

[论文解读] 阿里DIEN整体代码结构

原创 罗西的思考 罗西的思考 11月3日

[论文解读] 阿里DIEN整体代码结构

- 0x00 摘要
- 0x01 文件简介
- 0x02 总体架构
- 0x03 总体代码
- 0x04 模型基类
 - 4.1 基本逻辑
 - 4.2 模块分析
- 0x05 Model_DIN_V2_Gru_Vec_attGru_Neg
 - 5.1 第一层 'rnn_1'
 - 5.2 第二层 'Attention_layer_1'
- 0x06 全连接层
- 0x07 训练模型
- 0xFF 参考

0x00 摘要

DIEN是阿里深度兴趣进化网络(Deep Interest Evolution Network)的缩写。

本文将分析DIEN源码整体思路。因为DIEN是在DIN基础上演化，所以代码有大部分重复。

本文采用的是 <https://github.com/mouna99/dien> 中的实现。

0x01 文件简介

数据文件主要包括：

- **uid_voc.pkl**: 用户字典，用户名对应的id；

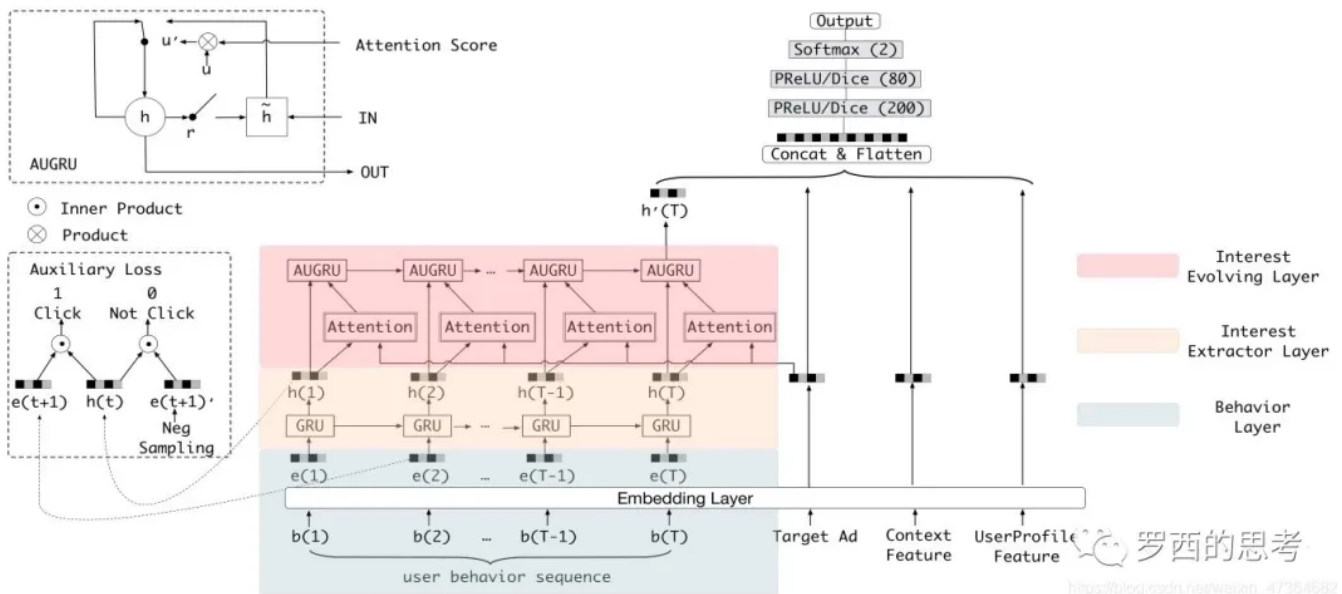
- **mid_voc.pkl**: movie字典,item对应的id;
- **cat_voc.pkl**: 种类字典, category对应的id;
- **item-info**: item对应的category信息;
- **reviews-info**: review 元数据, 格式为: userID, itemID, 评分, 时间戳, 用于进行负采样的数据;
- **local_train_splitByUser**: 训练数据, 一行格式为: label、用户名、目标item、 目标item类别、历史item、历史item对应类别;
- **local_test_splitByUser**: 测试数据, 格式同训练数据;

代码主要包含:

- **rnn.py**: 对tensorflow中原始的rnn进行修改, 目的是将attention同rnn进行结合
- **vecAttGruCell.py**: 对GRU源码进行修改, 将attention加入其中, 设计AUGRU结构
- **data_iterator.py**: 数据迭代器, 用于数据的不断输入
- **utils.py**: 一些辅助函数, 如dice激活函数、attention score计算等
- **model.py**: DIEN模型文件
- **train.py**: 模型的入口, 用于训练数据、保存模型和测试数据

0x02 总体架构

首先还是要从论文中摘取架构图进行说明。



在这里插入图片描述

深度兴趣进化网络分为几层, 从下到上依次是:

- 行为序列层（Behavior Layer）：主要作用是将用户浏览过的商品转换成对应的embedding，并且按照浏览时间做排序，即把原始的id类行为序列特征转换成Embedding行为序列；
- 兴趣抽取层（Interest Extractor Layer）：主要作用是通过模拟用户的兴趣迁移过程，基于行为序列提取用户兴趣序列；
- 兴趣进化层（Interest Evolving Layer）：主要作用是通过在兴趣抽取层基础上加入Attention机制，模拟与当前目标广告相关的兴趣进化过程，对与目标物品相关的兴趣演化过程进行建模；
- 将兴趣表示和ad、user profile、context的embedding向量进行拼接。最后使用MLP完成最后的预测；

0x03 总体代码

DIEN代码是从train.py开始。train.py 先用初始模型评估一遍测试集，然后调用 train：

- 获取 训练数据 和 测试数据，这两个都是数据迭代器，用于数据的不断输入
- 根据 model_type 生成相应的model
- 按照batch训练，每1000次评估测试集。

代码如下：

```
def train(  
    train_file = "local_train_splitByUser",  
    test_file = "local_test_splitByUser",  
    uid_voc = "uid_voc.pkl",  
    mid_voc = "mid_voc.pkl",  
    cat_voc = "cat_voc.pkl",  
    batch_size = 128,  
    maxlen = 100,  
    test_iter = 100,  
    save_iter = 100,  
    model_type = 'DNN',  
    seed = 2,  
):  
  
    with tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)) as sess:  
        ## 训练数据  
        train_data = DataIterator(train_file, uid_voc, mid_voc, cat_voc, batch_size, maxlen,  
        ## 测试数据
```


基本逻辑如下：

- 在 'Inputs' scope 下，构建各种 placeholder 变量；
- 在 'Embedding_layer' scope 下，构建user, item的embedding lookup table，将输入数据转换为对应的embedding；
- 把各种 embedding vector 结合起来，比如将item的id对应的embedding 以及 item对应的cateid的embedding进行拼接，共同作为item的embedding；

4.2 模块分析

下面的 B 是 batch size，T 是序列长度，H 是hidden size，程序中初始化变量如下：

```
EMBEDDING_DIM = 18
HIDDEN_SIZE = 18 * 2
ATTENTION_SIZE = 18 * 2
best_auc = 0.0
```

4.2.1 构建变量

首先是构建placeholder变量。

```
with tf.name_scope('Inputs'):
    # shape: [B, T] #用户行为特征(User Behavior)中的 movie id 历史行为序列。T为序列长度
    self.mid_his_batch_ph = tf.placeholder(tf.int32, [None, None], name='mid_his_batch_ph')
    # shape: [B, T] #用户行为特征(User Behavior)中的 category id 历史行为序列。T为序列长度
    self.cat_his_batch_ph = tf.placeholder(tf.int32, [None, None], name='cat_his_batch_ph')
    # shape: [B], user id 序列。(B: batch size)
    self.uid_batch_ph = tf.placeholder(tf.int32, [None, ], name='uid_batch_ph')
    # shape: [B], movie id 序列。(B: batch size)
    self.mid_batch_ph = tf.placeholder(tf.int32, [None, ], name='mid_batch_ph')
    # shape: [B], category id 序列。(B: batch size)
    self.cat_batch_ph = tf.placeholder(tf.int32, [None, ], name='cat_batch_ph')
    self.mask = tf.placeholder(tf.float32, [None, None], name='mask')
    # shape: [B]; sl: sequence Length, User Behavior中序列的真实序列长度(?)
    self.seq_len_ph = tf.placeholder(tf.int32, [None], name='seq_len_ph')
    # shape: [B, T], y: 目标节点对应的 Label 序列, 正样本对应 1, 负样本对应 0
    self.target_ph = tf.placeholder(tf.float32, [None, None], name='target_ph')
    # 学习速率
```

```

self.lr = tf.placeholder(tf.float64, [])
self.use_negsampling = use_negsampling
if use_negsampling:
    self.noclk_mid_batch_ph = tf.placeholder(tf.int32, [None, None, None], name='noclk_mid_batch_ph')
    self.noclk_cat_batch_ph = tf.placeholder(tf.int32, [None, None, None], name='noclk_cat_batch_ph')

```

具体各种shape可以参见下面运行时变量

```

self = {Model_DIN_V2_Gru_Vec_attGru_Neg}
cat_batch_ph = {Tensor} Tensor("Inputs/cat_batch_ph:0", shape=(?,), dtype=int32)
uid_batch_ph = {Tensor} Tensor("Inputs/uid_batch_ph:0", shape=(?,), dtype=int32)
mid_batch_ph = {Tensor} Tensor("Inputs/mid_batch_ph:0", shape=(?,), dtype=int32)

cat_his_batch_ph = {Tensor} Tensor("Inputs/cat_his_batch_ph:0", shape=(?, ?), dtype=int32)
mid_his_batch_ph = {Tensor} Tensor("Inputs/mid_his_batch_ph:0", shape=(?, ?), dtype=int32)

lr = {Tensor} Tensor("Inputs/Placeholder:0", shape=(), dtype=float64)
mask = {Tensor} Tensor("Inputs/mask:0", shape=(?, ?), dtype=float32)
seq_len_ph = {Tensor} Tensor("Inputs/seq_len_ph:0", shape=(?,), dtype=int32)
target_ph = {Tensor} Tensor("Inputs/target_ph:0", shape=(?, ?), dtype=float32)

noclk_cat_batch_ph = {Tensor} Tensor("Inputs/noclk_cat_batch_ph:0", shape=(?, ?, ?), dtype=int32)
noclk_mid_batch_ph = {Tensor} Tensor("Inputs/noclk_mid_batch_ph:0", shape=(?, ?, ?), dtype=int32)

use_negsampling = {bool} True

```

4.2.2 构建embedding

然后是构建user, item的embedding lookup table, 将输入数据转换为对应的embedding, 就是把稀疏特征转换为稠密特征。关于 embedding 层的原理和代码分析, 本系列会有专文讲解。

后续的 U 是user_id的hash bucket size, I 是item_id的hash bucket size, C 是cat_id的hash bucket size。

注意 self.mid_his_batch_ph这样的变量 保存用户的历史行为序列, 大小为 [B, T], 所以在进行embedding_lookup 时, 输出大小为 [B, T, H/2];

```

# Embedding Layer
with tf.name_scope('Embedding_layer'):

```

```

# shape: [U, H/2], user_id的embedding weight. U是user_id的hash bucket size, 即user count
self.uid_embeddings_var = tf.get_variable("uid_embedding_var", [n_uid, EMBEDDING_DIM])
# 从uid embedding weight 中取出 uid embedding vector
self.uid_batch_embedded = tf.nn.embedding_lookup(self.uid_embeddings_var, self.uid_batch_

# shape: [I, H/2], item_id的embedding weight. I是item_id的hash bucket size, 即movie count
self.mid_embeddings_var = tf.get_variable("mid_embedding_var", [n_mid, EMBEDDING_DIM])
# 从mid embedding weight 中取出 uid embedding vector
self.mid_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, self.mid_batch_
# 从mid embedding weight 中取出 mid history embedding vector, 是正样本
# 注意 self.mid_his_batch_ph这样的变量 保存用户的历史行为序列, 大小为 [B, T], 所以在进行 embedd
self.mid_his_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, self.mid_hi
# 从mid embedding weight 中取出 mid history embedding vector, 是负样本
if self.use_negsampling:
    self.noclk_mid_his_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, s

# shape: [C, H/2], cate_id的embedding weight. C是cat_id的hash bucket size
self.cat_embeddings_var = tf.get_variable("cat_embedding_var", [n_cat, EMBEDDING_DIM])
# 从 cid embedding weight 中取出 cid history embedding vector, 是正样本
self.cat_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat_batch_
# 从 cid embedding weight 中取出 cid embedding vector, 是正样本
self.cat_his_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat_hi
# 从 cid embedding weight 中取出 cid history embedding vector, 是负样本
if self.use_negsampling:
    self.noclk_cat_his_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, s

```

具体各种shape可以参见下面运行时变量

```

self = {Model_DIN_V2_Gru_Vec_attGru_Neg}
cat_embeddings_var = {Variable} <tf.Variable 'cat_embedding_var:0' shape=(1601, 18) dtype=fl
uid_embeddings_var = {Variable} <tf.Variable 'uid_embedding_var:0' shape=(543060, 18) dtype=
mid_embeddings_var = {Variable} <tf.Variable 'mid_embedding_var:0' shape=(367983, 18) dtype=

cat_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_4:0", shape=(?, 18),
mid_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_1:0", shape=(?, 18),
uid_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup:0", shape=(?, 18), dt

cat_his_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_5:0", shape=(?, ?
mid_his_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_2:0", shape=(?, ?

noclk_cat_his_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_6:0", shape=
noclk_mid_his_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_3:0", shape=

```

4.2.3 拼接embedding

这部分是把各种 embedding vector 结合起来, 比如将 item的id对应的embedding 以及 item对应的cateid的embedding 进行拼接, 共同作为item的embedding;

关于shape的说明:

- 注意上一步中, self.mid_his_batch_ph这样的变量 保存用户的历史行为序列, 大小为 [B, T], 所以在进行 embedding_lookup 时, 输出大小为 [B, T, H/2]。
- 这里将 Goods 和 Cate 的 embedding 进行 concat, 得到 [B, T, H] 大小. 注意到 tf.concat 中的 axis 参数值为 2。

关于逻辑的说明:

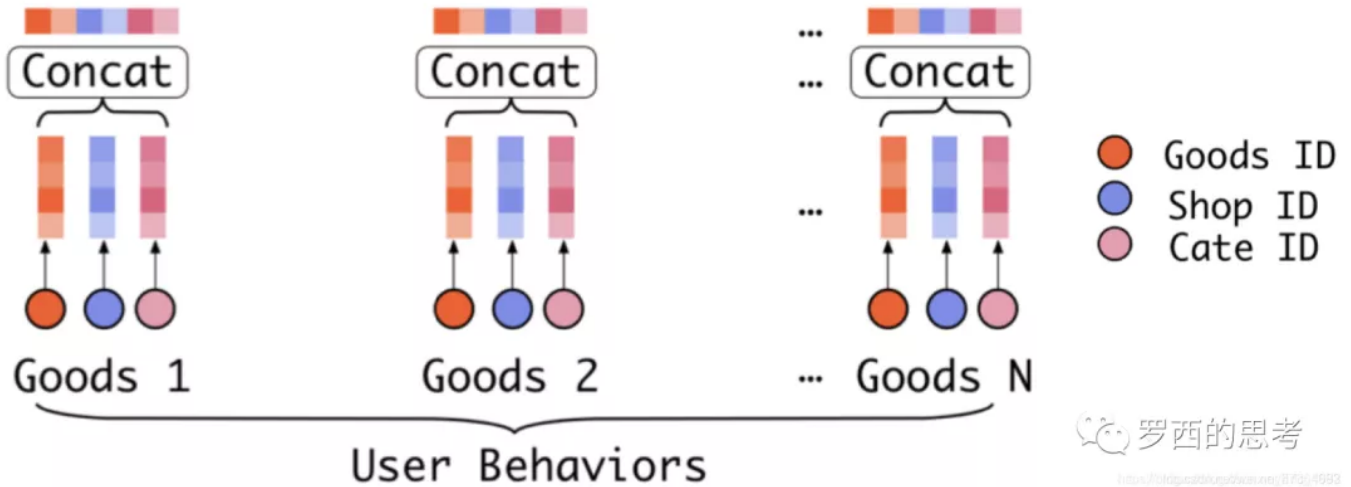
第一步是 `self.item_emb = tf.concat([self.mid_batch_embedded, self.cat_batch_embedded], 1)` 即获取一个 Batch 中目标节点对应的 embedding, 保存在 `i_emb` 中, 它由商品 (Goods) 和类目 (Cate) embedding 进行 concatenation。对应了架构图的:



在这里插入图片描述

第二步是 `self.item_his_emb = tf.concat([self.mid_his_batch_embedded, self.cat_his_batch_embedded], 2)` 逻辑上是对 两个历史矩阵 进行处理, 这两个历史矩阵保存了用户的历

史行为序列, 大小为 $[B, T]$, 所以在进行 `embedding_lookup` 时, 输出大小为 $[B, T, H/2]$ 。之后将 Goods 和 Cate 的 `embedding` 进行 `concat`, 得到 $[B, T, H]$ 大小. 注意到 `tf.concat` 中的 `axis` 参数值为 2。对应了架构图的:



在这里插入图片描述

具体代码如下:

```
# 正样本的embedding拼接, 正样本包括item和cate。即将目标节点对应的商品 embedding 和类目 embedding 进行
self.itemEb = tf.concat([self.mid_batch_embedded, self.cat_batch_embedded], 1)
# 将 Goods 和 Cate 的 embedding 进行 concat, 得到 [B, T, H] 大小. 注意到 tf.concat 中的 axis 参数
self.itemHisEb = tf.concat([self.mid_his_batch_embedded, self.cat_his_batch_embedded], 2)
self.itemHisEbSum = tf.reduce_sum(self.itemHisEb, 1)

# 负样本的embedding拼接, 负样本包括item和cate。即将目标节点对应的商品 embedding 和类目 embedding 进行
if self.use_negsampling:
    # 0 means only using the first negative item ID. 3 item IDs are inputted in the line 24.
    self.noclk_itemHisEb = tf.concat(
        [self.noclk_mid_his_batch_embedded[:, :, 0, :], self.noclk_cat_his_batch_embedded[:, :, 0, :]], 1
    )
    # cat embedding 18 concat item embedding 18.
    self.noclk_itemHisEb = tf.reshape(self.noclk_itemHisEb, [-1, tf.shape(self.noclk_mid_his_batch_embedded)[1],
        self.noclk_hisEb = tf.concat([self.noclk_mid_his_batch_embedded, self.noclk_cat_his_batch_embedded], 1)
    self.noclk_hisEbSum1 = tf.reduce_sum(self.noclk_hisEb, 2)
    self.noclk_hisEbSum = tf.reduce_sum(self.noclk_hisEbSum1, 1)
```

具体各种shape可以参见下面运行时变量

```
self = {Model_DIN_V2_Gru_Vec_attGru_Neg}
```

```

item_eb = {Tensor} Tensor("concat:0", shape=(?, 36), dtype=float32)
item_his_eb = {Tensor} Tensor("concat_1:0", shape=(?, ?, 36), dtype=float32)
item_his_eb_sum = {Tensor} Tensor("Sum:0", shape=(?, 36), dtype=float32)

noclk_item_his_eb = {Tensor} Tensor("Reshape:0", shape=(?, ?, 36), dtype=float32)
noclk_his_eb = {Tensor} Tensor("concat_3:0", shape=(?, ?, ?, 36), dtype=float32)
noclk_his_eb_sum = {Tensor} Tensor("Sum_2:0", shape=(?, 36), dtype=float32)
noclk_his_eb_sum_1 = {Tensor} Tensor("Sum_1:0", shape=(?, ?, 36), dtype=float32)

```

0x05 Model_DIN_V2_Gru_Vec_attGru_Neg

Model_DIN_V2_Gru_Vec_attGru_Neg 是 DIEN 对应的模型，用户历史肯定是一个时间序列，将其喂入RNN，则最后一个状态可以认为包含了所有历史信息。因此，作者用一个双层的GRU来建模用户兴趣。

Model_DIN_V2_Gru_Vec_attGru_Neg 的 `__init__` 函数就构建了这个双层GRU，其代码逻辑是：

- 第一层 'rnn_1' 对应架构图中黄色部分，即兴趣抽取层（Interest Extractor Layer）。
- 第二层 'Attention_layer_1' 对应架构图中的红色部分的兴趣进化层（Interest Evolving Layer），主要组件是 AUGRU。

5.1 第一层 'rnn_1'

这一层对应架构图中黄色部分，即兴趣抽取层（Interest Extractor Layer），主要组件是GRU。

主要作用是通过模拟用户的兴趣迁移过程，基于行为序列提取用户兴趣序列。即将用户行为历史的item embedding输入到dynamic rnn（第一层GRU）中，同时计算辅助loss，输出的就是用户各时刻的兴趣。

```

# RNN Layer(-s)
with tf.name_scope('rnn_1'):
    rnn_outputs, _ = dynamic_rnn(GRUCell(HIDDEN_SIZE), inputs=self.item_his_eb,
                                sequence_length=self.seq_len_ph, dtype=tf.float32,
                                scope="gru1")

    aux_loss_1 = self.auxiliary_loss(rnn_outputs[:, :-1, :], self.item_his_eb[:, 1:, :],
                                    self.noclk_item_his_eb[:, 1:, :],

```

```
self.mask[:, 1:], stag="gru")

self.aux_loss = aux_loss_1
```

5.1.1 GRU

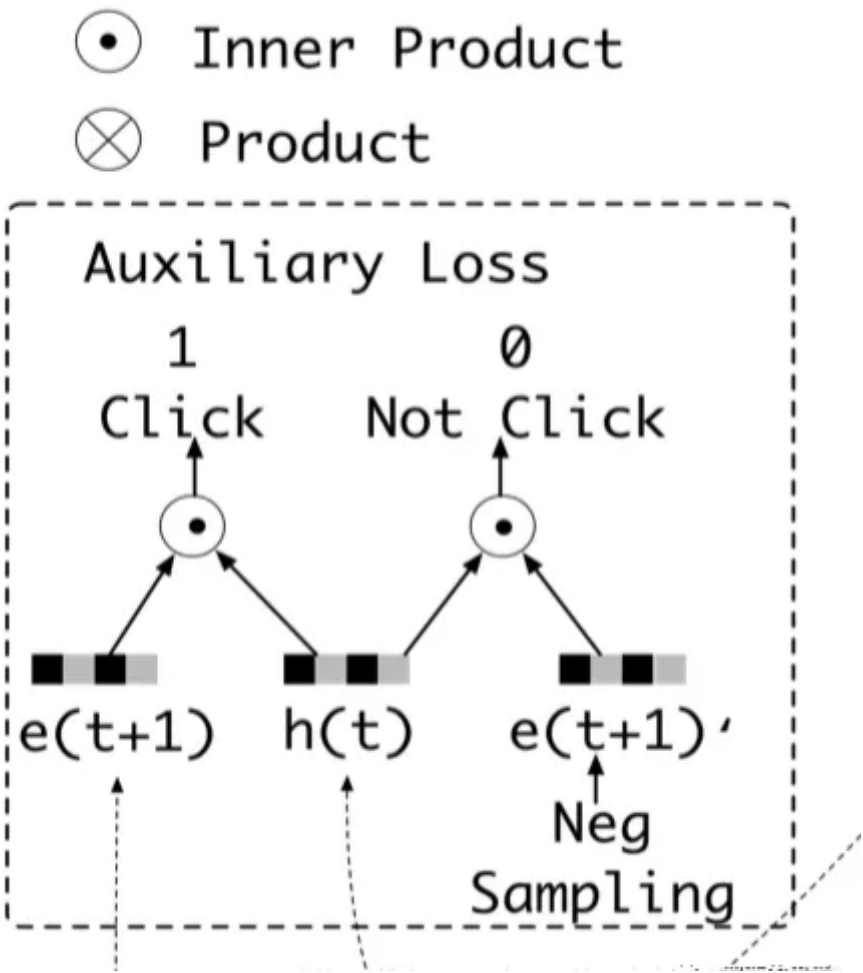
GRU如下，具体我们会另文介绍RNN。

$$\begin{aligned} \mathbf{u}_t &= \sigma(W^u \mathbf{i}_t + U^u \mathbf{h}_{t-1} + \mathbf{b}^u), \\ \mathbf{r}_t &= \sigma(W^r \mathbf{i}_t + U^r \mathbf{h}_{t-1} + \mathbf{b}^r), \\ \tilde{\mathbf{h}}_t &= \tanh(W^h \mathbf{i}_t + \mathbf{r}_t \circ U^h \mathbf{h}_{t-1} + \mathbf{b}^h), \\ \mathbf{h}_t &= (\mathbf{1} - \mathbf{u}_t) \circ \mathbf{h}_{t-1} + \mathbf{u}_t \circ \tilde{\mathbf{h}}_t, \end{aligned}$$

在这里插入图片描述

5.1.2 辅助损失

辅助loss的计算其实是一个二分类模型，对应论文中：



在这里插入图片描述

具体来讲，就是利用 t 时刻的行为 $b(t+1)$ 作为监督去学习隐含层向量 h_t 。正负样本分别代表了用户 点击/未点击 的第 t 个物品embedding向量。

- 利用真实的下一个行为作为正样本；
- 负例的选择或是从用户未交互过的商品中随机抽取，或是从已展示给用户但用户没有点击的商品中随机抽取；

$$L_{aux} = -\frac{1}{N} \left(\sum_{i=1}^N \sum_t \log \sigma(h_t^i, e_b^i[t+1]) + \log(1 - \sigma(h_t^i, \hat{e}_b^i[t+1])) \right) \quad (6)$$

罗西的思考

在这里插入图片描述

代码中用到了 `tf.concat([h_states, click_seq], -1)`，这里-1的意思是倒数第一维度增加，其余不变。

比如 $(3,2,4) + (3,2,4)$ 如果按照 -1 操作，则shape最后一维增加，变成 $(3,2,8)$ ，这样就把两个tensor合并起来了。

具体代码如下：

```
def auxiliary_loss(self, h_states, click_seq, noclick_seq, mask, stag = None):
    mask = tf.cast(mask, tf.float32)
    # 倒数第一维度concat，其余不变
    click_input_ = tf.concat([h_states, click_seq], -1)
    # 倒数第一维度concat，其余不变
    noclick_input_ = tf.concat([h_states, noclick_seq], -1)
    # 获取正样本最后一个y_hat
    click_prop_ = self.auxiliary_net(click_input_, stag = stag)[: , :, 0]
    # 获取正样本最后一个y_hat
    noclick_prop_ = self.auxiliary_net(noclick_input_, stag = stag)[: , :, 0]
    # 对数损失，并且mask出真实历史行为
    click_loss_ = - tf.reshape(tf.log(click_prop_), [-1, tf.shape(click_seq)[1]]) * mask
    noclick_loss_ = - tf.reshape(tf.log(1.0 - noclick_prop_), [-1, tf.shape(noclick_seq)[1]])
    loss_ = tf.reduce_mean(click_loss_ + noclick_loss_)
    return loss_
```

```
def auxiliary_net(self, in_, stag='auxiliary_net'):
    bn1 = tf.layers.batch_normalization(inputs=in_, name='bn1' + stag, reuse=tf.AUTO_REUSE)
    dnn1 = tf.layers.dense(bn1, 100, activation=None, name='f1' + stag, reuse=tf.AUTO_REUSE)
    dnn1 = tf.nn.sigmoid(dnn1)
    dnn2 = tf.layers.dense(dnn1, 50, activation=None, name='f2' + stag, reuse=tf.AUTO_REUSE)
    dnn2 = tf.nn.sigmoid(dnn2)
    dnn3 = tf.layers.dense(dnn2, 2, activation=None, name='f3' + stag, reuse=tf.AUTO_REUSE)
    y_hat = tf.nn.softmax(dnn3) + 0.00000001
    return y_hat
```

5.1.3 mask的作用

关于mask的作用，这里结合 Transformer 再说一下：

mask 表示掩码，它对某些值进行掩盖，使其在参数更新时不产生效果。Transformer 模型里面涉及两种 mask，分别是 padding mask 和 sequence mask。其中，padding mask 在所有的 scaled dot-product attention 里面都需要用到，而 sequence mask 只有在 decoder 的 self-attention 里面用到。

Padding Mask

什么是 padding mask 呢？因为每个批次输入序列长度是不一样的也就是说，我们要对输入序列进行对齐。具体来说，就是给在较短的序列后面填充 0。但是如果输入的序列太长，则是截取左边的内容，把多余的直接舍弃。因为这些填充的位置，其实是没什么意义的，所以 attention 机制不应该把注意力放在这些位置上，需要进行一些处理。

具体的做法是，把这些位置的值加上一个非常大的负数(负无穷)，这样的话，经过 softmax，这些位置的概率就会接近 0！而我们的 padding mask 实际上是一个张量，每个值都是一个 Boolean，值为 false 的地方就是我们要进行处理的地方。

Sequence mask

sequence mask 是为了使得 decoder 不能看见未来的信息。也就是对于一个序列，在 time_step 为 t 的时刻，我们的解码输出应该只能依赖于 t 时刻之前的输出，而不能依赖 t 之后的输出。因此我们需要想一个办法，把 t 之后的信息给隐藏起来。

那么具体怎么做呢？也很简单：产生一个上三角矩阵，上三角的值全为 0。把这个矩阵作用在每一个序列上，就可以达到我们的目的。

对于 decoder 的 self-attention，里面使用到的 scaled dot-product attention，同时需要 padding mask 和 sequence mask 作为 attn_mask，具体实现就是两个mask相加作为 attn_mask。

其他情况，attn_mask 一律等于 padding mask。

DIN这里使用的是padding mask。

5.2 第二层 'Attention_layer_1'

第二层 'Attention layer 1' 对应架构图中的红色部分的兴趣进化层（Interest Evolving Layer），主要组件是 AUGRU。

5.2.1 Attention机制

Attention机制是：将Source中的构成元素想象成是由一系列的< Key, Value >数据对构成，此时给定Target中的某个元素Query，通过计算Query和各个Key的相似性或者相关性，得到每个Key对应Value的权重系数，然后对Value进行加权求和，即得到了最终的Attention数值。所以本质上Attention机制是对Source中元素的Value值进行加权求和，而Query和Key用来计算对应Value的权重系数。即可以将其本质思想改写为如下公式：

$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_x} \text{Similarity}(\text{Query}, \text{Key}_i) * \text{Value}_i$$

这里写图片描述

当然，从概念上理解，把Attention仍然理解为从大量信息中有选择地筛选出少量重要信息并聚焦到这些重要信息上，忽略大多不重要的信息，这种思路仍然成立。聚焦的过程体现在权重系数的计算上，权重越大越聚焦于其对应的Value值上，即权重代表了信息的重要性，而Value是其对应的信息。

另外一种理解是：也可以将Attention机制看作一种软寻址（Soft Addressing）：Source可以看作存储器内存储的内容，元素由地址Key和值Value组成，当前有个Key=Query的查询，目的是取出存储器中对应的Value值，即Attention数值。通过Query和存储器内元素Key的地址进行相似性比较来寻址，之所以说是软寻址，指的不像一般寻址只从存储内容里面找出一条内容，而是可能从每个Key地址都会取出内容，取出内容的重要性根据Query和Key的相似性来决定，之后对Value进行加权求和，这样就可以取出最终的Value值，也即Attention值。所以不少研究人员将Attention机制看作软寻址的一种特例，这也是非常有道理的。

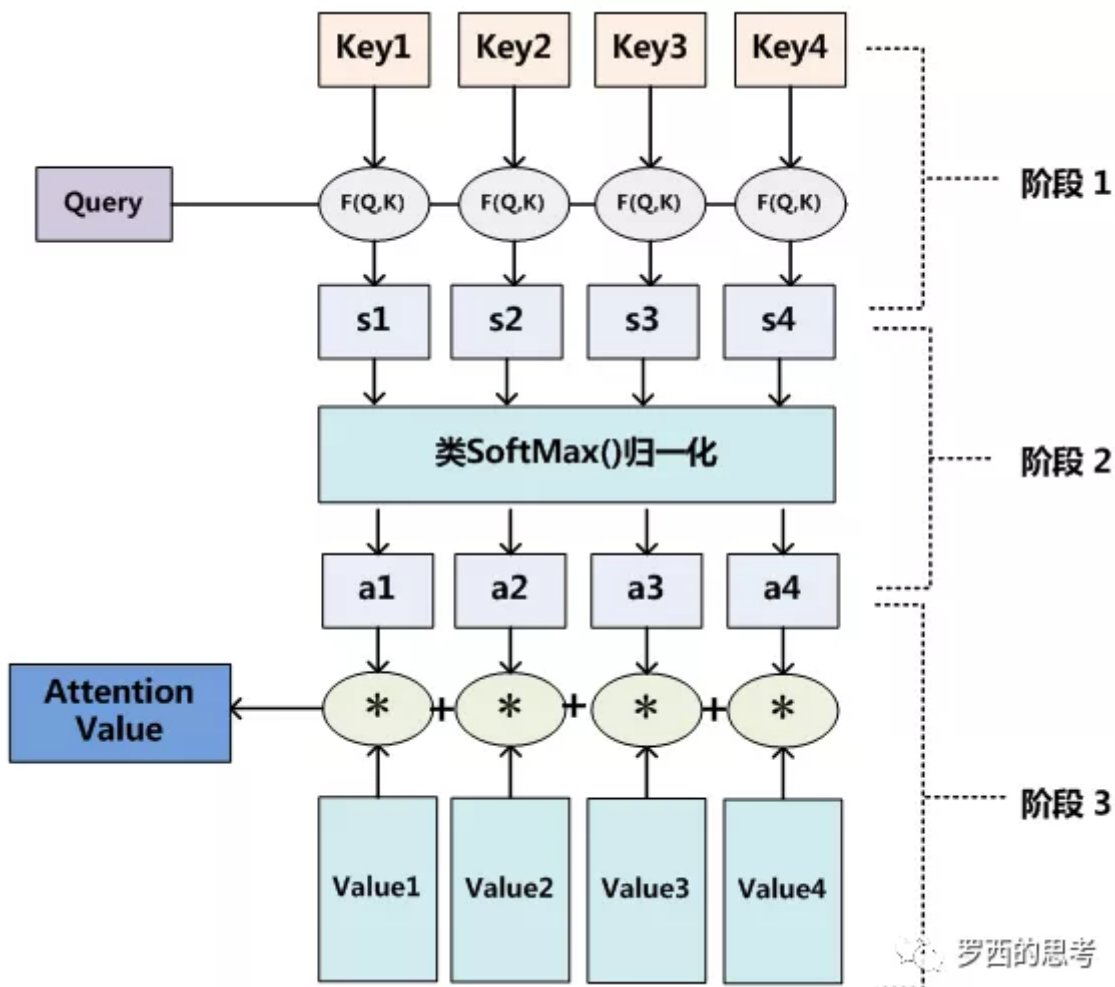
至于Attention机制的具体计算过程，如果对目前大多数方法进行抽象的话，可以将其归纳为两个过程：

- 第一个过程是根据Query和Key计算权重系数；
- 第二个过程根据权重系数对Value进行加权求和。

而第一个过程又可以细分为两个阶段：

- 第一个小阶段根据Query和Key计算两者的相似性或者相关性；
- 第二个小阶段对第一阶段的原始分值进行归一化处理；

这样，可以将Attention的计算过程抽象为如图展示的三个阶段。



在第一个阶段，可以引入不同的函数和计算机制，根据Query和某个Key_i，计算两者的相似性或者相关性。最常见的方法包括：求两者的向量点积、求两者的向量Cosine相似性或者通过再引入额外的神经网络来求值，即如下方式：

点积： $\text{Similarity}(\text{Query}, \text{Key}_i) = \text{Query} \cdot \text{Key}_i$

Cosine 相似性： $\text{Similarity}(\text{Query}, \text{Key}_i) = \frac{\text{Query} \cdot \text{Key}_i}{\|\text{Query}\| \cdot \|\text{Key}_i\|}$

MLP 网络： $\text{Similarity}(\text{Query}, \text{Key}_i) = \text{MLP}(\text{Query}, \text{Key}_i)$

这里写图片描述

第一阶段产生的分值根据具体产生的方法不同其数值取值范围也不一样。

第二阶段引入类似SoftMax的计算方式对第一阶段的得分进行数值转换，一方面可以进行归一化，将原始计算分值整理成所有元素权重之和为1的概率分布；另一方面也可以通过SoftMax的内在机制更加突出重要元素的权重。即一般采用如下公式计算：

$$a_i = \text{Softmax}(Sim_i) = \frac{e^{Sim_i}}{\sum_{j=1}^L Sim_j}$$

罗西的思考

这里写图片描述

第二阶段的计算结果 a_i 即为 $Value_i$ 对应的权重系数，然后进行加权求和即可得到Attention数值：

$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_x} a_i \cdot Value_i$$

罗西的思考

这里写图片描述

通过如上三个阶段的计算，即可求出针对Query的Attention数值，目前绝大多数具体的注意力机制计算方法都符合上述的三阶段抽象计算过程。

5.2.2 Attention layer

DIEN 中，'Attention_layer_1' 层的作用是：通过在兴趣抽取层基础上加入Attention机制，模拟与当前目标广告相关的兴趣进化过程，对与目标物品相关的兴趣演化过程进行建模。即将第一层的输出，喂进第二层GRU，并用attention score（基于第一层的输出向量与候选物料料计算得出）来控制第二层的GRU的update gate。

我们首先需要计算attention的score，后续会将其作为GRU的一部分输入。

此处对应论文中

$$a_t = \frac{\exp(\mathbf{h}_t W \mathbf{e}_a)}{\sum_{j=1}^T \exp(\mathbf{h}_j W \mathbf{e}_a)},$$

罗西的思考

在这里插入图片描述

代码如下：

```
# Attention layer
```



```
with tf.name_scope('Attention_layer_1'):
    att_outputs, alphas = din_fcn_attention(self.item_emb, rnn_outputs, ATTENTION_SIZE, self.n
                                         softmax_stag=1, stag='1_1', mode='LIST', return_e
```

代码经过以下几个步骤得到用户的兴趣分布，可以理解为，一个query过来了，先根据此query和一系列候选物的key(fact) 计算相似度，然后根据相似度计算候选物的具体value:

- 如果time_major，则会进行转换：(T,B,D) => (B,T,D);
- 转换mask。
 - 使用 `tf.ones_like(mask)` 构建一个和mask维度一样，元素都是 1 的张量;
 - 使用 `tf.equal` 把mask从 int 类型转成 bool 类型。`tf.equal`作用是判断两个输入是否相等，相等是True，不等就是False;
- 转换query维度，将query变为和 facts 同样的形状 $B * T * H$; 这里 T 随着每个具体训练数据不同而不同，比如某一个用户的某一个时间序列长度是5，另一个时间序列是15;
 - query是[B, H]，转换到 queries 维度为(B, T, H)。为了让pos_item和用户行为序列中每个元素计算权重。这里是用 `tf.tile(query, [1, tf.shape(facts)[1]])`。
`tf.shape(keys)[1]` 结果就是 T，query是[B, H]，经过 tile，就是把第一维按照 T 展开，得到[B, T * H] ;
 - 把 queries 进行 reshape，转换成和 facts 相同的大小: [B, T, H];
- 在MLP之前多做一些捕获行为item和候选item之间关系的操作：加减乘除等。然后得到了 Local Activation Unit 的输入。即 候选广告 queries 对应的 emb，用户历史行为序列 facts 对应的 embed，再加上它们之间的交叉特征, 进行 concat 后的结果;
- attention操作，目的是计算query和key的相关程度。通过三层神经网络得到queries和 facts 中每个key的权重，这个DNN 网络的输出节点为 1;
 - 最后一步 d_layer_3_all 的 shape 为 [B, T, 1];
 - 然后 reshape 为 [B, 1, T], axis=2 这一维表示 T 个用户行为序列分别对应的权重参数;
 - attention的输出, [B, 1, T];
- 得到有真实意义的score;
 - 使用 `key_masks = tf.expand_dims(mask, 1)` 把mask扩展维度，从 [B, T] 扩展到 [B, 1, T];
 - 使用 `tf.ones_like(scores)` 构建一个和scores维度一样，元素都是 1 的张量;
 - padding的mask后补一个很小的负数，这样后面计算 softmax 时, $e^{\{x\}}$ 结果就约等于 0;
 - 进行 [B, 1, T] padding操作。为了忽略了padding对总体的影响，代码中利用`tf.where` 将padding的向量(每个样本序列中空缺的商品)权重置为极小值($-2 ** 32 + 1$)，而不是

0;

- 利用 `tf.where(key_masks, scores, paddings)` 来得到真正有意义的score;
- Scale 是 attention的标准操作, 做完scaled后再送入softmax得到最终的权重。但是代码中没有用这部分, 注销掉了;
- 经过softmax进行标准化, 得到归一化后的权重;
- 这里已经得到了正确的权重 scores 以及用户历史行为序列 facts, 所以通过weighted sum 得到最终用户的兴趣表征;
 - 首先把 scores 进行reshape, 从 [B, 1, H] 变化成 Batch * Time;
 - 并且用expand_dims来把scores在最后增加一维;
 - 然后进行哈达码积, $[B, T, H] \times [B, T, 1] = [B, T, H]$;
 - 最后 reshape 成 Batch * Time * Hidden Size;
 - 如果是 SUM mode, 则进行矩阵相乘得到用户的兴趣表征; 具体是scores 的大小为 [B, 1, T], 表示每条历史行为的权重, facts 为历史行为序列, 大小为 [B, T, H], 两者用矩阵乘法做, 得到的结果 output 就是 [B, 1, H]。
 - 否则 进行哈达码相乘。

具体代码如下:

```
def din_fcn_attention(query, facts, attention_size, mask, stag='null', mode='SUM', softmax_st
...

query : 候选广告, shape: [B, H], 即i_emb;
facts : 用户历史行为, shape: [B, T, H], 即h_emb, T是padding后的长度, 每个长H的emb代表一个item;
mask : Batch中每个行为的真实意义, shape: [B, H];
...

if isinstance(facts, tuple):
    # In case of Bi-RNN, concatenate the forward and the backward RNN outputs.
    facts = tf.concat(facts, 2)

if len(facts.get_shape().as_list()) == 2:
    facts = tf.expand_dims(facts, 1)

if time_major:
    # (T,B,D) => (B,T,D)
    facts = tf.array_ops.transpose(facts, [1, 0, 2])

# Trainable parameters
mask = tf.equal(mask, tf.ones_like(mask))
facts_size = facts.get_shape().as_list()[-1] # D value - hidden size of the RNN layer
query_size = query.get_shape().as_list()[-1] # H, 这里是36
```

```

# 和DIN attention不同

query = tf.layers.dense(query, facts_size, activation=None, name='f1' + stag)
query = prelu(query)

# 1. 转换query维度, 变成历史维度T
# query是[B, H], 转换到 queries 维度为(B, T, H), 为了让pos_item和用户行为序列中每个元素计算权重
# 此时query是 Tensor("concat:0", shape=(?, 36), dtype=float32)
# tf.shape(keys)[1] 结果就是 T, query是[B, H], 经过tile, 就是把第一维按照 T 展开, 得到[B, T * H]
queries = tf.tile(query, [1, tf.shape(facts)[1]]) # [B, T * H], 想象成贴瓷砖
# 此时 queries 是 Tensor("Attention_Layer/Tile:0", shape=(?, ?), dtype=float32)
# queries 需要 reshape 成和 facts 相同的大小: [B, T, H]
queries = tf.reshape(queries, tf.shape(facts)) # [B, T * H] -> [B, T, H]
# 此时 queries 是 Tensor("Attention_Layer/Reshape:0", shape=(?, ?, 36), dtype=float32)

# 2. 这部分目的就是为了在MLP之前多做一些捕获行为item和候选item之间关系的操作: 加减乘除等。
# 得到 Local Activation Unit 的输入。即 候选广告 queries 对应的 emb, 用户历史行为序列 facts
# 对应的 embed, 再加上它们之间的交叉特征, 进行 concat 后的结果
din_all = tf.concat([queries, facts, queries-facts, queries*facts], axis=-1) # T*[B,H] -> [B,T,4H]

# 3. attention操作, 通过几层MLP获取权重, 这个DNN 网络的输出节点为 1
d_layer_1_all = tf.layers.dense(din_all, 80, activation=tf.nn.sigmoid, name='f1_att' + stag)
d_layer_2_all = tf.layers.dense(d_layer_1_all, 40, activation=tf.nn.sigmoid, name='f2_att' + stag)
d_layer_3_all = tf.layers.dense(d_layer_2_all, 1, activation=None, name='f3_att' + stag)
# 上一层 d_layer_3_all 的 shape 为 [B, T, 1]
# 下一步 reshape 为 [B, 1, T], axis=2 这一维表示 T 个用户行为序列分别对应的权重参数
d_layer_3_all = tf.reshape(d_layer_3_all, [-1, 1, tf.shape(facts)[1]])
scores = d_layer_3_all # attention的输出, [B, 1, T]

# 4. 得到有真实意义的score
# key_masks = tf.sequence_mask(facts_length, tf.shape(facts)[1]) # [B, T]
key_masks = tf.expand_dims(mask, 1) # [B, 1, T]
# padding的mask后补一个很小的负数, 这样后面计算 softmax 时, e^{x} 结果就约等于 0
paddings = tf.ones_like(scores) * (-2 ** 32 + 1) # 注意初始化为极小值
# [B, 1, T] padding操作, 为了忽略了padding对总体的影响, 代码中利用tf.where将padding的向量(每个元素都是-2**32+1)替换掉
if not forCnn:
    scores = tf.where(key_masks, scores, paddings) # [B, 1, T]

# 5. Scale # attention的标准操作, 做完scaled后再送入softmax得到最终的权重。
# scores = scores / (facts.get_shape().as_list()[-1] ** 0.5)

# 6. Activation, 得到归一化后的权重
if softmax_stag:

```

```

scores = tf.nn.softmax(scores) # [B, 1, T]

# 7. 得到了正确的权重 scores 以及用户历史行为序列 facts, 再进行矩阵相乘得到用户的兴趣表征
# Weighted sum,
if mode == 'SUM':
    # scores 的大小为 [B, 1, T], 表示每条历史行为的权重,
    # facts 为历史行为序列, 大小为 [B, T, H];
    # 两者用矩阵乘法做, 得到的结果 output 就是 [B, 1, H]
    #  $B * 1 * H$  三维矩阵相乘, 相乘发生在后两维, 即  $B * ((1 * T) * (T * H))$ 
    # 这里的output是attention计算出来的权重, 即论文公式(3)里的w,
    output = tf.matmul(scores, facts) # [B, 1, H]
    # output = tf.reshape(output, [-1, tf.shape(facts)[-1]])
else:
    # 从 [B, 1, H] 变化成 Batch * Time
    scores = tf.reshape(scores, [-1, tf.shape(facts)[1]])
    # 先把scores在最后增加一维, 然后进行哈达码积,  $[B, T, H] \times [B, T, 1] = [B, T, H]$ 
    output = facts * tf.expand_dims(scores, -1)
    output = tf.reshape(output, tf.shape(facts)) # Batch * Time * Hidden Size
return output

```

5.2.3 VecAttGRUCell

接下来, 就是AUGRU的结构, 这里设计一个新的VecAttGRUCell结构。具体RNN Cell的原理, 会有文论述。

基于深度学习的文本分类, 同样面临着如何将一段话中的多个词向量压缩成一个向量来表示这段话的问题。常用的方法, 就是将多个词向量喂入RNN, 最后一个时刻RNN的输出向量就代表了多个词向量的“合并”结果。DIEN则借鉴了这一思路, 并且改造了GRU的构造, 利用attention score来控制门。

阿里在这里做的修改主要是call函数, 是关于att_score的修改:

```

u = (1.0 - att_score) * u
new_h = u * state + (1 - u) * c
return new_h, new_h

```

具体代码是:

```

def call(self, inputs, state, att_score=None):

```

```

.....
c = self._activation(self._candidate_linear([inputs, r_state]))
u = (1.0 - att_score) * u # 这里是新增加的

new_h = u * state + (1 - u) * c # 这里是新增加的

return new_h, new_h

```

5.2.4 计算兴趣进化过程

设计好了新的GRU Cell，我们就能计算兴趣的进化过程，这就是 'rnn_2' 完成的。

```

with tf.name_scope('rnn_2'):
    rnn_outputs2, final_state2 = dynamic_rnn(VecAttGRUCell(HIDDEN_SIZE), inputs=rnn_outputs,
                                              att_scores = tf.expand_dims(alphas, -1),
                                              sequence_length=self.seq_len_ph, dtype=tf.float32,
                                              scope="gru2")

```

5.2.5 计算全联接层输入

得到兴趣进化的结果final_state2之后，需要与其他的embedding进行拼接，得到全联接层的输入：

```

inp = tf.concat([self.uid_batch_embedded, self.item_eb, self.item_his_eb_sum, self.item_eb *

```

0x06 全连接层

现在我们得到了连接后的稠密表示向量，接下来就是利用全连通层自动学习特征之间的非线性关系组合。

于是通过一个多层神经网络，得到最终的ctr预估值，这部分就是一个函数调用。

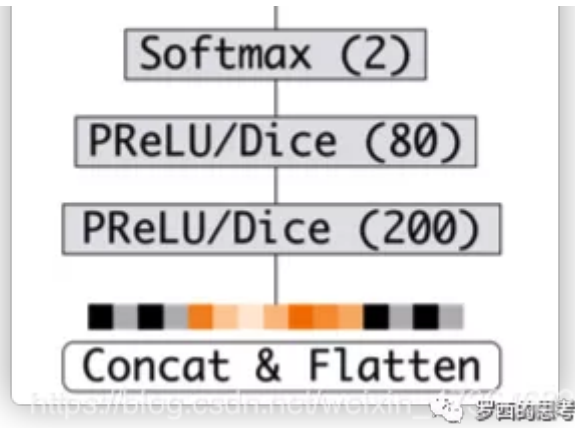
```

# Fully connected layer
self.build_fcn_net(inp, use_dice=True)

```

对应论文中的：

Output



在这里插入图片描述

其中逻辑如下：

- 首先进行Batch Normalization;
- 加入一个全连接层 `tf.layers.dense(bn1, 200, activation=None, name='f1')`;
- 用 dice 或者 prelu 进行激活;
- 加入一个全连接层 `tf.layers.dense(dnn1, 80, activation=None, name='f2')`;
- 用 dice 或者 prelu 进行激活;
- 加入一个全连接层 `tf.layers.dense(dnn2, 2, activation=None, name='f3')`;
- 得到输出 `y_hat = tf.nn.softmax(dnn3) + 0.00000001`;
- 进行交叉熵和optimizer初始化;
 - 得到交叉熵 - `tf.reduce_mean(tf.log(self.y_hat) * self.target_ph)`;
 - 如果有负采样，需要加上辅助损失;
 - 使用 AdamOptimizer;
- 计算 Accuracy;

具体代码参见如下：

```
def build_fcn_net(self, inp, use_dice = False):
    bn1 = tf.layers.batch_normalization(inputs=inp, name='bn1')
    dnn1 = tf.layers.dense(bn1, 200, activation=None, name='f1')
    if use_dice:
        dnn1 = dice(dnn1, name='dice_1')
    else:
        dnn1 = prelu(dnn1, 'prelu1')

    dnn2 = tf.layers.dense(dnn1, 80, activation=None, name='f2')
    if use_dice:
        dnn2 = dice(dnn2, name='dice_2')
```

```

else:
    dnn2 = prelu(dnn2, 'prelu2')
dnn3 = tf.layers.dense(dnn2, 2, activation=None, name='f3')
self.y_hat = tf.nn.softmax(dnn3) + 0.00000001

with tf.name_scope('Metrics'):
    # Cross-entropy Loss and optimizer initialization
    ctr_loss = - tf.reduce_mean(tf.log(self.y_hat) * self.target_ph)
    self.loss = ctr_loss
    if self.use_negsampling:
        self.loss += self.aux_loss
    tf.summary.scalar('loss', self.loss)
    self.optimizer = tf.train.AdamOptimizer(learning_rate=self.lr).minimize(self.loss)

    # Accuracy metric
    self.accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.round(self.y_hat), self.target_ph)
    tf.summary.scalar('accuracy', self.accuracy)

self.merged = tf.summary.merge_all()

```

0x07 训练模型

通过 `model.train` 来训练模型。

`model.train` 的输入数据有：

- 用户id;
- target的item id;
- target item对应的cateid;
- 用户历史行为的item id list;
- 用户历史行为item对应的cate id list;
- 历史行为的mask;
- 目标值;
- 历史行为的长度;
- learning rate;
- 负采样的数据;

`train`代码具体如下：

```
def train(self, sess, inps):
```

```
if self.use_negsampling:
    loss, accuracy, aux_loss, _ = sess.run([self.loss, self.accuracy, self.aux_loss, self
        self.uid_batch_ph: inps[0],
        self.mid_batch_ph: inps[1],
        self.cat_batch_ph: inps[2],
        self.mid_his_batch_ph: inps[3],
        self.cat_his_batch_ph: inps[4],
        self.mask: inps[5],
        self.target_ph: inps[6],
        self.seq_len_ph: inps[7],
        self.lr: inps[8],
        self.noclk_mid_batch_ph: inps[9],
        self.noclk_cat_batch_ph: inps[10],
    ])
    return loss, accuracy, aux_loss
else:
    loss, accuracy, _ = sess.run([self.loss, self.accuracy, self.optimizer], feed_dict={
        self.uid_batch_ph: inps[0],
        self.mid_batch_ph: inps[1],
        self.cat_batch_ph: inps[2],
        self.mid_his_batch_ph: inps[3],
        self.cat_his_batch_ph: inps[4],
        self.mask: inps[5],
        self.target_ph: inps[6],
        self.seq_len_ph: inps[7],
        self.lr: inps[8],
    })
    return loss, accuracy, 0
```

下一篇文章将介绍RNN Cell，敬请期待。

0xFF 参考

用NumPy手工打造 Wide & Deep

看Google如何实现Wide & Deep模型(1)

看Youtube怎么利用深度学习做推荐

也评Deep Interest Evolution Network

从DIN到DIEN看阿里CTR算法的进化脉络

第七章 人工智能，7.6 DNN在搜索场景中的应用(作者：仁重)

#Paper Reading# Deep Interest Network for Click-Through Rate Prediction

【paper reading】Deep Interest Evolution Network for Click-Through Rate Prediction

也评Deep Interest Evolution Network

论文阅读：《Deep Interest Evolution Network for Click-Through Rate Prediction》

【论文笔记】Deep Interest Evolution Network(AAAI 2019)

【读书笔记】Deep Interest Evolution Network for Click-Through Rate Prediction

DIN(Deep Interest Network):核心思想+源码阅读注释

计算广告CTR预估系列(五)--阿里Deep Interest Network理论

CTR预估之Deep Interest NetWork模型原理详解

人人都能看懂的LSTM

从动图中理解 RNN，LSTM 和 GRU

台大李宏毅机器学习(一)——RNN&LSTM

李宏毅机器学习(2016)

推荐系统遇上深度学习(二十四)--深度兴趣进化网络DIEN原理及实战！

from google.protobuf.pyext import _message，使用tensorflow出现 ImportError: DLL load failed

DIN 深度兴趣网络介绍以及源码浅析

CTR预估 论文精读(八)--Deep Interest Network for Click-Through Rate Prediction

阿里CTR预估三部曲(1)：Deep Interest Network for Click-Through Rate Prediction简析

阿里CTR预估三部曲(2)：Deep Interest Evolution Network for Click-Through Rate Prediction简析

Deep Interest Network解读

深度兴趣网络(DIN,Deep Interest Network)

DIN论文官方实现解析

阿里DIN源码之如何建模用户序列(1): base方案

阿里DIN源码之如何建模用户序列(2): DIN以及特征工程看法

阿里深度兴趣网络(DIN)论文翻译

推荐系统遇上深度学习(二十四)--深度兴趣进化网络DIEN原理及实战!

推荐系统遇上深度学习(十八)--探秘阿里之深度兴趣网络(DIN)浅析及实现

【论文导读】2018阿里CTR预估模型---DIN(深度兴趣网络), 后附TF2.0复现代码

【论文导读】2019阿里CTR预估模型---DIEN(深度兴趣演化网络)

推荐系统---深度兴趣网络DIN&DIEN

深度学习中的attention机制

Transform详解(超详细) Attention is all you need论文

阅读原文

喜欢此内容的人还喜欢

[从源码学设计]蚂蚁金服SOFARegistry之推拉模型

罗西的思考

从两张图看成为发达国家有多难

宁南山

面对青春期孩子, 做好这3点, 你才能“活”下来

团长说心理