dtype=torch.long)
    for i, uu in enumerate(u_users):
        u_user_pad[i, :len(uu)] = torch.LongTensor(uu)


    u_user_item_pad = torch.zeros([batch_size, u_users_maxlen, u_items_maxlen, 2], dtype=tor
    for i, uu_items in enumerate(u_users_items):
        for j, ui in enumerate(uu_items):
            u_user_item_pad[i, j, :len(ui), :] = torch.LongTensor(ui)

    i_user_pad = torch.zeros([batch_size, i_users_maxlen, 2], dtype=torch.long)
    for i, iu in enumerate(i_users):
        i_user_pad[i, :len(iu), :] = torch.LongTensor(iu)
```

```
        return torch.LongTensor(uids), torch.LongTensor(iids), torch.FloatTensor(labels), \
                u_item_pad, u_user_pad, u_user_item_pad, i_user_pad
```

```
import random
import pickle
import argparse
import numpy as np
import pandas as pd
from tqdm import tqdm
from scipy.io import loadmat

random.seed(1234)

workdir = 'datasets/'

# parser = argparse.ArgumentParser()
# parser.add_argument('--dataset', default='Epinions', help='dataset name: Ciao/Epinions')
# parser.add_argument('--test_prop', default=0.1, help='the proportion of data used for tes
# args = parser.parse_args()

# # load data
# if args.dataset == 'Ciao':
#   click_f = loadmat(workdir + 'Ciao/rating.mat')['rating']
#   trust_f = loadmat(workdir + 'Ciao/trustnetwork.mat')['trustnetwork']
# elif args.dataset == 'Epinions':
# click_f = np.loadtxt('./datasets/Epinions/ratings_data.txt', dtype = np.int32)
# trust_f = np.loadtxt('./datasets/Epinions/trust_data.txt', dtype = np.int32)
# else:
#   pass

click_list = []
trust_list = []

u_items_list = []
u_users_list = []
u_users_items_list = []
i_users_list = []

pos_u_items_list = []
pos_i_users_list = []

user_count = 0
item_count = 0
rate_count = 0

for s in click_f:
    uid = s[0]
    iid = s[1]
    label = s[2]

    if uid > user_count:
        user_count = uid
    if iid > item_count:
        item_count = iid
```

```
        if label > rate_count:
            rate_count = label
        click_list.append([uid, iid, label])

    pos_list = []
    for i in range(len(click_list)):
        pos_list.append((click_list[i][0], click_list[i][1], click_list[i][2]))

    # remove duplicate items in pos_list because there are some cases where a user may have dif
    pos_list = list(set(pos_list))

    # filter user less than 5 items
    pos_df = pd.DataFrame(pos_list, columns = ['uid', 'iid', 'label'])
    filter_pos_list = []
    user_in_set, user_out_set = set(), set()
    for u in tqdm(range(user_count + 1)):
        hist = pos_df[pos_df['uid'] == u]
        if len(hist) < 5:
            user_out_set.add(u)
            continue
        user_in_set.add(u)
        u_items = hist['iid'].tolist()
        u_ratings = hist['label'].tolist()
        filter_pos_list.extend([(u, iid, rating) for iid, rating in zip(u_items, u_ratings)])
    print('user in and out size: ', len(user_in_set), len(user_out_set))
    print('data size before and after filtering: ', len(pos_list), len(filter_pos_list))

    # train, valid and test data split
    pos_list = filter_pos_list

    random.shuffle(pos_list)
    num_test = int(len(pos_list) * 0.2)
    test_set = pos_list[:num_test]
    valid_set = pos_list[num_test:2 * num_test]
    train_set = pos_list[2 * num_test:]


    print('Train samples: {}, Valid samples: {}, Test samples: {}, Total samples: {}'.format(le


    pos_df = pd.DataFrame(pos_list, columns = ['uid', 'iid', 'label'])
    train_df = pd.DataFrame(train_set, columns = ['uid', 'iid', 'label'])
    valid_df = pd.DataFrame(valid_set, columns = ['uid', 'iid', 'label'])
    test_df = pd.DataFrame(test_set, columns = ['uid', 'iid', 'label'])

    click_df = pd.DataFrame(click_list, columns = ['uid', 'iid', 'label'])
    train_df = train_df.sort_values(axis = 0, ascending = True, by = 'uid')
    pos_df = pos_df.sort_values(axis = 0, ascending = True, by = 'uid')


    for u in tqdm(range(user_count + 1)):
```

```
    hist = train_df[train_df['uid'] == u]
    u_items = hist['iid'].tolist()
    u_ratings = hist['label'].tolist()
    if u_items == []:
        u_items_list.append([(0, 0)])
    else:
        u_items_list.append([(iid, rating) for iid, rating in zip(u_items, u_ratings)])



train_df = train_df.sort_values(axis = 0, ascending = True, by = 'iid')


userful_item_set = set()
for i in tqdm(range(item_count + 1)):
    hist = train_df[train_df['iid'] == i]
    i_users = hist['uid'].tolist()
    i_ratings = hist['label'].tolist()
    if i_users == []:
        i_users_list.append([(0, 0)])
    else:
        i_users_list.append([(uid, rating) for uid, rating in zip(i_users, i_ratings)])
        userful_item_set.add(i)

print('item size before and after filtering: ', item_count, len(userful_item_set))




count_f_origin, count_f_filter = 0,0
for s in trust_f:
    uid = s[0]
    fid = s[1]
    count_f_origin += 1
    if uid > user_count or fid > user_count:
        continue
    if uid in user_out_set or fid in user_out_set:
        continue
    trust_list.append([uid, fid])
    count_f_filter += 1

print('u-u relation filter size changes: ', count_f_origin, count_f_filter)
trust_df = pd.DataFrame(trust_list, columns = ['uid', 'fid'])
trust_df = trust_df.sort_values(axis = 0, ascending = True, by = 'uid')



count_0, count_1 = 0,0
for u in tqdm(range(user_count + 1)):
    hist = trust_df[trust_df['uid'] == u]
    u_users = hist['fid'].unique().tolist()
```

```
        if u_users == []:
            u_users_list.append([0])
            u_users_items_list.append([[(0,0)]])
            count_0 += 1
        else:
            u_users_list.append(u_users)
            uu_items = []
            for uid in u_users:
                uu_items.append(u_items_list[uid])
            u_users_items_list.append(uu_items)
            count_1 += 1
    print('trust user with items size: ', count_0, count_1)


    # with open(workdir + args.dataset + '/list_filter5.pkl', 'wb') as f:
    #   pickle.dump(u_items_list, f, pickle.HIGHEST_PROTOCOL)
    #   pickle.dump(u_users_list, f, pickle.HIGHEST_PROTOCOL)
    #   pickle.dump(u_users_items_list, f, pickle.HIGHEST_PROTOCOL)
    #   pickle.dump(i_users_list, f, pickle.HIGHEST_PROTOCOL)
    #   pickle.dump((user_count, item_count, rate_count), f, pickle.HIGHEST_PROTOCOL)
```

```
100%|██████████| 944/944 [00:00<00:00, 2035.55it/s]
user in and out size:  943 1
data size before and after filtering:  100000 100000
Train samples: 60000, Valid samples: 20000, Test samples: 20000, Total samples: 100000
100%|██████████| 944/944 [00:00<00:00, 2330.38it/s]
100%|██████████| 1683/1683 [00:00<00:00, 2320.00it/s]
item size before and after filtering:  1682 1603
u-u relation filter size changes:  68264 68264
100%|██████████| 944/944 [00:00<00:00, 1617.87it/s]trust user with items size:  15 929
```

```
import os
import time
import json
import argparse
import pickle
import numpy as np
import random
from tqdm import tqdm
from os.path import join

import torch
from torch import nn
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
from torch.autograd import Variable
from torch.backends import cudnn



device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_data = GRDataset(train_set, u_items_list,  u_users_list, u_users_items_list, i_users_
valid_data = GRDataset(valid_set, u_items_list,  u_users_list, u_users_items_list, i_users_
test_data = GRDataset(test_set, u_items_list, u_users_list,  u_users_items_list, i_users_lis
train_loader = DataLoader(train_data, batch_size = 256, shuffle = True, collate_fn = collat
valid_loader = DataLoader(valid_data, batch_size = 256, shuffle = False, collate_fn = colla
test_loader = DataLoader(test_data, batch_size = 256, shuffle = False, collate_fn = collate_

model = GraphRec(user_count+1, item_count+1, rate_count+1, 64).to(device)


device
```

```
device(type='cuda')
```

```
def trainForEpoch(train_list,train_loader, model, optimizer, epoch, num_epochs, criterion,
    model.train()

    sum_epoch_loss = 0

    start = time.time()
    for i, (uids, iids, labels, u_items, u_users, u_users_items, i_users) in tqdm(enumerat
        uids = uids.to(device)
        iids = iids.to(device)
        labels = labels.to(device)
        u_items = u_items.to(device)
        u_users = u_users.to(device)
        u_users_items = u_users_items.to(device)
        i_users = i_users.to(device)

        optimizer.zero_grad()
        outputs = model(uids, iids, u_items, u_users, u_users_items, i_users)
        loss = criterion(outputs, labels.unsqueeze(1))
        loss.backward()
        optimizer.step()

        loss_val = loss.item()
        sum_epoch_loss += loss_val

        iter_num = epoch * len(train_loader) + i + 1
        train_list[0].append(loss_val)
        train_list[1].append(sum_epoch_loss / (i + 1))
        if i % log_aggr == 0:
            print('[TRAIN WWW] epoch %d/%d batch loss: %.4f (avg %.4f) (%.2f im/s)'
                % (epoch + 1, num_epochs, loss_val, sum_epoch_loss / (i + 1),
                  len(uids) / (time.time() - start)))
        start = time.time()
    return train_list


def validate(valid_loader, model):
    model.eval()
    errors = []
    with torch.no_grad():
        for uids, iids, labels, u_items, u_users, u_users_items, i_users in tqdm(valid_loa
            uids = uids.to(device)
            iids = iids.to(device)
            labels = labels.to(device)
            u_items = u_items.to(device)
            u_users = u_users.to(device)
            u_users_items = u_users_items.to(device)
            i_users = i_users.to(device)

            preds = model(uids, iids, u_items, u_users, u_users_items, i_users)

            error = torch.abs(preds.squeeze(1) - labels)
```

```
            errors.extend(error.data.cpu().numpy().tolist())
optimizer = optim.RMSprop(model.parameters(), lr=0.001, weight_decay=1e-4)
criterion = nn.MSELoss()
scheduler = StepLR(optimizer, step_size = 30, gamma = 0.5)
train_list = [[],[]]
valid_loss_list, test_loss_list = [],[]
fn = 'graphrec'
for epoch in tqdm(range(30)):
    # train for one epoch
    scheduler.step(epoch = epoch)
    trainForEpoch(train_list,train_loader, model, optimizer, epoch, 30, criterion, log_aggr

    mae, rmse = validate(valid_loader, model)
    valid_loss_list.append([mae, rmse])

    test_mae, test_rmse = validate(test_loader, model)
    test_loss_list.append([test_mae, test_rmse])

    # store best loss and save a model checkpoint
    ckpt_dict = {
        'epoch': epoch + 1,
        'state_dict': model.state_dict(),
        'optimizer': optimizer.state_dict()
    }

    #torch.save(ckpt_dict, '%s/random_latest_checkpoint.pth.tar' %fn)

    if epoch == 0:
        best_mae = mae
    elif mae < best_mae:
        best_mae = mae
        print(ckpt_dict)

    # print('Epoch {} validation: MAE: {:.4f}, RMSE: {:.4f}, Best MAE: {:.4f}, test_MAE: {:
```