

第5章 召回

本章讲解推荐系统中的召回算法，分如下几个章节展开。

5.1节讲述出现DNN之前，推荐系统中使用的传统召回算法。这些算法主要基于规则和统计，较少训练模型。尽管不如后来的基于深度学习、图卷积的召回算法亮眼，但在某些场景下，仍有用武之地。

从5.2节开始进入各互联网大厂的绝对主力“向量化召回”的领域。向量化召回是一个算法族，其中各类算法在实现细节、应用场景上都有所差别。孤立、教条地学习单个算法，容易让人“只见树木，不见森林”，陷入实现细节而忽略了算法的精髓、本质。所以在5.2节中，作者提出了“向量化召回统一建模框架”，从“如何定义正样本”、“如何定义负样本”、“如何生成Embedding”、“如何定义优化目标”这4个维度，为读者梳理了向量化召回的脉络，帮助读者加深对算法的理解。

然后从5.3节到5.6节，作者根据“向量化召回统一建模框架”，为读者详细分析了Airbnb召回、FM召回、双塔召回、GCN召回等业界常用算法的异同。从中我们可以看到，这些算法，只是在某个维度上进行创新，而在其他维度上未必最优。因此，我们在技术选型时，没必要生搬硬套，而应该根据统一建模框架，博采多家算法之所长，取长补短。这一点，请读者在阅读时倍加注意。

另外，近年来以阿里的Tree-based Deep Model (TDM) [1,2]为代表的新型召回算法，异军突起。TDM算法突破了向量化召回“用户、物料必须解耦建模”的限制，允许模型中候选物料与用户信息发生交叉，大大增强了模型的表达能力。但是在召回中引入交叉结构是一柄双刃剑，它一方面提升了召回结果的质量，另一方面也增加了线上预测的耗时和工程优化、日常维护的难度。窃以为TDM在业界尚非主流，受篇幅所限，本书就不涵盖TDM主题了。感兴趣的同学可以参考文献[1]学习之，另外可以参考文献[3]了解一下TDM实战中可能会遇到的问题。

5.1 传统召回算法

现在各互联网大厂的主流召回都是基于深度学习、图卷积的复杂算法。尽管如此，本书还是决定花费一些章节介绍一下传统召回算法。

原因之一是，复杂的算法，虽然效果更好，但是训练、部署都需要消耗更多的资源，训练时也需要更多的数据。而这些条件，都是一些初创团队所不具备的。这时，传统召回算法，就体现出训练方便、易于部署、易于调试等优势。甚至在一些小项目中，整个推荐算法不需要排序，就只有一个召回模块就足够了。

第二个原因在于，即使是大厂的推荐系统，也并非是完全自动化、智能化的。推荐工程师也需要配合其他业务团队完成一些临时性、运营性的需求。比如，近期业务团队的目标是提升文章的转发分享率。如果要让你的DNN模型配合这个需求，你可以添加一些与转发分享相关的特征，还可以提升“转发”目标在最终损失中的权重。但是，众所周知，DNN是个黑盒，改进模型的效果未必是立竿见影的。

这时，传统召回就显现出巨大的优势。比如，要想提升转发率，

- 你可以新增一路叫“高转发”的召回。先离线统计出每个文章类别下转发率最高的若干篇文章，存进倒排索引；
- 线上来了一个用户请求，根据该用户喜欢的文章分类，找到相应的索引，返回其中高转发的文章；
- 再做一些策略上的调整，使被“高转发”召回的文章，能够在最终返回给用户的结果集中出现得尽量多一些。

由此可以看出，传统召回算法简单直接，使我们能够更敏捷地应对业务需求，在现代推荐系统中仍然有一席用武之地。本节将介绍三类常用的传统召回算法，以及如何合并多路召回的结果。

5.1.1 基于物料属性的倒排索引

离线时将具备相同属性的物料集合起来，每个集合内部按照后验消费指标（比如CTR）降序排列。比如图5-1中，所有带有“坦克”标签的文章组成集合 $T_{tag=tank}$ ，作者A发表的文章组成 $T_{author=A}$ ，当前最火的几篇文章组成 $T_{popular}$ 。这种类似Map<ItemAttribute, ItemSet>的数据结构，被称为倒排索引。

物料属性	物料集合
0 标签“坦克”	doc1,doc2,doc3,doc4
1 作者“A”	doc2,doc3
2 热门爆款	doc1,doc4

图 5-1 倒排索引示意

线上来了一个用户请求，提取用户喜欢的标签、关注的作者等信息，再据此检索倒排索引，把对应的物料集合作为召回结果返回。

5.1.2 基于统计的协同过滤算法

另一类常用的传统召回，是协同过滤算法（Collaborative Filtering, CF），又分为两种

- 基于用户的协同过滤（User CF）：给用户A找到与他有相似爱好的用户B，把B喜欢的东西推荐给A。
- 基于物料的协同过滤（Item CF）：用户A喜欢物料C，找到与C相似的其他物料D，把D推荐给A。

传统的协同过滤，无需训练模型，是完全基于统计的。以Item CF为例[4]，

- 首先，定义用户反馈矩阵 $\mathbf{A} \in R^{m \times n}$ ，m是用户总数，n是物料总数。如果用户u与物料t交互过，则 $A[u, t] = v$ 。v既可以来源于显式交互，比如u给t的打分；也可以来源于隐式交互，让v=1代表u点击过t。A超级稀疏，因为对某个用户u，绝大多数的物料都对他未曾曝光过。
- 再计算 $\mathbf{S} = \mathbf{A}^T \mathbf{A} \in R^{n \times n}$ ， $S[i, j]$ 代表物料i与物料j之间的相似度
- 为用户u召回时， $\mathbf{r}_u = \mathbf{A}[u, :] \mathbf{S}$ ，表示预估出来的 u 对所有物料的喜爱程度。从中选择预估值最大的前K个物料，作为召回结果返回。

从以上流程，可以看出Item CF的优势：

- 相比于庞大数量的用户，物料的数目更少而且稳定，因此 $\mathbf{A}^T \mathbf{A}$ 可以离线提前计算好
- $\mathbf{A}^T \mathbf{A}$ 有非常成熟的MapReduce分布式算法，计算方便
- 还是因为物料数目相对较少， $\mathbf{A}^T \mathbf{A}$ 也不会很大。在利用MapReduce分布式预测时，可以广播到各台 worker机器上，只用Mapper就可以完成分布式预测，避免复杂的Join操作。

感兴趣的读者可以参考文献[4]，其中介绍了一个基于MapReduce实现Item CF的通用框架，支持用cosine、Pearson相关系统、Euclidean距离、Jaccard距离等多种方式计算相似度。

5.1.3 矩阵分解算法

矩阵分解（Matrix Factorization, MF）算法如图 5-2 所示

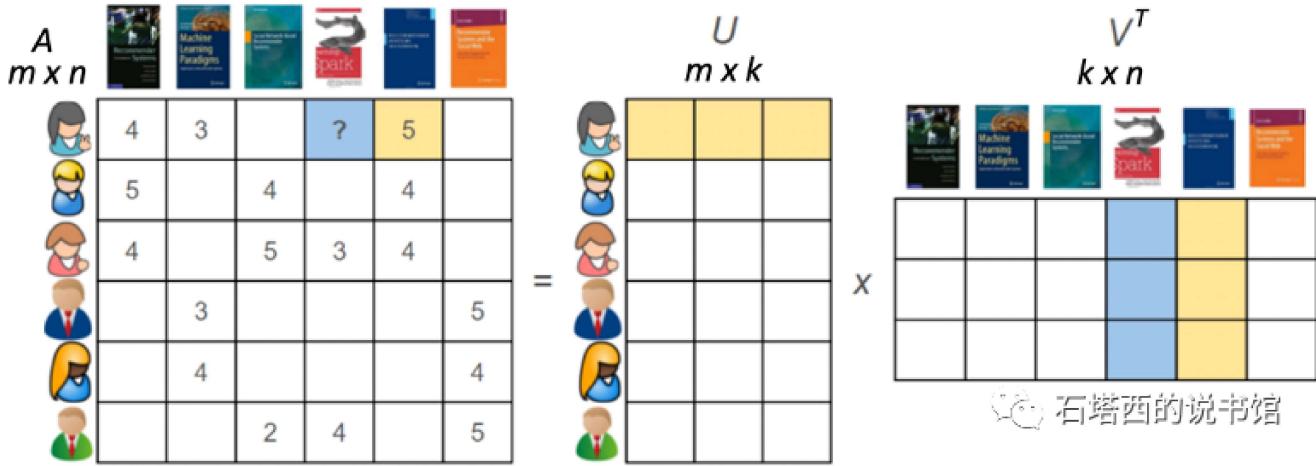


图 5-2 矩阵分解示意

- 和5.1.2节一样，定义矩阵 A 存储用户反馈。上图中 A 有数的地方，代表某用户对某物料做出了反馈，可能是显式的，也可能是隐式的。上图中 A 为空的地方，代表未知反馈，需要MF来预测。
- 再定义用户隐向量矩阵 $U \in R^{m \times k}$ ，和物料隐向量矩阵 $V \in R^{n \times k}$ ，是模型要优化求解的变量。定义预估反馈 $P = UV^T$
- 如果 $A[u, t]$ 有值，那么MF就以为 $A[u, t]$ 目标，迭代优化 U 和 V ，使 $P[u, t]$ 拟合、逼近 $A[u, t]$ 。
- 等训练好 U 、 V 后，对 $A[u, t]$ （真实反馈）为空的位置，就以 $P[u, t]$ （预测反馈）填充。对用户 u ，选择 $P[u, t]$ 最大而且 $A[u, t]$ 为空的前K个物料，作为对 u 的召回结果返回。

MF有两大缺点：

- 一是只能用User ID、Item ID当特征，信息来源受限
- 二是对未曾在训练集中出现过的新用户、新物料，无法给出预估结果。

基于以上两个缺点，MF在互联网大厂中已经不再流行。受篇幅所限，本书不再给出MF的实现细节。感兴趣的读者可以参考文献[5]，Spark中也有现成的API[6]可供调用。

5.1.4 如何合并多路召回

出于“冗余防错”和“互相补充”的目的，推荐系统中通常会有多路召回，这些召回各自的结果，需要合并进一个结果集，再传递给下游的粗排或精排模块。合并时，重复的召回结果只保留一份。如果合并后的结果集太大，超出下游粗精排的处理能力，需要截断。

笔者见过一种合并方法，如代码5-1所示。这种方法，人为给各路召回指定了一个插入顺序。如果前面的召回把结果集插满了，后面召回的结果只能被丢弃。

代码 5-1 合并多路召回结果的错误方式

```

def wrong_merge_recalls():

    merged_results = {}

    # recall_method: 某一路的召回算法名
    # recall_results: 此路召回的结果集

    for recall_method, recall_results in recall_results_list:
        capacity = MAX_NUM_RECALS - len(merged_results) # merged_results还剩余的额度
        if capacity == 0:
            # 已经插满了, 直接返回, 后面的召回的结果被丢弃
            return merged_results

        # 当前召回能插入最终结果集的额度
        quota = min(len(recall_results), capacity)
        # 把当前结果集中的top quota个物料, 插入到最终结果集中
        top_recall_results = recall_results[:quota]
        merged_results.update(top_recall_results)

    return merged_results

```

这种方式的缺点在于

- 人为规定的插入顺序太主观。如果排在前面的召回退化了，而排在后面的召回进步了，怎么办？
- 而且不同用户对不同召回的偏好不同，怎么会有个死板、统一的标准？

所以正确的方式，应该如代码 5-2 所示。

代码 5-2 合并多路召回结果的正确方式

```

def correct_merge_recalls():

    merged_results = {}

    while True:
        # recall_method: 某一路的召回算法名
        # recall_results: 此路召回的结果集

        for recall_method, recall_results in recall_results_list:
            if len(merged_results) == MAX_NUM_RECALS:
                return merged_results # 插满了, 返回

```

```

if len(recall_results) > 0: # 当前召回还有余量
    # 弹出当前召回认为的用户最喜欢的item，插入结果集
    top_item = recall_results.pop()
    merged_results.add(top_item)

```

- 各路召回的结果集，按照预估的用户喜欢程度降序排列，用户最可能喜欢的物料排在最前面。
- 每一轮合并，把所有召回都遍历一遍。每路召回，将各自结果集的第一名弹出，插入合并结果集。
- 重复以上合并过程若干轮，直到最终结果集被插满。

这种合并方式的优点在于，不必刻板地指定各路召回的插入顺序，也就不会出现顺序靠后的召回的结果被全部丢弃的情况，能够确保各路召回的精华肯定会被插入到合并结果集中，从而获得下游的处理机会。

5.2 向量化召回统一建模框架

从本小节开始介绍当前推荐系统中的主流召回算法，向量化召回（Embedding-Based Retrieval，EBR）算法。所谓向量化召回，就是将召回问题，建模成向量空间内的近邻搜索问题。

假设召回问题中包含两类实体Q和T

- 如果Q是用户，T是物料，给用户直接找他喜欢的物料，就是user-to-item召回（简称U2I）；
- 如果Q和T都是物料，那就是item-to-item召回（简称I2I），用于给用户找与他喜欢的物料相似的其他物料，比如“看了又看”场景；
- 如果Q和T都是用户，那就是user-to-item-to-item召回（简称U2I2I），先给当前用户查找与他相似的用户，再把相似用户喜欢的东西推荐给当前用户。

向量化召回的基本套路如下：

- 第一步，训练一个模型M，将Q中的每个实例 q ，和T中的每个实例 t ，都映射到同一个向量空间。
- 第二步，将T中几十万、上百万个实例，喂入模型M，映射成向量。再把这几十万、上百万个向量灌入FAISS[7]或Milvus[8]这样的向量数据库，建立索引。
- 第三步，在线服务时，来了一个Q类的实例 q ，通过模型M将其映射成向量 Emb_q 。再在向量数据库中，通过近似近邻搜索（Approximate Nearest Neighbors，ANN）算法，查找与 Emb_q 最近的 K 个T类的邻居向量 Emb_{t_i} 。这些邻居向量对应的 $t_i (1 \leq i \leq K)$ ，作为召回结果返回。

向量化召回不是单指某一个具体的算法，而是一个庞大的家族，比如：Item2Vec[9]、Youtube的召回算法[10]、Airbnb的召回算法[11]、FM召回、微软的DSSM[12]、双塔模型、百度的孪生网络、阿里的EGES[13]、Pinterest的PinSage[14]、腾讯的GraphTR[15]、……等。这些向量化召回算法，从召回方式上分，涵盖了U2I、I2I、U2U2I三大类；从算法实现上分，有的来自“前DNN”时代，有的基于深度学习，还有基于图算法；从优化目标上分，有的是按照多分类问题来求解，有的基于Learning-To-Rank(LTR)思路来优化。

尽管看上去形态迥异，让人眼花缭乱，但这些向量化召回算法，其实都可以用一套统一的建模框架所囊括。“向量化召回统一建模框架”（Embedding-based Retrieval Unified Framework，ERUF）由4个维度构成，也就是需要我们在建模时回答以下4个问题：

- 如何定义正样本，即哪些 q 和 t 在向量空间内应该相近；
- 如何定义负样本，即哪些 q 和 t 在向量空间内应该较远；
- 如何将 q 和 t 映射成Embedding；
- 如何定义优化目标，即如何制订损失函数；

笔者希望本章介绍的“向量化召回统一建模框架”能够给读者带来两方面的提升：

- 融会贯通。借助UEBRF，读者学习的不再是若干孤立的算法，而是一个算法体系，不仅能加深对现有算法的理解，还能轻松应对未来出现的新算法；
- 取长补短。大多数召回算法，只是在某一两个维度上，进行了创新，而在其他维度上的做法未必是最优的。我们在技术选型时，没必要照搬某个算法的全部，而是借助本章梳理的脉络，博采多家算法之所长，取长补短，组成最适合你的业务场景、数据环境的算法。

5.2.1 如何定义正样本

正样本的定义，即哪些 q 和 t 的向量表示应该相近，取决于不同的召回场景：

- I2I召回： q 和 t 都是物料。比如我们认为同一个用户在同一个“会话”（Session，彼此间隔时间较短的用户行为序列）交互过（e.g., 点击、观看、购买、……等）的两个物料，在向量空间是相近的。这体现的是，两个物料之间的“相似性”；
- U2I召回： q 是用户， t 是物料。一个用户与其交互过的物料，在向量空间中，应该是相近的。这体现的是，用户与物料之间的“匹配性”。
- U2U召回： q 和 t 都是用户。比如使用孪生网络， q 是用户一半的交互历史， t 是同一用户另一半交互历史，二者在向量空间应该是相近的，体现“同一性”。

5.2.2 重点关注负样本

负样本就是在举反例 (q, t_-) ，告诉模型 q 与 t_- 是不匹配的。负样本的要义是，要让模型见识到形形色色、五花八门、不同角度的 " q 与 t_- 之间差异性"，达到让模型"开眼界，见世面"的目的。听起来容易，但是实操方法，会违反许多初入推荐的新手、甚至许多只做过排序的老手的直觉。

负样本主要靠随机采样

笔者曾经仿照Youtube[10]复现过他们的召回算法。当时就特别不理解为什么Youtube不用"曝光未点击"做负样本，而是拿采样出的物料做负样本。而且这样做的还不仅仅Youtube一家，Microsoft的DSSM[12]中的负样本也是随机抽取来的。两篇文章都没说明这样随机抽取负样本的原因。

当时，笔者只有排序方面的经验，而排序是非常讲究所谓的"真负"样本的，即必须拿"曝光未点击"的样本做负样本。以至于还有Above Click的作法，即只拿点击文章以上的未点击文章做负样本，以保证确实给对户曝光过。所以，排序思维根深蒂固的笔者，觉得拿"曝光未点击"做负样本，简直是天经地义，何况还非常诱惑：

- "曝光未点击"是用户偏好的"真实"反馈。而随机抽取的，可能压根就没有曝光过，不能断定用户就一定不喜欢。
- "曝光未点击"的数量有限，处理起来更方便快捷。
- 用"曝光未点击"做负样本，能与排序共享同一套数据处理流程，很多数据都已经处理好了，无须重新开发和生成。
- 碰上接下来会讲到的万能FM模型，拿曝光数据训练出一版模型，既能做排序，又能做召回，一举两得，省时省力。

所以，笔者第一次实践Youtube召回算法时，直接拿"曝光未点击"样本做负样本。训练出来的模型，离线AUC达到0.7+，作为一个特征简化的召回模型，已经算是非常高了，但是线上表现一塌糊涂。排查时发现召回的物料，与用户画像与用户点击历史，完全没有相关性。而当笔者"照着论文画瓢"，拿随机采样的样本做负样本，线上结果却非常好。

拿"曝光未点击"哪里出错了？排序时的金科玉律，怎么到了召回环节就失灵了？这是因为笔者自以为是的做法，违反了机器学习的一条最最基本的原则铁律，就是**"离线训练时的数据分布，应该与线上服务时的数据分布，保持一致"**。

以前，我们谈到召回与排序的不同，往往只强调速度。即因为召回的候选集更大，所以要求预测速度更快，而模型可以简单一些，这是只知其一。而另一个往往被忽视的差异，就是排序与召回的候选集截然不同：

- 排序的目标是，从用户可能喜欢的当中，挑选出用户最喜欢的，是为了优中选优。排序面对的候选物料，是已经和用户兴趣比较匹配的优质集合。
- 召回的目标，是将用户可能喜欢的，和海量跟用户兴趣“八杆子打不着”的，分隔开，所以，召回所面对的候选物料集合，可谓“鱼龙混杂，良莠不齐”。

所以，要求喂入召回模型的样本，既要让模型见过最匹配 $\langle user, item \rangle$ 组合，也要让模型见过最不靠谱的 $\langle user, item \rangle$ 组合，才能让模型达到“开眼界、见世面”的目的，从而在“大是大非”上不犯错误。

- 最匹配的 $\langle user, item \rangle$ ，没有异议，就是用户与他点击过的物料。
- 对用户最不靠谱的物料，是对他曝光但未点击的物料吗？

这就牵扯到推荐系统里常见的“样本选择偏差”（Sample Selection Bias，SSB）问题，即我们从线上日志获得的曝光样本，已经是被上一版本的召回、粗排、精排替用户层层筛选过的，已经是对用户“比较靠谱”的物料了。拿这样的样本训练出来的模型做召回，一叶障目，只见树木，不见森林。

就好比：一个老学究，一辈子待在象牙塔中，遍查古籍，能够一眼看出一件文物是“西周”的还是“东周”的。但是这样的老学究，到了潘家园，却非常可能打眼，识别不出一件“上周”生产的假货。为什么？因为他一辈子只见过“老的”和“更老的”，“新的”被象牙塔那阅人无数的门卫给挡回去了，他压根就没见识过。

因此，为了让模型“开眼界、见世面”，领教最不靠谱的 $\langle q, t \rangle$ 组合，**喂给召回模型的负样本，主要依靠随机采样生成**。特别是在U2I召回场景中，**坚决不能（只）拿“曝光未点击”当负样本**。

细心的读者可能注意到了，笔者在表述中对“只”字做了保留。召回模型的负样本全部由“曝光未点击”组成，是绝对不行的。但是，“以随机负采样为主，以曝光未点击为辅”的混合方案，能否行得通？业界尚无定论。以笔者和Facebook的经验，都认为“曝光未点击”样本就是“鸡肋”，于召回模型无益。而在另外一些实践中，认为“曝光未点击”是Hard Negative（下一节会讲到），能够提升模型对细节的分辨能力。在随机负采样之外，加入“曝光未点击”能否提升召回性能？请读者在工作场景中亲自实践一下便知，毕竟推荐算法也属于实验科学，实验是检验算法成败的（唯一）标准。

随机负采样大有讲究

综上所述，喂入召回模型的负样本主要依靠随机采样生成。采样听起来简单，实践起来也大有讲究与门道。

- 推荐系统中不可避免地存在“2 ~ 8定律”，即20%的热门物料占据了80%的曝光流量。因此，在U2I、I2I召回中，正例中的物料以少数热门物料为主，可能带偏模型对所有用户只出热门物料，损害推荐系统的个性化与多样性。这就要求我们进行针对性的纠偏，具体方法在以下章节中再详细讨论。
- 最让模型“开眼界、见世面”的负采样方法，是在所有物料组成的大库中进行随机采样。但是考虑到推荐系统中的物料成百上千万，大库采样的方案代价太高，更是无法满足在线学习的实时性要求。因此在业

界也提出许多近似折中方案，在后续章节中也会加以详细讨论。

另外，只通过随机采样获得负样本，可能导致模型的精度不足。举个例子，假如你要训练一个“相似图片召回”算法。当 q 是一只狗时，正例 t_+ 是另外一只狗。负例 t_- ，在所有动物图片中随机采样得到，大概率是抽到猫、大象、乌鸦、海豚等图片。这些随机负样本，能够让模型“大开眼界”，从而快速“去伪存真”。但是猫、大象、乌鸦、海豚等随机负例，都与正例（另一只狗）相差太大，使模型觉得只要能分辨粗粒度差异就足够了，没有动力去注意细节。这些负样本被称为Easy Negative。

因此，我们要为“ $q=狗, t_+=另一只狗$ ”配备一些狼、狐狸的图片做负例。这些负例称为Hard Negative，它们与正例还有几分相似，能给模型增加难度，迫使其关注细节。至于如何挑选，业界也有不同作法，将在后续章节中详细阐述。

需要特别强调的是，Hard negative并非要替代随机采样得到的Easy Negative，而是作为Easy Negative的补充。在数量上，负样本还是应该以Easy Negative为主，Facebook的经验^[16]是将比例维持在 **Easy:Hard=100:1**。毕竟线上召回时，候选集中绝大多数的物料都是与用户八杆子打不着的，保证Easy Negative的数量优势，才能Hold住召回模型的及格线。

本书花费了大量笔墨论述了召回中的负样本策略，就是向读者想传递这样的观点：**如果说排序是特征的艺术，那么召回就是样本的艺术，特别是负样本的艺术**。做召回，“**负样本为王**”，负样本的选择对于算法成败是决定性的：选对了，你就稳过及格线；选错了，之后的特征工程、模型设计、推理优化都是南辕北辙，无本之木，平添无用功罢了。

5.2.3 解耦生成Embedding

以U2I召回为例，虽然与排序一样，都是建模用户与物料之间的匹配程度，但是二者在样本、特征、模型上都有显著不同。上一节，笔者详细论述了二者在样本选择上的区别，这一节将论述二者在特征、模型上的区别。简言之，就是：**排序鼓励交叉，召回要求解耦**。

排序鼓励交叉

- 特征上，排序除了单独使用用户和物料特征，最重要还使用了大量的交叉统计特征，比如“用户标签与物料标签的重合度”。这类交叉统计特征是衡量“用户物料匹配程度”的最强信号，但是它们无法离线事先计算好，服务于所有用户请求。
- 模型上，排序一般将用户特征、物料特征、交叉统计特征拼接成一个大向量，喂入DNN，让三类特征层层交叉。从第一个全链接层（Fully Connection, FC）之后，你就已经无法分辨，它输出的向量中，哪几位属于用户信息？哪几位属于物料信息？

召回要求解耦

排序之所以允许、鼓励交叉，还是因为它的候选集比较小，最多不过一两千个。换成召回那样，要面对十万、百万级别的海量候选物料，如果让每个用户与每个候选物料都计算交叉统计特征，都过一遍DNN那样的复杂运算，是无论如何也无法满足线上的实时性要求的。

所以，召回必须解耦、隔离用户信息与物料信息：

- 特征上，召回无法使用，用户物料之间的交叉统计特征（比如：用户标签与物料标签之间的重合度）。
- 模型上，召回不能将用户特征、物料特征一古脑扔进DNN，两者必须独立发展。用户子模型，只利用用户特征，生成用户向量 Emb_{user} 。物料子模型，只利用物料特征，生成物料向量 Emb_{item} 。只允许最后计算 $\text{Emb}_{\text{user}} \cdot \text{Emb}_{\text{item}}$ 或 $\text{cosine}(\text{Emb}_{\text{user}}, \text{Emb}_{\text{item}})$ 时，才产生用户信息与物料信息的唯一一次交叉。

这样解耦的目的

- 离线时，不必知道是哪个用户发出请求，提前计算好生成百万、千万量级的物料向量（i.e., Emb_{item} ），灌入FAISS并建立好索引；
- 在线时，独立生成用户向量（i.e., Emb_{user} ），去FAISS里利用近似近邻搜索算法（ANN），快速搜索出与 Emb_{user} 接近的 Emb_{item} 。避免让每个用户与几百万、上千万的候选物料，逐一进行“计算交叉特征”和“通过DNN”这样复杂耗时的操作。

5.2.4 如何定义优化目标

无论哪种召回方式，为了能够与FAISS/Milvus等向量数据库兼容，方便线上服务快速搜索近邻向量，我们拿 q 的embedding和 t 的embedding的点积或cosine结果，表示 q 与 t 之间的匹配程度。显然，点积或cosine越大，代表 q 与 t 越匹配。

至于如何定义损失函数，召回也与精排有很大不同。精排追求的预测值的“绝对准确性”，它使用Binary Cross-Entropy Loss，力图将每条样本的CTR/CVR预测得越准越好。这是因为：

- 精排的候选集小，而且每个负样本都来自用户的真实反馈，因此精排有条件把每个负样本的CTR/CVR都预测准确。
- 精排打分的准确性，是下游任务（比如广告计费、打散重排等）成败的关键。因此，如果精排打分只能保证“将用户喜欢的排在前面”的“相对准确性”，那是远远不够的。

而召回的优化目标、损失函数主要考虑的是排序的“相对准确性”

- 召回的候选集非常庞大，而且只有正样本来自用户真实反馈，而负样本往往都未曾向用户曝光过，所以召回没条件对负样本要求预测值绝对准确。召回常用的一类损失函数是多分类的Softmax Loss，只要求把正样本的概率预测得越高越好。
- 召回的作用是筛选候选集，只要能把用户喜欢的排在前面就好。因此，召回中另一类常用的损失函数，遵循Learning-To-Rank (LTR) 思想，不求预估值的绝对准确，只追求排序的相对准确性。

接下来介绍召回中常用的几种损失函数。提前声明一下：

- 为了方便行文，以上讲解都是针对U2I召回而举例，但是对U2U、I2I也同样适用。
- 以下几种Loss都被广泛应用。至于哪种Loss能为读者的项目带来效果，还是需要读者自己编码实现后，让离线和在线实验给出答案。

NCE Loss

如前所述，召回常用的一类损失函数是Softmax Loss。以U2I召回为例，公式如(5-1)所示，优化目标是使用用户 u_i 在T中选中正例 t_i 的概率 $\frac{\exp(u_i \cdot t_i)}{\sum_{j \in T} \exp(u_i \cdot t_j)}$ 越高越好。

$$L_{Softmax} = -\frac{1}{|B|} \sum_{(u_i, t_i) \in B} \log \frac{\exp(\mathbf{u}_i \cdot \mathbf{t}_i)}{\sum_{j \in T} \exp(\mathbf{u}_i \cdot \mathbf{t}_j)} \quad (5-1)$$

- B代表一个Batch，其中第*i*条样本 (u_i, t_i) 由用户 u_i 和他交互过的正例物料 t_i 组成。
- \mathbf{u}_i 表示模型对用户 u_i 生成的向量表示。
- \mathbf{t}_i 表示模型对物料 t_i 生成的向量表示。
- Softmax Loss将召回看成一个超大规模的多分类问题，每个候选物料算一个类别，所有候选物料组成的集合用T表示。

通过公式(5-1)可以看到，为最小化损失，要求分子 $\exp(\mathbf{u}_i \cdot \mathbf{t}_i)$ ，即用户与正例物料的匹配度，尽可能大；而分母中的 $\sum_{j \in T} \exp(\mathbf{u}_i \cdot \mathbf{t}_j)$ ，即用户与所有负例物料的总匹配度，尽可能小。体现出召回不追求绝对准确性，只追求排序的相对准确性的设计特点。但是，Softmax Loss中的分母 $\sum_{j \in T} \exp(\mathbf{u}_i \cdot \mathbf{t}_j)$ 要计算用户向量与T中所有物料向量的点积，而|T|属于百万或千万这个量级，分母的计算量大到离谱。

Noise Contrastive Estimation (NCE) 是简化分母的一种思路，它将召回原始的“超大规模多分类问题”简化为一系列“区别样本是否来自噪声”的二分类问题^[17]。以点击场景下的U2I召回为例，给定一个用户 u ，他点击的物料，组成正例集合 R ，再给 u 按照采样一部分物料当负例，组成负例集合 S ，也就是所谓的“噪声”。这样一来，用户 u 的候选物料集合，不再像原始Softmax那样是整个物料库T，而是一个有限集合 $C = R \cup S$ 。而NCE的二分类问题变成，对于每个候选物料 $\forall t \in C$ ，如果 t 属于 R ， (u, t) 算一个正样本，如果 t 属于 S ， (u, t) 算一个负样本。

我们用 $G(u, t)$ 表示一条样本 (u, t) 是正样本的logit，NCE计算 $G(u, t)$ 如公式(5-2)所示。

$$G(u, t) = \log \frac{P(t|u)}{Q(t|u)} = \log P(t|u) - \log Q(t|u) \quad (5-2)$$

- u 、 t 分别表示一条样本中的用户与物料。
- $P(t|u)$ 代表用户 u 真心喜欢物料 t 的概率，是召回模型的建模目标。
- $Q(t|u)$ 表示用户 u 随机点击物料 t 的概率，是一个超参，比如它可以在全体候选物料上的均匀采样，也可以在某些物料上进行重点采样。
- $G(t|u)$ 是 $P(t|u)$ 与 $Q(t|u)$ 的比值，这也就是NCE中Contrastive（对比）一词的含意

在向量化召回中，我们用向量点积来建模 $P(t|u)$ ，代入公式(5-2)，得到公式(5-3)。

$$G(u, t) = \mathbf{u} \cdot \mathbf{t} - \log Q(t|u) \quad (5-3)$$

- \mathbf{u} 是召回模型为用户 u 生成的向量表示。
- \mathbf{t} 是召回模型为物料 t 生成的向量表示。
- 点积 $\mathbf{u} \cdot \mathbf{t}$ 表示用户与物料之间的匹配度，代替公式中的 $\log P(t|u)$ 。

公式(5-3)告诉我们，由于负样本来自有限的随机采样，所以在我们拿 $\mathbf{u} \cdot \mathbf{t}$ 计算用户与物料之间的匹配度后，还要根据负采样概率 $Q(t|u)$ 进行修正。前面讲到过，由于少数热门物料占据大量的曝光与点击，可能带偏模型对所有用户只出热门物料，从而丧失推荐的个性化。修正的目的是为了打压召回结果中的热门物料，所以负采样概率 $Q(t|u)$ 应该是物料热度的递增函数。这样一来，物料 t 的热度越大，那么它被 u 点击的“个性化”成分越弱，被 $-\log Q(t|u)$ 修正得越厉害。具体如何设置 $Q(t|u)$ ，接下来我们讲解具体算法的时候，再加以详细说明。另外，需要提醒的是， $-\log Q(t|u)$ 修正只发生在训练阶段，预测时还是只拿 $\mathbf{u} \cdot \mathbf{t}$ 计算用户与物料的匹配度，和FAISS/Milvus等向量数据库兼容。

根据 $G(u, t)$ 计算Binary Cross-Entropy Loss，就得到NCE Loss，如公式(5-4)所示。

$$\begin{aligned} L_{NCE} &= \frac{-1}{|B|} \sum_{(u_i, t_i) \in B} \left[\log(\sigma(G(u_i, t_i))) + \sum_{j \in S_i} \log(1 - \sigma(G(u_i, t_j))) \right] \\ &= \frac{1}{|B|} \sum_{(u_i, t_i) \in B} \left[\log(1 + \exp(-G(u_i, t_i))) + \sum_{j \in S_i} \log(1 + \exp(G(u_i, t_j))) \right] \end{aligned} \quad (5-4)$$

- B 代表一个Batch，第 i 条样本由用户 u_i 和他点击过的一个物料 t_i 组成。
- S_i 表示替随机采集一些物料作为他的负样本。
- σ 表示sigmoid函数。
- 其中用到的 $G(u, t)$ 如公式(5-3)所示，注意其中的 $-\log Q(t|u)$ 修正项。

在实际操作中，为了进一步简化计算，干脆就在公式中忽略掉 $-\log Q(t|u)$ 修正项，还拿原始 $\mathbf{u} \cdot \mathbf{t}$ 的表示用户与物料之间的匹配度，于是得到了Negative Sampling Loss（NEG Loss），如公式(5-5)所示。

$$L_{NEG} = \frac{1}{|B|} \sum_{(u_i, t_i) \in B} \left[\log(1 + \exp(-u_i \cdot t_i)) + \sum_{j \in S_i} \log(1 + \exp(u_i \cdot t_j)) \right] \quad (5-5)$$

NEG Loss的优点是计算简便，而它的缺点是不像NCE那样有非常强的理论保证。已经证明，如果采样的负例足够多，NCE的梯度与原始超大规模Softmax的梯度趋于一致，但是NEG没有这种性质。但是由于我们的目标并不是为了将概率分布学准确，而是为了学习出高质量的用户向量与物料向量，因此NEG在理论上的瑕疵并不是大问题，照样在召回场景中被广泛应用。

Sampled Softmax Loss

还是拿用点击场景下的U2I召回为例。给定一个用户 u 和他点击的物料 p ，再给 u 按照 $Q(t|u)$ 的概率采样一批负样本 S 。原始超大规模Softmax需要估计整个物料库 T 中每个物料被 u 点击的概率，负采样后变成Sampled Softmax问题[17]，即在候选集 $C = \{p\} \cup S$ 中， $\forall t \in C$ 被 u 点击的概率有多大，即建模 $P(t|u, C)$ 。

首先将 $P(t|u, C)$ 根据条件概率展开成公式(5-6)。

$$P(t|u, C) = \frac{P(t, C|u)}{P(C|u)} \quad (5-6)$$

再对公式的分子按照Bayes公式展开，得到公式(5-7)，其中 $P(t|u)$ 是召回模型的建模目标。

$$P(t, C|u) = P(C|t, u) \times P(t|u) \quad (5-7)$$

再来看公式(5-7)中的 $P(C|t, u)$ ，它表示给定了用户 u 和被点击物料 t 的前提下，构建出整个候选集 C 的概率。由于假设 t 是被点击的，这也就意味着 $C - \{t\}$ 都是负采样得到的，而且 C 之外的其他物料 $T - C$ 都未被负采样到，由此展开 $P(C|t, u)$ 得到公式(5-8)。

$$\begin{aligned} P(C|t, u) &= \prod_{t' \in C - \{t\}} Q(t'|u) \times \prod_{t' \in T - C} (1 - Q(t'|u)) \\ &= \frac{1}{Q(t|u)} \times Q(t|u) \times \prod_{t' \in C - \{t\}} Q(t'|u) \times \prod_{t' \in T - C} (1 - Q(t'|u)) \\ &= \frac{1}{Q(t|u)} \times \prod_{t' \in C} Q(t'|u) \times \prod_{t' \in T - C} (1 - Q(t'|u)) \end{aligned} \quad (5-8)$$

把公式(5-6)、公式(5-7)、公式(5-8)结合起来，得到公式(5-9)。

$$\begin{aligned} P(t|u, C) &= \frac{P(t, C|u)}{P(C|u)} \\ &= \frac{P(C|t, u) \times P(t|u)}{P(C|u)} \\ &= \frac{P(t|u) \times \frac{1}{Q(t|u)} \times \prod_{t' \in C} Q(t'|u) \times \prod_{t' \in T - C} (1 - Q(t'|u))}{P(C|u)} \\ &= \frac{P(t|u)}{Q(t|u)} \times \frac{\prod_{t' \in C} Q(t'|u) \times \prod_{t' \in T - C} (1 - Q(t'|u))}{P(C|u)} \end{aligned} \quad (5-9)$$

公式(5-9)中的 $\frac{\prod_{t' \in C} Q(t'|u) \times \prod_{t' \in T - C} (1 - Q(t'|u))}{P(C|u)}$ 一项，只与 u 和他的候选集 C 有关，而与建模变量 t 在 C 中的取值无关，可以简写成 $K(u, C)$ ，于是公式(5-9)变成了公式(5-10)。

$$P(t|u, C) = \frac{P(t|u)}{Q(t|u)} \times K(u, C) \quad (5-10)$$

$$\Rightarrow \log P(t|u, C) = \log P(t|u) - \log Q(t|u) + \log K(u, C)$$

类似NCE的推导过程， $\log P(t|u)$ 是我们的建模目标，可以拿用户向量与物料向量的点积来替换。而 $\log K(u, C)$ 与建模变量 t 在 C 中的取值无关，不影响Softmax结果，可以忽略掉。我们用 $G(u, t)$ 表示用户 u 在他的候选集 C 中点击物料 t 的logit，将公式(5-10)写成公式(5-11)，其中符号含义可以参考公式(5-3)。

$$G(u, t) = \log P(t|u, C) = \mathbf{u} \cdot \mathbf{t} - \log Q(t|u) \quad (5-11)$$

公式(5-11)与公式(5-3)类似，同样告诉我们拿 $\mathbf{u} \cdot \mathbf{t}$ 计算用户与物料之间的匹配度后，还要根据负采样概率 $Q(t|u)$ 进行修正。而且与NCE Loss一样，修正只发生在训练阶段，预测时还是只计算 $\mathbf{u} \cdot \mathbf{t}$ 即可，和FAISS/Milvus等向量数据库兼容。

将公式(5-11)代入Softmax的公式中，计算Sampled Softmax如公式(5-12)所示。

$$L_{\text{SampledSoftmax}} = -\frac{1}{|B|} \sum_{(u_i, t_i) \in B} \log \frac{\exp(G(u_i, t_i))}{\exp(G(u_i, t_i)) + \sum_{j \in S_i} \exp(G(u_i, t_j))} \quad (5-12)$$

- B代表一个Batch。
- u_i 表示用户， t_i 表示 u_i 点击过的一个物料， S_i 表示给 u_i 采样生成的负样本集合。
- $G(u_i, t_i)$ 的计算如公式(5-11)所示，注意其中的 $-\log Q(t|u)$ 修正项。

Pairwise Loss

Pairwise Loss是Learning-To-Rank的一种实现。以U2I为例：

- 一个样本是由用户、正例物料、随机采样的负例物料组成的三元组 (u_i, t_{i+}, t_{i-})
- 优化目标是，针对同一个用户 u ，正例物料 t_+ 与他的匹配程度，要远远高于，负例物料 t_- 与他的匹配程度，即 $\text{Sim}(u, t_+) \gg \text{Sim}(u, t_-)$ 。

一种表示“远远高于”的方法是使用Marginal Hinge Loss，如公式(5-13)所示。

$$L_{\text{Hinge}} = \frac{1}{|B|} \sum_{(u_i, t_{i+}, t_{i-}) \in B} \max(0, m - \mathbf{u}_i \cdot \mathbf{t}_{i+} + \mathbf{u}_i \cdot \mathbf{t}_{i-}) \quad (5-13)$$

- B代表一个Batch，其中第*i*条样本由三元组 (u_i, t_{i+}, t_{i-}) 组成。
- u_i 表示用户， t_{i+} 是 u_i 点击过的一个物料作为正例， t_{i-} 是随机采样得到的一个物料作为负例。
- \mathbf{u}_i 、 \mathbf{t}_{i+} 、 \mathbf{t}_{i-} 分别是召回模型给 u_i 、 t_{i+} 、 t_{i-} 生成的向量表示。
- m 代表阈值（Margin），是一个超参数。
- 整个公式要求，用户 u 与他点击过的物料 t_+ 的匹配程度，要比一个随机采样得到的物料 t_- 的匹配程度，高出一个给定的阈值。

如果嫌调节超参数太麻烦，可以使用Bayesian Personalized Ranking(BPR) Loss。BPR的思想是给定一个由用户、正例物料、随机采样的负例物料组成三元组 (u_i, t_{i+}, t_{i-}) ，针对用户 u_i 的正确排序（将 t_{i+} 排在 t_{i-} 前面）的概率是 $P_{CorrectOrder} = \text{sigmoid}(\mathbf{u}_i \cdot \mathbf{t}_{i+} - \mathbf{u}_i \cdot \mathbf{t}_{i-})$ ，BPR Loss就是要将这一正确排序的概率最大化。因为 $P_{CorrectOrder}$ 对应的真实label永远是1，因此将 $P_{CorrectOrder}$ 代入Binary Cross-Entropy的公式，得到BPR Loss如公式(5-14)所示，符号含义参考公式(5-13)。

$$\begin{aligned} L_{BPR} &= \frac{1}{|B|} \sum_{(u_i, t_{i+}, t_{i-}) \in B} -\log(P_{CorrectOrder}^{(i)}) \\ &= \frac{1}{|B|} \sum_{(u_i, t_{i+}, t_{i-}) \in B} \log(1 + \exp(\mathbf{u}_i \cdot \mathbf{t}_{i-} - \mathbf{u}_i \cdot \mathbf{t}_{i+})) \end{aligned} \quad (5-14)$$

5.3 取经Word2Vec

之前讲Deep Interest Network时，笔者就曾经提到过，很多推荐算法都借鉴自NLP领域，本节要讲的“类Word2Vec”召回算法，就是这方面的又一个经典案例。

先简单介绍一下Word2Vec算法[18]，它的目标是为每个单词学习到能够表征其语义的稠密向量，即word embedding。如果使用Skip-Gram算法来达成这一目标，学习的方法就是给定一个中心词 w ，预测哪些词 o 能够出现在 w 的上下文（Context）中，与 w 搭配使用。比如给定一句话 "*The quick brown fox jumps over the lazy dog*"，当我们选中fox当中心词，并且取宽度为1的窗口作为上下文的范围，Word2Vec的训练目标就是使"fox, brown"和"fox, jump"这样的单词组合出现的概率越大越好。写成数学公式，Word2Vec理论上的损失函数如公式(5-15)所示。

$$L_{word2vec} = -\frac{1}{|B|} \sum_{(w_i, c_i) \in B} \log \frac{\exp(\mathbf{w}_i \cdot \mathbf{c}_i)}{\sum_{j \in V} \exp(\mathbf{w}_i \cdot \mathbf{c}_j)} \quad (5-15)$$

- B 代表一个Batch，其中一条样本 (w_i, c_i) 由一个中心词 w_i 和上下文 c_i 组成。
- V 是所有单词组成的集合。
- \mathbf{w}_i 、 \mathbf{c}_i 分别是 w_i 、 c_i 的Embedding，二者的点积 $\mathbf{w}_i \cdot \mathbf{c}_i$ 越大，说明关联性越强，越有可能出现在一个上下文中。

从公式(5-15)中可以看到，每次计算分母都要遍历一遍 V 中所有单词，计算量太大。按照第5.2.4节介绍的方法简化Softmax公式，Word2Vec可以通过最小化NEG Loss训练完成，如公式(5-16)所示。

$$L_{word2vec} = \frac{1}{|B|} \sum_{(w_i, c_i) \in B} \left[\log(1 + \exp(-\mathbf{w}_i \cdot \mathbf{c}_i)) + \sum_{j \in S_i} \log(1 + \exp(\mathbf{w}_i \cdot \mathbf{c}_j)) \right] \quad (5-16)$$

- S_i 表示给中心词 w_i 随机采样得到的一批单词，作为它的负样本。

- 其他符号含义参考公式(5-15)。

5.3.1 最简单的Item2Vec

将Word2Vec运用到推荐领域最直接的算法就是Item2Vec[9]。Item2Vec算法中，将用户的某一个行为序列（比如用户在一个Session内点击过的物料组成的序列）当成一个句子，序列中的每个Item ID当成一个单词。按照以上方式收集完训练数据后，直接套用Word2Vec训练，训练完成后，就能得到每个物料的embedding，可以用于I2I召回。

用"向量召回统一框架"理解Item2Vec

- **如何定义正样本？** Item2Vec认为被同一个用户在同一个Session交互过的物料，彼此之间应该是相似的，它们的向量应该是相近的。但是考虑到，让一个序列内部的物料两两组合，生成的正样本太多了。因此Item2Vec照搬Word2Vec，也采用滑窗，即只在某个物料前后出现的其他物料才认为彼此相似，成为正样本。
- **如何定义负样本？** 照搬Word2Vec，从整个物料库中随机采样一部分物料，与当前物料组合成负样本。
- **如何Embedding？** 没有采用模型，只是定义了一个待学习的矩阵 $V \in R^{|T| \times d}$ ， $|T|$ 是所有物料的个数， d 是Embedding的长度。第*i*个物料的embedding就是 V 矩阵的第*i*行。
- **如何定义损失函数？** 照搬Word2Vec的NEG Loss，见公式(5-5)。

基于TensorFlow实现NCE Loss

Item2Vec（或它的原型Word2Vec）的核心，比如负采样、概率修正、向量点积等，都可以直接调用TensorFlow自带的nce_loss函数^[19]来实现。nce_loss的代码和注释如代码5-3所示：

代码 5-3 TensorFlow自带的nce_loss实现

```
def nce_loss(weights,
             biases,
             labels,
             inputs,
             num_sampled,
             num_classes,
             num_true=1,.....):
```

```
weights: 待优化的矩阵, 形状[num_classes, dim]。可以理解为所有item embedding矩阵, 此时num_classes=1

biases: 待优化变量, [num_classes]。每个item还有自己的bias, 与user无关, 代表自己本身的受欢迎程度。

labels: 正例的item ids, [batch_size,num_true]的整数矩阵。center item拥有的最多num_true个positive

inputs: 输入的[batch_size, dim]矩阵, 可以认为是center item embedding

num_sampled: 整个batch要采集多少负样本

num_classes: 在i2i中, 可以理解成所有item的个数

num_true: 一条样本中有几个正例, 一般就是1

"""

# Logits: [batch_size, num_true + num_sampled]的float矩阵

# Labels: 与Logits相同形状, 如果num_true=1的话, 每行就是[1, 0, 0, ..., 0]的形式
logits, labels = _compute_sampled_logits(...)

# sampled_losses: 形状与logits相同, 也是[batch_size, num_true + num_sampled]

# 一行样本包含num_true个正例和num_sampled个负例
# 所以一行样本也有num_true + num_sampled个sigmoid Loss
sampled_losses = sigmoid_cross_entropy_with_logits(
    labels=labels,
    logits=logits,
    name="sampled_losses")

# 把每行样本的num_true + num_sampled个sigmoid Loss相加

return sum_rows(sampled_losses)
```

其中的实现细节都在 `_compute_sampled_logits` 函数中，它的代码和注释如代码5-4所示：

代码 5-4 TensorFlow 中的 `compute_sampled_logits` 函数

....

输入:

weights: 待优化的矩阵, 形状[num_classes, dim]。可以理解为所有item embedding矩阵, 那时num_classes = num_items。
biases: 待优化变量, [num_classes]。每个item还有自己的bias, 与user无关, 代表自己的受欢迎程度。
labels: 正例的item ids, [batch_size, num_true]的整数矩阵。center item拥有的最多num_true个positive samples。
inputs: 输入的[batch_size, dim]矩阵, 可以认为是center item embedding
num_sampled: 整个batch要采集多少负样本
num_classes: 在I2I中, 可以理解成所有item的个数
num_true: 一条样本中有几个正例, 一般就是1
subtract_log_q: 是否要对匹配度, 进行修正。如果是NEG Loss, 关闭此选项。
remove_accidental_hits: 如果采样到的某个负例, 恰好等于正例, 是否要补救

输出:

out_logits: [batch_size, num_true + num_sampled]
out_labels: 与out_logits同形状

....

```
# Labels原来是[batch_size, num_true]的int矩阵
# reshape成[batch_size * num_true]的数组
labels_flat = array_ops.reshape(labels, [-1])

# ----- 负采样
# 如果没有提供负例, 根据Log-uniform进行负采样
# 采样公式: P(class) = (Log(class + 2) - Log(class + 1)) / Log(range_max + 1)
# 在I2I场景下, class可以理解为item id, 排名靠前的item被采样到的概率越大
# 所以, 为了打压高热item, item id编号必须根据item的热度降序编号
# 越热门的item, 排前越靠前, 被负采样到的概率越高

if sampled_values isNone:
    sampled_values = candidate_sampling_ops.log_uniform_candidate_sampler(
        true_classes=labels,# 正例的item ids
        num_true=num_true,
        num_sampled=num_sampled,
        unique=True,
        range_max=num_classes,
        seed=seed)

# sampled: [num_sampled], 一个batch内的所有正样本, 共享一批负样本
# true_expected_count: [batch_size, num_true], 正例在log-uniform采样分布中的概率, 接下来修正loss
# sampled_expected_count: [num_sampled], 负例在log-uniform采样分布中的概率, 接下来修正logit时
sampled, true_expected_count, sampled_expected_count = (
    array_ops.stop_gradient(s) for s in sampled_values)
```

```

# ----- Embedding

# labels_flat is a [batch_size * num_true] tensor
# sampled is a [num_sampled] int tensor
# all_ids: [batch_size * num_true + num_sampled]的整数数组, 集中了所有正负item ids
all_ids = array_ops.concat([labels_flat, sampled], 0)
# 给batch中出现的所有item, 无论正负, 进行embedding
all_w = embedding_ops.embedding_lookup(weights, all_ids, ...)

# true_w: [batch_size * num_true, dim]
# 从all_w中抽取出对应正例的item embedding
true_w = array_ops.slice(all_w, [0, 0],
                        array_ops.stack([array_ops.shape(labels_flat)[0], -1]))

# sampled_w: [num_sampled, dim]
# 从all_w中抽取出对应负例的item embedding
sampled_w = array_ops.slice(all_w,
                            array_ops.stack([array_ops.shape(labels_flat)[0], 0]), [-1, -1])

# ----- 计算center item与每个negative context item的匹配度
# inputs: 可以理解成center item embedding, [batch_size, dim]
# sampled_w: 负例item的embedding, [num_sampled, dim]
# sampled_logits: [batch_size, num_sampled]
sampled_logits = math_ops.matmul(inputs, sampled_w, transpose_b=True)

# ----- 计算center item与每个positive context item的匹配度
# inputs: 可以理解成center item embedding, [batch_size, dim]
# true_w: 正例item embedding, [batch_size * num_true, dim]
# row_wise_dots: 是element-wise相乘的结果, [batch_size, num_true, dim]
.....
row_wise_dots = math_ops.multiply(
    array_ops.expand_dims(inputs, 1),
    array_ops.reshape(true_w, new_true_w_shape))
.....
# _sum_rows是把所有dim上的乘积相加, 得到dot-product的结果
# true_logits: [batch_size, num_true]
true_logits = array_ops.reshape(_sum_rows(dots_as_matrix), [-1, num_true])
.....
```

----- 修正结果

如果采样到的负例, 恰好也是正例, 就要补救

```

if remove_accidental_hits:
    .....
    # 补救方法是在冲突的位置(sparse_indices)的负例Logits(sampled_Logits)
    # 加上一个非常大的负数acc_weights (值为-FLOAT_MAX)
    # 这样在计算softmax时, 相应位置上的负例对应的exp值=0, 就不起作用了
    sampled_logits += gen_sparse_ops.sparse_to_dense(
        sparse_indices,
        sampled_logits_shape,
        acc_weights,
        default_value=0.0,
        validate_indices=False)

if subtract_log_q: # 如果是NEG Loss, subtract_Log_q=False
    # 对匹配度做修正, 对应上边公式中的
    #  $G(x,y)=F(x,y)-\log Q(y/x)$ 
    # item热度越高, 被修正得越多
    true_logits -= math_ops.log(true_expected_count)
    sampled_logits -= math_ops.log(sampled_expected_count)

# ----- 返回结果
# true_Logits: [batch_size, num_true]
# sampled_Logits: [batch_size, num_sampled]
# out_Logits: [batch_size, num_true + num_sampled]
out_logits = array_ops.concat([true_logits, sampled_logits], 1)

# We then divide by num_true to ensure the per-example
# Labels sum to 1.0, i.e. form a proper probability distribution.
# 如果num_true=n, 那么每行样本的Label就是[1/n, 1/n, ..., 1/n, 0, 0, ..., 0]的形式
# 对于下游的sigmoid loss或softmax loss, 属于soft Label
out_labels = array_ops.concat([
    array_ops.ones_like(true_logits) / num_true,
    array_ops.zeros_like(sampled_logits)], 1)

return out_logits, out_labels

```

5.3.2 Airbnb召回算法

Item2Vec直接把Word2Vec套用到用户行为序列组成的"句子"上, 尽管简单方便, 但是忽略了推荐领域与NLP领域的许多差异之处, 学习到的物料向量的质量受到影响。比如: 在NLP中, 受语法规则所限, 你只能说一

个词与它周围的几个词才是相关的，才能成为正样本。一句话的首尾两个词，相距太远，不太可能相关。但是在推荐领域，一个Session中用户点击的第一个物料与最后一个物料，非常有可能还是高度相关的。为此，Airbnb发表了论文[11]，针对“民宿中介”的业务特点，对Word2Vec进行了若干改进。

Airbnb的I2I召回

根据“向量召回统一框架”，分析一下Airbnb的I2I召回算法

- **如何定义正样本？** 如同前边分析的那样，给定一个用户点击的房屋（i.e., Airbnb叫listing）序列，序列内部的房屋，两两之间都应该是相似的。但由于两两组合太多，所以仿照Word2Vec采用滑窗，认为某个房屋 l 只与窗口内的有限几个房屋 c 才是相似的。但是，如果一个点击序列最终导致某个房屋 l_b 成功预订， l_b 业务信号非常强，必须保留。因此，Airbnb还额外增加了一批正样本，即点击序列中的每个房屋 l 与最终被成功预订的房屋 l_b 是相似的。
- **如何定义负样本？** 根据召回的一贯原则，随机采样得到的其他房屋，肯定是主力负样本。但是“民宿中介”的业务特色决定了，在一个点击序列中的房屋（i.e., 正样本）基本上都是同城的，而随机采样得到的负样本多是异地的。如果只有随机负采样，模型可能只使用“所处城市是否相同”这一粗粒度差异来判断两房屋相似与否，导致最终学到的房屋向量按所在城市聚类，而忽视了同一城市内部不同房屋的差异。为了弥补随机负采样的不足，Airbnb还为每个房屋在与其同城的其他房屋中采样一部分作为Hard Negative，迫使模型关注“所在城市”之外的更多细节。
- **如何Embedding？** 类似Item2Vec，定义一个大矩阵 V ，第 l 个房屋的Embedding v_l 就是 V 的第 l 行。
- **如何定义损失函数？** 与Word2Vec一样采用NEG Loss，但增加了额外的正负样本，如公式(5-17)所示。

$$L_{AirbnbI2I} = \frac{1}{|B|} \sum_{(l,c) \in B} \left[\log(1 + \exp(-v_l \cdot v_c)) + \sum_{nc \in N_l} \log(1 + \exp(v_l \cdot v_{nc})) + \log(1 + \exp(-v_l \cdot v_{lb})) + \sum_{nc \in l} \right]$$

- v 符号表示一个房屋的Embedding。
- 下标 l 表示当前房屋。
- 下标 c 表示在某点击序列中出现在 l 上下文中的房屋。
- 下标 nc 表示随机负采样得到的房屋。
- 下标 lb 代表最终成功预订的房屋。
- 下标 $ncity$ 代表与 l 同处一个城市采样到的负样本房屋。

Airbnb的U2I召回

Airbnb的第二个创新是将Word2Vec扩展到了U2I和冷启动领域。这一次，Airbnb希望从对业务更重要的预订数据中，学习出用户和房屋的embedding。但是问题在于，大多数用户预订和大多数房屋被预订的记录都非常有限，稀疏的预订数据不足以支持模型学习出高质量的用户和房屋embedding。

为此，Airbnb的作法是，根据属性和人工规则，将用户和房屋分类，比如 "20~30岁，使用英语，平均评价4.5星，平均消费每晚50美元的男性"算一类用户，"一床一卫，接纳两人，平均评价5星，平均收费60美元的美国房屋" 算一类房屋。单个用户与单个房屋的预订记录是稀疏的，但是某类用户与某类房屋的预订记录就丰富许多，允许我们使用Word2Vec算法学习出高质量的某一类用户或房屋的Embedding，有助于新用户与新房屋的冷启。

从"向量召回统一框架"，理解从用户类型到房屋类型的召回：

- **如何定义正样本？** 如果某用户 u 预订过某房屋 l ， u 所属的类别U和 l 所属的类别L，在向量空间应该是相似的，成为正样本。
- **如何负样本？** 对于一个用户类型U，随机采样一部分房屋类型，作为主力负样本。除此之外，如果 u 被房东拒绝了，该房屋的类型L就成为了U的Hard Negative，应该加入负样本。
- **如何Embedding？** 定义两个待学习的矩阵， $\mathbf{V}_{UT} \in R^{|UT| \times d}$ 和 $\mathbf{V}_{LT} \in R^{|LT| \times d}$ ， $|UT|$ 是所有用户类型的个数， $|LT|$ 是所有房屋类型的个数。第 i 类用户的Embedding就是 \mathbf{V}_{UT} 的第 i 行，第 i 类房屋的Embedding就是 \mathbf{V}_{LT} 的第 i 行。
- **如何定义损失函数？** 和Word2Vec一样采用NEG Loss，只不过增加了"被房东拒绝"作为Hard Negative，如公式(5-18)所示。

$$L_{AirbnbU2I} = \frac{1}{|B|} \sum_{(ut, lt) \in B} \left[\log(1 + \exp(-\mathbf{v}_{ut} \cdot \mathbf{v}_{lt})) + \sum_{nlt \in N_{ut}} \log(1 + \exp(\mathbf{v}_{ut} \cdot \mathbf{v}_{nlt})) + \sum_{rlt \in N_{rlt}} \log(1 + \exp(\mathbf{v}_{ut} \cdot \mathbf{v}_{rlt})) \right]$$

- v 符号表示Embedding。
- 下标 ut 代表当前用户类型。
- 下标 lt 代表与被 ut 这一类用户预订过的房屋类型。
- 下标 nlt 是给 ut 随机采集的一批负样本的房屋类型。
- 下标 rlt 是拒绝过 ut 的房屋类型。

5.3.3 阿里的EGES召回

阿里于2018年提出Enhanced Graph Embedding with Side Information (EGES) 模型[13]，是将Word2Vec移植到推荐领域的又一次针对性改进。根据"向量召回统一框架"，EGES的改进如下。

如何定义正样本？ 之前介绍的Item2Vec还有Airbnb I2I召回，都认为只有被同一个用户在同一个Session内交互过的两个物料之间，才有相似性，才可能成为正样本。EGES认为这个限制太狭隘了。比如一个用户点击过物料A和物料B，另一个用户点击过物料B和物料C。Item2Vec认为只有AB和BC之间才存在相似性，但是AC难道就不相似了吗？EGES应该把这种跨用户、跨Session的相似性考虑进去，能够提高模型的扩展性，也能够给那些冷门物料更多的训练机会。

具体作法如所示，分为三个步骤。

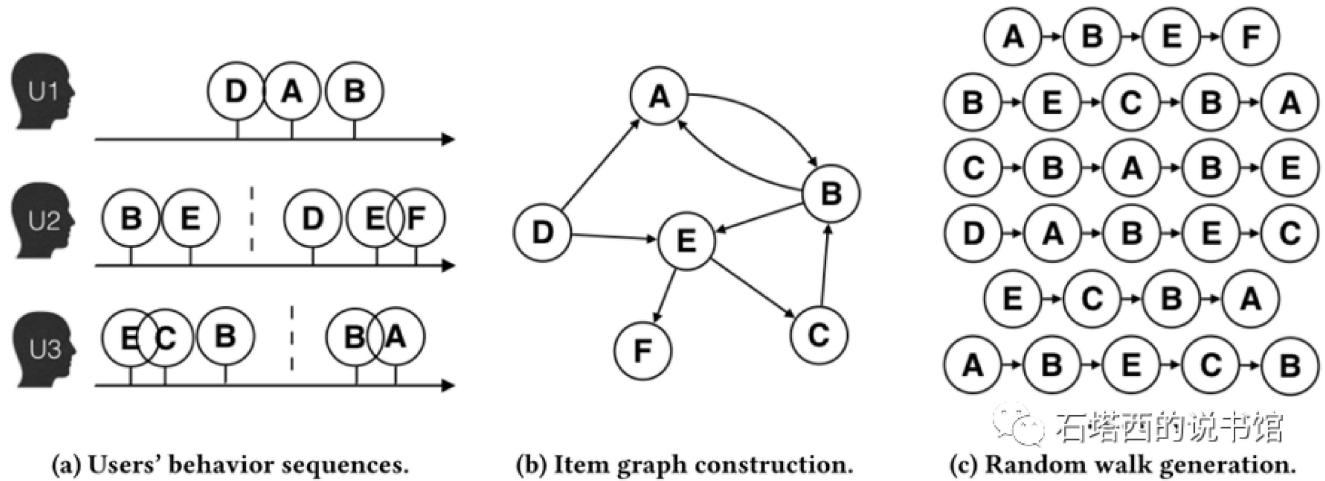


图 5-3 EGES正样本生成流程

- 首先，先根据用户行为序列（如图 5-3(a)所示），建立物料关系图（如图5-3(b)所示）。图上的每一节点代表一个物料，图片的每一条边代表两个物料当顺序交互过。比如，图5-3(a)中的用户U1先点击过物料D再点击过物料A，那么在图 5-3(b)的图上，就有一条边由D指向A。图上第 i 、 j 两个节点之间的边上的权重 M_{ij} ，等于数据集中"先点击物料*i*后点击物料*j*"的次数。
- 然后，沿图 5-3(b)中图上的边随机游走，产生一批新的序列（如图5-3(c)所示）。随机游走过程中，由节点*i*到节点*j*的转移概率 $P(v_j|v_i) = \frac{M_{ij}}{\sum_{j \in N_+(v_i)} M_{ij}}$ ， M_{ij} 是由*i*指向那条边上的权重， $N_+(v_i)$ 是由第*i*个节点出发的邻居节点的集合。
- 最后，在这些随机游走产生的新序列（如图5-3(c)所示）上，再套用Word2Vec的方法，定义滑窗，窗口内的两个物料是相似的，成为正样本。

如何定义负样本？ 没啥创新，照搬Word2Vec中的随机负采样方法。

如何Embedding？ 推荐系统与NLP的一个区别就在于，通常NLP中除了单词自身，就没有多少其他特征可资利用了；但是在推荐系统中，除了ID，每个物料还有丰富的属性信息（aka, Side Information），比如商品的类别、品牌、商铺等。这些信息在照搬Word2Vec的Item2Vec中没有被利用上，太可惜了。另外，加入这些额外的物料属性当特征，还有利于新物料的冷启。对于ID从未在训练集中出现过的新物料，Item2Vec无法给出其Embedding；但是新物料的属性大多在训练集中出现过，并且它们的Embedding已经被训练好了，EGES可以拿这些属性的Embedding合成新物料的Embedding，解一时之需。

EGES的解决方法是，首先，定义了 $1+n$ 个Embedding矩阵， $V_{s_i} \in R^{|s_i| \times d}$ ， $0 \leq i \leq n$ 。其中， $i=0$ 时对应物料的ID， $|s_i|$ 表示第*i*类属性（e.g., 类别、品牌、商铺等）的取值个数。

至于如何将每个物料的ID Embedding和它的*n*个属性Embedding合并成一个Embedding H_v ，最简单的方法是做Average Pooling，如公式(5-19)所示，其中 W_v^s 表示物料*v*的属性*s*的Embedding。

$$H_v = \frac{1}{n+1} \sum_{s=0}^n W_v^s \quad (5-19)$$

复杂一点的方法认为，不应该将所有属性一视同仁，而是应该允许每个物料使用不同的权重将 $1+n$ 个属性的Embedding合成物料Embedding，如公式(5-20)所示。

$$\mathbf{H}_v = \frac{\sum_{j=0}^n \exp(a_v^j) \mathbf{W}_v^j}{\sum_{j=0}^n \exp(a_v^j)} \quad (5-20)$$

- EGES又新定义了一个待学习的权重 $\mathbf{A} \in R^{|V| \times (1+n)}$ ， $|V|$ 是所有物料数， n 是物料属性的个数，列数+1是为了包含进Item ID这个属性。
- a_v^j 是矩阵 A 的第 v 行第 j 列，代表第 j 个属性对第 v 个物料的重要性，它的数值是通过训练得到的。 $\exp(a_v^j)$ 是为了保证权重永远是正数。
- \mathbf{W}_v^j 表示物料 v 的属性 j 的Embedding。

注意EGES与Airbnb U2I，都将除ID之外的属性信息（Side Information）引入到了Word2Vec算法之中，从而缓解了Word2Vec用于推荐时无法处理新用户、新物料的不足。二者引入属性信息的手法不同。

- Airbnb U2I引入了人工先验规则进行分类，以学习某类用户、某类物料的embedding来代替学习单个用户或物料的embedding。这种方法在引入属性信息的同时，也使训练数据变得更稠密，降低训练难度。
- EGES将各类属性信息也Embedding，这些Embedding是什么？它们如何与ID Embedding相融合？都交由算法自动学习出来。这种方法避免了人工规则的繁琐和可能引入的偏见，但是也增加了模型的参数量，需要更多的数据才能学习出来。

如何定义损失函数？ 没啥创新，套用了Word2Vec中的NEG Loss。

5.4 瑞士军刀FM的召回功能

我们在上一章已经介绍过FM及其在精排中的应用了。FM是推荐算法中名副其实的瑞士军刀，结构简单，但功能齐全，不仅能做精排，也能做召回，还能用于下一章要讲的粗排。虽然相比于业界正当红的双塔、基于GNN的召回等复杂模型，FM简单的结构限制了特征交叉，削弱了模型的表达能力，但是也使FM具备了易于上线部署、解释性强等优点，特别适合初创团队。让我们拿“向量召回统一框架”的分析一下FM召回。

5.4.1 打压热门物料

FM召回主要应用于U2I召回场景。很自然，一个用户与他正向交互过的物料构成正例。根据“**不能（只）拿曝光未点击做负例**”的基本原则，我们再为该用户随机采样一部分物料，当成他的负例。说起来简单，但是

这里存在一个打压热门物料的问题，值得大家重视。

任何一个推荐系统，都难逃“2~8”定律的影响，即20%的热门item占据了80%的曝光与点击。因为正例中的物料都是用户点击过的物料，可以说正例数据被少数热门物料所“垄断”。导致的后果就是，训练时，模型会迫使每个用户向量尽可能接近少数热门物料的向量；预测时，每个用户从FAISS检索出来的邻居都是那少数几个热门物料，使推荐结果丧失了个性化与多样性。由此可见，“打压热门物料”的必要性。视物料出现在正例还是在负例中，要采取截然不同的打压策略。

热门物料当正例要降采样

物料热度越高，越不容易成为正例。比如我们可以参考Word2Vec过滤高频停用词的作法[18]，定义一个物料 t_i 能够被任何用户选为正样本的概率 $P_{pos}(t_i)$ ，如公式(5-21)所示。

$$P_{pos}(t_i) = \sqrt{\frac{\alpha}{f(t_i)}} \quad (5-21)$$

- 其中 $f(t_i)$ 是物料 t_i 的曝光频率。可见， t_i 曝光越频繁（e.g., 热度越高），它成为正样本的概率越低。
- α 是一个超参。可以理解成定义“冷门物料”的门槛，如果 $f(t_i) \leq \alpha$ 还被点击过，必须成为正样本。

热门物料当负例要过采样

物料热度越高，越有可能成为负例。可以从两个角度来理解

- 既然热门物料已经“垄断”正样本，我们也需要提高热门物料在负样本中的比例，以抵消热门物料对损失函数的影响
- 如果在负采样时采取Uniform Sampling，因为候选集巨大而采样量有限，因此极有可能采样得到的物料与用户“八杆子打不着”，是所谓的Easy Negative。而因为绝大多数用户都喜欢热门，热门物料当负例构成所谓的Hard Negative，能极大地提升模型的分辨能力。

在随机负采样时，一方面需要采到当负例的物料，能够尽可能广泛地覆盖所有候选物料，另一方面又需要使它们多采样一些热门物料。为了平衡这两方面的需求，参考Word2Vec[18]，定义负采样概率如公式(5-22)所示。

$$P_{neg}(t_i) = \frac{F(t_i)^b}{\sum_{t' \in V} F(t')^b} \quad (5-22)$$

- V 代表所有候选物料。
- $F(t_i)$ 是第 i 个物料的曝光次数。
- 超参 $b=1$ 时，负采样完全遵循物料热度，对热门物料的打压最厉害，但对所有候选物料的覆盖不足。超参 $b=0$ 时，负采样变成Uniform Sampling，对所有候选物料的覆盖度最高，但是对热门物料无打压，采

集的都是Easy Negative。根据Word2Vec的经验， b 一般取0.75。

5.4.2 增广Embedding

首先，不同于Item2Vec、Airbnb召回那样只能使用User ID、Item ID特征，FM召回能使用的特征大为丰富。只有一个例外，为了能够独立、解耦生成用户向量与物料向量，FM召回不能使用交叉特征。

我们重温一下FM公式，对于给定的用户u与物料t，FM描述(u,t)之间的匹配度，如公式(5-23)所示

$$FM(u, t) = b + \sum_{i \in I(u, t)} w_i + \sum_{i \in I(u, t)} \sum_{(j=i+1) \in I(u, t)} \mathbf{v}_i \cdot \mathbf{v}_j \quad (5-23)$$

- 一条样本由一个用户u和一个物料t组成， $I(u, t)$ 表示这条样本中所有非零特征的集合。
- b 代表bias项。
- w_i 代表第*i*个特征的一阶权重。
- \mathbf{v}_i 代表第*i*个特征的Embedding。

将 $\sum_{i \in I} w_i$ 与 $\sum_{i \in I} \sum_{(j=i+1) \in I} \mathbf{v}_i \cdot \mathbf{v}_j$ 按照特征是用户特征还是物料特征进行拆解，得到公式(5-24)。

$$FM(u, t) = b + [W_u + W_t] + [V_{uu} + V_{tt} + V_{ut}] \quad (5-24)$$

- $W_u = \sum_{i \in I_u} w_i$ 表示所有用户特征的一阶权重之和， I_u 表示所有非零用户特征的集合。
- $W_t = \sum_{i \in I_t} w_i$ 表示所有物料特征的一阶权重之和， I_t 表示所有非零物料特征的集合。
- $V_{uu} = \sum_{i \in I_u} \sum_{(j=i+1) \in I_u} \mathbf{v}_i \cdot \mathbf{v}_j$ 表示用户特征集内部的两两交叉。
- $V_{tt} = \sum_{i \in I_t} \sum_{(j=i+1) \in I_t} \mathbf{v}_i \cdot \mathbf{v}_j$ 表示物料特征集内部的两两交叉。
- $V_{ut} = \sum_{i \in I_u} \sum_{j \in I_t} \mathbf{v}_i \cdot \mathbf{v}_j$ 表示每个用户特征与每个物料特征之间的两两交叉。

在给不同的候选物料打分的过程中，由于用户是固定的，因此公式(5-24)中 **b** 、 W_u 、 V_{uu} 对不同候选物料都是相同的，忽略掉也不影响排序结果，从而可以将公式(5-24)简化成公式(5-25)。

$$FM(u, t) = W_t + V_{tt} + V_{ut} \quad (5-25)$$

将 V_{tt} 变形一下，使其降为线性复杂度，写成公式(5-26)，其中的 **reducesum** 操作表示将一个向量所有位置上的数字加总求和。

$$V_{tt} = \sum_{i \in I_t} \sum_{(j=i+1) \in I_t} \mathbf{v}_i \cdot \mathbf{v}_j = \frac{1}{2} reducesum \left[\left(\sum_{i \in I_t} \mathbf{v}_i \right)^2 - \sum_{i \in I_t} \mathbf{v}_i^2 \right] \quad (5-26)$$

再将 V_{ut} 也拆解成两个向量的点积的形式，如公式(5-27)所示。

$$V_{ut} = \sum_{i \in I_u} \sum_{j \in I_t} \mathbf{v}_i \cdot \mathbf{v}_j = \left(\sum_{i \in I_u} \mathbf{v}_i \right) \cdot \left(\sum_{j \in I_t} \mathbf{v}_j \right) \quad (5-27)$$

将公式(5-26)与公式(5-27)代入公式(5-25)，将后者也写成两个向量点积的形式，如公式(5-28)所示，其中 **concat** 代表向量拼接。

$$FM(u, t) = \mathbf{E}_u \cdot \mathbf{E}_t$$

$$\mathbf{E}_u = concat(1, \sum_{i \in I_u} \mathbf{v}_i)$$

$$\mathbf{E}_t = concat(W_t + V_{tt}, \sum_{j \in I_t} \mathbf{v}_j)$$

- \mathbf{E}_u 代表用户向量，召回时，由线上实时计算生成。
- \mathbf{E}_t 代表物料向量，离线提前计算好，并灌入FAISS建立索引。注意前面增广的 $W_t + V_{tt}$ ，代表不随用户改变的、物料自身的受欢迎程度。面对新用户时， V_{ut} 提供的信息有限， $W_t + V_{tt}$ 会发挥主要作用。 W_t 见公式(5-24)， V_{tt} 见公式(5-26)，这两项都是线性复杂度。

提醒大家注意的是，公式(5-28)只是针对预测的时候。训练的时候，没必要将用户特征与物料特征拆开，假设采用BPR Loss，FM召回的损失函数如公式(5-29)所示。

$$FM(u, t) = b + \sum_{i \in I(u, t)} w_i + \frac{1}{2} reduce\sum \left[\left(\sum_{i \in I(u, t)} \mathbf{v}_i \right)^2 - \sum_{i \in I(u, t)} \mathbf{v}_i^2 \right] \quad (5-29)$$

$$L_{BPR} = \frac{1}{|B|} \sum_{(u_i, t_{i+}, t_{i-}) \in B} \log(1 + \exp(FM(u_i, t_{i-}) - FM(u_i, t_{i+})))$$

- B 代表一个Batch，其中一条样本由 (u_i, t_{i+}, t_{i-}) 三元组构成， u_i 表示用户， t_{i+} 是他点击过的物料， t_{i-} 是给他随机采样的一个物料。
- **reduce\sum** 操作表示将一个向量所有位置上的数字加总求和。
- 其他符号含义参考公式(5-23)。

5.5 大厂主力：双塔模型

双塔模型是各互联网大厂目前最主要的召回算法，本节让我们借助“向量化召回统一框架”理解该算法。

5.5.1 不同场景下的正样本

双塔模型的应用场景非常广泛，在不同的场景中，可以用不同规则生成正样本。

- U2I召回：一个用户 u 与他交互过的物料 t ，相互匹配，可以构成一对正样本。

- I2I召回：一个用户在一个Session交互过的两个物料 t_i 和 t_j ，应该具备相似性，可以成为一对正样本。
- U2U召回：一个用户的一半行为历史，与同一个用户另一半的行为历史，都源于同一个用户的兴趣爱好，应该彼此相似，可以组成一对正样本。

5.5.2 简化负采样

Batch内负采样

互联网大厂在实践双塔召回时，一个常见作法是采取Batch内负采样[20]。以点击场景下的U2I召回为例，

- 在一个Batch B中，第*i*条样本由用户 u_i 与他的正例物料 t_i 组成，是正样本。
- Batch内负采样是指，除了 t_i 之外，B中其他样本中的正例物料，都作为 u_i 的负例。即 $(u_i, t_j), \forall j \in B - i$ ，都是负样本。

Batch内负采样如图5-4所示。注意Batch内的每个用户向量，要与Batch内的每个物料向量做点积。这其实也是它的优点，由于 u_i 的某个负例 t_j 的向量，已经作为 u_j 的正例，被计算过了，可以被复用而避免重复计算。计算量的降低，对于面临“海量数据”和“模型实时更新”压力的各互联网大厂，是非常具有诱惑力的。

Batch内负采样的缺点是容易造成“样本选择偏差”（Sample Selection Bias, SSB）。这是因为，召回的正样本来自点击数据，而被点击的多是热门物料。再加上一个Batch的大小有限，其中的热门物料就更加集中，与召回要被应用于整个物料库的数据环境，差距较大。换句话说，Batch内负采样，采集到的负样本都是 Hard Negative（大多数用户都喜欢热门物料），缺少与用户兴趣“八杆子打不着”的Easy Negative。

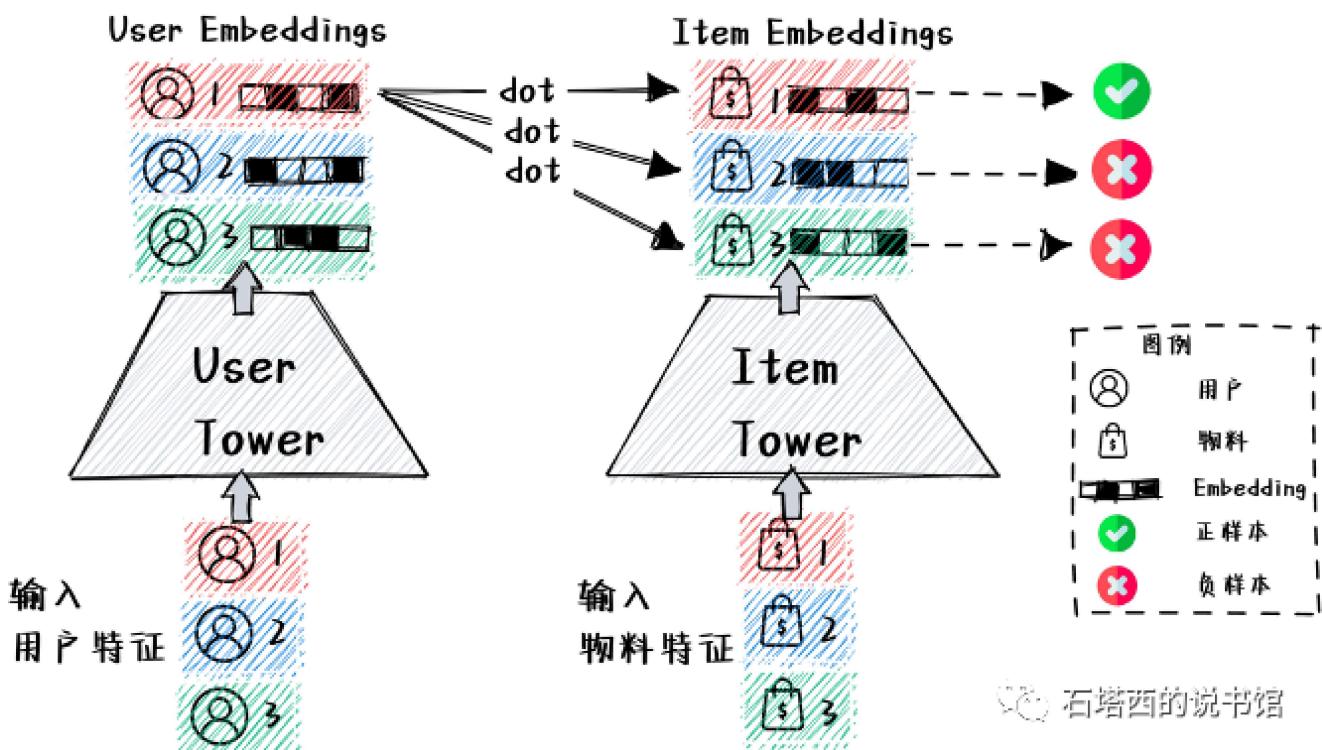


图 5-4 Batch内负采样示意

混合负采样

为了缓解“样本选择偏差”，Facebook[16]、Google[21]和华为[22]都在Batch内负采样的基础上发展了“混合负采样”（Mixed Negative Sampling）策略。混合负采样的作法，如图 5-5所示。

- 额外建立了一个“向量缓存”，存储“物料塔”在训练过程中得到的最新的物料向量。
- 在训练每个Batch的时候，先进行Batch内负采样，同一个Batch内两条样本中的物料互为Hard Negative。
- 额外从“向量缓存”采样一些由“物料塔”计算好的之前的物料向量 B' ，作为Easy Negative的Embedding。

尽管在一个Batch内部，热门物料比较集中，但是“向量缓存”汇集了多个Batch计算出的物料向量，从中还是能够采样到一些小众、冷门物料当Easy Negative的。所以，混合负采样对物料库的覆盖更加全面，更加符合负样本要让召回模型“开眼界、见世面”的一般原则。

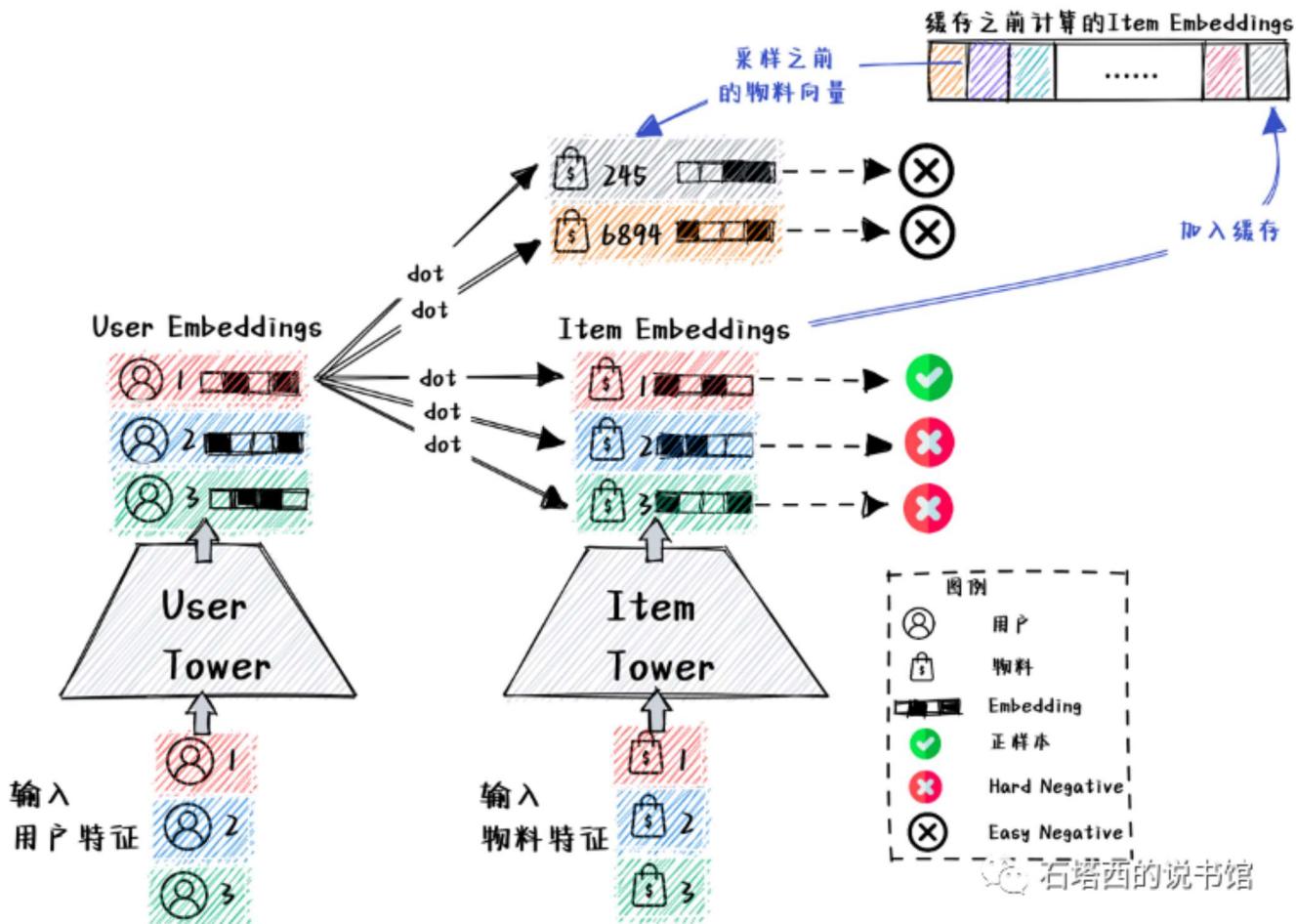


图 5-5 混合负采样示意

5.3.3 双塔结构特点

单塔可以很复杂

第一个特点在于"塔"字，这里的"塔"就是一个DNN，如图 5-6所示。

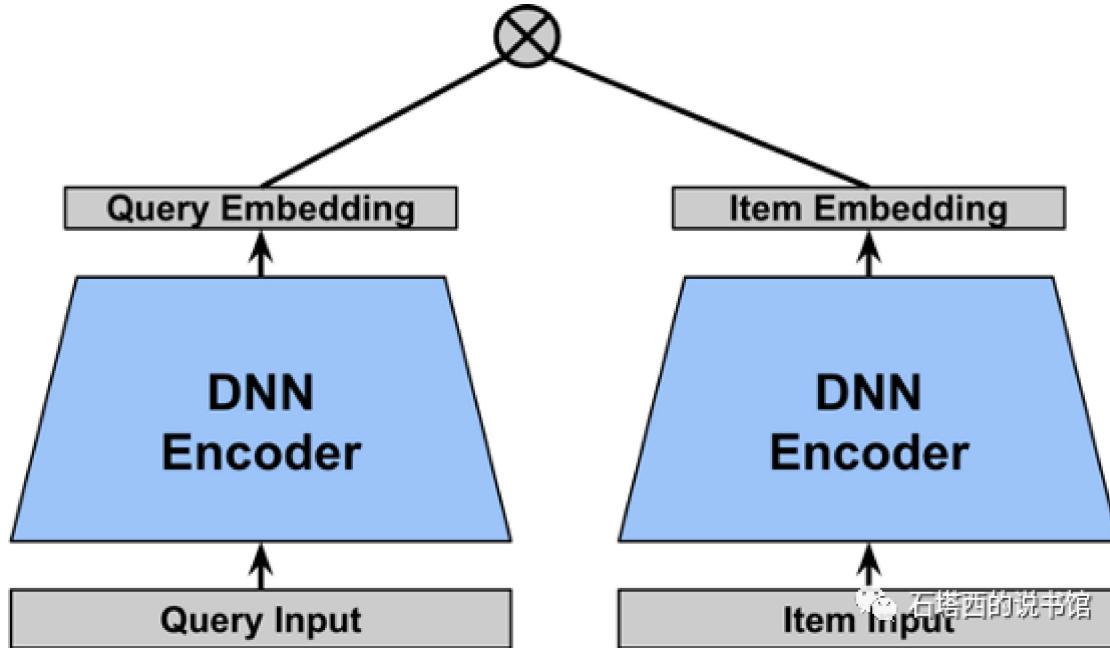


图 5-6 双塔模型结构示意

- 以U2I为例，用户特征喂入"用户塔"，输出用户向量；物料（无论正例还是负例）特征喂入"物料塔"，输出物料向量。
- 塔的底座宽。不像Item2Vec那样只能接受Item ID当特征，双塔能够接受的特征大为丰富。用户侧可以包括：User ID、用户的人口属性（e.g., 性别、年龄）、用户画像（e.g., 用户对某个标签的CTR）、用户各种行为历史（e.g., 点击序列）；物料侧可以包括：Item ID、物料的基本属性（e.g., 店铺、品牌）、物料静态画像（e.g., 所属类别）、物料动态画像（e.g., 过去1天的CTR），还可以包括由内容理解算法产生的各种文字、图像、音频、视频的向量。
- 塔身很高，每层的结构可以很复杂。这样一来，底层喂入的丰富信息，在沿塔向上流动的过程中，可以完成充分而复杂的交叉，模型的表达能力相较于之前的Item2Vec、FM召回大为增强。

双塔一定要解耦

双塔的另一个特点在于"双"字。遵循在5.2.3节介绍的召回模型的一般原则，双塔模型要求两塔之间，绝对独立、解耦。只允许在生成最终的用户向量和物料向量之后，才点积交叉一次。在此之前，用户信息只能沿着"用户塔"向上流动，物料信息只能沿着"物料塔"向上流动。绝对不允许出现跨塔的信息交流，避免双塔勾连成单塔。

为了解耦，特征上，绝对不能使用任何交叉特征（比如，用户标签与当前物料的标签的重合度）。结构上，像Deep Interest Network那样，由候选物料对用户行为序列做Attention，也是做不到的。

但是用户的各种行为序列是反映用户兴趣的最重要的信息来源，如何将它们接入“用户塔”，也是业界研究的重要课题。最简单的如Youtube的作法[20]，把用户过去观看的视频列表，先Embedding，再做Average Pooling聚合成一个向量，接入“用户塔”。

但是Average Pooling将用户的各个历史行为一视同仁，损失太多信息。我们还是希望能遵照Attention的方法，给每个历史行为赋以不同权重，在聚合过程中体现过轻重缓急。既然拿当前物料当Query做Attention违反了“双塔解耦”原则而行不通，业界提出了一些替代方案：

- 如果是在搜索场景下，用户输入的搜索文本是最能反映用户当下意图的，可以作为Attention中的Query，衡量用户各历史行为的重要性。
- 阿里的SDM[23]召回中，使用用户画像当Query，给各历史行为打分。
- 微信的CDR[24]模型，认为用户最后点击的物料，反映用户最新的兴趣偏好，可以当Query去衡量之前历史行为的重要性。

5.5.4 Sampled Softmax的技巧

双塔模型比较常用的是基于Batch内负采样的Sampled Softmax Loss。以U2I召回为例，双塔模型的Loss如公式(5-30)所示。

$$G(u, t) = \frac{\mathbf{u} \cdot \mathbf{t}}{\tau} - \log Q(t)$$

$$L_{Tower} = -\frac{1}{|B|} \sum_{(u_i, t_i) \in B} \log \frac{\exp(G(u_i, t_i))}{\exp(G(u_i, t_i)) + \sum_{j \in B, j \neq i} \exp(G(u_i, t_j))} \quad (5-30)$$

- B代表一个Batch，其中第*i*条样本(u_i, t_i)由用户 u_i 和他点击过的物料 t_i 组成。
- 第*i*条正样本对应的负样本，由(u_i, t_j), $j \in B, j \neq i$ 组成，也就是说Batch内其他样本中的物料 t_j 与 u_i 搭配成为负样本。
- $G(u, t)$ 计算用户 u 与物料 t 之间的匹配程度，其中的 u 、 t 、 τ 、 Q 的含义在本小节接下来的内容中加以详细说明。

L2正则化

u 代表由“用户塔”将用户 u 所有信息压缩成的向量， t 表示由“物料塔”将候选物料 t 所有信息压缩成的向量。

一个常见的技巧是在拿用户向量与物料向量计算点积之前，先除以它们各自的L2 Norm，转化成长度为1的向量。这样做点积，等价于用cosine(u, t)来计算用户向量 u 与物料向量 t 之间的匹配度。很多实践都证明

[20,25]，使用 `cosine` 要比使用点积的效果更好。其实也很好理解，毕竟衡量两个向量是否相似，是看它们之间的夹角，没有必要让优化器浪费精力在调整向量的长度上。

温度调整难度

公式中的 $0 < \tau < 1$ ，被称为"温度"。我们知道Softmax Loss的优化目标是，使正例的概率

$P_{pos} = \frac{\exp(G(\mathbf{u}_i, \mathbf{t}_i))}{\exp(G(\mathbf{u}_i, \mathbf{t}_i)) + \sum_{j \in B, j \neq i} \exp(G(\mathbf{u}_i, \mathbf{t}_j))}$ 尽可能大，这就要求分母中每个负例的得分 $G(u_i, t_j)$ 要尽可能小才行。所以 τ 起到一个放大器的作用，但凡有哪个负例 (u_i, t_j) 没被训练好，导致 $\text{cosin}(\mathbf{u}_i, \mathbf{t}_j) > 0$ ， $\frac{1}{\tau}$ 就能把这个"错误"放大许多倍，导致分母变大，损失增加。那个没被训练好的负例，就会被模型聚焦，被重点"关照"。

温度 τ 可以用来召回结果的"记忆性"与"扩展性"，或者说，精度与多样性。

- 如果 τ 设置得很小，它对错误的放大功能就很强。只要物料 t_j 未被用户 u_i 点击过，模型就会强力将它们拉开。但是我们知道， t_j 是负采样得到的，它未被 u_i 点击，可能只是还没有曝光给 u_i 而已。所以 τ 很小的话，将使模型牢牢记住用户历史点击反映出的兴趣爱好，召回时不敢偏离太远。推荐结果的精度高，但对用户潜在兴趣覆盖得不够，容易使用户陷入"信息茧房"而疲劳。
- 如果 τ 设置大了，它对错误的放大功能就很弱。对负样本 (u_i, t_j) ，也不会将 u_i 和 t_j 拉开得非常远，召回时仍有可能将 t_j 当成 u_i 的近邻推荐出去。好处是， t_j 真的覆盖了用户的一部分隐藏兴趣，推荐 t_j 是对用户兴趣的一次探索与扩展，丰富推荐结果的多样性，打破"信息茧房"。坏处是，有可能 t_j 真的不是 u_i 的"菜"，召回把关太松，有损用户体验。

采样概率修正

公式中的 $Q(t)$ 表示负采样到某个物料 t 的概率。如果我们使用Batch内负采样而且样本来自点击数据， $Q(t)$ 就等于物料 t 在所有点击样本中的占比，即 $Q(t_i) = \frac{C(t_i)}{\sum_{t_j} C(t_j)}$ ，其中 $C(t)$ 表示物料 t 的点击次数。通常对精度要求没那么高， $Q(t)$ 可以离线、定时统计得到。对精度要求高的，可以通过流式算法在线预估 $Q(t)$ 。

如果我们使用"混合负采样"，损失函数如公式(5-31)所示。

$$L_{Tower} = -\frac{1}{|B|} \sum_{(u_i, t_i) \in B} \log \frac{\exp(G(u_i, t_i))}{\exp(G(u_i, t_i)) + \sum_{j \in B + B', j \neq i} \exp(G(u_i, t_j))} \quad (5-31)$$

- 与公式(5-30)唯一的不同之处就是在生成负样本时，引入额外的负样本 B' ， B' 代表之前生成的物料向量的缓存，详情参考5.5.2节的"混合负采样"部分。
- $G(u, t)$ 和其他符号含义参考公式(5-30)。

因为现在负采样来自两种采样策略，因此 $Q(t)$ 也是一个组合概率，如公式(5-32)所示。

$$Q(t) = \frac{|B|}{|B| + |B'|} Q_{in\text{-}batch}(t) + \frac{|B'|}{|B| + |B'|} Q_{cache}(t) \quad (5-32)$$

- $Q_{in\text{-}batch}(t)$ 就是Batch内负采样的概率，前边已经介绍过了，就等于物料 t 在所有点击样本中的占比。
- $Q_{cache}(t)$ 是在"物料向量缓存" B' 中采到物料 t 的概率。如果在缓存中平均采样， $Q_{cache}(t) = \frac{1}{CacheSize}$ ，CacheSize就是"向量缓存"的大小。

5.5.5 双塔模型实现举例

TensorFlow Recommenders提供了一个双塔的实现[26]。受篇幅所限，本书只列举并注释核心重要代码。完整代码细节，请感兴趣的读者移步参考文献[26]。

电影推荐场景下的"双塔召回"如代码 5-5所示。

代码 5-5 双塔召回代码示例

```
class MovieLensModel(tftrs.models.Model):
    """电影推荐场景下的双塔召回模型"""

    def __init__(self, layer_sizes):
        super().__init__()
        self.query_model = QueryModel(layer_sizes) # 用户塔
        self.candidate_model = CandidateModel(layer_sizes) # 物料塔
        self.task = tftrs.tasks.Retrieval(...) # 负责计算Loss

    def compute_loss(self, features, training=False):
        # 只把用户特征喂入“用户塔”，得到user embedding "query_embeddings"
        query_embeddings = self.query_model({
            "user_id": features["user_id"],
            "timestamp": features["timestamp"],
        })
        # 只把物料特征喂入“物料塔”，生成item embedding "movie_embeddings"
        movie_embeddings = self.candidate_model(features["movie_title"])

        # 根据Batch内负采样方式，计算Sampled Softmax Loss
        return self.task(query_embeddings, movie_embeddings, ...)
```

`tftrs.tasks.Retrieval` 实现了基于Batch内负采样的Sampled Softmax Loss，如代码 5-6所示。

代码 5-6 实现基于Batch内采样的Sampled Softmax Loss

```
class Retrieval(tf.keras.layers.Layer, base.Task):

    def call(self, query_embeddings, candidate_embeddings,
             sample_weight, candidate_sampling_probability, .......) -> tf.Tensor:
        """
        query_embeddings: [batch_size, dim], 可以认为是user embedding
        candidate_embeddings: [batch_size, dim], 可以认为是item embedding
        """

        # query_embeddings: [batch_size, dim]
        # candidate_embeddings: [batch_size, dim]
        # scores: [batch_size, batch_size], batch中的每个user对batch中每个item的匹配度
        scores = tf.linalg.matmul(query_embeddings, candidate_embeddings, transpose_b=True)

        # labels: [batch_size, batch_size], 对角线上全为1, 其余位置都是0
        labels = tf.eye(tf.shape(scores)[0], tf.shape(scores)[1])

        if self._temperature isnotNone: # 通过温度, 调整训练难度
            scores = scores / self._temperature

        if candidate_sampling_probability isnotNone:
            # SamplingProbabilityCorrection的实现就是
            # Logits - tf.math.log(candidate_sampling_probability)
            # 因为负样本是抽样的, 而非全体item, Sampled Softmax进行了概率修正
            scores = layers.loss.SamplingProbablityCorrection()(scores, candidate_sampling_probabilty)

        .....

        # labels: [batch_size, batch_size]
        # scores: [batch_size, batch_size]
        # self._loss就是tf.keras.Losses.CategoricalCrossentropy
        # 对于第i个样本, 只有labels[i,i]等于1, scores[i,i]是正样本得分
        # 其他位置上的labels[i,j]都为0, scores[i,j]都是负样本得分
        # 所以这里实现的是Batch内负采样, 第i行样本的用户, 把除i之外所有样本中的正例物料, 当成负例物料
        loss = self._loss(y_true=labels, y_pred=scores, sample_weight=sample_weight)

    return loss
```

5.6 邻里互助：GCN召回

5.6.1 GCN基础

近年来，图卷积神经网络（Graph Convolutional Network，GCN）在推荐系统召回中得到了越来越广泛的应用。我们可以将推荐系统构建成一张图（如图 5-7 所示），各种实体（e.g., 用户、商品、店铺、品牌等）可以作为图的顶点，各种互动关系（e.g., 点击、购买、共现）等互动关系构成了图的边。

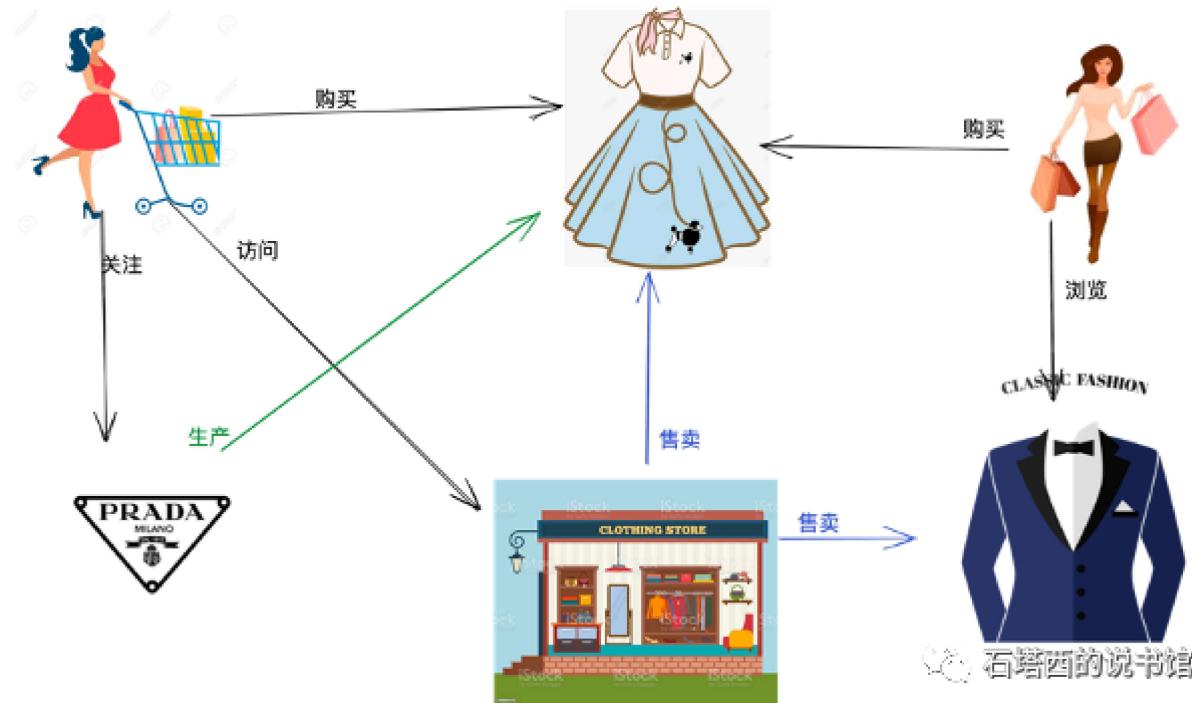


图 5-7 将推荐系统建模成图

在这张图上，GCN 将召回建模成边预测问题（Link Prediction），即预测图上两节点之间是否有边存在。如果 GCN 预测某用户节点 u 与物料节点 t 之间存在边，那么 t 可以作为 u 的 U2I 的召回结果；如果 GCN 预测两物料节点 t_1 和 t_2 之间存在边，那么 t_2 可以作为 t_1 的 I2I 召回结果。

从“向量化召回统一框架”的角度来看 GCN：

- **如何定义正样本？** 比如，图上一条边两端的节点 v_1 和 v_2 ，构成正样本
- **如何定义负样本？** 为 v_1 在图上随机采集一部分顶点，组成负样本
- **如何定义损失函数？** 5.2.4 节中介绍过的 NEG Loss、Sampled Softmax Loss、Marginal Hinge Loss、BPR Loss 都可以使用。

以上三个环节，GCN 都是“老生常谈”，只是在“如何 Embedding”这个环节上进行了创新，将图上的拓扑连接关系也考虑了进去。

- 像用户访问过的店铺、商品品牌这样的信息，之前只是单纯地为刻画用户和物料贡献了自己本身的信息，但是它们背后的“信息传递媒介”功能还未被开发和利用。
- GCN能够让两个用户的信息，通过他们共同购买过的商品、或共同关注的品牌、或共同逛过的商店等媒介，相互传递，从而丰富用户建模时的信息来源。
- GCN能够让两个物料的信息，通过它们共同属于的品牌、或共同拥有的标签、或被相同关键词搜索过等媒介，相互传递，从而丰富物料建模时的信息来源。

GraphSAGE[27]是GCN的一种实现思想。

- 在GraphSAGE出现之前，GCN的实现都是“直推式”（Transductive）的，也就是预测与训练只能使用相同的图，对于未曾在训练图上出现过的节点，传统GCN无法给出它们的向量表示。
- GraphSAGE并不直接学习图上各节点的向量，而是要学习出一个转换函数。只要输入节点的特征和它的连接关系，该转换函数就能返回该节点的向量表示。所以，GraphSAGE是“归纳式”（Inductive）的，能够在训练时未曾出现过的新节点上，也获得向量表示。

大型推荐系统构造出的图通常包含上亿节点，不可能因为新来一个用户或一个物料，就重新建图，把所有节点的向量都重新学一遍。因此，GraphSAGE的Inductive性质，能够将训练结果扩展至新用户和新物料，非常实用，成为GCN应用于推荐系统的主流实现方案。

GraphSAGE的思路，如公式(5-33)所示。

$$\begin{aligned}\mathbf{h}_v^0 &= \mathbf{x}_v \\ \mathbf{h}_v^k &= \sigma \left(\mathbf{W}_k \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \mathbf{B}_k \mathbf{h}_v^{k-1} \right), \forall k > 0 \\ \mathbf{z}_v &= \mathbf{h}_v^{last}\end{aligned}\quad (5-33)$$

- 节点 v 的第0层卷积结果是 \mathbf{h}_v^0 ，就等于节点 v 的特征向量 \mathbf{x}_v
- 节点 v 最后一层的卷积结果 \mathbf{h}_v^{last} ，就是节点 v 最终的向量表示 \mathbf{z}_v
- 公式(5-33)的第二行就代表第 k 层卷积，由两部分组成。 $\mathbf{W}_k \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}$ 代表聚合节点 v 的所有邻居（即 $N(v)$ ）向量，再用权重 \mathbf{W}_k 线性映射。 $\mathbf{B}_k \mathbf{h}_v^{k-1}$ 表示对节点 v 的上一层的输出 \mathbf{h}_v^{k-1} ，用权重 \mathbf{B}_k 线性映射。第 k 层的卷积结果是以上两部分之和再用函数 σ 非线性激活。
- 如果我们忽略邻居节点，第 k 层的卷积公式就变成 $\mathbf{h}_v^k = \sigma(\mathbf{B}_k \mathbf{h}_v^{k-1})$ ，它就是MLP中第 k 层的前代公式。从这里可以看出，GraphSAGE相当于MLP的一种改进，在第 k 层前代中，先把节点 v 的邻居信息与 v 本身旧信息融合成一个新向量，再通过过常规的线性映射和非线性激活。
- 公式(5-33)中用 $\sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}$ 这样的Average Pooling来聚合邻居节点的信息。实践中，我们还可以实现其他的聚合方式，比如拼接、Attention等，在接下来介绍具体算法时，我们再详细说明。

整个图卷积过程如图5-8所示[28]，图中的AGG表示“聚合”，PROJB 对应公式(5-33)中的 \mathbf{B}_k ，PROJW对应公式(5-33)中的 \mathbf{W}_k 。可以看到，虽然第3层建模A节点时，只用到了第2层的B/C/D三个邻居节点的信息，但

是第2层的B节点信息已经融合了第1层中的E/F两节点的信息，所以最终A节点的向量表示，实际上已经融合了这个小图中所有节点的信息。归纳下来，GCN中建模单个节点所使用的信息来源大为扩展，建模结果具备“全局视野”。按照CNN的术语来说，GCN扩大了单个节点的“感受野”（Receptive Field）。

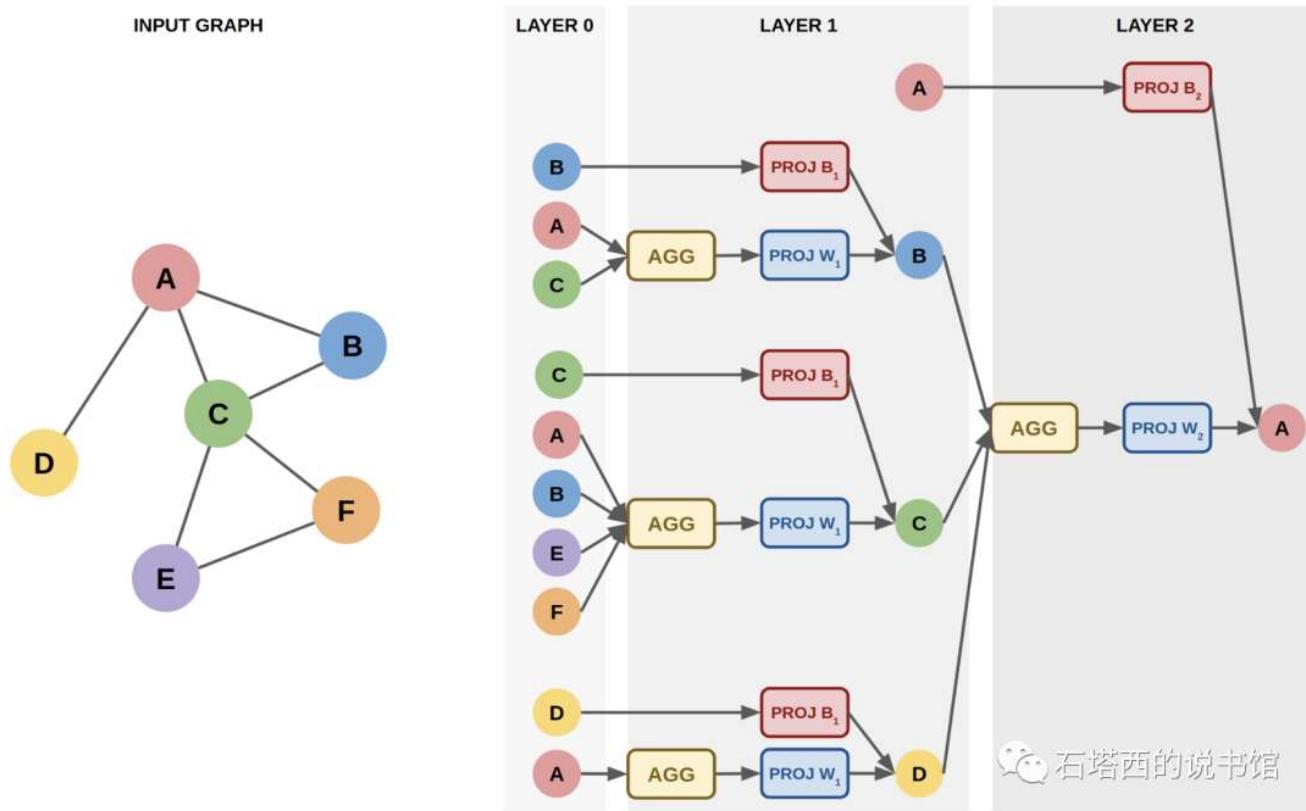


图 5-8 图卷积过程演示

5.6.2 PinSage：大规模图卷积的经典案例

PinSage[14]是Pinterest团队开发的召回算法系统。PinSage在一个含有30亿节点、180亿条边的超大型图上实现了GraphSAGE，被誉为GCN在互联网大型推荐系统中的首次实战。

PinSage建模的是只有Pin（Pinterest的业务概念，可以理解成一个网址）和Board（Pinterest的业务概念，可理解成网址的收藏夹）两类节点的二部图，即只有Pin和收藏它的Board之间才有边，如图5-9所示。特征上，每个Pin节点有ID、文本、图片特征，Board节点被视为一类特殊的、只有ID特征的Pin节点，将二部图按照同构图方式建模。

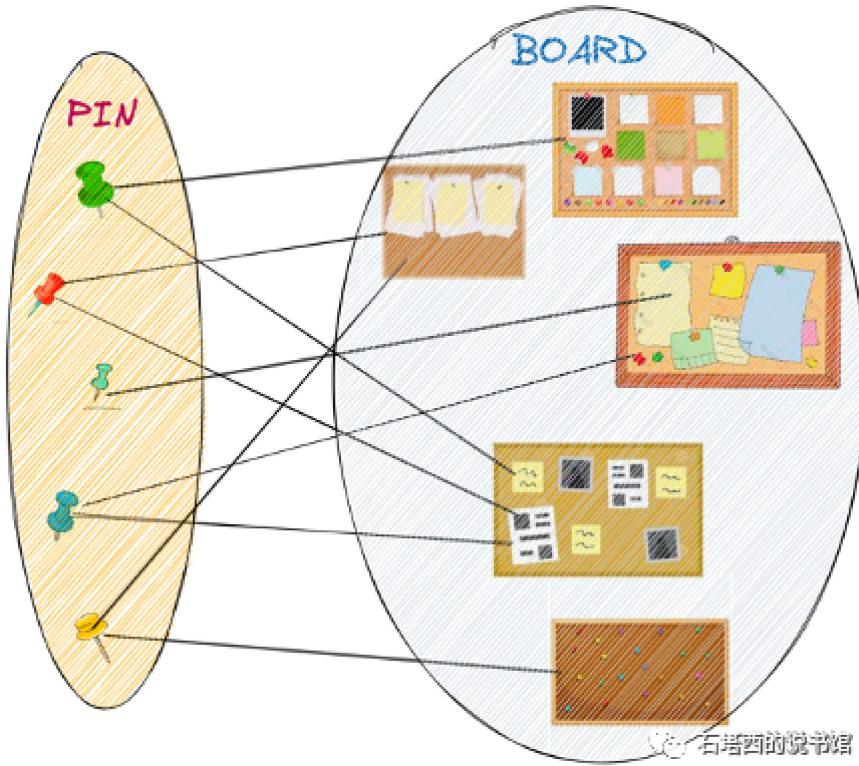


图 5-9 PinSage的二部图示意

PinSage的目标是学习出高质量的Pin Embedding，用于I2I召回，用于给用户推荐与他收藏的Pin相似的其他Pin。用户收藏了Pin~1~，系统会提示给用户一系列相似Pin，如果用户又收藏这其中的Pin~2~，那(Pin1,Pin2)就成为一个正样本，另外给Pin~1~随机采集一些Pin作为负样本。

单单从算法角度来看，PinSage并没有什么创新，就是普普通通的GraphSAGE，伪码实现如代码5-7所示。

代码 5-7 PinSage图卷积伪码实现

```

1. // 先将节点  $u$  的所有邻居节点的向量  $h_v$ ，先通过一层非线性映射（权重是  $Q$ ，bias 是  $q$ ）
2. // 再聚合成一个向量  $n_u$ ， $N(u)$ 是节点  $u$  的邻居节点集合，AGG 是聚合函数
3.  $n_u = \text{AGG}(\{\text{ReLU}(Qh_v + q) \mid v \in N(u)\})$ 
4.
5. //  $z_u$ 是节点  $u$  老的向量表示， $n_u$ 聚合了  $u$  的邻居信息
6. // 将  $n_u$ 与  $z_u$ ，拼接一起，再经过非线性映射（权重是  $W$ ，bias 是  $w$ ），得到  $u$  的新向量  $z_u^{\text{NEW}}$ 
7.  $z_u^{\text{NEW}} = \text{ReLU}(W \times \text{Concat}(n_u, z_u) + w)$ 
8.
9. // 再对  $z_u^{\text{NEW}}$ 做一下正则化，得到节点  $u$  最终的卷积结果
10.  $z_u^{\text{NEW}} = z_u^{\text{NEW}} / \|z_u^{\text{NEW}}\|$ 

```

PinSage的技术亮点，在于其一系列的工程优化技巧，使其能够在一个上亿规模的超大型图上高效地训练与推理。

Mini-Batch训练

PinSage面向的是互联网大型推荐系统，不可能把上十亿的节点和边一古脑装进内存，更甭提装载入显存，也不可能每次前代回代，把上十亿节点的向量都更新一遍。所以，我们必须采用基于Mini-Batch的迭代算法，每轮训练时，GPU只加载计算当前Mini-Batch所需要的子图，也只前代回代该Mini-Batch涉及到的节点。整个基于Mini-Batch的前代流程如伪代码5-8所示。

代码 5-8 基于Mini-Batch的PinSage前代算法流程

Input M: 一个节点的集合
Input N: 一个邻居采样函数 N , $N(u)$ 从节点 u 的所有邻居节点中采样一部分并返回
Output: M 中每个节点的 Embedding $\mathbf{z}_u, \forall u \in M$

```

1. // ***** 自上而下, 把计算  $M$  所需要的所有节点都提前收集好 *****
2.  $S^{(K)}=M$  // 第  $K$  层要计算的节点就是  $M$  指定的那些节点
3. for  $k=K, \dots, 1$  do
4.   // 找出第  $k-1$  层卷积要计算哪些节点的 Embedding
5.    $S^{(k-1)}=S^{(k)}$ 
6.   for  $u \in S^{(k)}$  do
7.     |  $S^{(k-1)}=S^{(k-1)} \cup N(u)$  // 第  $k$  层目标节点的邻居都要在  $k-1$  层计算
8.   end for
9. end for
10. // ***** 自下而上, 逐层卷积 *****
11. for  $u \in S^{(0)}$  do //  $S^{(0)}$ 代表第 0 层要计算哪些节点的卷积
12.   |  $\mathbf{h}_u^{\{0\}}=\mathbf{x}_u$  //  $\mathbf{h}_u^{\{0\}}$ 是节点  $u$  在第 0 层的卷积结果, 就等于该节点的原始输入  $\mathbf{x}_u$ 
13. end for
14. for  $k=1, \dots, K$  do //进行第  $k$  层的卷积
15.   for  $u \in S^{(k)}$  do //  $S^{(k)}$ 代表第  $k$  层要计算的节点
16.     |  $H=\{\mathbf{h}_v^{(k)}, \forall v \in N(u)\}$  // 节点  $u$  的邻居的向量集合
17.     |  $\mathbf{h}_u^{(k)}=\text{CONVOLVE}(\mathbf{h}_u^{(k-1)}, H)$  // 将本节点与邻居节点卷积, CONVOLVE 参考代码 5-7
18.   end for
19. end for
20. // ***** 把卷积结果再做一下映射 (可选) *****
21. for  $u \in M$  do
22.   | //  $\mathbf{h}_u^{(K)}$ 是最后一层的卷积结果,  $\mathbf{W}_1$ 、 $\mathbf{W}_2$ 、 $\mathbf{b}_1$ 是映射的权重
23.   |  $\mathbf{z}_u=\mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1 \mathbf{h}_u^{(K)} + \mathbf{b}_1)$ 
24. end for

```



代码 5-8中的第1 ~ 9行就是根据Mini-Batch逐层生成所需要的子图的过程，如图5-10所示意。

- 代码中的 $S^{(k)}$ 代表第 k 层卷积所在子图的输出节点（即目标节点）。第 k 层卷积子图的输入节点，也就是第 $k-1$ 层卷积的输出节点 $S^{(k-1)}$ 。
- 如果一个Batch所包含的节点是 M ，最后一层卷积（即第 K 层）的输出节点 $S^{(K)}$ 就是 M ，也就是代码5-8中的第2行。
- 从此向下迭代，第 $k-1$ 层卷积所需要的子图，由第 k 层的子图扩展而来。扩展方法是将 $S^{(k)}$ 中每个节点 u 都采样一部分它的邻居，得到 $N(u)$ ，合并到 $S^{(k-1)}$ 中，也就是代码5-8中的第7行。具体采样方法，即 $N(u)$ 是如何实现的，将在下一小节中详细描述。

- 这个过程如图5-10演示的一样，假设 $S^{(k)}$ 只有一个节点{A}， $S^{(k-1)}$ 通过在A的邻居中采样扩展生成，就是图中的{A, B, C, D}。

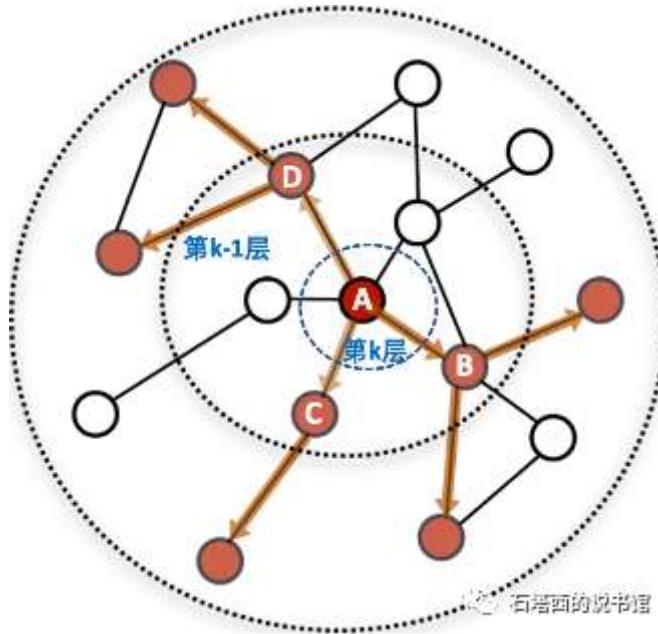


图 5-10 Mini-Batch逐层生成子图

代码 5-8的10 ~ 19行就是自下向上逐层卷积的过程，注意两点：

- 第15行，遍历 $u \in S^{(k)}$ ，说明第 k 层卷积并不会更新全图上亿个节点，而只涉及 $S^{(k)}$ 那一小部分节点。
- 第16行获取 u 的邻居向量，也只包含采样后 $N(u)$ 那有限几个邻居，不受节点 u 可能非常庞大的邻居数目影响。

由此可以看到，基于Mini-Batch的图卷积，每层卷积只计算有限个节点的向量，每个节点只考虑有限个邻居，不受原始图的节点规模和边规模的影响，既能加快计算速度，又能减轻GPU显存的存储压力。

代码5-8的20 ~ 24，只不过是在图卷积完成后，再套上一个简单的MLP进行简单变换，其实可以忽略。

为了进一步提升计算效率，PinSage还实现了"生产者~消费者"（Producer~Consumer）结构：

- 为当前Mini-Batch生成各层卷积的计算子图（代码的1 ~ 7行）运行在CPU上，允许利用CPU的大内存优势，能够加载十亿规模的超大图。
- GPU负责在各层子图上实现卷积的前代与回代。各层子图规模有限，GPU显存装载得下，计算也快。
- CPU为当前Batch抽取好的各层子图后，就发往GPU去卷积，接下来就开始为下一个Batch抽取子图。“抽取子图”与“子图上卷积”并发执行，进一步提高了训练效率。

邻居采样

代码5-8中的第5行，在生成各层子图时，并没有采用 u 的全部邻居，而是从 u 的邻居中抽样得到 $N(u)$ 。这是因为在大型推荐系统构成的图中，一个节点的邻居可能会有海量的邻居，比如图5-9中热门的Pin可能会被收藏进上万个不同的Board中，也就是热门Pin节点可能有上万个邻居。每次卷积热门Pin节点时，都要聚合那么多邻居的信息，计算和存储的压力都非常大，所以“邻居采样”势在必行。

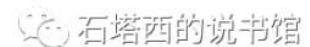
PinSage采用的是基于“随机游走”（Random Walk）的采样方式，也就是由图上某个节点 p 开始，进行若干次随机游走（代码5-9中的5~9行），并记录游走过程中访问到的其他节点和访问次数（代码5-9中的第7行）。最后， $Visits[p]$ 中访问次数最多的前 T 个邻居，也就是对节点 p 影响最大的前 T 个邻居，作为采样结果，被纳入计算节点 p 的子图中。

代码 5-9 基于随机游走的邻居采样

Input: Bipartite Graph G , number of steps per node nw , random-walk reset probability α

Output: top-k highly visited nodes

```
1: Initialize Visits={}
2: for each pin  $p$  in  $G$  do
3:   current_node= $p$ 
4:   for each step  $i$  in  $nw$  do
5:     while true do
6:       current_node=SampleFromNeighbors(current_node)
7:       Visits[p][current_node]+=1
8:       Flip a random coin with heads probability= $\alpha$ . Break if heads.
9:     end while
10:    end for
11:  end for
12:  Return top-k neighbors of each pin from Visits.
```



$Visits[p]$ 记录了从节点 p 出发游走过程中，对其他节点的访问次数，相当于其他节点对节点 p 的重要性，除了用于邻居采样，还在PinSage其他地方发挥了重要作用：

- 代码5-7中的AGG是聚合邻居向量的函数，PinSage将AGG实现成一个加权平均函数，权重就来自 $Visits$ 记录的访问次数。
- 对于某个Pin节点 p ，PinSage对 $Visits[p]$ 进行降序排列，排名在3000 ~ 5000的其他Pin节点被认为与 p 相关但是相关性又没那么强，从中采样一部分作为 p 的HardNegative参与训练，可以提升模型的分辨力。

基于MapReduce分布式推理

模型按照代码5-8训练完毕，接下来我们要生成图上所有节点的Embedding，这依然不是一个简单的任务。

- 图上有上亿个节点，单机计算肯定来不及，必须采用分布式并发计算。
- 代码5-8描述的是训练时前代生成Embedding的过程，并不适用于推荐预测。原因一，代码中采用了“邻居采样”而引入了随机性，预测时必须聚合节点的全部邻居。
- 第二个原因，代码5-8是基于Mini-Batch的。尽管两个Batch所包含的节点不同，但是为了计算这两个Batch而生成的各层子图，很可能包含了相同的邻居节点。这些共享邻居节点上的卷积结果被反复计算，浪费了算力。

因为以上三点原因，PinSage开发了基于MapReduce的分布式算法，并生成各节点的Embedding。

第 k 层卷积的map伪代码见代码5-10，各节点独立地将上一轮卷积得到的老向量，映射成新向量，并发送给邻居节点，对应代码5-7的第一行。

代码 5-10 PinSage第 k 层卷积时的 `Mapper` 伪代码

```
class Mapper:  
    def __init__(self, k) -> None:  
        # 装载第k层卷积的权重  
        self._Q = load_variables(k, "Q")  
        self._q = load_variables(k, "q")  
  
    def map(self, node, embedding, neighNodes, weightsToNeigh):  
        """  
            node: 某个节点  
            embedding: node上一轮卷积后得到的向量  
            neighNodes: node的邻居节点  
            weightsToNeigh: node对其每个邻居的重要程度  
        """  
        # 线性映射+非线性激活，获得要发向邻居的新向量  
        emb4neigh = ReLU(self._Q * embedding + self._q)  
  
        for (destNode, weight2neigh) in zip(neighNodes, weightsToNeigh):  
            message = (node, emb4neigh, weight2neigh)  
            # node作为消息来源，将其新向量发往，目标destNode  
            # MapReduce框架会以destNode为key，将所有message聚合起来  
            emit(destNode, message)
```

```
# node自己作为目标节点，也要参与reduce，所以也要发出
emit(node, (node, embedding, 1))
```

第k层卷积的reduce伪代码见代码5-11，各节点聚合邻居节点发来的向量，再映射成新向量，对应代码5-7中的1~3行。

代码 5-11 PinSage第k层卷积时的 `reducer` 伪代码

```
class Reducer:

    def __init__(self, k) -> None:
        # 装载第k层卷积的权重
        self._W = load_variables(k, "W")
        self._w = load_variables(k, "w")

    def reduce(self, node, messages):
        """
        node: 当前节点
        messages: node的所有邻居发向node的消息集合
        """

        old_self_embedding = None
        neigh_agg_embedding = zero_vector() # 初始化空向量
        neigh_sum_weight = 0

        for (nbNode, nbEmbedding, nbWeight) in messages:
            # 每个消息由三部分组成：
            # nbNode: 从哪个邻居节点发来的
            # nbEmbedding: 邻居节点发来的向量
            # nbWeight: 邻居节点对当前节点node的重要程度
            if nbNode == node:
                old_self_embedding = nbEmbedding # 当前节点上一轮的向量
            else:
                neigh_agg_embedding += nbWeight * nbEmbedding
                neigh_sum_weight += nbWeight

        # 所有邻居向当前节点扩散信息的加权平均
        neigh_agg_embedding = neigh_agg_embedding / neigh_sum_weight

        new_embedding = ReLU(self._W * concat(neigh_agg_embedding, old_self_embedding) + self._w)
        new_embedding = new_embedding / 10.0 * (new_embedding - 10.0 * mean(neigh_agg_embedding))
```

```

new_embedding = new_embedding / l2_norm(new_embedding) # L2 normalization

emit(node, new_embedding) # MapReduce会把每个节点和它的新向量，保存到HDFS上

```

代码 5-12 调用代码 5-10 和代码 5-10，将以上 map、reduce 工作迭代 num_layers 轮，就得到各节点的最终的 Embedding。

代码 5-12 PinSage 分布式推理的入口函数

```

def inference_embeddings():

    # 把每层卷积所需要的权重，广播到集群中的每台机器上

    for k in range(num_layers):
        broadcast(Q[k], q[k], W[k], w[k])

    for k in range(1, num_layers+1):
        old_emb_path = get_emb_path(k-1) # 上一轮卷积结果的保存路径

        # 上一轮的卷积结果，每个节点只有(node, embedding)信息
        # 还要再拼接上，每个节点的邻居列表，和当前节点对每个邻居的重要性
        # 拼接后，每个节点的数据包括：(node, embedding, neighNodes, weightsToNeigh)，才符合mapper的需求
        # 保存到HDFS的input_path路径下
        input_path = join_emb_with_neighbors(old_emb_path)

        run_distributedly(input=input_path,
                           job=Mapper(k),
                           output=temp_path)

        output_path = get_emb_path(k)
        run_distributedly(input=temp_path,
                           job=Reducer(k),
                           output=output_path)

    # 各节点最终embedding的保存路径
    final_path = get_emb_path(num_layers)

```

PinSage 源码演示

开源项目 Deep Graph Library (DGL) [29] 提供了一个 PinSage 的实现。由于完整实现代码繁复，而本书篇幅有限，所以这里只列出并注释 PinSage 的两个关键步骤“抽样生成每层的计算子图”和“图卷积”的实现代码，起到一个提纲挈领的作用。剩下的实现细节，请感兴趣的读者移步源码或文献[30]。

通过邻居采样生成各层的计算子图的代码如代码 5-13所示，对应代码 5-8中的1 ~ 7行。

代码 5-13 DGL实现的邻居采样

```
class NeighborSampler(object):

    """邻居采样，生成各层卷积所需要的计算子图"""

    def __init__(self, g, ....):
        self.g = g
        # 每层都有一个采样器，根据随机游走来决定某节点邻居的重要性
        # 可以认为经过多次游走，落脚于某邻居节点的次数越多，则这个邻居越重要，就更应该优先作为邻居
        self.samplers = [dgl.sampling.PinSAGESampler(g, ....) for _ in range(num_layers)]

    def sample_blocks(self, seeds, heads=None, tails=None, neg_tails=None):
        blocks = []
        for sampler in self.samplers:
            # 通过随机游走，选择重要邻居，构成子图
            frontier = sampler(seeds)

            if heads isnotNone:
                # 如果是在训练，需要将heads->tails和head->neg_tails这些待预测的边都去掉
                # 否则head/tails的信息会沿着边相互传统，引发信息泄漏
                eids = frontier.edge_ids(torch.cat([heads, heads]), torch.cat([tails, neg_tails]))
                if len(eids) > 0:
                    old_frontier = frontier
                    frontier = dgl.remove_edges(old_frontier, eids)

            # 只保留seeds这些节点，将frontier压缩成block
            block = compact_and_copy(frontier, seeds)

            # 本层的输入节点就是下一层的seeds
            seeds = block.srcdata[dgl.NID]
            blocks.insert(0, block)

        return blocks # 各层卷积所需要的计算子图
```

单层图卷积的代码如代码 5-14所示，对应代码 5-7。

代码 5-14 DGL实现的单层图卷积

```

class WeightedSAGEConv(nn.Module):

    """单层图卷积"""

    def forward(self, g, h, weights):
        """
        g : 某一层的计算子图, 就是NeighborSampler生成的block
        h : 是一个tuple, 包含源节点、目标节点上一层的embedding
        weights : 边上的权重
        """

        h_src, h_dst = h # 源节点、目标节点上一层的embedding
        with g.local_scope():
            # 将src节点上的原始特征映射成hidden_dims长, 存储于各节点的'n'字段
            # Q是线性映射的权重, act是激活函数
            g.srcdata['n'] = self.act(self.Q(self.dropout(h_src)))
            # 边上的权重, 存储于各边的'w'字段
            g.edata['w'] = weights.float()

            # DGL采取"消息传递"方式来实现图卷积
            # g.update_all是更新全部节点, 更新方式是:
            # fn.u_mul_e: src节点上的特征'n'乘以边权重'w', 构成消息'm'
            # fn.sum:      dst节点将所有接收到的消息'm', 相加起来, 更新dst节点的'n'字段
            g.update_all(fn.u_mul_e('n', 'w', 'm'), fn.sum('m', 'n'))

            # 将边上的权重w拷贝成消息'm'
            # dst节点将所有接收到的消息'm', 相加起来, 存入dst节点的'ws'字段
            g.update_all(fn.copy_e('w', 'm'), fn.sum('m', 'ws'))

            # 某个dst节点的n字段, 已经被更新成, 它的所有邻居节点的embedding的加权和
            n = g.dstdata['n']
            # 某个dst节点的ws字段, 是指向它的所有边上权重之和
            ws = g.dstdata['ws'].unsqueeze(1).clamp(min=1)

            # n / ws: 将邻居节点的embedding, 做加权平均
            # 再拼接上一轮卷积后, dst节点自身的embedding
            # 再经过线性映射 (W) 与非线性激活 (act), 得到这一轮卷积后各dst节点的embedding
            z = self.act(self.W(self.dropout(torch.cat([n / ws, h_dst], 1)))) 

            # 本轮卷积后, 各dst节点的embedding除以模长, 进行归一化
            z_norm = z.norm(2, 1, keepdim=True)

```

```

z_norm = z_norm.clamp_(_, 1e-12)
z_norm = torch.where(z_norm == 0, torch.tensor(1.).to(z_norm), z_norm)
z = z / z_norm
return z

```

5.6.3 异构图上的GCN

上一节介绍的PinSage，主要用于同构图，也就是图上只有一类节点或一类边关系。但是推荐系统中显然存在着多种实体（比如图5-7中的用户、商品、品牌、店铺），和多种交互关系（比如图5-7中的购买、浏览、关注）。因此，业界也有一些实践在异构图上做图卷积，利用多元化的节点和边信息，学习出高质量的节点向量。我们知道，图卷积的核心在于，邻居节点的信息沿着边，传递到目标节点上再聚合。所以异构图上卷积的难点就在于，如何让不同类型的邻居节点“分门别类”地传递、聚合信息，而不是无视类型差别，简单粗暴地“一锅烩”。

思路之一就是将异构图拆解成多个同构图。比如Pinterest提出的PinSage的升级版MultiBiSage[31]：

- 首先，按照每种交互关系构建出一个二部图。比如，根据“Board收藏Pin”这个关系构建出Pin~Board二部图，再根据“通过关键词搜索出Pin”这个关系构建出Pin~Keyword二部图，……，以此类推，一共构建出K个二部图。
- 接下来，在每种单一关系的二部图上，采用PinSage或其他图卷积算法，得到Pin节点的Embedding。这时一个Pin节点 p 会得到 K 个Embedding，记为 $[h_p^{(1)}, h_p^{(2)}, \dots, h_p^{(K)}]$ 。
- 最后，将 p 节点的这 K 个Embedding，当成一个序列，喂入Transformer进行Self-Attention，其结果就是节点 p 融合了各种交互关系的Embedding。

MultiBiSage的思路是让节点间的信息传递只发生在具有单一关系的同构图上，只在最后才融合多种关系下的卷积结果生成节点向量。微信的GraphTR[15]采用了完全不同的思路，GraphTR直接在异构图上进行卷积，但是不同类型的邻居节点在卷积时要“分门别类”。

- 在GraphTR的异构图上，一共有用户、视频、视频标签、视频来源四类节点。在进行第 k 层卷积时，先将目标节点 t 的邻居节点，按照节点类型分组。
- 假设目标节点 t 的第 $k-1$ 层的卷积结果用 $h_t^{(k-1)}$ ，而它有视频邻居节点有 $[v_1, v_2, \dots, v_M]$ ，它们的 $k-1$ 层卷积结果表示为 $NS_{video} = [h_{v_1}^{(k-1)}, h_{v_2}^{(k-1)}, \dots, h_{v_M}^{(k-1)}]$ 。我们可以直接拿 $h_t^{(k-1)}$ 当Query去和 NS_{video} 做Attention，得到 $h_{t_{video}}^{(k)}$ ，表示所有视频邻居的信息向目标节点 t 聚合的结果。也可以像论文原文那样，将 $h_t^{(k-1)}$ 与 NS_{video} 拼接起来，再喂入Transformer得到 $1+M$ 个新向量，然后再Average Pooling得到 $h_{t_{video}}^{(k)}$ ，即 $h_{t_{video}}^{(k)} = AvgPooling(Transformer(concat(h_t^{(k-1)}, NS_{video})))$ 。
- 再把以上方法实施在其他类型的邻居上，最终得到 $[h_{t_{video}}^{(k)}, h_{t_{user}}^{(k)}, h_{t_{tag}}^{(k)}, h_{t_{media}}^{(k)}]$ ，分别表示“视频”/“用户”/“标签”/“来源”邻居的信息向目标节点 t 聚合的结果。

- 最后在 $\mathbf{H}_t^{(k)} = [\mathbf{h}_{t_{\text{video}}}^{(k)}, \mathbf{h}_{t_{\text{user}}}^{(k)}, \mathbf{h}_{t_{\text{tag}}}^{(k)}, \mathbf{h}_{t_{\text{media}}}^{(k)}]$, 上做一个FM Pooling, 得到目标节点t第k层的卷积结果, 即

$$\mathbf{h}_t^{(k)} = \sum_{i=1}^4 \sum_{j=i+1}^4 \mathbf{H}_t^{(k)}[i] \odot \mathbf{H}_t^{(k)}[j]。$$

GraphTR的信息聚合方式, 先用Transformer实现同一类型的邻居节点内部的信息交叉, 再用FM实现不同类型邻居节点之间的信息交叉, 给模型性能带来显著提升。

5.7 本章小结

本章讲解推荐系统中的召回算法

- 5.1节介绍了传统召回算法, 和多路召回合并算法。重点是, 合并多路召回结果时, 不要人为规定每路召回的优先级, 而要确保各路召回的精华都能够被合并时最终结果集。
- 5.2节介绍了作者提出的“向量化召回统一建模框架”, 帮助读者系统化地理解向量召回, 还能指导读者从各种算法中取长补短, 构建适合自己业务的算法。重点是, 召回中的负采样主要依靠随机负采样, 千万不能(只)拿“曝光未点击”当负样本。
- 5.3节介绍类Word2Vec召回算法。重点是Airbnb如何根据自己的业务需求, 创造性地增加额外的正负样本, 改造经典算法。
- FM作为推荐算法中的瑞士军刀, 5.4节介绍它的召回功能。重点是如何通过采样打压热门物料, 增强召回结果的个性化与多样性。另外, FM增广Embedding, 使其不仅能刻画用户、物料间的匹配程度, 还反映物料本身的受欢迎程度, 也值得学习。
- 5.5节讲解大厂召回的绝对主力, 双塔模型。重点包括, Batch内负采样, 进一步简化计算; 还有实现Sampled Softmax Loss时的各种实战技巧。
- 5.6节讲解基于GCN的召回算法。重点是介绍PinSage是如何在一个上十亿规模的超大型图上实现图卷积, 包括Mini-Batch训练、邻居采样、基于MapReduce分布式推理等实战性极强的技巧。

5.8 本章参考文献

- [1] ZHU H, LI X, ZHANG P, 等. Learning Tree-based Deep Model for Recommender Systems[C/OL]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &

Data Mining. 2018: 1079-1088[2022-06-29]. <http://arxiv.org/abs/1801.02294>.

DOI:10.1145/3219819.3219826.

[2] ZHU H, CHANG D, XU Z, 等. Joint Optimization of Tree-based Index and Deep Model for Recommender

Systems[M/OL]. arXiv, 2019[2022-06-29]. <http://arxiv.org/abs/1902.07565>.

DOI:10.48550/arXiv.1902.07565.

[3] 石塔西. 三问TDM[EB/OL]. <https://mp.weixin.qq.com/s/VFXLwZxKbmbJUYpSAPdSHg>.

[4] SCHELTER S, BODEN C, MARKL V. Scalable similarity-based neighborhood methods with
MapReduce[C/OL]//Proceedings of the sixth ACM conference on Recommender systems - RecSys '12. Dublin,
Ireland: ACM Press, 2012: 163[2022-06-30]. <http://dl.acm.org/citation.cfm?doid=2365952.2365984>.
DOI:10.1145/2365952.2365984.

[5] HU Y, KOREN Y, VOLINSKY C. Collaborative Filtering for Implicit Feedback Datasets[C/OL]//2008 Eighth
IEEE International Conference on Data Mining. Pisa, Italy: IEEE, 2008: 263-272[2022-07-01].
<http://ieeexplore.ieee.org/document/4781121/>. DOI:10.1109/ICDM.2008.22.

[6] Spark Collaborative Filtering[CP/OL]. <https://spark.apache.org/docs/latest/ml-collaborative-filtering.html>.

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

[8] Milvus[CP/OL]. <https://milvus.io/>.

[9] BARKAN O, KOENIGSTEIN N. Item2Vec: Neural Item Embedding for Collaborative
Filtering[EB/OL]//arXiv.org. (2016-03-14)[2023-02-07]. <https://arxiv.org/abs/1603.04259v3>.
DOI:10.48550/arXiv.1603.04259.

[10] COVINGTON P, ADAMS J, SARGIN E. Deep Neural Networks for YouTube
Recommendations[C]//Proceedings of the 10th ACM Conference on Recommender Systems. New York, NY,
USA, 2016.

[11] GRBOVIC M, CHENG H. Real-time Personalization using Embeddings for Search Ranking at
Airbnb[C/OL]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &
Data Mining. London United Kingdom: ACM, 2018: 311-320[2022-05-27].
<https://dl.acm.org/doi/10.1145/3219819.3219885>. DOI:10.1145/3219819.3219885.

[12] HUANG P S, HE X, GAO J, 等. Learning deep structured semantic models for web search using clickthrough
data[C/OL]//Proceedings of the 22nd ACM international conference on Conference on information &

knowledge management - CIKM '13. San Francisco, California, USA: ACM Press, 2013: 2333-2338[2022-05-31]. <http://dl.acm.org/citation.cfm?doid=2505515.2505665>. DOI:10.1145/2505515.2505665.

[13] WANG J, HUANG P, ZHAO H, 等. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba: arXiv:1803.02349[R/OL]. arXiv, 2018[2022-05-27]. <http://arxiv.org/abs/1803.02349>. DOI:10.48550/arXiv.1803.02349.

[14] YING R, HE R, CHEN K, 等. Graph Convolutional Neural Networks for Web-Scale Recommender Systems[C/OL]//Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018: 974-983[2022-06-24]. <http://arxiv.org/abs/1806.01973>. DOI:10.1145/3219819.3219890.

[15] LIU Q, XIE R, CHEN L, 等. Graph Neural Network for Tag Ranking in Tag-enhanced Video Recommendation[C/OL]//Proceedings of the 29th ACM International Conference on Information & Knowledge Management. Virtual Event Ireland: ACM, 2020: 2613-2620[2022-06-24]. <https://dl.acm.org/doi/10.1145/3340531.3416021>. DOI:10.1145/3340531.3416021.

[16] HUANG J T, SHARMA A, SUN S, 等. Embedding-based Retrieval in Facebook Search[C/OL]//Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020: 2553-2561[2022-05-27]. <http://arxiv.org/abs/2006.11632>. DOI:10.1145/3394486.3403305.

[17] candidate_sampling.pdf[Z/OL]. [2022-05-27].
https://www.tensorflow.org/extras/candidate_sampling.pdf.

[18] MIKOLOV T, SUTSKEVER I, CHEN K, 等. Distributed Representations of Words and Phrases and their Compositionality[C/OL]//Advances in Neural Information Processing Systems: 卷 26. Curran Associates, Inc., 2013[2022-05-27].
<https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html>.

[19] tf.nn.nce_loss[CP/OL]. Google. https://www.tensorflow.org/api_docs/python/tf/nn/nce_loss.

[20] YI X, YANG J, HONG L, 等. Sampling-bias-corrected neural modeling for large corpus item recommendations[C/OL]//Proceedings of the 13th ACM Conference on Recommender Systems. Copenhagen Denmark: ACM, 2019: 269-277[2022-05-27]. <https://dl.acm.org/doi/10.1145/3298689.3346996>. DOI:10.1145/3298689.3346996.

[21] YANG J, YI X, ZHIYUAN CHENG D, 等. Mixed Negative Sampling for Learning Two-tower Neural Networks in Recommendations[C/OL]//Companion Proceedings of the Web Conference 2020. Taipei Taiwan: ACM, 2020: 441-447[2022-05-27]. <https://dl.acm.org/doi/10.1145/3366424.3386195>. DOI:10.1145/3366424.3386195.

[22] WANG J, ZHU J, HE X. Cross-Batch Negative Sampling for Training Two-Tower Recommenders: arXiv:2110.15154[R/OL]. arXiv,2021[2022-05-27]. <http://arxiv.org/abs/2110.15154>. DOI:10.48550/arXiv.2110.15154.

[23] LV F, JIN T, YU C, 等. SDM: Sequential Deep Matching Model for Online Large-scale Recommender System: arXiv:1909.00385[R/OL]. arXiv, 2019[2022-05-23]. <http://arxiv.org/abs/1909.00385>. DOI:10.48550/arXiv.1909.00385.

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

[25] LIU Y, RANGADURAI K, HE Y, 等. Que2Search: Fast and Accurate Query and Document Understanding for Search at Facebook[C/OL]//Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, Virtual Event Singapore: ACM, 2021: 3376-3384[2022-05-05].
<https://dl.acm.org/doi/10.1145/3447548.3467127>. DOI:10.1145/3447548.3467127.

[26] Building deep retrieval models[EB/OL].
https://www.tensorflow.org/recommenders/examples/deep_recommenders.

[27] HAMILTON W L, YING R, LESKOVEC J. Inductive Representation Learning on Large Graphs[M/OL]. arXiv, 2018[2022-06-25]. <http://arxiv.org/abs/1706.02216>. DOI:10.48550/arXiv.1706.02216.

[28] ANKIT JAIN. Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations[EB/OL]. (2019-12-04). <https://eng.uber.com/uber-eats-graph-learning/>.

[29] Deep Graph Library[CP/OL]. <https://github.com/dmlc/dgl>.

[30] 石塔西. PinSAGE召回模型及源码分析(3): PinSAGE模型及训练[EB/OL].
<https://mp.weixin.qq.com/s/aM2o4unfVO0EE16PVwDfzg>.

[31] GURUKAR S, PANCHAL N, ZHAI A, 等. MultiBiSage: A Web-Scale Recommendation System Using Multiple Bipartite Graphs at Pinterest[M/OL]. arXiv, 2022[2022-06-24]. <http://arxiv.org/abs/2205.10666>. DOI:10.48550/arXiv.2205.10666.