[阿里DIN]从论文源码学习 之 embedding_lookup

原创 罗西的思考 罗西的思考 10月25日

[阿里DIN]从论文源码学习 之 embedding_lookup

0x00 摘要

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

本系列文章解读论文以及源码，顺便梳理一些深度学习相关概念和TensorFlow的实现。

本文通过DIN源码 https://github.com/mouna99/dien 分析，来深入展开看看embedding层原理 以及 embedding_lookup如何使用。

0x01 DIN代码

1.1 Embedding概念

我们首先简要提一下Embedding概念及作用。

`Embedding` 译为“嵌入”，被翻译为“向量化”。主要作用：**将稀疏向量转化为稠密向量，便于上层神经网络的处理。**例如，做一个推荐用户看视频的推荐系统，该模型的输入可能是用户属性（看视频，yoghurt搜索词，用户年龄，性别等等）的 `Embedding` 向量，模型的输出是多分类的 `softmax` 层，预测的是用户看了哪个视频。

定义：用一个低维稠密的向量“表示”一个对象。 对象可以是一个词，一个商品，一部电影等等。“表示”：意味着 `Embedding` 向量能够表达相应对象的某些特征、向量之间的距离，可以反应对象之间的相似性。

`Embedding` 对深度学习推荐系统的重要性：

- 在输入层和全连接层之间使用 `Embedding` 层将高维稀疏特征向量转换成低维稠密特征向量；
- 可以引入任何信息进行编码，本身包含大量有价值的信息；
- 通过计算用户和物品的 Embedding 相似度，Embedding 可以直接作为推荐系统或计算广告系统的召回层或者召回方法之一；

1.2 在DIN中的使用

下面是一个缩减版的代码，可以看到，DIN是用 `self.mid_batch_ph` 作为id，在 `self.uid_embeddings_var` 中查找变量。

在DIN中，我们只有这一处初始化 embeddings 的地方，没有找到迭代更新的代码，这会给初学者带来一些困扰。

```
class Model(object):
    def __init__(self, n_uid, n_mid, n_cat, EMBEDDING_DIM, HIDDEN_SIZE, ATTENTION_SIZE, use_negsar
        with tf.name_scope('Inputs'):
            self.mid_his_batch_ph = tf.placeholder(tf.int32, [None, None], name='mid_his_batch_ph
            self.cat_his_batch_ph = tf.placeholder(tf.int32, [None, None], name='cat_his_batch_ph
            self.uid_batch_ph = tf.placeholder(tf.int32, [None, ], name='uid_batch_ph')
            self.mid_batch_ph = tf.placeholder(tf.int32, [None, ], name='mid_batch_ph')
            self.cat_batch_ph = tf.placeholder(tf.int32, [None, ], name='cat_batch_ph')
```

```
# Embedding layer
with tf.name_scope('Embedding_layer'):
    self.uid_embeddings_var = tf.get_variable("uid_embedding_var", [n_uid, EMBEDDING_DIM]
    self.uid_batch_embedded = tf.nn.embedding_lookup(self.uid_embeddings_var, self.uid_bat

    self.mid_embeddings_var = tf.get_variable("mid_embedding_var", [n_mid, EMBEDDING_DIM]
    self.mid_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, self.mid_bat
    self.mid_his_batch_embedded = tf.nn.embedding_lookup(self.mid_embeddings_var, self.mi

    self.cat_embeddings_var = tf.get_variable("cat_embedding_var", [n_cat, EMBEDDING_DIM]
    self.cat_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat_bat
    self.cat_his_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat
```

1.3 问题

于是我们遇到几个问题：

- embedding层在这里起到什么作用？
- embedding_lookup究竟用来做什么？
- 如何更新mid_embeddings_var这样的embedding层？

下面就让我们一一研究。

0x02 相关概念

2.1 one-hot编码

one-hot编码是保证每个样本中的单个特征只有1位处于状态1，其他的都是0。

具体编码举例如下，把语料库中，杭州、上海、宁波、北京每个都对应一个向量，向量中只有一个值为1，其余都为0，我们得到如下。

```
杭州 [0,0,0,0,0,0,0,1,0,……, 0,0,0,0,0,0,0]
上海 [0,0,0,0,1,0,0,0,0,……, 0,0,0,0,0,0,0]
宁波 [0,0,0,1,0,0,0,0,0,……, 0,0,0,0,0,0,0]
北京 [0,0,0,0,0,0,0,0,0,……, 1,0,0,0,0,0,0]
```

这就是独热编码。

优势：计算方便快捷、表达能力强。

缺点：

- 每一维包含信息量太少；
- 维度会随着城市数量的增加而增加，可能造成维度爆炸，计算复杂度过高。过于稀疏时，过度占用资源。比如统计全球城市，那么对应矩阵维度就太大了。

2.2 转换

因为独热编码有使用上的困难，所以在实践中，人们会对其进行转换，我们大致讲解如下：

比如用one-hot编码来表示4个梁山好汉。

```
李逵   [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
刘唐   [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
武松   [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
鲁智深 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

这样编码的**优势**是：所有梁山好汉，都能在一个一维的数组里用 0 1 表示出来。不同的好汉绝对不一样，一点重复都没有，表达本征的能力极强。

缺点是：因为其完全独立，表达关联特征的能力几乎为0。你从这个稀疏矩阵无法看出来这四人中有任何联系，因为每个人都只有一个1，而且这个1的位置彼此完全不同。

但实际上，这几个好汉都是有内在联系的。

- 4个好汉武力值都不错。
- 武松和鲁智深两个人都是出家人，都做过官。鲁智深是提辖，官阶大些。武松是都头，官阶略小。
- 李逵和刘唐两个人都是二货。

所以我们构建如下矩阵：

```
          二  出  官   武
          货  家  阶   力
李逵     [1   0   0   0.5]
刘唐     [1   0   0   0.4]
武松     [0   1   0.5 0.8]
鲁智深   [0   1   0.75 0.8]
```

由此我们将四位好汉同 "二货"，"出家"，"官阶"，"武力" 这几个特征关联起来，我们可以认为：

- 李逵 = 1.0 二货 + 0 出家 + 0 官阶 + 0.5 武力
- 刘唐 = 1.0 二货 + 0 出家 + 0 官阶 + 0.4 武力
- 武松 = 0 二货 + 1 出家 + 0.5 官阶 + 0.8 武力
- 鲁智深 = 0 二货 + 1 出家 + 0.75 官阶 + 0.8 武力

于是乎，我们把好汉的one-hot编码，从稀疏态变成了密集态，并且让相互独立向量变成了有内在联系的关系向量。这就得倒了 Embedding层。

2.3 Embedding层

2.3.1 意义

Embedding的意义是对于高维、稀疏的id类特征，通过将单个id（可能是某个词的id，也可能是某个商品的id）映射成一个稠密向量，**变id特征的“精确匹配”为embedding向量的“模糊查找”，从而降低了特征的维度和计算复杂度，提升算法的扩展能力**。

Embedding最重要的属性是：**越“相似”的实体，Embedding之间的距离越小。**以word2vec模型为例，如果两个词的上下文几乎相同，就意味着它们的输出值几乎相同，在模型收敛的前提下，两个词在Embedding层的输出值一定非常相近。在推荐系统里，可以计算实体间的余弦相似度，召回相似度高的商品作为备选推荐商品，这是Embedding内在属性的一种简单的应用。

Embedding层把我们的稀疏矩阵，通过一些线性变换（比如用全连接层进行转换，也称为查表操作），变成了一个密集矩阵，这个密集矩阵用了N（例子中N=4）个特征来表征所有的好汉。在这个密集矩阵中，表象上代表着密集矩阵跟单个好汉的一一对应关系，实际上还蕴含了大量的好汉与好汉之间的内在关系（如：我们得出的李逵跟刘唐的关系）。它们之间的关系，用嵌入层学习来的参数进行表征。这个从稀疏矩阵到密集矩阵的过程，叫做embedding，很多人也把它叫做查表，因为它们之间也是一个一一映射的关系。

这种映射关系在反向传播的过程中一直在更新。因此能在多次epoch后，使得这个关系变成相对成熟，即：正确的表达整个语义以及各个语句之间的关系。这个成熟的关系，就是embedding层的所有权重参数。

Embedding是NPL领域最重要的发明之一，他把独立的向量一下子就关联起来了。这就相当于什么呢，相当于你是你爸的儿子，你爸是A的同事，B是A的儿子，似乎跟你是八竿子才打得着的关系。结果你一看B，是你的同桌。Embedding层就是用来发现这个秘密的武器。

Embedding最大的劣势是无法解释每个维度的含义，这也是复杂机器学习模型的通病。

2.3.2 常规作用

Embedding除了把独立向量联系起来之外，还有两个作用：降维，升维。

embedding层 **降维**的原理就是矩阵乘法。

比如一个 1 x 4 的矩阵，乘以一个 4 x 3 的矩阵，得倒一个 1 x 3 的矩阵。4 x 3 的矩阵缩小了 1 / 4。

$$[0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = [7 \quad 8 \quad 9]$$

假如我们有一个100W X 10W的矩阵，用它乘上一个10W X 20的矩阵，我们可以把它降到100W X 20，瞬间量级降了。

升维可以理解为：前面有一幅图画，你离远了看不清楚，离近了看就可以看清楚细节。当对低维的数据进行升维时，可能把一些其他特征给放大了，或者把笼统的特征给分开了。同时这个embedding是一直在学习在优化的，就使得整个拉近拉远的过程慢慢形成一个良好的观察点。

2.3.3 如何生成

生成Embedding的方法可以归类为三种，分别是**矩阵分解，无监督建模和有监督建模**。

矩阵分解

矩阵分解是将两种实体间的关系矩阵分解为两个Embedding矩阵，得到每一种实体的Embedding，比如在推荐系统里，我们已知用户与商品的共现矩阵，通过矩阵分解可以得到每个用户的Embedding和每个商品的Embedding。

无监督建模

无监督建模是生成Embedding的常用方法，按组织方式可以将数据分为序列和图两类，针对序列数据生成Embedding常采用word2vec或类似算法（item2vec, doc2vec等），针对图数据生成Embedding的算法称为Graph Embedding，这类算法包括deepwalk、node2vec、struc2vec等，它们大多采用随机游走方式生成序列，底层同样也是word2vec算法。

有监督建模

有监督建模也可以用于生成Embedding，主要分为两类，一类是因子分解机及其衍生算法，包括FM、FFM、DeepFM等，另一类是图卷积算法，包括GCN、GraphSAGE、GAT等，这些模型中都包含Embedding层，在建模有监督问题时，每个实体在Embedding层的输出向量可以作为这个实体的Embedding使用。

2.4 Embedding与深度学习推荐系统的结合

2.4.1 重要性

为什么说Embedding技术对于深度学习如此重要，甚至可以说是深度学习的“基本核心操作”呢？原因主要有以下：

- 在深度学习网络中作为Embedding层，完成从高维稀疏特征向量到低维稠密特征向量的转换（比如Wide&Deep、DIN等模型）。推荐场景中大量使用One-hot编码对类别、id型特征进行编码，导致样本特征向量极度稀疏，而深度学习的结构特点使其不利于稀疏特征向量的处理，因此几乎所有的深度学习推荐模型都会由Embedding层负责将高维稀疏特征向量转换成稠密低维特征向量。
- 作为预训练的Embedding特征向量，与其他特征向量连接后，一同输入深度学习网络进行训练（比如FNN模型）。Embedding本身就是极其重要的特征向量。相比传统方法产生的特征向量，Embedding的表达能力更强，特别是Graph Embedding技术被提出后，Embedding几乎可以引入任何信息进行编码，使其本身就包含大量有价值的信息。在此基础上，Embedding向量往往会与其他推荐系统特征连接后一同输入后续深度学习网络进行训练。
- 通过计算用户和物品的Embedding相似度，Embedding可以直接作为推荐系统的召回层或者召回策略之一（比如Youtube推荐模型等）。Embedding对物品、用户相似度的计算是常用的推荐系统召回层技术。在局部敏感哈希（Locality-Sensitive Hashing）等快速最近邻搜索技术应用于推荐系统后，Embedding更适用于对海量备选物品进行快速“筛选”，过滤出几百到几千量级的物品交由深度学习网络进行“精排”。

2.4.2 预训练方法

由于 `Embedding` 预训练开销巨大，一般让 `Embedding` 的预训练往往独立于深度学习网络进行。

`Embedding` 层进行低频训练就可以了，上层神经网络为抓住最新的数据整体趋势，需要进行高频训练、实时训练。

更彻底的 `Embedding` 训练方法就是固定 `Embedding` 层权重，仅更新上层神经网络权重。

模型部署，只需要将用户 `Embedding` 和物品 `Embedding` 存储到线上内存数据库，通过内积运算再排列得到物品的排列，再取Top N 的物品，即可得到召回的候选集合，这就是利用 `Embedding` 作为召回层的过程。没必要部署整个深度神经网络来完成从原始特征向量到最终输出的预测过程。

0x03 embedding_lookup

embedding_lookup 函数在 DIN 实现了完成高维稀疏特征向量到低维稠密特征向量的转换。其接收的是类别特征的 `one-hot` 向量，转换的目标是低维的 `Embedding` 向量。本质上就求解一个m × n 维的权重矩阵的过程，其列向量就是相应维度的 `one-hot` 特征的 `Embedding` 向量。

3.1 函数说明

在NLP领域中，通常都会先将文字转换成普通整数编码，然后再用embedding层进行可更新向量编码。Tensorflow提供给了embedding_lookup函数来进行转换。比如从one_hot到矩阵编码的转换过程需要在embedding进行查找：

```
one_hot * embedding_weights = embedding_code
```

TensorFlow 的 embedding_lookup(params, ids) 函数的**目的是按照ids从params这个矩阵中拿向量（行）**，所以ids就是这个矩阵索引（行号），需要int类型。即按照ids顺序返回params中的第ids行。比如说，ids=[1,3,2],就是返回params中第1,3,2行。返回结果为由params的1,3,2行组成的tensor。

```
embedding_lookup(
    params,   # embedding_params 对应的转换向量
    ids,      # inputs_ids，标记着要查询的id
    partition_strategy='mod',   #分割方式
    name=None,
    validate_indices=True, # deprecated
    max_norm=None
)
```

参数和返回值如下:

- **params**: 由一个tensor或者多个tensor组成的列表(多个tensor组成时，每个tensor除了第一个维度其他维度需相等)。
- **ids**: 一个整型的tensor，ids的每个元素代表要在params中取的每个元素的第0维的逻辑index。
- **partition_strategy**: 逻辑index是由partition_strategy指定，partition_strategy用来设定ids的切分方式，目前有两种切分方式'div'和'mod'，默认是'mod'。
- **返回值**: 是一个dense tensor，返回的shape为shape(ids)+shape(params)[1:]。

3.2 函数本质

3.2.1 全连接层

embedding_lookup是一种特殊的全连接层的实现方法，其针对 输入是超高维 one hot向量的情况。

神经网络处理不了onehot编码。embedding_lookup虽然是随机化地映射成向量，看起来信息量相同，但其实却更加超平面可分。

问题本质只是做一次常规的线性变换而已，Z = WX + b。由于输入是One-Hot Encoding 的原因，WX 的矩阵乘法看起来就像是取了Weights矩阵中对应的一列，看起来就像是在查表。等于说变相的进行了一次矩阵相乘运算，其实就是一次线性变换。

embedding_lookup不是简单的查表，id对应的向量是可以训练的，训练参数个数应该是category num * embedding size，也就是说lookup是一个**特殊的“全连接层”**。

3.2.2 映射向量

一般做自然语言相关的。需要把每个词都映射成向量，这个向量可以是word2vec预训练好的，也可以是在网络里训练的。在网络里需要先把词的id转换成对应的向量，这个函数就是做这件事的。

假设embedding权重矩阵是一个[vocab_size, embed_size]的稠密矩阵W，vocab_size是需要embed的所有item的个数（比如：所有词的个数，所有商品的个数），embed_size是映射后的向量长度。

所谓embedding_lookup(W, id1)，可以想像成一个只在id1位为1的[1, vocab_size]的one_hot向量，与[vocab_size, embed_size]的W矩阵相乘，结果是一个[1, embed_size]的向量，它就是id1对应的embedding向量，实际上就是W矩阵的第id1行。

但是，以上过程只是前代，因为**W一般是随机初始化的，是待优化的变量**。因此，embedding_lookup除了要完成以上矩阵相乘的过程（实现成“抽取id对应的行”），还要完成自动求导，以实现对W的更新。

embedding_lookup一般在NLP中用得比较多，将一个[batchsize, sequence_len]的输入，映射成[batchsize, sequence_len, embed_size]的矩阵。而在**推荐/搜索领域，我们往往需要先embedding, 再将embedding后的多个向量合并成一个向量（即pooling过程）**。比如，用户过去一周用过3次微信，1次支付宝，那我们将用户过去一周的app使用习惯表示成：**用户app使用习惯向量 = 3 * 微信向量 + 1 * 支付宝向量**

3.3 函数示例

3.3.1 示例

示例代码如下：

```
import numpy as np
import tensorflow as tf

sess = tf.InteractiveSession()

embedding = tf.Variable(np.identity(6, dtype=np.int32))
input_ids = tf.placeholder(dtype=tf.int32, shape=[None])
input_embedding = tf.nn.embedding_lookup(embedding, input_ids)

sess.run(tf.global_variables_initializer())
print("====== the embedding ====== ")
print(sess.run(embedding) )
print("====== the input_embedding ====== ")
print(sess.run(input_embedding, feed_dict={input_ids: [4, 0, 2]}))
```

3.3.2 输出

输出如下

```
====== the embedding ======
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
====== the input_embedding ======
[[0 0 0 0 1 0]
 [1 0 0 0 0 0]
 [0 0 1 0 0 0]]
```

3.3.3 解释

从以上可以看出：

embedding将变量表现成了one-hot形式，简单来说是创建了一个embedding词典；

```
====== the embedding ======
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

而 `input_embedding = tf.nn.embedding_lookup(embedding, input_ids)` 就是把input_ids中给出的tensor表现成embedding中的形式；

简单来说是通过输入的input_ids查询上部的字典得到embedding后的值。而字典是可以由用户随意创建的，例中给出的是一个one-hot字典，还可以自由创建其他字典，例如使用正态分布或均匀分布产生（0，1）的随机数创建任意维度的embedding字典。

```
====== the input_embedding ======
[[0 0 0 0 1 0]
 [1 0 0 0 0 0]
 [0 0 1 0 0 0]]
```

3.4 DIN应用

回到DIN代码，从注释中可以看出DIN在此构建user，item的embedding lookup table，将输入数据转换为对应的embedding，就是把稀疏特征转换为稠密特征。具体就是用各种变量作为id，在对应的embeddings_var中查找变量。比如用 `self.mid_batch_ph` 作为id，在 `self.uid_embeddings_var` 中查找变量。

```
class Model(object):
    def __init__(self, n_uid, n_mid, n_cat, EMBEDDING_DIM, HIDDEN_SIZE, ATTENTION_SIZE, use_negsar
        with tf.name_scope('Inputs'):
            self.mid_his_batch_ph = tf.placeholder(tf.int32, [None, None], name='mid_his_batch_ph
            self.cat_his_batch_ph = tf.placeholder(tf.int32, [None, None], name='cat_his_batch_ph
            self.uid_batch_ph = tf.placeholder(tf.int32, [None, ], name='uid_batch_ph')
            self.mid_batch_ph = tf.placeholder(tf.int32, [None, ], name='mid_batch_ph')
            self.cat_batch_ph = tf.placeholder(tf.int32, [None, ], name='cat_batch_ph')
```

```
        # Embedding layer
        with tf.name_scope('Embedding_layer'):
            # shape: [U, H/2], user_id的embedding weight. U是user_id的hash bucket size，即user cou
            self.uid_embeddings_var = tf.get_variable("uid_embedding_var", [n_uid, EMBEDDING_DIM]
            # 从uid embedding weight 中取出 uid embedding vector
            self.uid_batch_embedded = tf.nn.embedding_lookup(self.uid_embeddings_var, self.uid_bat

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

            # shape: [C, H/2], cate_id的embedding weight. C是cat_id的hash bucket size
            self.cat_embeddings_var = tf.get_variable("cat_embedding_var", [n_cat, EMBEDDING_DIM]
            # 从 cid embedding weight 中取出 cid history embedding vector，是正样本
            # 比如cat_embeddings_var 是(1601, 18)，cat_batch_ph 是(?,)，则cat_batch_embedded 就是 (
            self.cat_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat_bat
            # 从 cid embedding weight 中取出 cid embedding vector，是正样本
            self.cat_his_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_var, self.cat
            # 从 cid embedding weight 中取出 cid history embedding vector，是负样本
            if self.use_negsampling:
                self.noclk_cat_his_batch_embedded = tf.nn.embedding_lookup(self.cat_embeddings_va
```

至此我们解决了关于embedding的前两个问题：

- embedding层起到什么作用？
- embedding_lookup究竟用来做什么?

对于第三个问题：如何更新mid_embeddings_var这样的embedding层？我们将在下文进行讲解，敬请期待。

0xFF 参考

tf.nn.embedding_lookup中关于partition_strategy参数详解

深度学习中 Embedding层两大作用的个人理解

深入理解 Embedding层的本质

tf.nn.embedding_lookup函数原理？

Embedding原理和Tensorflow-tf.nn.embedding_lookup()

求通俗讲解下tensorflow的embedding_lookup接口的意思？

embedding_lookup的学习笔记

tf.nn.embedding_lookup

Logit究竟是个啥？——离散选择模型之三

用NumPy手工打造 Wide & Deep

详解 Wide & Deep 结构背后的动机

见微知著，你真的搞懂Google的Wide&Deep模型了吗？

Embedding技术在深度学习推荐系统中的应用

深度学习推荐系统 | Embedding，从哪里来，到哪里去

Embedding技术在深度学习推荐系统中的应用

深度学习推荐系统中各类流行的Embedding方法（上）

深度学习推荐系统中各类流行的Embedding方法（下）

阅读原文

喜欢此内容的人还喜欢

[从源码学设计]蚂蚁金服SOFARegistry之推拉模型
罗西的思考

作恶的不是资本，是刚吃了几天饱饭的傻逼
地瓜园