

Attention机制的实现及其在社区资讯推荐中的应用 (tensorflow2)

原创 xulu1352 数据与智能 1月5日

点击上方“数据与智能”，“星标或置顶公众号”

第一时间获取好内容



数据与智能

用AI探索数据背后的秘密



长按关注我们

作者 | xulu1352 目前在一家互联网公司从事推荐算法工作 (知乎: xulu1352)

编辑 | lily

Attention机制的实现及其在社区资讯推荐中的应用 (tensorflow2)

0. 前序



Attention机制近年来在NLP领域大放异彩，尤其Bert等模型的走红，使Attention机制获得的关注量大增，那Attention机制应用到推荐领域又是以怎样形式的存在？说到这就不得不提阿里的深度兴趣网络(Deep Interest Network, DIN)，这个模型算得上是个经典的推荐系统Attention机制模型了；本文会重点围绕着DIN中Attention机制实现而展开，关于原理部分的解读本文下面只说说概要了，更深层次的解读可以参看文章末附录的文献。

1.Attention机制的思想



Attention机制缘起于人类视觉注意力机制，比如人们在看东西的时候一般会快速扫描全局，根据需求将观察焦点锁定在特定的位置上，是模仿人类注意力而提出的一种解决问题的办法；抽象点说它是一种权重参数的分配机制，目标是协助模型捕捉重要信息。具体一点就是，给定一组<key,value>，以及一个目标（查询）向量query，Attention机制就是通过计算query与各个key的相似性，得到每个key的权重系数，再通过对value加权求和，得到最终attention数值。所以本质上Attention机制是对给定元素的value值进行加权求和，而query和key用来计算对应value的权重系数。可以将其本质思想用如下公式来表达：

$$Attention(query, source) = \sum_{i=1}^n Similarity(query, key_i) * value_i \quad n \text{ is the length of source}$$

如果就目前大多数Attention计算方法进行抽象，可以将其归纳为两个过程三个阶段：

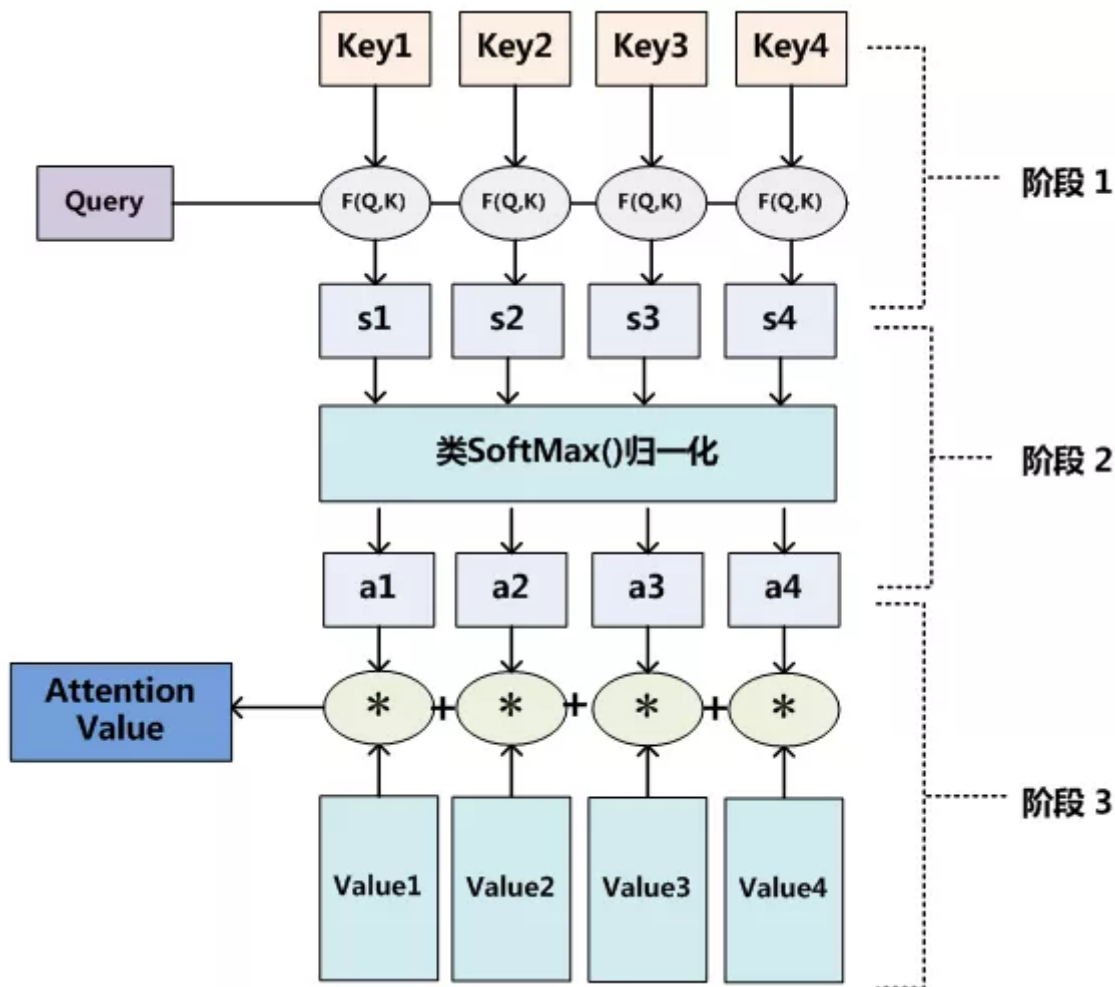
过程1：根据query和key计算权重系数，这一过程又可细分为两个阶段；

阶段1：根据query和key计算两者的相似性或者相关性；

阶段2：阶段1的原始分值进行归一化处理；

过程2：也即是阶段3，根据权重系数对value进行加权求和。

可以用下图来展示上述Attention计算过程的三个阶段。



2.Attention机制的实现

在讲述实现之前，这里先概括石塔西的一段论述作为铺叙。

深度学习应用于推荐算法，经典操作就是将高维、稀疏categorical/id类特征通过embedding映射成一个低维、稠密向量，但是，表达用户兴趣时，用户的历史行为往往涉及到多个稀疏categorical/id特征，比如点击过的多个商品、看过的多个视频、输入过的多个搜索词，需要将这些id特征embedding之后的多个低维向量，“合并”成一个向量，作为用户兴趣的表示，喂入DNN。这个“合并”就是所谓**Pooling**。

关于pooling，通常会用一个average/sum pooling层把用户交互过的所有物料embedding向量平均为一个定长的vector来作为用户的兴趣UE，但是用户的兴趣是多样性的(Diversity)，用户在点击某个物品往往是基于他历史部分的行为兴趣，而不是全部行为兴趣(Local activation)，可见如果把用户的历史行为映射到固定长度的低维向量，可能会丢失这部分信息（用户的历史行为中的物料Embedding对用户UE的贡献力度是一样）。

在DIN网络结构中，是通过Attention来实现Pooling，针对当前候选物料局部地激活用户的历史兴趣，赋予和候选物料相关的历史兴趣更高的weight，从而实现Local Activation，而weight的多样性同时也实现了用户兴趣的多样性表达。

在DIN的attention机制实现中，用户兴趣向量是历史上交互过的item embedding向量的加权平均，而第*i*个历史item的权重 W_i 由该历史item的embedding向量 V_i 与候选物料的embedding向量 V_a 共同决定（函数g）。可见同一个用户当面对不同候选物料时，其兴趣向量也不相同，从而实现了用户兴趣的“千物千面”。

$$V_u = f(v_a) = \sum_{i=1}^N w_i * V_i = \sum_{i=1}^N g(V_i, V_a) * V_i$$



Figure 2: Network Architecture. The left part illustrates the network of base model (Embedding&MLP). Embeddings of cate_id, shop_id and goods_id belong to one goods are concatenated to represent one visited goods in user's behaviors. Right part is our proposed DIN model. It introduces a local activation unit, with which the representation of user interests varies adaptively given different candidate ads.

原理如果了解差不多了，下面就来看看在实践中关于attention机制的一种代码实现。

```
1 class AttentionPoolingLayer(Layer):
2     """
3     Input shape
4         - A list of three tensor: [query,keys,his_seq]
5         - query is a 3D tensor with shape: ``(batch_size, 1, embedding_size)``
6         - keys is a 3D tensor with shape: ``(batch_size, T, embedding_size)``
7         - his_seq is a 2D tensor with shape: ``(batch_size, T)``
8     Output shape
9         - 3D tensor with shape: ``(batch_size, 1, embedding_size)``.
10    Arguments
11        - **att_hidden_units**:list of positive integer, the attention net lay
12        - **att_activation**: Activation function to use in attention net.
13        - **weight_normalization**: bool.Whether normalize the attention score
```

```

14     - **hist_mask_value**: the mask value of his_seq.
15     References
16     - [Zhou G, Zhu X, Song C, et al. Deep interest network for click-throu
17     """
18
19     def __init__(self, att_hidden_units=(80, 40), att_activation='sigmoid', we
20                 mode="sum", hist_mask_value=0, **kwargs):
21
22         self.att_hidden_units = att_hidden_units
23         self.att_activation = att_activation
24         self.weight_normalization = weight_normalization
25         self.mode = mode
26         self.hist_mask_value = hist_mask_value
27         super(AttentionLayer, self).__init__(**kwargs)
28
29
30     def build(self, input_shape):
31
32         self.fc = tf.keras.Sequential()
33         for unit in self.att_hidden_units:
34             self.fc.add(layers.Dense(unit, activation=self.att_activation, nar
35             self.fc.add(layers.Dense(1, activation=None, name="fc_att_out"))
36
37         super(AttentionLayer, self).build(
38             input_shape) # Be sure to call this somewhere!
39
40     def call(self, inputs, **kwargs):
41         query, keys, his_seq = inputs
42         # 计算掩码
43         key_masks = tf.not_equal(his_seq, tf.constant(self.hist_mask_value , c
44         key_masks = tf.expand_dims(key_masks, 1)
45
46         # 1. 转换query维度, 变成历史维度T
47         # query是[B, 1, H], 转换到 queries 维度为(B, T, H), 为了让pos_item和用户
48         queries = tf.tile(query, [1, tf.shape(keys)[1], 1]) # [B, T, H]
49
50         # 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.
51         # 得到 Local Activation Unit 的输入。即 候选queries 对应的 emb, 用户历史行
52         # 对应的 embed, 再加上它们之间的交叉特征, 进行 concat 后的结果
53         din_all = tf.concat([queries, keys, queries-keys, queries*keys], axis=

```

```

54 # 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
55 attention_score = self.fc(din_all) # [B, T, 1]
56 # attention的输出, [B, 1, T]
57 outputs = tf.transpose(attention_score, (0, 2, 1)) # [B, 1, T]
58
59 # 4. 得到有真实意义的score
60 if self.weight_normalization:
61     # padding的mask后补一个很小的负数, 这样后面计算 softmax 时,  $e^{\{x\}}$  结果
62     paddings = tf.ones_like(outputs) * (-2 ** 32 + 1)
63 else:
64     paddings = tf.zeros_like(outputs)
65 outputs = tf.where(key_masks, outputs, paddings) # [B, 1, T]
66
67 # 5. Activation, 得到归一化后的权重
68 if self.weight_normalization:
69     outputs = tf.nn.softmax(outputs) # [B, 1, T]
70
71 # 6. 得到了正确的权重 outputs 以及用户历史行为序列 keys, 再进行矩阵相乘得到
72 # Weighted sum,
73 if self.mode == 'sum':
74     # outputs 的大小为 [B, 1, T], 表示每条历史行为的权重,
75     # keys 为历史行为序列, 大小为 [B, T, H];
76     # 两者用矩阵乘法做, 得到的结果 outputs 就是 [B, 1, H]
77     outputs = tf.matmul(outputs, keys) # [B, 1, H]
78 else:
79     # 从 [B, 1, H] 变化成 Batch * Time
80     outputs = tf.reshape(outputs, [-1, tf.shape(keys)[1]])
81     # 先把scores在最后增加一维, 然后进行哈达码积,  $[B, T, H] \times [B, T, 1] =$ 
82     outputs = keys * tf.expand_dims(outputs, -1)
83     outputs = tf.reshape(outputs, tf.shape(keys)) # Batch * Time * Hic
84
85 return outputs
86
87 def get_config(self, ):
88
89     config = {'att_hidden_units': self.att_hidden_units, 'att_activation':
90             'weight_normalization': self.weight_normalization, 'mode': s
91     base_config = super(AttentionLayer, self).get_config()
92     return config.update(base_config)

```


3.Attention机制在社区推荐的应用



在社区feed流推荐中，用户过去交互过的内容肯定对他将要点击的内容会产生动态影响，也就是，用户的最近的历史行为序列富含了非常重要的信息值得我们挖掘，参照阿里的DIN模型，我们将深度DNN模型加入attention机制，并输入用户最近的访问历史行为序列特征来辅助挖掘用户的兴趣信息。我们与阿里DIN那篇论文做法稍有不同，我们是取物料的Topic与权重最大的关键词来表征物料（一般我们的每一个物料都会标记多个关键词，我们这里取权重最大的那个关键词），好了，下面就来看看我们的整体模型搭建的过程吧。

为了更好的演示完整模型的搭建，这里拟造部分我们真实输入模型数据，如下：

	act	client_id	client_type	post_type	topic_id	follow_topic_id	all_topic_fav_7	hist_topic_id	keyword_id	hist_keyword_id
0	0	144629	0	0	1	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	225,158	1	1,4
1	1	144629	1	0	1	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	129,123	2	2,5
2	0	144629	0	1	1	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	129,124,123	3	3,9,5
3	1	144629	0	0	177	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	225,158,123	4	1,2,4
4	1	144629	1	0	1	225,158,139,138,140,130,129,124,123	1:0.4074,140:0.1217,502:0.4286	225,129,124,123	4	1,4,6,7
5	0	144629	0	1	1	225,158,139,138,140,130,129,124,123	1:0.4074,127:0.1217,502:0.4286	139,140,130	5	5,7,3
6	1	144629	0	0	278	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	124,123	6	6,8
7	0	144629	1	0	278	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	225,158,139,138,140	7	7,1,2,3,4
8	1	144629	0	1	606	225,158,139,138,140,130,129,124,123	1:0.4074,13:0.1217,502:0.4286	225,158	8	8,9
9	0	144629	0	0	127	225,158,139,138,140,130,129,124,123	1:0.4074,177:0.1217,502:0.4286	225,158,139	9	1,9,3

字段介绍：

- act: 为label数据 1:正样本，0：负样本
- client_id: 用户id
- post_type: 物料item形式 图文，视频
- client_type: 用户客户端类型
- follow_topic_id: 用户关注话题分类id
- all_topic_fav_7: 用户画像特征，用户最近7天对话题偏爱度刻画，kv键值对形式
- topic_id: 物料所属的话题
- hist_topic_id: 用户历史访问item话题id序列
- keyword_id: 物料item对应的权重最大的关键词id
- hist_key_word_id: 用户历史访问item关键词id序列

```
1 import numpy as np
2 import datetime
3 import itertools
4 import tensorflow as tf
5 from collections import namedtuple, OrderedDict
```

```

6 from tensorflow.keras.layers import *
7 import tensorflow.keras.backend as K
8 from tensorflow.keras import layers
9 from tensorflow.keras.models import Model
10 from tensorflow.keras.callbacks import TensorBoard
11
12 #####
13 ##### 数据预处理#####
14 #####
15
16 # 定义输入数据参数类型
17 SparseFeat = namedtuple('SparseFeat', ['name', 'voc_size', 'hash_size', 'sha
18 DenseFeat = namedtuple('DenseFeat', ['name', 'pre_embed', 'reduce_type', 'dim'
19 VarLenSparseFeat = namedtuple('VarLenSparseFeat', ['name', 'voc_size', 'hash_s
20
21 # 筛选实体标签categorical
22 DICT_CATEGORICAL = {"topic_id": [str(i) for i in range(0, 700)],
23                     keyword_id": [str(i) for i in range(0, 100)],
24                     }
25
26
27 feature_columns = [SparseFeat(name="topic_id", voc_size=700, hash_size= None,
28                               SparseFeat(name="keyword_id", voc_size=10, hash_size= None,
29                               SparseFeat(name='client_type', voc_size=2, hash_size= None,
30                               SparseFeat(name='post_type', voc_size=2, hash_size= None,
31                               VarLenSparseFeat(name="follow_topic_id", voc_size=700, ha
32                               VarLenSparseFeat(name="all_topic_fav_7", voc_size=700, ha
33                               VarLenSparseFeat(name="hist_topic_id", voc_size=700, hash
34                               VarLenSparseFeat(name="hist_keyword_id", voc_size=10, hash
35 #                               DenseFeat(name='client_embed', pre_embed='read_post_id',
36                               ]
37
38 # 用户行为序列特征
39 history_feature_names = ['topic_id', 'keyword_id']
40
41 DEFAULT_VALUES = [[0], [''], [0.0], [0.0], [''],
42                  ['', [''], [''], [''], ['']]
43 COL_NAME = ['act', 'client_id', 'client_type', 'post_type', "topic_id", 'follow_
44
45 def _parse_function(example_proto):

```



```

46
47 item_feats = tf.io.decode_csv(example_proto, record_defaults=DEFAULT_VALU
48 parsed = dict(zip(COL_NAME, item_feats))
49
50 feature_dict = {}
51 for feat_col in feature_columns:
52     if isinstance(feat_col, VarLenSparseFeat):
53         if feat_col.weight_name is not None:
54             kvpairs = tf.strings.split([parsed[feat_col.name]], ',').valu
55             kvpairs = tf.strings.split(kvpairs, ':')
56             kvpairs = kvpairs.to_tensor()
57             feat_ids, feat_vals = tf.split(kvpairs, num_or_size_splits=2,
58             feat_ids = tf.reshape(feat_ids, shape=[-1])
59             feat_vals = tf.reshape(feat_vals, shape=[-1])
60             if feat_col.dtype != 'string':
61                 feat_ids= tf.strings.to_number(feat_ids, out_type=tf.int32)
62                 feat_vals= tf.strings.to_number(feat_vals, out_type=tf.float32)
63             feature_dict[feat_col.name] = feat_ids
64             feature_dict[feat_col.weight_name] = feat_vals
65         else:
66             feat_ids = tf.strings.split([parsed[feat_col.name]], ',').valu
67             feat_ids = tf.reshape(feat_ids, shape=[-1])
68             if feat_col.dtype != 'string':
69                 feat_ids= tf.strings.to_number(feat_ids, out_type=tf.int32)
70             feature_dict[feat_col.name] = feat_ids
71
72     elif isinstance(feat_col, SparseFeat):
73         feature_dict[feat_col.name] = parsed[feat_col.name]
74
75     elif isinstance(feat_col, DenseFeat):
76         if not feat_col.is_embed:
77             feature_dict[feat_col.name] = parsed[feat_col.name]
78         elif feat_col.reduce_type is not None:
79             keys = tf.strings.split(parsed[feat_col.is_embed], ',')
80             emb = tf.nn.embedding_lookup(params=ITEM_EMBEDDING, ids=ITEM_
81             emb = tf.reduce_mean(emb,axis=0) if feat_col.reduce_type ==
82             feature_dict[feat_col.name] = emb
83         else:
84             emb = tf.nn.embedding_lookup(params=ITEM_EMBEDDING, ids=ITEM_
85             feature_dict[feat_col.name] = emb

```

```
86         else:
87             raise "unknown feature_columns...."
88
89
90         label = parsed['act']
91
92
93         return feature_dict, label
94
95
96     pad_shapes = {}
97     pad_values = {}
98
99     for feat_col in feature_columns:
100         if isinstance(feat_col, VarLenSparseFeat):
101             max_tokens = feat_col.maxlen
102             pad_shapes[feat_col.name] = tf.TensorShape([max_tokens])
103             pad_values[feat_col.name] = '0' if feat_col.dtype == 'string' else 0
104             if feat_col.weight_name is not None:
105                 pad_shapes[feat_col.weight_name] = tf.TensorShape([max_tokens])
106                 pad_values[feat_col.weight_name] = tf.constant(-1, dtype=tf.float32)
107
108     # no need to pad labels
109     elif isinstance(feat_col, SparseFeat):
110         if feat_col.dtype == 'string':
111             pad_shapes[feat_col.name] = tf.TensorShape([])
112             pad_values[feat_col.name] = '0'
113         else:
114             pad_shapes[feat_col.name] = tf.TensorShape([])
115             pad_values[feat_col.name] = 0.0
116     elif isinstance(feat_col, DenseFeat):
117         if not feat_col.is_embed:
118             pad_shapes[feat_col.name] = tf.TensorShape([])
119             pad_values[feat_col.name] = 0.0
120         else:
121             pad_shapes[feat_col.name] = tf.TensorShape([feat_col.dim])
122             pad_values[feat_col.name] = 0.0
123
124
125     pad_shapes = (pad_shapes, (tf.TensorShape([])))
```

```

126 pad_values = (pad_values, (tf.constant(0, dtype=tf.int32)))
127
128
129
130 filenames= tf.data.Dataset.list_files([
131 './user_item_act_test.csv',
132 ])
133 dataset = filenames.flat_map(
134     lambda filepath: tf.data.TextLineDataset(filepath).skip(1))
135
136 batch_size = 2
137 dataset = dataset.map(_parse_function, num_parallel_calls=60)
138 dataset = dataset.repeat()
139 dataset = dataset.shuffle(buffer_size = batch_size) # 在缓冲区中随机打乱数据
140 dataset = dataset.padded_batch(batch_size = batch_size,
141                                padded_shapes = pad_shapes,
142                                padding_values = pad_values) # 每1024条数据为一
143 dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
144
145 # 验证集
146 filenames_val= tf.data.Dataset.list_files(['./user_item_act_test_val.csv'])
147 dataset_val = filenames_val.flat_map(
148     lambda filepath: tf.data.TextLineDataset(filepath).skip(1))
149
150 val_batch_size = 2
151 dataset_val = dataset_val.map(_parse_function, num_parallel_calls=60)
152 dataset_val = dataset_val.padded_batch(batch_size = val_batch_size,
153                                         padded_shapes = pad_shapes,
154                                         padding_values = pad_values) # 每1024条数据为一
155 dataset_val = dataset_val.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
156
157 #####
158 #####自定义Layer#####
159 #####
160
161
162 # 多值查找表稀疏SparseTensor >> EncodeMultiEmbedding
163 class VocabLayer(Layer):
164     def __init__(self, keys, mask_value=None, **kwargs):
165         super(VocabLayer, self).__init__(**kwargs)

```

```

166     self.mask_value = mask_value
167     vals = tf.range(2, len(keys) + 2)
168     vals = tf.constant(vals, dtype=tf.int32)
169     keys = tf.constant(keys)
170     self.table = tf.lookup.StaticHashTable(
171         tf.lookup.KeyValueTensorInitializer(keys, vals), 1)
172
173     def call(self, inputs):
174         idx = self.table.lookup(inputs)
175         if self.mask_value is not None:
176             masks = tf.not_equal(inputs, self.mask_value)
177             paddings = tf.ones_like(idx) * (0) # mask成 0
178             idx = tf.where(masks, idx, paddings)
179         return idx
180
181     def get_config(self):
182         config = super(VocabLayer, self).get_config()
183         config.update({'mask_value': self.mask_value, })
184         return config
185
186
187 class EmbeddingLookupSparse(Layer):
188     def __init__(self, embedding, has_weight=False, combiner='sum', **kwargs):
189
190         super(EmbeddingLookupSparse, self).__init__(**kwargs)
191         self.has_weight = has_weight
192         self.combiner = combiner
193         self.embedding = embedding
194
195
196     def build(self, input_shape):
197         super(EmbeddingLookupSparse, self).build(input_shape)
198
199     def call(self, inputs):
200         if self.has_weight:
201             idx, val = inputs
202             combiner_embed = tf.nn.embedding_lookup_sparse(self.embedding, sp_
203         else:
204             idx = inputs
205             combiner_embed = tf.nn.embedding_lookup_sparse(self.embedding, sp_

```

```
206         return tf.expand_dims(combiner_embed, 1)
207
208     def get_config(self):
209         config = super(EmbeddingLookupSparse, self).get_config()
210         config.update({'has_weight': self.has_weight, 'combiner':self.combiner})
211         return config
212
213
214 class EmbeddingLookup(Layer):
215     def __init__(self, embedding, **kwargs):
216
217         super(EmbeddingLookup, self).__init__(**kwargs)
218         self.embedding = embedding
219
220
221     def build(self, input_shape):
222         super(EmbeddingLookup, self).build(input_shape)
223
224     def call(self, inputs):
225         idx = inputs
226         embed = tf.nn.embedding_lookup(params=self.embedding, ids=idx)
227         return embed
228
229     def get_config(self):
230         config = super(EmbeddingLookup, self).get_config()
231         return config
232
233
234
235 # 稠密转稀疏
236 class DenseToSparseTensor(Layer):
237     def __init__(self, mask_value= -1, **kwargs):
238         super(DenseToSparseTensor, self).__init__()
239         self.mask_value = mask_value
240
241
242     def call(self, dense_tensor):
243         idx = tf.where(tf.not_equal(dense_tensor, tf.constant(self.mask_value)))
244         sparse_tensor = tf.SparseTensor(idx, tf.gather_nd(dense_tensor, idx))
245         return sparse_tensor
```

```

246
247     def get_config(self):
248         config = super(DenseToSparseTensor, self).get_config()
249         config.update({'mask_value': self.mask_value})
250         return config
251
252
253 # 自定义dense层 含BN, dropout
254 class CustomDense(Layer):
255     def __init__(self, units=32, activation='tanh', dropout_rate =0, use_bn=True, seed=None, tag_name=''):
256         self.units = units
257         self.activation = activation
258         self.dropout_rate = dropout_rate
259         self.use_bn = use_bn
260         self.seed = seed
261         self.tag_name = tag_name
262
263         super(CustomDense, self).__init__(**kwargs)
264
265     #build方法一般定义Layer需要被训练的参数。
266     def build(self, input_shape):
267         self.weight = self.add_weight(shape=(input_shape[-1], self.units),
268                                       initializer='random_normal',
269                                       trainable=True,
270                                       name='kernel_' + self.tag_name)
271         self.bias = self.add_weight(shape=(self.units,),
272                                    initializer='random_normal',
273                                    trainable=True,
274                                    name='bias_' + self.tag_name)
275
276         if self.use_bn:
277             self.bn_layers = tf.keras.layers.BatchNormalization()
278
279         self.dropout_layers = tf.keras.layers.Dropout(self.dropout_rate)
280         self.activation_layers = tf.keras.layers.Activation(self.activation,
281                                                             name='activation_' + self.tag_name)
282
283         super(CustomDense, self).build(input_shape) # 相当于设置self.built = True
284
285     #call方法一般定义正向传播运算逻辑, __call__方法调用了它。
286     def call(self, inputs, training=None, **kwargs):

```



```

286         fc = tf.matmul(inputs, self.weight) + self.bias
287         if self.use_bn:
288             fc = self.bn_layers(fc)
289         out_fc = self.activation_layers(fc)
290
291         return out_fc
292
293     # 如果能让自定义的Layer通过Functional API 组合成模型时可以序列化, 需要自定义get_
294     def get_config(self):
295         config = super(CustomDense, self).get_config()
296         config.update({'units': self.units, 'activation': self.activation, 'u
297                        'dropout_rate': self.dropout_rate, 'seed': self.seed,
298         return config
299
300
301 class HashLayer(Layer):
302     """
303     hash the input to [0,num_buckets)
304     if mask_zero = True, 0 or 0.0 will be set to 0, other value will be set in
305     """
306
307     def __init__(self, num_buckets, mask_zero=False, **kwargs):
308         self.num_buckets = num_buckets
309         self.mask_zero = mask_zero
310         super(HashLayer, self).__init__(**kwargs)
311
312     def build(self, input_shape):
313         # Be sure to call this somewhere!
314         super(HashLayer, self).build(input_shape)
315
316     def call(self, x, mask=None, **kwargs):
317         zero = tf.as_string(tf.zeros([1], dtype='int32'))
318         num_buckets = self.num_buckets if not self.mask_zero else self.num_bu
319         hash_x = tf.strings.to_hash_bucket_fast(x, num_buckets, name=None)
320         if self.mask_zero:
321             mask = tf.cast(tf.not_equal(x, zero), dtype='int64')
322             hash_x = (hash_x + 1) * mask
323
324         return hash_x
325     def get_config(self, ):

```

```

326         config = super(HashLayer, self).get_config()
327         config.update({'num_buckets': self.num_buckets, 'mask_zero': self.mask_zero})
328         return config
329
330 # Attention池化层
331 class AttentionPoolingLayer(Layer):
332     """
333     Input shape
334         - A list of three tensor: [query,keys,his_seq]
335         - query is a 3D tensor with shape: ``(batch_size, 1, embedding_size)``
336         - keys is a 3D tensor with shape: ``(batch_size, T, embedding_size)``
337         - his_seq is a 2D tensor with shape: ``(batch_size, T)``
338     Output shape
339         - 3D tensor with shape: ``(batch_size, 1, embedding_size)``.
340     Arguments
341         - **att_hidden_units**:list of positive integer, the attention net layers.
342         - **att_activation**: Activation function to use in attention net.
343         - **weight_normalization**: bool.Whether normalize the attention score.
344         - **hist_mask_value**: the mask value of his_seq.
345     References
346         - [Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction. 2018.]
347     """
348
349     def __init__(self, att_hidden_units=(80, 40), att_activation='sigmoid', weight_normalization=True, mode="sum", hist_mask_value=0, **kwargs):
350         self.att_hidden_units = att_hidden_units
351         self.att_activation = att_activation
352         self.weight_normalization = weight_normalization
353         self.mode = mode
354         self.hist_mask_value = hist_mask_value
355         super(AttentionPoolingLayer, self).__init__(**kwargs)
356
357     def build(self, input_shape):
358
359         self.fc = tf.keras.Sequential()
360         for unit in self.att_hidden_units:
361             self.fc.add(layers.Dense(unit, activation=self.att_activation, name="att_hidden"))
362         self.fc.add(layers.Dense(1, activation=None, name="fc_att_out"))

```

```

366
367     super(AttentionPoolingLayer, self).build(
368         input_shape) # Be sure to call this somewhere!
369
370     def call(self, inputs, **kwargs):
371         query, keys, his_seq = inputs
372         # 计算掩码
373         key_masks = tf.not_equal(his_seq, tf.constant(self.hist_mask_value ,
374         key_masks = tf.expand_dims(key_masks, 1)
375
376         # 1. 转换query维度, 变成历史维度T
377         # query是[B, 1, H], 转换到 queries 维度为(B, T, H), 为了让pos_item和用
378         queries = tf.tile(query, [1, tf.shape(keys)[1], 1]) # [B, T, H]
379
380         # 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
381         # 得到 Local Activation Unit 的输入。即 候选queries 对应的 emb, 用户历史
382         # 对应的 embed, 再加上它们之间的交叉特征, 进行 concat 后的结果
383         din_all = tf.concat([queries, keys, queries-keys, queries*keys], axis=
384         # 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
385         attention_score = self.fc(din_all) # [B, T, 1]
386         # attention的输出, [B, 1, T]
387         outputs = tf.transpose(attention_score, (0, 2, 1)) # [B, 1, T]
388
389         # 4. 得到有真实意义的score
390         if self.weight_normalization:
391             # padding的mask后补一个很小的负数, 这样后面计算 softmax 时,  $e^{\{x\}}$  结
392             paddings = tf.ones_like(outputs) * (-2 ** 32 + 1)
393         else:
394             paddings = tf.zeros_like(outputs)
395         outputs = tf.where(key_masks, outputs, paddings) # [B, 1, T]
396
397         # 5. Activation, 得到归一化后的权重
398         if self.weight_normalization:
399             outputs = tf.nn.softmax(outputs) # [B, 1, T]
400
401         # 6. 得到了正确的权重 outputs 以及用户历史行为序列 keys, 再进行矩阵相乘得到
402         # Weighted sum,
403         if self.mode == 'sum':
404             # outputs 的大小为 [B, 1, T], 表示每条历史行为的权重,
405             # keys 为历史行为序列, 大小为 [B, T, H];

```

```

406         # 两者用矩阵乘法做, 得到的结果 outputs 就是 [B, 1, H]
407         outputs = tf.matmul(outputs, keys) # [B, 1, H]
408     else:
409         # 从 [B, 1, H] 变化成 Batch * Time
410         outputs = tf.reshape(outputs, [-1, tf.shape(keys)[1]])
411         # 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] =
412         outputs = keys * tf.expand_dims(outputs, -1)
413         outputs = tf.reshape(outputs, tf.shape(keys)) # Batch * Time * H:
414
415     return outputs
416
417     def get_config(self, ):
418
419         config = {'att_hidden_units': self.att_hidden_units, 'att_activation':
420                 'weight_normalization': self.weight_normalization, 'mode':
421         base_config = super(AttentionPoolingLayer, self).get_config()
422         return config.update(base_config)
423
424
425 class Add(tf.keras.layers.Layer):
426     def __init__(self, **kwargs):
427         super(Add, self).__init__(**kwargs)
428
429     def build(self, input_shape):
430         # Be sure to call this somewhere!
431         super(Add, self).build(input_shape)
432
433     def call(self, inputs, **kwargs):
434         if not isinstance(inputs, list):
435             return inputs
436         if len(inputs) == 1 :
437             return inputs[0]
438         if len(inputs) == 0:
439             return tf.constant([[0.0]])
440         return tf.keras.layers.add(inputs)
441
442
443 #####
444 ##### 定义输入帮助函数#####
445 #####

```

```

446 # 定义model输入特征
447 def build_input_features(features_columns, prefix=''):
448     input_features = OrderedDict()
449
450     for feat_col in features_columns:
451         if isinstance(feat_col, DenseFeat):
452             input_features[feat_col.name] = Input([feat_col.dim], name=feat_col.name)
453         elif isinstance(feat_col, SparseFeat):
454             input_features[feat_col.name] = Input([1], name=feat_col.name, dtype=tf.float32)
455         elif isinstance(feat_col, VarLenSparseFeat):
456             input_features[feat_col.name] = Input([None], name=feat_col.name, dtype=tf.float32)
457             if feat_col.weight_name is not None:
458                 input_features[feat_col.weight_name] = Input([None], name=feat_col.weight_name, dtype=tf.float32)
459         else:
460             raise TypeError("Invalid feature column in build_input_features: {}".format(feat_col))
461
462     return input_features
463
464 # 构造 自定义embedding层 matrix
465 def build_embedding_matrix(features_columns):
466     embedding_matrix = {}
467     for feat_col in features_columns:
468         if isinstance(feat_col, SparseFeat) or isinstance(feat_col, VarLenSparseFeat):
469             vocab_name = feat_col.share_embed if feat_col.share_embed else feat_col.get_vocab_name()
470             vocab_size = feat_col.voc_size + 2
471             embed_dim = feat_col.embed_dim
472             if vocab_name not in embedding_matrix:
473                 embedding_matrix[vocab_name] = tf.Variable(initial_value=tf.zeros([vocab_size, embed_dim]), dtype=tf.float32, name=vocab_name + '_embedding_matrix')
474
475     return embedding_matrix
476
477 # 构造 自定义embedding层
478 def build_embedding_dict(features_columns, embedding_matrix):
479     embedding_dict = {}
480     for feat_col in features_columns:
481         if isinstance(feat_col, SparseFeat):
482             vocab_name = feat_col.share_embed if feat_col.share_embed else feat_col.get_vocab_name()
483             embedding_dict[feat_col.name] = EmbeddingLookup(embedding=embedding_matrix[vocab_name], num_embeddings=vocab_size, embedding_dim=embed_dim, name=feat_col.name)
484         elif isinstance(feat_col, VarLenSparseFeat):
485             vocab_name = feat_col.share_embed if feat_col.share_embed else feat_col.get_vocab_name()

```

```

486         if feat_col.combiner is not None:
487             if feat_col.weight_name is not None:
488                 embedding_dict[feat_col.name] = EmbeddingLookupSparse(embedding=emb
489             else:
490                 embedding_dict[feat_col.name] = EmbeddingLookupSparse(embedding=emb
491         else:
492             embedding_dict[feat_col.name] = EmbeddingLookup(embedding=emb
493
494     return embedding_dict
495
496
497 # dense 与 embedding 特征输入
498 def input_from_feature_columns(features, features_columns, embedding_dict):
499     sparse_embedding_list = []
500     dense_value_list = []
501
502     for feat_col in features_columns:
503         if isinstance(feat_col, SparseFeat):
504             _input = features[feat_col.name]
505             if feat_col.dtype == 'string':
506                 if feat_col.hash_size is None:
507                     vocab_name = feat_col.share_embed if feat_col.share_embed
508                     keys = DICT_CATEGORICAL[vocab_name]
509                     _input = VocabLayer(keys)(_input)
510                 else:
511                     _input = HashLayer(num_buckets=feat_col.hash_size, mask_val
512
513             embed = embedding_dict[feat_col.name](_input)
514             sparse_embedding_list.append(embed)
515         elif isinstance(feat_col, VarLenSparseFeat):
516             _input = features[feat_col.name]
517             if feat_col.dtype == 'string':
518                 if feat_col.hash_size is None:
519                     vocab_name = feat_col.share_embed if feat_col.share_embed
520                     keys = DICT_CATEGORICAL[vocab_name]
521                     _input = VocabLayer(keys, mask_value='0')(_input)
522                 else:
523                     _input = HashLayer(num_buckets=feat_col.hash_size, mask_val
524             if feat_col.combiner is not None:
525                 input_sparse = DenseToSparseTensor(mask_value=0)(_input)

```



```
526         if feat_col.weight_name is not None:
527             weight_sparse = DenseToSparseTensor()(features[feat_col.name])
528             embed = embedding_dict[feat_col.name]([input_sparse, weight_sparse])
529         else:
530             embed = embedding_dict[feat_col.name](input_sparse)
531     else:
532         embed = embedding_dict[feat_col.name](_input)
533
534     sparse_embedding_list.append(embed)
535
536     elif isinstance(feat_col, DenseFeat):
537         dense_value_list.append(features[feat_col.name])
538
539     else:
540         raise TypeError("Invalid feature column in input_from_feature_columns")
541
542     return sparse_embedding_list, dense_value_list
543
544
545 def concat_func(inputs, axis=-1):
546     if len(inputs) == 1:
547         return inputs[0]
548     else:
549         return Concatenate(axis=axis)(inputs)
550
551 def combined_dnn_input(sparse_embedding_list, dense_value_list):
552     if len(sparse_embedding_list) > 0 and len(dense_value_list) > 0:
553         sparse_dnn_input = Flatten()(concat_func(sparse_embedding_list))
554         dense_dnn_input = Flatten()(concat_func(dense_value_list))
555         return concat_func([sparse_dnn_input, dense_dnn_input])
556     elif len(sparse_embedding_list) > 0:
557         return Flatten()(concat_func(sparse_embedding_list))
558     elif len(dense_value_list) > 0:
559         return Flatten()(concat_func(dense_value_list))
560     else:
561         raise "dnn_feature_columns can not be empty list"
562
563
564 def get_linear_logit(sparse_embedding_list, dense_value_list):
565
```

```

566     if len(sparse_embedding_list) > 0 and len(dense_value_list) > 0:
567         sparse_linear_layer = Add()(sparse_embedding_list)
568         sparse_linear_layer = Flatten()(sparse_linear_layer)
569         dense_linear = concat_func(dense_value_list)
570         dense_linear_layer = Dense(1)(dense_linear)
571         linear_logits = Add()([dense_linear_layer, sparse_linear_layer])
572         return linear_logits
573     elif len(sparse_embedding_list) > 0:
574         sparse_linear_layer = Add()(sparse_embedding_list)
575         sparse_linear_layer = Flatten()(sparse_linear_layer)
576         return sparse_linear_layer
577     elif len(dense_value_list) > 0:
578         dense_linear = concat_func(dense_value_list)
579         dense_linear_layer = Dense(1)(dense_linear)
580         return dense_linear_layer
581     else:
582         raise "linear_feature_columns can not be empty list"
583
584     #####
585         ##### 定义模型 #####
586     #####
587
588     def DIN(feature_columns, history_feature_names, hist_mask_value, dnn_use_bn=False,
589             dnn_hidden_units=(200, 80), dnn_activation='relu', att_hidden_size=(8, 8),
590             att_weight_normalization=True, dnn_dropout=0, seed=1024):
591
592         """Instantiates the Deep Interest Network architecture.
593         Args:
594             dnn_feature_columns: An iterable containing all the features used by
595             history_feature_names: list, to indicate sequence sparse field
596             dnn_use_bn: bool. Whether use BatchNormalization before activation or not
597             dnn_hidden_units: list, list of positive integer or empty list, the layers number of
598             dnn_activation: Activation function to use in deep net
599             att_hidden_size: list, list of positive integer, the layer number and units of
600             att_activation: Activation function to use in attention net
601             att_weight_normalization: bool. Whether normalize the attention score
602             dnn_dropout: float in [0,1), the probability we will drop out a given
603             seed: integer, to use as random seed.
604
605         return: A Keras model instance.
606         """

```

```

606 features = build_input_features(feature_columns)
607
608 sparse_feature_columns = list(
609     filter(lambda x: isinstance(x, SparseFeat), feature_columns)) if feat
610 dense_feature_columns = list(
611     filter(lambda x: isinstance(x, DenseFeat), feature_columns)) if feat
612 varlen_sparse_feature_columns = list(
613     filter(lambda x: isinstance(x, VarLenSparseFeat), feature_columns)) :
614
615 query_feature_columns = []
616 for fc in sparse_feature_columns:
617     feature_name = fc.name
618     if feature_name in history_feature_names:
619         query_feature_columns.append(fc)
620 key_feature_columns = []
621 sparse_varlen_feature_columns = []
622 history_fc_names = list(map(lambda x: "hist_" + x, history_feature_names))
623 for fc in varlen_sparse_feature_columns:
624     feature_name = fc.name
625     if feature_name in history_fc_names:
626         key_feature_columns.append(fc)
627     else:
628         sparse_varlen_feature_columns.append(fc)
629
630 inputs_list = list(features.values())
631
632 # 构建 embedding_dict
633 embedding_matrix = build_embedding_matrix(feature_columns)
634 embedding_dict = build_embedding_dict(feature_columns, embedding_matrix)
635
636 query_emb_list, _ = input_from_feature_columns(features, query_feature_co
637 keys_emb_list, _ = input_from_feature_columns(features, key_feature_colu
638 merge_dnn_columns = sparse_feature_columns + sparse_varlen_feature_colum
639 dnn_sparse_embedding_list, dnn_dense_value_list = input_from_feature_col
640
641 keys_emb = concat_func(keys_emb_list)
642 query_emb = concat_func(query_emb_list)
643 keys_seq = features[key_feature_columns[0].name]
644
645 hist_attn_emb = AttentionPoolingLayer(att_hidden_units=att_hidden_size,

```

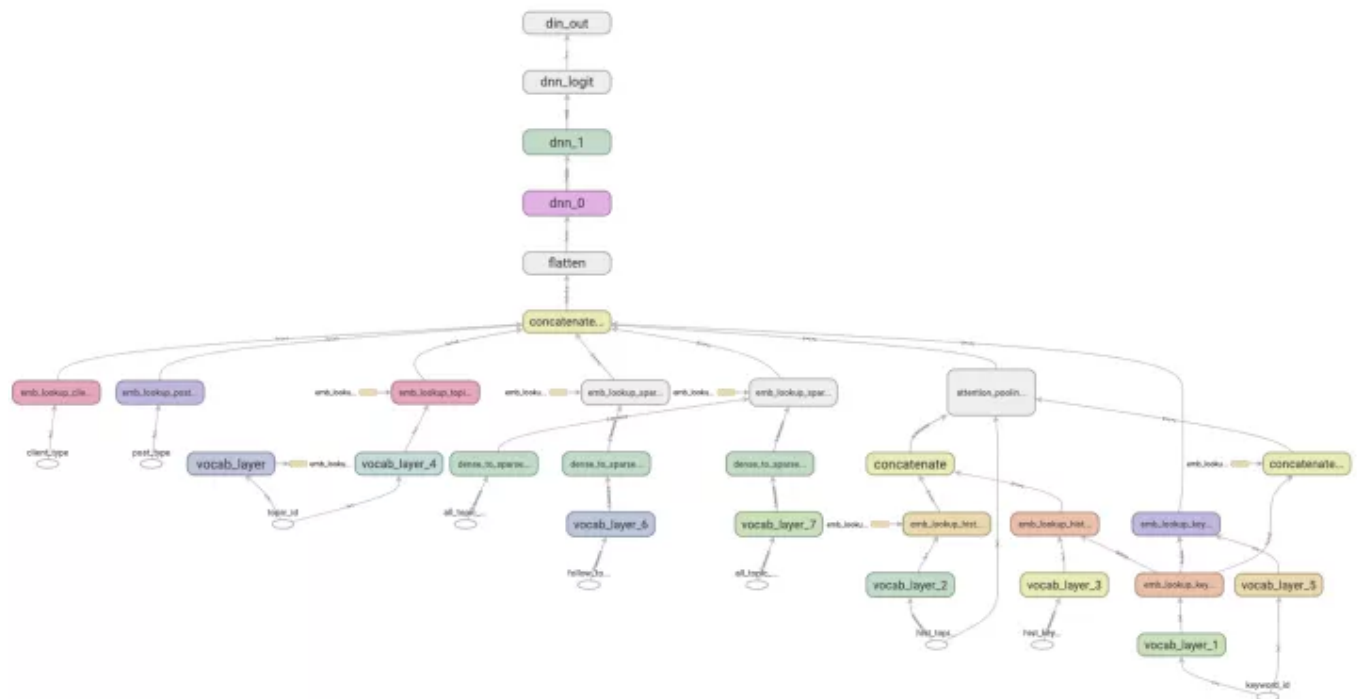
```

646     dnn_input = combined_dnn_input(dnn_sparse_embedding_list+[hist_attn_emb]).
647
648     # DNN
649     for i in range(len(dnn_hidden_units)):
650         if i == len(dnn_hidden_units) - 1:
651             dnn_out = CustomDense(units=dnn_hidden_units[i],dropout_rate=dnn_dropout_rate,
652                                     use_bn=dnn_use_bn, activation=dnn_activation)
653             break
654             dnn_input = CustomDense(units=dnn_hidden_units[i],dropout_rate=dnn_dropout_rate,
655                                     use_bn=dnn_use_bn, activation=dnn_activation)
656     dnn_logit = Dense(1, use_bias=False, activation=None, kernel_initializer=glorot_uniform)
657     output = tf.keras.layers.Activation("sigmoid", name="din_out")(dnn_logit)
658     model = Model(inputs=inputs_list, outputs=output)
659
660     return model
661
662
663
664     model = DIN(feature_columns, history_feature_names, hist_mask_value='0', dnn_hidden_units=(200, 80), dnn_activation='relu', att_hidden_size=(8, 8), att_weight_normalization=True, dnn_dropout=0, seed=1024)
665
666
667
668     model.compile(optimizer="adam", loss= "binary_crossentropy", metrics=tf.keras.metrics.BinaryAccuracy())
669
670     log_dir = '/mywork/tensorboardshare/logs/' + datetime.datetime.now().strftime("%Y-%m-%d-%H-%M-%S")
671     tbCallBack = TensorBoard(log_dir=log_dir, # log 目录
672                               histogram_freq=0, # 按照何等频率（epoch）来计算直方图，0为不计算
673                               write_graph=True, # 是否存储网络结构图
674                               write_images=True,# 是否可视化参数
675                               update_freq='epoch',
676                               embeddings_freq=0,
677                               embeddings_layer_names=None,
678                               embeddings_metadata=None,
679                               profile_batch = 20)
680
681     total_train_sample = 10000
682     total_test_sample = 10
683     train_steps_per_epoch=np.floor(total_train_sample/batch_size).astype(np.int32)
684     test_steps_per_epoch = np.ceil(total_test_sample/val_batch_size).astype(np.int32)
685     history_loss = model.fit(dataset, epochs=3,

```

```
686         steps_per_epoch=train_steps_per_epoch,  
687         validation_data=dataset_val, validation_steps=test_steps_per_epoch,  
688         verbose=1,callbacks=[tbCallback])
```

搭建模型的整体代码就如上了，感兴趣的同学可以copy代码跑跑，动手才是王道，下面我们看看上述搭建的模型结构：



参考文献

[王多鱼：注意力机制在深度推荐算法中的应用之DIN模型](#)

[推荐系统与Attention机制--详解Attention机制_caizd2009的博客-CSDN博客](#)

[张俊林：深度学习中的注意力模型（2017版）](#)

[石塔西：也评Deep Interest Evolution Network](#)

[Attention Is All You Need](#)

[Deep Interest Network for Click-Through Rate Prediction \(DIN\) ——KDD2018 shenweichen/DeepCTR](#)