

第4章 精排

之前的第2、3两章主要讨论的是推荐系统中的特征，即如何构建特征，和将特征喂入模型的主要形式 Embedding。从本章开始，我们开始讨论推荐算法模型。

4.1 推荐算法的五个维度

在介绍具体的模型之前，让我们先看看当今算法研究的现状。如今我们面临的问题，不是文章太少，而是文章太多，信息爆炸。每年KDD、SIGIR、CIKM上有那么多中外的王婆一起卖瓜，各种各样的DNN、GNN、FM、Attention满天飞，其中不乏实打实的干货，更不缺乏湿漉漉的灌水文，让人不知道哪个方法才是解决自己问题的灵丹妙药（当然抱着找银弹的想法来读论文，也稍微天真了一些）。

造成这种“永远追新、无所适从”的原因是，有些同学孤立地读论文，结果只能是一叶障目，只见树木，不见森林。正确的姿势应该是，梳理一门学问的脉络，然后在读论文的时候，就根据这个脉络分门别类，比如某篇文章到底是在哪个分支上进行了改进。等日后，你遇到实际问题，先拆解问题，再到问题涵盖的分支上去寻找合适的解决方案。只有这样，才能真正将各篇论文中的观点融汇贯通，读的论文越多，对学问掌握得越清晰，而不是“狗熊掰棒子”，读得越多反而越糊涂。

笔者根据个人经验将推荐算法模型梳理成如下5个维度。

第1维：记忆与扩展

“记忆”与“扩展”是推荐系统的两大永恒主题。推荐模型要能记住高频、常见模式以应对“红海”，而能够扩展发现低频、小众模式以开拓“蓝海”。这个观点，我们已经在第3.1节讲Embedding的前世今生的时候提到过了，本章第节将会讲到，经典的Wide & Deep模型就是“记忆与扩展”在模型设计中的体现。

第2维：Embedding

Embedding技术将推荐算法由“根据概念的精确匹配”（生搬硬套）升级为“基于向量的模糊查找”（举一反三），极大增强了模型的扩展能力，是推荐系统中所有深度学习模型的基础。这个观点，我们已经在第3.1.2节讨论过了。

第3维：高维稀疏的类别特征

高维稀疏的类别特征是推荐系统中的一等公民，充分认识这一数据特性，是理解推荐系统中很多技术的前提。比如：

- 为了加速对高维稀疏特征的训练，基于Parameter Server的分布式训练架构（见3.3节）应运而生，成为各互联网大厂的标配。
- 为了解决稀疏特征受训机会不均衡的问题，很多优化算法为每个特征采用不同的学习率和正则系数。
- 为了缓解单个类别特征表达能力弱的问题，我们采用Embedding来扩展其内涵，使用各种交叉结构扩展其外延。

第4维：交叉结构

正如上文提到的，我们需要设计交叉结构，使喂入模型的特征“融会贯通”，增强它们的表达能力。本章4.2节将讨论推荐模型中常见的交叉结构。除了大家耳熟能详的DNN这种隐式交叉方案，还有一阶、二阶、多阶等显式交叉方案，及这两类交叉方案的相互融合。

第5维：用户行为序列建模

推荐系统的核心任务是猜用户喜欢什么，而用户在APP内部的各种行为（e.g., 浏览、点击、点赞、评论、购买、观看、划过、……）组成的序列，隐藏着用户最真实的兴趣，是等待我们挖掘的宝藏。因此，基于用户行为序列的兴趣建模，是推荐模型的重中之重。

用户行为序列数据量大（短期行为几十、上百，长期行为成千上万），单个行为包含信息有限而且随机性大，将这些行为序列压缩成一个或几个固定长度的用户兴趣向量，绝非易事。本章4.3节将介绍建模用户短期/长期行为序列的几种经典模型。

4.2 交叉结构

4.2.1 FTRL：传统时代的记忆大师

在深度学习大行其道之前，推荐系统的精排环节主要依赖的就是Logistic Regression (LR)，一个说起来简单、实现起来又大有讲究的算法。

LR本身很简单，如公式(4-1)所示。

$$CTR_{predict} = \text{sigmoid}(\mathbf{w}^T \mathbf{x}) = \text{sigmoid}\left(\sum_i w_i x_i\right) \quad (4-1)$$

- x 是某条样本的所有特征组成的向量， x_i 表示其中的第*i*个特征。
- w 是权重向量，需要LR算法将其学习出来。 w_i 是对应特征 x_i 的权重。
- $CTR_{predict}$ 代表模型预测出来的点击率。

而我们在第2章中提到过，推荐系统中的特征以类别特别为主，所以 x_i 不是0就是1，因此LR公式可以再次简化为公式(4-2)：

$$CTR_{predict} = \text{sigmoid}\left(\sum_{j \in I} w_j\right), I = \{i | x_i = 1\} \quad (4-2)$$

- I 代表一条样本中所有非零特征的集合
- 其他符号含义参考公式(4-1)。

按照4.1节提到的"推荐算法5维度"，我们可以将LR做如下归纳：

- LR就是一个大的评分卡。它强于记忆，把每个特征 (e.g., 原始特征、或组合特征) 的重要性 (即权重) 都牢牢记住。LR在预测时，看当前用户、当前物料、当前场景命中了哪些特征，再把这些特征对应的权重相加求和，就得到了排序得分。整个模型预测过程就是一个提取记忆的过程。
- LR中每个特征只贡献一个权重，没有Embedding，缺乏"内涵"。LR在模型侧不会对特征进行交叉（但是可以通过人工设计交叉特征来弥补），缺乏"外延"。基于以上两点，LR的扩展性比较弱。

推荐系统对模型的技术要求

但是推荐系统中所使用的LR又不简单，因为所有推荐系统所使用的模型，都需要满足以下两条技术要求。

模型必须能够"在线学习"

在一些简单的机器学习场景下，当我们训练完一版模型并部署上线后，很长一段时间内就不再更新了。下次更新要等一天、一周甚至更久，然后再用这段时间间隔内收集到的数据重新训练一版新版模型，以替换线上老模型。

但是对于互联网大厂的推荐模型来说，天级、甚至小时级的更新都是不可接受的。在推荐系统中，用户与系统的交互非常频繁。模型需要根据用户对上一次推荐结果的反馈，快速调整自己：

- 上一次推荐错了，需要及时修正；
- 即使上次推荐对了，因为用户兴趣变化很快，模型也要能够及时察觉这种变化，并做出调整。

因此，互联网大厂的推荐模型都需要具备实时、在线学习（Online Learning）的能力，如图 4-1 所示：

1. 用户在APP前端的动作，触发后台服务向排序服务Ranker发一条请求，其中包括了当前用户的信息和当前所有候选物料的信息。
2. Ranker从当前用户与候选物料信息中提取出特征，喂给排序模型，模型给所有候选物料打分并排序，排序好的结果发往APP展现给用户。
3. 与此同时，抽取好的特征组成“特征快照”，发送给“拼接服务”Joiner。
4. 用户对第2步所展示结果进行反馈（e.g., 点击、观看、购买、……），反馈结果也发送给Joiner。
5. Joiner将对应同一条请求的“特征快照”和“用户反馈”拼接起来，组成一批新样本发往训练服务Trainer。
6. Trainer利用这批新样本，增量更新模型参数。
7. 更新后的模型参数，推送至Ranker，服务用户的下次请求。

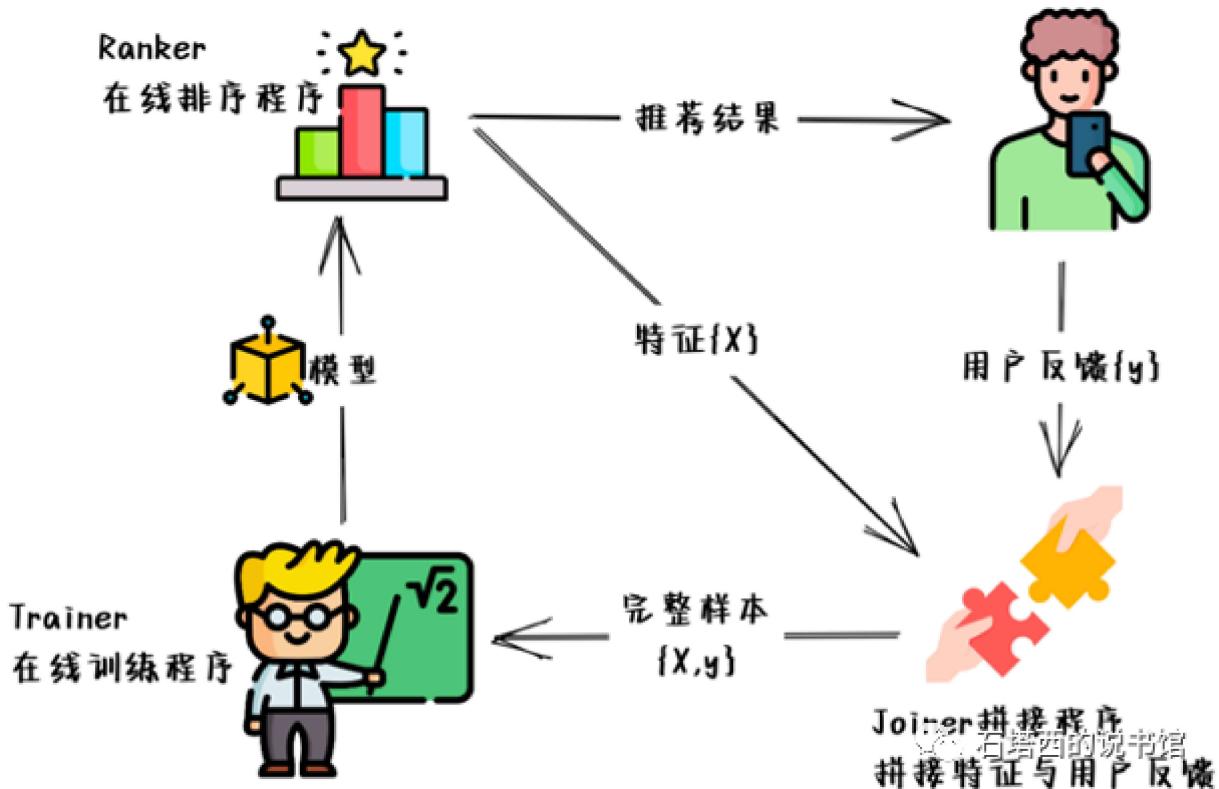


图 4-1 在线学习流程示意

LR很容易实现“在线学习”，因为它的经典解法Stochastic Gradient Descent (SGD) 天生就是支持增量更新的。当一次只输入一条样本时，SGD就变成了Online Gradient Descent (OGD)，如公式(4-3)所示。

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \mathbf{g}_t \quad (4-3)$$

- \mathbf{g}_t 是模型根据第 t 个样本计算出的梯度向量

- η_t 是遇到第 t 个样本时，模型决定采用的迭代步长
- \mathbf{w}_t 是模型对第 t 个样本前代所使用的权重向量
- \mathbf{W}_{t+1} 是模型被第 t 个样本回代后得到的新的权重向量。

模型参数需要尽可能稀疏

LR就是一个评分卡。推荐系统中的特征是高维稀疏的，上亿是平常，上十亿、上百亿也不新鲜。如果每个特征都在评分卡中占据一项存储权重参数，这么庞大的评分卡在存储、查询时都面临着巨大的性能压力。

因此，我们希望模型在保证预测精度的同时，它的输出（e.g., 权重或Embedding）也尽可能稀疏。如果某个特征 x_j 不重要，那么模型能够将它的权重 w_j 直接赋成0，而不是还保留一个非常接近0的小数来浪费空间。数值为0的权重被剔除出评分卡，“瘦身”成功后的评分卡在存储、查询时的性能都得到大幅提升。

LR的OGD解法，其预测精度还是不错的，但缺点在于它输出的权重不够稀疏。原因在于， \mathbf{g}_t 是根据单一样本计算出的梯度，随机性比较大，使得即使L1正则这种在批量输入时效果还不错的手段，在OGD环境下也很难找到足够稀疏的解。

除了OGD，业界还进行了很多尝试，比如TG[1]、FOBOS[2]、RDA[3]等，效果都不能令人满意，无法在“预测精度”与“稀疏解”这两方面取得很好的平衡。

FTRL算法原理

为了解决以上难题，Google花费三年时间（2010年~2013年），从理论推导到工程化实现FTRL（Follow The Regularized Leader）算法[4]。FTRL兼顾预测精度与解的稀疏性，性能出色，被各互联网大厂效仿采纳。

FTRL的前身是FTL（Follow The Leader）。FTL其实不单指某一个算法，而是Online Learning的一种思路，即：为了减少单个样本的随机扰动，第 t 步的最优参数，不是单单最小化第 t 步的损失，而是让之前所有步骤的损失之和最小。FTRL只不过在FTL的基础上添加了正则项，如公式(4-4)所示。

$$\mathbf{w}_{t+1} = \operatorname{argmin}_{\mathbf{w}} \left[\left(\sum_{s=1}^t Loss(s, \mathbf{w}) \right) + R(\mathbf{w}) \right] \text{tag4 - 4}$$

- \mathbf{w}_t 表示被第 t 条样本优化之后的模型权重。
- $Loss(s, \mathbf{w})$ 表示第 s 条样本在权重 \mathbf{w} 下计算得到的损失。
- $R(\mathbf{w})$ 表示对权重 \mathbf{w} 的正则项。

但是，公式(4-4)中的 $\sum_{i=1}^t Loss(i, \mathbf{w})$ 不容易求解。因此，我们需要引入代理损失函数。FTRL采用的代理损失函数如公式(4-5)所示：

$$\mathbf{w}_{t+1} = \operatorname{argmin}_{\mathbf{w}} \left[\mathbf{g}_{1:t} \cdot \mathbf{w} + \frac{1}{2} \left(\sum_{s=1}^t \sigma_s \|\mathbf{w} - \mathbf{w}_s\|_2^2 \right) + \lambda_1 \|\mathbf{w}\|_1 + \frac{1}{2} \lambda_2 \|\mathbf{w}\|_2^2 \right] \quad (4-5)$$

- $\mathbf{g}_{1:t} = \sum_{s=1}^t \mathbf{g}_s$, 其中 \mathbf{g}_s 表示由第 s 个样本回代得到的梯度向量, 所以 $\mathbf{g}_{1:t}$ 表示第1~ t 个样本的梯度向量之和。由于Online Learning的随机性较大, 使用累计梯度可以避免使用单样本梯度带来的较大抖动, 这也是FTL思想的体现。
- $\sigma_s = \frac{1}{\eta_s} - \frac{1}{\eta_{s-1}}$, 主要是为了推导公式方便而设立的。其中, η_s 代表第 s 步的步长。
- \mathbf{w}_s 代表前代第 s 个样本时, 模型所使用的权重。同时也表示被前 $s-1$ 个样本优化过后的权重。
- \mathbf{w} 代表模型被前 t 个样本优化后的权重, 是我们的求解目标。其优化结果就是 \mathbf{w}_{t+1} , 将被用于前代第 $t+1$ 个样本。
- $\sum_{s=1}^t \sigma_s \|\mathbf{w} - \mathbf{w}_s\|_2^2$ 一项, 相当于一个正则项, 要求当前正在被优化的权重 \mathbf{w} , 与它的每个历史版本 \mathbf{w}_s , 不能相距过远。这是为了防止模型只一味迎合新样本, 而损害对前边老样本的拟合能力。
- $\lambda_1 \|\mathbf{w}\|_1$ 是传统的L1正则, $\frac{1}{2} \lambda_2 \|\mathbf{w}\|_2^2$ 代表传统的L2正则。

由于篇幅所限, 本书中不详细列出的优化过程, 直接给出解, 如公式(4-6)~(4-7)所示。对推导过程感兴趣的同学, 可以参考文献[5]。

$$\mathbf{z}_{t,i} = \mathbf{g}_{1:t,i} - \sum_{s=1}^t \sigma_s \mathbf{w}_{s,i} \quad (4-6)$$

$$\mathbf{w}_{t+1,i} = \begin{cases} 0 & , |\mathbf{z}_{t,i}| \leq \lambda_1 \\ -\frac{1}{\lambda_2 + \sum_{s=1}^t \sigma_s} (\mathbf{z}_{t,i} - \operatorname{sgn}(\mathbf{z}_{t,i}) \times \lambda_1) & , |\mathbf{z}_{t,i}| > \lambda_1 \end{cases} \quad (4-7)$$

- 以上公式中的下标 t 表示第 t 步, 也就是模型针对第 t 个样本的训练过程。下标 i 表示一个向量的第 i 个元素。所以, $w_{t,i}$ 表示第 i 个特征在第 t 步时的权重。
- $\mathbf{z}_{t,i}$ 是为了推导方便而引入的中间变量。其中 σ_s 的定义参考公式(4-5)。
- 特别当 $|\mathbf{z}_{t,i}| \leq \lambda_1$ 时, 第 i 个特征的新权重 $w_{t+1,i}$ 会被直接设置成0, 这就是FTRL算法能够产生稀疏解的原因。
- sgn 代表符号函数。
- 其他符号的含义, 请参考公式(4-5)。

另外, 因为推荐系统中特征超级稀疏, 不同特征在训练数据中的分布不均, 导致不同特征受训的机会严重不均等。为此, FTRL放弃使用统一的步长, 而是为每个特征独立设置步长 (Per-Coordinate Learning Rate), 如公式(4-8)所示。

$$\eta_{t,i} = \frac{\alpha}{\beta + \sqrt{\sum_{s=1}^t \mathbf{g}_{s,i}^2}} \quad (4-8)$$

- $g_{s,i}$ 代表由第 s 个样本计算出来的、针对第 i 个特征的梯度。
- 频繁出现的特征, 很多样本都贡献了对它的梯度, $\sum_s g_{s,i}^2$ 累积得比较大, 因此步长 η_i 会小一些。不会导致已经学得很好的 w_i 剧烈变化;

- 较少出现的特征，之前积累的 $\sum_s g_{s,i}^2$ 还比较少，步长 η_i 会大一些。因为对超级稀疏的特征，每个样本都非常珍贵，大步长有利于对这些样本的应用。
- α 和 β 是两个超参数。

将公式(4-8)代入公式(4-7)，最终FTRL中对每个特征的权重 w_i 的迭代公式如公式(4-9)所示：

$$w_{t+1,i} = \begin{cases} 0 & , |z_{t,i}| \leq \lambda_1 \\ -\left(\lambda_2 + \frac{\beta + \sqrt{\sum_{s=1}^t g_{s,i}^2}}{\alpha}\right)^{-1} (z_{t,i} - \text{sgn}(z_{t,i}) \times \lambda_1) & , |z_{t,i}| > \lambda_1 \end{cases}$$

基于FTRL的CTR预估程序的伪代码实现如代码 4-1所示。

代码 4-1 基于FTRL的CTR预估伪码实现

Algorithm: FTRL for CTR prediction

```

1. Input: hyper-parameters  $\alpha, \beta, \lambda_1, \lambda_2$ 
2. for  $i = 1$  to  $d$  do // 一共有  $d$  个特征
3.    $z_i = 0, d_i = 0$  // 为每个特征初始化中间状态变量  $z_i$  和  $n_i$ 
4. end for
5. for  $t = 1$  to  $T$  do // 遍历所有样本， $T$  是样本总数
6.   receive feature vector  $x_t$  for  $t$ -th sample
7.   let  $I = \{i \mid x_i \neq 0\}$  // 第  $t$  条样本中非零特征组成的集合
8.
9.   for all  $i \in I$  do // 利用中间状态变量，为每个特征计算出最新权重
10.     $w_{t,i} = \begin{cases} 0 & , |z_{t,i}| \leq \lambda_1 \\ -\left(\lambda_2 + \frac{\beta + \sqrt{n_i}}{\alpha}\right)^{-1} (z_{t,i} - \text{sgn}(z_{t,i}) \times \lambda_1) & , |z_{t,i}| > \lambda_1 \end{cases}$ 
11.   end for
12.
13.    $p_t = \text{sigmoid}(w_t \cdot x_t)$  // 前代得到第  $t$  条样本的打分  $p_t$ 
14.   observe label  $y_t \in \{0,1\}$  // 得到了用户反馈
15.
16.   for all  $i \in I$  do // 开始回代
17.      $g_i = (p_t - y_t)x_i$  // 针对第  $i$  个特征对应权重的梯度
18.      $\sigma_i = \frac{\sqrt{n_i + g_i^2} - \sqrt{n_i}}{\alpha}$  // 参考公式 x
19.      $z_i = z_i + g_i - \sigma_i w_{t,i}$  // 更新每个特征的状态
20.      $n_i = n_i + g_i^2$  // 更新每个特征的状态
21.   end for
22. end for

```

石塔西的说书馆

用Python实现FTRL

作者基于Python实现了FTRL算法，如代码 4-2所示。请读者结合代码4-1中的伪码，对照学习，加深理解。

代码 4-2 基于Python实现的FTRL

```
class FtrlEstimator:  
    """  
    每个field对应一个FtrlEstimator。类比于在TensorFlow WDL中，一个feature column对应一个FtrlEstimator  
    """  
  
    def __init__(self, alpha, beta, L1, L2):  
        self._alpha = alpha # 用于调节步长的超参  
        self._beta = beta # 用于调节步长的超参  
        self._L1 = L1 # L1正则系数  
        self._L2 = L2 # L2正则系数  
  
        self._n = defaultdict(float) # n[i]: i-th feature's squared sum of past gradients  
        self._z = defaultdict(float)  
  
        # lazy weights, 实际上是一个临时变量，只在：  
        # 1. 对应的feature value != 0, 并且  
        # 2. 之前累积的abs(z) > L1  
        # 两种情况都满足时，w才在feature id对应的位置上存储一个值  
        # 而且w中数据的存储周期，只在一次前代、后代之间，在新的前代开始之前，就清空上次的w  
        self._w = {}  
  
        self._current_feat_ids = None  
        self._current_feat_vals = None  
  
    def predict_logit(self, feature_ids, feature_values):  
        """ 前代过程  
        :param feature_ids: non-zero feature ids for one example  
        :param feature_values: non-zero feature values for one example  
        :return: logit for this example, i.e., wTx  
        """  
        self._current_feat_ids = feature_ids  
        self._current_feat_vals = feature_values  
  
        logit = 0  
        self._w.clear() # lazy weights, 所以没有必要保留之前的weights  
  
        # 如果当前样本在这个field下所有feature都为0，则feature_ids==feature_values==[]
```

```

# 则没有以下循环, Logit=0

for feat_id, feat_val in zip(feature_ids, feature_values):
    z = self._z[feat_id]
    sign_z = -1 if z < 0 else 1.

    # build w on the fly using z and n, hence the name - Lazy weights
    # this allows us for not storing the complete w
    # if abs(z) <= self._L1: self._w[feat_id] = 0. # w[i] vanishes due to L1 regularization
    if abs(z) > self._L1:

        # apply prediction time L1, L2 regularization to z and get w
        w = (sign_z * self._L1 - z) / ((self._beta + np.sqrt(self._n[feat_id])) / self._alpha)
        self._w[feat_id] = w
        logit += w * feat_val

return logit


def update(self, pred_proba, label):
    """
    回代过程
    :param pred_proba: 与last_feat_ids/last_feat_vals对应的预测CTR
    注意pred_proba并不一定等于sigmoid(predict_logit(...)), 因为要还要考虑deep侧贡献的logit
    :param label:      与last_feat_ids/last_feat_vals对应的true label
    """

    grad2logit = pred_proba - label

    # 如果当前样本在这个field下所有feature都为0, 则没有以下循环, 没有更新
    for feat_id, feat_val in zip(self._current_feat_ids, self._current_feat_vals):
        g = grad2logit * feat_val
        g2 = g * g
        n = self._n[feat_id]

        self._z[feat_id] += g

        if feat_id in self._w: # if self._w[feat_id] != 0
            sigma = (np.sqrt(n + g2) - np.sqrt(n)) / self._alpha
            self._z[feat_id] -= sigma * self._w[feat_id]

        self._n[feat_id] = n + g2

```

4.2.2 FM: 半脚迈入DNN的门槛

FM的前身就是LR，担心只包含一阶特征表达能力弱，而加入了二阶特征交叉，见公式(4-10)。

$$\logit_{FM} = b + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j \quad (4-10)$$

- x_i 代表输入样本中第*i*个特征对应的特征值。
- b 代表要学习的偏置（bias）项。
- w_i 代表一阶特征的权重。
- w_{ij} 代表二阶特征交叉的权重。

但是，引入二阶特征交叉后，也增加了训练难度：

- 假设一共有*n*个特征，所有特征之间两两交叉，只 w_{ij} 一项就引入了 n^2 个要训练的参数。而要将这么多的参数训练好，就需要更多的数据，否则就容易过拟合（overfitting）。
- 看公式x可知，只有遇到 x_i 和 x_j 都不为0的样本， w_{ij} 才得到一次训练机会。但是前面我们说了，推荐系统的一大特点就是类别特征高维稀疏，符合条件的样本少之又少，导致 w_{ij} 得不到充分训练。

为了解决以上难题，业界提出了FM算法[6]，见公式(4-11)。

$$\logit_{FM} = b + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n (v_i \cdot v_j) x_i x_j \quad (4-11)$$

- 模型对每个特征，除了要学习它的一阶权重 w_i ，还要再多学习一个Embedding，也就是公式x中的 v_i 。
- 而 x_i 和 x_j 共现特征之前的权重 w_{ij} 等于对应特征的Embedding的点积，即 $w_{ij} = v_i \cdot v_j$ 。

FM的想法虽然简单，但作用巨大

- 要学习的参数由 n^2 变成了 nk ，I是每个特征Embedding的长度，大大减少了学习的参数量。
- 只要 $x_i \neq 0$ 的样本都能训练 v_i ， $x_j \neq 0$ 的样本都能训练 v_j ，也就都间接训练了 w_{ij} 。数据利用率更高，训练更加充分。

除了使训练变得更容易，FM还提升了模型的"扩展性"：

- 如果还使用"手动二阶交叉的LR"（见公式(4-10)），对于不曾在训练样本中出现过特征组合 $x_i x_j$ ，LR对这一特征对前面的权重 w_{ij} 无从学起，只能赋成0，也就是剥夺了小众模式发挥作用的机会。
- 如果用FM模型，虽然 $x_i x_j$ 这种组合从来没有在训练样本中共现过，但是 x_i 和 x_j 在训练样本中都单独出现，因此模型训练好了 v_i 和 v_j 。因此在预测时，FM能够预估出 $w_{ij} = v_i \cdot v_j$ 从而给小众模式提供了一个发挥作用的机会。

观察公式(4-11)，我们发现二阶交叉的部分仍然是 $O(n^2)$ 的复杂度，并不实用。而且实际场景中以稀疏类别特征为主， x_i 、 x_j 不是0就是1。因此，公式x可以等价变换为更加实用的公式(4-12)：

$$\begin{aligned}
logit_{FM} &= b + \sum_{i \in I} w_i + \sum_{i \in I} \sum_{j=i+1} v_i \cdot v_j \\
&= b + \sum_{i \in I} w_i + \text{reducesum} \left(\sum_{i \in I} \sum_{j=i+1} v_i \odot v_j \right) \\
&= b + \sum_{i \in I} w_i + \frac{1}{2} \text{reducesum} \left[\left(\sum_{i \in I} v_i \right)^2 - \sum_{i \in I} v_i^2 \right]
\end{aligned} \tag{4-12}$$

- 公式(4-12)中的 \odot 代表两个向量按位相乘。
- reducesum** 表示将一个向量所有位置上的元素相加。
- I** 是某个样本中所有非零特征的集合。

相比公式(4-11)的平方级的复杂度，公式(4-12)拥有线性复杂度，而且只需要遍历样本中的非零特征。得益于推荐系统中的特征超级稀疏，一个样本包含的非零特征非常有限，因此根据公式(4-12)训练与预测FM的速度都非常快。

FM的经典实现请参考alphaFM[7]，由于本书篇幅所限，这里就不讲解它的代码了。alphaFM在前代时遵循公式(4-12)，我们将在4.2.4节演示如何用TensorFlow来实现。alphaFM在回代更新w和Embedding时，遵循FTRL算法，我们已经在4.2.1节用Python实现过了，请读者前往相关章节学习。

按照之前提到的"推荐算法5维度"，FM为每个特征引入了Embedding，还引入了允许所有特征进行自动二阶交叉的结构，这两个措施大大提升了模型的"扩展性"。因为Embedding和交叉，所以我说FM已经把半只脚迈进了"深度学习"的大门。另外，除了用于精排环节，FM还能应用于召回和粗排，可谓"推荐模型界的瑞士军刀"，我们在第5.4节还会再提到它。

4.2.3 Wide & Deep：兼顾记忆与扩展

如前文3.1节所述，"记忆与扩展"是推荐系统面临的两大永恒主题，需要推荐系统的各个环节携手解决。Google在2016年发表的经典模型Wide & Deep[8]，正是在模型设计上对这两大主题的回应。Wide & Deep深深影响了模型设计思路的发展，之后的DeepFM、DCN身上都能看到Wide & Deep的影子。

Wide & Deep算法原理

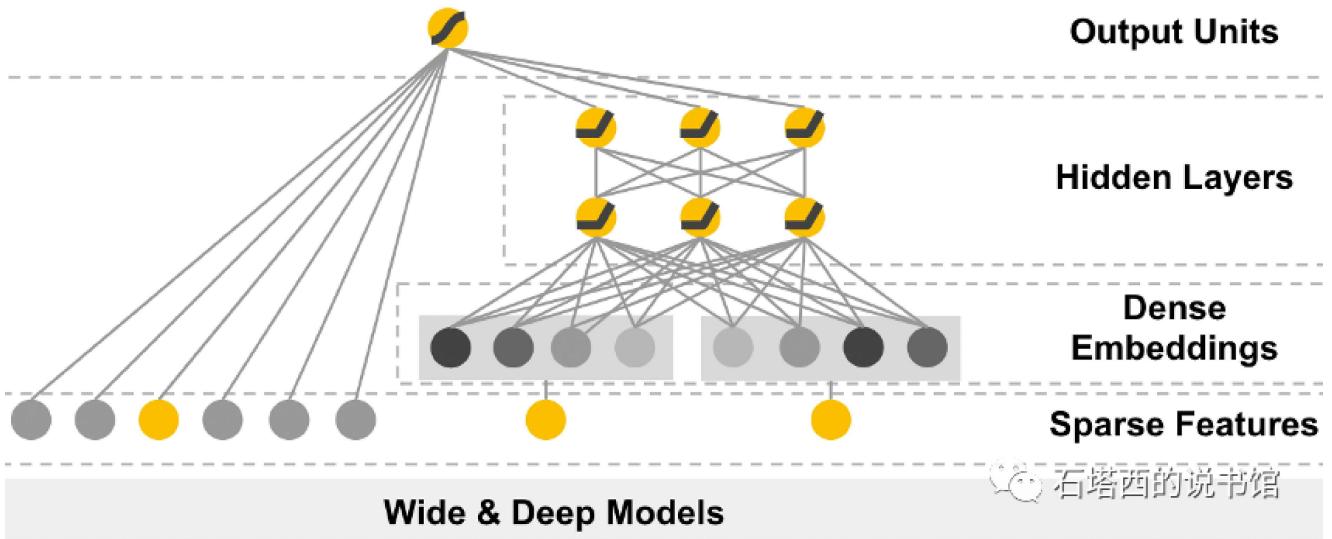


图 4-2 Wide & Deep网络结构图

石塔西的说书馆

Deep侧是一个DNN

Deep侧遵循推荐模型的经典设计范式：Embedding + MLP，可以简单描述成公式(4-13)。

$$logit_{dnn} = DNN(Concat(Embedding(\mathbf{x}_{deep}))) \quad (4-13)$$

- \mathbf{x}_{deep} 代表要输入Deep侧的那些特征。
- Embedding代表将稀疏特征映射成稠密向量的操作。其最底层是若干Embedding Layer。每个Field对应一个Embedding Layer，负责将每个Field内部的一个或多个Feature映射成一个固定长度的稠密浮点数向量。如果某个Field里面又包含多个Feature，比如某个物料的“物料标签”这个Field里面包含多个标签作为Feature，那么这个Field的Embedding就是那多个Feature Embedding“池化”（Pooling）后的结果。Pooling可以有多种选择，比如sum pooling, average pooling, max pooling等。Embedding的具体实现细节，可以参考3.1.4一节的Python代码。
- Concat代表将各个Field输出的Embedding，拼接成一个大向量。
- 拼接好的向量喂入上层的DNN。DNN的最后一层输出 $logit_{dnn}$ ，代表Deep侧的预估结果。

Deep侧对类别特征使用了Embedding以扩展它们的内涵，再加上DNN对特征进行高阶隐式交叉，大大增强了模型的“扩展性”和推荐系统的“多样性”，有利于满足低频、小众、个性化的用户需求。

Wide侧是一个LR

Wide侧其实就是一个LR，可以简写成如下公式(4-14)：

$$logit_{wide} = \mathbf{w}_{wide} \cdot \mathbf{x}_{wide} \quad (4-14)$$

- \mathbf{x}_{wide} 是要喂入Wide侧的特征。

- w_{wide} 是Wide侧那个LR的权重。

- $logit_{wide}$ 是Wide侧的预测结果。

Wide侧发挥LR"强于记忆"的优势，把那些在训练数据中高频、大众的模式牢牢记住。此外，Wide侧的另一个作用是防止Deep侧过分扩展而影响预测精度，起到一个类似"正则化"的作用。

基于以上两点考虑，Wide & Deep中Deep侧是主力，Wide侧主要起到一个"查漏补缺"的助攻作用。因此，Wide侧的LR不必像单独使用时那样大而全，喂入其中的都是一些被先验知识认定非常重要的精华特征。

- 主要是一些人工设计的交叉、共现特征，比如"用户喜欢军事类，而当前视频的标签是坦克"。
- 另外，影响推荐系统的bias特征，比如position bias，只能喂入Wide侧，避免与其他真实特征交叉。这么做原因，已经在2.2.4节讨论过了。

Wide & Deep共同训练

模型最终的预测结果是： $CTR_{predict} = \text{sigmoid}(logit_{wide} + logit_{deep})$ 。

训练的时候，Wide侧与Deep侧一同优化：

- 为了保证Wide侧解的稀疏性，Wide侧一般采用FTRL优化器
- Deep侧采用DNN的常规优化器，比如Adagrad、Adam等。

Wide & Deep源码解析

TensorFlow2中自带对Wide & Deep的实现[9]。关键代码和注释如代码4-3所示。

代码 4-3 TensorFlow自带Wide & Deep实现

```
class WideDeepModel(keras_training.Model):

    def call(self, inputs, training=None):
        linear_inputs, dnn_inputs = inputs

        # Wide部分前代，得到Logit
        linear_output = self.linear_model(linear_inputs)
        # Deep部分前代，得到Logit
        dnn_output = self.dnn_model(dnn_inputs)

        # Wide Logits与Deep Logits相加
        output = tf.nest.map_structure(
```

```

    lambda x, y: (x + y), linear_output, dnn_output)

# 一般采用sigmoid激活函数, 由logit得到ctr
return tf.nest.map_structure(self.activation, output)

def train_step(self, data):
    x, y, sample_weight = data_adapter.unpack_x_y_sample_weight(data)

    # ----- 前代
    # GradientTape是TF2自带功能, GradientTape内的操作能够自动求导
    with tf.GradientTape() as tape:
        y_pred = self(x, training=True) # 前代
        # 由外界设置的compiled_loss计算loss
        loss = self.compiled_loss(
            y, y_pred, sample_weight, regularization_losses=self.losses)

    # ----- 回代
    linear_vars = self.linear_model.trainable_variables # Wide部分的待优化参数
    dnn_vars = self.dnn_model.trainable_variables # Deep部分的待优化参数

    # 分别计算loss对linear_vars的导数linear_grads
    # 和loss对dnn_vars的导数dnn_grads
    linear_grads, dnn_grads = tape.gradient(loss, (linear_vars, dnn_vars))

    # 一般用FTRL优化Wide侧, 以得到更稀疏的解
    linear_optimizer = self.optimizer[0]
    linear_optimizer.apply_gradients(zip(linear_grads, linear_vars))

    # 用Adam、Adagrad优化Deep侧
    dnn_optimizer = self.optimizer[1]
    dnn_optimizer.apply_gradients(zip(dnn_grads, dnn_vars))

```

4.2.4 DeepFM: 融合二阶交叉

DeepFM算法原理

前边讲过了, Wide & Deep中的Wide侧是一个LR, 主要喂入一些手工设计的交叉特征, 耗费精力并严重依赖工程师的经验。为了解决这一问题, 华为提出在Wide侧增加FM[10], 自动进行二阶特征交叉。

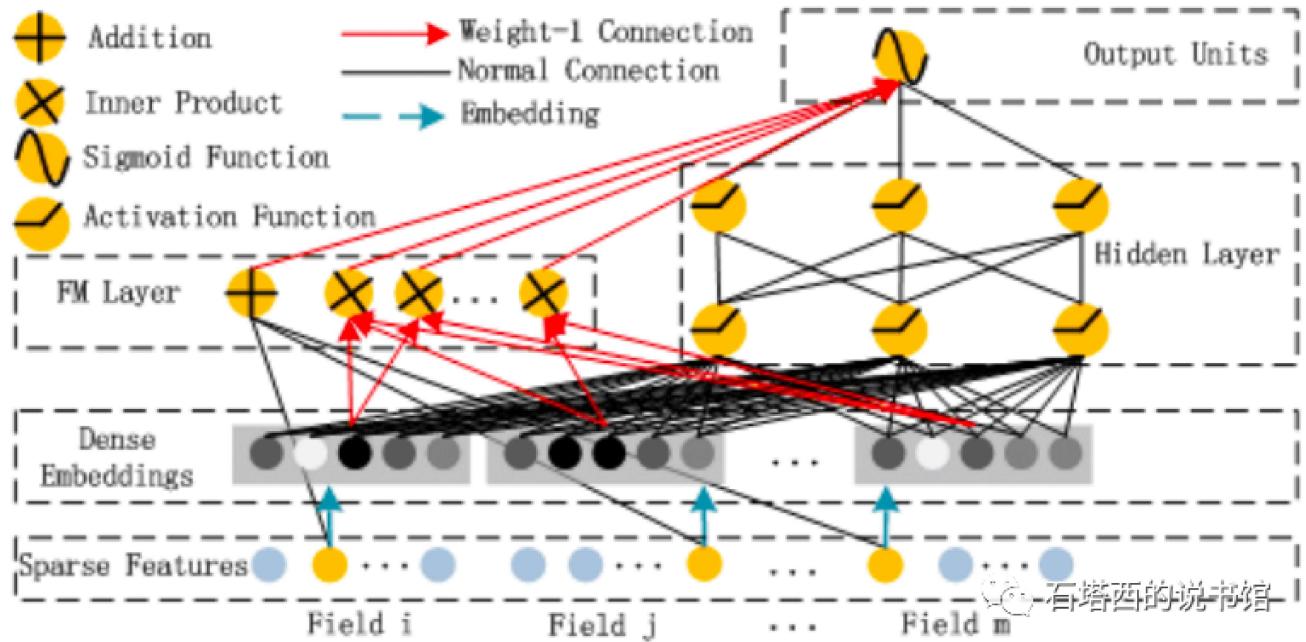


图 4-3 DeepFM结构图

DeepFM的原理可以简写成如下公式(4-15):

$$\begin{aligned}
 logit_{dnn} &= DNN(Concat(Embedding(\mathbf{x}_{dnn}))) & (a) \\
 logit_{fm} &= FM(\mathbf{x}_{fm}, Embedding(\mathbf{x}_{fm})) & (b) \\
 logit_{lr} &= \mathbf{w}_{lr} \cdot \mathbf{x}_{lr} & (c) \\
 CTR_{predict} &= sigmoid(logit_{lr} + logit_{fm} + logit_{dnn}) & (d)
 \end{aligned} \tag{4-15}$$

- 公式(4-15)(a)表示DeepFM中Deep侧的建模过程， \mathbf{x}_{dnn} 是输入的特征， $logit_{dnn}$ 代表对特征进行高阶隐式交叉的预测结果。
- 公式(4-15)(b)表示DeepFM中FM侧的建模过程。 \mathbf{x}_{fm} 是输入的特征，一般情况下和喂给DNN的 \mathbf{x}_{dnn} 相同。 $logit_{fm}$ 代表对特征进行二阶交叉的预测结果。
- (a)和(b)两个子公式都要进行Embedding。华为原论文中，FM部分与DNN部分共享Embedding。这样做有一个缺点，就是FM要求所有特征的Embedding的长度都必须相同。这个要求太死板，而且不同模块共享Embedding也的确有可能相互干扰，因此在实现时可以让FM部分与DNN部分各自拥有独立的Embedding。
- 公式(4-15) (c)和Wide & Deep中的Wide侧一样，就是一个LR。目的是发挥LR"强于记忆"的优势，牢牢记住一些高频、大众的模式，也起到防止Deep侧与FM侧过分扩展的正则化作用。 \mathbf{x}_{lr} 作为输入特征，主要包括那些被先验认定为重要的特征，比如position bias。
- 公式(4-15)中的其他符号参考公式(4-13)。

DeepFM源码演示

笔者用TensorFlow实现了支持Multi-Hot Encoding、稀疏、共享权重的DeepFM[11]。由于篇幅所限，这里只列出并注释关键部分，即用TensorFlow实现的FM。代码的其他部分，请读者参考文献[11]。代码4-4中的

FM公式请参考公式(4-12)。

代码 4-4 TensorFlow实现FM

```
def output_logits_from_bi_interaction(features, embedding_table, params):
    # 见《Neural Factorization Machines for Sparse Predictive Analytics》论文的公式(4)
    fields_embeddings = [] # 每个field的embedding, 是每个field所包含的feature embedding的和
    fields_squared_embeddings = [] # 每个元素, 是当前field所有feature embedding的平方的和

    for fieldname, vocabname in field2vocab_mapping.items():
        sp_ids = features[fieldname + "_ids"] # 当前field下所有稀疏特征的feature id
        sp_values = features[fieldname + "_values"] # 当前field下所有稀疏特征对应的值

        # ----- embedding
        embed_weights = embedding_table.get_embed_weights(vocabname) # 得到embedding矩阵
        # 当前field下所有feature embedding求和
        # embedding: [batch_size, embed_dim]
        embedding = embedding_ops.safe_embedding_lookup_sparse(
            embed_weights, sp_ids, sp_values,
            combiner='sum',
            name='{}_embedding'.format(fieldname))
        fields_embeddings.append(embedding)

        # ----- square of embedding
        squared_emb_weights = tf.square(embed_weights) # embedding矩阵求平方
        # 稀疏特征的值求平方
        squared_sp_values = tf.SparseTensor(indices=sp_values.indices,
                                             values=tf.square(sp_values.values),
                                             dense_shape=sp_values.dense_shape)

        # 当前field下所有feature embedding的平方的和
        # squared_embedding: [batch_size, embed_dim]
        squared_embedding = embedding_ops.safe_embedding_lookup_sparse(
            squared_emb_weights, sp_ids, squared_sp_values,
            combiner='sum',
            name='{}_squared_embedding'.format(fieldname))
        fields_squared_embeddings.append(squared_embedding)

    # 所有feature embedding, 先求和, 再平方
    sum_embedding_then_square = tf.square(tf.add_n(fields_embeddings)) # [batch_size, embed_dim]
    # 所有feature embedding 先平方 再求和
```

```

    " / / / F M C o m b i n i n g , S U I T S , T R A Y H

square_embedding_then_sum = tf.add_n(fields_squared_embeddings) # [batch_size, embed_dim]

# 所有特征两两交叉的结果, 形状是[batch_size, embed_dim]

bi_interaction = 0.5 * (sum_embedding_then_square - square_embedding_then_sum)

# 由FM部分贡献的Logits

logits = tf.layers.dense(bi_interaction, units=1, use_bias=True, activation=None)

# 因为FM与DNN共享embedding, 所以除了Logits, 还返回各field的embedding, 方便搭建DNN

return logits, fields_embeddings

```

4.2.5 DCN：不再迷信DNN

显式交叉 vs. 隐式交叉

业界流传着这样的说法，"DNN是万能函数模拟器，只要网络层数足够多，每层足够宽，DNN能够模拟任何函数"。这一优点对于严重依赖特征交叉的推荐模型是非常有吸引力的。这意味着，DNN能够让喂入的特征产生任意高阶的交叉。由于DNN各层非线性激活函数的使用，DNN产生的交叉是无穷高阶的，是无法明确写出一个多项式的。因此，我们称DNN产生的交叉为"隐式交叉"。

但是随着研究的日渐深入，笼罩在DNN头上的"万能函数模拟器"的光环也渐渐退去。有研究表明[12]，某些场景下，DNN甚至连2阶或3阶特征交叉都模拟不好。笔者对于这种结论一点也不奇怪，否则业界也就不必挖空心思研究各种初始化方法、或者Residual Network这样的结构了。所谓的"万能函数模拟"，只是一种理论假设。真要实现，恐怕需要非常宽与深的网络，但是各种梯度消失、梯度爆炸的问题也都随之而来。而且即便这种"宽且深"DNN训练出来，线上推理的性能也会成为一个大问题。

因此，在模型设计领域出现一种思路：实现特征交叉，只依靠DNN这种隐式交叉是远远不够的，还应该加上显式、指定阶数的交叉作为补充。

- 4.2.1节介绍的Wide & Deep就是用一阶LR作为Deep侧的补充。
- 4.2.2节介绍的DeepFM用FM实现二阶交叉作为Deep侧的补充。
- 本节介绍DCN，可以任意指定阶数的显式交叉，补充Deep侧。

DCN算法原理

DCN (Deep & Cross Network) [12]是Google提出的交叉结构，目前已经发展出了两个版本。

DCN V1

DCN V1中，每个Cross Layer的前代公式如公式(4-16)所示：

$$\mathbf{x}_{l+1} = \mathbf{x}_0 \mathbf{x}_l^T \mathbf{w}_l + \mathbf{b}_l + \mathbf{x}_l \quad (4-16)$$

- \mathbf{x}_0 是最底下一层Cross Layer的输入，由各种类别特征的Embedding和稠密特征拼接而成，是一个长度为 d 的稠密浮点数向量。
- $\mathbf{x}_l, \mathbf{x}_{l+1}$ 是第 l 个Cross Layer的输入与输出。
- $\mathbf{w}_l, \mathbf{b}_l$ 是第 l 个Cross Layer要学习的参数。
- $\mathbf{x}_0, \mathbf{x}_l, \mathbf{x}_{l+1}, \mathbf{w}_l, \mathbf{b}_l$ 都是长度为 d 的向量。

对于一个包含 L 个Cross Layer的Cross Network，其最终结果包含了原始输入 $\mathbf{x}_0 = [f_1, f_2, \dots, f_d]$ 所有 d 个元素之间小于等于 $L+1$ 阶的交叉，即包含了所有 $f_1^{\alpha_1} f_2^{\alpha_2} \dots f_d^{\alpha_d}$ 的可能组合，并且 $0 \leq \sum_{i=1}^d \alpha_i \leq L + 1$ 。

DCN V2

在第2个版本中，DCN的作者认为V1版本中，每层要学习的参数只有 $\mathbf{w}_l, \mathbf{b}_l$ 两个 d 维向量，参数容量有限，限制了模型的表达能力。为此，在V2版本中，用一个 $d \times d$ 的矩阵 \mathbf{W}_l 代替了V1中的 d 维向量 \mathbf{w}_l 。DCN v2中每层Cross Layer的前代如(4-17)所示。

$$\mathbf{x}_{l+1} = \mathbf{x}_0 \odot (\mathbf{W}_l \mathbf{x}_l + \mathbf{b}_l) + \mathbf{x}_l \quad (4-17)$$

- \mathbf{x}_l 表示第 l 层的输入向量， \mathbf{x}_0 表示原始特征向量，长度都等于 d 。
- $\mathbf{W}_l, \mathbf{b}_l$ 是第 l 个Cross Layer要学习的参数，其中 \mathbf{W}_l 是 $d \times d$ 的矩阵， \mathbf{b}_l 是 d 维向量。
- \odot 表示按位相乘。
- 其余符号的含义与形状参考公式(4-16)。

但是在实现场景中，原始输入 x_0 的长度 d 一般都很长，通常都要大于1000。每层Cross Layer都要配备一个 $d \times d$ 的矩阵，给模型的计算、存储都带来非常大的压力。因此，DCN V2中还提出把 $d \times d$ 的大矩阵 \mathbf{W}_l 分解成为两个 $d \times r$ ($r \ll d$) 的小矩阵相乘的形式，即 $\mathbf{W}_l = \mathbf{U}_l \mathbf{V}_l^T$ ，从而将要学习的参数量从 d^2 减少为 $2 \times d \times r$ 。使用矩阵分解形式的DCN V2迭代公式如(4-18)所示，其他符号参考公式(4-17)。

$$\mathbf{x}_{l+1} = \mathbf{x}_0 \odot (\mathbf{U}_l (\mathbf{V}_l^T \mathbf{x}_l) + \mathbf{b}_l) + \mathbf{x}_l \quad (4-18)$$

尽管如此，由于原始输入的长度 d 一般很大，每层Cross Layer的参数量仍然不小，所以喂入Cross Network的输入一般都是经过挑选的潜在重要特征，不能放开把所有特征一古脑扔进去。而且，每层Cross Layer的输入输出都是 d 维，相当于只做信息交叉，而不做信息的压缩和提炼。以上两个缺点，使得DCN在实战中的表现也未必总是尽如人意。

DCN与DNN融合

代表显式交叉的DCN与代表隐式交叉DNN，有两种融合方式，见公式(4-19)，如图4-4所示。

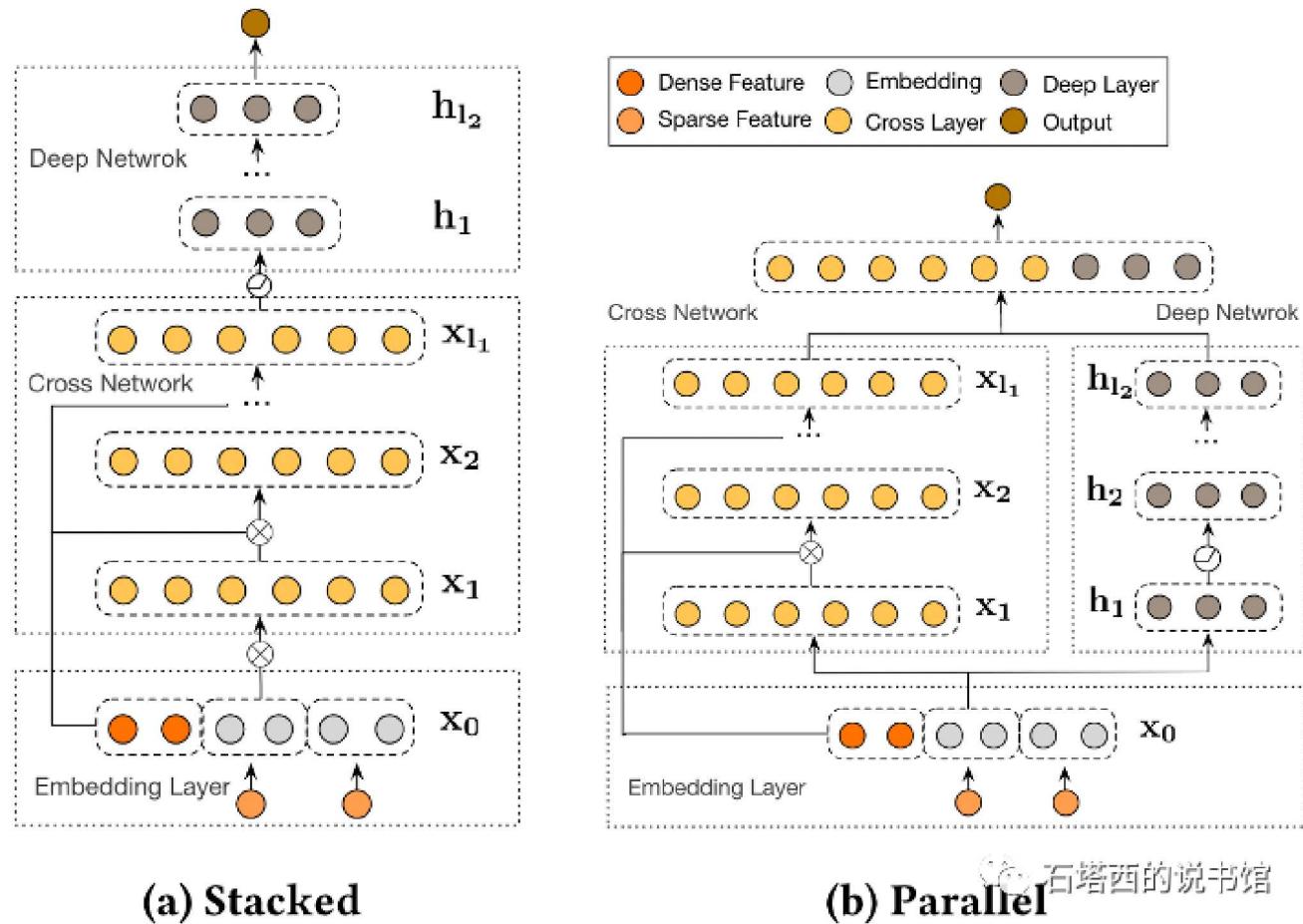


图 4-4 DCN与DNN的两种融合方式

$$\begin{aligned} CTR_{stacked} &= \text{sigmoid}(DNN(\text{CrossNetwork}(\mathbf{x}_0))) & (a) \\ CTR_{parallel} &= \text{sigmoid}(\text{CrossNetwork}(\mathbf{x}_0) + DNN(\mathbf{x}_0)) & (b) \end{aligned} \quad (4-19)$$

- 串联 (Stacked)：原始特征 \mathbf{x}_0 先经过Cross Network进行显式交叉，其结果再喂入DNN进行隐式交叉。其结构参考图4-4(a)。
- 并联 (Parallel)：原始特征 \mathbf{x}_0 分别沿Cross Network与DNN向上传递，最终显式交叉与隐式交叉的预测结果相加。其结构参考图4-4(b)。

DCN源码解析

TensorFlow Recruiters[13]这个库提供一个DCN的实现，其中的关键，“一层Cross Layer”的实现，如所代码4-5示。

代码4-5 DCN中一层Cross的代码

```
class Cross(tf.keras.layers.Layer):
    """一层Cross Layer"""
    pass
```

```

def __init__(self, ....):
    super(Cross, self).__init__(**kwargs)

    self._projection_dim = projection_dim # 矩阵分解时采用的中间维度
    self._diag_scale = diag_scale # 非负小数, 用于改善训练稳定性
    self._use_bias = use_bias
    .....

def build(self, input_shape):# 定义本层要优化的参数
    last_dim = input_shape[-1] # 输入的维度

    # [d,r]的小矩阵, d是原始特征的长度, r就是这里的projection_dim
    # r << d以提升模型的计算效率, 一般取r=d/4
    self._dense_u = tf.keras.layers.Dense(self._projection_dim, use_bias=False, )
    # [r,d]的小矩阵
    self._dense_v = tf.keras.layers.Dense(last_dim, use_bias=self._use_bias, )

def call(self, x0: tf.Tensor, x: Optional[tf.Tensor] = None) -> tf.Tensor:
    """ x0与x计算一次交叉
    x0: 原始特征, 一般是embedding layer的输出。一个[B,D]的矩阵
        B=batch_size, D是原始特征的长度
    x: 上一个Cross层的输出结果, 形状也是[B,D]
    输出: 也是形状为[B,D]的矩阵
    """
    if x isNone:
        x = x0 # 针对第一层

    # 输出是x_{i+1} = x0 .* (W * xi + bias + diag_scale * xi) + xi,
    # 其中.* 代表按位相乘,
    # W分解成两个小矩阵的乘积, W=U*V, 以减少计算开销,
    # diag_scale非负小数, 加到W的对角线上, 以增强训练稳定性
    prod_output = self._dense_v(self._dense_u(x))

    if self._diag_scale:# 加大W的对角线, 增强训练稳定性
        prod_output = prod_output + self._diag_scale * x

    return x0 * prod_output + x

```

4.2.6 AutoInt: 变形金刚做交叉

同样是出于对DNN交叉能力的不满意，北大团队提出AutoInt模型[14]，借用NLP领域经典的Transformer结构[15]实现推荐模型中的特征交叉。

Transformer简介

由于篇幅所限，本书只列出Transformer的具体作法，原理细节请参考文献[15]。Transformer的核心是Attention，其思路是：我们的目标是要将一系列Value压缩成一个Embedding，而压缩方式需要根据输入条件Query而变化。Attention的压缩方式是将所有Value Embedding加权平均，而权重就是Query与Key之间的相似度，如公式(4-20)所示。

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \quad (4-20)$$

- 公式x对Q/K/V的形状有一定要求：
 - B代表Batch Size。
 - L_q 是Query序列的长度。
 - Key/Value两序列的长度必须相等，都是 L_k 。
 - Query/Key两序列中Embedding的长度必须相等，都是 d_k 。
 - d_v 是Value序列中Embedding的长度。
- Q代表Query，是一个形状为 $[B, L_q, d_k]$ 的矩阵。
- K代表Key，是一个形状为 $[B, L_k, d_k]$ 的矩阵。
- V代表Value，是一个形状为 $[B, L_k, d_v]$ 的矩阵。
- $\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)$ 的形状是 $[B, L_q, L_k]$ ，代表每个Query与每个Key的相关性。分母除以 $\sqrt{d_k}$ 是为了防止训练中出现数值问题。
- $\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$ 表示对Value序列加权求和，权重就是Query与Key的相关程度。
- 最终结果 $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ 是一个 $[B, L_q, d_v]$ 的矩阵。它的第*i*行、第*j*列表示，根据第*i*个样本中的第*j*个Query的视角，将所有Value压缩成的长度为 d_v 的向量。

前面提到过，Attention的输入必须满足一定形状上的要求，因此我们首先需要将原始输入线性映射成合适的形状。而且为了进一步增强表达能力，Transformer还采用Multi-Head机制，也就是将原始的Query/Key/Value序列分别映射到不同的子空间，在每个子空间学习出不同的特征交叉。整个过程可以描述成公式(4-21)，其结构如图4-5所示。

$$\begin{aligned} \mathbf{head}_i &= \text{Attention}(\mathbf{QW}_i^Q, \mathbf{KW}_i^K, \mathbf{VW}_i^V) \\ \text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{Concat}(\mathbf{head}_1, \dots, \mathbf{head}_H)\mathbf{W}^O \end{aligned} \quad (4-21)$$

- Q 代表原始Query，是一个形状为 $[B, L_q, d_q]$ 的矩阵。
- K 代表原始Key，是一个形状为 $[B, L_k, d_k]$ 的矩阵。
- V 代表原始Value，是一个形状为 $[B, L_v, d_v]$ 的矩阵。
- $\mathbf{W}_i^Q \in R^{d_q' \times d_k}$, $\mathbf{W}_i^K \in R^{d_k' \times d_k}$, $\mathbf{W}_i^V \in R^{d_v' \times d_v}$ 是第 i 个头的三个映射矩阵，负责将原始输入（最后一维的长度分别是 d_q', d_k', d_v' ）映射成Attention希望的形状 (d_k, d_k, d_v) 。这三个矩阵对应图4-5中底层的三个Linear模块。
- Attention 对应公式(4-20)。
- head_i 是第 i 个头的Attention结果，一共有 H 个头，每个头输出的形状都是 $[B, L_q, d_v]$ 。
- $\text{Concat}(\text{head}_1, \dots, \text{head}_H)$ 将所有头的Attention结果拼接起来，结果的形状是 $[B, L_q, H \times d_v]$ 。
- $\mathbf{W}^O \in R^{Hd_v \times o}$ ，对应图4-5中顶层的Linear模块，将Concat的结果再映射成希望的形状。

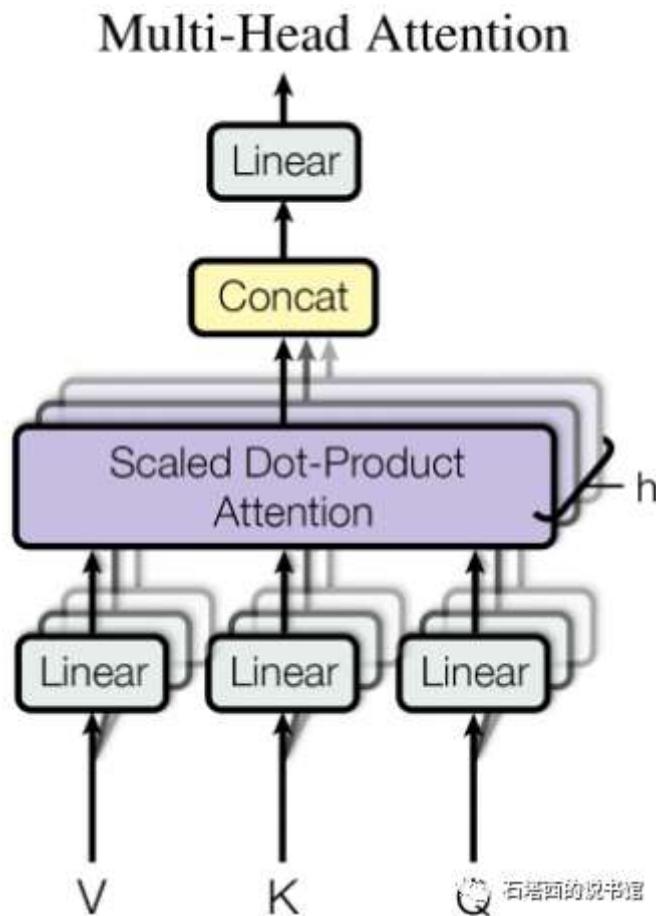


图 4-5 Multi-Head Attention结构示意图

Multi-Head Attention的结果再喂入MLP（图4-6中的Feed Forward）做进一步的非线性变换。为了改善训练稳定性，防止出现数值问题，再引入Layer Norm和Residual结构（图4-6中的Add & Norm）。至此，一层Transformer Layer就构造完毕了。我们可以叠加多层Transform Layer（图4-6中的N），以实现序列特征间更高阶的交叉，整体结构如图4-6所示。

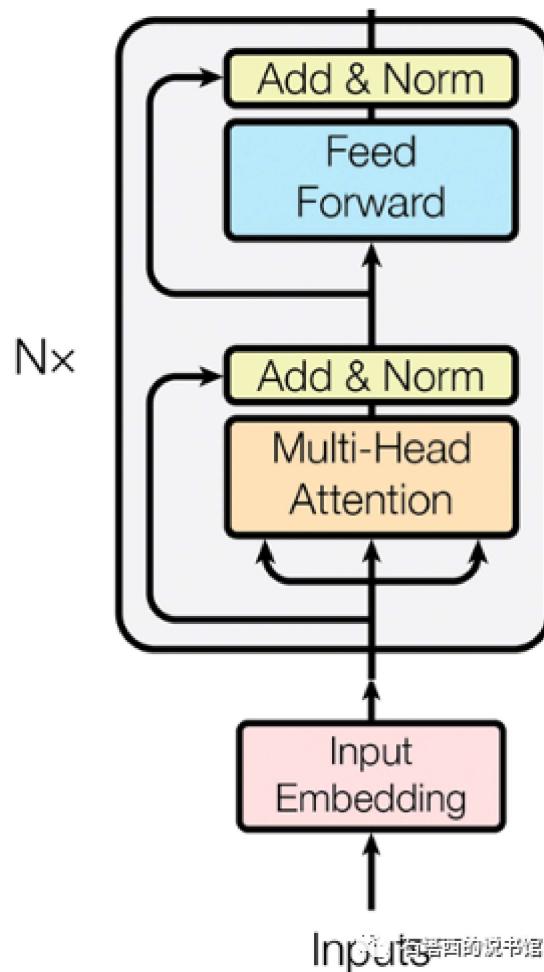


图 4-6 Transformer结构示意图

MultiHeadAttention源码解析

Transformer的核心是Multi-Head Attention[16]，其核心实现如代码4-6所示。

代码 4-6 实现Multi-Head Attention的具体函数

```
def scaled_dot_product_attention(q, k, v, mask):
    """
    输入:
        q: (batch_size, num_heads, seq_len_q, dim_key)
        k: (batch_size, num_heads, seq_len_k, dim_key)
        v: (batch_size, num_heads, seq_len_k, dim_val)
        mask: 必须能够broadcastable to (... , seq_len_q, seq_len_k) 的形状
    输出:
        output: q对k/v做attention的结果, (batch_size, num_heads, seq_len_q, dim_val)
        attention weights: q对k的注意力权重. (batch_size, num_heads, seq_len_q, seq_len_k)
```

```

    # q: (batch_size, num_heads, seq_len_q, dim_key)
    # k: (batch_size, num_heads, seq_len_k, dim_key)
    # matmul_qk: 每个head下，每个q对每个k的注意力权重（尚未归一化）
    # (batch_size, num_heads, seq_len_q, seq_len_k)
    matmul_qk = tf.matmul(q, k, transpose_b=True)

    # 为了使训练更稳定，除以sqrt(dim_key)
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

    # 在mask的地方，加上一个极负的数，-1e9，保证在softmax后，mask位置上的权重都是0
    if mask is not None:
        # mask的形状一般是(batch_size, 1, 1, seq_len_k)
        # 但是能够broadcast成与scaled_attention_Logits相同的形状
        # (batch_size, num_heads, seq_len_q, seq_len_k)
        scaled_attention_logits += (mask * -1e9)

    # 沿着最后一维(i.e., seq_len_k)用softmax归一化
    # 保证一个query对所有key的注意力权重之和==1
    # attention_weights: (batch_size, num_heads, seq_len_q, seq_len_k)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)

    # attention_weights: (batch_size, num_heads, seq_len_q, seq_len_k)
    # v: (batch_size, num_heads, seq_len_k, dim_val)
    # output: (batch_size, num_heads, seq_len_q, dim_val)
    output = tf.matmul(attention_weights, v)

    # output: (batch_size, num_heads, seq_len_q, dim_val)
    # attention_weights: (batch_size, num_heads, seq_len_q, seq_len_k)
    return output, attention_weights

```

再加上定义映射矩阵、对输入、输出做映射、变换形状，完整的Multi-Head Attention的代码如代码4-7所示。其中具体执行的函数scaled_dot_product_attention见代码4-6。

代码 4-7 Multi-Head Attention完整实现

```
class MultiHeadAttention(tf.keras.layers.Layer):
```

```

class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self, num_heads, dim_key, dim_val, dim_out):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.dim_key = dim_key # 每个query和key都要映射成相同的长度
        # 每个value要映射成的长度
        self.dim_val = dim_val if dim_val is not None else dim_key

        # 定义映射矩阵
        self.wq = tf.keras.layers.Dense(num_heads * dim_key)
        self.wk = tf.keras.layers.Dense(num_heads * dim_key)
        self.wv = tf.keras.layers.Dense(num_heads * dim_val)
        self.wo = tf.keras.layers.Dense(dim_out) # dim_out: 希望输出的维度长

    def split_heads(self, x, batch_size, dim):
        # 输入x: (batch_size, seq_len, num_heads * dim)
        # 输出x: (batch_size, seq_len, num_heads, dim)
        x = tf.reshape(x, (batch_size, -1, self.num_heads, dim))

        # 最终输出: (batch_size, num_heads, seq_len, dim)
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, q, k, v, mask):
        """
        输入:
            q: (batch_size, seq_len_q, old_dq)
            k: (batch_size, seq_len_k, old_dk)
            v: (batch_size, seq_len_k, old_dv), 与k序列相同长度
            mask: 可以为空, 否则形状为(batch_size, 1, 1, seq_len_k), 表示哪个key不需要做attention
        输出:
            output: Attention结果, (batch_size, seq_len_q, dim_out)
            attention_weights: Attention权重, (batch_size, num_heads, seq_len_q, seq_len_k)
        """
        # ***** 将输入映射成希望的形状
        batch_size = tf.shape(q)[0]

        q = self.wq(q) # (batch_size, seq_len_q, num_heads * dim_key)
        k = self.wk(k) # (batch_size, seq_len_k, num_heads * dim_key)
        v = self.wv(v) # (batch_size, seq_len_k, num_heads * dim_val)

        # (bs, nh, seq_len_q, dim_key)

```

```

q = self.split_heads(q, batch_size, self.dim_key)
# (bs, nh, seq_len_k, dim_key)
k = self.split_heads(k, batch_size, self.dim_key)
# (bs, nh, seq_len_k, dim_val)
v = self.split_heads(v, batch_size, self.dim_val)

# **** Multi-Head Attention

# scaled_attention: (batch_size, num_heads, seq_len_q, dim_val)
# attention_weights:(batch_size, num_heads, seq_len_q, seq_len_k)
scaled_attention, attention_weights = scaled_dot_product_attention(
    q, k, v, mask)

# **** 将Attention结果映射成希望的形状
# (batch_size, seq_len_q, num_heads, dim_val)
scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])

# (batch_size, seq_len_q, num_heads * dim_val)
concat_attention = tf.reshape(scaled_attention,
                               (batch_size, -1, self.num_heads * self.dim_val))

output = self.wo(concat_attention) # (batch_size, seq_len_q, dim_out)

return output, attention_weights

```

注意代码4-6中第24 ~ 28行中的mask，其作用是用来指定一个序列哪些位置上的Key、Value不需要参与Attention。比如一个batch内，各用户观看过的视频有多有少，对于较长的观看序列，我们需要截断(Truncate)，对于较短的观看序列，我们需要填充(Padding)，总之将每条样本中的观看序列整理成相同长度，才好喂入模型。假设有两个用户的观看序列组成一个batch，模型允许的最大序列长度为5，喂入模型的特征如图4-7所示， $v_1 \sim v_4$ 表示真正的视频ID，0表示因为长度不足而填充的占位符。

$$\text{用户看过的视频序列} = \begin{bmatrix} v_1 & v_2 & v_3 & 0 & 0 \\ v_4 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 4-7 用户行为序列填充示意

对于这样的输入数据，我们必须制造一个mask，告诉Attention凡是0的位置只是占位符，并非真正的视频，没必要做Attention，其实现如代码4-8所示。

代码 4-8 制作mask，避免不必要的Attention

```

def create_padding_mask(seq):
    """
    seq: [batch_size, seq_len]的整数矩阵。如果某个元素==0, 代表那个位置是padding
    """

    # (batch_size, seq_len)
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

    # 返回结果: (batch_size, 1, 1, seq_len)
    # 加入中间两个长度=1的维度, 是为了能够broadcast成希望的形状
    return seq[:, tf.newaxis, tf.newaxis, :] # (batch_size, 1, 1, seq_len)

```

基于Transformer的特征交叉

熟悉了Transformer，看AutoInt的过程就非常简单了，可以由代码 4-9 来描述。

代码 4-9 AutoInt的伪代码实现

Algorithm: AutoInt for CTR prediction

1. **for** $m=1$ to M **do** // 将每个 Field 映射成向量，一共有 M 个 Field
 2. | $e_m = \text{Embedding}_m(feature_m)$ // e_m 是第 m 个 Field 的 Embedding，是一个 d 维向量
 3. **end for**
 4. // 将所有 Field 的 Embedding 拼接成一个 $[B, M, d]$ 的矩阵
 5. // B =batch size, M 是 Field 个数, d 是每个 Field Embedding 的长度
 6. $X = \text{Concat}(e_1, \dots, e_M)$
 - 7.
 8. // 进行 N 层 Transformer 特征交叉
 9. **for** $n=1$ to N **do**
 10. | // 每层 Q/K/V 都使用 X , 相当于做 Self-Attention
 11. | // 输入输出的 X 都是 $[B, M, d]$ 形状的矩阵
 12. | $X = \text{Transformer}(\text{query}=X, \text{key}=X, \text{value}=X)$
 13. **end for**
 - 14.
 15. // 充分交叉后的特征，喂入一个浅层小网络，得到最终的 CTR
 16. $X' = \text{reshape}(X)$ // 改变输入的 $[B, M, d]$ 矩阵的形状成 $[B, Md]$ 的矩阵
 17. $\text{CTR}_{\text{predict}} = \text{sigmoid}(\text{DNN}(X'))$ // 映射成 CTR

 石塔西的说书馆

第一步，代码4-9的1 ~ 3行，准备各Field的Embedding。假设一共有 M 个Field，其中第 m 个Field的 Embedding 是 $e_m \in R^d$ 。至于如何得到 e_m 就非常常规了，请参考4.2.3中介绍Deep时的方法。

第二步，代码4-9的6 ~ 13行，将所有Field的Embedding组成一个 $[B, M, d]$ 的矩阵 X ，然后套用 Transformer。

- Transformer的技术细节请参考图4-5与公式(4-21)。
- 这个Transformer中 $Q = K = V = X$, 所以这里面的Attention属于Self-Attention。
- Transformer中, 每个头中的三个映射矩阵 W_i^Q, W_i^K, W_i^V 的形状都是 $[d, d']$, d 是特征向量的原始长度, d' 是便于计算的中间长度。一般 $d' < d$, 可以减少一些计算量。
- Transformer中, W^O 的形状是 $[Hd', d]$, 将Multi-Head Attention的结果重新映射回 d , 以便接入下一层Transformer。

第三步, 代码4-9的15~17行, 多层Transformer的结果仍然是一个形状是 $[B, M, d]$ 的矩阵, 将其拼接起来喂入一个浅层DNN, 得到最终预测结果。

AutoInt有两个缺点:

- 为了使用Self-Attention, 要求各Field Embedding的长度必须相等, 这显然太死板了。
- 和DCN一样, 每层Transformer对信息只交叉不压缩, 每层输出的形状都是 $[B, M, d]$ 。而且对 M 个Field做Self-Attention的时间复杂度是 $O(M^2)$ 。推荐系统的 M 和 d 都比较大, 所以AutoInt的时间开销不小。

因此, 在实践中, 我们并不让AutoInt独立预测CTR, 而只是将它作为一个特征交叉模块, 嵌入更大的推荐模型中。这样一来, 我们可以只选择一部分重要特征喂入AutoInt做交叉, 从而减小了计算量。

实际上, 推荐算法中的很多成果都借鉴NLP领域的理论与思想, AutoInt只是其中的一个代表。AutoInt的核心是Attention, 在4.3.3节中, 我们还会看到Attention在推荐模型中发挥的重要作用。

4.3 用户行为序列建模

本节将聚焦于推荐系统中最重要的一类特征, 用户行为序列 (e.g., 点击序列、观看序列、购买序列、……)。这些行为序列之所以重要, 是因为它们蕴含着丰富的用户兴趣信息等待我们去挖掘。本节将讨论将一个行为序列提炼、压缩成一个反映用户兴趣的Embedding的技术方法。

4.3.1 行为序列信息的构成

以“用户最近观看的50个视频”这个序列为例。序列中每个元素的Embedding, 一般由以下几个部分拼接组成:

- 由每个视频的ID进行Embedding得到的向量。

- 时间差信息：计算观看该视频的时刻距离本次请求时刻之间的时间差，将这个时间差桶化成一个整数，再Embedding。这个时间差信息非常重要，序列中不同元素之间的相互影响，历史序列元素对当前候选物料之间的影响，肯定是随着彼此间隔时间而衰减的，因此我们必须将如此重要的信息喂入模型，方便模型刻画时间衰减。
- 以上两点是每个序列元素最重要的信息。除此之外，还可以加入观看视频的一些元信息（e.g., 视频的作者、来源、分类、标签等），和动作的程度（e.g., 观看时长、观看完成度）。

4.3.2 简单Pooling

将一个用户的行为序列压缩成一个兴趣向量，最简单的方法就是进行如下按位（element-wise）操作：

- Sum Pooling: $E_{interest} = \sum_i e_i$
- Average Pooling: $E_{interest} = \frac{1}{N} \sum_i e_i$
- Weighted-sum Pooling: $E_{interest} = \frac{1}{N} \sum_i w_i e_i$ 。权重 w_i 根据时间差或动作程度计算。比如某个历史上观看过的视频，观看的时间距离当前时间越近，观看完成度越高，就越能反映用户兴趣，权重 w_i 就应该越大。

简单Pooling提取出来的用户兴趣是固定的，不像接下来介绍的方法那样，能够随当前候选物料而变化。对于精排，这当然是个缺点。但是对于召回、粗排这种用户、物料必须解耦建模的场景，反正提取用户兴趣时也拿不到候选物料信息，简单Pooling仍然是最常见的选择。

4.3.3 用户建模要"千物千面"

简单Pooling的问题在于它将序列中的所有元素一视同仁，而在现实中，不同历史记忆对当下决策的影响程度并不相同。举个例子，一个用户过去买过泳衣和iPhone手机。

- 当被展示的商品是游泳镜时，用户是否点击，更多是出于他对买过泳衣的历史记忆。
- 当被展示蓝牙耳机时，过去购买iPhone的历史记忆将主导他此次是否会点击。换句话说，从用户行为序列中提取出来的兴趣向量，应该随当前候选物料的变化而变化，实现"千物千面"的效果。

阿里妈妈于2018年提出的Deep Interest Network (DIN) [17]借鉴NLP中的Attention，实现了这样的"千物千面"效果。DIN中提取用户兴趣的结构如图4-8所示：

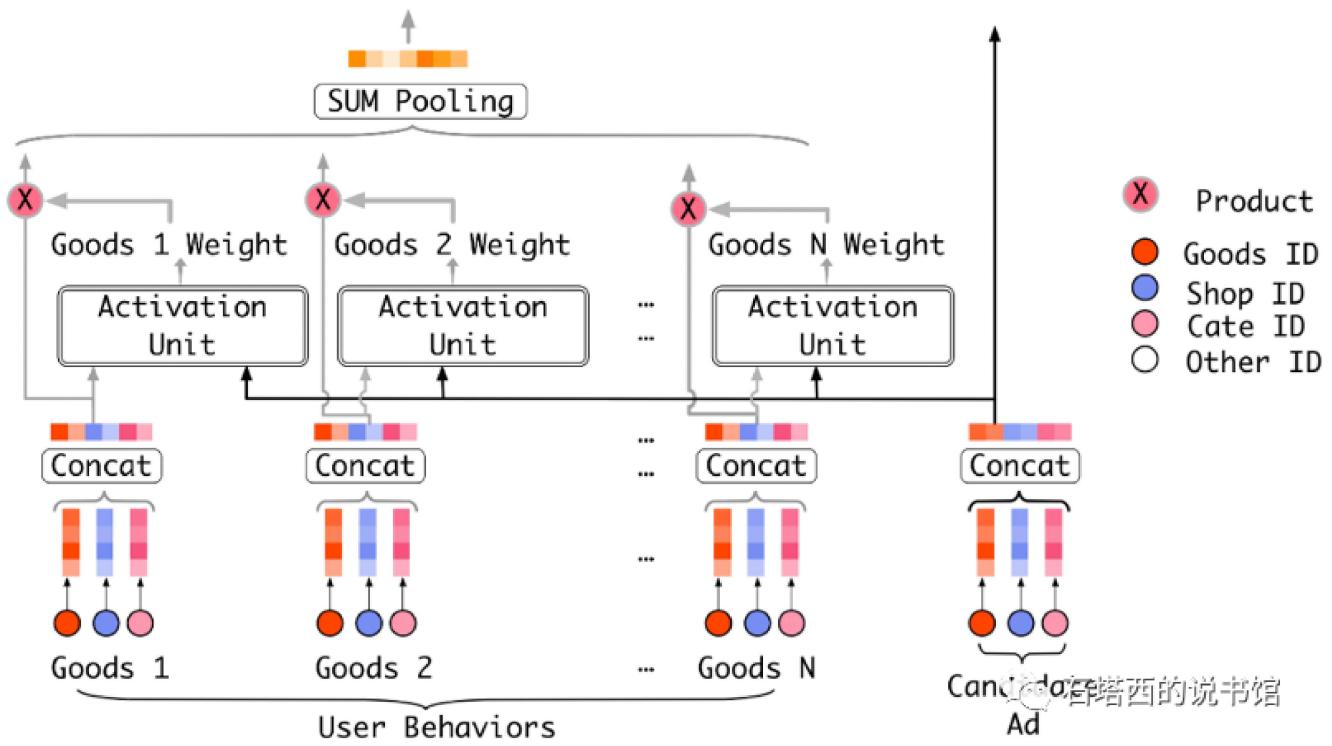


图 4-8 DIN提取用户兴趣示意

DIN的核心思想可以参考公式(4-22)。

$$UE_{u,t} = \sum_{j=1}^H w_j h_j = \sum_{j=1}^H A(h_j, t) h_j \quad (4-22)$$

- UE是User Embedding的缩写，下标*u*和*t*表示户兴趣向量取决于用户*u*与候选物料*t*双方面信息。
- *h_j*表示用户交互过的第*j*个历史物料的Embedding。它可以由该物料的各种属性（比如图4-8中的Goods ID、Shop ID、Cate ID）的Embedding拼接而成。
- *w_j = A(h_j, t)*是历史物料*h_j*在构成用户兴趣中的权重，由当前候选物料*t*与*e_i*的相似度决定
- *A*是生成*w_j*的函数，对应图 4-8中的Activation Unit，可以简单如点积，也可以复杂为一个小型MLP。

对比4.2.6节，我们可以发现，DIN就是在拿候选物料*t*当Query对用户的历史序列[h₁, ..., h_H]做Attention，从而将历史行为序列压缩成一个根据候选物料变化的稠密向量，从而实现用户兴趣建模的“千物千面”。

DIN中建模用户兴趣的部分如代码4-10所示，其中的MultiHeadAttention实现在代码 4-7中。

代码 4-10 DIN中建模用户兴趣

```

target_item_embedding = ... # 候选item的embedding, [batch_size, dim_target]
user_behavior_seq = ... # 某个用户行为序列, [batch_size, seq_len, dim_seq]
padding_mask = ... # user_behavior_seq中哪些位置是填充的, 不需要Attention

# 把候选item, 变形成一个长度为1的序列

```

```

query = tf.reshape(target_item_embedding, [-1, 1, dim_target])

# atten_result: (batch_size, 1, dim_out)
attention_layer = MultiHeadAttention(num_heads, dim_key, dim_val, dim_out)
atten_result, _ = attention_layer(
    q=query, # query就是候选物料
    k=user_behavior_seq,
    v=user_behavior_seq,
    mask=padding_mask)

# reshape去除中间不必要的1维
# user_interest_emb是提取出来的用户兴趣向量，喂给上层模型，参与CTR建模
user_interest_emb = tf.reshape(atten_result, [-1, dim_out])

```

另外需要注意的是，DIN中的Attention需要拿候选物料当Query，这在召回、粗排这些要求用户、物料解耦建模的场景是做不到的。这时，可以尝试拿用户行为序列中的最后一个物料当Query对整个行为序列Attention[18]。毕竟最后的行为反映用户最近的兴趣，可以当尺子衡量序列中其他历史物料的重要性。

4.3.4 建模序列内的依赖关系

DIN实现了用户兴趣的“千物千面”，但是仍有不足，就是它只刻画了候选物料与序列元素的交叉，却忽略了行为序列内部各元素之间的依赖关系。比如一个用户购买过MacBook和iPad，这两个历史行为的组合将产生非常强烈的信号，但是这种序列内部的交叉组合却未能在DIN中体现。为此，业界提出了一系列模型，使从行为序列中提取出来的用户兴趣向量，既能反映候选物料与历史记忆之间的相关性，又能反映不同历史记忆之间的依赖性。

目前较为常用的建模用户行为序列的方法是采用双层Attention[19,20]，其结构如图4-9所示。

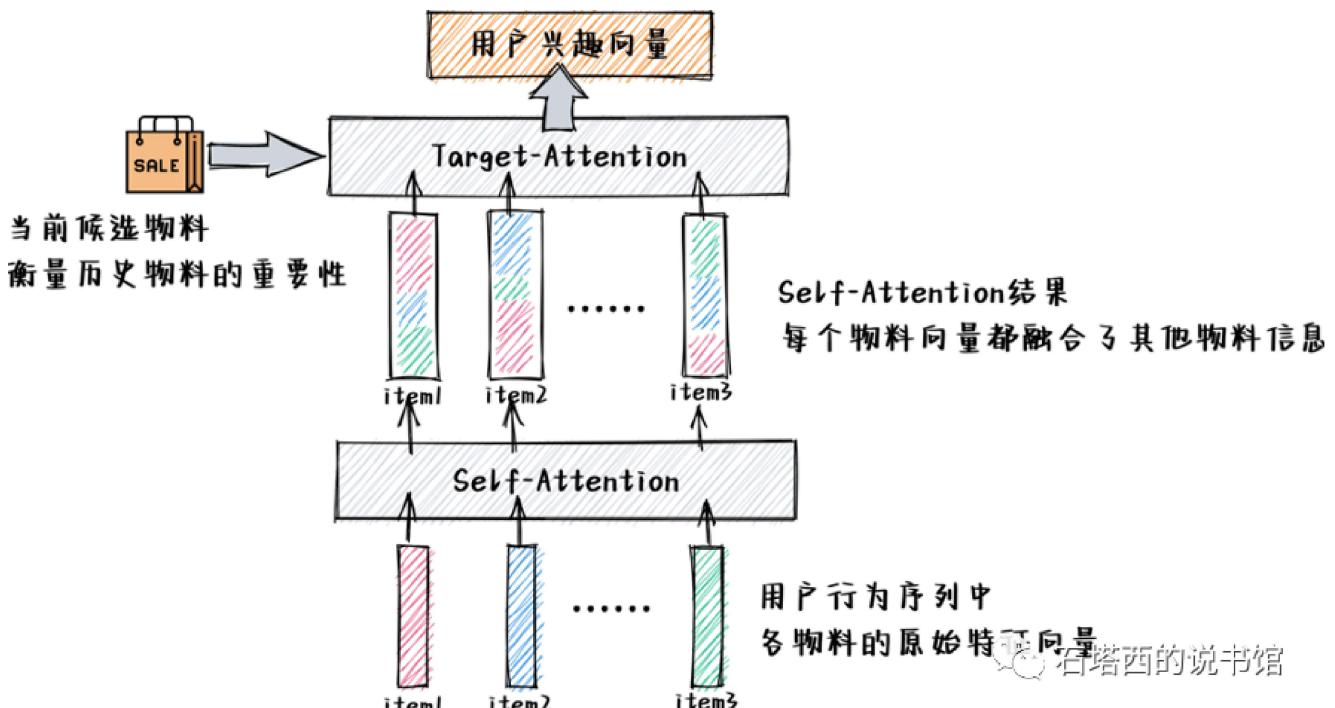


图 4-9 双层Attention建模用户行为序列

第一步，用Multi-Head Self-Attention建模行为序列内部的依赖关系。Multi-Head Self-Attention的结果是一个与原始序列相同长度的新序列，新序列中的每个元素都以不同方式（i.e., Multi-Head）融合了原始序列中其他物料的信息。

还举上面的例子，在原始用户行为序列中，“购买过Macbook”这一历史行为反映的只是单次购买本身。而对行为序列Self-Attention之后，同样的历史行为不仅反映了本次购买，还体现了2天前用户还购买过iPhone，和1天后用户又购买了iPad。通过与前后历史行为的交叉、关联，一个苹果忠实粉丝的形象跃然纸上。就这样，底层提取出来的用户兴趣的质量得以提高，上层的CTR建模自然也就降低了难度。

这里还可以像Transformer那样，叠加多层Self-Attention建模更深的特征交叉，但是那样做的计算开销也更大，通常一层Self-Attention足矣。

第二步，用当前候选物料和Self-Attention产生的新序列上套用DIN，建模候选物料与历史行为序列之间的相关性，得到最终的用户兴趣向量。

基于双层Attention建模用户兴趣，如代码4-11所示。其中MultiHeadAttention的实现请参考代码 4-7。

代码 4-11 双层Attention建模用户兴趣

```

target_item_embedding = ... # 候选item的embedding, [batch_size, dim_target]
user_behavior_seq = ... # 某个用户行为序列, [batch_size, seq_len, dim_in_seq]
padding_mask = ... # user_behavior_seq中哪些位置是填充的, 不需要attention
dim_in_seq = tf.shape(user_behavior_seq)[-1] # sequence中每个element的长度

```

```

# ***** 第一层做Self-Attention, 建模序列内部的依赖性
self_atten_layer = MultiHeadAttention(num_heads=n_heads1,
                                       dim_key=dim_in_seq,
                                       dim_val=dim_in_seq,
                                       dim_out=dim_in_seq)

# 做self-attention, q=k=v=user_behavior_seq
# 输入q/k/v与输出self_atten_seq, 它们的形状都是
# [batch_size, Len(user_behavior_seq), dim_in_seq]
self_atten_seq, _ = self_atten_layer(q=user_behavior_seq,
                                      k=user_behavior_seq,
                                      v=user_behavior_seq,
                                      mask=padding_mask)

# ***** 第二层做Target-Attention, 建模候选item与行为序列的相关性
target_atten_layer = MultiHeadAttention(num_heads=n_heads2,
                                         dim_key=dim_key,
                                         dim_val=dim_val,
                                         dim_out=dim_out)

# 把候选item, 变形成一个长度为1的序列
target_query = tf.reshape(target_item_embedding, [-1, 1, dim_target])

# atten_result: (batch_size, 1, dim_out)
atten_result, _ = target_atten_layer(
    q=target_query, # 代表候选物料
    k=self_atten_seq, # 以self-attention结果作为target-attention的对象
    v=self_atten_seq,
    mask=padding_mask)

# reshape去除中间不必要的1维
# user_interest_emb是提取出来的用户兴趣向量, 喂给上层模型, 参与CTR建模
user_interest_emb = tf.reshape(atten_result, [-1, dim_out])

```

4.3.5 多多益善：建模长序列

以上章节介绍的用户行为序列建模方式主要是基于Attention的，观察一下它们的时间复杂度：

- 假设一个batch的大小为 B , 用户行为序列的长度为 L , 序列中每个Embedding的长度为 d
- DIN中, 拿候选物料当Query对整个序列做Attention时, 时间复杂度= $O(B \times L \times d)$
- 双层Attention中, 用户行为序列内部做Self-Attention的时间复杂度= $O(B \times L^2 \times d)$ 所以, Attention的建模方式的时间复杂度, 与用户行为序列的长度呈线性甚至平方的关系, 在 L 较小的时候还算可行, 而

当 L 非常大的时候，是无法满足在线预测与训练更新的实时性要求的。

而推荐系统是有对用户长期行为序列建模的需求的。因为如果建模的序列太短，其中难免会包含一些用户临时起意的行为，算是一种噪声。另外，太短的行为序列也无法反映用户的一些周期性行为，比如每周、每月的习惯性采购。

为了解决以上矛盾，业界提出了一些方法。根据用户兴趣向量的提取时机，可以分为“动态在线”与“静态离线”两个技术流派。

在线提取用户兴趣

在线派的代表是阿里妈妈于提出的Search-based Interest Model (SIM) [21]，如图 4-10 所示。

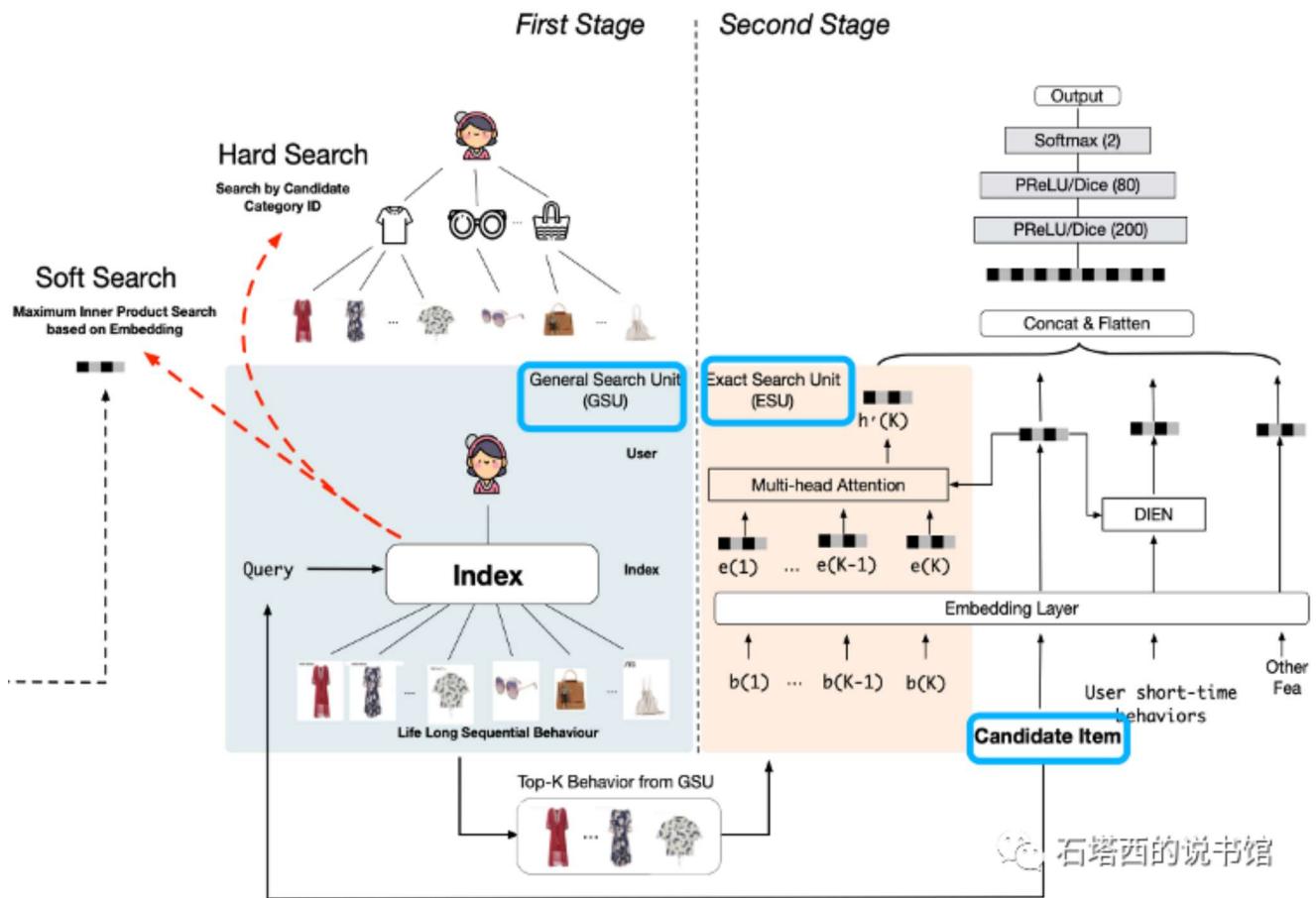


图 4-10 SIM 结构图

- DIN 是拿候选物料 t 对行为序列 $[e_1, e_2, \dots, e_H]$ 做 Attention。和 t 相似的历史物料 e_h ，权重会高一些，反之，权重会低一些。这里的 Attention 相当于对历史行为序列做“软过滤”。之前说过了，它的时间复杂度与序列长度 L 成线性关系。
- 既然在长序列上做 Attention 软过滤的代价太大，就干脆直接上硬过滤。在长序列中筛选（搜索）出与候选物料 t 相关的一个短序列（一般长度在 200 以下），称为 Sub user Behavior Sequence (SBS)。这个硬过滤的过程就是原文中的 General Search Unit (GSU)。

- 由于长度大大缩短（万级→百级），在SBS再套用DIN就变得可行。拿候选物料 t 和SBS做Attention，加权平均后的结果就是用户长期兴趣的向量表达。这个过程就是原文中的Exact Search Unit (ESU)。

而根据在General Search Unit中如何搜索，SIM又有Hard Search和Soft Search两种实现方式。

Hard Search

所谓Hard Search，就是拿候选物料 t 的某个属性（比如，物料分类或标签），在用户完整的长期历史中搜索与其有相同属性的历史物料，组成SBS。比如当前候选物料是一件衣服，SIM将该用户过去所有购买过的衣服挑选出来，组成针对这个候选物料的SBS。

当然，真正的搜索过程，不可能为每个候选物料都把用户全部历史都遍历一遍。为了加速这一过程，阿里特别设计了User Behavior Tree (UBT) 数据库，将每个用户的长期行为序列，“分门别类”地缓存起来。UBT的结构如图4-11所示，类似一个双层的HashMap。

- 外层HashMap的key是UserId。
- 内层HashMap的key就是某个属性（比如，物料分类）。
- 内层HashMap的value就是某个用户在某个属性下的SBS。

这样一来，通过两层哈希查找，模型就能当前用户针对当前候选物料的SBS，效率足以满足线上预测与训练的需要。

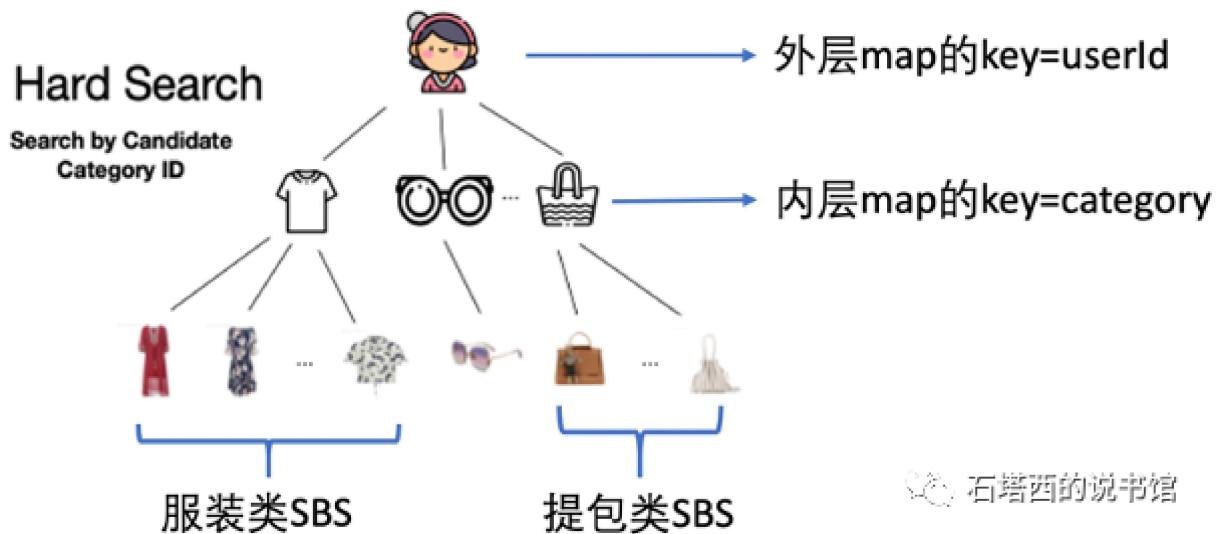


图 4-11 基于UBT的Hard Search

虽然简单，但是据原论文中所说，搜索效果和接下来要介绍的Soft-Search差不多，反而实时性能优秀，维护更加方便。

Soft Search

正如本文3.1节中指出的那样，Hard Search是拿物料属性进行精确匹配，不如用Embedding进行“模糊查找”的扩展性好。于是，很自然想到用候选物料的Item Embedding，在用户长期行为序列中通过“近似近邻搜索算法”（Approximate Nearest Neighbors，ANN），查找与之距离最近的前 K 个历史物料，组成SBS，这就是所谓的Soft Search。

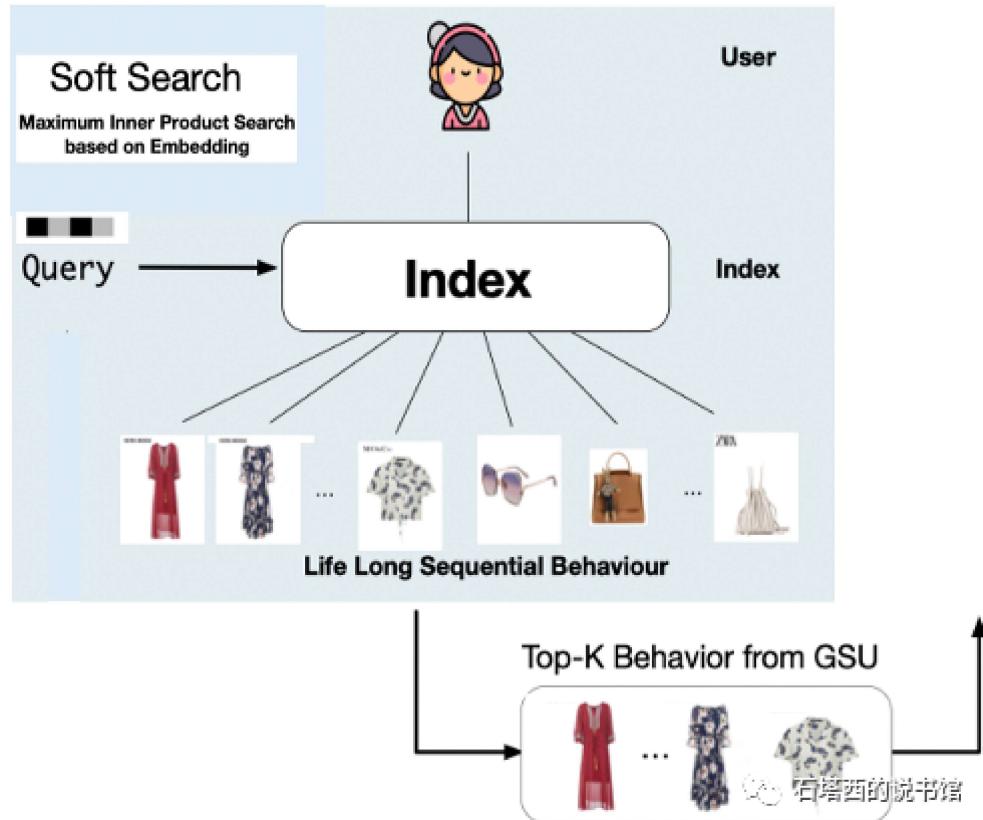


图 4-12 基于向量近邻搜索的Soft Search

至于Item Embedding从何而来？原论文里，是只用候选物料和长期行为序列，构建了一个小模型预测CTR。模型训练完成后的复产品就是Item Embedding。除此之外，用双塔召回/粗排得到的Item Embedding行不行？用Word2vec算法套用在长期行为序列上得到的Item Embedding，行不行？笔者觉得，理论上是没毛病的，都可以试一试，让离线指标和在线A/B testing的结果来告诉我们具体用哪种是最好的。

值得注意的是，SIM的论文里指出，用户长短期行为历史的数据分布差异较大，建模用户短期行为序列的Item Embedding，不宜复用于建模用户长期行为序列。也就是说，同一个物料在用于建模用户长期兴趣与短期兴趣时，应该对应完全不同的Item Embedding。基于同样的原因，如果你想用双塔模型的Item Embedding来进行Soft Search，你的双塔模型最好也拿长期行为序列来训练。如果嫌序列太长，拖慢双塔的速度，可以对长序列进行采样。

得到的Item Embedding之后，要喂入Faiss[22]这样的向量数据库，离线建立好索引（图4-13中的Search Index），以加速线上预测和训练时的Soft Search。

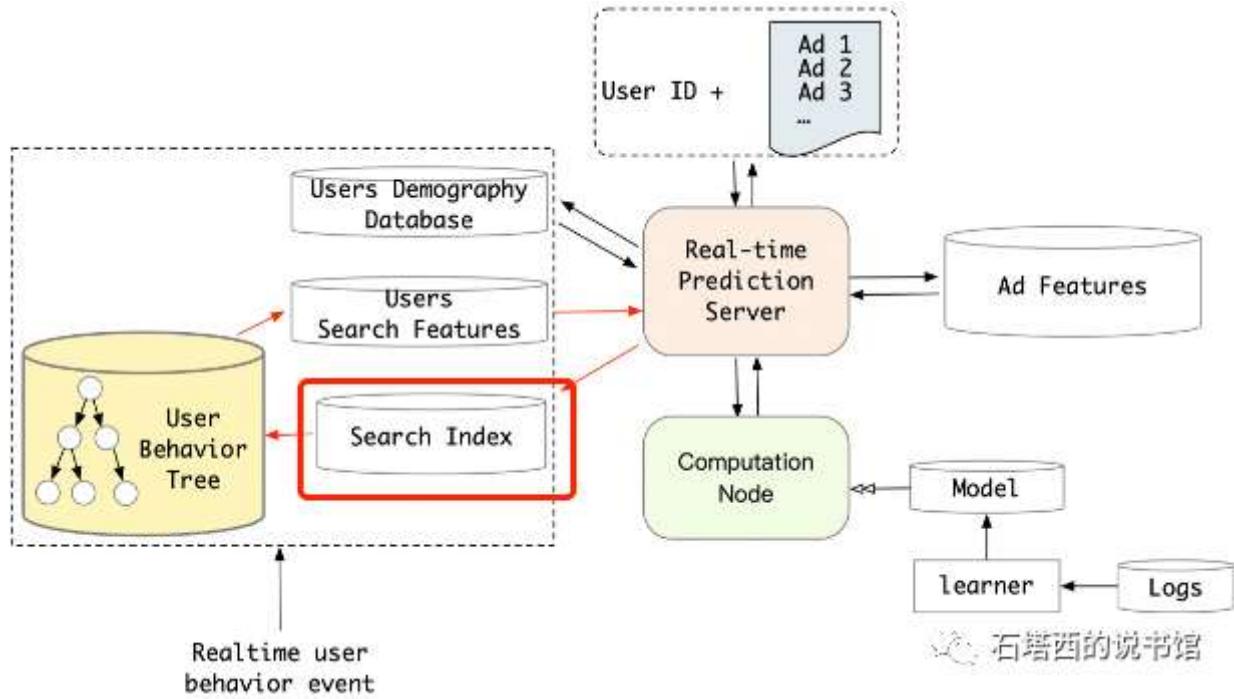


图 4-13 SIM线上架构示意

对于SIM，注意如下两点：

- 无论是在线预测还是训练更新，面对不同的候选物料时，GSU筛选出来的SBS是不同的，ESU根据SBS压缩提炼出来的用户长期兴趣向量也是不同的，从而实现“千物千面”。这也就是我称以SIM为代表的技术路线为“动态在线”的原因。
- 在SIM中，无论Soft Search还是Hard Search都要离线维护一套索引，以加速在线搜索的过程，如图4-13所示。离线索引中的Item Embedding是另一套模型训练出来的，当初的训练目标可能未必与推荐主模型完全一致。而且索引中向量的更新频率，肯定远不如推荐主模型频繁。因为以上两个原因，第1阶段“搜索SBS”与第2阶段“推荐主模型”会存在性能差异。目前有一些方法试图取消离线索引，让“搜索”与“推荐”两阶段共用一套Item Embedding，以减少两阶段之间的gap。这种方式尚未成为主流，感兴趣的的同学可以参考文献[23]。

离线预训练用户兴趣

虽然SIM模型提取出来的长期用户兴趣有“千物千面”的优点，但其缺点就是实现起来太过复杂，线上预测耗时增加得比较多。需要有工程团队的强有力配合，才能将在线预测和训练更新的耗时，降低到满足实时性要求。即便如此，一次请求中的候选物料也不能太多，因此SIM只适用于精排这种候选集有限的环节。

没有阿里那么强的工程架构能力，或者想在召回、粗排阶段引入用户长期兴趣，这时抄不了SIM的作业，怎么办？别急，除了阿里的“在线派”技术路线，我们可以离线将用户的长期兴趣挖掘好，缓存起来供线上模型调用。这种“离线派”技术路线，将费时的“挖掘用户长期兴趣”这一任务由线上转移到线下，省却了优化线上架构的麻烦，实现起来更加简单便捷。

一种方法就是人工统计长期兴趣。本书的2.2.2节就提到手工挖掘用户长期兴趣，在某些场合下代替SIM这种“强但重”的模型。比如，我们可以统计出每个用户针对某个商品分类或视频标签，在过去1周、过去1个月等较长时段内的CTR，代表用户长期兴趣，喂进推荐模型。

另一种方法就是离线预训练一个辅助模型，提取用户长期兴趣。其一般流程是：

- 预训练该模型，输入用户长期行为序列，输出一个Embedding代表用户长期兴趣。
- 训练好这个辅助模型之后，将行为序列超过一定长度的用户，都过一遍这个模型。得到代表这些用户长期兴趣的Embedding，存入Redis之类的KV数据库。
- 当线上预测或训练需要用户长期兴趣时，就拿用户的UserId检索Redis，得到代表他的长期兴趣的Embedding，喂入推荐主模型。
- 尽管用户一天之内会频繁动作，但是各用户的长期兴趣向量和生成它们的预训练模型，无须实时更新，只需要每天更新一次即可。因为用户刚刚发生的行为属于短期兴趣的建模范畴，留给DIN、双层Attention等模型去应付足矣。

美团建模超长用户行为序列，遵循的就是这种“离线派”技术路线。据说在美团场景下，取得了比SIM更好的效果[24]。

至于如何构建这样的预训练模型，一种方法是，用同一个用户的长期行为序列，预测他的短期行为序列，如图4-14所示：

- 模型采用双塔结构（5.5节会有详细介绍）。
- 喂入模型的样本是一个三元组 $\langle LS_A, SS_A, SS_B \rangle$ 。其中 LS_A 和 SS_A 是一个用户A的长期行为序列和短期行为序列， SS_B 是随机采样到的另一个用户B的短期行为序列。
- LS_A 喂入左塔得到A用户的长期兴趣向量 UL_A ， SS_A 和 SS_B 喂入右塔得到A、B两用户的短期兴趣向量 US_A 和 US_B 。
- 建模目标是，同一用户的长短期兴趣向量应该相近，即 $\text{cosine}(UL_A, US_A)$ 越大越好；不同用户的长短期向量相距较远，即 $\text{cosine}(UL_A, US_B)$ 越小越好。
- 双塔模型训练好之后，离线将老用户的长期行为序列喂入左塔，就得到表示各用户长期兴趣的向量。

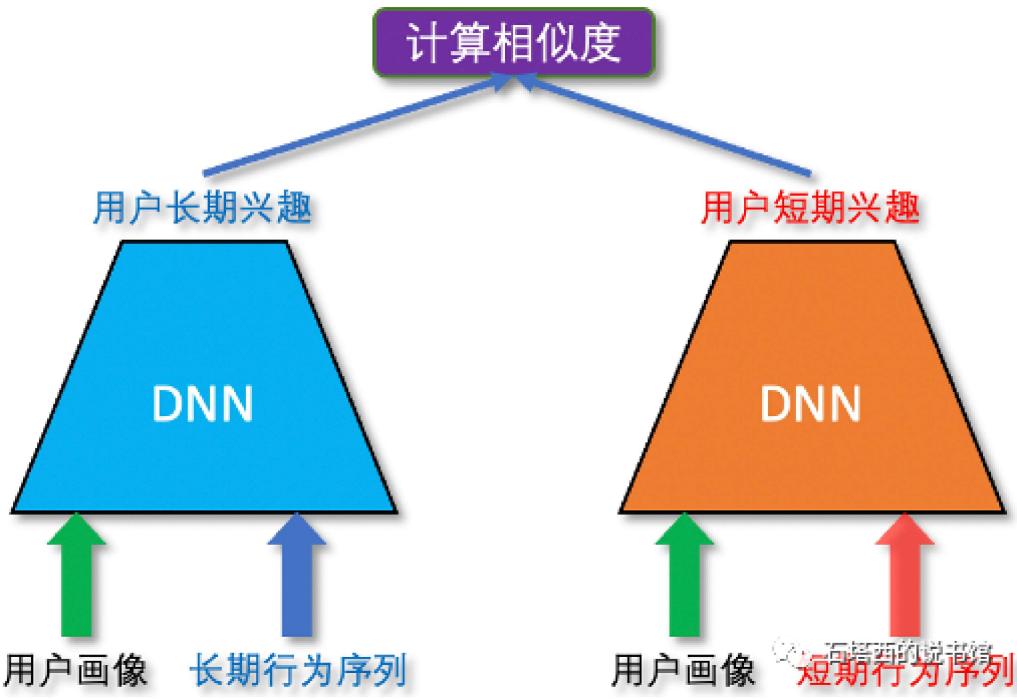


图 4-14 双塔模型预训练用户长期兴趣

再次强调，“离线预训练用户长期兴趣”的作法，优点是不会增加线上耗时，实现简单；缺点就是得到的用户长期兴趣，不会随着候选物料而变化，无法做到“千物千面”。离线 vs. 在线，两种技术路线各有优劣，请读者根据自己的实际情况，权衡选择。

4.4 本章小结

本章从“交叉结构”与“用户行为序列建模”两个维度，讨论了精排环节使用到的模型。

第4.2节，从“交叉结构”维度出发，讨论了：

- 前深度学习时代，偏重记忆的FTRL算法、FM模型。
- 深度学习时代，兼顾“记忆”与“扩展”的Wide & Deep、DeepFM模型
- 不再迷信DNN，特征的“显式交叉”与“隐式交叉”并举的DCN、AutoInt模型

用户行为序列，隐藏着用户最真实的兴趣，是推荐模型最重要的信息来源。对它的建模，是推荐模型的重中之重。第4.3节重点介绍了：

- DIN通过候选物料对历史行为序列做Attention，使挖掘出来的用户兴趣，在面对不同候选物料时，呈现“千物千面”的效果。

- 双层Attention模式，既建模了行为序列与当前候选物料之间的相关性，又建模了行为序列内部的相互依赖关系。
- "在线提取"和"离线预训练"两类方法，使我们能够在更长的用户行为序列上建模用户兴趣，同时满足在线预测与训练的实时性要求。

4.5 本章参考文献

- [1] LANGFORD J, LI L, ZHANG T. Sparse Online Learning via Truncated Gradient[J].
- [2] DUCHI J, SINGER Y. Efficient Online and Batch Learning Using Forward Backward Splitting[J].
- [3] XIAO L. Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization[J].
- [4] MCMAHAN H B, HOLT G, SCULLEY D, 等. Ad click prediction: a view from the trenches[C/OL]//Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. Chicago Illinois USA: ACM, 2013: 1222-1230[2022-04-15]. <https://dl.acm.org/doi/10.1145/2487575.2488200>. DOI:10.1145/2487575.2488200.
- [5] MASSQUANTITY. 在线优化算法 FTRL 的原理与实现[EB/OL].
<https://www.cnblogs.com/massquantity/p/12693314.html>.
- [6] RENDLE S. Factorization Machines[C/OL]//2010 IEEE International Conference on Data Mining. 2010: 995-1000. DOI:10.1109/ICDM.2010.127.
- [7] CASTELLANZHANG. alphaFM[CP/OL]. <https://github.com/CastellanZhang/alphaFM>.
- [8] CHENG H T, KOC L, HARMSEN J, 等. Wide & Deep Learning for Recommender Systems[C/OL]//Proceedings of the 1st Workshop on Deep Learning for Recommender Systems. Boston MA USA: ACM, 2016: 7-10[2022-04-21]. <https://dl.acm.org/doi/10.1145/2988450.2988454>. DOI:10.1145/2988450.2988454.
- [9] tf.keras.experimental.WideDeepModel[CP/OL]. Google. https://github.com/keras-team/keras/blob/v2.9.0/keras/premade_models/wide_deep.py#L33-L217.
- [10] GUO H, TANG R, YE Y, 等. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction[J/OL]. arXiv:1703.04247 [cs], 2017[2022-04-10]. <http://arxiv.org/abs/1703.04247>.
- [11] 石塔西. 用TensorFlow实现支持多值、稀疏、共享权重的DeepFM[EB/OL].
<https://zhuanlan.zhihu.com/p/48057256>.

[12] WANG R, SHIVANNA R, CHENG D Z, 等. DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems[J/OL]. 2020[2022-03-12]. <https://arxiv.org/abs/2008.13535v2>. DOI:10.1145/3442381.3450078.

[13] TensorFlow Recommenders[CP/OL]. Google. <https://www.tensorflow.org/recommenders>.

[14] SONG W, SHI C, XIAO Z, 等. AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks[J/OL]. Proceedings of the 28th ACM International Conference on Information and Knowledge Management, 2019: 1161-1170. DOI:10.1145/3357384.3357925.

[15] VASWANI A, SHAZER N, PARMAR N, 等. Attention Is All You Need[J/OL]. arXiv:1706.03762 [cs], 2017[2022-05-07]. <http://arxiv.org/abs/1706.03762>.

[16] tfa.layers.MultiHeadAttention[CP/OL]. Google.

https://github.com/tensorflow/addons/blob/v0.17.0/tensorflow_addons/layers/multihead_attention.py.

[17] ZHOU G, SONG C, ZHU X, 等. Deep Interest Network for Click-Through Rate Prediction[J/OL]. arXiv:1706.06978 [cs, stat], 2018[2022-04-23]. <http://arxiv.org/abs/1706.06978>.

[18] CHEN H, CHEN Y, WANG X, 等. Curriculum Disentangled Recommendation with Noisy Multi-feedback[J]. 13.

[19] GU Y, DING Z, WANG S, 等. Deep Multifaceted Transformers for Multi-objective Ranking in Large-Scale E-commerce Recommender Systems[J]. 2020: 8.

[20] 阿里云开发者. 搜索模型核心技术公开，淘宝如何做用户建模？ [EB/OL].
<https://mp.weixin.qq.com/s/3urKkPfhi4JC7Qe2pLcls>.

[21] QI P, ZHU X, ZHOU G, 等. Search-based User Interest Modeling with Lifelong Sequential Behavior Data for Click-Through Rate Prediction[J/OL]. arXiv:2006.05639 [cs, stat], 2020[2022-04-04].
<http://arxiv.org/abs/2006.05639>.

[22] FAISS[CP/OL]. Meta. <https://ai.facebook.com/tools/faiss/>.

[23] CHEN Q, PEI C, LV S, 等. End-to-End User Behavior Retrieval in Click-Through RatePrediction Model[J/OL]. 2021[2022-04-04]. <https://arxiv.org/abs/2108.04468v1>. DOI:10.48550/arXiv.2108.04468.

[24] 胡可. 广告深度预估技术在美团到店场景下的突破与畅想[EB/OL].
<https://tech.meituan.com/2021/10/14/breakthrough-and-prospect-of-deep-ctr-prediction-in-meituan-ads.html>.