

第3章 推荐系统中的Embedding

第3章 推荐系统中的Embedding

上一章讲到，高维、稀疏的类别型特征是推荐系统中的一等公民，接下来的问题是，我们如何将这些类别型特征喂入推荐模型？

在深度学习大行其道的今天，最常见的类别特征接入方式，是将稀疏的类别特征映射成一个稠密向量，即所谓的Embedding。本章将深度聚焦于Embedding这个话题，帮助读者了解Embedding技术的前世今生，知其然，也知其所以然。

本章将分以下几个部分展开：

- 第3.1节，先从传统推荐算法开始讲起，逐步过渡到Embedding。读完这一节，读者将明白Embedding技术并非凭空出现的，而是为了应对推荐算法的两大永恒主题之一的“扩展性”，应运而生。
- 第3.2节，将讨论不同推荐算法在Embedding层的两大技术路线：“共享”与“独占”。
- Embedding的引入极大提升了推荐模型的扩展性，但是也增加了训练难度。为了解决这一难度，基于Parameter Server的分布式训练范式，应运而生，并且已经成为各大厂推荐系统的标配。第3.3节，将为读者介绍Parameter Server，并通过代码揭示它的具体工作原理。

3.1 无中生有：推荐算法中的Embedding

如同在前言中所说的那样，任何一门技术，要想获得互联网打工人的青睐，都必须能够实实在在解决我们面临的问题。那推荐算法面临的经典问题，无非两个，“记忆”与“扩展”。

3.1.1 传统推荐算法：博闻强记

我们希望推荐系统记住什么？能够记住的肯定是那些常见、高频的模式。举个简单的例子：

- 到了春节，来了中国客户，电商网站给他推饺子，大概率能够购买
- 到了感恩节，来了美国客户，电商网站给他推火鸡，大概率也能购买 为什么？因为<春节，中国人，饺子>的模式、<感恩节、美国人、火鸡>的模式在训练样本中出现得太多太多了，推荐系统只需要记得住，下次遇到同样的场景，“照方抓药”，就能“药到病除”。

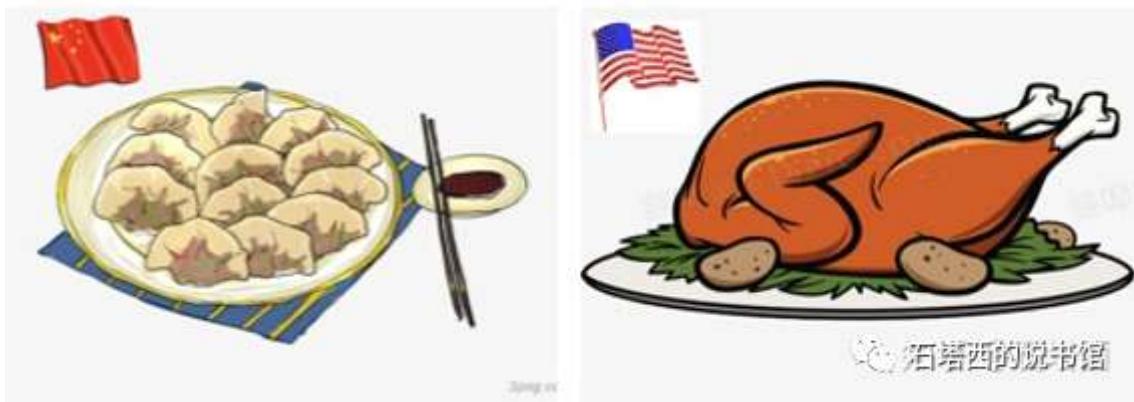


图 3-1 常见消费模式

那怎么才能让模型记住？上“评分卡”。“评分卡”是金融风控中的一种常用手段，说白了，其实就是Logistic Regression（LR）模型。推荐系统中的“评分卡”，如果形象地画出来，如图3-2所示。

特征类别	特征值	成交可能性
客户国籍	中国	2
	美国	3

时机	工作日	0.5
	春节	10
	感恩节	10

商品	饺子	16
	火鸡	13

	坦克	-100
国籍 X 商品	中国人, 饺子	100
	中国人, 鲑鱼	-100000

时机 X 国籍 X 商品	春节, 中国人, 饺子	500
	感恩节, 美国人, 火鸡	500

.....	石塔西的说书馆

图 3-2 推荐系统的“评分卡”

- 一个特征（比如：中国、美国），或特征组合（比如：`<春节、中国人、饺子>`）占据“推荐评分卡”中的一项。可想而知，一个工业级的推荐LR的评分卡里面，条目会有上亿项。
- 每项特征或特征组合，都对应一个分数（SCORE）。

评分卡中的分数是由LR模型学习出来的，有正有负，代表对最终目标（比如成交）的贡献。

- 比如 `SCORE(<春节, 中国人, 饺子>)=5`，代表这种组合非常容易成交。
- 反之 `SCORE(<中国人、鲑鱼>)=-100`，代表这个组合极不容易成交。简单理解，可以认为在正样本中出现越多的特征或特征组合，得分越高；反之在负样本中出现越多的特征或特征组合，得分越低

LR模型的最终得分是一条样本能够命中的评分卡中所有条目的得分总和。比如，春节期间一个中国客户访问购物网站，LR模型预测他对一款“鲱鱼馅水饺”的购买欲望= $SCORE(<\text{春节、中国人、饺子}>) + SCORE(<\text{中国人, 鲱鱼}>)$ = $500 - 100000 = -99500$ ，也就是铁定不会购买。因此，推荐系统也就不会向该用户展示该款商品。

LR（评分卡）模型的特点：

- LR的特点就是强于记忆，只要评分卡足够大（比如几千亿项），它能够记住历史上的发生过的所有模式（i.e., 特征及其组合）。
- 所有的模式，都依赖人工输入。所以在推荐模型的LR时代，特征工程既需要创意，同时也是一项体力活。
- LR本身并不能够发掘出新模式，它只负责评估各模式的重要性。这个重要性是通过大量的历史数据拟合得到的。
- LR不发掘新模式，反之它能够通过“正则”（Regularization），剔除一些得分较低的罕见模式（比如<中国人，于谦在非洲吃的同款恩希玛>），既避免过拟合，又降低了评分卡的规模。

LR（评分卡）模型，强于记忆，但是弱于扩展。还举刚才的例子，中国顾客来了推饺子，美国客户来了推火鸡，效果都不错，毕竟LR记性好。但是，当一个中国客户来了，你的推荐系统会给他推荐一只火鸡吗？如果你的推荐系统只有LR，只有记忆功能，答案是：不会。因为 <中国人, 火鸡> 毕竟属于小众模式，在历史样本罕有出现，LR的L1正则直接将打分置为0，从而被从评分卡中剔除。

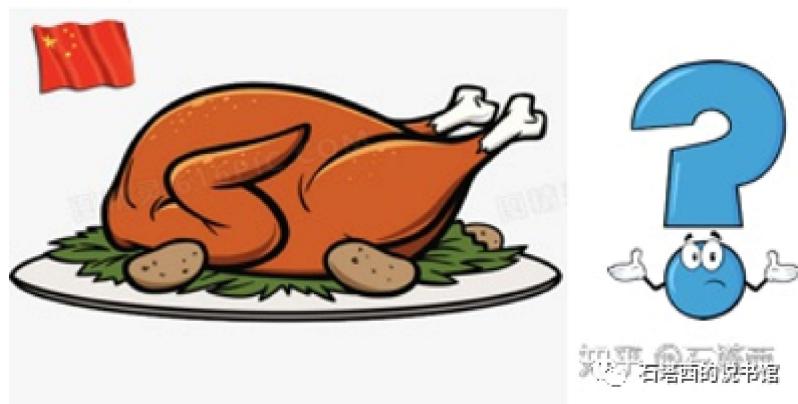


图 3-3 小众模式

不要小看这个问题，它关乎到企业的生死，也就关系到你老板和你的腰包：

- 记住的肯定是那些常见、高频、大众的模式，能够处理80%用户的80%的日常需求，但是对小众用户的小众需求呢（比如：某些中国人喜欢开洋荤的需求、于老师的超级粉丝希望和偶像体验相同美食的需求）？只凭好记性是无能为力的，因为缺乏历史样本的支持，换句话说，推荐的个性化太弱。
- 另一个问题，大众的需求，你能记住，别家电商也能记住。所以你和你的同行，只能在“满足大众需求”的这一片红海里相互厮杀。套用如今最时髦的词，“内卷”。

3.1.2 推荐算法的刚需：扩展性

综上所述，为了避开“大众推荐”这一片内卷严重的红海，而拥抱“个性化精准推荐”的蓝海，推荐算法不能只满足于记住训练数据中频繁出现的、常见、高频的模式，而必须能够自动挖掘出训练数据中罕见的、低频、长尾的模式。这也要求推荐模型必须具备“扩展性”，翻译成白话，也就是能够“举一反三”。

如何让模型扩展？看似神秘，其实就是将细粒度的概念，拆解成一系列粗粒度的特征，从此以后“**看山非山、看水非水**”。还举刚才饺子、火鸡的例子。在之前讲记忆的时候，饺子、火鸡都是独立的概念，看起来没什么相似性。但是，如果我们根据业务知识，将概念拆解成特征向量，如图3-4所示。

- 两个特征向量的第一位表示“是否是食物”，从这个角度来看，饺子、火鸡非常相似
- 两个特征向量的第二位表示“是否和节日相关”，从这个角度来看，饺子、火鸡也非常相似。

具备相似性

商品	是否是食物	是否和节日相关	能够素食	价格	产地
	1	1	1
	1	1	0

© 石塔西的说书馆

图 3-4 细粒度的概念拆解成粗粒度的特征

在训练LR模型的时候，每条样本除了将原来细粒度的概念 **<春节，中国人，饺子>** 和 **<感恩节，美国人，火鸡>** 作为特征，也将扩展后的 **<节日，和节日相关的食物>** 作为特征，一同喂入LR模型。这样训练后的“评分卡”不仅包含对 **<春节，中国人，饺子>** 和 **<感恩节，美国人，火鸡>** 的打分，也包含对粗粒度特征 **<节日，和节日相关的食物>** 的打分。而且因为 **<节日，和节日相关的食物>** 在训练数据中出现频繁，所以这一项必然在“评分卡”中占据一席之地，不会被正则机制过滤掉。

这样一来，当模型考虑是否应该在春节期间为一个中国顾客推荐火鸡时，虽然 **<春节，中国人，火鸡>** 这样的细粒度模式，因为太过小众，没能命中评分卡。但是扩展后的粗粒度模式 **<节日，和节日相关的食物>**，却命中了评分卡，并获得一个中等分数，从而有可能获得曝光的机会。相比于原来被L1正则优化掉，“一些中国人喜欢开洋荤”的小众模式也终于有了出头之日。

这样看来，只要我们喂入算法的，不是细粒度的概念，而是粗粒度的特征向量，即便是LR这样强记忆的算法，也能够具备扩展能力。但是，上述方法依赖于人工拆解，也就是所谓的“特征工程”，从而带来两方面的缺点：

- 工作量大，劳神费力。
- 比如饺子、火鸡这两个概念，还能不能从其他角度拆解，从而发现更多的相似性？这就要受到工程师的业务水平、理解能力、创意水平的制约。

既然人工拆解有困难、受局限，即能不能让算法自动将概念拆解成特征向量？如果你能够想到这一步，恭喜你，你一只脚已经迈入了深度学习的大门。你已经悟到了“道”，剩下的只是“技”而已。

3.1.3 深度学习的核心套路：无中生有的Embedding

我用“无中生有”来概括深度学习的思想精髓。所谓“无中生有”是指，你需要用到一个概念的特征 v （比如前面例子里的饺子、火鸡），或者一个函数 f （比如阿里Deep Interest Network中的“注意力”函数，后面第4.3.3节会详细讲解），但是却不知道如何定义它们。没关系，我们按以下三步走：

1. 先将 v 声明为特征向量，将 f 声明为一个小的神经网络，并随机初始化。
2. 然后让 v 和 f ，随着主目标一同被“随机梯度下降法”（Stochastic Gradient Descent，SGD）优化。
3. 当主目标被成功优化之后，我们也就获得了有意义的 v 和 f 。

这种“无中生有”的套路，好似“上帝说，要有光，于是便有了光”。以讹传讹，后来就变成了初学者口中“深度学习不需要特征工程”的神话，给了某些人“我只做深度学习，不做机器学习”的盲目自信。其实这种“将特征、函数转化为待优化变量”的思想，并不是深度学习发明的，早在用矩阵分解[1]进行推荐的“古代”就已经存在了，只不过那时候，它还不叫Embedding，而叫“隐向量”（Latent Vector）。

深度学习对于推荐算法的贡献与提升，其核心就在于Embedding。如前文所述，Embedding是一门自动将概念拆解为特征向量的技术，目标是提升推荐算法的扩展能力，从而能够自动挖掘那些低频、长尾、小众的模式，拥抱“个性化推荐”的“蓝海”。那么Embedding到底是如何提升扩展能力的？简单来说，Embedding将推荐算法从“精确匹配”转化为“模糊查找”，从而让模型能够“举一反三”。

比如在使用倒排索引的召回中，是无法给一个喜欢“科学”的用户，推出一篇带“科技”标签的文章的（如果不考虑近义词扩展的话），因为“科学”与“科技”是两个完全不同的词。但是经过Embedding，我们发现在向量空间中，表示“科学”与“科技”的两个向量，并不是正交的，而是有很小的夹角。设想一个极其简化的场景，用户就用“科学”向量来表示，文章用其标签的向量来表示。那么用“科学”向量在所有标签向量里做Top-K近邻搜索，一篇带“科技”标签的文章就能够被检索出来，有机会呈现在用户眼前，从而破除之前因为只能精确匹配“科学”标签给用户造成的“信息茧房”。

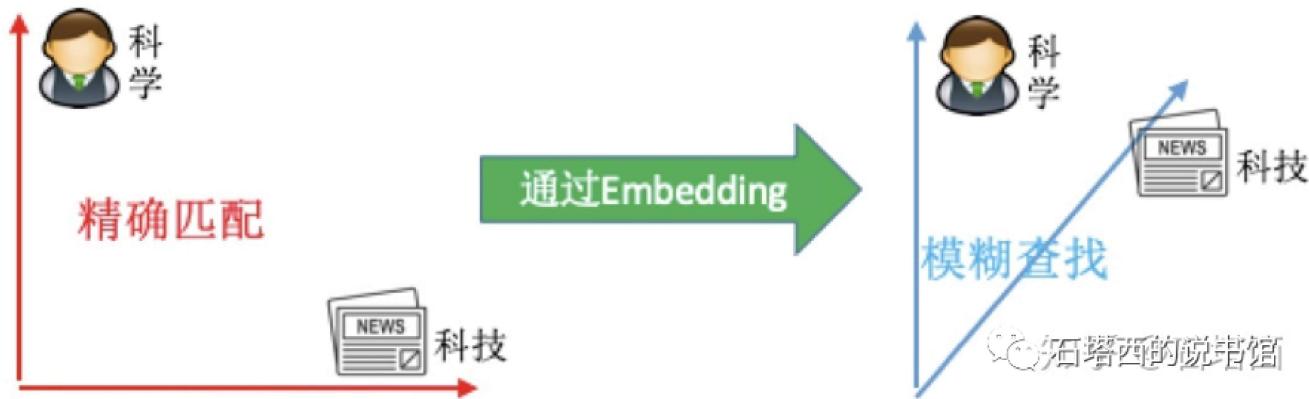


图 3-5 Embedding将“精确匹配”转变为“模糊查找”

再回到原来饺子、火鸡的例子里，借助Embedding，算法能够自动学习到“火鸡”与“饺子”这两个概念的相似性，从而给 **〈中国人，火鸡〉** 的小众组合打一个中等的分数，使火鸡得到了被推荐给中国人的机会，从而能更好地给那些喜欢外国品味的中国人提供了更好的个性化服务。

3.1.4 Embedding的实现细节

Embedding在操作起来还是非常简单的。假设我们有{"音乐"、"影视"、"财经"、"游戏"、"军事"、"历史"}这6个文章类别，编号从0 ~ 5，我们想把每个类别映射成一个长度为4的稠密浮点数向量，整个过程如图3-6所示。

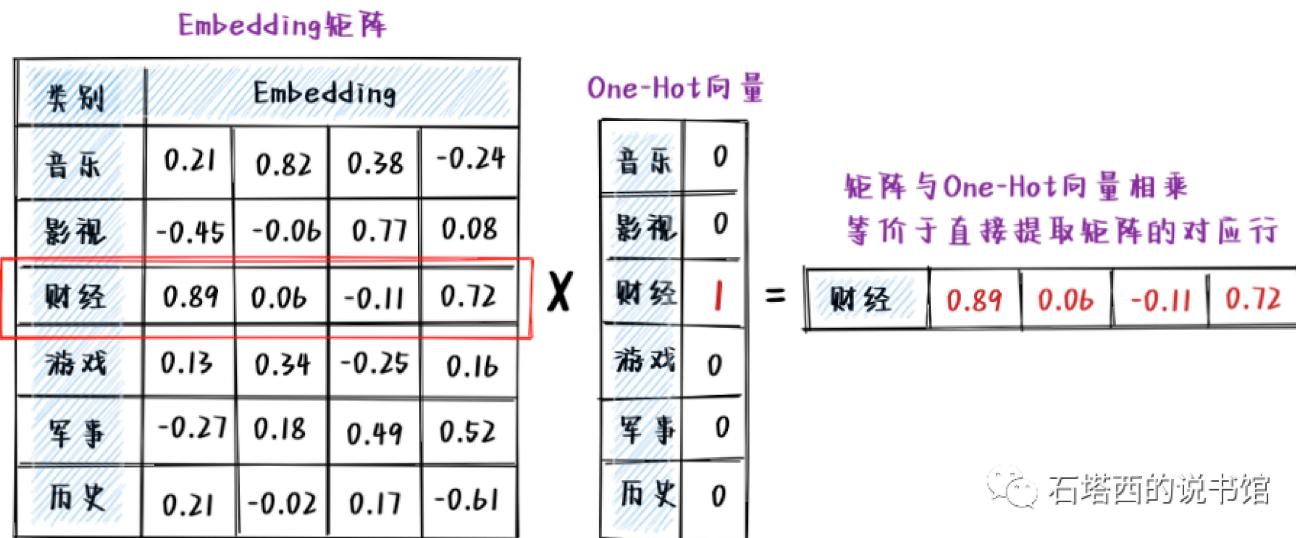


图 3-6 Embedding示意

- 首先，我们定义一个 6×4 的矩阵，行数是要Embedding的特征的总数，列数是希望得到的向量的长度。
- 先随机初始化这个Embedding矩阵。矩阵的内容会随着主目标优化，训练结束时，矩阵内容会变成能表达“文章分类”语义的、有意义的数字。

- 以Embedding"财经"这个类别为例，整个Embedding过程从数学上相当于，一个稠密的Embedding矩阵，与只有"财经"所在的2号位置为1、其余位置全是0的One-Hot向量，相乘。得益于One-Hot向量的稀疏性，以上相乘过程又相当于，从Embedding矩阵直接提取"财经"所在的第2行（首行是0行）。

以上过程如果用TensorFlow实现非常简单，以下几行代码就搞定了，如代码3-1所示。

代码 3-1 TensorFlow实现"文章分类"Embedding

```
import tensorflow as tf

# ----- 准备
unq_categories = ["music", "movie", "finance", "game", "military", "history"]
# 这一层负责将string转化为int型id
id_mapping_layer = tf.keras.layers.StringLookup(vocabulary=unq_categories)

emb_layer = tf.keras.layers.Embedding(
    # 多加一维是为了处理，当输入不包含在unq_categories的情况
    input_dim=len(unq_categories) + 1,
    output_dim=4) # output_dim指明映射向量的长度

# ----- Embedding
cate_input = ... # [batch_size,1]的string型"文章分类"向量
cate_ids = id_mapping_layer(cate_input) # string型输入的“文章分类”映射成int型id
# 得到形状=[batch_size,4]的float稠密向量，表示每个“文章分类”的语义
cate_embeddings = emb_layer(cate_ids)
```

但是TensorFlow提供的tf.keras.layers.Embedding类[2]封装得太好了，傻瓜式的开箱即用，反而不利于我们掌握算法的技术细节。为此，笔者用Python从头实现了一遍对稀疏ID类特征的Embedding，以加深自己对算法的理解。受篇幅所限，本书中只列举其中的关键部分，详细代码请参考文献[3]。

实现难点在于，尽管整个Embedding过程在数学上等价于"稠密Embedding矩阵"与"One-hot/Multi-hot向量"相乘，但是在实现的时候，是万万不能按照矩阵乘法的方式进行的。因为推荐系统中的特征高维稀疏，把稀疏ID特征展开成稠密的One-hot/Multi-hot向量，一个几亿长的向量里最多只有一百来个非零项，计算和存储的代价都是无法接受的。所以，我们必须实现稀疏的前代和回代，回代时不用更新整个Embedding矩阵，而只更新一个batch中出现的那有限几个非零特征对应的那几行。自己将稀疏Embedding实现一遍后，也能帮助你加深对TensorFlow的理解。比如TensorFlow里面的IndexedSlices类就是为了实现稀疏的前代和回代，而用来记录那些有必要更新的位置。

针对单一Field的Embedding的稀疏前代与回代如代码 3-2所示。

代码 3-2 单独一个Field的稀疏前代与回代

```
class EmbeddingLayer:  
    """ 每个Field都有自己独立的Embedding Layer，底层有自己独立的Embedding矩阵  
    """  
  
    def __init__(self, W, vocab_name, field_name):  
        self.vocab_name = vocab_name  
        self.field_name = field_name # 这个Embedding Layer对应的Field  
        self._W = W # 底层的Embedding矩阵，形状是[vocab_size, embed_size]  
        self._last_input = None  
  
    def forward(self, X):  
        """  
        :param X: 属于某个Field的一系列稀疏类别Feature的集合  
        :return: [batch_size, embed_size]  
        """  
        self._last_input = X # 保存本次前代时的输入，回代时要用到  
  
        # output: 该Field的embedding，形状是[batch_size, embed_size]  
        output = np.zeros((X.n_total_examples, self._W.shape[1]))  
  
        # 稀疏输入是一系列三元组的集合，每个三元组由以下三个元素组成  
        # example_idx: 可以认为是sample的id  
        # feat_id: 每个类别特征(不是Field)的id  
        # feat_val: 每个类别特征对应的特征值(一般情况下都是1)  
        for example_idx, feat_id, feat_val in X.iterate_non_zeros():  
            # 根据feature id从Embedding矩阵中取出embedding  
            embedding = self._W[feat_id, :]  
            # 某个Field的embedding是，属于这个Field各个feature embedding的加权和，权重就是各feature v  
            output[example_idx, :] += embedding * feat_val  
  
        return output # 这个Field的embedding  
  
    def backward(self, prev_grads):  
        """  
        :param prev_grads: loss对这个Field output的梯度，[batch_size, embed_size]  
        :return: dw, 对Embedding Matrix部分行的梯度
```

```

"""
# 只有本次前代中出现的feature id, 才有必要计算梯度
# 其结果肯定是非常稀疏的, 用dict来保存
dW = {}

# _last_input是前代时的输入, 只有其中出现的feature_id才有必要计算梯度
for example_idx, feat_id, feat_val in self._last_input.iterate_non_zeros():
    # 由对field output的梯度, 根据链式法则, 计算出对feature embedding的梯度
    # 形状是[1, embed_size]
    grad_from_one_example = prev_grads[example_idx, :] * feat_val

    if feat_id in dW:
        # 一个batch中的多个样本, 可能引用了相同的feature
        # 因此对某个feature embedding的梯度, 应该是来自多个样本的累加
        dW[feat_id] += grad_from_one_example
    else:
        dW[feat_id] = grad_from_one_example

return dW

```

EmbeddingCombineLayer类负责将多个Field Embedding拼接起来向上层传递，并支持多个Field共享Embedding矩阵，其实现如代码3-3所示，其中用到了代码3-2中定义的EmbeddingLayer。

代码3-3 多个Field的稀疏前代与回代

```

class EmbeddingCombineLayer:
    """
    多个EmbeddingLayer的集合, 每个EmbeddingLayer对应一个Field
    允许多个Field共享同一套Embedding Matrix (用vocab_name标识)
    """

    .....

    def forward(self, sparse_inputs):
        """
        所有Field, 先经过Embedding, 再拼接
        :param sparse_inputs: dict {field_name: SparseInput}
        :return: 每个SparseInput贡献一个embedding vector, 返回结果是这些embedding vector的拼接
        """

        embedded_outputs = []
        for embed_layer in self._embed_layers:
            # 获得属于这个Field的稀疏特征输入, sp_input是一组<example_idx, feat_id, feat_val>
            sp_input = sparse_inputs[embed_layer.field_name]

```

```

# 得到属于当前Field的embedding
embedded_outputs.append(embed_layer.forward(sp_input))

# 最终结果是所有Field Embedding的拼接
# [batch_size, sum of all embed-layer's embed_size]
return np.hstack(embedded_outputs)

def backward(self, prev_grads):
    """
    :param prev_grads: [batch_size, sum of all embed-layer's embed_size]
        上一层传入的, Loss对本层输出 (i.e., 所有field embedding拼接) 的梯度
    """

    # prev_grads是loss对“所有field embedding拼接”的导数
    # prev_grads_splits把prev_grads拆解成数组,
    # 数组内每个元素对应loss对某个field embedding的导数
    col_sizes = [layer.output_dim for layer in self._embed_layers]
    prev_grads_splits = utils.split_column(prev_grads, col_sizes)

    # _grads_to_embed也只存储"本次前代中出现的各field的各feature"的embedding
    # 其结果是超级稀疏的, 因此_grads_to_embed是一个dict
    self._grads_to_embed.clear() # reset

    for layer, layer_prev_grads in zip(self._embed_layers, prev_grads_splits):
        # Layer_prev_grads: 上一层传入的, Loss对某个field embedding的梯度
        # Layer_grads_to_feat_embed: dict, feat_id=>grads,
        # 某个field的embedding Layer造成对某vocab的embedding矩阵的某feat_id对应行的梯度
        layer_grads_to_embed = layer.backward(layer_prev_grads)

        for feat_id, g in layer_grads_to_embed.items():
            # 表示"对某个vocab的embedding weight中的第feat_id行的总导数"
            key = "{}@{}".format(layer.vocab_name, feat_id)

            if key in self._grads_to_embed:
                # 由于允许多个field共享embedding matrix,
                # 因此对某个embedding矩阵的某一行的梯度应该是多个field贡献梯度的叠加
                self._grads_to_embed[key] += g
            else:
                self._grads_to_embed[key] = g

```

3.2 共享还是独占Embedding

Embedding本身实现起来比较简单，无非就是随机初始化一个矩阵，其中的每一行对应一个类别特征，然后就是由SGD随模型一同优化。TensorFlow和PyTorch都有成熟的接口，直接调用就行，没啥好说的。但是，还是有一个需要我们在建模时抉择取舍的点，就是“共享”还是“独占”Embedding。

3.2.1 共享Embedding

所谓共享Embedding，是指同一套Embedding要喂入模型的多个地方，发挥多个作用。共享Embedding的好处有二：

- 一来，能够缓解由于特征稀疏、数据不足所导致的训练不充分。
- 二来，Embedding矩阵一般都很大，复用能够节省存储空间。

比如，模型要用到“近7天安装的APP”、“近7天启动过的APP”、“近7天卸载的APP”这三个Field，而每个具体的APP是一个Feature，要映射成Embedding向量。如果三个Field不共享Embedding：

- “装启卸”三个Field都使用独立的Embedding矩阵来将APP映射成稠密向量，整个模型需要优化的参数变量是共享模型的3倍，需要更多的训练数据，否则容易过拟合。
- 每个Field的稀疏程度是不一样的，同一个APP，在“启动列表”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。如果担心以上两点，那么我们可以让“装启卸”这三个Field共享一个Embedding矩阵。

再比如，在5.5节要介绍的双塔模型中，

- Item ID Embedding既是重要的物料特征，要喂入Item Tower；
- 同时，用户行为序列作为最重要的用户侧特征，也是由一系列的Item ID组成，因此Item ID Embedding也要喂入User Tower。

如果选择让喂入User Tower和Item Tower的Item ID Embedding共享同一个Embedding矩阵，模型结构如图3-7所示。

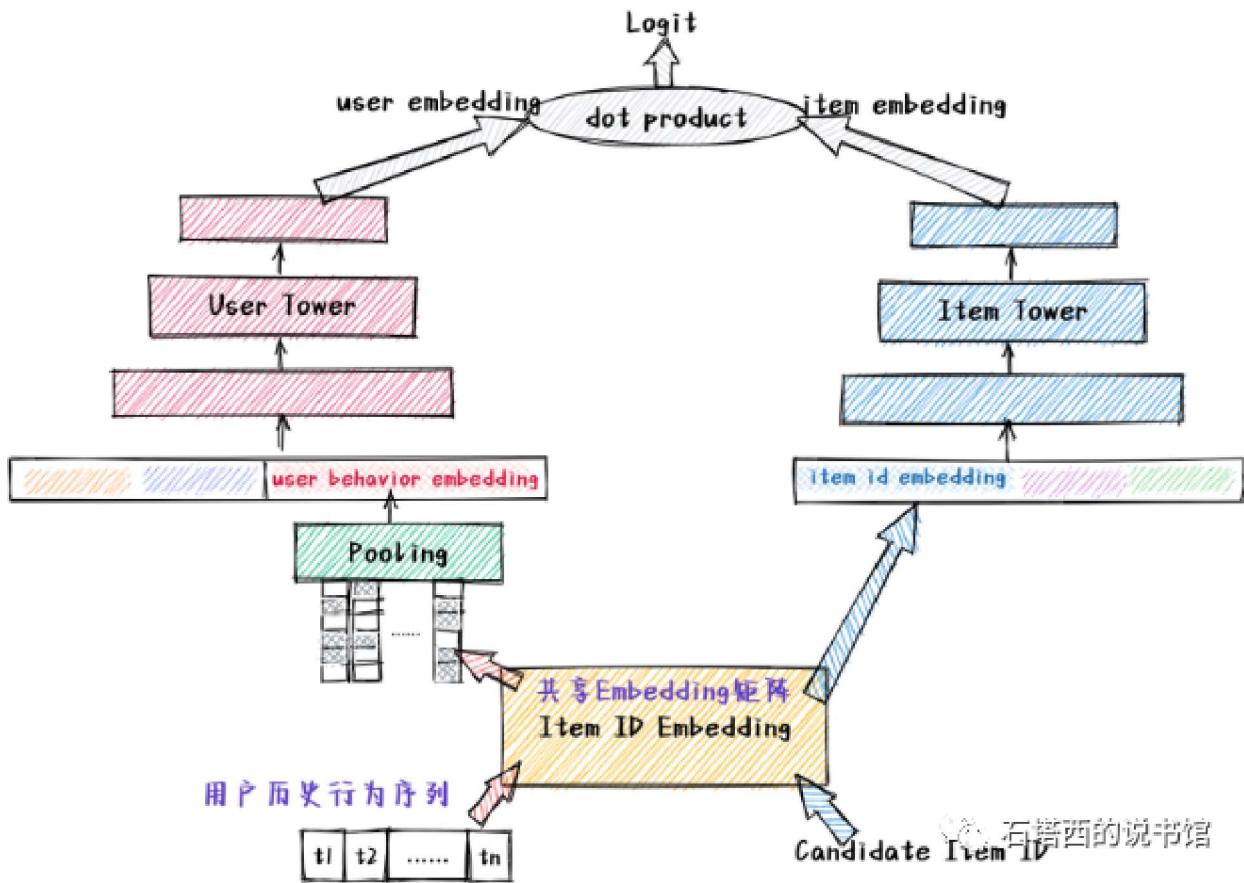


图 3-7 共享Embedding矩阵的双塔模型

另一类Embedding共享发生在特征交叉的时候。这方面的典型代表就是Factorization Machine (FM)。FM的算法原理请见4.2.2节。注意，在FM中，每个特征只有一个Embedding，在与其他不同特征交叉时，共同使用这唯一的Embedding。FM也因为共享Embedding，而获得有利于训练稀疏特征、提升模型扩展性等益处。

3.2.2 独占Embedding

独占Embedding以避免相互干扰

如前所述，共享Embedding最大的优点，就是缓解因为数据不足而导致的稀疏特征训练不充分的问题。但是各互联网大厂最不缺的就是数据，这时共享Embedding的缺陷就暴露出来，即不同目标在训练同一套Embedding时可能相互干扰。

比如在之前的案例中，APP的安装、启动、卸载，对于要学习的APP Embedding有着不同的要求。理想情况下，“安装”与“启动”两个Field要求APP Embedding能够反映出APP为什么能够招人喜欢，而“卸载”这个Field要能够反映出APP为什么招人烦。举个例子，有两款音乐APP，它们都因为曲库丰富被人所喜欢，“安装”与“启动”这两个Field要求这两个音乐APP的Embedding距离相近；但是这两个APP的缺点不同，一个是因为

为收费高，另一个是因为广告频繁，因此“卸载”Field要求这两个音乐APP的Embedding相距远一些。显然，用一套APP Embedding很难满足以上两方面的需求，所以大厂一般选择让“装/启/卸”三个Field各自拥有独立的Embedding矩阵。

同理，用户有着不同类型的行为历史，比如点击历史、购买历史、收藏历史、点赞历史、……，各种历史行为序列都是由一堆Item ID组成。各种类型的动作对Item ID Embedding所表达的语义有着不同的要求，为了避免相互干扰，同一个物料在不同的行为序列中可以使用不同的Item ID Embedding。而且参与刻画用户行为历史的Item ID Embedding，与被用于物料特征的Item ID Embedding，也彼此隔离，互不共享。

更有甚者，大厂的推荐系统都是多目标的，比如要同时优化点击率、购买率、转发率、……等多个目标。有一些重要特征，在参与不同目标的建模时，也要使用不同的Embedding。

特征交叉时的Embedding独占

另一个独占Embedding的重要应用场景，发生在特征交叉的时候。比如在Factorization Machine (FM) 中，每个Feature与不同Feature交叉时，使用的是同一个Embedding，如公式(3-1)所示。FM的详细讲解见本书的4.2.2节。

$$\begin{aligned} logit_{FM} &= \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j \\ &= \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n (\mathbf{v}_i \cdot \mathbf{v}_j) x_i x_j \end{aligned} \quad (3-1)$$

- x_i 、 x_j 分别表示一条样本的第*i*、*j*个Feature，*n*表示样本中的特征总数。
- \mathbf{v}_i 、 \mathbf{v}_j 分别表示第*i*、*j*两个Feature的Embedding。
- $w_{ij} = \mathbf{v}_i \cdot \mathbf{v}_j$ 是特征组合 $x_i x_j$ 的系数。

公式(3-1)说明，无论特征*i*与哪个特征交叉，FM都是用相同 v_i 来生成交叉特征的系数，即Embedding是共享的。这可能存在互相干扰的问题，比如模型调整 v_i 以便把 w_{ij} 学习好，但是却可能对另一对特征组合的系数造成负面影响。

为了解决这一问题，业界提出了FM的改进版本Field-aware Factorization Machine (FFM) [4]，在Kaggle比赛中取得了更好的效果。其核心思想是，每个特征在与不同特征交叉时，根据对方特征所属的Field要使用不同的Embedding，见公式(3-2)。

$$\begin{aligned} x &= \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j \\ &= \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n (\mathbf{v}_{i,f_j} \cdot \mathbf{v}_{j,f_i}) x_i x_j \end{aligned} \quad (3-2)$$

- f_j 表示第*j*个特征所属的Field。

- v_{i,f_j} 表示第*i*个Feature针对第*j*个Feature所在Field的Embedding。说明第*i*个特征在与不同的特征交叉时，使用了不同的Embedding。
- 其他符号含义参考公式(3-1)。

FFM的一大缺点在参数爆炸。原来FM中，每个Feature只有一个Embedding，如果系统中有*n*个feature，每个Feature Embedding的长度为*k*，Embedding部分的参数总量是*nk*。而到了FFM，如果这*n*个feature属于*f*个field，那么Embedding部分的参数总量就变成了*nfk*，需要更多的训练数据。

2021年，阿里妈妈提出了Co-Action Network (CAN) [5]，在“独占Embedding”这条技术路线上又迈进了一大步。在笔者看来，CAN的目标有两个：

- 既想像FFM那样，让每个特征在与其他不同特征交叉时，使用完全不同的Embedding。
- 又不想像FFM那样引入那么多参数，导致参数空间爆炸，增加训练的难度。

为了兼得以上两个目标，CAN提出如图 3-8所示的网络结构进行特征交叉。

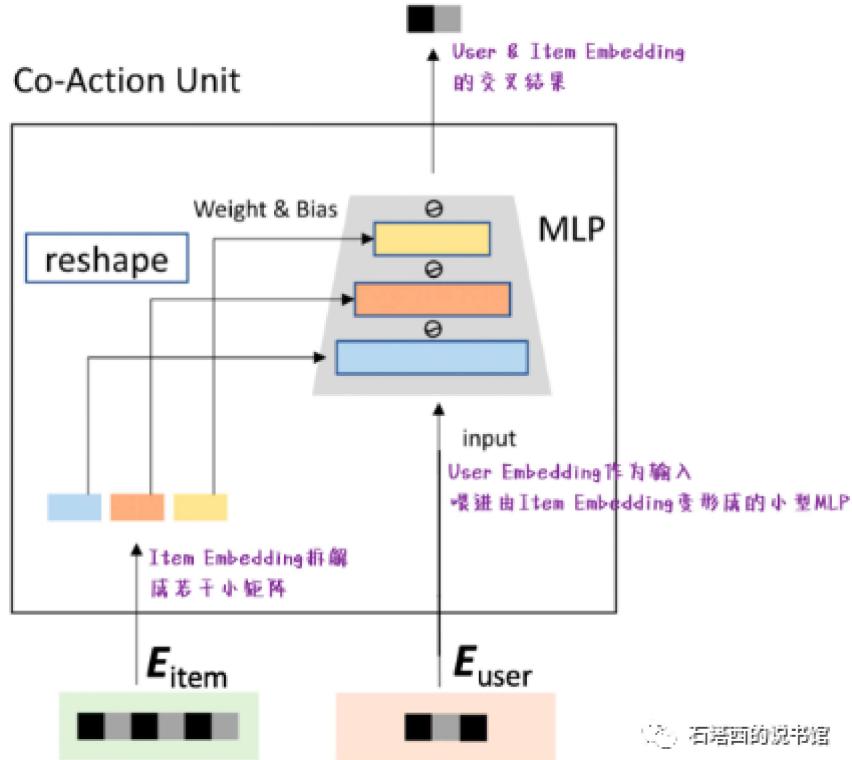


图 3-8 CAN中特征交叉的结构示意

P_{item} 是参与交叉的某个物料特征的Embedding。首先将 P_{item} 拆解成*K*段，每段子向量变形（Reshape）成一个小矩阵 \mathbf{W}_i 。这个拆解过程可以由公式(3-3)来表示，其中Concat表示拼接操作，Flatten表示将一个二维矩阵摊平成一个一维向量的操作。

$$E_{item} = \text{Concat}(\{\text{Flatten}(\mathbf{W}_i) | i = 0, \dots, K - 1\}) \quad (3-3)$$

然后将各 \mathbf{W}_i 组成一个小型的“多层感知器”（Multilayer Perceptron，MLP），第*i*层的权重就是 \mathbf{W}_i ，层与层之间插入非线性激活函数ReLU。

使用的时候，将某个用户特征的Embedding喂入这个由物料特征Embedding变形出来的小型MLP，MLP的输出就是这两个特征的交叉结果，如公式(3-4)所示。其实反过来，将用户特征 E_{user} 变形成为MLP，拿物料特征 E_{item} 喂入MLP，也是可以的。

$$\begin{aligned} h^{(0)} &= E_{user} & (a) \\ h^{(i)} &= \text{ReLU}(\mathbf{W}_i h^{(i-1)}) & (b) \\ h^{(K)} &= H(E_{user}, E_{item}) & (c) \end{aligned} \quad (3-4)$$

- $h^{(i)}$ 表示由变形出来的小型MLP中第*i*层的输出。
- 公式(3-4)(a)表示将用户特征Embedding " E_{user} "作为这个小型MLP的原始输入。
- 公式(3-4)(c)中， $H(E_{user}, E_{item})$ 表示用户特征 E_{user} 与物料特征 E_{item} 的交叉结果，就是这个小型MLP最后一层的输出 $h^{(K)}$ 。

CAN的优势在于：

- 根据公式(3-4)(b)， $h^{(i-1)}$ 也是非线性激活函数ReLU的结果，其中有一些位置会是0，这就意味着 \mathbf{W}_i 有些神经元不会发挥作用。由于 \mathbf{W}_i 是由 E_{item} 中的某段子向量变型而成的，这也就意味着同一个物料特征的Embedding " E_{item} "，在与不同的用户特征交叉时，向量中的不同区域发挥使用。相当于同一个物料特征在与不同用户特征交叉时，使用了不同的Embedding，从而降低了不同交叉之间的相互干扰。
- 与此同时，CAN并没有像FFM那样为每对特征交叉都引入独立的参数。参数空间没有爆炸，计算压力、存储压力、训练不充分等问题都得以缓解。

3.3 Parameter Server: 推荐算法的训练加速器

前面的章节已经介绍了，推荐系统的数据特点，一是海量训练数据，二是特征空间高维、稀疏。在这样的严苛的数据环境下，我们还能做到在线实时训练，使模型能够追踪用户与物料的最新状态，背后的利器就是一个功能强大、鲁棒可靠的"参数服务器"（Parameter Server，PS）。因此，作为在推荐、广告、搜索算法里卷得不亦乐乎的咱们互联网打工人，不了解点PS的技术原理是说不过去的。毕竟，搞不定PS，再强的模型也训练不出来。

3.3.1 传统分布式计算的不足

如前文所述，推荐系统对Parameter Server的需求并不是凭空产生的，而是针对推荐系统的两大痛点，海量数据+高维稀疏特征空间，应运而生。

其实推荐系统对海量数据并不陌生，我们有Hadoop/Spark这样的大数据工具对付它们。所以很直觉的想法就是，能不能也拿Hadoop/Spark来分布式地训练模型？想想也很简单：

1. 将训练数据分散到所有Slave节点。
2. Master节点将模型的最新参数广播到所有Slave节点。
3. 每个Slave节点收到最新的参数后，用本地训练数据，先前代再回代，计算出梯度并上传至Master。
4. Master节点收集齐所有Slave节点发来的梯度后，平均之，再用平均后的梯度更新模型参数。
5. 回到步骤1，开始下一轮训练。

看上去似乎行得通。但是该方案忽略了推荐系统中数据的第二个特点，“高维稀疏的特征空间”，给以上方案造成了两个困难：

- 推荐系统的特征动辄上亿、上十亿，每个特征的Embedding是16位、32位甚至更长，这么大的参数量是一台Master所容纳不下的。
- 每轮训练中，Master节点都要将这么大的参数量广播到各Slave节点，每个Slave还要将相同大小的梯度回传，占据的带宽、造成的时延都是不敢想像的，绝对达不到在线实时训练的需求。

3.3.2 基于PS的分布式训练范式

为了解决推荐系统中大规模分布式训练的难题，经典论文《Scaling Distributed Machine Learning with the Parameter Server》提出了Parameter Server架构[6]。简单来说，Parameter Server就是一个分布式的KV数据库，两点设计使它能够克服Master/Slave架构应用于大规模分布式训练时的困难：

- 模型参数不再集中存储于单一的Master节点，而是由一群PS server节点共同存储、读写，从而突破了单台机器的资源限制，也避免了“单点失效”问题。
- 得益于推荐系统的特征是超级特征的特点，一个batch的训练数据所包含的非零特征的数目是极其有限的。因此，我们在训练每个batch时，没必要将整个模型的所有参数（i.e., 上亿个Embedding）在集群内部传来传去，而只需要传递当前batch所涵盖的有限几个非零特征的参数就可以了，从而能够大大节省带宽与传输时间。

结构上一个Parameter Server主要由三大类节点组成，如图3-9所示，各类节点的功能如表 3-1所示。

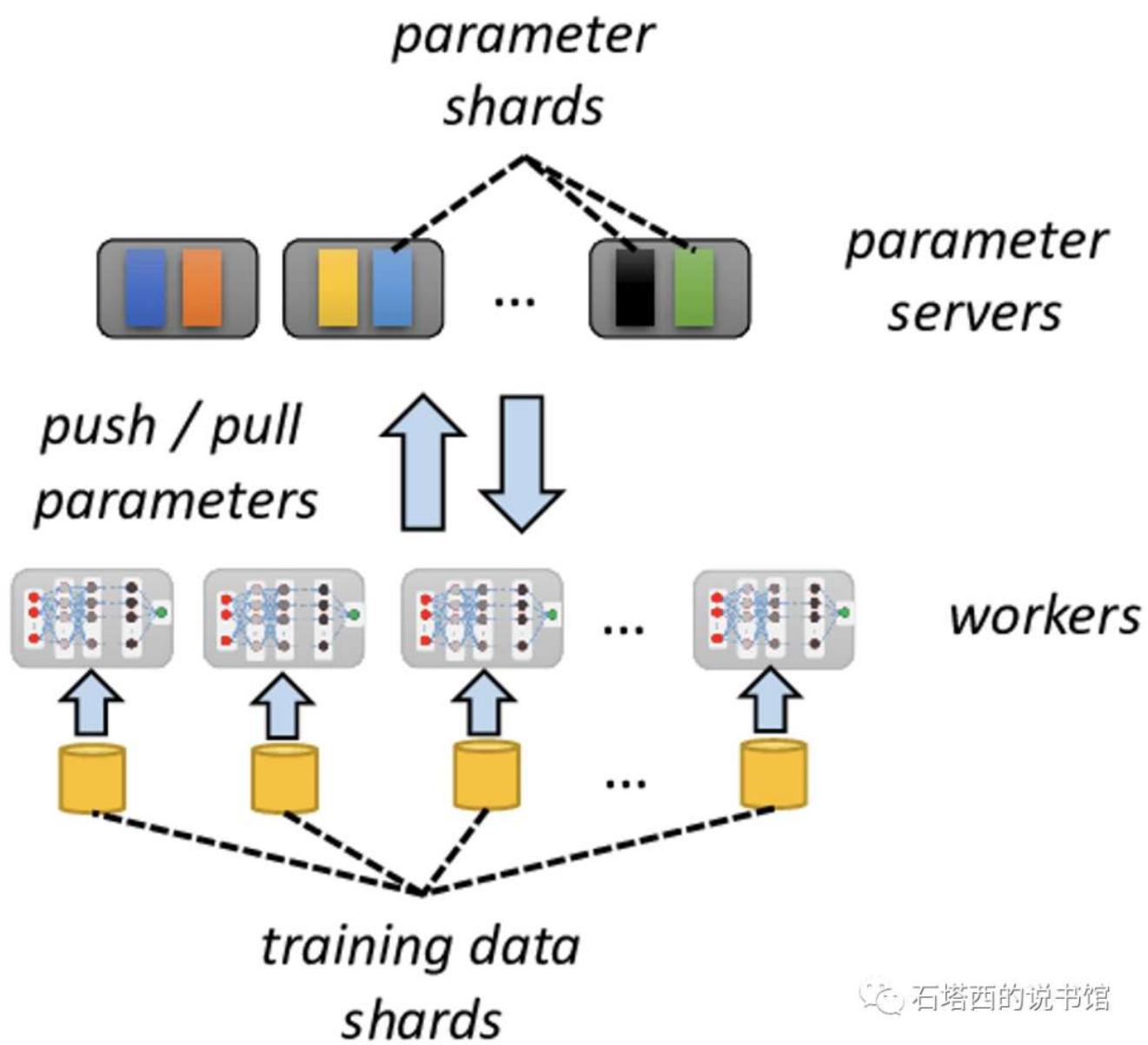


图 3-9 PS中的三类节点

表 3-1 PS中的三类节点

节点类型	节点功能
Worker	PS 中会有多个 Worker 节点。每个 Worker 节点会进行如下操作： <ul style="list-style-type: none">● 从 Server 拉 (Pull) 取最新的模型参数● 用本地数据训练，计算出梯度● 向 Server 推送 (Push) 梯度
Server	PS 中会有多个 Server 节点，每个其实就是一个 Key/Value 数据库： <ul style="list-style-type: none">● 若干 Server 共同存储推荐模型的上亿参数，一个 Server 只负责处理海量参数的其中一部分（又称为 Shard）。● 应对 Pull 请求，将 Worker 请求的参数的最新值发送回去。● 应对 Push 请求，聚合各 Worker 发送过来的梯度，再利用各种 SGD 算法（比如 Adam、Adagrad）更新模型参数。
Scheduler	负责整个 PS 集群的管理，比如接受新节点的注册、将 Pull 请求路由到合适的 Server 节点等

基于Parameter Server的训练流程如下图 3-10所示，每个步骤的具体描述见表3-2。

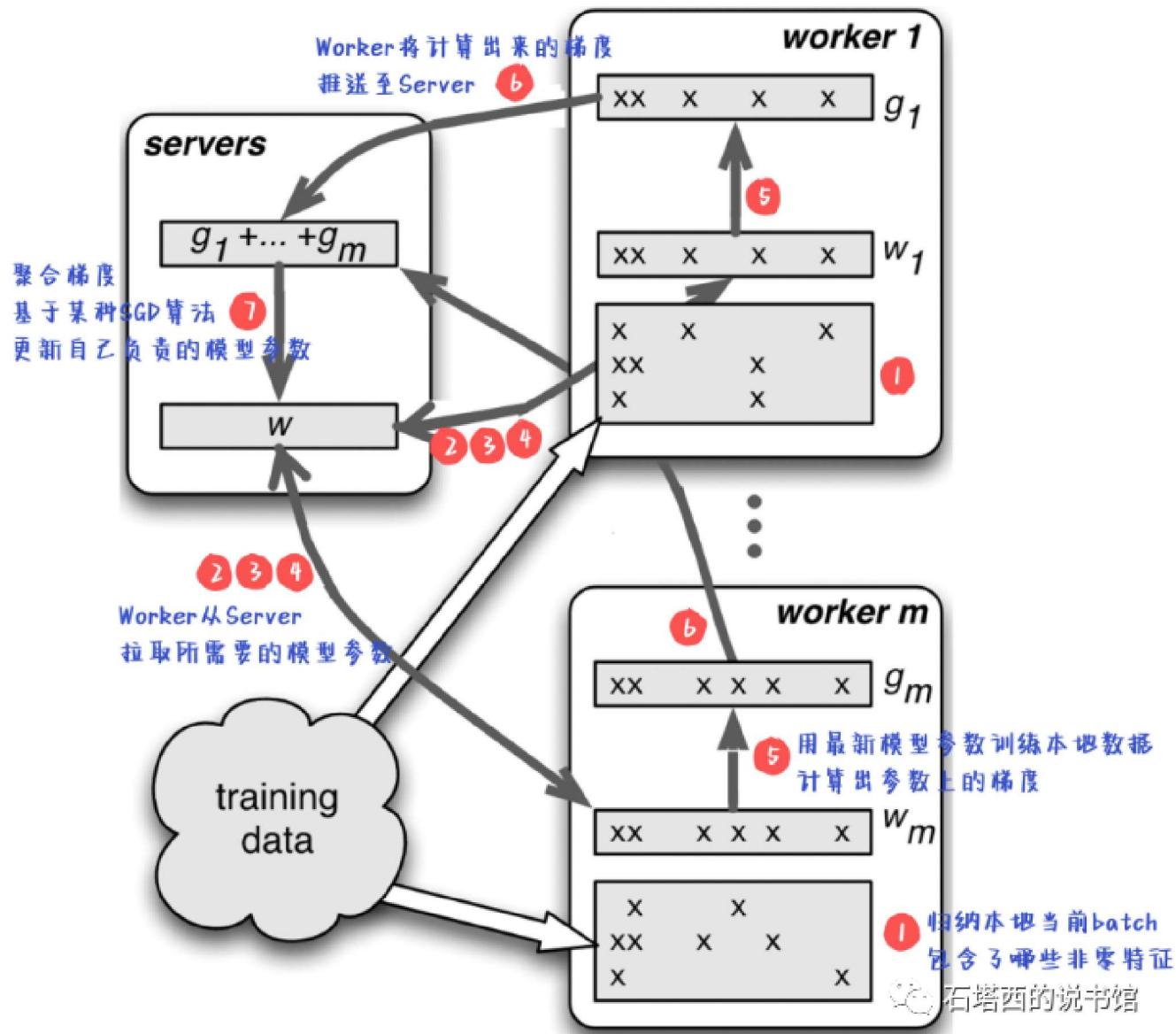


图 3-10 基于Parameter Server的训练流程示意

表 3-2 基于PS的训练步骤描述

步骤	操作节点	操作
1	Worker	训练前，每个Worker先归纳一下当前batch的本地训练数据涵盖了哪些Feature。观察图3-10中的每个"x"标志，其所在行代表一个样本，其所在列代表一个Feature，可以发现训练数据的确是非常稀疏的。因此一个batch所涵盖的非零特征相对于整个特征空间，仍然是十分稀少的。
2	Worker	每个Worker根据第1步的结果，向PS Servers集群发出Pull请求，拉取训练当前batch需要用到的模型参数（比如特征的一阶权重和Embedding）
3	Scheduler	因为每台Server只拥有一部分特征的参数，所以需要一个路由机制，将Worker的Pull请求拆分，分别路由到合适的Servers上。
4	Server	收到Pull请求的Server，从本地存储中找到所请求的那些特征的最新参数，回复给Worker。
5	Worker	收集齐Servers返回的最新模型参数，worker就可以在当前batch的训练数据上，先前代后回代，得到这些参数上的梯度。
6	Worker	Worker向Server发送Push请求，将计算出来的梯度发往Server。
7	Server	接到Workers发来的梯度后，Server聚合汇总梯度（比如求平均），再用某种SGD算法（比如，FTRL、Adam、Adagrad等）更新其负责的那一部分模型参数，至此完成一轮训练。

总结下来，Parameter Server训练模式是Data Parallelism（数据并行）与Model Parallelism（模型并行）的这两种分布式计算范式的结合体：

- Data Parallelism：数据并行很好理解，海量的训练数据分散在各个节点上，每个节点只训练本地的一部分数据，多机并行计算加快了训练速度。
- Model Parallelism：推荐系统中的特征动辄上亿，每个Embedding又包含多个浮点数，这么大的参数量是单机无法承载的，必然分布在一个集群中。接下来，我们也会讲到，由于推荐系统中特征高度稀疏的性质，一轮迭代中，不同Worker节点不太会在同一个特征的参数上产生竞争，因此多个Worker节点相对解耦，天然适合Model Parallelism。

3.3.3 PS中的并发策略

根据各Worker节点冲突的频繁程度，Parameter Server中的并发策略可以划分如下三类。

同步并发 (BSP)

同步并发策略 (Bulk Synchronous Parallel, BSP) 下，顺序执行以下四步，如图3-11所示：

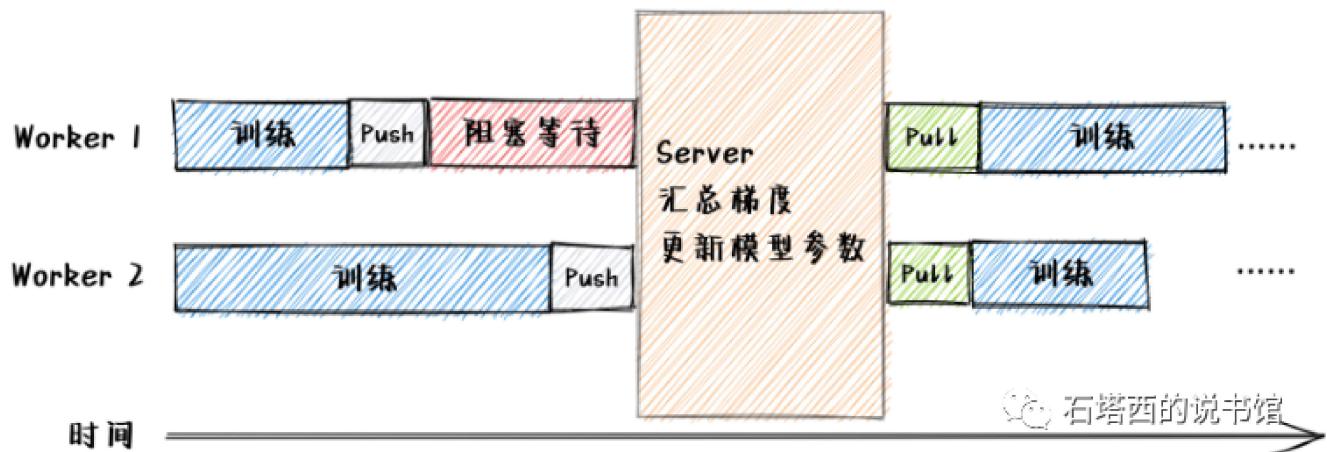


图 3-11 BSP并发策略示意

1. 各Worker完成自己的本轮计算，将梯度汇报给Server，然后阻塞等待。
2. Server在收集齐所有Worker上报的梯度后，聚合梯度，用SGD算法更新自己负责的那部分模型参数。
3. Server通知各Worker解除阻塞。
4. Worker接到解除阻塞的通知，从Server拉取更新过的模型参数，开始下一轮训练。

这种模式的优点是，多个Worker节点更新Server上的参数时不会发生冲突，所以分布式训练的效果赞同于单机训练的效果。缺点是，一轮迭代中，速度快的节点要停下来等待速度慢的节点，从而形成了"短板效应"，一个慢节点就能拖累整个集群的计算速度。

异步并发 (ASP)

如图 3-12所示，在异步并发策略 (Asynchronous Parallel, ASP) 下，每台Worker在推送自己的梯度至Server后，不用等待其他Worker，就可以开始训练下一个batch的数据。由于无须同步，不存在"短板效应"，ASP具有明显的速度优势。

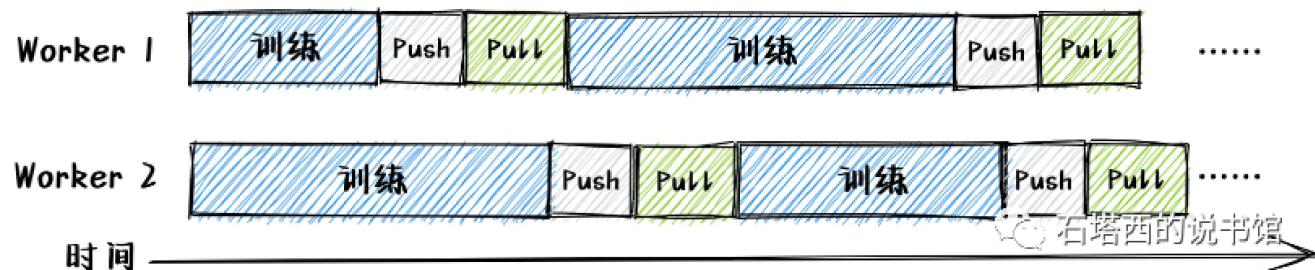


图 3-12 ASP并发策略示意

但是由于缺乏同步控制，ASP可能发生“梯度失效”（Stale Gradient）的问题，从而影响收敛速度。举一个极度简化的例子：

- 当前Server端上模型参数的版本是 θ_0 ，有两个Worker节点，都从Server拉取 θ_0 ，同时开始一轮训练。
- Worker 1的速度比较快，很快训练完本地数据并向Server上报梯度 g_1 。
- Server收到 g_1 后，根据SGD算法迭代一步（步长为 λ ），将Server端的参数值由 θ_0 更新为 $\theta_1 = \theta_0 - \lambda g_1$ 。
- 此时Worker 2才完成计算并向Server上报了自己的梯度 g_2 。
- Server收到 g_2 后，如果像 $\theta_2 = \theta_1 - \lambda g_2$ 这样更新模型参数，反而可能损害收敛。这是因为 g_2 是Worker 2基于 θ_0 计算得到的，而Server端的参数此时已经变成了 θ_1 ， g_2 已经失效。

但是在实践中，这个“梯度失效”问题并没有那么严重。得益于推荐系统中的特征超级稀疏的特点，在一轮迭代中，各个Worker节点的局部训练数据所包含的非零特征，相互重叠得并不严重。多个Worker节点同时更新同一个特征的参数（i.e., 一阶权重或Embedding）的可能性非常小，所以Server端的冲突也就没有那么频繁和严重，ASP模式在推荐系统中依然比较常用的。

细心的读者可能会有疑问，在一轮训练中，特征是稀疏的，两个Worker不太可能同时更新同一个特征的参数，使用ASP也较少发生冲突，但是DNN中各层的权重是所有Worker都要更新的吧，使用ASP导致了冲突怎么办？这是一个非常好的问题，也是现代Parameter Server改进的方向之一，接下来在3.3.5节介绍现代Parameter Server的时候会提到一些解决方案。

半同步半异步（SSP）

半同步半异步（Staleness Synchronous Parallel, SSP）是BSP与ASP的折衷方案。SSP允许各Worker节点在一定迭代轮数之内保持异步。如果发现最快Worker节点与最慢Worker节点的迭代步数之差已经超过了允许的最大值，所有Worker都要停下来进行一次参数同步，如所示。SSP希望通过折衷，实现“计算效率”与“收敛精度”之间的平衡。

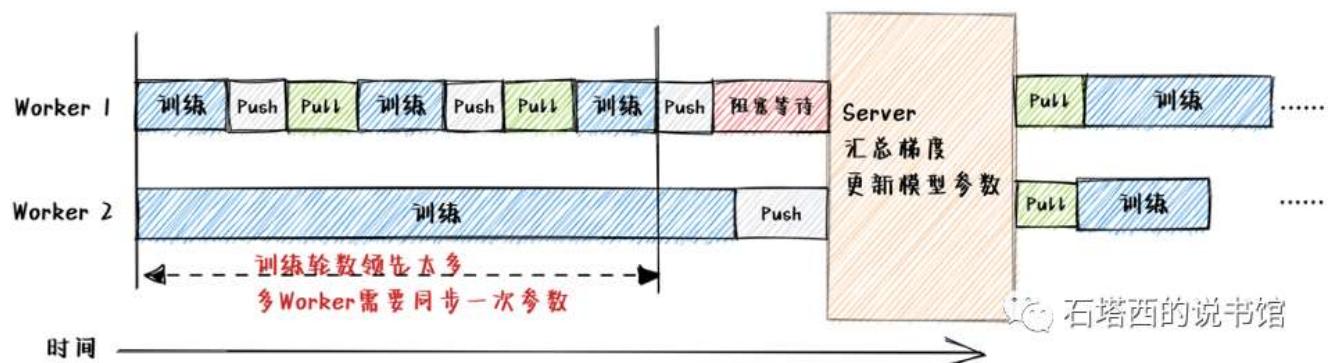


图 3-13 SSP并发策略示意

3.3.4 基于ps-lite实现分布式算法

为了加深对Parameter Server的理解，本节演示如何利用ps-lite框架实现分布式FM算法。

ps-lite简介

ps-lite[7]是Parameter Server的早期代表，首开PS大规模应用之先河。看名字里的lite就能猜出，“简洁轻巧”是它的最大特点。相比于它的那些大块头后辈，比如阿里的XDL[8]、百度的PaddlePaddle Fleet[9]、腾讯的Angel[10]、Uber的Horovod[11]等，ps-lite所提供的功能是相对简单和有限的，缺乏现代PS针对推荐系统特色而专门做的一些优化，比如Embedding和DNN权重使用不同的通信与同步模式、为了避免存储膨胀而引入的特征准入与逐出机制、...等。但是“麻雀虽小，五脏俱全”，得益于简洁的代码与轻巧的结构，ps-lite给了我们一个深入PS内部、一探PS运行机理的机会。也为我们在实际工作中，调优ps-lite那些更强大也更复杂的后辈，奠定了扎实的理论基础。

ps-lite中有5个重要角色，见表3-3。其中，PostOffice、Van、Customer都隐藏在ps-lite内部，ps-lite的使用者无需过多关心。要想基于ps-lite实现分布式算法，我们只需要继承Worker并实现“计算梯度”的逻辑，和继承Server实现“训练参数的存储和更新”逻辑。

表 3-3 ps-lite中的5个重要的角色

角色	功能
PostOffice	每个进程（无论Scheduler、Worker还是Server），有且只有一个“邮局”（PostOffice），是该进程的消息集散中心。它掌管着在它那里等消息的Customer的名单。
Van	每个邮局只有一辆“邮车”（Van），负责收发信息的具体工作。派发信息时，Van通过Post Office掌握的顾客名单，找到某个Customer，将信件交到他手上。
Customer	邮局的顾客，其实就是一个中介。Worker和Server为了更好地专心业务工作，将在邮局排队等消息的工作外包给了“顾客”。Customer从Van手中接到消息后，转手就交给它的雇主，Worker或Server。
Worker	Customer的雇主，训练模型时，主要负责处理样本、计算梯度等工作。
Server	Customer的雇主，训练模型时，主要负责读取参数、接收梯度、汇总梯度（同步模式下）、更新模型参数等工作。

受篇幅所限制，本书只介绍ps-lite的基本概念。对ps-lite的技术细节感兴趣的同学，可以参考文献[12]，里面有比较详细的ps-lite源码解析。

介绍完基本概念，接下来演示一下，如何利用ps-lite实现分布式FM算法。本节使用的分布式FM源码来自xflow[13]这个开源项目，我在原作者代码的基础上添加了详细注释，帮助读者理解。关于FM的算法理论，将在4.2.2节加以详细介绍。

Server端示范代码

Server端的构造函数，如代码 3-4所示，先建立并启动了两个 KVServer， KVServer 是 ps-lite 提供的类，可以理解为在Server端建立了两个Key-Value (KV) 数据库。建立KVServer时传入的整形参数叫 app_id，可以理解为某个KV数据库的唯一标识。代码指定 app_id=0 的数据库存储各特征的一阶权重，指定 app_id=1 的数据库存储各特征的Embedding向量。

```
class Server
{
public:
    Server()
    {
        // server_w_是app_id=0的KV数据库，用于读写各特征的一阶权重
        server_w_ = new ps::KVServer<float>(0);
        // 处理一阶权重的具体逻辑，实现在KVServerSGDHandle_w中
        server_w_->set_request_handle(SGD::KVServerSGDHandle_w());

        // server_v_是app_id=1的KV数据库，用于读写各特征的embedding
        server_v_ = new ps::KVServer<float>(1);
        // 处理embedding的具体逻辑，实现在KVServerSGDHandle_v中
        server_v_->set_request_handle(SGD::KVServerSGDHandle_v());
    }
    ~Server() {}

    ps::KVServer<float> *server_w_;
    ps::KVServer<float> *server_v_;
};
```

代码 3-4 分布式FM Server构造函数

Server接收到Worker发来的梯度后，用FTRL算法更新一阶权重和Embedding，以上业务逻辑分别实现在 KVServerSGDHandle_w 和 KVServerSGDHandle_v 中。这两个类的逻辑差不多， KVServerSGDHandle_w 只不过是将一阶权重看成长度为1的Embedding向量，所以，接下来重点看一下 KVServerSGDHandle_v 是如何存储、更新Embedding的，如代码3-5所示。

```
struct KVServerSGDHandle_v
{
```

```

// operator()是Server端处理每个Pull & Push请求的加高函数
// req_meta包含请求的元信息, req_data包含请求的数据
// server是调用这个回调函数的KVServer实例, 提供一些API可在回调函数中使用
void operator()(const ps::KVMeta &req_meta, const ps::KVPairs<float> &req_data, ps::KVServer<
{
    size_t keys_size = req_data.keys.size(); // keys_size代表了一共请求了多少个特征的参数
    size_t vals_size = req_data.vals.size();
    ps::KVPairs<float> res; // 用于填充回复结果

    if (req_meta.pull) // 如果是一个Pull请求
    {
        res.keys = req_data.keys; // 请求了哪些特征, 就是回复哪些特征
        // v_dim是每个embedding的长度, 一共有keys_size个特征
        // 结果的vals一共要开辟长度=keys_size * v_dim的数组
        res.vals.resize(keys_size * v_dim);
    }

    for (size_t i = 0; i < keys_size; ++i) // 遍历请求的每个特征
    {
        ps::Key key = req_data.keys[i]; // key是请求的第i个特征的ID

        // store是KVServerSGDHandle_v的成员变量, 是一个unordered_map
        // SGDEntry_v就是只有一个vector<float>成员的struct
        SGDEntry_v &val = store[key]; // 根据请求的特征的ID, 找到Server端存储的embedding

        for (int j = 0; j < v_dim; ++j) // 遍历embedding的每一位
        {
            if (req_meta.push) // 如果是push请求, 就是要更新本地的embedding
            {
                // 请求数据req_data的数据域vals, 是所有特征embedding的梯度拼接成的大向量
                // g表示对第i个特征的embedding的第j位的梯度
                float g = req_data.vals[i * v_dim + j];
                // val是指向本地存储的引用, 这里用SGD算法更新第i个特征的embedding的第j位
                val.w[j] -= learning_rate * g;
            }
            else // 否则就是pull请求, 从Server端提取embedding
            {
                // val.w就是本地存储的第i个特征的embedding
                // 这里是将其按位复制到结果res.vals中
                res.vals[i * v_dim + j] = val.w[j];
            }
        }
    }
}

```

```

        }
        // for遍历每一位
    }
    // for遍历每个特征
    server->Response(req_meta, res); // 回复给worker
}

private:
    // 用一个map来存储各特征的embedding
    // Key是一个特征的唯一标识ID, sgdentry_v是对一个vector<float>的封装, 用于存储embedding向量
    std::unordered_map<ps::Key, sgdentry_v> store;
};

```

代码 3-5 FM的Server端读写Embedding

Worker端示范代码

FMWorker类运行在每个Worker节点，负责利用FM算法训练本地数据，计算出参数梯度，汇报给Server。FMWorker的构造函数如代码3-6所示，其中构建了两个 KVWorker 的实例，你可以理解为两个操作KV数据库的客户端。其中 `kv_w` 的 `app_id=0`，与Server端的 `server_w_` 的 `app_id` 相同，负责各特征一阶权重的通信； `kv_v` 的 `app_id=1`，与Server端 `server_v_` 的 `app_id` 相同，负责各特征Embedding的通信。

```

FMWorker(.....)
{
    // app_id=0, 与Server端server_w_的app_id相同, 负责特征一阶权重的通信
    kv_w = new ps::KVWorker<float>(0);
    // app_id=1, 与Server端server_v_的app_id相同, 负责特征Embedding的通信
    kv_v = new ps::KVWorker<float>(1);
    .....
}

```

代码 3-6 FM Worker构造函数

Worker侧训练的入口在 `FMWorker::batch_training` 函数，多线程并发计算，如代码3-7所示。

```

void FMWorker::batch_training(ThreadPool *pool)
{
    ..... for (int epoch = 0; epoch < epochs; ++epoch) // 训练上若干epoch
    {
        xflow::LoadData train_data_loader(train_data_path, block_size << 20);

```

```

train_data = &(train_data_loader.m_data); // 用于存储读进来的一个mini-batch的数据

while (1) // 循环直到将本次的训练数据都读完
{
    // 读取一个mini-batch的数据存放在train_data->fea_matrix中
    // fea_matrix包含一个batch的样本，是一个vector<vector<kv>>
    // 外层vector代表各个样本，内层的vector代表一个样本中的各个feature
    train_data_loader.load_minibatch_hash_data_fread();

    // 把这个mini-batch的训练数据平分到各个thread上
    // 每个thread分到thread_size条训练数据
    int thread_size = train_data->fea_matrix.size() / core_num;
    gradient_thread_finish_num = core_num;

    for (int i = 0; i < core_num; ++i) // 遍历并启动各训练线程
    {
        int start = i * thread_size; // 当前线程分到的数据，在mini-batch中的起始位置
        int end = (i + 1) * thread_size; // 当前线程分到的数据，在mini-batch中的截止位置
        // 启动线程运行FMWorker::update，训练当前mini-batch[start:end]这部分局部数据
        pool->enqueue(std::bind(&FMWorker::update, this, start, end));
    }
    while (gradient_thread_finish_num > 0)
    { // 等待所有训练thread结束
        usleep(5);
    }
}
}
}

```

代码 3-7 FM Worker训练入口

在代码 3-7 中第24行被调用的update函数，如代码3-8所示。它运行在一个线程中，拿本地一部分数据进行前代回代，计算出对每个特征的一阶权重和Embedding的梯度，汇报给Server。

```

void FMWorker::update(int start, int end) // 运行在独立线程中，训练mini-batch[start:end]这部分数据
{
    size_t idx = 0;
    auto unique_keys = std::vector<ps::Key>();

    // **** 遍历分配给自己的数据，统计其中包含了哪些非零特征

```

```

for (int row = start; row < end; ++row)
{
    // sample_size是第row行样本内部包含的特征个数
    int sample_size = train_data->fea_matrix[row].size();

    for (int j = 0; j < sample_size; ++j) // 遍历当前样本的每个非零特征
    {
        idx = train_data->fea_matrix[row][j].fid;
        .....(unique_keys).push_back(idx); // idx就是feature id
    }
} // 遍历每条样本

// 把这部分数据中出现的所有feature_id去重
// 因为向ps server pull参数的时候，没必要重复pull相同的key
std::sort((unique_keys).begin(), (unique_keys).end());
(unique_keys).erase(unique((unique_keys).begin(), (unique_keys).end()), (unique_keys).end());
int keys_size = (unique_keys).size(); // 去重后的非零feature_id个数

// ***** 用去重后的非零feature_ids从ps server拉取最新的参数
// 拉取这部分样本所包含的非零特征的一阶权重 "w"
auto w = std::vector<float>();
kv_w->Wait(kv_w->Pull(unique_keys, &w));

// 拉取这部分样本所包含的非零特征的embedding "v"
auto v = std::vector<float>();
kv_v->Wait(kv_v->Pull(unique_keys, &v));

// ***** 前代
// Loss这个名字取得不好，其实里面存储的是每个样本loss->final logit的导数
auto loss = std::vector<float>(end - start);
..... calculate_loss(w, v, ..., unique_keys, start, end, ..., loss); // loss是计算结果

// ***** 回代
// push_w_gradient: 存放Loss对各feature的一阶权重'w'上的导数
auto push_w_gradient = std::vector<float>(keys_size);

// push_v_gradient: 存放Loss对各feature的embedding的每一位上的导数
// 由于embedding是个向量，
// 所以需要开辟的空间是keys_size(去重后有多少feaute)*v_dim_(每个embedding的长度)
auto push_v_gradient = std::vector<float>(keys_size * v_dim_);

```

```

// 计算结果输出至push_w_gradient和push_v_gradient
calculate_gradient(..., unique_keys, start, end, v, ..., loss,
                  push_w_gradient, push_v_gradient);

// **** 向Server push梯度, 让server端更新模型参数
// 注意! ! ! 这里的Wait只是等待异步Push完成
// 每个worker thread各push各自的, 完全没有与其他worker同步, 因此这里实现的还是异步模式
kv_w->Wait(kv_w->Push(unique_keys, push_w_gradient));
kv_v->Wait(kv_v->Push(unique_keys, push_v_gradient));

--gradient_thread_finish_num; // 表示有一个线程完成训练
}

```

代码 3-8 FM Worker中一个训练线程的业务逻辑

至此, 基于ps-lite实现分布式FM就演示完毕, 读者可以从中体会一下分布式训练推荐模型的基本流程。受篇幅所限, 更多的代码细节就不在这里展示了, 感兴趣的同学可以阅读参考文献[12]。

3.3.5 更先进的Parameter Server

前面已经提到, ps-lite只是Parameter Server的早期代表, 功能比较简单, 已经无法支持现代推荐模型那越来越复杂的网络结构。为此各互联网大厂纷纷研制开发自己的Parameter Server, 针对推荐系统的特点进行了专门优化, 功能越来越强大。本小节将介绍两款有特色的实现, 使读者能够对当今Parameter Server的发展现状有所了解。如今的Parameter Server越来越复杂, 每个单独拎出来, 都值得花上一章大书特书。受篇幅所限, 本书这里只是概括性地介绍它们针对推荐系统进行了哪些创新, 具体实现细节还请感兴趣的读者移步相关参考文献。

阿里XDL

XDL[8]是阿里开发的大规模、高性能、分布式深度学习平台。XDL的一个特色就是采用纯异步(ASP)并发的方式来训练模型。前面讲过, 得益于推荐系统的特征超级稀疏的特点, 一轮迭代中, 不同Worker节点上的训练数据所包含的非零特征较少重叠, 不太会出现多个Worker节点同时更新Server端同一个特征的Embedding的冲突情况。所以, 用纯异步的方式来更新Embedding参数是合理并且常见的。

但是, DNN权重是所有Worker节点都要参与更新的, 采用纯异步更新会导致冲突, 造成前面提到过的“梯度失效”问题, 可能会对模型收敛造成负面影响。对此, XDL认为这也并不是什么大问题, 失效的梯度可以理解为一种Dropout[8], 对于训练结果的影响并不大。所以, XDL还是推荐使用采用异步方式来训练模型。

XDL的一大创新是引入了“流水线”机制，从而大大加快了训练速度。训练一个推荐模型，从流程上可以划分为“读取训练数据”、“从Server拉取模型参数”、“Worker前代回代模型”这三个步骤（Stage）。传统上，这三个步骤是顺序执行的，比如“读数据”模块在读取了batch0的训练数据之后，必须等后面的“拉参数”和“前代回代”两模块都结束了，才能开始读取batch1的数据，如图3-14(a)和代码3-8所示。

XDL的作法是为每个步骤分配专门的线程池，并在步骤间引入队列（Queue）作为“流水线”，从而让多个步骤可以并发执行，如图3-14(b)所示。比如“读数据”模块在读取batch0的训练数据之后，只需将训练数据插入队列，就可以开始读取batch1的数据，而不必等待“拉参数”与“前代回代”也将batch0执行完毕。另外两个步骤也如是，从而充分利用了多核处理器的资源优势，大大提高了训练效率。

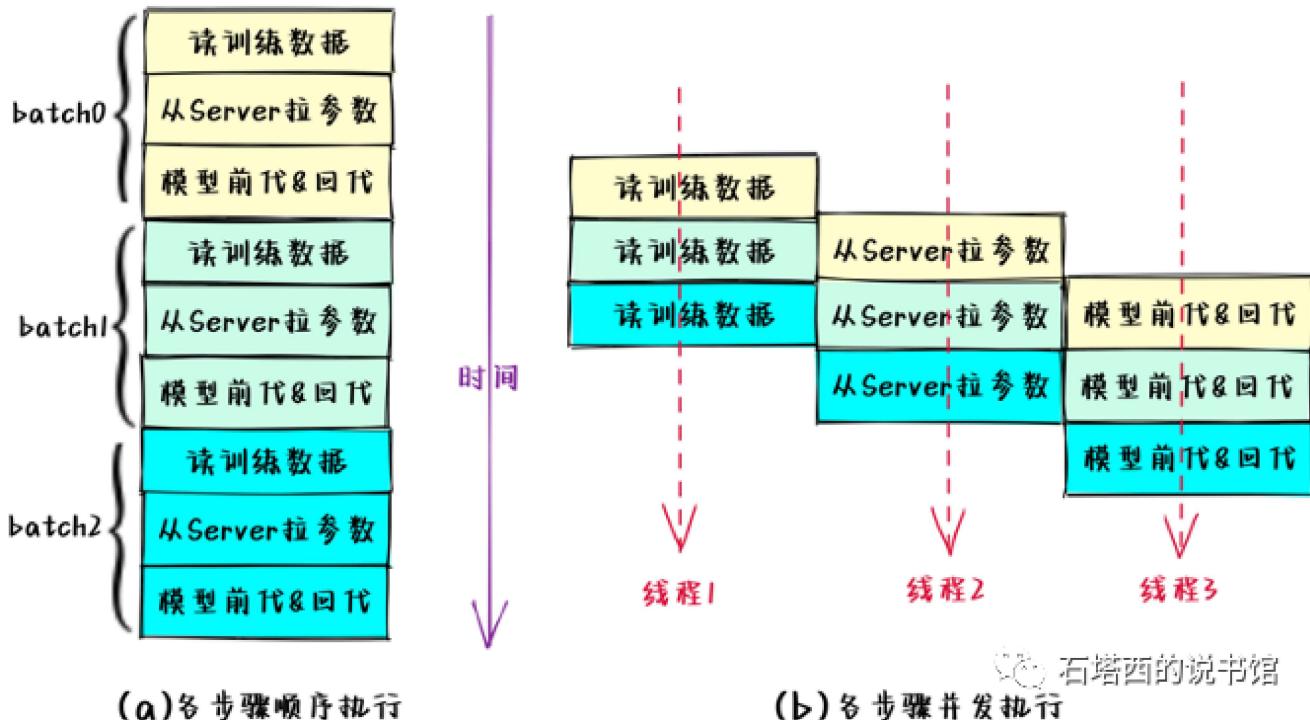


图 3-14 推荐训练过程中的顺序执行与并发执行

XDL的另一个创新是在PS的Server节点上也能够部署、训练模型。传统PS中的训练只发生在Worker节点上，Server节点相当于一个KV数据库，只负责模型参数的存储和更新，不涉及模型的前代回代。阿里遇到的问题是，他们的模型使用到了一些图片、视频之类的多媒体特征，而这些特征的Embedding都比较大，比如4096维。在Server与Worker之间频繁传递这么长的向量，将占用大量带宽，也导致很大的时间延迟。

对此，XDL的解决方案是[14]：

- 在每个Server节点部署一个可学习的压缩模型，其实就是一个简单的MLP。
- 当Worker向Server请求某个图片特征的Embedding时，Server提取出4096维的原始Embedding向量，经过Server本地的压缩模型，降低成12维的小向量，再回传给Worker节点，从而将网络通讯量减少到原来的 $\frac{1}{340}$ 。
- Server端这个压缩模型的参数，由每个Server利用存储在本地的图片，随主模型一起优化得到。在一轮迭代结束时，各个Server上的压缩模型也需要同步参数。

显然XDL中Server端的功能更加强大，超出了传统的功能范围，因此XDL将其命名为“先进模型服务器”（Advanced Model Server，AMS）。

快手的Persia

Persia[15]是快手和苏黎世联邦理工学院（ETH）联合开发的、针对推荐场景的、大规模分布式深度学习训练平台。Persia最引人注目的创新就是“混合”（Hybrid），即针对推荐模型中Embedding和DNN权重的不同特点，在训练时采取了不同的更新策略和通信策略，如图3-15所示。

- 对于模型Embedding的参数，由于推荐系统中的特征超级稀疏，多个Worker节点同时更新同一个特征的Embedding的可能性不大，因此Persia对Embedding采用“异步更新”（ASP）策略，以加快训练速度，而不必过分担心冲突。对Embedding的通信还是基于Parameter Server模式，即最新的Embedding保存在Server端，多个Worker节点通过Server作为中介，保证访问到的Embedding统一一致，都是及时更新过的。
- 对于DNN各层的权重，这些参数是所有Worker都要更新的，采用ASP模式肯定会导致冲突。为此，Persia采用同步模式（BSP）来更新DNN权重，也就是对DNN权重的更新必须等所有Worker节点都完成了回代、上报了梯度之后才能进行，避免出现“梯度失效”问题，从而保证模型的收敛精度。对DNN权重的通信，Persia采用了AllReduce[16]的通信模式。所谓AllReduce是一类高效的数据同步算法，让各Worker节点无须通过Server当中介，就能够保证各自的DNN权重都是统一一致、及时更新过的。

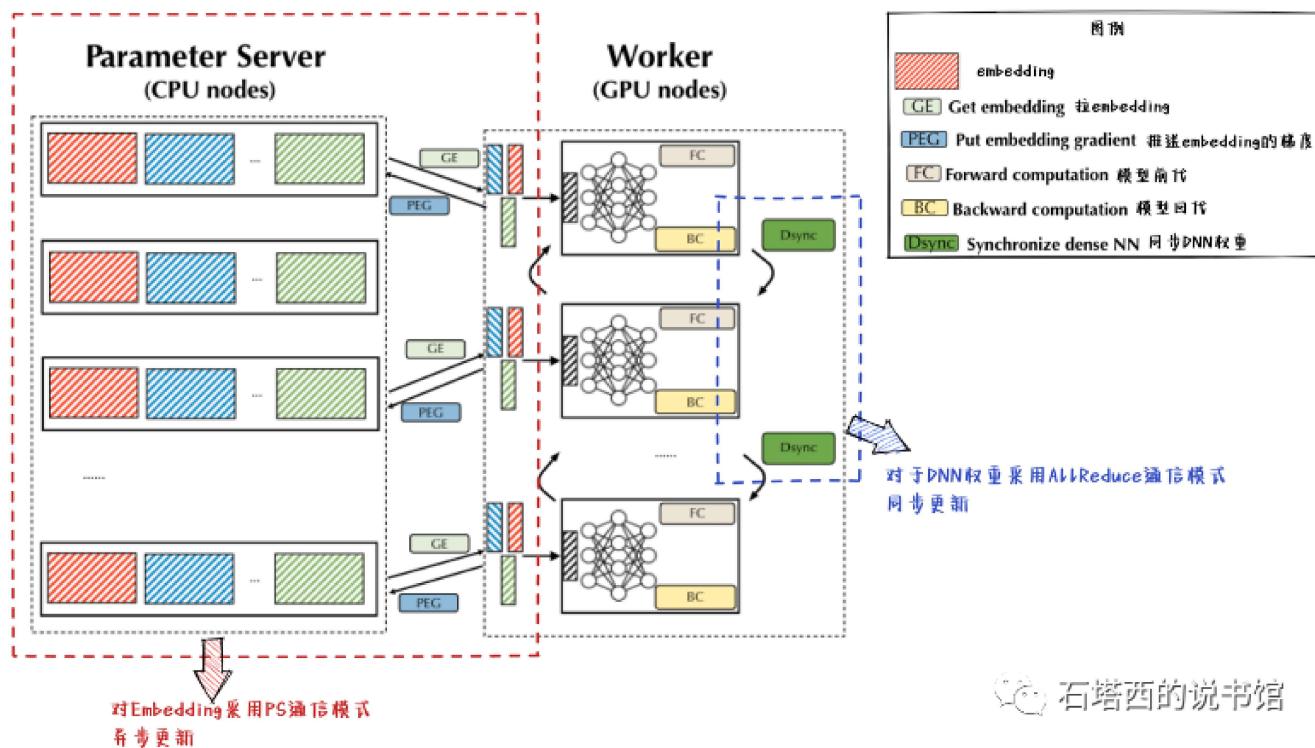


图 3-15 快手Persia平台的架构示意图

另外，Persia还优化了模型参数的存储空间。推荐模型为了提高推荐结果的个性化水平，大量采用了“用户标识”（User ID）和“物料标识”（Item ID）等细粒度特征，这就给Parameter Server如何存储这些特征的参数提出了挑战：

- 一来，现代大型推荐系统有上亿级别的用户和上百万、上千万的物料，本来就要耗费大量的存储空间。这个问题通过多个Server组成集群分布式存储，已经得到了很好的解决。
- 二来，每天都会有新用户登陆和新物料出现，如果它们的特征参数都保存进Parameter Server，再多资源也有耗尽的那一天。而且Parameter Server还为那些久未使用的“僵尸用户”和早就过期的老物料保存参数，也是对资源的一种浪费。

为了避免Server上的存储空间无限制膨胀下去，包括Persia在内的现代Parameter Server都要具备“特征准入”和“特征逐出”的功能：

- 特征准入：当PS第一次遇到某个特征时，接纳它并为它分配存储空间的概率是 p ，所以一个特征要平均出现 $\frac{1}{p}$ 次才能在Server端拥有一席之地。这么做的目的是避免为了只出现一两次的特征浪费存储空间。
- 特征逐出：如果有一个特征已经很久没有被更新了，就将这个特征的模型参数从Server端删除，以节省存储空间。

3.4 本章小结

本章聚焦于推荐算法中的核心概念Embedding：

- 3.1节，介绍了Embedding的技术的来龙去脉，指出Embedding的目的和优势就是提高推荐模型的扩展性，使其能够“举一反三”，自动挖掘出低频、长尾模式，更好地为用户提供个性化服务。同时，本书也在这一节演示了如何用Python实现稀疏特征的Embedding，向读者揭示了Embedding的实现细节。
- 3.2节，介绍了Embedding两大技术路线，“共享Embedding”和“独占Embedding”。共享Embedding的优点在于节省参数和训练数据，而独占Embedding的优点在于避免相互干扰。互联网大厂不缺少训练数据，因此“独占Embedding”在大厂实践中更为常见。
- 海量数据和高维稀疏的特征空间，增加了推荐模型的训练难度。为此，各互联网大厂都纷纷研制开发Parameter Server以提高训练效率。3.3节介绍了Parameter Server的结构、训练流程和并发策略，通过一个例子演示了基于PS的分布式推荐算法的开发过程，最后介绍了现代Parameter Server针对推荐系统特点所进行的一系列创新。

3.5 本章参考文献

[1] Matrix factorization (recommender systems)[Z/OL].

[https://en.wikipedia.org/wiki/Matrix_factorization_\(recommender_systems\)](https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems)).

[2] tf.keras.layers.Embedding[CP/OL]. Google.

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding.

[3] 石塔西. 用NumPy手工打造 Wide & Deep[EB/OL]. <https://zhuanlan.zhihu.com/p/53110408>.

[4] JUAN Y, ZHUANG Y, CHIN W S, 等. Field-aware Factorization Machines for CTR Prediction[C/OL]//Proceedings of the 10th ACM Conference on Recommender Systems. Boston Massachusetts USA: ACM, 2016: 43-50[2023-01-14]. <https://dl.acm.org/doi/10.1145/2959100.2959134>. DOI:10.1145/2959100.2959134.

[5] BIAN W, WU K, REN L, 等. CAN: Feature Co-Action for Click-Through Rate Prediction[J/OL].

arXiv:2011.05625 [cs, stat], 2021[2022-04-10]. <http://arxiv.org/abs/2011.05625>.

[6] LI M. Scaling Distributed Machine Learning with the Parameter Server[C/OL]//Proceedings of the 2014 International Conference on Big Data Science and Computing - BigDataScience '14. Beijing, China: ACM Press, 2014: 1-1[2022-04-13]. <http://dl.acm.org/citation.cfm?doid=2640087.2644155>. DOI:10.1145/2640087.2644155.

[7] MU LI. ps-lite[CP/OL]. <https://github.com/dmlc/ps-lite>.

[8] JIANG B, DENG C, YI H, 等. XDL: an industrial deep learning framework for high-dimensional sparse data[C/OL]//Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data. Anchorage Alaska: ACM, 2019: 1-9[2022-04-10].

<https://dl.acm.org/doi/10.1145/3326937.3341255>. DOI:10.1145/3326937.3341255.

[9] PaddlePaddle Fleet[CP/OL]. 百度.

https://www.paddlepaddle.org.cn/documentation/docs/zh/guides/06_distributed_training/cluster_quick_start_ps_cn.html.

[10] Angle[CP/OL]. 腾讯. <https://github.com/Angel-ML/angel>.

[11] Horovod[CP/OL]. Uber. <https://www.uber.com/en-TW/blog/horovod/>.

[12] 石塔西. ps-lite源码解析[EB/OL]. <https://mp.weixin.qq.com/s/sPV5wMlfzQePXyWIPSUv7A>.

[13] xflow[CP/OL]. <https://github.com/xswang/xflow>.

[14] GE T, ZHAO L, ZHOU G, 等. Image Matters: Visually modeling user behaviors using Advanced Model Server[M/OL]. arXiv, 2018[2023-01-16]. <http://arxiv.org/abs/1711.06505>. DOI:10.48550/arXiv.1711.06505.

[15] LIAN X, YUAN B, ZHU X, 等. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters[M/OL]. arXiv, 2021[2023-01-15]. <http://arxiv.org/abs/2111.05897>. DOI:10.48550/arXiv.2111.05897.

[16] MPI Reduce and Allreduce[CP/OL]. <https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>