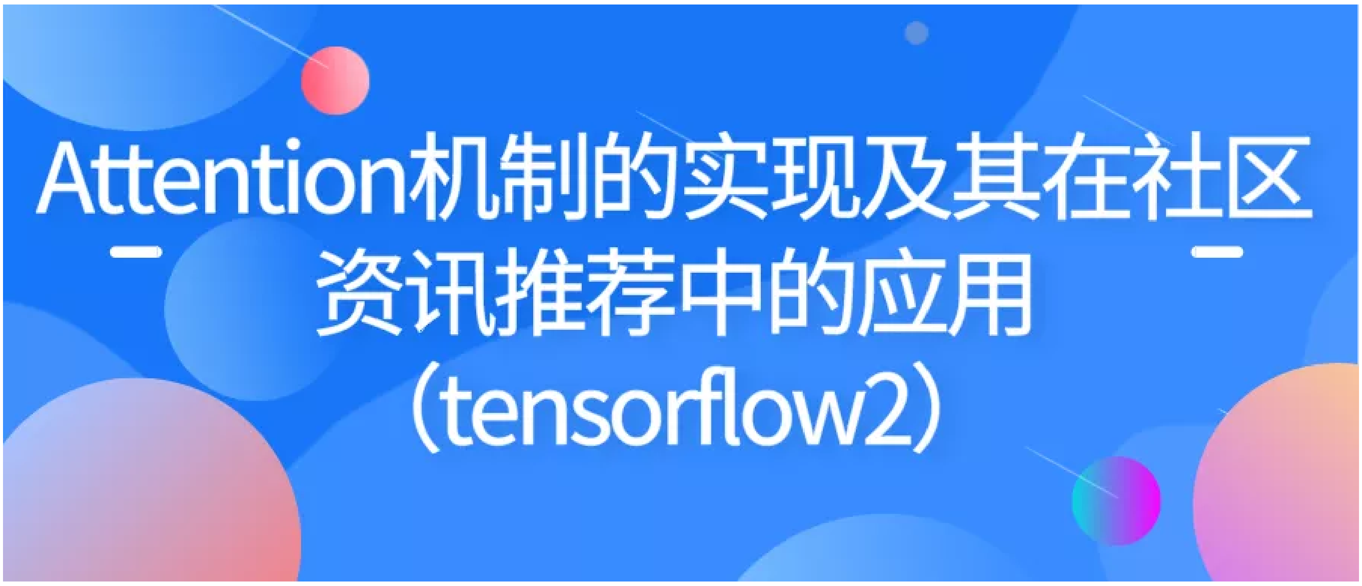
Attention机制的实现及其在社区资讯推荐中的应用（tensorflow2）

原创 xulu1352 数据与智能 1月5日

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



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

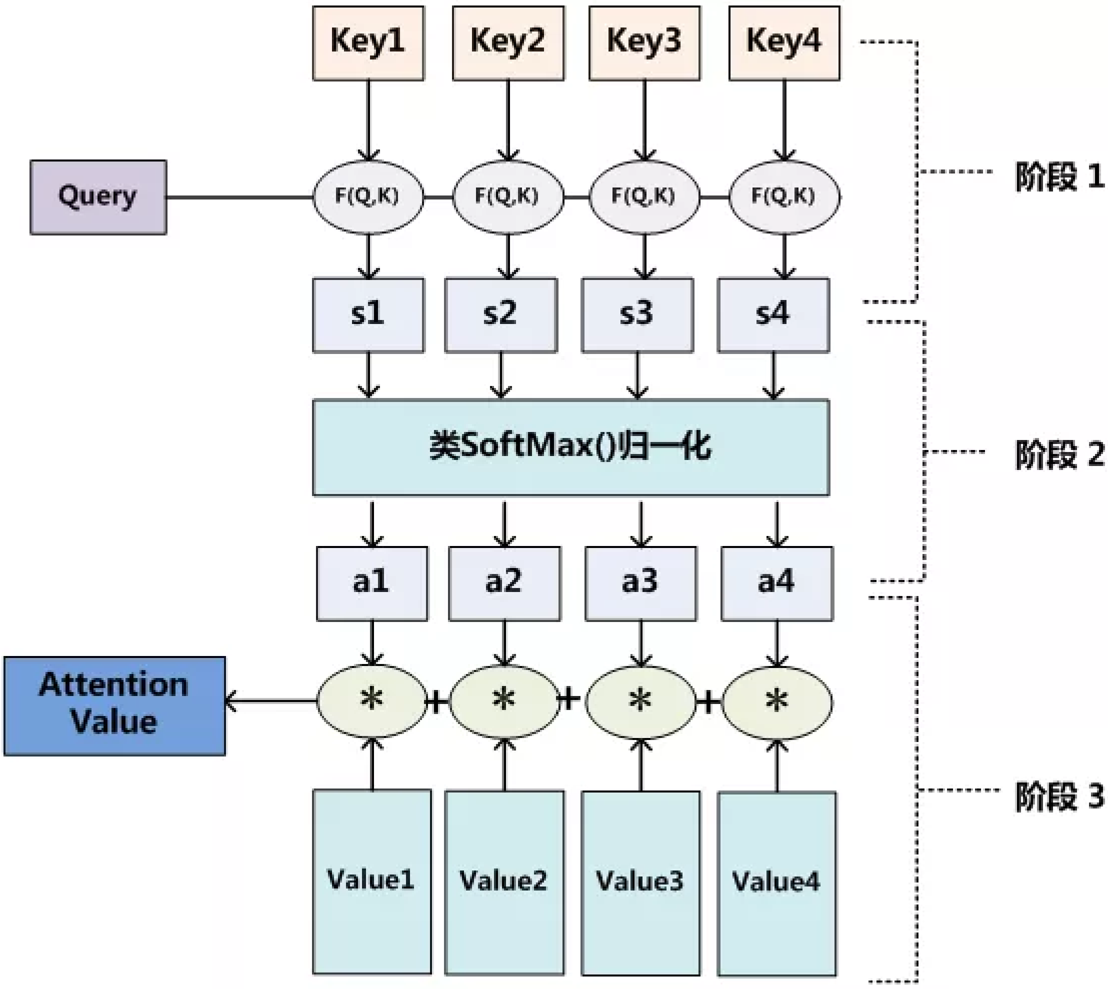
Attention机制缘起于人类视觉注意力机制，比如人们在看东西的时候一般会快速扫描全局，根据需求将观察焦点锁定在特定的位置上，是模仿人类注意力而提出的一种解决问题的办法；抽象点说它是一种权重参数的分配机制，目标是协助模型捕捉重要信息。具体一点就是，给定一组<key,value>，以及一个目标（查询）向量query，Attention机制就是通过计算query与各个key的相似性，得到每个key的权重系数，再通过对value加权求和，得到最终attention数值。所以本质上Attention机制是对给定元素的value值进行加权求和，而query和key用来计算对应value的权重系数。可以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

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

如果对目前大多数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，通常我们会用一个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”。

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

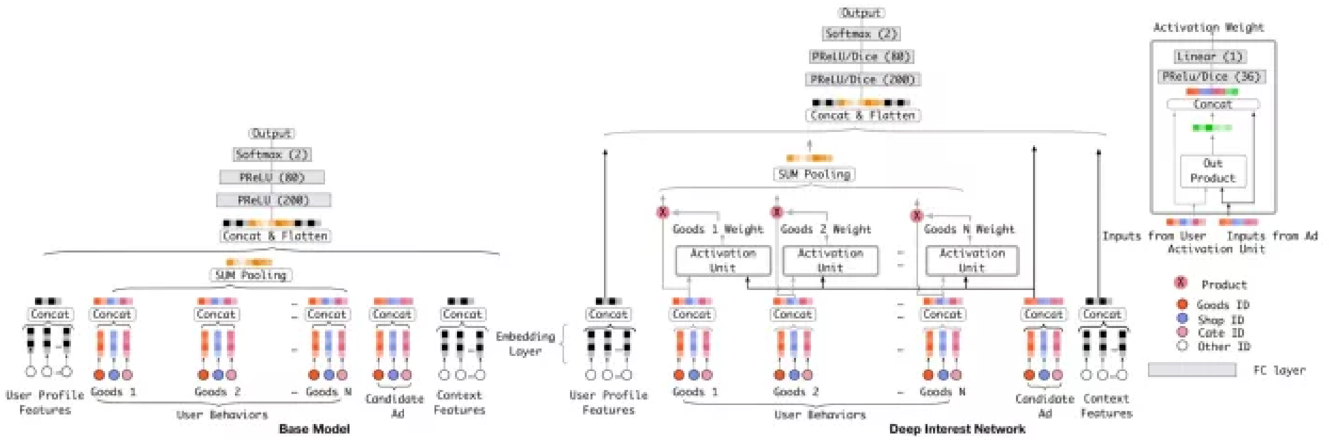


Figure 2: Network Architecture. The left part illustrates the network of base model (Embedding&MLP). Embeddings of cate_id, shop_id and goods_id belong to one goods are concatenated to represent one visited goods in user's behaviors. Right part is our proposed DIN model. It introduces a local activation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

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

```
            - **hist_mask_value**: the mask value of his_seq.
          References
            - [Zhou G, Zhu X, Song C, et al. Deep interest network for click-throu
        """

        def __init__(self, att_hidden_units=(80, 40), att_activation='sigmoid', we
                     mode="sum",hist_mask_value=0, **kwargs):

            self.att_hidden_units = att_hidden_units
            self.att_activation = att_activation
            self.weight_normalization = weight_normalization
            self.mode = mode
            self.hist_mask_value = hist_mask_value
            super(AttentionLayer, self).__init__(**kwargs)


        def build(self, input_shape):

            self.fc = tf.keras.Sequential()
            for unit in self.att_hidden_units:
                self.fc.add(layers.Dense(unit, activation=self.att_activation, nar
            self.fc.add(layers.Dense(1, activation=None, name="fc_att_out"))

            super(AttentionLayer, self).build(
                input_shape)  # Be sure to call this somewhere!

        def call(self, inputs, **kwargs):
            query, keys, his_seq = inputs
            # 计算掩码
            key_masks = tf.not_equal(his_seq, tf.constant(self.hist_mask_value , c
            key_masks = tf.expand_dims(key_masks, 1)

            # 1. 转换query维度，变成历史维度T
            # query是[B, 1,  H]，转换到 queries 维度为(B, T, H)，为了让pos_item和用户
            queries = tf.tile(query, [1, tf.shape(keys)[1], 1]) # [B, T, H]

            # 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
            # 得到 Local Activation Unit 的输入。即 候选queries 对应的 emb，用户历史行
            # 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, keys, queries-keys, queries*keys], axis=
```

```
        # 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
        attention_score = self.fc(din_all) # [B, T, 1]
        # attention的输出, [B, 1, T]
        outputs = tf.transpose(attention_score, (0, 2, 1)) # [B, 1, T]

        # 4. 得到有真实意义的score
        if self.weight_normalization:
            # padding的mask后补一个很小的负数，这样后面计算 softmax 时, e^{x} 结果
            paddings = tf.ones_like(outputs) * (-2 ** 32 + 1)
        else:
            paddings = tf.zeros_like(outputs)
        outputs = tf.where(key_masks, outputs, paddings)  # [B, 1, T]

        # 5. Activation，得到归一化后的权重
        if self.weight_normalization:
            outputs = tf.nn.softmax(outputs)  # [B, 1, T]

        # 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用
        # Weighted sum,
        if self.mode == 'sum':
            # outputs 的大小为 [B, 1, T], 表示每条历史行为的权重,
            # keys 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 outputs 就是 [B, 1, H]
            outputs = tf.matmul(outputs, keys)  # [B, 1, H]
        else:
            # 从 [B, 1, H] 变化成 Batch * Time
            outputs = tf.reshape(outputs, [-1, tf.shape(keys)[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] =
            outputs = keys * tf.expand_dims(outputs, -1)
            outputs = tf.reshape(outputs, tf.shape(keys)) # Batch * Time * Hid

        return outputs

    def get_config(self, ):

        config = {'att_hidden_units': self.att_hidden_units, 'att_activation':
                  'weight_normalization': self.weight_normalization, 'mode': s
        base_config = super(AttentionLayer, self).get_config()
        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

```
import numpy as np
import datetime
import itertools
import tensorflow as tf
from collections import namedtuple, OrderedDict
```

```
from tensorflow.keras.layers import *
import tensorflow.keras.backend as K
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import TensorBoard

##########################################################################
                ##################数据预处理##############
##########################################################################

# 定义输入数据参数类型
SparseFeat = namedtuple('SparseFeat', ['name', 'voc_size', 'hash_size', 'shar
DenseFeat = namedtuple('DenseFeat', ['name', 'pre_embed','reduce_type','dim',
VarLenSparseFeat = namedtuple('VarLenSparseFeat', ['name', 'voc_size','hash_s

# 筛选实体标签categorical
DICT_CATEGORICAL = {"topic_id": [str(i) for i in range(0, 700)],
                    keyword_id": [str(i) for i in range(0, 100)],
           }


feature_columns = [SparseFeat(name="topic_id", voc_size=700, hash_size= None,
                   SparseFeat(name="keyword_id", voc_size=10, hash_size= None
                    SparseFeat(name='client_type', voc_size=2, hash_size= Non
                   SparseFeat(name='post_type', voc_size=2, hash_size= None,s
                    VarLenSparseFeat(name="follow_topic_id", voc_size=700, ha
                    VarLenSparseFeat(name="all_topic_fav_7", voc_size=700, ha
                    VarLenSparseFeat(name="hist_topic_id", voc_size=700, hash
                   VarLenSparseFeat(name="hist_keyword_id", voc_size=10, hash
#                    DenseFeat(name='client_embed',pre_embed='read_post_id',
                   ]

# 用户行为序列特征
history_feature_names = ['topic_id', 'keyword_id']

DEFAULT_VALUES = [[0],[''],[0.0],[0.0], [''],
                  [''], [''],[''], [''],['']]
COL_NAME = ['act','client_id','client_type', 'post_type',"topic_id", 'follow_

def _parse_function(example_proto):
```

```
    item_feats = tf.io.decode_csv(example_proto, record_defaults=DEFAULT_VALU
    parsed = dict(zip(COL_NAME, item_feats))

    feature_dict = {}
    for feat_col in feature_columns:
        if isinstance(feat_col, VarLenSparseFeat):
            if feat_col.weight_name is not None:
                kvpairs = tf.strings.split([parsed[feat_col.name]], ',').valu
                kvpairs = tf.strings.split(kvpairs, ':')
                kvpairs = kvpairs.to_tensor()
                feat_ids, feat_vals = tf.split(kvpairs, num_or_size_splits=2,
                feat_ids = tf.reshape(feat_ids, shape=[-1])
                feat_vals = tf.reshape(feat_vals, shape=[-1])
                if feat_col.dtype != 'string':
                    feat_ids= tf.strings.to_number(feat_ids, out_type=tf.int3
                feat_vals= tf.strings.to_number(feat_vals, out_type=tf.float3
                feature_dict[feat_col.name] = feat_ids
                feature_dict[feat_col.weight_name] = feat_vals
            else:
                feat_ids = tf.strings.split([parsed[feat_col.name]], ',').val
                feat_ids = tf.reshape(feat_ids, shape=[-1])
                if feat_col.dtype != 'string':
                    feat_ids= tf.strings.to_number(feat_ids, out_type=tf.int3
                feature_dict[feat_col.name] = feat_ids

        elif isinstance(feat_col, SparseFeat):
            feature_dict[feat_col.name] = parsed[feat_col.name]

        elif isinstance(feat_col, DenseFeat):
            if not feat_col.is_embed:
                feature_dict[feat_col.name] = parsed[feat_col.name]
            elif feat_col.reduce_type is not None:
                keys = tf.strings.split(parsed[feat_col.is_embed], ',')
                emb = tf.nn.embedding_lookup(params=ITEM_EMBEDDING, ids=ITEM_
                emb = tf.reduce_mean(emb,axis=0) if feat_col.reduce_type ==
                feature_dict[feat_col.name] = emb
            else:
                emb = tf.nn.embedding_lookup(params=ITEM_EMBEDDING, ids=ITEM_
                feature_dict[feat_col.name] = emb
```

```
        else:
            raise "unknown feature_columns...."


    label = parsed['act']


    return feature_dict, label


pad_shapes = {}
pad_values = {}

for feat_col in feature_columns:
    if isinstance(feat_col, VarLenSparseFeat):
        max_tokens = feat_col.maxlen
        pad_shapes[feat_col.name] = tf.TensorShape([max_tokens])
        pad_values[feat_col.name] = '0' if feat_col.dtype == 'string' else 0
        if feat_col.weight_name is not None:
            pad_shapes[feat_col.weight_name] = tf.TensorShape([max_tokens])
            pad_values[feat_col.weight_name] = tf.constant(-1, dtype=tf.floa

# no need to pad labels
    elif isinstance(feat_col, SparseFeat):
        if feat_col.dtype == 'string':
            pad_shapes[feat_col.name] = tf.TensorShape([])
            pad_values[feat_col.name] = '0'
        else:
            pad_shapes[feat_col.name] = tf.TensorShape([])
            pad_values[feat_col.name] = 0.0
    elif isinstance(feat_col, DenseFeat):
        if not feat_col.is_embed:
            pad_shapes[feat_col.name] = tf.TensorShape([])
            pad_values[feat_col.name] = 0.0
        else:
            pad_shapes[feat_col.name] = tf.TensorShape([feat_col.dim])
            pad_values[feat_col.name] = 0.0


pad_shapes = (pad_shapes, (tf.TensorShape([])))
```

```
pad_values = (pad_values, (tf.constant(0, dtype=tf.int32)))



filenames= tf.data.Dataset.list_files([
'./user_item_act_test.csv',
])
dataset = filenames.flat_map(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1))

batch_size = 2
dataset = dataset.map(_parse_function, num_parallel_calls=60)
dataset = dataset.repeat()
dataset = dataset.shuffle(buffer_size = batch_size) # 在缓冲区中随机打乱数据
dataset = dataset.padded_batch(batch_size = batch_size,
                               padded_shapes = pad_shapes,
                              padding_values = pad_values) # 每1024条数据为一个
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

# 验证集
filenames_val= tf.data.Dataset.list_files(['./user_item_act_test_val.csv'])
dataset_val = filenames_val.flat_map(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1))

val_batch_size = 2
dataset_val = dataset_val.map(_parse_function, num_parallel_calls=60)
dataset_val = dataset_val.padded_batch(batch_size = val_batch_size,
                               padded_shapes = pad_shapes,
                              padding_values = pad_values) # 每1024条数据为一个
dataset_val = dataset_val.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

########################################################################
               #################自定义Layer##############
########################################################################


# 多值查找表稀疏SparseTensor >>  EncodeMultiEmbedding
class VocabLayer(Layer):
    def __init__(self, keys, mask_value=None, **kwargs):
        super(VocabLayer, self).__init__(**kwargs)
```

```
        self.mask_value = mask_value
        vals = tf.range(2, len(keys) + 2)
        vals = tf.constant(vals, dtype=tf.int32)
        keys = tf.constant(keys)
        self.table = tf.lookup.StaticHashTable(
            tf.lookup.KeyValueTensorInitializer(keys, vals), 1)

    def call(self, inputs):
        idx = self.table.lookup(inputs)
        if self.mask_value is not None:
            masks = tf.not_equal(inputs, self.mask_value)
            paddings = tf.ones_like(idx) * (0) # mask成 0
            idx = tf.where(masks, idx, paddings)
        return idx

    def get_config(self):
        config = super(VocabLayer, self).get_config()
        config.update({'mask_value': self.mask_value, })
        return config


class EmbeddingLookupSparse(Layer):
    def __init__(self, embedding, has_weight=False, combiner='sum',**kwargs)

        super(EmbeddingLookupSparse, self).__init__(**kwargs)
        self.has_weight = has_weight
        self.combiner = combiner
        self.embedding = embedding


    def build(self, input_shape):
        super(EmbeddingLookupSparse, self).build(input_shape)

    def call(self, inputs):
        if self.has_weight:
            idx, val = inputs
            combiner_embed = tf.nn.embedding_lookup_sparse(self.embedding,sp_
        else:
            idx = inputs
            combiner_embed = tf.nn.embedding_lookup_sparse(self.embedding,sp_
```

```
            return tf.expand_dims(combiner_embed, 1)

    def get_config(self):
        config = super(EmbeddingLookupSparse, self).get_config()
        config.update({'has_weight': self.has_weight, 'combiner':self.combin
        return config


class EmbeddingLookup(Layer):
    def __init__(self, embedding, **kwargs):

        super(EmbeddingLookup, self).__init__(**kwargs)
        self.embedding = embedding


    def build(self, input_shape):
        super(EmbeddingLookup, self).build(input_shape)

    def call(self, inputs):
        idx = inputs
        embed = tf.nn.embedding_lookup(params=self.embedding, ids=idx)
        return embed

    def get_config(self):
        config = super(EmbeddingLookup, self).get_config()
        return config



# 稠密转稀疏
class DenseToSparseTensor(Layer):
    def __init__(self, mask_value= -1, **kwargs):
        super(DenseToSparseTensor, self).__init__()
        self.mask_value = mask_value


    def call(self, dense_tensor):
        idx = tf.where(tf.not_equal(dense_tensor, tf.constant(self.mask_valu
        sparse_tensor = tf.SparseTensor(idx, tf.gather_nd(dense_tensor, idx)
        return sparse_tensor
```

```

    def get_config(self):
        config = super(DenseToSparseTensor, self).get_config()
        config.update({'mask_value': self.mask_value})
        return config


# 自定义dnese层 含BN， dropout
class CustomDense(Layer):
    def __init__(self, units=32, activation='tanh', dropout_rate =0, use_bn=l
        self.units = units
        self.activation = activation
        self.dropout_rate = dropout_rate
        self.use_bn = use_bn
        self.seed = seed
        self.tag_name = tag_name

        super(CustomDense, self).__init__(**kwargs)

    #build方法一般定义Layer需要被训练的参数。
    def build(self, input_shape):
        self.weight = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True,
                                 name='kernel_' + self.tag_name)
        self.bias = self.add_weight(shape=(self.units,),
                                 initializer='random_normal',
                                 trainable=True,
                                 name='bias_' + self.tag_name)

        if self.use_bn:
            self.bn_layers = tf.keras.layers.BatchNormalization()

        self.dropout_layers = tf.keras.layers.Dropout(self.dropout_rate)
        self.activation_layers = tf.keras.layers.Activation(self.activation,

        super(CustomDense,self).build(input_shape) # 相当于设置self.built = Tr

    #call方法一般定义正向传播运算逻辑，__call__方法调用了它。
    def call(self, inputs,training=None, **kwargs):
```

```
        fc = tf.matmul(inputs, self.weight) + self.bias
        if self.use_bn:
            fc = self.bn_layers(fc)
        out_fc = self.activation_layers(fc)

        return out_fc

    #如果要让自定义的Layer通过Functional API 组合成模型时可以序列化，需要自定义get_
    def get_config(self):
        config = super(CustomDense, self).get_config()
        config.update({'units': self.units, 'activation': self.activation, 'u
                       'dropout_rate': self.dropout_rate, 'seed': self.seed,
        return config


class HashLayer(Layer):
    """
    hash the input to [0,num_buckets)
    if mask_zero = True,0 or 0.0 will be set to 0,other value will be set in
    """

    def __init__(self, num_buckets, mask_zero=False, **kwargs):
        self.num_buckets = num_buckets
        self.mask_zero = mask_zero
        super(HashLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Be sure to call this somewhere!
        super(HashLayer, self).build(input_shape)

    def call(self, x, mask=None, **kwargs):
        zero = tf.as_string(tf.zeros([1], dtype='int32'))
        num_buckets = self.num_buckets if not self.mask_zero else self.num_bu
        hash_x = tf.strings.to_hash_bucket_fast(x, num_buckets, name=None)
        if self.mask_zero:
            mask = tf.cast(tf.not_equal(x, zero), dtype='int64')
            hash_x = (hash_x + 1) * mask

        return hash_x
    def get_config(self, ):
```

```
        config = super(HashLayer, self).get_config()
        config.update({'num_buckets': self.num_buckets, 'mask_zero': self.ma
        return config

#  Attention池化层
class AttentionPoolingLayer(Layer):
    """
      Input shape
        - A list of three tensor: [query,keys,his_seq]
        - query is a 3D tensor with shape:  ``(batch_size, 1, embedding_size
        - keys is a 3D tensor with shape:   ``(batch_size, T, embedding_size
        - his_seq is a 2D tensor with shape: ``(batch_size, T)``
      Output shape
        - 3D tensor with shape: ``(batch_size, 1, embedding_size)``.
      Arguments
        - **att_hidden_units**:list of positive integer, the attention net la
        - **att_activation**: Activation function to use in attention net.
        - **weight_normalization**: bool.Whether normalize the attention scor
        - **hist_mask_value**: the mask value of his_seq.
      References
        - [Zhou G, Zhu X, Song C, et al. Deep interest network for click-thro
    """

    def __init__(self, att_hidden_units=(80, 40), att_activation='sigmoid', w
                 mode="sum",hist_mask_value=0, **kwargs):

        self.att_hidden_units = att_hidden_units
        self.att_activation = att_activation
        self.weight_normalization = weight_normalization
        self.mode = mode
        self.hist_mask_value = hist_mask_value
        super(AttentionPoolingLayer, self).__init__(**kwargs)


    def build(self, input_shape):

        self.fc = tf.keras.Sequential()
        for unit in self.att_hidden_units:
            self.fc.add(layers.Dense(unit, activation=self.att_activation, na
        self.fc.add(layers.Dense(1, activation=None, name="fc_att_out"))
```

```

        super(AttentionPoolingLayer, self).build(
            input_shape)  # Be sure to call this somewhere!

    def call(self, inputs, **kwargs):
        query, keys, his_seq = inputs
        # 计算掩码
        key_masks = tf.not_equal(his_seq, tf.constant(self.hist_mask_value ,
        key_masks = tf.expand_dims(key_masks, 1)

        # 1. 转换query维度，变成历史维度T
        # query是[B, 1,  H]，转换到 queries 维度为(B, T, H)，为了让pos_item和用
        queries = tf.tile(query, [1, tf.shape(keys)[1], 1]) # [B, T, H]

        # 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
        # 得到 Local Activation Unit 的输入。即 候选queries 对应的 emb，用户历史
        # 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, keys, queries-keys, queries*keys], axi
        # 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
        attention_score = self.fc(din_all) # [B, T, 1]
        # attention的输出, [B, 1, T]
        outputs = tf.transpose(attention_score, (0, 2, 1)) # [B, 1, T]

        # 4. 得到有真实意义的score
        if self.weight_normalization:
            # padding的mask后补一个很小的负数，这样后面计算 softmax 时, e^{x} 结
            paddings = tf.ones_like(outputs) * (-2 ** 32 + 1)
        else:
            paddings = tf.zeros_like(outputs)
        outputs = tf.where(key_masks, outputs, paddings)  # [B, 1, T]

        # 5. Activation，得到归一化后的权重
        if self.weight_normalization:
            outputs = tf.nn.softmax(outputs)  # [B, 1, T]

        # 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
        # Weighted sum,
        if self.mode == 'sum':
            # outputs 的大小为 [B, 1, T], 表示每条历史行为的权重,
            # keys 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 outputs 就是 [B, 1, H]
                outputs = tf.matmul(outputs, keys)  # [B, 1, H]
            else:
                # 从 [B, 1, H] 变化成 Batch * Time
                outputs = tf.reshape(outputs, [-1, tf.shape(keys)[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] =
                outputs = keys * tf.expand_dims(outputs, -1)
                outputs = tf.reshape(outputs, tf.shape(keys)) # Batch * Time * H

            return outputs

        def get_config(self, ):

            config = {'att_hidden_units': self.att_hidden_units, 'att_activation
                      'weight_normalization': self.weight_normalization, 'mode':
            base_config = super(AttentionPoolingLayer, self).get_config()
            return config.update(base_config)


    class Add(tf.keras.layers.Layer):
        def __init__(self, **kwargs):
            super(Add, self).__init__(**kwargs)

        def build(self, input_shape):
            # Be sure to call this somewhere!
            super(Add, self).build(input_shape)

        def call(self, inputs, **kwargs):
            if not isinstance(inputs,list):
                return inputs
            if len(inputs) == 1  :
                return inputs[0]
            if len(inputs) == 0:
                return tf.constant([[0.0]])
            return tf.keras.layers.add(inputs)


    ########################################################################
                   ################定义输入帮助函数##############
    ########################################################################
```

```
# 定义model输入特征
def build_input_features(features_columns, prefix=''):
    input_features = OrderedDict()

    for feat_col in features_columns:
        if isinstance(feat_col, DenseFeat):
            input_features[feat_col.name] = Input([feat_col.dim], name=feat_c
        elif isinstance(feat_col, SparseFeat):
            input_features[feat_col.name] = Input([1], name=feat_col.name, dt
        elif isinstance(feat_col, VarLenSparseFeat):
            input_features[feat_col.name] = Input([None], name=feat_col.name,
            if feat_col.weight_name is not None:
                input_features[feat_col.weight_name] = Input([None], name=fea
        else:
            raise TypeError("Invalid feature column in build_input_features:

    return input_features

# 构造 自定义embedding层 matrix
def build_embedding_matrix(features_columns):
    embedding_matrix = {}
    for feat_col in features_columns:
        if isinstance(feat_col, SparseFeat) or isinstance(feat_col, VarLenSpa
            vocab_name = feat_col.share_embed if feat_col.share_embed else fe
            vocab_size = feat_col.voc_size + 2
            embed_dim = feat_col.embed_dim
            if vocab_name not in embedding_matrix:
                embedding_matrix[vocab_name] = tf.Variable(initial_value=tf.r
                                                                            st
    return embedding_matrix

# 构造 自定义embedding层
def build_embedding_dict(features_columns, embedding_matrix):
    embedding_dict = {}
    for feat_col in features_columns:
        if isinstance(feat_col, SparseFeat):
            vocab_name = feat_col.share_embed if feat_col.share_embed else fe
            embedding_dict[feat_col.name] = EmbeddingLookup(embedding=embeddi
        elif isinstance(feat_col, VarLenSparseFeat):
            vocab_name = feat_col.share_embed if feat_col.share_embed else fe
```

```
            if feat_col.combiner is not None:
                if feat_col.weight_name is not None:
                    embedding_dict[feat_col.name] = EmbeddingLookupSparse(eml
                else:
                    embedding_dict[feat_col.name] = EmbeddingLookupSparse(eml
            else:
                embedding_dict[feat_col.name] = EmbeddingLookup(embedding=eml

    return embedding_dict


# dense 与 embedding特征输入
def input_from_feature_columns(features, features_columns, embedding_dict):
    sparse_embedding_list = []
    dense_value_list = []

    for feat_col in features_columns:
        if isinstance(feat_col, SparseFeat):
            _input = features[feat_col.name]
            if feat_col.dtype == 'string':
                if feat_col.hash_size is None:
                    vocab_name = feat_col.share_embed if feat_col.share_embe
                    keys = DICT_CATEGORICAL[vocab_name]
                    _input = VocabLayer(keys)(_input)
                else:
                    _input = HashLayer(num_buckets=feat_col.hash_size, mask_

            embed = embedding_dict[feat_col.name](_input)
            sparse_embedding_list.append(embed)
        elif isinstance(feat_col, VarLenSparseFeat):
            _input = features[feat_col.name]
            if feat_col.dtype == 'string':
                if feat_col.hash_size is None:
                    vocab_name = feat_col.share_embed if feat_col.share_embe
                    keys = DICT_CATEGORICAL[vocab_name]
                    _input = VocabLayer(keys, mask_value='0')(_input)
                else:
                    _input = HashLayer(num_buckets=feat_col.hash_size, mask_
            if feat_col.combiner is not None:
                input_sparse =  DenseToSparseTensor(mask_value=0)(_input)
```

```
                if feat_col.weight_name is not None:
                    weight_sparse = DenseToSparseTensor()(features[feat_col.w
                    embed = embedding_dict[feat_col.name]([input_sparse, weig
                else:
                    embed = embedding_dict[feat_col.name](input_sparse)
            else:
                embed = embedding_dict[feat_col.name](_input)

            sparse_embedding_list.append(embed)

        elif isinstance(feat_col, DenseFeat):
            dense_value_list.append(features[feat_col.name])

        else:
            raise TypeError("Invalid feature column in input_from_feature_co

    return sparse_embedding_list, dense_value_list


def concat_func(inputs, axis=-1):
    if len(inputs) == 1:
        return inputs[0]
    else:
        return Concatenate(axis=axis)(inputs)

def combined_dnn_input(sparse_embedding_list, dense_value_list):
    if len(sparse_embedding_list) > 0 and len(dense_value_list) > 0:
        sparse_dnn_input = Flatten()(concat_func(sparse_embedding_list))
        dense_dnn_input = Flatten()(concat_func(dense_value_list))
        return concat_func([sparse_dnn_input, dense_dnn_input])
    elif len(sparse_embedding_list) > 0:
        return Flatten()(concat_func(sparse_embedding_list))
    elif len(dense_value_list) > 0:
        return Flatten()(concat_func(dense_value_list))
    else:
        raise "dnn_feature_columns can not be empty list"


def get_linear_logit(sparse_embedding_list, dense_value_list):

```

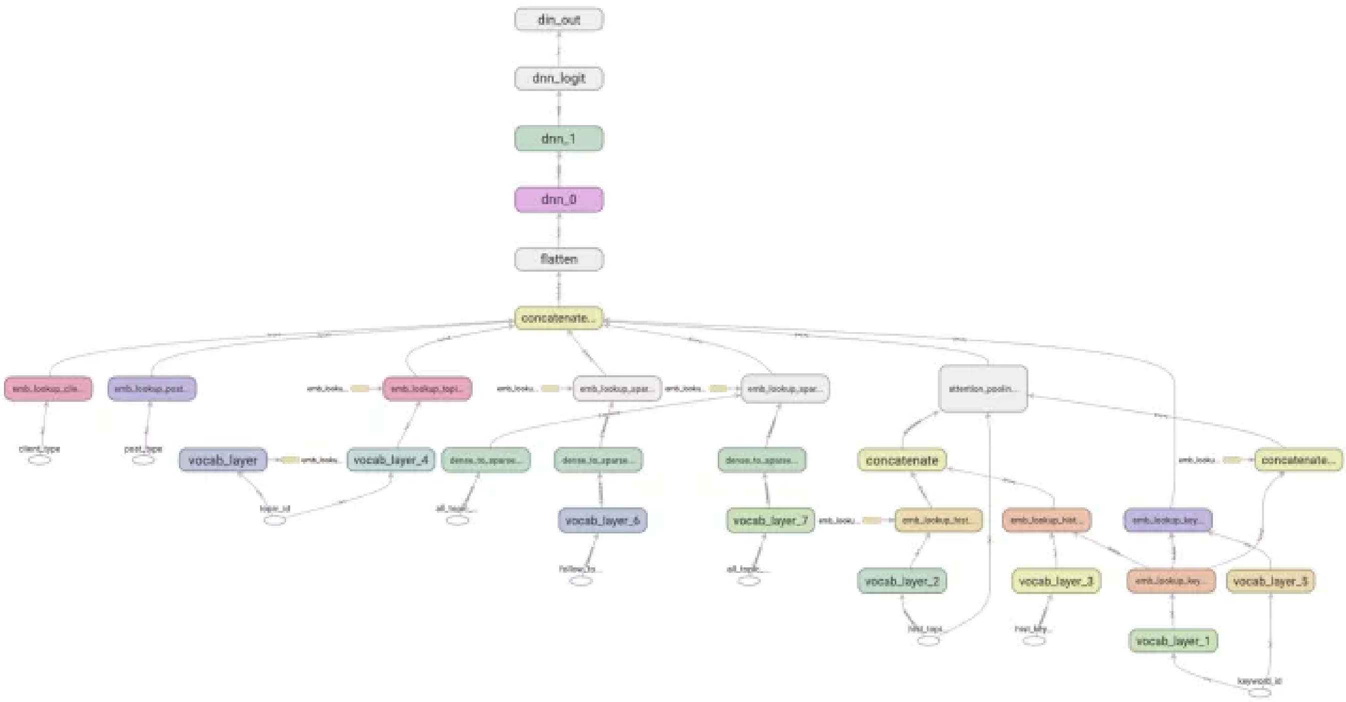
```
    if len(sparse_embedding_list) > 0 and len(dense_value_list) > 0:
        sparse_linear_layer = Add()(sparse_embedding_list)
        sparse_linear_layer = Flatten()(sparse_linear_layer)
        dense_linear = concat_func(dense_value_list)
        dense_linear_layer = Dense(1)(dense_linear)
        linear_logit = Add()([dense_linear_layer, sparse_linear_layer])
        return linear_logit
    elif len(sparse_embedding_list) > 0:
        sparse_linear_layer = Add()(sparse_embedding_list)
        sparse_linear_layer = Flatten()(sparse_linear_layer)
        return sparse_linear_layer
    elif len(dense_value_list) > 0:
        dense_linear = concat_func(dense_value_list)
        dense_linear_layer = Dense(1)(dense_linear)
        return dense_linear_layer
    else:
        raise "linear_feature_columns can not be empty list"

########################################################################
               ################定义模型##############
########################################################################

def DIN(feature_columns, history_feature_names, hist_mask_value,dnn_use_bn=Fa
        dnn_hidden_units=(200, 80), dnn_activation='relu', att_hidden_size=(8
        att_weight_normalization=True, dnn_dropout=0, seed=1024):

    """Instantiates the Deep Interest Network architecture.
    Args:
        dnn_feature_columns: An iterable containing all the features used by
        history_feature_names: list,to indicate  sequence sparse field
        dnn_use_bn: bool. Whether use BatchNormalization before activation or
        dnn_hidden_units: list,list of positive integer or empty list, the la
        dnn_activation: Activation function to use in deep net
        att_hidden_size: list,list of positive integer , the layer number and
        att_activation: Activation function to use in attention net
        att_weight_normalization: bool.Whether normalize the attention score
        dnn_dropout: float in [0,1), the probability we will drop out a given
        seed: integer ,to use as random seed.
    return: A Keras model instance.
    """
```

```
    features = build_input_features(feature_columns)

    sparse_feature_columns = list(
        filter(lambda x: isinstance(x, SparseFeat), feature_columns)) if fea
    dense_feature_columns = list(
        filter(lambda x: isinstance(x, DenseFeat), feature_columns)) if featu
    varlen_sparse_feature_columns = list(
        filter(lambda x: isinstance(x, VarLenSparseFeat), feature_columns)) i

    query_feature_columns = []
    for fc in sparse_feature_columns:
        feature_name = fc.name
        if feature_name in history_feature_names:
            query_feature_columns.append(fc)
    key_feature_columns = []
    sparse_varlen_feature_columns = []
    history_fc_names = list(map(lambda x: "hist_" + x, history_feature_names)
    for fc in varlen_sparse_feature_columns:
        feature_name = fc.name
        if feature_name in history_fc_names:
            key_feature_columns.append(fc)
        else:
            sparse_varlen_feature_columns.append(fc)

    inputs_list = list(features.values())

    # 构建 embedding_dict
    embedding_matrix = build_embedding_matrix(feature_columns)
    embedding_dict = build_embedding_dict(feature_columns, embedding_matrix)

    query_emb_list, _ = input_from_feature_columns(features, query_feature_co
    keys_emb_list, _ = input_from_feature_columns(features, key_feature_colu
    merge_dnn_columns = sparse_feature_columns + sparse_varlen_feature_colum
    dnn_sparse_embedding_list, dnn_dense_value_list = input_from_feature_colu

    keys_emb = concat_func(keys_emb_list)
    query_emb = concat_func(query_emb_list)
    keys_seq = features[key_feature_columns[0].name]

    hist_attn_emb = AttentionPoolingLayer(att_hidden_units=att_hidden_size, a
```

```
    dnn_input = combined_dnn_input(dnn_sparse_embedding_list+[hist_attn_emb],

    # DNN
    for i in range(len(dnn_hidden_units)):
        if i == len(dnn_hidden_units) - 1:
            dnn_out = CustomDense(units=dnn_hidden_units[i],dropout_rate=dnn_
                                  use_bn=dnn_use_bn, activation=dnn_act
            break
        dnn_input = CustomDense(units=dnn_hidden_units[i],dropout_rate=dnn_d
                                use_bn=dnn_use_bn, activation=dnn_activa
    dnn_logit = Dense(1, use_bias=False, activation=None, kernel_initializer=
    output = tf.keras.layers.Activation("sigmoid", name="din_out")(dnn_logit)
    model = Model(inputs=inputs_list, outputs=output)

    return model



model = DIN(feature_columns, history_feature_names, hist_mask_value='0', dnn_
        dnn_hidden_units=(200, 80), dnn_activation='relu', att_hidden_size=(8
        att_weight_normalization=True, dnn_dropout=0, seed=1024)

model.compile(optimizer="adam", loss= "binary_crossentropy",  metrics=tf.kera

log_dir = '/mywork/tensorboardshare/logs/' + datetime.datetime.now().strftime
tbCallBack = TensorBoard(log_dir=log_dir,  # log 目录
                 histogram_freq=0,  # 按照何等频率（epoch）来计算直方图，0为不计算
                 write_graph=True,  # 是否存储网络结构图
                 write_images=True,# 是否可视化参数
                 update_freq='epoch',
                 embeddings_freq=0,
                 embeddings_layer_names=None,
                 embeddings_metadata=None,
                        profile_batch = 20)

total_train_sample =  10000
total_test_sample =    10
train_steps_per_epoch=np.floor(total_train_sample/batch_size).astype(np.int32
test_steps_per_epoch = np.ceil(total_test_sample/val_batch_size).astype(np.in
history_loss = model.fit(dataset, epochs=3,
```

```
            steps_per_epoch=train_steps_per_epoch,
            validation_data=dataset_val, validation_steps=test_steps_per_epoch
            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]