

基于深度学习的序列推荐任务

摘要

本项目使用深度学习方法，基于pyTorch进行编程建模，完成了基于序列的推荐任务。

在任务完成的过程中，一共尝试了五种神经网络模型，分别是基于RNN的GRU4REC和NARM、基于GNN的SRGNN，以及两个为了更好地适应数据格式，而由本人自行进行了一些改进的两个模型GRUwATT和GGNNwATT。

训练过程中，将给定的test数据划分为训练集和验证集，根据验证集上的评价指标选择了在最优参数下的模型。更复杂的并不一定更好，在经过多次尝试后，最终选择了GRU4REC模型来得到最后的输出。

在完成任务的过程中，通过自行搭建Dataset和Dataloader（见[代码文件 process_data.py 介绍](#)），感受到了数据增强起到的重要作用。更加复杂的模型的效果并不及简单的GRU4REC，甚至相差较大，可能就是在复杂模型中由于网络结构的限制，没有使用自行创建的TrainDataset类。基于此猜测，本项目又自行对模型进行了修改，以适应增强后的数据格式，最后GRUwATT模型的结果明显好于NARM，验证了这一想法。

- [一、问题描述和定义](#)
- [二、方法介绍](#)
- [三、实验设置](#)
 - [数据集介绍](#)
 - [对比方法](#)
 - [评价指标](#)
 - [超参数设置](#)
 - [代码架构和逻辑](#)
 - [最终模型训练和结果输出](#)
- [四、实验细节](#)
 - [1. GRU4REC](#)
 - [\(1\) 数据处理](#)
 - [\(2\) 模型建立](#)
 - [\(3\) 实验细节和结果](#)
 - [2. NARM](#)
 - [3. SRGNN](#)
 - [\(1\) 数据处理](#)
 - [\(2\) 模型建立](#)
 - [\(3\) 实验细节和结果](#)
 - [4. 模型改进: GRUwATT, GGNNwATT](#)
- [五、总结和展望](#)
- [参考资料](#)

一、问题描述和定义

在基于序列的推荐任务中，模型接受一条序列（session）的输入，一条序列中包含了若干物品（item）。基于输入的序列，模型希望学习到该用户对物品的喜好特征，对用户的行为偏好进行建模，并预测该用户下一个可能喜欢的物品。

二、方法介绍

本项目主要尝试了五个基于深度学习的推荐系统模型，分别是基于RNN的GRU4REC和NARM、基于GNN的SRGNN，以及两个为了更好地适应数据格式而由我自行进行了一些改进的模型GRUwATT和GGNNwATT

三、实验设置：

数据集介绍

本次作业的数据集来自 Amazon 电商平台，包括 Beauty 和 Cellphones 两个类别。

两个数据集均已划分好训练数据和测试数据，数据中每一行为一条训练样本。其中：

- train_sessions.csv 包括两个字段：session 字段为用户的历史匿名交互序列，label 用户在实际中下一次交互的产品
- test_sessions.csv 中仅包括session字段

对比方法

项目在训练过程中，将给定的test数据划分为训练集和验证集，根据验证集上的评价指标选择了在最优参数的模型。最后比较不同模型在验证集上的最佳表现，并选择最好的模型进行最后的结果输出。

评价指标

两个关注的指标：

- HR@K: 命中率（hit ratio），表示真实label是否出现在推荐列表中
- MRR@K: 排名的倒数的平均值（mean reciprocal rank），表示真实label在推荐列表中排名靠前的程度

训练过程中，主要参考的评价指标为HR@20和MRR@20，同时输出验证集上的accuracy作为参考

最后采用模型GRU4REC，输出了该模型上的HR@5、MRR@5、HR@20和MRR@20，见[结果输出](#)

超参数设置

在每个模型的实验中都进行了一系列的超参数尝试。在最后使用的模型GRU4REC中，参数设置为：

```
num_epoch = 200
batch_size = 256
lr = 0.01
dropout = 0
embedding_size = 128
hidden_size = 128
```

代码架构和逻辑

代码目录结构如下：

```
.
|--- dataset # 数据集
|    |--- ...

|--- pretrained_model # 保存的模型，分别为两个数据集保存
|    |--- Beauty
|        |--- ...
|    |--- Cell
|        |--- ...

|--- results # 最终输出结果
|    |--- test_result_beauty.csv
|    |--- test_result_cell.csv

# 模型搭建和训练代码
|--- 0 GRU4REC.ipynb
|--- 1 SRGNN.ipynb
|--- 2 NARM.ipynb
|--- 3 GRUwATT.ipynb
|--- 4 GGNNwATT.ipynb

# 输出最终结果
|--- get_final_result.ipynb

# 数据处理
|--- process_data.py
```

各文件说明：

1. `pretrained_model`：分为Beauty和Cell两个子文件夹，保存了用test_session训练好的模型
2. `process_data.py`：
 - 对数据进行预处理，除去总出现次数小于5的item、只有一个item的session并将item编码
 - 定义了 `TrainDataset` 和 `MyDataset` 两个类，继承了PyTorch的 `Dataset` 类

`TrainDataset` 和 `TestDataset` 类的区别在于其 `__getitem__()` 方法返回的label不同，具体区别为：

- `TrainDataset`：返回的label除了数据集中给定的label外，还有该session除了第一个item外剩下的item序列，相当于做了一种适合RNN的data augmentation（灵感来源于最后一次作业的文本生成任务）

例：当session = [1, 2, 3, 4]、label=[5]时，`__getitem__()` 返回的样本是X = [1, 2, 3, 4]和y = [2, 3, 4, 5]

- `MyDataset`：返回的label就是数据集中的label，即单个item

需要注意的是，由于NARM和SRGNN模型中的attention机制需要取最后一个时间步的特征，所以得到的输出形状是[batch_size, embedding_size]，因此并不适合用 `TrainDataset` 类进行训练。

- 定义了 `get_dataloader()` 函数，返回一个PyTorch `DataLoader` 类的对象，用于在训练和测试时获取batch数据。

在 `get_dataloader()` 中进行了以下两个操作：

- 设置了 `DataLoader` 的 `batch_sampler` 属性，使得loader在抽样时，会先将样本按照session的长度进行排序，使batch中每个样本session的真实长度都相近

- 设置了 DataLoader 的 `collate_fn` 属性，令 loader 为 batch 中的样本进行零填充，使得返回的 batch 中，每个样本的 session 长度都相等，且等于该 batch 中最长 session 的长度
- 说明：关于数据加载部分的 `get_dataloader()`，我与同班的张陈卓同学进行了较多讨论，有一定可能现代码相似的情况，希望老师谅解，谢谢老师！

2. 模型训练代码文件：以下文件为不同网络结构模型的搭建和训练代码

- 0 GRU4REC.ipynb：实现了 GRU4REC 模型的搭建和训练，保存最佳模型为 `gru4rec.pt`
- 1 SRGNN.ipynb：实现了 SRGNN 模型的搭建和训练
- 2 NARM.ipynb：实现了 NARM 模型的搭建和训练
- 4 GRUwATT.ipynb：实现了 GRUwATT 模型的搭建和训练，保存最佳模型为 `GRUwATT.pt`
- 5 GGNNwATT.ipynb：实现了 GGNNwATT 模型的搭建和训练

每个文件的编写逻辑基本相同，分为模型搭建、模型训练和保存模型三个部分，其中模型训练部分包括评价指标的定义、模型训练打包（`ModelTrain` 类的定义）、超参数设置和训练过程。

调参和最终全数据训练过程都保存在 notebook 文件中。

3. `get_final_result.ipynb` 用于读取保存的模型，并输出最后的结果文件，保存在 `results` 文件夹中

最终模型训练和结果输出

在建立的五個模型上，由于 SRGNN 和 NARM 表现并不优秀，所以没有用于最终结果的输出。

最后在最优参数下，选择 GRU4REC 和 GRUwATT 两个模型，用两个数据集的 `train_session` 分别进行训练，并分别保存模型至 `pretrained_model` 文件夹。

两个模型模型在验证集上的最佳表现分别为：

	MRR@20	HR@20	accuracy
GRU4REC	0.450	50.2%	0.38
GRUwATT (lr=0.005, bs=256)	0.400	46.4%	0.35

最后采用了 GRU4REC 模型，输出最后结果

验证集上的指标为：

MRR@5	HR@5	MRR@20	HR@20
0.448	47.7%	0.450	50.2%

四、实验细节

(在代码中加入了很详细的注释，因此在报告中就不详细解释代码了！)

1. GRU4REC

(1) 数据处理

数据处理部分，把session转换为列表类型，去除了训练集中仅含一个item的session，并重新进行了编码，最后得到共10674种item，共22569条session数据，并在后续将20%的数据划分为验证集，用于查看模型性能。

(2) 模型建立

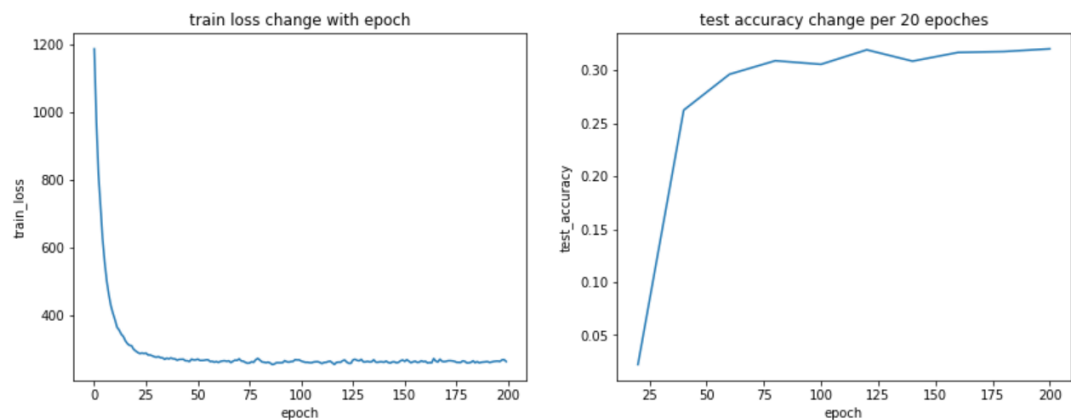
模型主体部分定义了 GRU4REC 类，细节为：

- 1. 网络结构为embedding→dropout→GRU→feed forward，共四层
- 由于文章的experiment部分指出单层GRU的效果比多层更好，所以在模型中只设置了单层GRU
- 2. 参数初始化：Xavier初始化
- 3. 需要指出的是，模型的feed forward层并不是直接映射回dim=output_size的空间，而是先回到dim=embedding_size的空间，再将返回值与embedding层的权重矩阵做内积，最后回到output_size空间。
- 4. 根据论文的experiment部分，我们还设置了一个dropout层

(3) 实验细节和结果

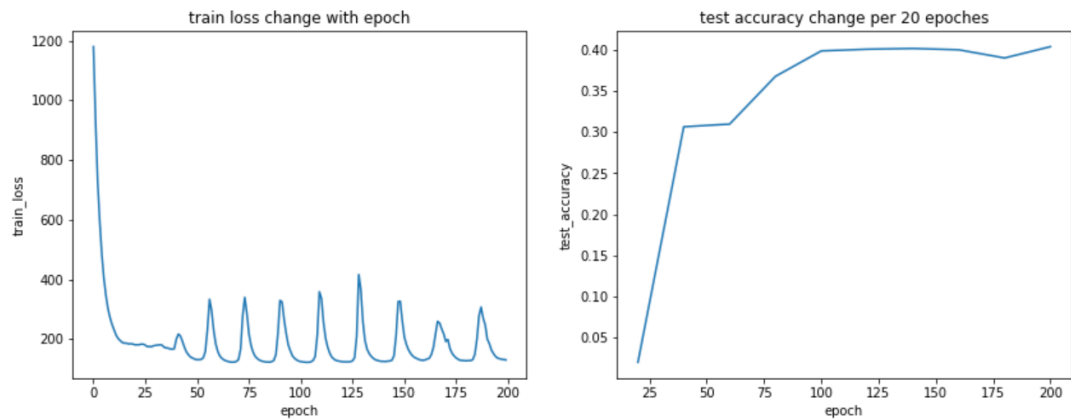
主要调参尝试如下，每次n_epoch = 200：

lr, bs, dp	MRR@20	HR@20	accuracy
0.01, 128, 0.3	0.378	49.9%	0.30



lr, bs, dp	MRR@20	HR@20	accuracy
0.01, 128, 0	0.436	49.4%	0.40

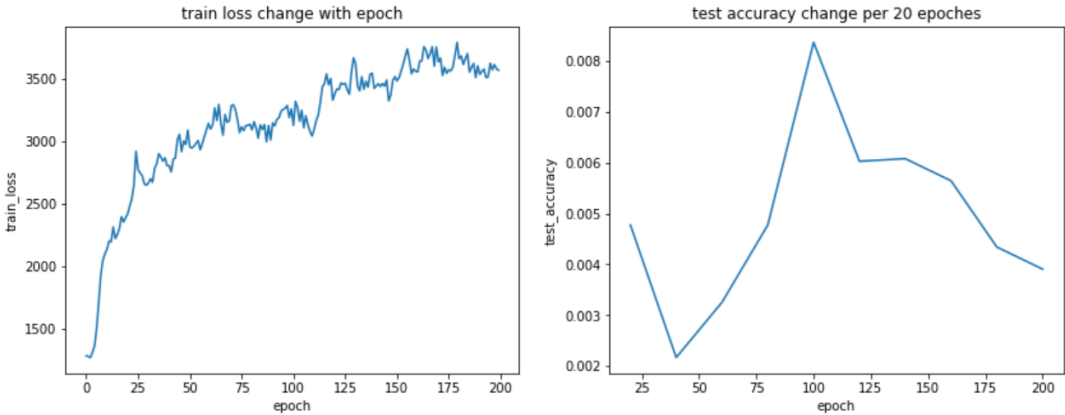
没有dropout，指标的波动变大了很多，但最终表现和没有dropout差不多



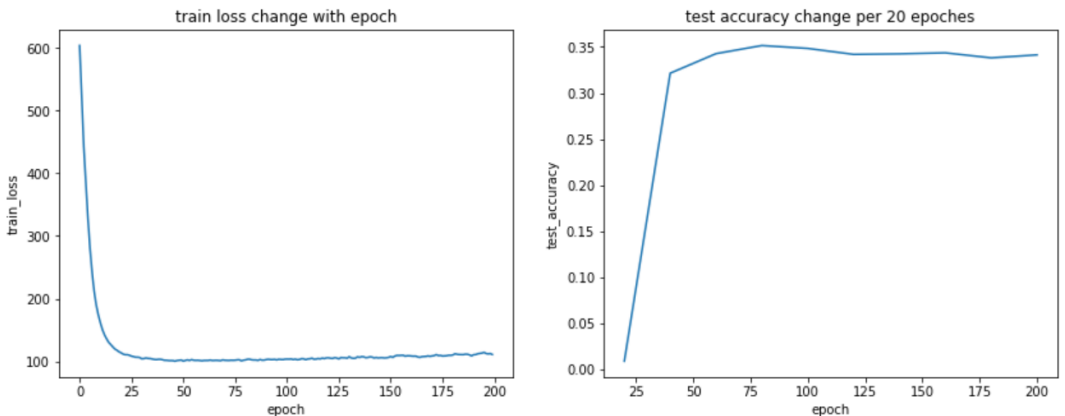
lr, bs, dp	MRR@20	HR@20	accuracy
------------	--------	-------	----------

lr, bs, dp	MRR@20	HR@20	accuracy
0.05, 128, 0.3	0.012	4.8%	0.00

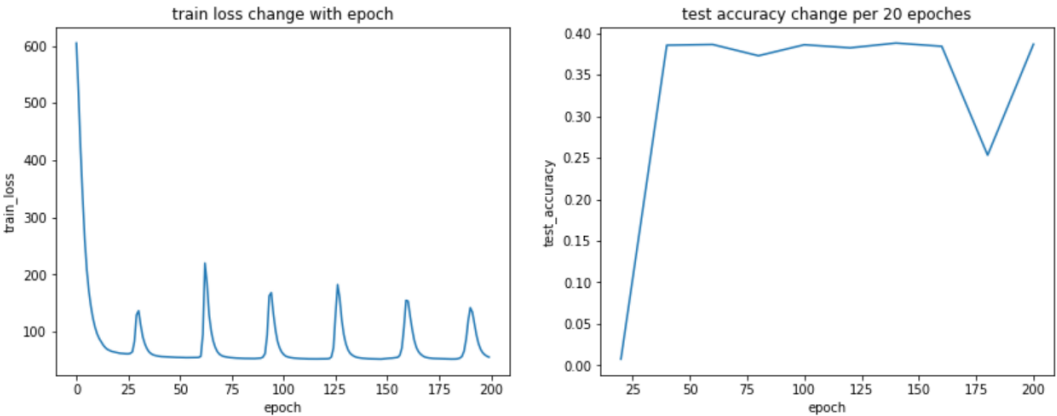
(表现极差)



lr, bs, dp	MRR@20	HR@20	accuracy
0.01, 256, 0.3	0.413	50.3%	0.34



lr, bs, dp	MRR@20	HR@20	accuracy
0.01, 256, 0	0.452	50.2%	0.38



最后取参数: lr=0.01, dropout=0, batch_size=256

2. NARM

在阅读文献过程中，注意到NARM模型即为在GRU4REC模型的基础上增加了注意力机制，于是进行了论文的精读和模型尝试。

NARM模型是一个encoder-decoder架构，且模型的encoder部分包括两个组件，使用的都是GRU结构：

- global encoder：用来捕捉s全局特征，输输出为GRU的最后一个hidden state
- local encoder：用来捕捉当前偏好特征，使用了attention机制，考虑了GRU在每个时间的hidden state

模型将两个encoder结合在一起，得到 $\mathbf{c}_t = [\mathbf{c}_t^g, \mathbf{c}_t^l]$ ，然后使用一个双线性解码函数来计算 \mathbf{c}_t 与每个item embedding的相似度，用来预测对应item的推荐得分：

$$S_i = emb_i^T B \mathbf{c}_t$$

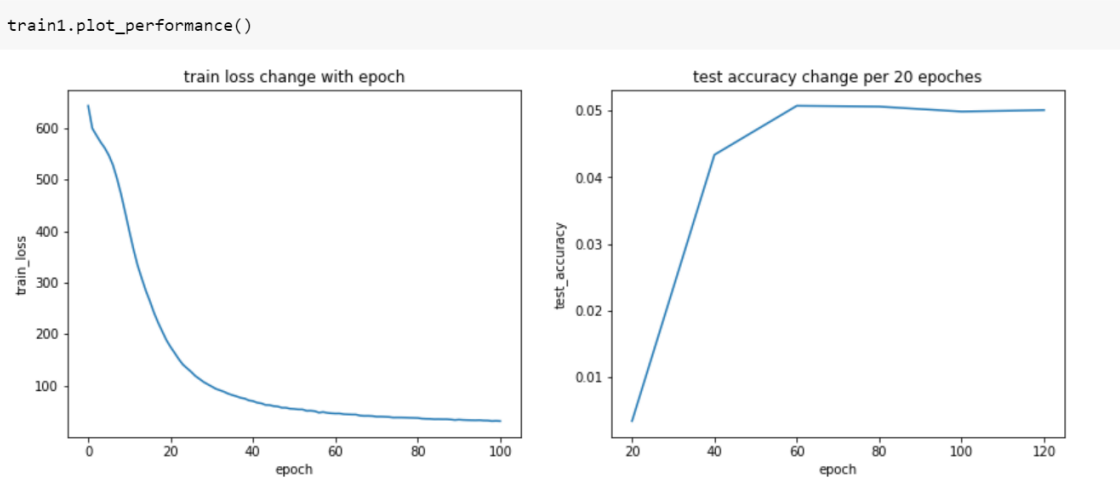
模型同样使用cross-entropy loss进行训练，超参数（dropout、batch_size、hidden_size）由论文的experiment部分给出。

得到了挺差的训练结果：

```
epoch 60 .....
train loss = 45.37
test accuracy = 0.05
MRR = 0.0748 | HR = 15.6845%

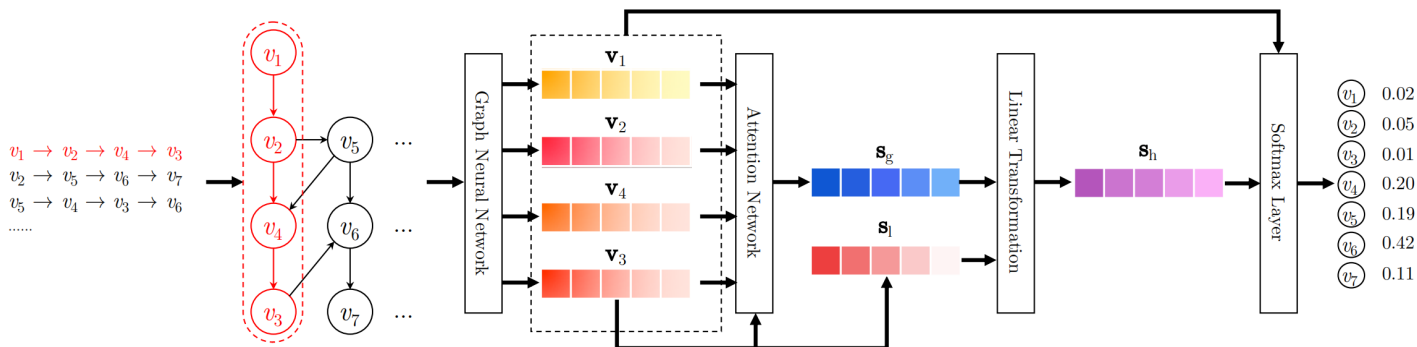
epoch 80 .....
train loss = 36.69
test accuracy = 0.05
MRR = 0.0730 | HR = 15.0864%

epoch 100 .....
train loss = 30.73
test accuracy = 0.05
MRR = 0.0733 | HR = 15.0421%
```



经过思考，觉得可能是NARM没有用 `TrainDataset` 类进行训练的缘故，因此表现不及GRU4TRC；因此，我将二者结合，进行了模型的改进，在报告的第五部分进行详述。

3. SRGNN



(1) 数据处理

与GRU4Rec模型的预处理相同，把session转换为列表类型，去除了仅含一个item的session，并重新进行了编码。此外，还根据文章中实验的预处理部分，除去了总出现次数小于5的item。

除此之外，数据处理部分有一个需要重点理解的函数 `get_slice()`，它接受一个batch数据 `x`（已经完成零填充）的输入，并返回四个值：`alias_inputs`，`A`，`items`，`mask`，其含义分别为：

- `A`：归一化的邻接矩阵 \mathbf{A} ，包括 $\mathbf{A}^{(in)}$ 和 $\mathbf{A}^{(out)}$ 两个矩阵
- `items`：去重排序后的session，其语义是每个session的物品空间，不考虑时效性，和`alias_input`搭配使用查找A中的内容，用它来生成item embedding
- `mask`：记录session的零填充位置，item为True，填充位为False，用于获取 $s_l = \mathbf{v}_n$
- `alias_inputs`：session中的item在去重排序后的索引

假设一个session是`s = [5, 11, 7, 0, 0, 0]`，那么它去重排序的结果是`items = [0, 5, 7, 11]`

`alias_input`的值是`[1, 3, 2, 0, 0, 0]`，item 5在`alias_input`中对应的值是1，则`items[1]`即为item 5

在代码中，我通过以下操作，将建立在`items`上的item embedding转换为建立在输入batch数据`x`上的item embedding：

```
v = torch.gather(hidden, dim=1, index=alias_inputs)
```

这么做是因为构建矩阵A是要先对session进行去重排序的。矩阵A的下标并不是以session中的第一个物品开始，而是从最小的item id开始，所以要用一个额外的序列来帮助标记。

示例如下 (`batch_size = 1`)：

```
1 X: [[464, 1150, 0]] # [batch_size, seq_len], 排序后为[0, 464, 1150]
2
3 A: [[[0, 0, 0, 0, 0, 0], # [batch_size, seq_len, 2*seq_len], 前三列是in, 后三列是out
4      [0, 0, 0, 0, 0, 1],
5      [0, 1, 0, 1, 0, 0]]]
6
7 items: [[0, 464, 1150]] # [batch_size, seq_len]
8
9 mask: [[True, True, False]] # [batch_size, seq_len]
10
11 alias_inputs: [[1, 2, 0]] # [batch_size, seq_len]
```

(2) 模型建立

模型主体的搭建包括两个部分：GGNN模块的搭建，完整SR-GNN的搭建。在代码中都加入了很详细的注释，所以不在报告中解释代码细节。

- **GGNN部分**，学习item embedding，得到 v

GGNN 类实现了论文中的式(1)~(5)，将一个batch的session数据转换成item embedding

输入参数：A 为会话图的邻接矩阵，hidden 为初始embedding

- **SR-GNN部分**，得到session embedding s_h

网络结构为embedding→GGNN，将GGNN得到的结果进行后续一系列加工，得到最终的session embedding

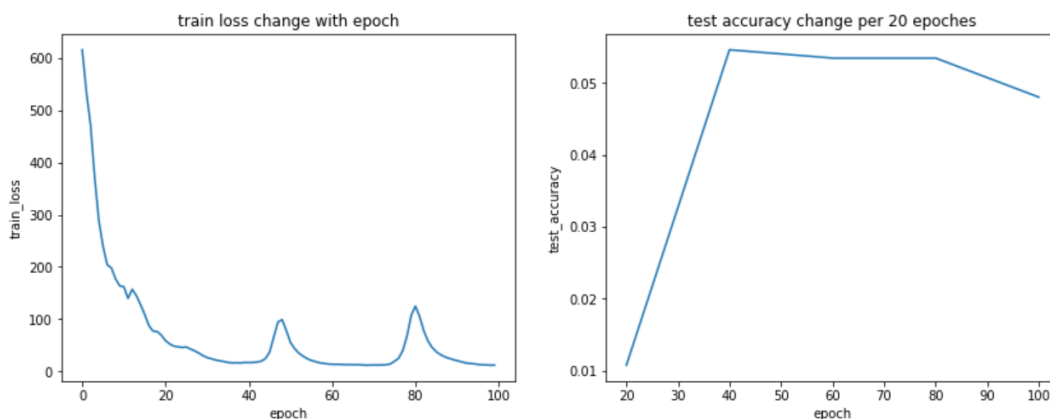
(3) 实验细节和结果

与论文相同，使用cross-entropy loss进行训练。然而和NARM一样，训练结果并不太好：

```
epoch 60 .....
train loss = 13.95
test accuracy = 0.05
MRR = 0.0721 | HR = 12.6495%

epoch 80 .....
train loss = 125.01
test accuracy = 0.05
MRR = 0.0652 | HR = 11.8520%
```

```
[12] train.plot_performance()
```



4. 模型改进：GRUwATT, GGNNwATT

在尝试了NARM和SRGNN后，发现这两个模型的表现并不如GRU4REC。经过思考，考虑到可能是由于GRU4REC使用了

TrainDataset 类进行训练，而这两个模型由于attention的结构问题而不能使用该类进行训练

所以我尝试着对原始的GRU4REC模型进行了改进，参考NARM，我在GRU4REC的基础上增加了attention机制，但没有考虑NARM中的global encoder，修改的思路是：为了配合 TrainDataset 类的数据格式，我希望网络返回的数据形状是

[batch_size, seq_len, n_items]，不希望维度1消失。

记gru的输出hidden state为 $h = [h_1, ..., h_t]$ ，参照NARM的local encoder部分，首先计算：

$$\alpha_{tj} = v^T \cdot \sigma(A_1 h_t + A_2 h_j)$$

然后，区别于NARM中的 $c_t = \sum_{j=1}^t \alpha_{tj} h_j$

，下一步计算为：

$$c_t = \alpha_{tj} h_j \in \mathbb{R}^{bs \times \text{hidden_state}}$$

由此得到矩阵 $c = [c_1, ..., c_t]$ ，维度为 [batch_size, seq_len, hidden_size]，接下来再将 c 与 h 合并到一起后输入feed forward层，就可以得到想要的形状。

经过实验和调参，很开心地得到了比较好的结果！

```
epoch 140 .....
train loss = 27.33
test accuracy = 0.35
MRR = 0.3998 | HR = 47.2087%

epoch 160 .....
train loss = 26.38
test accuracy = 0.36
MRR = 0.4015 | HR = 46.8764%

epoch 180 .....
train loss = 25.50
test accuracy = 0.36
MRR = 0.4007 | HR = 46.5884%

epoch 200 .....
train loss = 25.53
test accuracy = 0.35
MRR = 0.3986 | HR = 46.4112%
```

于是很兴奋的把这套逻辑也搬到了GNN上，搭建了 `GGNNwATT` 模型，但结果还是不好：

```
epoch 60 .....
train loss = 8.41
test accuracy = 0.05
MRR = 0.0759 | HR = 14.9535%

epoch 80 .....
train loss = 19.61
test accuracy = 0.05
MRR = 0.0764 | HR = 14.4883%

epoch 100 .....
train loss = 9.19
test accuracy = 0.05
MRR = 0.0775 | HR = 14.9313%
```







最终保存了GRU的改进模型GRUwATT。

五、总结和展望

- 在完成过程中最困难的部分并不是网络结构的搭建，而是数据的处理。比如为了更好地利用已有数据和获取batch数据，定义了TrainDataset和MyDataset两个类，以及花费不少精力写的 `get_dataloader()` 函数；此外，在SRGNN的搭建过程中，我花费了大量的时间来实现 `get_slice()` 函数的功能，以及在 `SRGNN` 类的forward函数中对得到的item embedding进行加工。
- 搭建中期模型时，没有考虑到一些功能的通用性，如数据处理、batch数据加载等，导致每开一个新的文件就得把以上函数搬运一遍。最后还是为了方便和更好的可读性，花了不少时间把这些通用功能单独开了一个代码文件，然后再修改已经写好的代码。以后在代码结构的设置上要注意这类问题。
- 在数据预处理中，进行item2index编码时，没有考虑到test数据中可能有train数据中并不包含的item，只对train中的item进行了编码。这个错误直到在训练好并选择了最终模型后才发现，并因此需要在数据预处理步骤进行对应修改后重新训练模型。这是一个值得记住的错误。
- GRUwATT模型的搭建是出于自己的思考与尝试，第一次跑通并看到较好的结果时是非常惊喜的（虽然并不是什么厉害的改进），在整个大作业的完成过程中体会到了自行尝试和动手搭建模型解决问题的快乐和好处。
- 一个有些出乎意料的结果是，最终表现最好的模型是第一个尝试的也是最简单的GRU4REC，这让我进一步感受到了奥卡姆的剃刀原则。可能是因为我们用的是比较轻量 and 简单的小数据集，因此更复杂的模型反而并不会得到更好的结果，适用的才是最好的。
- 在完成任务的过程中，通过自行搭建Dataset和Dataloader，感受到了数据增强起到的重要作用。更加复杂的模型的效果并不及简单的GRU4REC，甚至相差较大，可能就是因为复杂模型中由于网络结构的限制，没有使用自行创建的TrainDataset类。GRUwATT模型的结果明显好于NARM，也验证了这一想法。

再进一步的探索和展望中，除了尝试其他不同结构的模型之外，还可以尝试探索更好的数据增强方法；此外，可以尝试一些有更多contextual信息的推荐系统任务（如非匿名的序列推荐，可以创建用户画像/商品画像等），以期在深度学习以及推荐系统的学习上获得更深的理解和更有趣的体验。

参考资料

- SBR综述 (survey01) : [A Survey on Session-based Recommender Systems \(arxiv.org\)](#) 
- GRU4Rec论文: [1511.06939.pdf \(arxiv.org\)](#) 
- NARM论文: [Neural Attentive Session-based Recommendation](#) 
- GGNN论文: [1511.05493.pdf \(arxiv.org\)](#) 
- SR-GNN论文: [1811.00855.pdf \(arxiv.org\)](#) 
- 代码:
 - <https://github.com/hungthanhpham94/GRU4REC> 
 - <https://github.com/CRIPAC-DIG/SR-GNN> 