如何实现支持多值、稀疏、共享权重的DeepFM

原创 石塔西 深度传送门 2019-10-29

作者：石塔西
来源：https://zhuanlan.zhihu.com/p/48057256
整理：深度传送门

缘起

DeepFM不算什么新技术了，用TensorFlow实现DeepFM也有开源实现，**那我为什么要炒这个冷饭，重复造轮子?**

用Google搜索“TensorFlow+DeepFM”，一般都能搜索到“**tensorflow-DeepFM**”[1] 和“**TensorFlow Estimator of DeepFM**”[2]这二位的实现。二位不仅用TensorFlow实现了DeepFM，还在Criteo数据集上，给出了完整的训练、测试的代码，的确给了我很大的启发，在这里要表示感谢。

但是，同样是由于二位的实现都是根据Criteo简单数据集的，使他们的代码，**如果移植到实际的推荐系统中，存在一定困难**。比如：

稀疏要求。尽管criteo的原始数据集是排零存储的，但是以上的两个实现，都是用稠密矩阵来表示输入，将0又都补了回来。这种做法，在criteo这种只有39列的简单数据集上是可行的，但是**实际系统中，特征数量以千、万计，这种稀疏转稠密的方式是不可取的**。

一列多值的要求。Criteo数据集有13列numeric特征+26列categorical特征，所有列都只有一个值。但是，**在实际系统中，一个field下往往有多个<feature:value>对**。比如，我们用三个field来描述一个用户的手机 使用习惯，“近xxx天活跃app”+“近xxx天新安装app”+“近xxx天卸载app”。每个field下，再有“微信:0.9，微博:0.5，淘宝:0.3，……”等一系列的feature和它们的数值。

这个要求固然可以通过，去除field这个“特征单位”，只针对一个个独立的feature来建模。但是，这样一来，既凭空增加了模型的规模，又破坏模型的“层次化”与“模块化”，使代码不易扩展与维护。

权值共享的要求。Criteo数据集经过脱敏感处理，我们无法知道每列的具体含义，自然也就没有列与列之间共享权重的需求，以上提到的两个实现也就只用一整块稠密矩阵来建模embedding矩阵。

但是，以上面提到的“近xxx天活跃app”+“近xxx天新安装app”+“近xxx天卸载app”这三个field为例，这些 field中的feature都来源于同一个“app字典”。如果不做权重共享，

- 每个field都使用独立的embedding矩阵来映射app向量，整个模型需要优化的变量是共享权重模型的3倍，既耗费了更多的计算资源，也容易导致过拟合。
- 每个field的稀疏程度是不一样的，同一个app，在“活跃列表”中出现得更频繁，其embedding向量就有更多的训练机会，而在“卸载列表”中较少出现，其embedding向量得不到足够训练，恐怕最后与随机初始化无异。

因此，**在实际系统中，“共享权重”是必须的**，

- 减小优化变量的数目，既节省计算资源，又减轻“过拟合”风险
- 同一个embedding矩阵，为多个field提供映射向量，类似于“多任务学习”，使每个embedding向量得到更多的训练机会，同时也要满足多个field的需求（比如同一个app的向量，既要体现‘经常使用它’对y的影响，也要体现‘卸载它’对y值的影响），也降低了“过拟合”的风险。

正因为在目前我能够找到的基于TensorFlow实现的DeepFM中，没有一个能够满足以上“**稀疏**”、“**多值**”、“**共享权重**”这三个要求的，所以，我自己动手实现了一个，代码见我的Github[3]。接下来，我简单讲解一下我的代码。

数据预处理

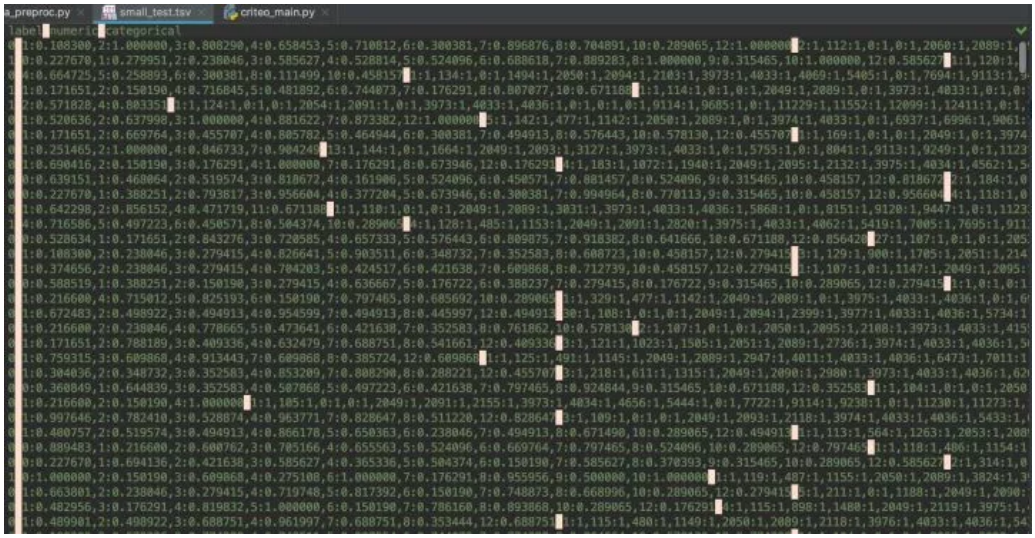
我依然用criteo数据集来做演示之用。为了演示“**一列多值**”和“**稀疏**”，我把criteo中的特征分为两个field，所有数值特征I1~I13归为numeric field，所有类别特征C1~C26归为categorical field。需要特别指出的是：

- 这种处理方法，不是为了提高criteo数据集上的模型性能，只是为了模拟实际系统中将会遇到的“一列多值”和“稀疏”数据集。接下来会看到，DeepFM中，FM中的二阶交叉，不会受拆分成两个field的影响。受影响的主要是Deep侧的输入层，详情见”DNN预测部分”一节 。
- 另外，criteo数据集**无法**演示“**权重共享**”的功能。

对criteo中数值特征与类别特征，都是最常规的预处理，不是这次演示的重点

- 数值特征，因为多数表示"次数"，因此先做了一个log变化，减弱长尾数据的影响，再做了一个min/max scaling，毕竟底层还是线性算法，要排除特征间不同scale的影响。注意，**千万不能做“zero mean, unit variance”的standardize**，因为那样会破坏数据的稀疏性。
- 类别特征，剔除了一些生僻的tag，建立字典，将原始数据中的字符串tag转化为整数的index

预处理的代码见criteo_data_preproc.py，处理好的数据文件如下所示，图中的亮块是列分隔符。可以看到，每列是由多个tag_index:value“键值对”组成的，而**不同行中“键值对”个数互不同，而value绝没有0，实现排零、稀疏存储。**



输入数据

input_fn

为了配合TensorFlow Estimator，我们需要定义input_fn来读取上图所示的数据。看似简单的任务，实现起来，却很花费了我一番功夫：

- 网上能够搜到的TensorFlow读文本文件的代码，都是读“每列只有一个值的csv”这样规则的数据格式。但是，上图所示的数据，**却非常不规则**，每行先是由“\t”分隔，第列中再由“,”分隔成数目不同的“键值对”，每个‘键值对’再由“:”分隔。
- 我希望提供给model稀疏矩阵，方便model中排零计算，提升效率。

最终，解析一行文本的代码如下。

```
def _decode_tsv(line):
    columns = tf.decode_csv(line, record_defaults=DEFAULT_VALUES, field_delim=
    y = columns[0]

    feat_columns = dict(zip((t[0] for t in COLUMNS_MAX_TOKENS), columns[1:]))
    X = {}
    for colname, max_tokens in COLUMNS_MAX_TOKENS:
        # 调用string_split时，第一个参数必须是一个list，所以要把columns[colname]放
        # 这时每个kv还是'k:v'这样的字符串
        kvpairs = tf.string_split([feat_columns[colname]], ',').values[:max_to

        # k,v已经拆开，kvpairs是一个SparseTensor，因为每个kvpair格式相同，都是"k:v
```

```
        # 既不会出现"k"，也不会出现"k:v1:v2:v3:..."
        # 所以，这时的kvpairs实际上是一个满阵
        kvpairs = tf.string_split(kvpairs, ':')

        # kvpairs是一个[n_valid_pairs,2]矩阵
        kvpairs = tf.reshape(kvpairs.values, kvpairs.dense_shape)

        feat_ids, feat_vals = tf.split(kvpairs, num_or_size_splits=2, axis=1)
        feat_ids = tf.string_to_number(feat_ids, out_type=tf.int32)
        feat_vals = tf.string_to_number(feat_vals, out_type=tf.float32)

        # 不能调用squeeze，squeeze的限制太多，当原始矩阵有1行或0行时，squeeze都会报
        X[colname + "_ids"] = tf.reshape(feat_ids, shape=[-1])
        X[colname + "_values"] = tf.reshape(feat_vals, shape=[-1])

    return X, y
```

然后，将整个文件转化成TensorFlow Dataset的代码如下所示。**每一个field“xxx”在dataset中将由两个SparseTensor表示**，“xxx_ids”表示sparse ids，“xxx_values”表示sparse values。

```
def input_fn(data_file, n_repeat, batch_size, batches_per_shuffle):
    # ------------ prepare padding
    pad_shapes = {}
    pad_values = {}
    for c, max_tokens in COLUMNS_MAX_TOKENS:
        pad_shapes[c + "_ids"] = tf.TensorShape([max_tokens])
        pad_shapes[c + "_values"] = tf.TensorShape([max_tokens])

        pad_values[c + "_ids"] = -1  # 0 is still valid token-id, -1 for paddi
        pad_values[c + "_values"] = 0.0

    # no need to pad labels
    pad_shapes = (pad_shapes, tf.TensorShape([]))
    pad_values = (pad_values, 0)

    # ------------ define reading ops
    dataset = tf.data.TextLineDataset(data_file).skip(1)  # skip the header
    dataset = dataset.map(_decode_tsv, num_parallel_calls=4)
```

```
19
20      if batches_per_shuffle > 0:
21          dataset = dataset.shuffle(batches_per_shuffle * batch_size)
22
23      dataset = dataset.repeat(n_repeat)
24      dataset = dataset.padded_batch(batch_size=batch_size,
25                                     padded_shapes=pad_shapes,
26                                     padding_values=pad_values)
27
28      iterator = dataset.make_one_shot_iterator()
29      dense_Xs, ys = iterator.get_next()
30
31      # ----------- convert dense to sparse
32      sparse_Xs = {}
33      for c, _ in COLUMNS_MAX_TOKENS:
34          for suffix in ["ids", "values"]:
35              k = "{}_{}".format(c, suffix)
36              sparse_Xs[k] = tf_utils.to_sparse_input_and_drop_ignore_values(der
37
38      # ----------- return
39      return sparse_Xs, ys
```

其中也不得不调用padded_batch补齐，这一步也将稀疏格式转化成了稠密格式，不过**只是在一个batch**(batch_size=128已经算很大了)**中临时稠密一下**，很快就又通过调用to_sparse_input_and_drop_ignore_values这个函数重新转化成稀疏格式了。to_sparse_input_and_drop_ignore_values实际上是从feature_column.py这个module中的_to_sparse_input_and_drop_ignore_values 函数拷贝而来，因为原函数不是public的，无法在featurecolumn.py以外调用，所以我将它的代码拷贝到tf_utils.py中。

建立共享权重

重申几个概念。比如我们的特征集中包括active_pkgs（app活跃情况）、install_pkgs（app安装情况）、uninstall_pkgs（app卸载情况）。每列包含的内容是一系列feature和其数值，比如qq:0.1, weixin:0.9, taobao:1.1, ……。这些feature都来源于同一份名为package的字典

- **field就是**active_pkgs、install_pkgs、uninstall_pkgs**这些大类**，是DataFrame中的每一列

- feature就是每个field下包含的具体内容，**一个field下允许存在多个feature**，比如前面提到的qq, weixin, taobao这样的app名称。
- vocabulary对应例子中的“package字典”。**不同field下的feature可以来自同一个vocabulary**，即若干field共享vocabulary

建立共享权重的代码如下所示：

- 一个vocab对应两个embedding矩阵，一个对应FM中的线性部分的权重，另一个对应FM与DNN共享的隐向量（用于二阶与高阶交叉）。
- 所有embedding矩阵，以”字典名”存入dict。**不同field只要指定相同的“字典名”，就可以共享同一套embedding矩阵。**

```
class EmbeddingTable:
    def __init__(self):
        self._weights = {}

    def add_weights(self, vocab_name, vocab_size, embed_dim):
        """
        :param vocab_name: 一个field拥有两个权重矩阵，一个用于线性连接，另一个用于非
        :param vocab_size: 字典总长度
        :param embed_dim: 二阶权重矩阵shape=[vocab_size, order2dim]，映射成的emb
                          既用于接入DNN的第一屋，也是用于FM二阶交互的隐向量
        :return: None
        """
        linear_weight = tf.get_variable(name='{}_linear_weight'.format(vocab_n
                                        shape=[vocab_size, 1],
                                        initializer=tf.glorot_normal_initializ
                                        dtype=tf.float32)

        # 二阶（FM）与高阶（DNN）的特征交互，共享embedding矩阵
        embed_weight = tf.get_variable(name='{}_embed_weight'.format(vocab_nam
                                       shape=[vocab_size, embed_dim],
                                       initializer=tf.glorot_normal_initialize
                                       dtype=tf.float32)

        self._weights[vocab_name] = (linear_weight, embed_weight)

    def get_linear_weights(self, vocab_name): return self._weights[vocab_name]

    def get_embed_weights(self, vocab_name): return self._weights[vocab_name][
```

```
29
30
31 def build_embedding_table(params):
32     embed_dim = params['embed_dim']  # 必须有统一的embedding长度
33
34     embedding_table = EmbeddingTable()
35     for vocab_name, vocab_size in params['vocab_sizes'].items():
36         embedding_table.add_weights(vocab_name=vocab_name, vocab_size=vocab_si
37
38     return embedding_table
```

线性预测部分

```
1  def output_logits_from_linear(features, embedding_table, params):
2      field2vocab_mapping = params['field_vocab_mapping']
3      combiner = params.get('multi_embed_combiner', 'sum')
4
5      fields_outputs = []
6      # 当前field下有一系列的<tag:value>对，每个tag对应一个bias（待优化），
7      # 将所有tag对应的bias，按照其value进行加权平均，得到这个field对应的bias
8      for fieldname, vocabname in field2vocab_mapping.items():
9          sp_ids = features[fieldname + "_ids"]
10         sp_values = features[fieldname + "_values"]
11
12         linear_weights = embedding_table.get_linear_weights(vocab_name=vocabna
13
14         # weights: [vocab_size,1]
15         # sp_ids: [batch_size, max_tags_per_example]
16         # sp_weights: [batch_size, max_tags_per_example]
17         # output: [batch_size, 1]
18         output = embedding_ops.safe_embedding_lookup_sparse(linear_weights, sp
19                                                             combiner=combiner,
20                                                             name='{}_linear_ou
21
22         fields_outputs.append(output)
23
24     # 因为不同field可以共享同一个vocab的linear weight，所以将各个field的output相加,
```

```
    # 因此，所有field对应的output拼接起来，反正每个field的output都是[batch_size,1]
    # whole_linear_output: [batch_size, total_fields]
    whole_linear_output = tf.concat(fields_outputs, axis=1)
    tf.logging.info("linear output, shape={}".format(whole_linear_output.shape

    # 再映射到final logits（二分类，也是[batch_size,1]）
    # 这时，就不要用任何activation了，特别是ReLU
    return tf.layers.dense(whole_linear_output, units=1, use_bias=True, activa
```

二阶交互预测部分

二阶交互部分与DeepFM论文中稍有不同，而是使用了《Neural Factorization Machines for Sparse Predictive Analytics》中Bi-Interaction的公式。这也是网上实现的通用做法。

$$f_{BI}(\mathcal{V}_x) = \sum_{i=1}^{n}\sum_{j=i+1}^{n} x_i\mathbf{v}_i \odot x_j\mathbf{v}_j = \frac{1}{2}\left[(\sum_{i=1}^{n} x_i\mathbf{v}_i)^2 - \sum_{i=1}^{n}(x_i\mathbf{v}_i)^2\right]$$

而我的实现与上边公式最大的不同，就是**不再只有一个embedding矩阵V**，而是每个feature根据自己所在的field，再根据超参指定的field与vocabulary的映射关系，找到自己对应的embedding矩阵。某个field对应的embedding矩阵有可能是与另外一个field共享的。

另外，$x_i v_i$实现了**稀疏矩阵相乘**，基于embedding_ops. safe_embedding_lookup_sparse实现。

```
def output_logits_from_bi_interaction(features, embedding_table, params):
    field2vocab_mapping = params['field_vocab_mapping']

    # 论文上的公式就是要求sum，而且我也试过mean和sqrtn，都比用mean要差上很多
    # 但是，这种情况，仅仅是针对criteo数据的，还是理论上就必须用sum，而不能用mean和sq
    # 我还不太确定，所以保留一个接口能指定其他combiner的方法
    combiner = params.get('multi_embed_combiner', 'sum')

    # 见《Neural Factorization Machines for Sparse Predictive Analytics》论文的
    fields_embeddings = []
    fields_squared_embeddings = []

```

```
    for fieldname, vocabname in field2vocab_mapping.items():
        sp_ids = features[fieldname + "_ids"]
        sp_values = features[fieldname + "_values"]

        # --------- embedding
        embed_weights = embedding_table.get_embed_weights(vocabname)
        # embedding: [batch_size, embed_dim]
        embedding = embedding_ops.safe_embedding_lookup_sparse(embed_weights,
                                                               combiner=combir
                                                               name='{}_embedc
        fields_embeddings.append(embedding)

        # --------- square of embedding
        squared_emb_weights = tf.square(embed_weights)

        squared_sp_values = tf.SparseTensor(indices=sp_values.indices,
                                            values=tf.square(sp_values.values)
                                            dense_shape=sp_values.dense_shape)

        # squared_embedding: [batch_size, embed_dim]
        squared_embedding = embedding_ops.safe_embedding_lookup_sparse(squarec
                                                                       combine
                                                                       name='{
        fields_squared_embeddings.append(squared_embedding)

    # calculate bi-interaction
    sum_embedding_then_square = tf.square(tf.add_n(fields_embeddings))  # [bat
    square_embedding_then_sum = tf.add_n(fields_squared_embeddings)  # [batch_
    bi_interaction = 0.5 * (sum_embedding_then_square - square_embedding_then_
    tf.logging.info("bi-interaction, shape={}".format(bi_interaction.shape))

    # calculate logits
    logits = tf.layers.dense(bi_interaction, units=1, use_bias=True, activatic

    # 因为FM与DNN共享embedding，所以除了logits，还返回各field的embedding，方便搭建D
    return logits, fields_embeddings
```

DNN预测部分

再次声明，将criteo中原来的39列，拆分成2个field，**并不是为了提升预测性能，只是为了模拟实际场景**。导致的后果就是，Deep侧第一层的输入由原来的[batch_size, 39*embed_dim]变成了[batch_size, 2*embed_dim]，使Deep侧交叉不足。

尽管在criteo数据集上，deep侧的输入由**feature_size***embed_dim变成了**field_size***embed_dim，限制了交叉能力。但是，**在实际系统中，field_size已经是成千上万了**，而每个field下的feature又是成千上万，而且，因为embedding是稠密的，没有稀疏优化的可能性。因此，**在接入deep侧之前，每个field内部先做一层pooling**，将deep侧输入由feature_size*embed_dim压缩成field_size*embed_dim，**对于大规模机器学习，是十分必要的**。

DNN的代码如下所示。可以看到，其中没有加入L1/L2 regularization，这是模仿TensorFlow自带的Wide & Deep实现DNNLinearCombinedClassifier的写法。L1/L2正则将通过设置optimizer的参数来实现。

```
def output_logits_from_dnn(fields_embeddings, params, is_training):
    dropout_rate = params['dropout_rate']
    do_batch_norm = params['batch_norm']

    X = tf.concat(fields_embeddings, axis=1)
    tf.logging.info("initial input to DNN, shape={}".format(X.shape))

    for idx, n_units in enumerate(params['hidden_units'], start=1):
        X = tf.layers.dense(X, units=n_units, activation=tf.nn.relu)
        tf.logging.info("layer[{}] output shape={}".format(idx, X.shape))

        X = tf.layers.dropout(inputs=X, rate=dropout_rate, training=is_trainir
        if is_training:
            tf.logging.info("layer[{}] dropout {}".format(idx, dropout_rate))

        if do_batch_norm:
            # BatchNormalization的调用、参数，是从DNNLinearCombinedClassifier源码
            batch_norm_layer = normalization.BatchNormalization(momentum=0.999
                                                                name='batchnor
            X = batch_norm_layer(X, training=is_training)

            if is_training:
                tf.logging.info("layer[{}] batch-normalize".format(idx))

```

```
    # connect to final logits, [batch_size,1]
    return tf.layers.dense(X, units=1, use_bias=True, activation=None)
```

model_fn

前面的代码完成了“线性预测”+“二次交叉预测”+“深度预测”，则model_fn的实现就非常简单了，只不过将三个部分得到的logits相加就可以了。

```
def model_fn(features, labels, mode, params):
    for featname, featvalues in features.items():
        if not isinstance(featvalues, tf.SparseTensor):
            raise TypeError("feature[{}] isn't SparseTensor".format(featname))

    # ============= build the graph
    embedding_table = build_embedding_table(params)

    linear_logits = output_logits_from_linear(features, embedding_table, paran

    bi_interact_logits, fields_embeddings = output_logits_from_bi_interaction(

    dnn_logits = output_logits_from_dnn(fields_embeddings, params, (mode == tf

    general_bias = tf.get_variable(name='general_bias', shape=[1], initializer

    logits = linear_logits + bi_interact_logits + dnn_logits
    logits = tf.nn.bias_add(logits, general_bias)  # bias_add，获取broadcastin

    # reshape [batch_size,1] to [batch_size], to match the shape of 'labels'
    logits = tf.reshape(logits, shape=[-1])

    probabilities = tf.sigmoid(logits)

    # ============= predict spec
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(
            mode=mode,
            predictions={'probabilities': probabilities})

```

```
    # ============= evaluate spec
    # STUPID TENSORFLOW CANNOT AUTO-CAST THE LABELS FOR ME
    loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=logit

    eval_metric_ops = {'auc': tf.metrics.auc(labels, probabilities)}
    if mode == tf.estimator.ModeKeys.EVAL:
        return tf.estimator.EstimatorSpec(
            mode=mode,
            loss=loss,
            eval_metric_ops=eval_metric_ops)

    # ============= train spec
    assert mode == tf.estimator.ModeKeys.TRAIN
    train_op = params['optimizer'].minimize(loss, global_step=tf.train.get_glo
    return tf.estimator.EstimatorSpec(mode,
                                      loss=loss,
                                      train_op=train_op,
                                      eval_metric_ops=eval_metric_ops)
```

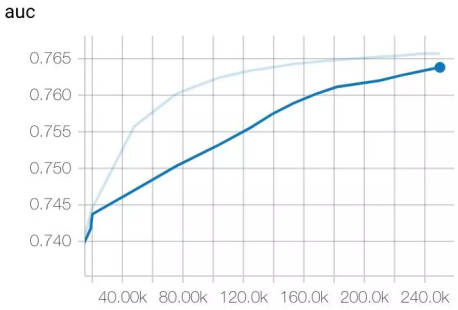
训练与评估

完成了model_fn之后，拜TensorFlow Estimator框架所赐，训练与评估变得非常简单，设定超参数之后（注意在指定optimizer时设置了L1/L2的正则权重），调用tf.estimator.train_and_evaluate即可。

```
def get_hparams():
    vocab_sizes = {
        'numeric': 13,
        # there are totally 14738 categorical tags occur >= 200
        # since 0 is reserved for OOV, so total vocab_size=14739
        'categorical': 14739
    }

    optimizer = tf.train.ProximalAdagradOptimizer(
        learning_rate=0.01,
        l1_regularization_strength=0.001,
        l2_regularization_strength=0.001)

```

```
    return {
        'embed_dim': 128,
        'vocab_sizes': vocab_sizes,
        # 在这个case中，没有多个field共享同一个vocab的情况，而且field_name和vocab_n
        'field_vocab_mapping': {'numeric': 'numeric', 'categorical': 'categori
        'dropout_rate': 0.3,
        'batch_norm': False,
        'hidden_units': [64, 32],
        'optimizer': optimizer
    }


if __name__ == "__main__":
    tf.logging.set_verbosity(tf.logging.INFO)
    tf.set_random_seed(999)

    hparams = get_hparams()
    deepfm = tf.estimator.Estimator(model_fn=model_fn,
                                    model_dir='models/criteo',
                                    params=hparams)

    train_spec = tf.estimator.TrainSpec(input_fn=lambda: input_fn(data_file='c
                                                                  n_repeat=10,
                                                                  batch_size=1
                                                                  batches_per_

    eval_spec = tf.estimator.EvalSpec(input_fn=lambda: input_fn(data_file='dat
                                                                n_repeat=1,
                                                                batch_size=128
                                                                batches_per_sh

    tf.estimator.train_and_evaluate(deepfm, train_spec, eval_spec)
```

测试集上的部分结果所下所示，测试集上的AUC在0.765左右，没有Kaggle solution上0.8+的AUC高。正如前文所说的，将原来criteo数据集中的39列拆分成2个field，只是为了演示“一列多值”、“稀疏”的DeepFM实现，但限制了Deep侧的交叉能力，**对最终模型的性能造成一定负面影响**。不过，仍然证明，文中展示的DeepFM实现是正确的。



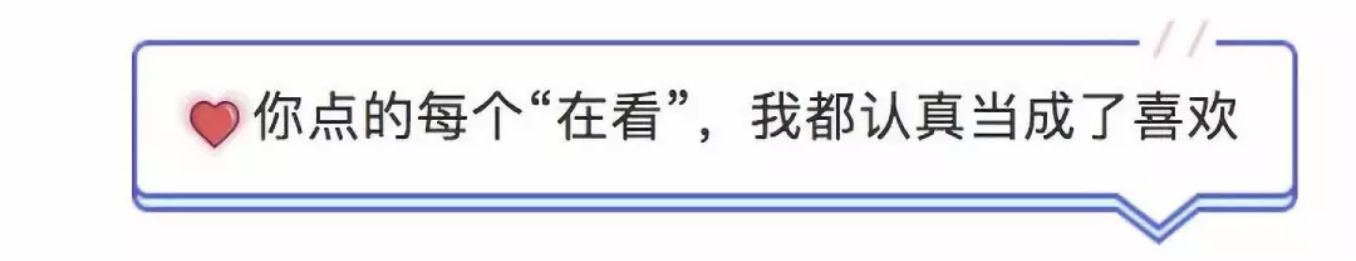
小结

本文展示了我写的一套基于TensorFlow的DeepFM的实现。重点阐述了“**一列多值**”、“**稀疏**”、“**权重共享**”在实际推荐系统中的重要性，和我是如何在DeepFM中实现以上需求的。欢迎各位看官指正。

如果有耐心读到这里的话，就关注一下公众号吧：）

参考

[1] https://github.com/ChenglongChen/tensorflow-DeepFM
[2] https://zhuanlan.zhihu.com/p/33699909
[3] https://github.com/stasi009/Recommend-Estimators/blob/master/deepfm.py



阅读原文