[阿里DIN] 深度兴趣网络源码分析 之 整体代码结构

原创 罗西的思考 罗西的思考 10月22日

[阿里DIN] 深度兴趣网络源码分析 之 整体代码结构

0x00 摘要

Deep Interest Network（DIN）是阿里妈妈精准定向检索及基础算法团队在2017年6月提出的。其针对电子商务领域（e-commerce industry）的CTR预估，重点在于充分利用/挖掘用户历史行为数据中的信息。

本文为系列第三篇，将分析DIN源码整体思路。采用的是 https://github.com/mouna99/dien 中的实现。

因为此项目包括 DIN，DIEN 等几个模型，所以部分文件是 DIEN 模型使用，这里也顺带提一下，后续会有专门文章讲解。

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；

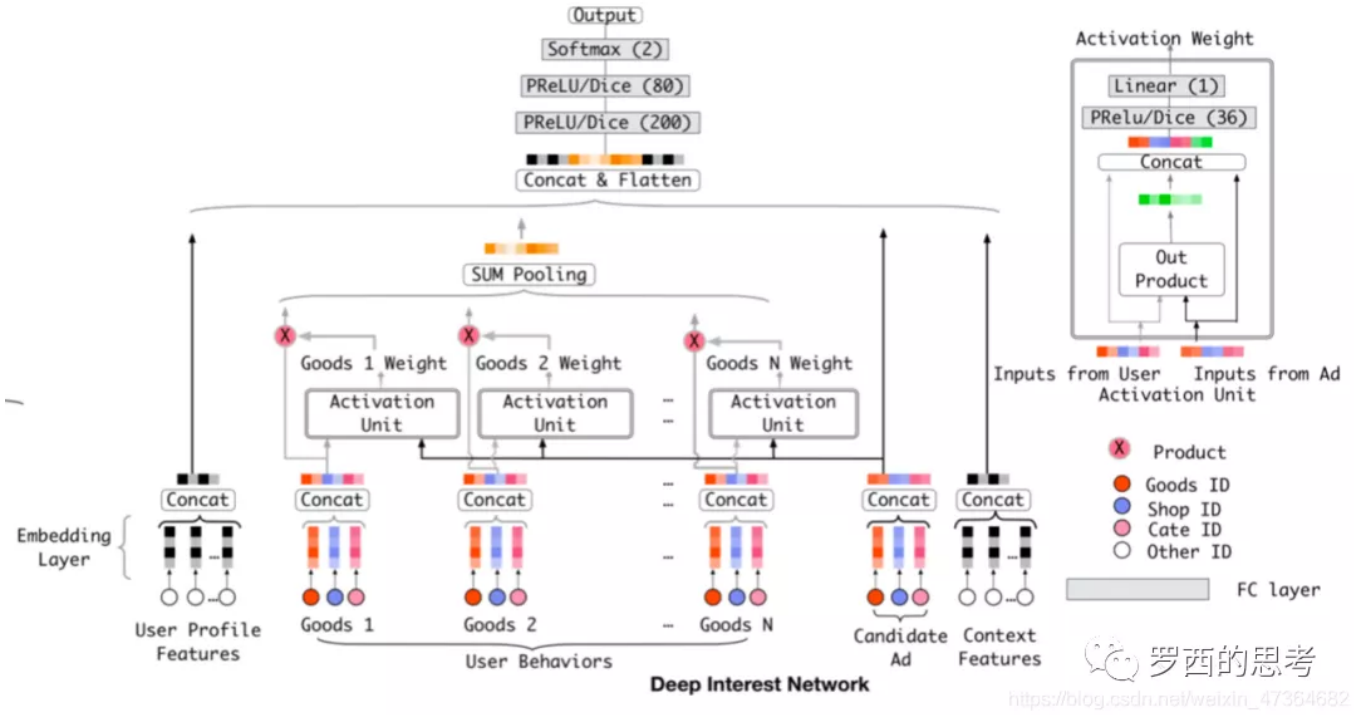
代码主要包含：

- **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

DIN 试图捕获之前点击的 item 和目标 item 之间的不同相似性。

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

- Deep Interest NetWork有以下几点创新：

1. **针对Diversity：** 针对用户广泛的兴趣，DIN用*an interest distribution*去表示，即用 Pooling（weighted sum）对Diversity建模（对用户多种多样的兴趣建模）。
2. **针对Local Activation：**利用attention机制实现 Local Activation，从用户历史行为中动态学习用户兴趣的embedding向量，针对不同的广告构造不同的用户抽象表示，从而实现了在数据维度一定的情况下，更精准地捕捉用户当前的兴趣。对用户历史行为进行了不同的加权处理，针对不同的广告，用户历史行为的权重不一致。即针对当前候选Ad，去局部的激活（*Local Activate*）相关的历史兴趣信息。和当前候选Ad相关性越高的历史行为，会获得更高的*attention score*，从而会主导这一次预测。
3. CTR中**特征稀疏而且维度高**，通常利用L1、L2、Dropout等手段防止过拟合。由于传统L2正则计算的是全部参数，CTR预估场景的模型参数往往数以亿计。DIN提出了一种正则化方法，在每次小批量迭代中，给与不同频次的特征不同的正则权重；
4. 由于传统的**激活函数**，如Relu在输入小于0时输出为0，将导致许多网络节点的迭代速度变慢。PRelu虽然加快了迭代速度，但是其分割点默认为0，实际上分割点应该由数据决定。因此，DIN提出了一种数据动态自适应激活函数Dice。
5. **针对大规模稀疏数据的模型训练：**当DNN深度比较深（参数非常多），输入又非常稀疏的时候，很容易过拟合。DIN提出**Adaptive regularizaion**来防止过拟合，效果显著。

0x03 总体代码

DIN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, ma
        ## 测试数据
        test_data = DataIterator(test_file, uid_voc, mid_voc, cat_voc, batch_size, maxl
        n_uid, n_mid, n_cat = train_data.get_n()

        ......

        elif model_type == 'DIN':
            model = Model_DIN(n_uid, n_mid, n_cat, EMBEDDING_DIM, HIDDEN_SIZE, ATTENTIC
        elif model_type == 'DIEN':
            model = Model_DIN_V2_Gru_Vec_attGru_Neg(n_uid, n_mid, n_cat, EMBEDDING_DIM,

        ......

        sess.run(tf.global_variables_initializer())
        sess.run(tf.local_variables_initializer())

        iter = 0
        lr = 0.001
        for itr in range(3):
            loss_sum = 0.0
            accuracy_sum = 0.
            aux_loss_sum = 0.
            for src, tgt in train_data:
                uids, mids, cats, mid_his, cat_his, mid_mask, target, sl, noclk_mids, r
                loss, acc, aux_loss = model.train(sess, [uids, mids, cats, mid_his, cat
                loss_sum += loss
                accuracy_sum += acc
                aux_loss_sum += aux_loss
                iter += 1
                if (iter % test_iter) == 0:
                    eval(sess, test_data, model, best_model_path)
```

```
                loss_sum = 0.0
                accuracy_sum = 0.0
                aux_loss_sum = 0.0
            if (iter % save_iter) == 0:
                model.save(sess, model_path+"--"+str(iter))
        lr *= 0.5
```

0x04 模型基类

模型的基类是 Model，其构造函数 __init__ 可以理解为 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。

4.1 基本逻辑

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_
    # 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_
    # 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='nc
        self.noclk_cat_batch_ph = tf.placeholder(tf.int32, [None, None, None], name='nc
```

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

 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=(?, ?, ?), d
 noclk_mid_batch_ph = {Tensor} Tensor("Inputs/noclk_mid_batch_ph:0", shape=(?, ?, ?), d

 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 c
    self.uid_embeddings_var = tf.get_variable("uid_embedding_var", [n_uid, EMBEDDING_DI
    # 从uid embedding weight 中取出 uid embedding vector
    self.uid_batch_embedded = tf.nn.embedding_lookup(self.uid_embeddings_var, self.uid_

    # shape: [I, H/2], item_id的embedding weight. I是item_id的hash bucket size，即movie
    self.mid_embeddings_var = tf.get_variable("mid_embedding_var", [n_mid, EMBEDDING_DI
    # 从mid embedding weight 中取出 uid embedding vector
    self.mid_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, self.mid_
    # 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
    self.mid_his_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, self.
    # 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_

    # 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_DI
    # 从 cid embedding weight 中取出 cid history embedding vector，是正样本
    # 比如cat_embeddings_var 是(1601, 18)，cat_batch_ph 是(?,)，则cat_batch_embedded 就是
    self.cat_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat_
    # 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.
    # 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_
```

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) dt
 uid_embeddings_var = {Variable} <tf.Variable 'uid_embedding_var:0' shape=(543060, 18)
 mid_embeddings_var = {Variable} <tf.Variable 'mid_embedding_var:0' shape=(367983, 18)

 cat_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_4:0", shape=(?,
 mid_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_1:0", shape=(?,
 uid_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup:0", shape=(?, 1

 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",
 noclk_mid_his_batch_embedded = {Tensor} Tensor("Embedding_layer/embedding_lookup_3:0",
```

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;

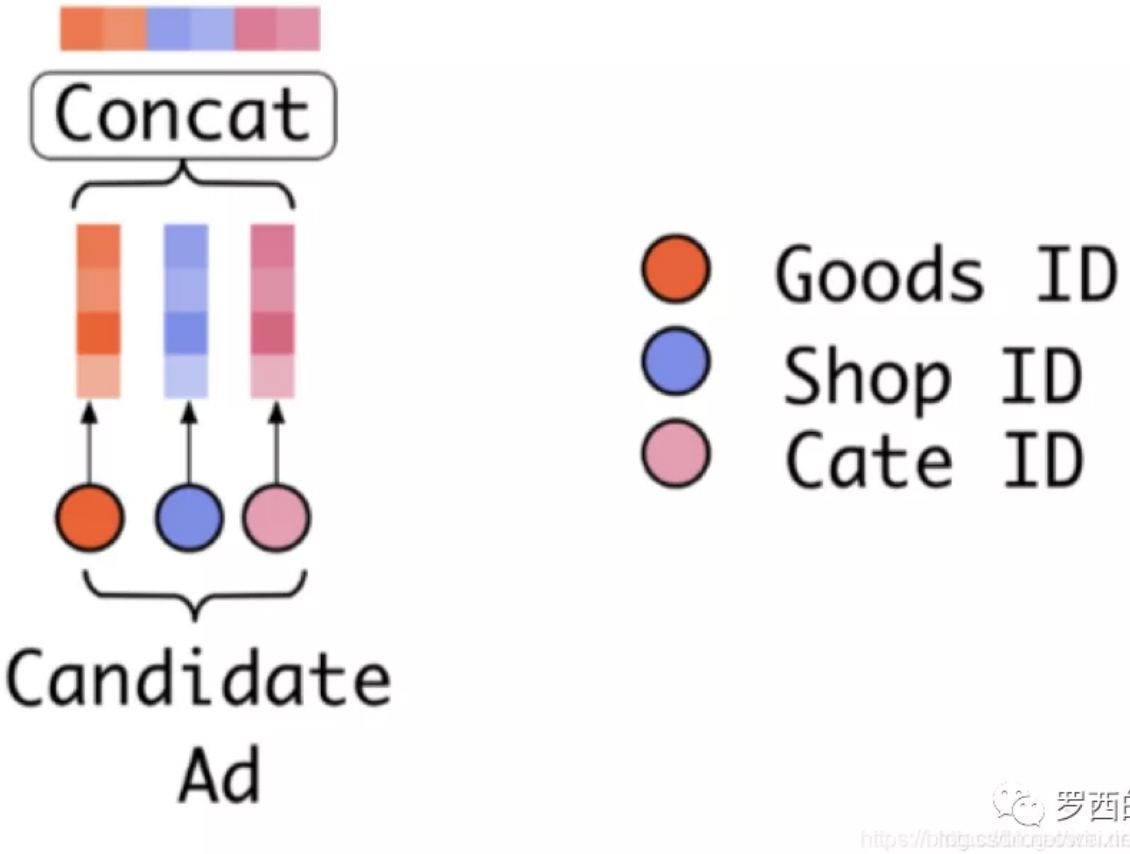
关于shape的说明：

- 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_eb = 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。比如 [[mid1, mid2] , [mid3, mid4]] 和 [[cid1, cid2], [cid3, cid4]] ，拼接得到 [[mid1, mid2,cid1, cid2] , [mid3, mid4，cid3, cid4]] 。

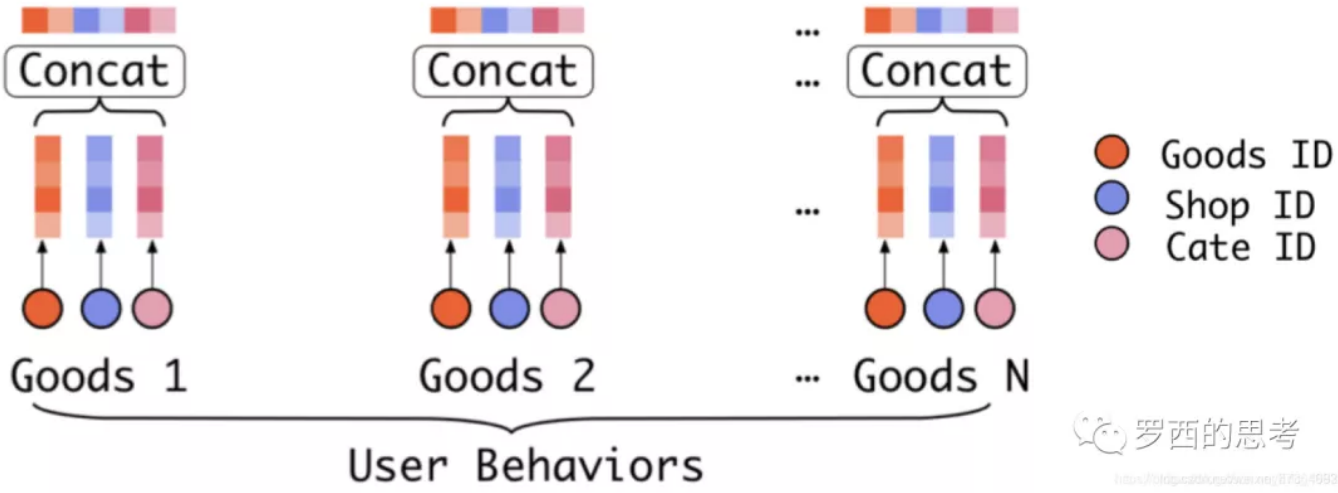
对应了架构图的：



在这里插入图片描述

第二步是 `self.item_his_eb = 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。比如 `[[[mid1, mid2]]]` 和 `[[[cid1, cid2]]]`，拼接得到 `[[[mid1, mid2, cid1, cid2]]]` 。

对应了架构图的:



在这里插入图片描述

第三步是用 tf.reduce_sum(self.item_his_eb, 1) 按照第一维度求和，会降维。

比如 [[[mid1, mid2,cid1, cid2] , [mid3, mid4, cid3, cid4]]] 得到 [[mid1 + mid3, mid2 + mid4, cid1 + cid3, cid2 + cid4]] 。

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 embec
self.item_eb = 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 a
self.item_his_eb = tf.concat([self.mid_his_batch_embedded, self.cat_his_batch_embedded]
# 按照第一维度求和，会降维
self.item_his_eb_sum = tf.reduce_sum(self.item_his_eb, 1)
# 举例如下，item_eb是 (128, 36)，item_his_eb是(128, ?, 36)，这个是从真实数据读取出来的，比如

# 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 embec
if self.use_negsampling:
    # 0 means only using the first negative item ID. 3 item IDs are inputed in the line
    self.noclk_item_his_eb = tf.concat(
        [self.noclk_mid_his_batch_embedded[:, :, 0, :], self.noclk_cat_his_batch_embedc
    # cat embedding 18 concate item embedding 18.
    self.noclk_item_his_eb = tf.reshape(self.noclk_item_his_eb,
                                        [-1, tf.shape(self.noclk_mid_his_batch_embeddec
    self.noclk_his_eb = tf.concat([self.noclk_mid_his_batch_embedded, self.noclk_cat_hi
    self.noclk_his_eb_sum_1 = tf.reduce_sum(self.noclk_his_eb, 2)
    self.noclk_his_eb_sum = tf.reduce_sum(self.noclk_his_eb_sum_1, 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

Model_DIN 是DIN实现的模型。

```
class Model_DIN(Model):
    def __init__(self, n_uid, n_mid, n_cat, EMBEDDING_DIM, HIDDEN_SIZE, ATTENTION_SIZE,
        super(Model_DIN, self).__init__(n_uid, n_mid, n_cat, EMBEDDING_DIM, HIDDEN_SIZE
                                        ATTENTION_SIZE,
```

```
                                                    use_negsampling)

        # Attention layer
        with tf.name_scope('Attention_layer'):
            attention_output = din_attention(self.item_eb, self.item_his_eb, ATTENTION_
            att_fea = tf.reduce_sum(attention_output, 1)

        inp = tf.concat([self.uid_batch_embedded, self.item_eb, self.item_his_eb_sum, s
        # Fully connected layer
        self.build_fcn_net(inp, use_dice=True)
```

整体思路比较简单：

- Attention layer
- Fully connected layer

具体分析如下。

5.1 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：

$$\mathbf{Attention(Query, Source)} = \sum_{i=1}^{L_x} \boldsymbol{Similarity(Query, Key_i) * Value_i}$$

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。

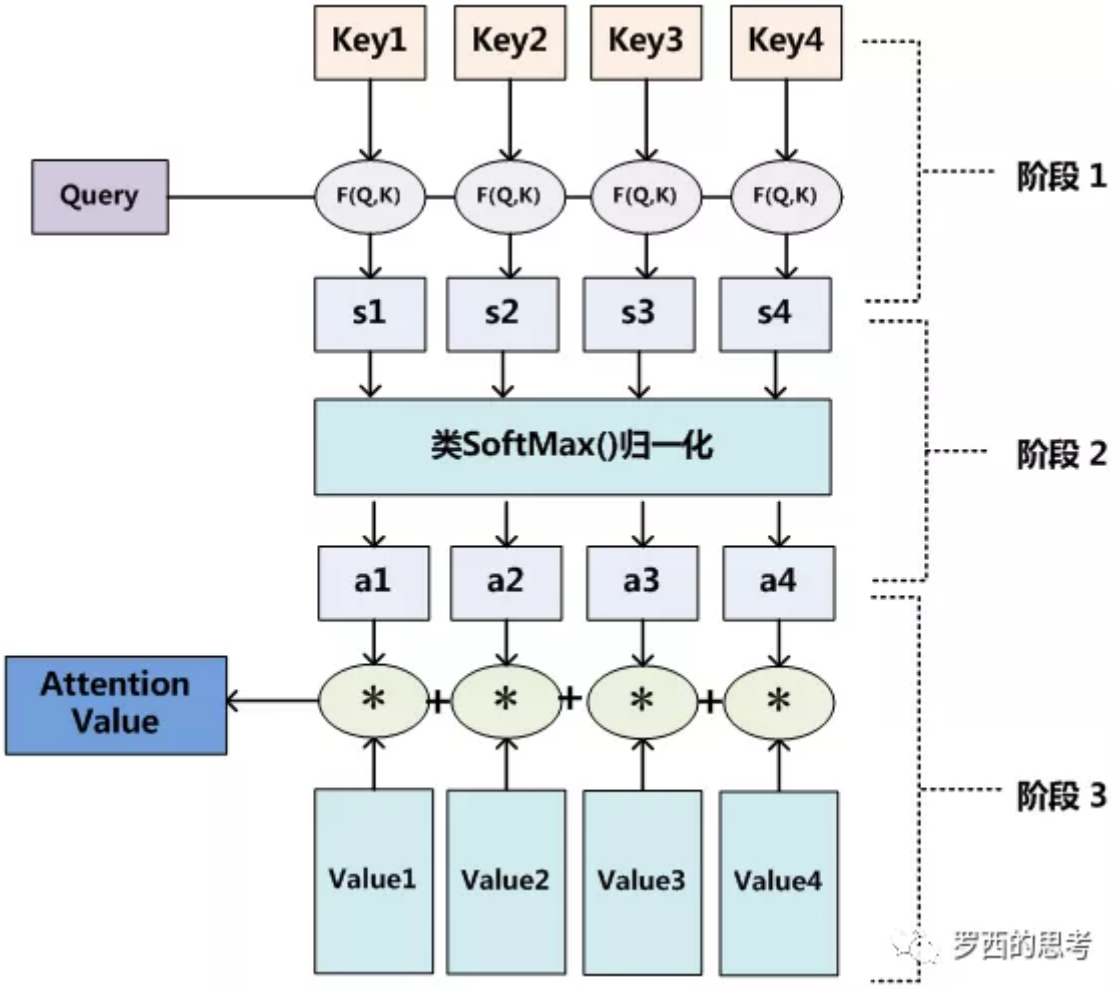
至于Attention机制的具体计算过程，如果对目前大多数方法进行抽象的话，可以将其归纳为两个过程：

- 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小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。



这里写图片描述

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Keyi，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： $\mathbf{Similarity}(\mathbf{Query}, \boldsymbol{Key}_i) = \mathbf{Query} \cdot \boldsymbol{Key}_i$

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

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

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：

$$\boldsymbol{a}_i = \boldsymbol{Softmax}(\boldsymbol{Sim}_i) = \frac{e^{\boldsymbol{Sim}_i}}{\sum_{j=1}^{L_x} e^{\boldsymbol{Sim}_i}}$$

这里写图片描述

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

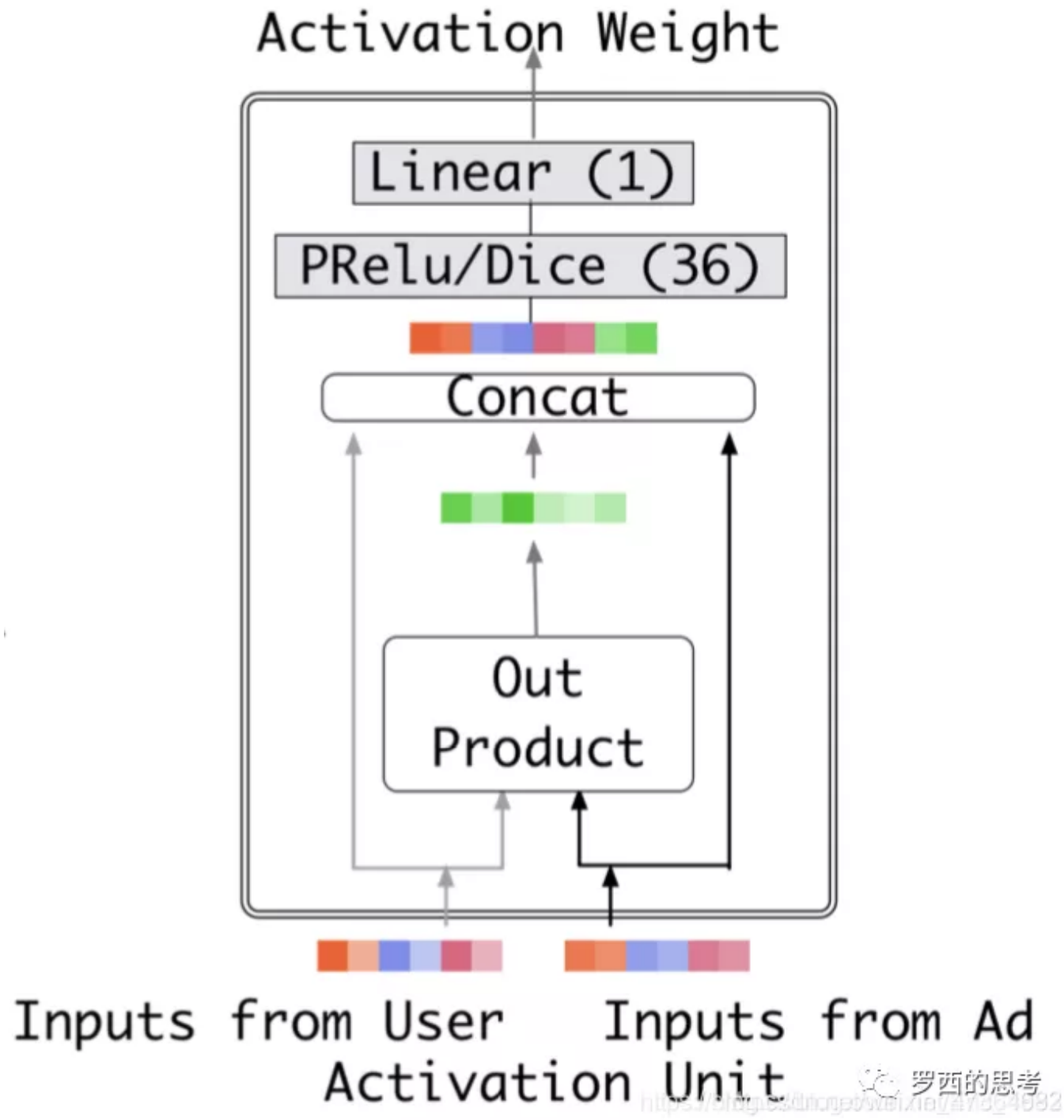
$$\mathbf{Attention}(\mathbf{Query}, \mathbf{Source}) = \sum_{i=1}^{L_x} \boldsymbol{a}_i \boldsymbol{Value}_i$$

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 Attention实现

DIN中会对于用户的行为序列，将其中每个item的所有field特征concat后形成该item的临时emb之后，不再是对序列中所有临时item emb做简单的sum pooling，而是对每个item emb计算和候选item emb的相关性权重，即activation unit模块。



在这里插入图片描述

这部分功能实现在attention中：

5.2.1 调用

如何调用：

```
attention_output = din_attention(self.item_eb, self.item_his_eb, ATTENTION_SIZE, self.m
```

其中，相关参数等：

- query： 候选广告对应的 embedding，shape: [B, H], 即 i_emb；
- facts： 用户历史行为对应的 embedding，shape: [B, T, H], 即 h_emb；
- mask : Batch中每个行为的真实意义，shape: [B, H]，由于一个 Batch 中的用户行为序列不一定都相同，但是输入的keys维度是固定的(都是历史行为最大的长度)，其真实长度保存在 self.sl 中，所以之前产生了 masks 来选择真正的历史行为，以告诉模型哪些行为是没用的，哪些是用来计算用户兴趣分布的；
- B:batch size; T: 用户历史行为序列的最大长度；H：embedding size；
- attention_output： 输出为用户兴趣的表征；

参数变量动态如下：

```
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)
 mask = {Tensor} Tensor("Inputs/mask:0", shape=(?, ?), dtype=float32)
```

5.2.2 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。

5.2.3 基本逻辑

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(facts) 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。这里是用了 `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。具体是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；
  - 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。
    - 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] x [B, T, 1] = [B, T, H]；
    - 最后 reshape 成 Batch * Time * Hidden Size；

具体代码如下：

```
def din_attention(query, facts, attention_size, mask, stag='null', mode='SUM', softmax_
    '''
    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
    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)
        print ("querry_size mismatch")
        query = tf.concat(values = [
        query,
        query,
        ], axis=1)

    if time_major:
```

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

    # 转换mask
    mask = tf.equal(mask, tf.ones_like(mask))
    facts_size = facts.get_shape().as_list()[-1]  # D value - hidden size of the RNN la
    querry_size = query.get_shape().as_list()[-1] # H，这里是36

    # 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
    # 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[
    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=float

    # 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 f
    # 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*[E

    # 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
    d_layer_2_all = tf.layers.dense(d_layer_1_all, 40, activation=tf.nn.sigmoid, name='
    d_layer_3_all = tf.layers.dense(d_layer_2_all, 1, activation=None, name='f3_att' +
      # 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
    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] x [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
```

程序运行时候的变量如下：

```
attention_size = {int} 36
d_layer_1_all = {Tensor} Tensor("Attention_layer/f1_attnull/Sigmoid:0", shape=(?, ?, 80
d_layer_2_all = {Tensor} Tensor("Attention_layer/f2_attnull/Sigmoid:0", shape=(?, ?, 40
d_layer_3_all = {Tensor} Tensor("Attention_layer/Reshape_1:0", shape=(?, 1, ?), dtype=f
din_all = {Tensor} Tensor("Attention_layer/concat:0", shape=(?, ?, 144), dtype=float32)
facts = {Tensor} Tensor("concat_1:0", shape=(?, ?, 36), dtype=float32)
facts_size = {int} 36
key_masks = {Tensor} Tensor("Attention_layer/ExpandDims:0", shape=(?, 1, ?), dtype=bool
mask = {Tensor} Tensor("Attention_layer/Equal:0", shape=(?, ?), dtype=bool)
mode = {str} 'SUM'
output = {Tensor} Tensor("Attention_layer/MatMul:0", shape=(?, 1, 36), dtype=float32)
paddings = {Tensor} Tensor("Attention_layer/mul_1:0", shape=(?, 1, ?), dtype=float32)
queries = {Tensor} Tensor("Attention_layer/Reshape:0", shape=(?, ?, 36), dtype=float32)
querry_size = {int} 36
query = {Tensor} Tensor("concat:0", shape=(?, 36), dtype=float32)
return_alphas = {bool} False
scores = {Tensor} Tensor("Attention_layer/Reshape_3:0", shape=(?, 1, ?), dtype=float32)
softmax_stag = {int} 1
stag = {str} 'null'
time_major = {bool} False
```

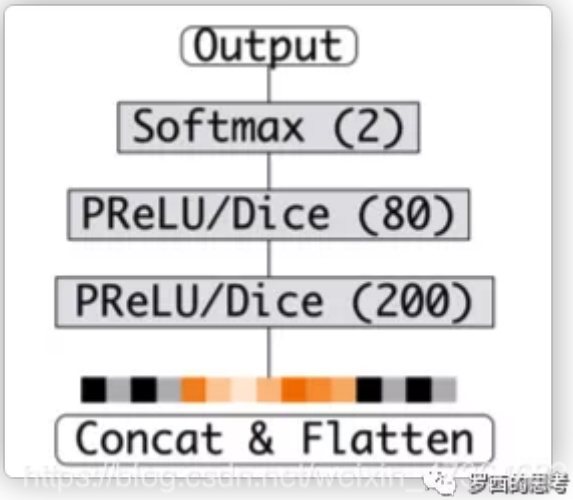
0x06 全连接层

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。

```
# Attention layers
with tf.name_scope('Attention_layer'):
  attention_output = din_attention(self.item_eb, self.item_his_eb, ATTENTION_SIZE, self
  att_fea = tf.reduce_sum(attention_output, 1)
  tf.summary.histogram('att_fea', att_fea)
  inp = tf.concat([self.uid_batch_embedded, self.item_eb, self.item_his_eb_sum, self.it
# Fully connected layer
self.build_fcn_net(inp, use_dice=True) # 调用多层神经网络
```

对应论文中的：



在这里插入图片描述

这个多层神经网络包含了多个全连接层，全连接层本质就是由一个特征空间线性变换到另一个特征空间。目标空间的任一维——也就是隐层的一个 cell——都认为会受到源空间的每一维的影响。可以不严谨的说，目标向量是源向量的加权和。

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 ，`tf.train.AdamOptimizer(learning_rate=self.lr).minimize(self.loss)` ，这样后续就会通过这个minimize来进行优化；
- 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.lc

            # Accuracy metric
            self.accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.round(self.y_hat), self.targ
            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.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_d
            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
```

0x08 AUC

提一下auc这个函数，起初以为是复杂算法，后来发现原来就是最淳朴的实现方式。

```
def calc_auc(raw_arr):
    arr = sorted(raw_arr, key=lambda d:d[0], reverse=True)
    pos, neg = 0., 0.
    for record in arr: # 先计算正样本，负样本个数
        if record[1] == 1.:
            pos += 1
        else:
            neg += 1

    fp, tp = 0., 0.
    xy_arr = []
```

```
    for record in arr:
        if record[1] == 1.:
            tp += 1
        else:
            fp += 1
        xy_arr.append([fp/neg, tp/pos])

    auc = 0.
    prev_x = 0.
    prev_y = 0.
    # 就是计算auc面积，y + prev_y = prev_y + prev_y + (y - prev_y)
    # y + prev_y 再乘以 delta_x，就是 2 * delta_x * prev_y + 2 * delta_x * prev_y
    # 再除以 2，正好就是梯形面积
    for x, y in xy_arr:
        if x != prev_x:
            auc += ((x - prev_x) * (y + prev_y) / 2.)
            prev_x = x
            prev_y = y

    return auc
```

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