**Introduction to Recommender System (Spring 2023)**

**Homework #4 (150 pts, Due date: May 17)**

**Student ID 2022712151**

**Name 정지원**

**Instruction: You should submit your code and report to i-campus. Follow the submission format below.**

* **RS\_HW4\_STUDENT\_ID\_NAME.zip: Compress 1) your code (models/ folder and code** **for the model that scores the highest in the kaggle competition) and 2) the document.**

**NOTE 1**: You should write your code **only in ‘EDIT HERE.’**

**NOTE 2:** You need to **install the latest version** of Python, NumPy, PyTorch, Scikit-learn (sklearn), Pandas, tqdm, and Matplotlib libraries.

**NOTE 3**: You should **delete the ‘saves’ directory** in your submission zip file.

**(1) [30 pts]** We provide datasets and template code in Python. Fill out the codes for LightGCN using a reference code ‘Neural\_Graph\_CF/models/MF\_implicit.py’. You can run the models with the code ‘0\_main.py.’

**(a)** **[15 pts]** Write your code to implement the Light Graph Convolutional Network model in ‘Neural\_Graph \_CF/models/LightGCN\_implicit.py’. Given the connected neighbors, the light graph convolutional operations, the loss function are defined as follows:

where , , . and denote a set of connected neighbors, the observed ratings of user , and the binary cross entropy, respectively.

**Note: Write your code. You also have to submit your code to i-campus.**

**Answer:**

|  |
| --- |
| *import* torch  *import* torch.nn *as* nn  *import* torch.nn.functional *as* F  *import* torch.optim *as* optim  *from* time *import* time  *import* numpy *as* np  *import* scipy.sparse *as* sp  *from* IPython *import* embed  *from* utils *import* eval\_implicit  class LightGCN\_implicit(nn.Module):  def \_\_init\_\_(self, train, valid, learning\_rate, regs, batch\_size, num\_epochs, emb\_size, num\_layers, node\_dropout, device='cpu'):  super(LightGCN\_implicit, *self*).\_\_init\_\_()    *self*.train\_data = train  *self*.valid\_data = valid  *self*.train\_mat = sp.csr\_matrix(train)  *self*.valid\_mat = sp.csr\_matrix(valid)  *self*.num\_users, *self*.num\_items = *self*.train\_mat.shape  *self*.R = sp.csr\_matrix(train)  *self*.norm\_adj = *self*.create\_adj\_mat()  *self*.learning\_rate = learning\_rate  *self*.device = device  *self*.emb\_size = emb\_size  *self*.num\_layers = num\_layers  *self*.batch\_size = batch\_size  *self*.num\_epochs = num\_epochs  *self*.node\_dropout = node\_dropout  *self*.decay = regs    *self*.embedding\_dict = *self*.init\_weight()  *self*.optimizer = optim.Adam(*self*.parameters(), lr=*self*.learning\_rate)  *self*.sparse\_norm\_adj = *self*.\_convert\_sp\_mat\_to\_sp\_tensor(*self*.norm\_adj).to(*self*.device)  *self*.to(*self*.device)  def init\_weight(self):  *# xavier init*  initializer = nn.init.xavier\_uniform\_  embedding\_dict = nn.ParameterDict({  'user\_emb': nn.Parameter(initializer(torch.empty(*self*.num\_users, *self*.emb\_size))),  'item\_emb': nn.Parameter(initializer(torch.empty(*self*.num\_items, *self*.emb\_size)))  })    *return* embedding\_dict  def \_convert\_sp\_mat\_to\_sp\_tensor(self, X):  coo = X.tocoo()  i = torch.LongTensor([coo.row, coo.col])  v = torch.from\_numpy(coo.data).float()  *return* torch.sparse.FloatTensor(i, v, coo.shape)  def sparse\_dropout(self, x, rate, noise\_shape):  random\_tensor = 1 - rate  random\_tensor += torch.rand(noise\_shape).to(x.device)  dropout\_mask = torch.floor(random\_tensor).type(torch.bool)  i = x.\_indices()  v = x.\_values()  i = i[:, dropout\_mask]  v = v[dropout\_mask]  out = torch.sparse.FloatTensor(i, v, x.shape).to(x.device)  *return* out \* (1. / (1 - rate))  def rating(self, u\_g\_embeddings, pos\_i\_g\_embeddings):  *return* torch.matmul(u\_g\_embeddings, pos\_i\_g\_embeddings.t())  def forward(self, users, pos\_items, neg\_items, drop\_flag=False):  A\_hat = *self*.sparse\_dropout(*self*.sparse\_norm\_adj,  *self*.node\_dropout,  *self*.sparse\_norm\_adj.\_nnz()) *if* drop\_flag *else* *self*.sparse\_norm\_adj  ego\_embeddings = torch.cat([*self*.embedding\_dict['user\_emb'],  *self*.embedding\_dict['item\_emb']], 0)  all\_embeddings = [ego\_embeddings]  *'''*  *Implement GCN process*  *'''*  *# ========================= EDIT HERE ========================*  *# Light Graph Convolutional Operations*  *for* k *in* range(*self*.num\_layers):  *# Message Aggregation*  side\_embeddings = torch.sparse.mm(A\_hat, ego\_embeddings)  sum\_embeddings =F.leaky\_relu(side\_embeddings)  *# Store k-th embeddings*  ego\_embeddings = sum\_embeddings  all\_embeddings += [ego\_embeddings]  *# ========================= EDIT HERE ========================*  all\_embeddings = torch.stack(all\_embeddings, 1)  final\_embeddings = torch.mean(all\_embeddings, 1)  u\_g\_embeddings = final\_embeddings[:*self*.num\_users, :]  i\_g\_embeddings = final\_embeddings[*self*.num\_users:, :]  u\_g\_embeddings = u\_g\_embeddings[users, :]  pos\_i\_g\_embeddings = i\_g\_embeddings[pos\_items, :]  neg\_i\_g\_embeddings = i\_g\_embeddings[neg\_items, :]  *return* u\_g\_embeddings, pos\_i\_g\_embeddings, neg\_i\_g\_embeddings, i\_g\_embeddings  def fit(self):  user\_idx = np.arange(*self*.num\_users)  *for* epoch *in* range(*self*.num\_epochs):  epoch\_loss = 0.0  *self*.train()  np.random.RandomState(12345).shuffle(user\_idx)  batch\_num = int(len(user\_idx) / *self*.batch\_size) + 1  *for* batch\_idx *in* range(batch\_num):  batch\_users = user\_idx[batch\_idx\**self*.batch\_size:(batch\_idx+1)\**self*.batch\_size]  batch\_matrix = torch.FloatTensor(*self*.train\_mat[batch\_users, :].toarray()).to(*self*.device)  batch\_users = torch.LongTensor(batch\_users).to(*self*.device)  batch\_loss = *self*.train\_model\_per\_batch(batch\_matrix, batch\_users)  *if* torch.isnan(batch\_loss):  print('Loss NAN. Train finish.')  *break*    epoch\_loss += batch\_loss    *if* epoch % 20 == 0:  *with* torch.no\_grad():  *self*.eval()    top\_k=50  print("[LightGCN] epoch %d, loss: %f"%(epoch, epoch\_loss))  prec, recall, ndcg = eval\_implicit(*self*, *self*.train\_data, *self*.valid\_data, top\_k)  print(f"(LightGCN VALID) prec@{top\_k} {prec}, recall@{top\_k} {recall}, ndcg@{top\_k} {ndcg}")  *self*.train()  def train\_model\_per\_batch(self, train\_matrix, batch\_users, pos\_items=0, neg\_items=0):  *self*.optimizer.zero\_grad()  u\_g\_embeddings, pos\_i\_g\_embeddings, neg\_i\_g\_embeddings, i\_g\_embeddings = *self*.forward(batch\_users, 0, 0)  *# Implement output and loss function (binary CE loss)*  *# ========================= EDIT HERE ========================*  output = *self*.rating(u\_g\_embeddings, pos\_i\_g\_embeddings)  loss = F.binary\_cross\_entropy\_with\_logits(output, torch.ones\_like(output))  loss += F.binary\_cross\_entropy\_with\_logits(output, torch.zeros\_like(output))    neg\_output = *self*.rating(u\_g\_embeddings, neg\_i\_g\_embeddings)  loss += F.binary\_cross\_entropy\_with\_logits(neg\_output, torch.zeros\_like(neg\_output))  *for* embeeding *in* i\_g\_embeddings:  loss += *self*.decay \* torch.mean(embeeding.pow(2))  *# ========================= EDIT HERE ========================*  loss.backward()  *self*.optimizer.step()  *return* loss  def predict(self, user\_ids, item\_ids):  *with* torch.no\_grad():  u\_g\_embeddings, \_, \_, i\_g\_embeddings = *self*.forward(user\_ids, 0, 0)  *# ========================= EDIT HERE ========================*  output = *self*.rating(u\_g\_embeddings, i\_g\_embeddings)    *# ========================= EDIT HERE ========================*    predict\_ = output.detach().cpu().numpy()  *return* predict\_[item\_ids]  def create\_adj\_mat(self):  adj\_mat = sp.dok\_matrix((*self*.num\_users + *self*.num\_items, *self*.num\_users + *self*.num\_items), dtype=np.float32)  adj\_mat = adj\_mat.tolil()  R = sp.csr\_matrix(*self*.R).tolil()  adj\_mat[:*self*.num\_users, *self*.num\_users:] = R  adj\_mat[*self*.num\_users:, :*self*.num\_users] = R.T  adj\_mat = adj\_mat.todok()  *# D^-1/2 \* A \* D^-1/2*  rowsum = np.array(adj\_mat.sum(axis=1))  d\_inv = np.power(rowsum, -0.5).flatten()  d\_inv[np.isinf(d\_inv)] = 0.  d\_mat = sp.diags(d\_inv)    norm\_adj = d\_mat.dot(adj\_mat).dot(d\_mat)  norm\_adj = norm\_adj.tocsr()  *return* norm\_adj  def \_convert\_sp\_mat\_to\_sp\_tensor(self, X):  coo = X.tocoo().astype(np.float32)  row = torch.Tensor(coo.row).long()  col = torch.Tensor(coo.col).long()  index = torch.stack([row, col])  data = torch.FloatTensor(coo.data)  *return* torch.sparse.FloatTensor(index, data, torch.Size(coo.shape)) |

**(b)** **[15 pts]** Draw the plots of Precision, Recall, and NDCG by adjusting the cutoff of MF and LightGCN on the ‘movielens\_100k’ and ‘naver\_movie\_dataset\_small’ datasets. Run ‘1\_cutoff.py’ to run the code.

**Note: Draw your plots and explain the results in short (3-5) lines.**

**Answer:**

|  |
| --- |
| **By adjusting the cutoff MF and LightGCN on the datasets, we can observe the performance of both models at different levels of recommendation list legenths. As the cutoff increases, Precision may decrease since more items are recommended, increasing the chance of including irrelevant items. However, Recall and NDCG may increases as more relevant items are likely to be included in the longer recommendation lists.** |

**(2) [60 pts]** We provide a dataset and template code in Python. Fill out the codes for SASRec, CORE, and NISER using a reference code ‘Session\_Rec/models/SRGNN.py’. You can run the models with the code ‘0\_main.py.’

**(a)** **[15 pts]** Write your code to implement the SASRec model in ‘SessionRec/models/SASRec.py’. Given the item sequences , the embedding layer, the self-attention layer are defined as follows:

where and denote learnable item embedding and positional embedding, respectively.

**Note: Write your code. You also have to submit your code to i-campus.**

**Answer:**

|  |
| --- |
| def forward(self, inputs):  *# inputs: (batch\_size, maxlen)*  seqs = *self*.item\_emb(inputs)    *# Item Embedding + Positional Embedding*  *# ========================= EDIT HERE ========================*  positions = torch.arange(inputs.shape[1], device=*self*.device).unsqueeze(0)  seqs += *self*.pos\_emb(positions)  seqs = *self*.emb\_dropout(seqs)  *# ========================= EDIT HERE ========================*    timeline\_mask = (inputs == 0)  seqs = seqs \* ~timeline\_mask.unsqueeze(-1) *# broadcast in last dim*  tl = seqs.shape[1]  attention\_mask = ~torch.tril(torch.ones((tl, tl), dtype=torch.bool, device=*self*.device))  *# Self-attention block*  *for* i *in* range(len(*self*.attention\_layers)):  *# ========================= EDIT HERE ========================*  *# layer normalization*  seqs = *self*.attention\_layernorms[i](seqs)  *# self-attention*  attn\_output, \_ = *self*.attention\_layers[i](seqs, seqs, seqs, attn\_mask=attention\_mask)  *# residual connection*  seqs = seqs + attn\_output  *# layer normalization*  seqs = *self*.forward\_layernorms[i](seqs)  *# position-wise feed-forward*  seqs = *self*.forward\_layers[i](seqs)    *# ========================= EDIT HERE ========================*  seqs = seqs \* ~timeline\_mask.unsqueeze(-1)  *# (batch, maxlen, hidden\_dim)*  hidden\_feats = *self*.last\_layernorm(seqs)  final\_feat = hidden\_feats[:, -1, :]  item\_embs = *self*.item\_emb.weight[1:].T  logits = final\_feat @ item\_embs    *return* logits |

**(b)** **[15 pts]** Write your code to implement the CORE model in ‘Session\_Rec/models/CORE.py’. Given the item sequence , the output of Representation-Consistent Encoder (RCE) with SASRec (Transformer) and the Robust Distance Measuring (RDM) loss function are defined as follows:

where is a vector of learnable parameters, , , and denote the length of the session , the item embedding of the item , and the output of SASRec with respect to the session , respectively. Additionally, , , , and denote the target item of the session , a set of whole items, a hyperparameter for temperature, and a matrix of whole item’s embedding, respectively.

**Note 1: To calculate , you should remove the effects of the zero padding in item sequence .**

**Note 2: Write your code here. You also have to submit your code to i-campus.**

**Answer:**

|  |
| --- |
| def Transformer(self, item\_emb, inputs):  *# NOTE: Copy and paste your SASRec code*    *# Item Embedding + Positional Embedding*  *# ========================= EDIT HERE ========================*  seqs = *self*.item\_emb(inputs)  positions = torch.arange(inputs.shape[1], device=*self*.device).unsqueeze(0).repeat(inputs.size(0),1)  seqs += *self*.pos\_emb(positions)  seqs = *self*.emb\_dropout(seqs)    *# ========================= EDIT HERE ========================*    timeline\_mask = ~(inputs == 0)  seqs = seqs \* timeline\_mask.unsqueeze(-1) *# broadcast in last dim*  tl = seqs.shape[1]  attention\_mask = ~torch.tril(torch.ones((tl, tl), dtype=torch.bool, device=*self*.device))  *# Self-attention block*  *for* i *in* range(len(*self*.attention\_layers)):  *# ========================= EDIT HERE ========================*  *# layer normalization*  seqs = *self*.attention\_layernorms[i](seqs)  *# self-attention*  attn\_output,\_=*self*.attention\_layers[i](seqs,seqs,seqs,attn\_mask=attention\_mask)  *# residual connection*  seqs = seqs + attn\_output  *# layer normalization*  seqs = *self*.forward\_layernorms[i](seqs)  *# position-wise feed-forward*  seqs = *self*.forward\_layernorms[i](seqs)  *# ========================= EDIT HERE ========================*  seqs = seqs \* timeline\_mask.unsqueeze(-1)  *# (batch, maxlen, hidden\_dim)*  hidden\_feats = *self*.last\_layernorm(seqs)    *return* hidden\_feats, timeline\_mask    def RCE(self, dnn\_hidden\_feats, item\_emb, timeline\_mask):  *# ========================= EDIT HERE ========================*  *# wF^T*  wF = *self*.fn(dnn\_hidden\_feats).squeeze(-1)  *# remove effects of zero padded sequences*  wF = wF.masked\_fill(~timeline\_mask, float('-inf'))  *# softmax*  alpha = F.softmax(wF/*self*.temperature, dim=-1)  hidden\_feats = torch.bmm(alpha.unsqueeze(1), dnn\_hidden\_feats).squeeze(1)  *# ========================= EDIT HERE ========================*  hidden\_feats = F.normalize(hidden\_feats, dim=-1)    *return* hidden\_feats  def RDM(self, hidden\_feats, item\_embs):  logits = None  item\_embs = F.normalize(item\_embs, dim=-1)  *# ========================= EDIT HERE ========================*  logits = hidden\_feats @ item\_embs / *self*.temperature  *# ========================= EDIT HERE ========================*  *return* logits |

**(c)** **[15 pts]** Write your code to implement the NISER model in ‘Session\_Rec/models/NISER.py’. Given the item sequences , the item embedding and the scores are normalized by the L2 norm.

**Note: Write your code. You also have to submit your code to i-campus.**

**Answer:**

|  |
| --- |
| *import* os  *import* math  *import* datetime  *import* time  *from* tqdm *import* tqdm  *import* numpy *as* np  *import* pandas *as* pd  *import* torch  *from* torch *import* nn  *import* torch.nn.functional *as* F  class NISER(nn.Module):  def \_\_init\_\_(self, hidden\_size, batch\_size, nonhybrid, step, lr, l2, lr\_dc, lr\_dc\_step, n\_node, max\_pos):  super(NISER, *self*).\_\_init\_\_()  *self*.hidden\_size = hidden\_size  *self*.batch\_size = batch\_size  *self*.nonhybrid = nonhybrid  *self*.step = step  *self*.lr = lr  *self*.l2 = l2  *self*.lr\_dc = lr\_dc  *self*.lr\_dc\_step = lr\_dc\_step  *self*.n\_node = n\_node  *self*.max\_pos = max\_pos    *self*.build\_graph()  *self*.reset\_parameters()  *self*.dropout = nn.Dropout(0.3)  def build\_graph(self):  *self*.embedding = nn.Embedding(*self*.n\_node, *self*.hidden\_size)  *self*.gnn = GNN(*self*.hidden\_size, step=*self*.step)  *self*.pos\_embedding = nn.Embedding(*self*.max\_pos, *self*.hidden\_size)    *self*.linear\_one = nn.Linear(*self*.hidden\_size, *self*.hidden\_size, bias=True)  *self*.linear\_two = nn.Linear(*self*.hidden\_size, *self*.hidden\_size, bias=True)  *self*.linear\_three = nn.Linear(*self*.hidden\_size, 1, bias=False)    *self*.linear\_W = nn.Linear(*self*.hidden\_size \* 2, *self*.hidden\_size, bias=True)    *self*.loss\_function = nn.CrossEntropyLoss()  *self*.optimizer = torch.optim.Adam(*self*.parameters(), lr=*self*.lr, weight\_decay=*self*.l2)  *self*.scheduler = torch.optim.lr\_scheduler.StepLR(*self*.optimizer, step\_size=*self*.lr\_dc\_step, gamma=*self*.lr\_dc)    def reset\_parameters(self):  stdv = 1.0 / math.sqrt(*self*.hidden\_size)  *for* weight *in* *self*.parameters():  weight.data.uniform\_(-stdv, stdv)  def compute\_scores(self, hidden, mask):  *# hidden: GNN embedding*  batch\_size = hidden.shape[0]  length = hidden.shape[1]  pos\_emb = *self*.pos\_embedding.weight[:length]  pos\_emb = pos\_emb.unsqueeze(0).repeat(batch\_size, 1, 1)  hidden = hidden + pos\_emb  *# v\_t: last item vector*  ht = hidden[torch.arange(mask.shape[0]).long(), torch.sum(mask, 1) - 1] *# batch\_size x latent\_size*    *# a: alpha\_j \* v\_j*  q1 = *self*.linear\_one(ht).view(ht.shape[0], 1, ht.shape[1]) *# batch\_size x 1 x latent\_size*  q2 = *self*.linear\_two(hidden) *# batch\_size x seq\_length x latent\_size*  alpha = *self*.linear\_three(torch.sigmoid(q1 + q2))  a = torch.sum(alpha \* hidden \* mask.view(mask.shape[0], -1, 1).float(), 1)  *# final scoring*  *# # ========================= EDIT HERE ========================*  *if* not *self*.nonhybrid:  item\_embeddings = *self*.embedding.weight  item\_embeddings\_norm = item\_embeddings / torch.norm(item\_embeddings, dim=1).unsqueeze(1)  a\_norm = a / torch.norm(a, dim=1).unsqueeze(1)      scores = torch.matmul(a\_norm, item\_embeddings\_norm.transpose(1, 0))    *else*:  scores = torch.matmul(a,*self*.linear\_W.weight.transpose(1, 0))  *# ========================= EDIT HERE ========================*    scores \*= 12 *# scaling factor*  *return* scores  def forward(self, inputs, A):  item\_embs = *self*.embedding(inputs)    *# ========================= EDIT HERE ========================*  hidden = F.normalize(item\_embs, dim=-1)    *# ========================= EDIT HERE ========================*  hidden = F.normalize(*self*.gnn(A, hidden),dim=-1)    *return* hidden |

**(d)** **[15 pts]** Fill the table with the results of SASRec, CORE, SRGNN, and NISER.

**Note: Do not change the hyperparameter settings of them. You should write your code only in ‘EDIT HERE.’**

**Answer:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Models | SASRec | CORE | SRGNN | NISER |
| Recall@20 | 0.0922 | 0.4511 | 0.1247 | 0.2349 |
| MRR@20 | 0.0528 | 0.1328 | 0.0490 | 0.1194 |

**(2) [60 pts] (Kaggle competition.)** The goal is to predict top-N recommended items as the last item in each session. Please read the notes below and visit the following link: <https://www.kaggle.com/competitions/skku-rs2023-assignment-4-test>. This is a private competition, so you should only enter using the link above.

**[Notes]**

* You can make a Kaggle submission file using the function ‘save\_submission’ in ‘Session\_Rec/utils.py’. This function is used in ‘Session\_Rec/0\_main.py’ for each model.
* If you are not able to access the link above, try the following link:  
  https://www.kaggle.com/t/76212b3c5fe74dbdaa9c48d19b1db7c4
* Please submit your code for the model that scores the highest in the Kaggle competition in ‘**Kaggle/**’.

**[Scoring policy]**

The final evaluation will be made by adding the private leaderboard score and the idea score. Please write a report on your project solution with a maximum of 2 pages.

**[Competition Rules]**

* Do not cheat.
* Use Python.
* No limitation on Python libraries. (Pytorch, Tensorflow, etc.)
* You must use “**{Student ID}\_{Name}**” for your team name in the Kaggle competition.
* No late submission in the Kaggle competition.
* Any use of external data is prohibited.
* Your submission to Kaggle is limited to five times a day.

**Please write your solution for the Kaggle competition.**

|  |
| --- |
| **I used SRGNN, SASRec, CORE and NISER models and obtained the optimal hyper-parameters through grid search.**  **The best model uploaded to Kaggle is SRGNN, and the hyperparameter is in SRGNN\_session of SRGNN.py** |