第 5 章

7min

知识图谱与推荐算法

经过第 4 章的学习，大家应该已经对图有了不错的认识。知识图谱也属于一种图，并且属于异构图。因为知识图谱的节点类型与边类型不止一个，在处理知识图谱时，不能忽略边本身的特征。对于算法工程师而言，知识图谱这种异构图显然更复杂，但是在物理世界中，反而知识图谱更贴近实际的应用场景，所以只懂得图论及图神经网络还不足以将自己称为能够处理图的推荐算法工程师。

本章会带领大家学习更贴近实际场景的基于知识图谱的推荐算法，但是知识图谱其实并不是从图论出发的知识，而是起源于 RDF 资源描述框架，所以基于知识图谱的推荐其实已发展多年，图神经网络兴起后再与图神经网络结合便产生了知识图谱结合图神经网络的推荐算法，如 KGCN 和 KGAT 等。这些推荐算法出现之后，会使人感觉之前的一些知识图谱推荐算法略显过时且烦琐，但是梳理过去的脉络有助于大家更好地理解目前算法的情况，以及未来其他基础算法或者神经网络领域发展后大家能很快地跟上甚至引领其发展。

所以本章的前五节与图神经网络无关，介绍的是知识图谱推荐领域的一些发展脉络及出众的算法。尤其 5.1.1 节与 5.1.2 节是知识图谱很基础的内容，希望大家不要跳过此部分内容。

5.1 知识图谱基础

5.1.1 知识图谱定义

4min

知识图谱(Knowledge Graph)简称 KG，正如其名字一样。知识图谱是表示知识的图谱，图 5-1 展示了一个知识图谱的图例。

知识图谱在图论中属于很复杂的异构图，因为节点与边的类型均大于一，且重点是在做算法时不能忽略边所起的作用。知识图谱是为了给人类观看而存在的，假设把图 5-1 中所有的边上的注释遮盖掉，则该图表示的信息就很不直观了。

通常知识图谱是有向图，对称的边往往会是不同的边类型，例如图中(波尔 老师 海森堡) 表示的是波尔是海森堡的老师，而与其对称的(海森堡 学生 波尔)表示的是海森堡是波尔的学生。

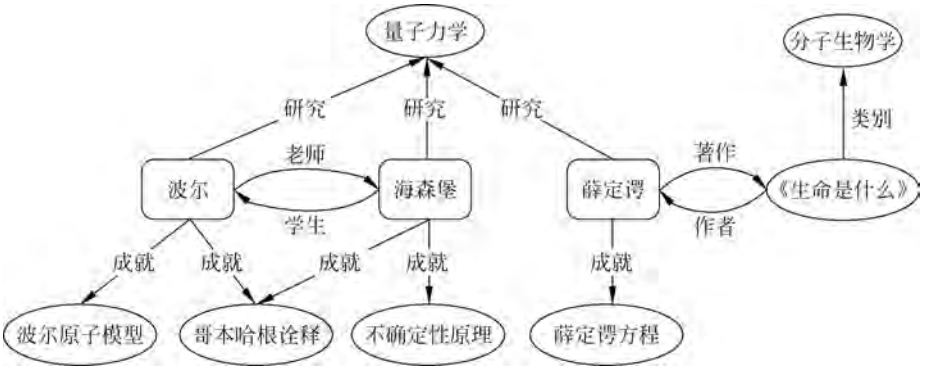


图 5-1　知识图谱示意图

可以用三元组来表示一对实体的关系，例如(波尔 老师 海森堡)被称为三元组事实，而一整张知识图谱则可以由很多三元组事实的集合表示。

5.1.2　RDF 到 HRT 三元组

资源描述框架(Resource Description Framework，RDF) 是描述网络资源的 W3C 标准。W3C 指万维网联盟(World Wide Web Consortium)，W3C 标准也是互联网数据传输要遵守的所有标准，RDF 是其中一个描述网络资源的标准。

在 2004 年 2 月，RDF 成为 W3C 标准之一。通俗地讲，RDF 标准是一个试图把天下所有信息都以同一种方式描述的结构。这个结构就是后来俗称的 RDF 三元组，简单地说是三元组结构，在其发展过程中也有很多改革，但到今天，仅需要理解为它的表现形式是由(头实体，关系，尾实体)构成的，即(Head，Relation，Tail)，简称为 HRT 三元组。该结构形成了今天知识图谱的数据形式。

回顾第 4 章的图表示方法，可以发现三元组与边集很相似，唯一不同的是中间多了边的表示。假设 HRT 三元组是 $(1,a,2)$、$(1,a,3)$、$(1,b,5)$、$(2,b,4)$、$(3,c,4)$，则由此表示的图就如图 5-2 所示。

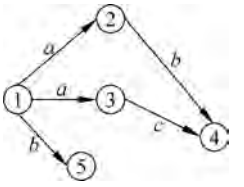


图 5-2　三元组边集

4min

所以对于知识图谱而言，是先有三元组后有图，直到 2012 年 5 月 17 日，谷歌公司正式提出了知识图谱的概念。

5.1.3　知识图谱推荐算法与图神经网络推荐算法的发展脉络

知识图谱推荐算法与图神经网络推荐算法的发展脉络如图 5-3 所示。

推荐算法的确无孔不入，任何基础知识都能衍生出推荐算法，图论本身也衍生出了推荐算法，例如链路预测系列，而深度学习本身的推荐算法就更多了。总而言之，这里主要是让大家搞清楚知识图谱和图神经网络的关系，知道它们各自都由不同领域发展而来而又汇聚在一起。

3min

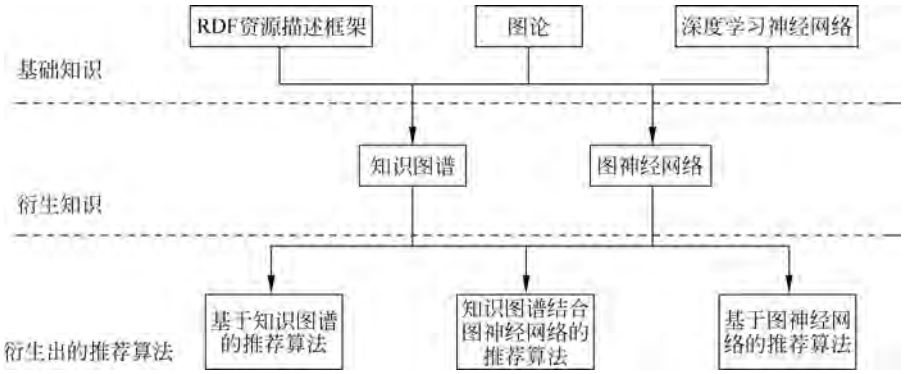


图 5-3　知识图谱与图神经网络推荐算法的发展脉络图

5.1.4　知识图谱推荐算法的概览

首先需要提醒大家的是知识图谱本身的知识点非常多，基于知识图谱的推荐算法也很多。如果要专门研究知识图谱与知识图谱推荐仅看本书是不够的，但是本书的优势在于提取了非常基础且非常重要的入门关键知识点，可以带领大家梳理琐碎的推荐算法知识。图 5-4 是根据专业从事知识图谱推荐算法的学者发表的一篇综述整理出的知识图谱推荐算法概览图。

5min

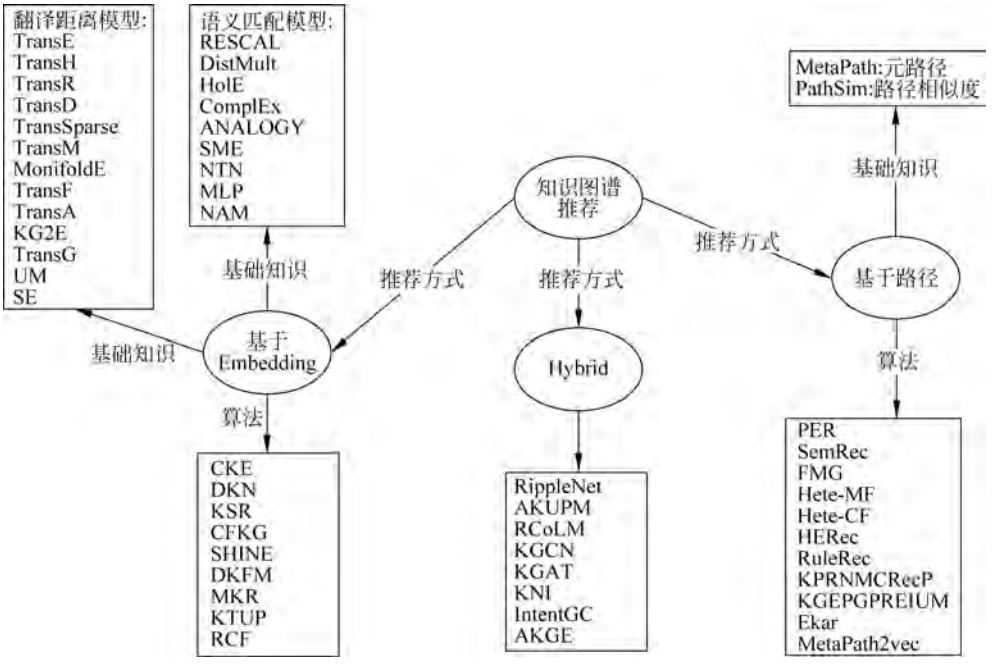


图 5-4　知识图谱推荐算法业内概览图谱

学者大体将基于知识图谱的推荐算法分成了三类。

(1) 基于知识图谱 Embedding 的推荐算法。

该类算法的基础知识是知识图谱 Embedding (简称 KGE,在 5.2 章会细讲),仅 KGE 算法就有很多,而这些是知识图谱极其基础的内容,并且并非是图 Embedding,而是由 HRT 三元组为核心衍生出的一系列 Embedding 方法。

(2) 基于知识图谱路径的推荐算法。

此类算法与图论相关,因为需要用到图路径的知识点。

(3) HyBrid,即两者结合。

此类算法是图路径结合 KGE 的推荐算法,值得注意的是如今基于图神经网络的知识图谱推荐算法也被分到了这里。

当然以上仅仅是一种分类方式,大家完全可以自己建立自己的算法分类索引,例如将结合图神经网络的知识图谱推荐算法分为第四类。只有自己心目中有了自己的分类索引后,才能梳理出属于自己的推荐算法体系,并且还能基于此进一步细分之后更有助于自己学习推荐算法及自己推导推荐算法。

本书对于这些算法的分类其实是这一章节的目录。图 5-5 展示的是本书的知识分类及准备介绍的算法。

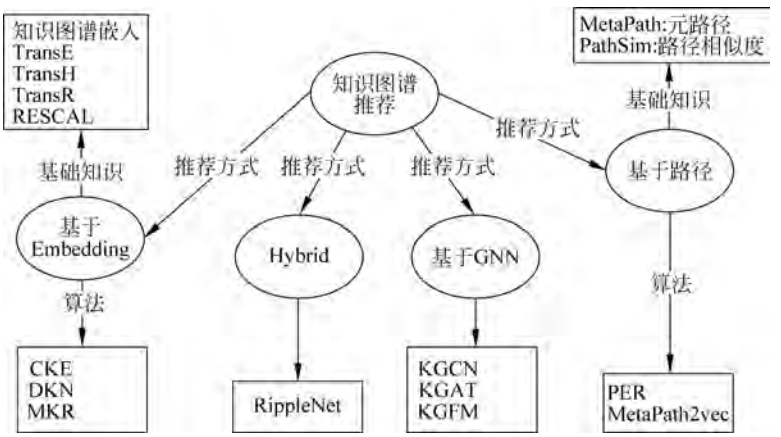


图 5-5　知识图谱推荐算法(本书知识梳理图谱)

5.1.5　基于知识图谱推荐的优劣势

1. 劣势

(1) 前置知识太多,学习成本大。一个知识图谱推荐算法工程师起码得掌握知识图谱本身的基础知识,也得对图论有一定了解。目前还需加上图神经网络,所以相对于 ALS 或者 FM 等简单有效的算法,基于知识图谱的推荐算法前置知识实在太多。

(2) 量级较重,因为要做知识图谱的推荐算法需要先建立知识图谱,这本身是一个大工程,中小型企业没有这个余力,也的确没有必要这样去构建推荐系统。因为中小型企业尤其

是创业公司讲究的是最小可执行原则,近邻 CF+FM 可以快速将推荐系统搭建起来。如果建立知识图谱后再去构建推荐系统会显得雷声大雨点小。

6min

2. 优势

(1) 能够更好地挖掘特征间的隐藏关联,因为万物都被图谱连接着。

(2) 与图的优势一样能更有效地描述不规则的数据。

(3) 可解释性更好,因为知识图谱本身是数据可视化的一种操作,人类在看到图时总能有一图胜千言的感触。深度学习下的推荐算法往往会推荐出来与用户特征或者用户历史浏览记录看似毫无关联的物品。此时外行人便会怀疑推荐系统的可靠性,而如果有知识图谱的存在,则可以沿着路径找到被推荐物品和历史记录间的关系,从而提高推荐系统的可解释性。

综合来讲,基于知识图谱的推荐系统一定不是轻量级的,但是如果要深入优化推荐系统,则基于知识图谱是非常好的选择。

5min

5.1.6 Freebase 数据集介绍

既然本章要学习的算法是基于知识图谱的,仅靠 Movielens 的开源数据集会显得不够,所以需要专业的开源知识图谱数据,比较合适的是 Freebase。

Freebase 是个类似维基百科的知识库网站,Freebase 中的数据是结构化的,所以是提炼知识图谱数据的绝佳网站,像一些知识图谱开源的数据集 FB15k 和 FB237 等都源自 Freebase。

但 Freebase 仅仅是知识图谱数据集,如果要学习推荐算法,则还需要将这些 Freebase 数据与推荐系统开源数据做连接。本书附带项目中的 recbyhand\data_set\ml-100k\kg_index.tsv 是通过 Freebase 与另外一个名为 Kb4rec 的开源项目结合后处理得到的数据集。

基于知识信息的推荐系统数据(Knowledge Base Information For Recommender System,Kb4rec[2])是中国人民大学信息学院的一个项目。此项目将知识图谱数据与推荐开源数据做了一个连接,例如将 Movielens 数据集中的电影与 Freebase 中的实体建立了映射,如图 5-6 所示。

图 5-6 Movielens id 映射 Freebase entity id

图 5-6 中左边的数字是 Movielens 中的电影 id,而右边的 m. 开头的那些字符串便是这些电影在 Freebase 上对应的实体 id,所以有了这个映射表后,再通过一些处理便可以得到索引化的数据,如 kg_index.tsv,中间的过程不是本书的重点,所以就不展开讲解了。

总之大家知道 kg_index.tsv 中的数据是源自 Freebase 的 HRT 三元组即可,而 H 与 T 包含了 Movielens 中的 movie 数据索引,示例代码中的数据已经将该文件中的索引与 recbyhand\data_set\ml-100k\rating_index.tsv 及 recbyhand\data_set\ml-100k\rating_index_5.tsv 中的第二列(电影的索引)对应,意味着 rating_index.tsv 中第二列的数字如果与 kg_index.tsv 中第一列或者第三列的

数字一样，则表示它们在物理上代表着同一部电影。

5.2 Knowledge Graph Embedding 知识图谱嵌入

按下来学习知识图谱最基础的知识，即知识图谱嵌入(Knowledge Graph Embedding，KGE)。知识图谱嵌入指的是用向量表示知识图谱中实体和关系的操作。作为知识图谱最基础的知识与图论无关，而是通过 HRT 三元组相互间的运算训练得到各自的向量表示。具体的训练方法可分为两个大类。

1. **翻译距离模型**(Translational Distance Models)

2min

翻译距离模型是将 **Tail** 向量视作由 **Head** 向量经过 **Relation** 向量的翻译距离所得到的，评分函数可认为向量间的欧氏距离。

2. **语义匹配模型**(Semantic Matching Models)

语义匹配模型是通过求 **Head** 向量经过 Relation 空间的线性变换后与 **Tail** 向量之间的语义相似度来评分的。评分函数可认为向量间的夹角大小。

知识图谱嵌入在知识图谱的学科中属于一门大课，也有一个专门的研究方向。在推荐系统的书籍中不会讲太多这方面的知识，但是本书会很详细地介绍几个最基础的方法 TransE、TransH、TransR 和 RESCAL。翻译距离模型可被视为 TransE 的变种，而语义匹配模型可被视为 RESCAL 的变种。

5.2.1 翻译距离模型 TransE

15min

TransE 全称为 Translating Embeddings，直译为翻译嵌入。2013 年被提出，当时还没有翻译距离模型家族这种概念，所以 TransE 直接用了最笼统的名字。

假如 $\boldsymbol{h}$(Head)、$\boldsymbol{r}$(Relation)和 $\boldsymbol{t}$(Tail)均是二维向量，则可假设它们在空间中的位置如图 5-7 所示。

根据向量运算的规则，该图表达的是 $\boldsymbol{h}+\boldsymbol{r}=\boldsymbol{t}$，正如翻译距离模型的定义一般，**Head** 向量经过了 **Relation** 向量的翻译距离得到了 **Tail** 向量。

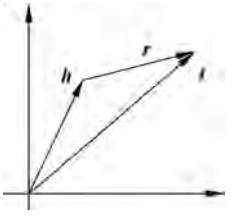


图 5-7 TransE 二维示意图[3]

是否可以直接写出如下的式子作为 TransE 的损失函数呢？

$$\| \boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t} \|_2 \tag{5-1}$$

$\| X \|_2$ 是 L^2 范数的计算符号，即计算向量的模长。直观上似乎式(5-1)越接近 0 代表模型越好，因为这表示 $\boldsymbol{h}+\boldsymbol{r}$ 越接近 $\boldsymbol{t}$。的确如此，但是如果这么做，随着 $\boldsymbol{h}$、$\boldsymbol{r}$ 和 $\boldsymbol{t}$ 这 3 个向量自身模长的减少，同样能够越来越接近 0。

所以需要采取负例采样的方式来避免上述情况。具体的损失函数如下[3]：

$$\text{loss}=\max(0, \| \boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t} \|_2 - \| \boldsymbol{h}'+\boldsymbol{r}-\boldsymbol{t}' \|_2 + m) \tag{5-2}$$

其中$(\boldsymbol{h},\boldsymbol{r},\boldsymbol{t})\in\boldsymbol{R}^k$。$k$ 是超参，代表它们的向量维度。$\max(0,x)$代表如果 x 比 0 大，

则取 x，反之取 0，这一步操作的意义是为了避免出现负的损失值。理解这些之后，重点只剩下如下这个式子了：

$$\| \boldsymbol{h} + \boldsymbol{r} - \boldsymbol{t} \|_2 - \| \boldsymbol{h}' + \boldsymbol{r} - \boldsymbol{t}' \|_2 + m \tag{5-3}$$

其中，$\| \boldsymbol{h} + \boldsymbol{r} - \boldsymbol{t} \|_2$ 这一项可视作由正采样得到的向量模长，$\| \boldsymbol{h}' + \boldsymbol{r} - \boldsymbol{t}' \|_2$ 这一项代表由负采样得到的向量模长，$\boldsymbol{h}'$ 与 $\boldsymbol{t}'$ 代表由负采样得到的 Head 和 Tail。$\boldsymbol{m}$ 是一个超参，是一个标量。因为期待由正采样得到的向量模长越低越好，而由负采样得到的向量模长越高越好，所以设置一个 m，代表它们的差距。在实际工作中对 m 的调参需要注意的是，如果 m 设得过大，则模型很难学，并且容易过拟合，但如果设得过小，则模型的精度不高。其实这种形式的损失函数叫作铰链损失函数(Hinge Loss)，公式如下：

$$\text{hingeloss} = \max(0, y - y' + m) \tag{5-4}$$

其中，y 是由正采样得到的预测值，y' 是由负采样得到的预测值。

负例采样的具体的操作是在原有正例的基础上，随机替换一个 Head 或者 Tail，注意每次仅需替换一个就可以了，具体会在之后的代码中详细介绍。

另外值得一提的是，每次迭代时都将 $\boldsymbol{h}$、$\boldsymbol{r}$ 和 $\boldsymbol{t}$ 的向量归一化(Normalize)一下，帮助模型迭代学习，即将它们的 L^2 范数等于 1，记作 $\| \boldsymbol{h} \|_2 = \| \boldsymbol{r} \|_2 = \| \boldsymbol{t} \|_2 = 1$。

TransE 模型类的代码如下：

```
#recbyhand\chapter5\s21_TransE.py
class TransE( nn.Module ):

    def __init__( self, n_entitys, n_Relations, dim = 128, margin = 1 ):
        super( ).__init__( )
        self.margin = margin                    #hinge_loss 中的差距
        self.n_entitys = n_entitys              #实体的数量
        self.n_Relations = n_Relations          #关系的数量
        self.dim = dim                          #Embedding 的长度

        #随机初始化实体的 Embedding
        self.e = nn.Embedding( self.n_entitys, dim, max_norm = 1 )
        #随机初始化关系的 Embedding
        self.r = nn.Embedding( self.n_Relations, dim, max_norm = 1 )

    def forward( self, X ):
        x_pos, x_neg = X
        y_pos = self.predict( x_pos )
        y_neg = self.predict( x_neg )
        return self.hinge_loss( y_pos, y_neg )

    def predict( self, x ):
        h, r, t = x
        h = self.e( h )
```

```
        r = self.r( r )
        t = self.e( t )
        score = h + r - t
        return torch.sum( score ** 2, dim = 1 ) ** 0.5

    def hinge_loss( self, y_pos, y_neg ):
        dis = y_pos - y_neg + self.margin
        return torch.sum( torch.ReLU( dis ) )
```

值得注意的是前向传播传入的 X 为经过负例采样包含正负例三元组的数据集，所以第一步是将正例 x_pos 与负例 x_neg 拆解开来。分别计算 TransE 的得分后传入 hinge_loss()函数中，以便输出损失函数的值。

读取数据及负例采样的方法的地址为 recbyhand\chapter5\dataloader4kge.py，其中重要的代码如下：

```
#recbyhand\chapter5\dataloader4kge.py
from torch.utils.data import Dataset

#继承 torch 自带的 Dataset 类,重构__getitem__与__len__方法
class KgDatasetWithNegativeSampling( Dataset ):

    def __init__( self, triples, entitys ):
        self.triples = triples                          #知识图谱 HRT 三元组
        self.entitys = entitys                          #所有实体集合列表

    def __getitem__( self, index ):
        '''
        :param index: 一批次采样的列表索引序号
        '''
        #根据索引取出正例
        pos_triple = self.triples[ index ]
        #通过负例采样的方法得到负例
        neg_triple = self.negtiveSampling( pos_triple )
        return pos_triple, neg_triple

    #负例采样方法
    def negtiveSampling( self, triple ):
        seed = random.random( )
        neg_triple = copy.deepcopy( triple )
        if seed > 0.5: #替换 Head
        rand_Head = triple[0]
            while rand_Head == triple[0]:               #如果采样得到自己,则继续循环
                #从所有实体中随机采样一个实体
                rand_Head = random.sample( self.entitys, 1 )[0]
```

```
            neg_triple[0] = rand_Head
        else: #替换 Tail
            rand_Tail = triple[2]
            while rand_Tail == triple[2]:
                rand_Tail = random.sample( self.entitys, 1 )[0]
            neg_triple[2] = rand_Tail
        return neg_triple

    def __len__( self ):
        return len( self.triples )
```

该方法继承自 torch.utils.data.Dataset,并重构了__getitem__()与__len__()方法。Dataset 实体可以传入 torch.utils.data.DataLoader 作为批次采样的迭代器。主要因为需要自己实现负例采样,所以这种继承 Dataset 类的写法会更方便。文件中__main__之后也有一段调用的例子,代码如下:

```
#recbyhand\chapter5\dataloader4kge.py
if __name__ == '__main__':
    #读取文件,得到所有实体列表、所有关系列表,以及 HRT 三元组列表
    entitys, Relations, triples = readKGData( )
    #传入 HRT 三元组与所有实体,得到包含正例与负例三元组的 Dataset
    train_set = KgDatasetWithNegativeSampling( triples, entitys )

    from torch.utils.data import DataLoader
    #通过 torch 的 DataLoader()方法按批次迭代三元组数据
    for set in DataLoader( train_set, batch_size = 8, shuffle = True ):
        #将正负例数据拆解开
        pos_set, neg_set = set
        #可以打印一下
        print( pos_set )
        print( neg_set )
        sys.exit()
```

更完整的代码可在给定的地址查看。KGE 系列的方法主要为了得到实体与关系的 Embedding,平时会作为类似但不完全等同于一个网络层而出现在各种知识图谱算法中,所以重点是实体与关系的 Embedding 可为后道工序服务,而并非预测 HingeLoss 本身。至于如何在推荐算法中使用 KGE 本书会在后面讲解。

5.2.2 翻译距离模型 TransH

TransH 全称为 Knowledge Graph Embedding by Translating on Hyperplanes[4],直译为在超平面上的知识图谱词嵌入。

由于 TransE 在一对多、多对一、多对多关系时或者自反关系上效果不是很好,所以

TransH 被提出。

自反关系：指 Head 和 Tail 相同。例如：

(曹操、欣赏、曹操) 这是自反关系，(曹操、欣赏、司马懿)这不是自反关系。

一对一：指同一组 Head 和 Relation 只会对应一个 Tail。例如：

(司马懿、妻子、张春华)，(诸葛亮、妻子、黄月英)。

一对多：指同一组 Head 和 Relation 会对应多个不同的 Tail。例如：

(司马懿、儿子、司马师)，(司马懿、儿子、司马昭)。

多对一：指多个 Head 会对应同一组 Relation 和 Tail。例如：

(司马师、父亲、司马懿)，(司马昭、父亲、司马懿)。

多对多：指多组 Head 和 Relation 对应多个 Tail。例如：

(司马懿、懂得、孙子兵法)，(司马懿、懂得、三略)，(司马懿、懂得、六韬)。

(诸葛亮、懂得、孙子兵法)，(诸葛亮、懂得、三略)，(诸葛亮、懂得、六韬)。

(周公瑾、懂得、孙子兵法)，(周公瑾、懂得、三略)，(周公瑾、懂得、六韬)。

图 5-8 区别了一对一与多对多的关系。

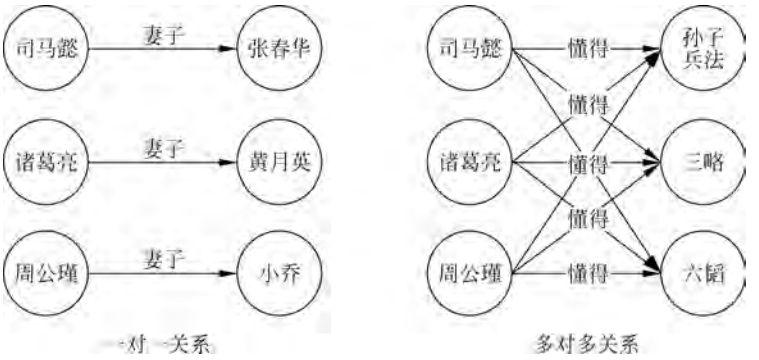


图 5-8　一对一与多对多关系的区别

为什么 TransE 会在一对多等关系上效果不好呢？例如这两组关系，(司马懿、儿子、司马师)，(司马懿、儿子、司马昭)。因为两组关系中都存在实体“司马懿”和关系“儿子”，如果只简单考虑 $\boldsymbol{h}+\boldsymbol{r}=\boldsymbol{t}$，则“司马师”=“司马懿”+“儿子”和“司马昭”=“司马懿”+“儿子”，所以“司马师”=“司马昭”，很显然，这并不是我们想要的结果。

再例如自反关系，(曹操、欣赏、曹操)，“曹操”+“欣赏”=曹操，所以“欣赏”=0。如果非要说假如存在自反关系，Relation 就该为 0，则轮到计算(曹操、欣赏、司马懿)时，如果将“欣赏”=0 代入则会产生“曹操”=“司马懿”的结果。当然 TransE 在实际迭代中不会这样，其原因是因为(曹操、欣赏、司马懿)这类的数据大概率会在训练集中占大多数，而(曹操、欣赏、曹操)会被它当作噪声数据，所以对效果的影响较小。

为了解决上述问题，2014 年 TransH 模型被提出，其中心思想是对每个关系定义一个超平面 $\boldsymbol{W}_r$，而 $\boldsymbol{h}_\perp$ 与 $t_\perp$ 作为 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 在超平面上的投影，将 $\boldsymbol{h}_\perp$ 与 $\boldsymbol{t}_\perp$ 代替 $\boldsymbol{h}$ 与 $\boldsymbol{t}$ 并满足：

$$\| \boldsymbol{h}_{\perp} + \boldsymbol{r} - \boldsymbol{t}_{\perp} \|_2 = 0 \tag{5-5}$$

Trans 示意图如图 5-9 所示。

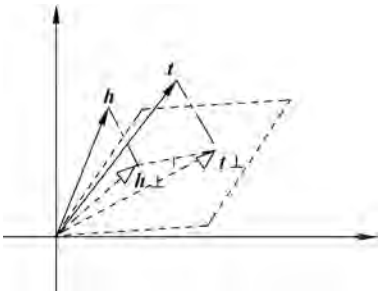


图 5-9 Trans H 示意图[4]

基础知识——超平面与法向量

超平面是指将 n 维空间的维度分割为 $n-1$ 维度的子空间。例如一个三维空间可以被一个二维的平面分成不可相交的两部分,一个四维的空间可被一个三维空间分为两部分,所以干脆就可以把负责分割空间的这个子空间统称为“超平面”,“平面”可视作“二维超平面”,线可视作为“一维超平面”。

法向量是正交于超平面的向量,通俗点讲是垂直,如图 5-10 所示,所以一个超平面可对应无数个法向量。如果法向量的 L^2 范数等于 1,则称为单位法向量。

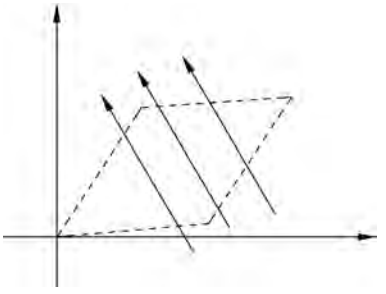


图 5-10 法向量与超平面

这么一来,像(司马懿、儿子、司马师),(司马懿、儿子、司马昭)在 TransH 中就不需要“柏灵筠”的向量等于“静姝”的向量了,仅仅需要它们在 $\boldsymbol{W}_r$ 上的投影相同。要知道投影相同不需要它们自身也相同,如图 5-11 所示,$\boldsymbol{a}$ 向量与 $\boldsymbol{b}$ 向量在 $\boldsymbol{x}$ 轴上的投影虽相同,但 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 可以是两个不同的向量。

该如何在数学上完成在超平面投影这个操作呢?首先需定义一个 $\boldsymbol{w}_r$,此 $\boldsymbol{w}_r$ 为 $\boldsymbol{W}_r$ 超平面的单位法向量,即 $\| \boldsymbol{w}_r \| = 1$。根据点积的定义:

$$\boldsymbol{h} \cdot \boldsymbol{w}_r = \| \boldsymbol{h} \| \times \| \boldsymbol{w}_r \| \times \cos\theta \tag{5-6}$$

其中，$\boldsymbol{h} \cdot \boldsymbol{w}_r$ 是点乘操作，即求内积，可表示为 $\boldsymbol{w}_r^T\boldsymbol{h}$。且因为 $\| \boldsymbol{w}_r \| = 1$，所以 $\boldsymbol{w}_r^T\boldsymbol{h} = \| \boldsymbol{h} \| \times \cos\theta$，即 $\boldsymbol{h}$ 在 $\boldsymbol{w}_r$ 方向上的长度，用此长度，再乘以单位法向量 $\boldsymbol{w}_r$，即是图 5-12 中的向量 $\boldsymbol{h}_{wr}$。

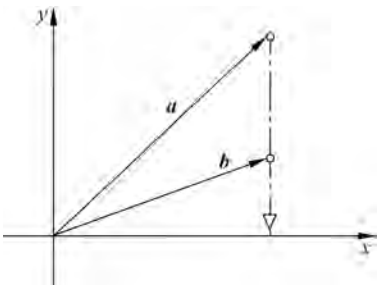


图 5-11　将 a 和 b 向量投影到 x 轴

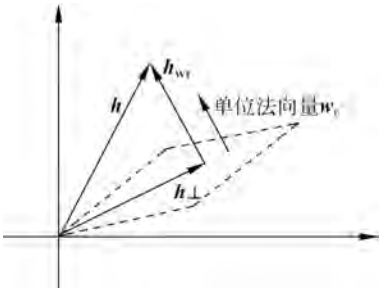


图 5-12　将向量投影到超平面

所以 $\boldsymbol{h}_{wr} = \boldsymbol{w}_r^T\boldsymbol{h}\boldsymbol{w}_r$，从图 5-14 中还可以看出 $\boldsymbol{h}_\perp = \boldsymbol{h} - \boldsymbol{h}_{wr}$。对于 $\boldsymbol{t}$ 向量的投影操作一样，所以最终 $\boldsymbol{h}$ 与 $\boldsymbol{t}$ 在超平面 $\boldsymbol{W}_r$ 上的投影 $\boldsymbol{h}_\perp$ 和 $\boldsymbol{t}_\perp$ 可表示为：

$$\begin{aligned} \boldsymbol{h}_\perp &= \boldsymbol{h} - \boldsymbol{w}_r^T\boldsymbol{h}\boldsymbol{w}_r \\ \boldsymbol{t}_\perp &= \boldsymbol{t} - \boldsymbol{w}_r^T\boldsymbol{t}\boldsymbol{w}_r \end{aligned} \tag{5-7}$$

损失函数其余的部分和 TransE 是一样的，公式如下[4]：

$$\text{hingeloss} = \max(0, \| \boldsymbol{h}_\perp + \boldsymbol{r} - \boldsymbol{t}_\perp \|_2 - \| \boldsymbol{h'}_\perp + \boldsymbol{r} - \boldsymbol{t'}_\perp \|_2 + m) \tag{5-8}$$

本节代码的地址为 recbyhand\chapter5\s22_TransH.py。

TransH 的代码仅仅需要在 TransE 的基础上略微改动，首先在__init__函数中多初始化一个法向量的 Embedding。对于每个关系都会有一个对应的超平面空间，所以法向量 Embedding 的数量是关系的数量，而将维度设置为和实体关系向量的长度一样即可，代码如下：

```
#recbyhand\chapter5\s22_TransH.py
#随机初始化法向量的 Embedding
self.wr = nn.Embedding( self.n_Relations, dim, max_norm = 1 )
```

将 max_norm 设定为 1 自然就将该法向量规范为单位法向量了。

然后定义一个 Htransfer()方法，进行式(5-7)的计算过程。传入的 e 和 wr 是一批次的实体 Embedding 与法向量 Embedding，代码如下：

```
#recbyhand\chapter5\s22_TransH.py
def Htransfer( self, e, wr ):
    return e - torch.sum( e * wr, dim = 1, keepdim = True ) * wr
```

此时对 predict()函数进行修改，即进行式(5-8)的计算过程，代码如下：

```
#recbyhand\chapter5\s22_TransH.py
def predict( self, x ):
    h, r_index, t = x
    h = self.e( h )
    r = self.r( r_index )
    t = self.e( t )
    wr = self.wr( r_index )
    score = self.Htransfer( h, wr ) + r - self.Htransfer( t, wr )
    return torch.sum( score ** 2, dim = 1 ) ** 0.5
```

代码其余的部分和 TransE 一模一样。

11min

5.2.3 翻译距离模型 TransR

TransR 并不是某个名字的简称，它的原论文名字是 Learning Entity and Relation Embeddings for Knowledge Graph Completion[5]，2015 年提出。其中 R 代表的是 Relation Space，即关系向量空间的意思。为什么会叫作 TransR？主要还是为了和 Translation Models 保持队形。可以认为 TransR 的意思是基于关系向量空间的知识图谱嵌入(Knowledge Graph Embedding by Translating on Relation Space)。

TransR 的示意图如图 5-13 所示。

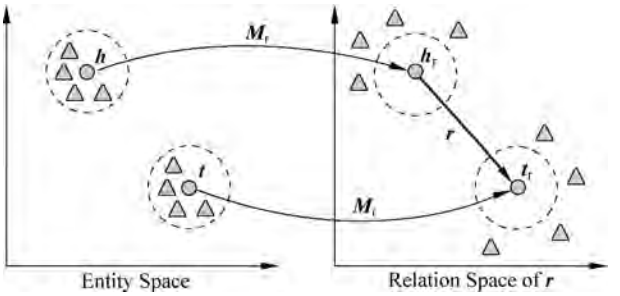


图 5-13 TransR 示意图[5]

Trans R 的中心思想是将 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 向量映射到 $\boldsymbol{r}$ 向量空间，然后在 $\boldsymbol{r}$ 向量空间进行 $\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}=0$ 的操作。可记作：

$$\| \boldsymbol{h}_r + \boldsymbol{r} - \boldsymbol{t}_r \|_2 = 0 \tag{5-9}$$

所谓的将 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 向量映射到 $\boldsymbol{r}$ 向量空间，其实是将 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 做一个线性变换，使它们的维度和 $\boldsymbol{r}$ 向量一样，所以这就要求原本的 $\boldsymbol{h}$、$\boldsymbol{t}$ 和 $\boldsymbol{r}$ 不在同一个向量空间。假设 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 的向量维度为 k，$\boldsymbol{r}$ 向量的维度为 d，则可将 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 点乘一个形状为 $k \times d$ 的矩阵，使其维度变为 d 的 $\boldsymbol{h}_r$ 和 $\boldsymbol{t}_r$ 向量。以上这些操作可由以下的数学语言表示：

$$\begin{aligned} &\boldsymbol{h}_r = \boldsymbol{h}\boldsymbol{M}_r \\ &\boldsymbol{t}_r = \boldsymbol{t}\boldsymbol{M}_r \\ &\boldsymbol{h}, \boldsymbol{t} \in \mathbf{R}^k, \quad \boldsymbol{r} \in \mathbf{R}^d, \quad \boldsymbol{M}_r \in \mathbf{R}^{k \times d}, \quad k \neq d \end{aligned} \tag{5-10}$$

损失函数其余的部分和 TransE 是一样的，公式如下[5]：

$$\text{hingeloss} = \max(0, \| \boldsymbol{h}_r + \boldsymbol{r} - \boldsymbol{t}_r \|_2 - \| \boldsymbol{h}'_r + \boldsymbol{r} - \boldsymbol{t}'_r \|_2 + m) \tag{5-11}$$

TransR 的映射操作与 TransH 的投影操作一样具备解决 TransE 在一对多等关系上效果不好的问题，而 TransR 相较 TransH 的优势就在于它假设了实体和关系是不同语义空间的向量。如果实体和关系在同一个语义空间，则训练起来会将语义相似的实体训练成在空间中很相近的向量，这等于没有把知识图谱嵌入的优势体现出来，而如果关系向量在不同的语义空间，就能更好地训练出同一个实体在不同关系中的差异。

当然 TransR 的劣势也很明显，虽然从公式上看 TransR 的操作过程比 TransH 更直接，但其实 TransR 的模型参数比 TransH 多很多。因为 TransH 中法向量的维度是 k，而 TransR 中矩阵的维度是 $k \times d$。

TransR 代码的地址为 recbyhand\chapter5\s23_TransR.py。仍然仅仅需要在 TransE 的基础上略微改动，首先在__init__()函数中多初始化一个矩阵 $\boldsymbol{M}_r$ 的 Embedding。对于每个关系都会有一个对应的线性变化矩阵，所以矩阵 Embedding 的数量是关系的数量，而将维度设置为 k_dim×r_dim，所以这次除了需要传入 $\boldsymbol{h}$ 和 $\boldsymbol{t}$ 的向量的维度超参 k_dim，还需要传入超参 r_dim 作为 $\boldsymbol{r}$ 向量的维度，具体的代码如下：

```
#recbyhand\chapter5\s23_TransR.py
class TransR( nn.Module ):
    def __init__( self, n_entitys, n_Relations, k_dim = 128, r_dim = 64 , margin = 1 ):
        super( ).__init__( )
        self.margin = margin                    #hinge_loss 中的差距
        self.n_entitys = n_entitys              #实体的数量
        self.n_Relations = n_Relations          #关系的数量
        self.k_dim = k_dim                      #实体 Embedding 的长度
        self.r_dim = r_dim                      #关系 Embedding 的长度
        #随机初始化实体的 Embedding
        self.e = nn.Embedding( self.n_entitys, k_dim, max_norm = 1 )
        #随机初始化关系的 Embedding
        self.r = nn.Embedding( self.n_Relations, r_dim, max_norm = 1 )
        #随机初始化变换矩阵
        self.Mr = nn.Embedding(self.n_Relations,k_dim * r_dim,max_norm = 1 )
```

然后定义一个 Rtransfer()函数，传入 $\boldsymbol{e}$（实体的向量），mr(r 对应的矩阵)。前两行是将它们的形状变为正确的形状，torch.matmul()方法在 PyTorch 框架中用于对三维张量的点乘操作。最后将 result 的形状变为(batch_size,r_dim)，r_dim 是 $\boldsymbol{r}$ 向量的长度。这就完成了将实体向量映射到 $\boldsymbol{r}$ 向量空间的操作，代码如下：

```
#recbyhand\chapter5\s23_TransR.py
def Rtransfer( self, e, mr ):
    #[ batch_size, 1, e_dim ]
    e = torch.unsqueeze( e, dim = 1 )
```

```
        #[ batch_size, e_dim, r_dim ]
        mr = mr.reshape( -1, self.k_dim, self.r_dim )
        #[ batch_size, 1, r_dim ]
        result = torch.matmul( e, mr )
        #[ batch_size, r_dim ]
        result = torch.squeeze( result )
        return result
```

随后将 predict()函数重写,即进行式(5-11)的计算过程,代码如下:

```
#recbyhand\chapter5\s23_TransR.py
def predict( self, x ):
    h, r_index, t = x
    h = self.e( h )
    r = self.r( r_index )
    t = self.e( t )
    mr = self.Mr( r_index )
    score = self.Rtransfer( h, mr ) + r - self.Rtransfer( t, mr )
    return torch.sum( score ** 2, dim = 1 ) ** 0.5
```

代码的其余部分和 TransE 一模一样。


7min

5.2.4 其他翻译距离模型

对于其他翻译距离模型包括 TransE 等接下来用一句话简单介绍一下。

TransE:用$-\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|$作为评分函数,采取负例采样,以正负例的差距作为损失函数,从而学出实体和关系的 Embedding。

TransH:将实体投影到超平面上再进行计算,从而解决 TransE 在多对多等关系上的缺陷。

TransR:与 TransH 不同的是,将实体投影到关系向量空间。

TransD:将 TransR 的映射矩阵分解为两个向量的积,可理解为简化了的 TransR。

TransSparse:在投影矩阵上强化稀疏性来简化 TransR。

TransM:以另一种途径优化 TransE 在一对多等关系中的缺陷,即对每个事实分配权重,一对多和多对多等事实会分配较小的权重,虽不如 TransH 和 TransR 那么彻底,但优势是模型训练性较好。

ManifoldE:通过把 $\boldsymbol{t}$ 近似位于流形上,即以超球体为中心在 $\boldsymbol{h}+\boldsymbol{r}$ 处,半径为 θ_r,而不是接近于 $\boldsymbol{h}+\boldsymbol{r}$ 的点,从而优化一对多等问题。

TransF:通过优化 $\boldsymbol{t}$ 与($\boldsymbol{h}+\boldsymbol{r}$)的点积相似度与 $\boldsymbol{h}$($\boldsymbol{t}-\boldsymbol{r}$)的点积相似度来训练 Embedding。

TransA:为每个关系 $\boldsymbol{r}$ 引入一个对称的非负矩阵 $\boldsymbol{M}_r$,并使用马氏距离定义评分函数。

KG2E:实体和关系使用高斯分布来表示。

TransG：实体采用高斯分布，它认为关系具有多重语义，所以采用混合高斯分布来表示。

UM(Unstructured model)非结构化模型：TransE 的极简版，直接将所有关系向量设置为 0。

SE(Structured Embedding)结构化嵌入：通过两个独立的矩阵为每个关系 $\boldsymbol{r}$ 对头尾实体进行投影。

表 5-1 是截取自参考文献[7]的对翻译距离模型总结得非常好的一张表格。其中第四列的评分方程中不少都加了负号。这是因为常识认为一个值往往越高代表它表现越好，这里也不例外，所以为了满足常识的需要在这个表达评分时，诸如 $\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|$ 这样明明越小越好的值前加一个负号就能表述成 $-\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|$ 越大越好了。

表 5-1　翻译距离模型总结[7]

Method	Ent.Embedding	Rel.Embedding	Scoring function $f_r(h,t)$	Constraints/Regularization
TransE[14]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$-\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|_{1/2}$	$\|\boldsymbol{h}\|_2=1,\|\boldsymbol{t}\|_2=1$
TransH[15]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r},\ \boldsymbol{w}_r\in\mathbf{R}^d$	$-\|(\boldsymbol{h}-\boldsymbol{w}_r^{\mathrm{T}}\boldsymbol{h}\boldsymbol{w}_r)+\boldsymbol{r}-(\boldsymbol{t}-\boldsymbol{w}_r^{\mathrm{T}}\boldsymbol{t}\boldsymbol{w}_r)\|_2^2$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1$
TransR[16]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^k,\ \boldsymbol{M}_r\in\mathbf{R}^{k\times d}$	$-\|\boldsymbol{M}_r\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{M}_r\boldsymbol{t}\|_2^2$	$\|\boldsymbol{w}_r^{\mathrm{T}}\boldsymbol{r}\|/\|\boldsymbol{r}\|_2\leqslant\varepsilon,\ \|\boldsymbol{w}_r\|_2=1$ $\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
TransD[50]	$\boldsymbol{h},\ \boldsymbol{w}_h\in\mathbf{R}^d$ $\boldsymbol{t},\ \boldsymbol{w}_t\in\mathbf{R}^d$	$\boldsymbol{r},\ \boldsymbol{w}_r\in\mathbf{R}^k$	$-\|(\boldsymbol{w}_r\boldsymbol{w}_h^{\mathrm{T}}+\boldsymbol{I})\boldsymbol{h}+\boldsymbol{r}-(\boldsymbol{w}_r\boldsymbol{w}_t^{\mathrm{T}}+\boldsymbol{I})\boldsymbol{t}\|_2^2$	$\|\boldsymbol{M}_r\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{M}_r\boldsymbol{t}\|_2\leqslant 1$ $\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$ $\|(\boldsymbol{w}_r\boldsymbol{w}_h^{\mathrm{T}}+\boldsymbol{I})\boldsymbol{h}\|_2\leqslant 1$ $\|(\boldsymbol{w}_r\boldsymbol{w}_t^{\mathrm{T}}+\boldsymbol{I})\boldsymbol{t}\|_2\leqslant 1$
TransSparse[51]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^k,\ \boldsymbol{M}_r(\theta_r)\in\mathbf{R}^{k\times d}$ $\boldsymbol{M}_r^1(\theta_r^1),\ \boldsymbol{M}_r^2(\theta_r^2)\in\mathbf{R}^{k\times d}$	$-\|\boldsymbol{M}_r(\theta_r)\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{M}_r(\theta_r)\boldsymbol{t}\|_{1/2}^2$ $-\|\boldsymbol{M}_r^1(\theta_r^1)\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{M}_r^2(\theta_r^2)\boldsymbol{t}\|_{1/2}^2$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$ $\|\boldsymbol{M}_r(\theta_r)\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{M}_r(\theta_r)\boldsymbol{t}\|_2\leqslant 1$ $\|\boldsymbol{M}_r^1(\theta_r^1)\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{M}_r^2(\theta_r^2)\boldsymbol{t}\|_2\leqslant 1$
TransM[52]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$-\theta_r\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|_{1/2}$	$\|\boldsymbol{h}\|_2=1,\|\boldsymbol{t}\|_2=1$
ManifoldE[53]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$-(\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|_2^2-\theta_r^2)^2$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
TransF[54]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$(\boldsymbol{h}+\boldsymbol{r})^{\mathrm{T}}\boldsymbol{t}+(\boldsymbol{t}-\boldsymbol{r})^{\mathrm{T}}\boldsymbol{h}$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
TransA[55]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d,\ \boldsymbol{M}_r\in\mathbf{R}^{d\times d}$	$-(\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|)^{\mathrm{T}}\boldsymbol{M}_r(\|\boldsymbol{h}+\boldsymbol{r}-\boldsymbol{t}\|)$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
KG2E[45]	$\boldsymbol{h}\sim\mathcal{N}(\boldsymbol{\mu}_h,\boldsymbol{\Sigma}_h)$ $\boldsymbol{t}\sim\mathcal{N}(\boldsymbol{\mu}_t,\boldsymbol{\Sigma}_t)$ $\boldsymbol{\mu}_h,\ \boldsymbol{\mu}_t\in\mathbf{R}^d$ $\boldsymbol{\Sigma}_h,\ \boldsymbol{\Sigma}_t\in\mathbf{R}^{d\times d}$	$\boldsymbol{r}\sim\mathcal{N}(\boldsymbol{\mu}_r,\boldsymbol{\Sigma}_r)$ $\boldsymbol{\mu}_r\in\mathbf{R}^d,\ \boldsymbol{\Sigma}_r\in\mathbf{R}^{d\times d}$	$-\mathrm{tr}(\boldsymbol{\Sigma}_r^{-1}(\boldsymbol{\Sigma}_h-\boldsymbol{\Sigma}_t))-\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\Sigma}_r^{-1}\boldsymbol{\mu}-\ln\frac{\det(\boldsymbol{\Sigma}_r)}{\det(\boldsymbol{\Sigma}_h-\boldsymbol{\Sigma}_t)}$ $-\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}-\ln(\det(\boldsymbol{\Sigma}))$ $\boldsymbol{\mu}=\boldsymbol{\mu}_h+\boldsymbol{\mu}_r-\boldsymbol{\mu}_t$ $\boldsymbol{\Sigma}=\boldsymbol{\Sigma}_h+\boldsymbol{\Sigma}_r-\boldsymbol{\Sigma}_t$	$\|\boldsymbol{M}_r\|_F\leqslant 1,[\boldsymbol{M}_r]_{ij}=[\boldsymbol{M}_r]_{ji}\geqslant 0$ $\|\boldsymbol{\mu}_h\|_2\leqslant 1,\|\boldsymbol{\mu}_t\|_2\leqslant 1,\|\boldsymbol{\mu}_r\|_2\leqslant 1$ $c_{\min}\boldsymbol{I}\leqslant\boldsymbol{\Sigma}_h\leqslant c_{\max}\boldsymbol{I}$ $c_{\min}\boldsymbol{I}\leqslant\boldsymbol{\Sigma}_t\leqslant c_{\max}\boldsymbol{I}$ $c_{\min}\boldsymbol{I}\leqslant\boldsymbol{\Sigma}_r\leqslant c_{\max}\boldsymbol{I}$
TransG[46]	$\boldsymbol{h}\sim\mathcal{N}(\mu_h,\sigma_h^2\boldsymbol{I})$ $\boldsymbol{t}\sim\mathcal{N}(\mu_t,\sigma_t^2\boldsymbol{I})$ $\mu_h,\ \mu_t\in\mathbf{R}^d$	$\mu_r^i\sim\mathcal{N}(\mu_t-\mu_h,\ (\sigma_h^2+\sigma_t^2)\boldsymbol{I})$ $\boldsymbol{r}=\sum_i\pi_r^i\mu_r^i\in\mathbf{R}^d$	$\sum_i\pi_r^i\exp\left(-\frac{\|\mu_h-\mu_t-\mu_r^i\|_2^2}{\sigma_h^2+\sigma_t^2}\right)$	$\|\mu_h\|_2\leqslant 1,\|\mu_t\|_2\leqslant 1,\|\mu_r^i\|_2\leqslant 1$
UM[56]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	—	$-\|\boldsymbol{h}-\boldsymbol{t}\|_2^2$	$\|\boldsymbol{h}\|_2=1,\|\boldsymbol{t}\|_2=1$
SE[57]	$\boldsymbol{h},\ \boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{M}_r^1,\ \boldsymbol{M}_r^2\in\mathbf{R}^{d\times d}$	$-\|\boldsymbol{M}_r^1\boldsymbol{h}-\boldsymbol{M}_r^2\boldsymbol{t}\|_1$	$\|\boldsymbol{h}\|_2=1,\|\boldsymbol{t}\|_2=1$

5.2.5　语义匹配模型 RESCAL

6min

RESCAL 的原论文名为 A Three-Way Model for Collective Learning on Multi-Relational Data[6]（基于多关系数据的集体学习三方模型），2011 年提出。所谓 Three-Way(三方)指的是 $\boldsymbol{h}$、$\boldsymbol{r}$ 和 $\boldsymbol{t}$。具体的评分公式如下：

$$\begin{aligned}&\mathrm{score}(\boldsymbol{h},\boldsymbol{r},\boldsymbol{t})=\boldsymbol{h}^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t}\\&\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d,\quad\boldsymbol{M}_r\in\mathbf{R}^{d\times d}\end{aligned}\tag{5-12}$$

Head 与 Tail 分别作为 $\boldsymbol{h}$ 向量和 $\boldsymbol{t}$ 向量，Relation 作为矩阵维度为 $d \times d$ 的矩阵 $\boldsymbol{M}_r$。对 $\boldsymbol{h}$、$\boldsymbol{t}$ 和 $\boldsymbol{M}_r$ 都做归一化处理，物理含义是 Head 的向量表示经过 Relation 空间的线性变换后与 Tail 的向量表示计算点积相似度。

与 TransE 等模型一样，也需要进行负例采样，$\boldsymbol{h}'$ 与 $\boldsymbol{t}'$ 代表负的 Head 与 Tail。最终的损失函数如下：

$$\text{Hingeloss} = \max(0, -\boldsymbol{h}^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t} + \boldsymbol{h}'^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t}' + \boldsymbol{m}) \tag{5-13}$$

再次提醒，虽然公式中同时存在 $\boldsymbol{h}'$ 与 $\boldsymbol{t}'$，但负例采样每次只需随机替换一个 $\boldsymbol{h}$ 或者 $\boldsymbol{t}$，公式这样写是为了避免写两个差不多的公式。

另外，在 RESCAL 及其他语义匹配模型中与绝大多数翻译距离模型不同的是，评分方程得到的值越大越好，并不是越小越好，所以在 RESCAL 损失计算过程中正例伴随的是一个负号，而负例伴随的反而是一个正号。

RESCAL 代码的地址为 recbyhand\chapter5\s25_RESCAL.py。具体的代码如下：

```
#recbyhand\chapter5\s25_RESCAL.py
class RESCAL( nn.Module ):

    def __init__( self, n_entitys, n_Relations, dim = 128, margin = 1 ):
        super( ).__init__( )
        self.margin = margin #hinge_loss 中的差距
        self.n_entitys = n_entitys #实体的数量
        self.n_Relations = n_Relations #关系的数量
        self.dim = dim #Embedding 的长度

        #随机初始化实体的 Embedding
        self.e = nn.Embedding( self.n_entitys, dim, max_norm = 1 )
        #随机初始化关系的 Embedding
        self.r = nn.Embedding( self.n_Relations, dim * dim, max_norm = 1 )

    def forward( self, X ):
        x_pos, x_neg = X
        y_pos = self.predict( x_pos )
        y_neg = self.predict( x_neg )
        return self.hinge_loss( y_pos, y_neg )

    def predict( self, x ):
        h, r, t = x
        h = self.e( h )
        r = self.r( r )
        t = self.e( t )
        #[ batch_size, dim, 1 ]
        t = torch.unsqueeze( t, dim = 2 )
        #[ batch_size, dim, dim ]
```

```
        r = r.reshape( -1, self.dim, self.dim )
        #[ batch_size, dim, 1 ]
        tr = torch.matmul( r, t )
        #[ batch_size, dim ]
        tr = torch.squeeze(tr)
        #[ batch_size ]
        score = torch.sum( h * tr, -1 )
        return -score

    def hinge_loss( self, y_pos, y_neg ):
        dis = y_pos - y_neg + self.margin
        return torch.sum( torch.ReLU( dis ) )
```

5.2.6　其他语义匹配模型

下面简单地用一句话介绍一下其他语义匹配模型，包括 RESCAL。

RESCAL：将头实体向量与关系矩阵点乘后与尾实体向量求点积相似度，用以评分函数。

DistMult：将关系矩阵简化为对角矩阵，优点是效率极高，但过于简化，只能处理对称关系，不能完全适用于所有场景。

HolE(Holographic Embeddings)：使用循环相关操作，将 RESCAL 的表达能力与 DistMult 的效率相结合。

ComplEx(Complex Embeddings)：在 DistMult 的基础上引入复数空间，非对称关系三元组中的实体或关系也能在复数空间得到分数。

ANALOGY：基于 RESCAL 的扩展，DistMult、HolE 和 ComplEx 都可归为 ANALOGY 的特例。

图 5-14 展示的是 4 个基于神经网络的语义匹配模型。

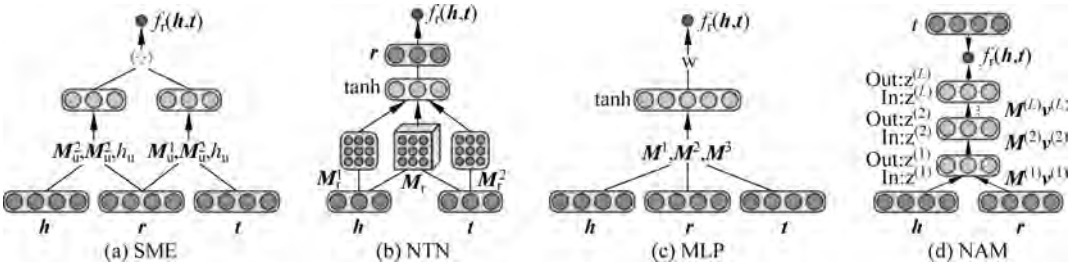


图 5-14　基于神经网络的语义匹配模型[7]

SME：语义匹配能量模型，先在输入层初始化 Head、Tail 和 Relation 的向量表示，然后关系向量与头尾实体向量分别在隐藏层组合，最后输出的是评分函数。

NTN：神经张量网络模型，在输入层初始化 Head 和 Tail 的向量，然后将头尾实体向量

在隐藏层与关系张量组合，最后输入评分函数。NTN是目前最具表达能力的模型之一，但是参数过多，效率较差。

MLP：多层感知机，是最基础的深度学习神经网络MLP。Head、Tail和Relation都在输入层初始化Embedding后经过几个隐藏层，最后输出评分函数。

NAM：神经关联模型，在输入层初始化Head和Relation的向量，经过一系列的隐藏层最终输出的结果与Tail向量建立损失函数。当然Tail实体有时也会作为Head实体出现在输入层，由此达到迭代学习的效果。

表5-2总结了语义匹配模型。

表5-2 语义匹配模型总结[7]

Method	Ent.Embedding	Rel.Embedding	Scoring function $f_r(\boldsymbol{h},\boldsymbol{t})$	Constraints/Regularization
RESCAL[13]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{M}_r\in\mathbf{R}^{d\times d}$	$\boldsymbol{h}^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t}$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{M}_r\|_F\leqslant 1$ $\boldsymbol{M}_r=\sum_i \boldsymbol{\pi}_r^i\boldsymbol{u}_i\boldsymbol{v}_i^{\mathrm{T}}$(required in[17])
TATEC[64]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d,\boldsymbol{M}_r\in\mathbf{R}^{d\times d}$	$\boldsymbol{h}^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t}+\boldsymbol{h}^{\mathrm{T}}\boldsymbol{r}+\boldsymbol{t}^{\mathrm{T}}\boldsymbol{r}+\boldsymbol{h}^{\mathrm{T}}\mathrm{Dt}$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$ $\|\boldsymbol{M}_r\|_F\leqslant 1$
DistMult[65]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$\boldsymbol{h}^{\mathrm{T}}\mathrm{diag}(\boldsymbol{r})\boldsymbol{t}$	$\|\boldsymbol{h}\|_2=1,\|\boldsymbol{t}\|_2=1,\|\boldsymbol{r}\|_2\leqslant 1$
HolE[62]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$\boldsymbol{r}^{\mathrm{T}}(\boldsymbol{h}*\boldsymbol{t})$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
ComplEx[66]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{C}^d$	$\boldsymbol{r}\in\mathbf{C}^d$	$\mathrm{Re}(\boldsymbol{h}^{\mathrm{T}}\mathrm{diag}(\boldsymbol{r})\bar{\boldsymbol{t}})$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
ANALOGY[68]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{M}_r\in\mathbf{R}^{d\times d}$	$\boldsymbol{h}^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t}$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{M}_r\|_F\leqslant 1$ $\boldsymbol{M}_r\boldsymbol{M}_r^{\mathrm{T}}=\boldsymbol{M}_r^{\mathrm{T}}\boldsymbol{M}_r$ $\boldsymbol{M}_r\boldsymbol{M}_{r'}=\boldsymbol{M}_{r'}\boldsymbol{M}_r$
SME[18]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$(\boldsymbol{M}_u^1\boldsymbol{h}+\boldsymbol{M}_v^2\boldsymbol{r}+\boldsymbol{b}_u)^{\mathrm{T}}(\boldsymbol{M}_v^1\boldsymbol{t}+\boldsymbol{M}_v^2\boldsymbol{r}+\boldsymbol{b}_v)$ $((\boldsymbol{M}_u^1\boldsymbol{h})\circ(\boldsymbol{M}_u^2\boldsymbol{r})+\boldsymbol{b}_u)^{\mathrm{T}}((\boldsymbol{M}_v^1\boldsymbol{t})\circ(\boldsymbol{M}_v^2\boldsymbol{r})+\boldsymbol{b}_v)$	$\|\boldsymbol{h}\|_2=1,\|\boldsymbol{t}\|_2=1$
NTN[19]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r},\boldsymbol{b}_r\in\mathbf{R}^k,\underline{\boldsymbol{M}}_r\in\mathbf{R}^{d\times d\times k}$ $\boldsymbol{M}_r^1,\boldsymbol{M}_r^2\in\mathbf{R}^{k\times d}$	$\boldsymbol{r}^{\mathrm{T}}\tanh(\boldsymbol{h}^{\mathrm{T}}\boldsymbol{M}_r\boldsymbol{t}+\boldsymbol{M}_r^1\boldsymbol{h}+\boldsymbol{M}_r^2\boldsymbol{t}+\boldsymbol{b}_r)$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$ $\|\boldsymbol{b}_r\|_2\leqslant 1,\|\underline{\boldsymbol{M}}_r^{[:,:,l]}\|_F\leqslant 1$ $\|\boldsymbol{M}_r^1\|_F\leqslant 1,\|\boldsymbol{M}_r^2\|_F\leqslant 1$
SLM[19]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^k,\boldsymbol{M}_r^1,\boldsymbol{M}_r^2\in\mathbf{R}^{k\times k}$	$\boldsymbol{r}^{\mathrm{T}}\tanh(\boldsymbol{M}_r^1\boldsymbol{h}+\boldsymbol{M}_r^2\boldsymbol{t})$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$ $\|\boldsymbol{M}_r^1\|_F\leqslant 1,\|\boldsymbol{M}_r^2\|_F\leqslant 1$
MLP[69]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$\boldsymbol{w}^{\mathrm{T}}\tanh(\boldsymbol{M}^1\boldsymbol{h}+\boldsymbol{M}^2\boldsymbol{r}+\boldsymbol{M}^3\boldsymbol{t})$	$\|\boldsymbol{h}\|_2\leqslant 1,\|\boldsymbol{t}\|_2\leqslant 1,\|\boldsymbol{r}\|_2\leqslant 1$
NAM[63]	$\boldsymbol{h},\boldsymbol{t}\in\mathbf{R}^d$	$\boldsymbol{r}\in\mathbf{R}^d$	$f_r(\boldsymbol{h},\boldsymbol{t})=\boldsymbol{t}^{\mathrm{T}}\boldsymbol{z}^{(L)}$ $\boldsymbol{z}^{(l)}=\mathrm{ReLU}(\boldsymbol{a}^{(l)}),\boldsymbol{a}^{(l)}=\boldsymbol{M}^{(l)}\boldsymbol{z}^{(l-1)}+\boldsymbol{b}^{(l)}$ $\boldsymbol{z}^{(0)}=[\boldsymbol{h};\boldsymbol{r}]$	—

5.3 基于知识图谱嵌入的推荐算法

了解完毕知识图谱嵌入后，接下来学习基于知识图谱嵌入的推荐算法。

8min

5.3.1 利用知识图谱嵌入做推荐模型的基本思路

首先来讲解最简单的思路，如图5-15所示。

图5-15中的左半部分，是一个ALS的结构。用于随机初始化用户与物品的隐向量，从而求内积后做出评分预测，与真实的评分建立损失函数。图5-17中的$S(\boldsymbol{u}\cdot\boldsymbol{v})$代表对$\boldsymbol{u}$和$\boldsymbol{v}$的内积做Sigmoid。

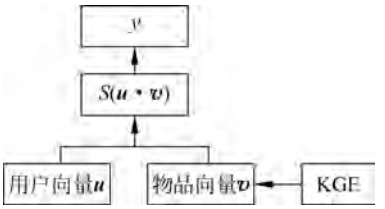


图 5-15　利用 KGE 的推荐算法模型的基本结构

右半部分代表用知识图谱的 Embedding 影响物品向量，因为物品在知识图谱中也是一个实体。具体怎么做大体上有如下 3 种方法。

方法 1：用 KGE 方法事先训练好所有的实体 Embedding，将实体中与用户发生交互的物品 Embedding 去初始化物品隐向量。

方法 2：用 KGE 方法事先训练好所有的实体 Embedding，将实体中与用户发生交互的物品 Embedding 初始化物品隐向量。且固定住物品隐向量不迭代更新，仅更新用户 Embedding 及其他模型参数。

方法 3：一边训练左半边的推荐模型，一边训练知识图谱数据中所有或部分实体，以及关系的 Embedding。

每种方法都有其优势，方法 1 的优势就在于简单直接，物品向量获得了知识图谱的信息，同时在训练迭代的过程中也会学得越来越精准。

方法 2 相较于方法 1 的优势就在于对冷启动的帮助显著，因为推荐模型所用的物品向量并不是和用户发生关系的交互数据所训练出的，而是知识图谱嵌入所得的向量。只要新物品在知识图谱中是一个实体，则直接可以用这个实体的向量作为这个物品的向量。当然缺点在于模型的收敛速率会降低甚至有些情况下学不出来有效的模型，原因就在于物品向量无法更新，仅仅更新的是用户向量，如果物品数量与用户数量的比例很失衡，则会比较难学。

方法 3 看似略微复杂，并且也不具备方法 2 可以解决冷启动问题的能力，但是经实战证明，方法 3 对于获取知识图谱信息这方面要优于方法 1 和方法 2。因为方法 3 当中物品向量同时被用户交互数据与知识图谱数据更新着，有着你中有我，我中有你互相影响的感觉。如果知识图谱的数据远多于用户交互数据，则只需取与用户交互数据中物品相关的知识图谱事实。

方法 3 具体的结构图如图 5-16 所示。

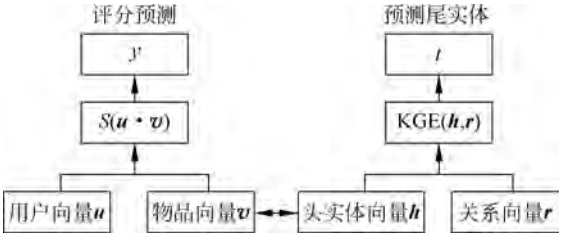


图 5-16　知识图谱嵌入与用户评分预测同时进行的示意图

图 5-16 中右半部分代表的是 KGE 的训练。⟷这样一个箭头表示的是物品向量 $\boldsymbol{v}$ 和头实体向量 $\boldsymbol{h}$ 在同一个向量空间。当然物品向量 $\boldsymbol{v}$ 与尾实体向量 $\boldsymbol{t}$ 也在同一空间,因为头尾实体的向量在同一个向量空间。

5.3.2 最简单的知识图谱推荐算法 CKE

协同基于知识嵌入的推荐系统(Collaborative Knowledge Base Embedding for Recommender Systems,CKE)[8],由微软大数据研究中心和电子科技大学在 2016 年提出。CKE 严格来讲不能说是一种算法,应该算是一种思想。C 代表 Collaborative 即协同,而 KE 代表 Knowledge Base Embedding,即基于知识的编码。知识包含知识图谱、文字信息、图像信息等。

这些信息分别的编码方式如下:

(1) 文字信息,通过自然语言处理的方式编码。

(2) 图像,通过计算机视觉的方式编码。

(3) 知识图谱的结构化信息,自然是通过知识图谱嵌入(KGE)的方式。

所以 CKE 最初本身囊括的东西很多,但是目前业内提到的 CKE 泛指用知识图谱嵌入的方式进行 Embedding 辅助推荐的算法。5.3.1 节中的 3 种方法均是从 CKE 的思想中提炼而来,所以它们都被称为 CKE 算法。

本节具体实现 5.3.1 节中 3 种方法中的第 3 种方法,即联合训练的 CKE。为什么不实现前两个算法呢? 因为那两个太简单了。

首先由于是推荐预测与知识图谱嵌入的联合训练,所以最终的损失函数会是推荐产生的损失函数与 KGE 产生的损失函数相加,公式如下:

$$\text{loss} = \text{rec}_{\text{loss}} + \alpha \cdot \text{kge}_{\text{loss}} \tag{5-14}$$

其中,rec_{loss} 是推荐预测产生的损失函数,目前采用最简单的 ALS 推荐算法,所以:

$$\begin{aligned} &\text{rec}_{\text{loss}} = \text{BCELoss}(\text{sigmoid}(\boldsymbol{u} \cdot \boldsymbol{v}), y^{\text{true}}) \\ &\boldsymbol{u}, \quad \boldsymbol{v} \in \mathbf{R}^k \end{aligned} \tag{5-15}$$

其中,$\boldsymbol{u}$ 是用户表示向量,$\boldsymbol{v}$ 是物品表示向量,y^{true} 是用户与物品真实的交互标注。

kge_{loss} 是知识图谱嵌入产生的损失函数,与所有联合训练一样可以设一个超参 α 来调整辅助损失函数的权重。

KGE 的方法有很多,在此仅用一个 kge_loss(*)函数来代替 KGE 方法所得的损失函数。公式如下:

$$\begin{aligned} &\text{kg}_{\text{loss}} = \text{kge\_loss}(\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t}) \\ &\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t} \in \mathbf{R}^k \end{aligned} \tag{5-16}$$

其中,$\boldsymbol{r}$ 是关系向量,$\boldsymbol{h}$ 和 $\boldsymbol{t}$ 分别是头实体和尾实体的向量,物品向量 $\boldsymbol{v}$ 在知识图谱中也是一个实体,所以 $\boldsymbol{v}$、$\boldsymbol{h}$ 和 $\boldsymbol{t}$ 都在同一个向量空间,当初始化 Embedding 时,仅需初始化一个 $\boldsymbol{e}$(Entity),即以实体的 Embedding 代替所有 $\boldsymbol{v}$、$\boldsymbol{h}$ 和 $\boldsymbol{t}$ 的 Embedding 即可,它们在数学上可表达为 $\boldsymbol{v} \cong \boldsymbol{h} \cong \boldsymbol{t} \cong \boldsymbol{e} \in \mathbf{R}^k$。

联合训练版 CKE 的示例代码的地址为 recbyhand\chapter5\s32_cke.py。具体的代码如下：

```
#recbyhand\chapter5\s32_cke.py
class CKE( nn.Module ):

    def __init__( self, n_users, n_entitys, n_Relations, e_dim = 128, margin = 1, alpha = 0.2 ):
        super( ).__init__( )
        self.margin = margin
        self.u_emb = nn.Embedding( n_users, e_dim )        #用户向量
        self.e_emb = nn.Embedding( n_entitys, e_dim )      #实体向量
        self.r_emb = nn.Embedding( n_Relations, e_dim )    #关系向量

        self.BCEloss = nn.BCELoss( )

        self.alpha = alpha                                 #kge 损失函数的计算权重

    def hinge_loss( self, y_pos, y_neg ):
        dis = y_pos - y_neg + self.margin
        return torch.sum( torch.ReLU( dis ) )

    #kge 采用最基础的 TransE 算法
    def kg_predict( self, x ):
        h, r, t = x
        h = self.e_emb( h )
        r = self.r_emb( r )
        t = self.e_emb( t )
        score = h + r - t
        return torch.sum( score ** 2, dim = 1 ) ** 0.5

    #计算 kge 损失函数
    def calculatingKgeLoss( self, kg_set ):
        x_pos, x_neg = kg_set
        y_pos = self.kg_predict( x_pos )
        y_neg = self.kg_predict( x_neg )
        return self.hinge_loss( y_pos, y_neg )

    #推荐采取最简单的 ALS 算法
        def rec_predict( self, u, i ):
        u = self.u_emb( u )
        i = self.e_emb( i )
        y = torch.sigmoid( torch.sum( u * i, dim = 1 ) )
        return y

    #计算推荐损失函数
```

```
    def calculatingRecLoss( self, rec_set ):
        u, i ,y = rec_set
        y_pred = self.rec_predict( u, i )
        y = torch.FloatTensor( y.detach().NumPy() )
        return self.BCEloss( y_pred, y )

    #前向传播
    def forward( self, rec_set, kg_set ):
        rec_loss = self.calculatingRecLoss( rec_set )
        kg_loss = self.calculatingKgeLoss( kg_set )
        #分别得到推荐产生的损失函数与kge产生的损失函数加权相加后返回
        return rec_loss + self.alpha * kg_loss
```

其中前向传播时会传入两个参数,分别叫作 rec_set 与 kg_set,rec_set 是批次采样得到的用户、物品、标注三元组数据,而 kg_set 是包含正负例的知识图谱三元组,与 5.2 节中学习 KGE 时的那个知识图谱三元组是一样的。

所以这是一种联合采样的操作,在使用 PyTorch 的 DataLoader 进行批次采样时,会用一个 zip 方法来同时采样 rec_set 与 kg_set,示例代码如下:

```
#recbyhand\chapter5\s32_cke.py
from torch.utils.data import DataLoader
#同时采样用户物品三元组及知识图谱三元组数据,因计算过程中互相独立,所以batch_size可设
#成不一样的值
for rec_set, kg_set in tqdm( zip( DataLoader( train_set, batch_size = rec_batchSize, shuffle =
True ),DataLoader( kgTrainSet, batch_size = kg_batchSize, shuffle = True ) ) ):
    optimizer.zero_grad( )
    loss = net( rec_set, kg_set )
```

值得一提的是,因为推荐预测与知识图谱嵌入用的是同一套 Embedding,而在后续计算过程中其实是相对独立的,所以在批次采样时可设置不一样的 batch_size。

5.3.3 CKE 扩展及演化

目前实现的 CKE 代码实际上处在过拟合状态,因为代码仅仅是最简单地实现一下模型结构而已,并没有加入更多的(例如 Drop Out 等)缓解过拟合的操作,抑或是加几个隐藏层来使模型更具备泛化能力,如图 5-17 所示。

大家也可以尝试其他不同的变化,例如想办法加入某种注意力机制等。也可以尝试用不同的 KGE 方法来试试效果。

其实目前实现的 CKE 是用 KGE 的方法影响 ALS 中物品的隐向量。且前文讲过 CKE 严格来讲不是算法,而是一种思想,这种思想是 CKE 的全称名字“协同基于知识嵌入的推荐系统”,所以大家可以基于此思想扩展及演化出各种算法。例如将用户作为一个实体融入知识图谱中,把用户与物品之间的交互当作知识图谱中用户实体与物品实体之间的关系。

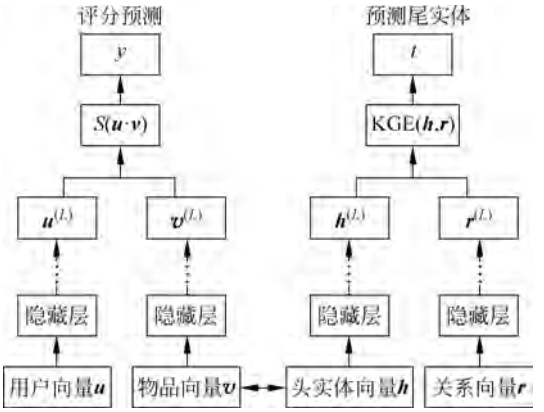


图 5-17　加入隐含层的 CKE

而在 2019 年，上海交通大学与微软亚洲研究中心推出的一个算法是在 CKE 思想的基础上的一种相当好的演化。

5.3.4　加强知识图谱信息的影响：MKR

28min

多任务特征学习知识图谱增强推荐（Multi-Task Feature Learning for Knowledge Graph Enhanced Recommendation，MKR）[9]由上海交通大学与微软亚洲研究中心在 2019 年提出。多任务是说推荐预测和 KGE 两个任务同时进行。

MKR 的模型结构如图 5-18 所示。

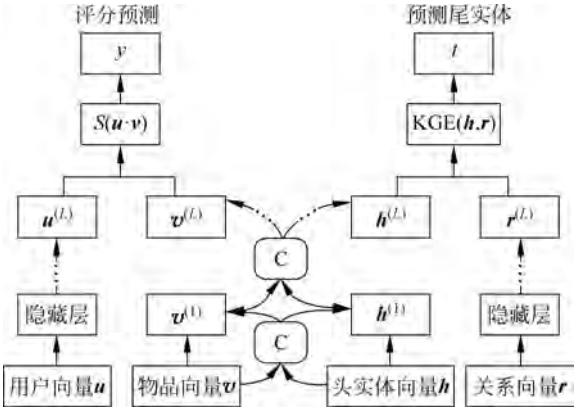


图 5-18　MKR 结构图

其实是在加入隐藏层的 CKE 的基础上，将物品向量和头实体向量用一个 C 单元去更新，而 C 单元的结构如图 5-19 所示。

这个 C 单元被称为 Cross&Compress 单元。意为交叉与压缩，交叉代表物品向量与实体向量全元素相乘。压缩是指将向量做一些映射操作。图 5-19 可理解为第 $L+1$ 层的物品

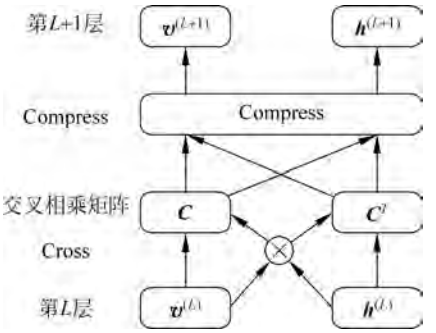


图 5-19　MKR 中的 Cross&Compress 单元

向量和实体向量是由第 L 层的物品向量和实体向量经过交叉与压缩的操作得到。具体的计算过程如下。

设第 L 层的物品向量 $\boldsymbol{v}_l=[v_l^1,v_l^2\cdots v_l^d]$。头实体向量 $\boldsymbol{h}_l=[h_l^1,h_l^2\cdots h_l^d]$，则 C 单元中第 L 层的 Cross($\boldsymbol{C}_l$) 的计算公式如下[9]：

$$\boldsymbol{C}_l=\boldsymbol{v}_l\boldsymbol{h}_l^{\mathrm{T}}=\begin{bmatrix} v_l^1h_l^1 & \cdots & v_l^1h_l^d \\ \vdots & & \vdots \\ v_l^dh_l^1 & \cdots & v_l^dh_l^d \end{bmatrix} \tag{5-17}$$

$\boldsymbol{C}_l$ 的维度是 $d\times d$，然后进行所谓的压缩操作，其实也是通过一个全连接层将输入向量维度重新恢复至 $d\times 1$。可以暂且用一个更简单的方程来表示，则压缩层，即 Compress 层的计算公式如下：

$$\begin{aligned}&\mathrm{Compress}(\boldsymbol{C}_l)=\sigma(\boldsymbol{C}_l\boldsymbol{w}_l+\boldsymbol{b}_l)\\&\boldsymbol{w}_l\in\mathbf{R}^d,\quad \boldsymbol{b}_l\in\mathbf{R}^d\end{aligned} \tag{5-18}$$

其中，$\sigma(\cdot)$表示任意激活函数，如 ReLU 和 Sigmoid 等。针对不同向量会有不同的权重和偏置项，下面用 $\boldsymbol{w}_l^v$ 和 $\boldsymbol{b}_l^v$ 表示第 L 层针对物品向量 $\boldsymbol{v}$ 的压缩单元权重及偏置项。$\boldsymbol{w}_l^h$ 和 $\boldsymbol{b}_l^h$ 表示第 L 层针对实体向量 $\boldsymbol{h}$ 的压缩单元权重及偏置项，则 C 单元第 $L+1$ 层的输出计算公式如下：

$$\begin{aligned}\boldsymbol{v}_{l+1}&=\sigma(\boldsymbol{C}_l\boldsymbol{w}_l^v+\boldsymbol{b}_l^v)\\\boldsymbol{h}_{l+1}&=\sigma(\boldsymbol{C}_l\boldsymbol{w}_l^h+\boldsymbol{b}_l^h)\end{aligned} \tag{5-19}$$

作者为了进一步增加知识图谱信息的影响，别出心裁地将 $\boldsymbol{C}_l$ 进行了一个转置，得到 $\boldsymbol{C}_l^T$，然后将 $\boldsymbol{C}_l^T$ 也定制了权重，这么一来水平方向和垂直方向的线性变换就都有了，所以总共有 4 个 $\boldsymbol{w}$ 和 2 个 $\boldsymbol{b}$。公式如下[9]：

$$\begin{aligned}\boldsymbol{v}_{l+1}&=\boldsymbol{C}_l\boldsymbol{w}_l^{vv}+\boldsymbol{C}_l^T\boldsymbol{w}_l^{hv}+\boldsymbol{b}_l^v\\\boldsymbol{h}_{l+1}&=\boldsymbol{C}_l\boldsymbol{w}_l^{vh}+\boldsymbol{C}_l^T\boldsymbol{w}_l^{hh}+\boldsymbol{b}_l^h\end{aligned} \tag{5-20}$$

可以用一个 $C(\cdot)$来代替式(5-17)到式(5-20)的过程，则

$$\boldsymbol{v}_L,\boldsymbol{h}_L=C^L(\boldsymbol{v},\boldsymbol{h}) \tag{5-21}$$

相比 C 单元，其余位置的迭代就容易多了。假设 $M(\cdot)$代表一个全连接层的函数(通常是 $y=\sigma(\boldsymbol{w}x+\boldsymbol{b})$)，模型的层数总共为 L 层，则用户向量 $\boldsymbol{u}$ 和尾实体向量 $\boldsymbol{t}$ 的迭代公式如下：

$$\begin{aligned}\boldsymbol{u}_L &= \boldsymbol{M}_u^L(\boldsymbol{u})\\ \boldsymbol{t}_L &= \boldsymbol{M}_t^L(\boldsymbol{t})\end{aligned} \tag{5-22}$$

注意：图 5-20 中虽然画的是关系向量 $\boldsymbol{r}$ 经隐藏往上迭代，但如果 KGE 的方法是翻译距离模型，则通常迭代的是尾实体向量 $\boldsymbol{t}$。

之后计算损失函数的公式组如下：

$$\begin{aligned}&\text{rec}_{\text{loss}} = \text{rec\_loss\_function}(\sigma(\boldsymbol{u}_L \cdot \boldsymbol{v}_L), y^{\text{true}})\\ &\text{kge}_{\text{loss}} = \text{kge\_loss\_function}(\boldsymbol{h}_L, \boldsymbol{r}, \boldsymbol{t}_L)\\ &\text{loss} = \text{rec}_{\text{loss}} + \alpha \cdot \text{kge}_{\text{loss}}\\ &\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t} \in \mathbf{R}^d\end{aligned} \tag{5-23}$$

其中，rec_{loss} 是评分预测损失函数，kge_{loss} 是知识图谱嵌入(KGE)损失函数，MKR 是一个评分预测与 KGE 的联合训练，而 α 是 KGE 损失函数的计算权重。

公式就写到这里，实际上在代码实现中会有很多技巧，下面就来带大家用最基础的代码实现。

MKR 在配套代码中的地址为 recbyhand\chapter5\s34_MKR.py。

先定义一个类，名称为 CrossCompress()，以此来作为 C 单元，代码如下：

```
#recbyhand\chapter5\s34_MKR.py
class CrossCompress( nn.Module ):

    def __init__( self, dim ):
        super( CrossCompress, self ).__init__()
        self.dim = dim

        self.weight_vv = init.xavier_uniform_( Parameter( torch.empty( dim, 1 ) ) )
        self.weight_ev = init.xavier_uniform_( Parameter( torch.empty( dim, 1 ) ) )
        self.weight_ve = init.xavier_uniform_( Parameter( torch.empty( dim, 1 ) ) )
        self.weight_ee = init.xavier_uniform_( Parameter( torch.empty( dim, 1 ) ) )

        self.bias_v = init.xavier_uniform_(Parameter(torch.empty(1, dim)))
        self.bias_e = init.xavier_uniform_(Parameter(torch.empty(1,dim)))

    def forward( self, v, e ):
        #[ batch_size, dim ]
        #[ batch_size, dim, 1 ]
        v = v.reshape( -1, self.dim, 1 )
        #[ batch_size, 1, dim ]
        e = e.reshape( -1, 1, self.dim )
        #[ batch_size, dim, dim ]
```

```
        c_matrix = torch.matmul( v, e )
        #[ batch_size, dim, dim ]
        c_matrix_transpose = torch.transpose( c_matrix, dim0 = 1, dim1 = 2 )
        #[ batch_size * dim, dim ]
        c_matrix = c_matrix.reshape( ( -1, self.dim ) )
        c_matrix_transpose = c_matrix_transpose.reshape( ( -1,self.dim ))
        #[batch_size, dim]
        v_output = torch.matmul( c_matrix, self.weight_vv ) + torch.matmul( c_matrix_
transpose, self.weight_ev )
        e_output = torch.matmul( c_matrix, self.weight_ve ) + torch.matmul( c_matrix_
transpose, self.weight_ee )
        #[batch_size, dim]
        v_output = v_output.reshape( -1, self.dim ) + self.bias_v
        e_output = e_output.reshape( -1, self.dim ) + self.bias_e
        return v_output, e_output
```

然后定义 MKR 的主类,它的初始化方法如下,为了更好地让大家理解算法的原理,示例代码尽可能会写得直接一点。

```
#recbyhand\chapter5\s34_MKR.py
class MKR( nn.Module ):

    def __init__( self, n_users, n_entitys, n_Relations, dim = 128, margin = 1, alpha = 0.2,
DropOut_prob = 0.5):
        super( ).__init__( )
        self.margin = margin
        self.u_emb = nn.Embedding( n_users, dim )        #用户向量
        self.e_emb = nn.Embedding( n_entitys, dim )      #实体向量
        self.r_emb = nn.Embedding( n_Relations, dim )    #关系向量

        self.user_dense1 = DenseLayer( dim, dim, DropOut_prob )
        self.user_dense2 = DenseLayer( dim, dim, DropOut_prob )
        self.user_dense3 = DenseLayer( dim, dim, DropOut_prob )
        self.Tail_dense1 = DenseLayer( dim, dim, DropOut_prob )
        self.Tail_dense2 = DenseLayer( dim, dim, DropOut_prob )
        self.Tail_dense3 = DenseLayer( dim, dim, DropOut_prob )
        self.cc_unit1 = CrossCompress( dim )
        self.cc_unit2 = CrossCompress( dim )
        self.cc_unit3 = CrossCompress( dim )

        self.BCEloss = nn.BCELoss( )

        self.alpha = alpha #kge 损失函数的计算权重
```

其中除了三层 C 单元外,还包含了三层用户全连接层与三层尾实体全连接层,全连接

层的具体结构如下：

```
#recbyhand\chapter5\s34_MKR.py
#附加 DropOut 的全连接网络层
class DenseLayer( nn.Module ):

    def __init__( self, in_dim, out_dim, DropOut_prob ):
        super( DenseLayer, self ).__init__( )
        self.liner = nn.Linear( in_dim, out_dim )
        self.drop = nn.DropOut( DropOut_prob )

    def forward( self, x, isTrain ):
        out = torch.ReLU( self.liner( x ) )
        if isTrain: #训练时加入 DropOut,防止过拟合
        out = self.drop( out )
        return out
```

加入 DropOut 主要是为了防止过拟合，像 MKR 这种模型参数众多且属于联合训练的神经网络，数据量少时极容易过拟合，所以一些防止过拟合的技巧还是很需要的。

为了突出 MKR 的模型结构，这次 KGE 方法仅采用最简单的 TransE，TransE 部分的代码如下：

```
#recbyhand\chapter5\s34_MKR.py
def hinge_loss( self, y_pos, y_neg ):
    dis = y_pos - y_neg + self.margin
    return torch.sum( torch.ReLU( dis ) )

#kge 采用最基础的 TransE 算法
def TransE( self, h, r, t ):
    score = h + r - t
    return torch.sum( score ** 2, dim = 1 ) ** 0.5
```

重点的前向传播方法的代码如下：

```
#recbyhand\chapter5\s34_MKR.py
#前向传播
def forward( self, rec_set, kg_set, isTrain = True ):
    #推荐预测部分的提取,初始 Embedding
    u, v ,y = rec_set
    y = torch.FloatTensor( y.detach().NumPy() )
    u = self.u_emb( u )
    v = self.e_emb( v )

    #分开知识图谱三元组的正负例
```

```
        x_pos, x_neg = kg_set

        #提取知识图谱三元组正例 h、r 和 t 的初始 Embedding
        h_pos, r_pos, t_pos = x_pos
        h_pos = self.e_emb( h_pos )
        r_pos = self.r_emb( r_pos )
        t_pos = self.e_emb( t_pos )

        #提取知识图谱三元组负例 h、r 和 t 的初始 Embedding
        h_neg, r_neg, t_neg = x_neg
        h_neg = self.e_emb( h_neg )
        r_neg = self.r_emb( r_neg )
        t_neg = self.e_emb( t_neg )

        #将用户向量经三层全连接层传递
        u = self.user_dense1( u, isTrain )
        u = self.user_dense2( u, isTrain )
        u = self.user_dense3( u, isTrain )
        #将 KG 正例的尾实体向量经三层全连接层传递
        t_pos = self.Tail_dense1( t_pos, isTrain )
        t_pos = self.Tail_dense2( t_pos, isTrain )
        t_pos = self.Tail_dense3( t_pos, isTrain )

        #将物品与 KG 正例头实体一同经三层 C 单元传递
        v, h_pos = self.cc_unit1( v, h_pos )
        v, h_pos = self.cc_unit2( v, h_pos )
        v, h_pos = self.cc_unit3( v, h_pos )

        #计算推荐预测的预测值及损失函数
        rec_pred = torch.sigmoid( torch.sum( u * v, dim = 1 ) )
        rec_loss = self.BCEloss( rec_pred, y )

        #计算 KG 正例的 TransE 评分
        kg_pos = self.TransE( h_pos, r_pos, t_pos )
        #计算 KG 负例的 TransE 评分,注意负例的实体不要与物品向量一同经 C 单元
        kg_neg = self.TransE( h_neg, r_neg, t_neg )
        #计算 KGE 的 hing loss
        kge_loss = self.hinge_loss( kg_pos, kg_neg )

        #将推荐产生的损失函数与 KGE 产生的损失函数加权相加后返回
    return rec_loss + self.alpha * kge_loss
```

相信这个完全按顺序结构编写的代码的可读性应该是不错的,并且注释也很详细,这里就不做过多的解释了。另外要讲的是 MKR 在预测时有一点麻烦,因为 MKR 中的 C 单元同时输入物品向量和那一批次的知识图谱头实体向量迭代更新。假设要预测一个用户与一

个给定物品间的评分，则也得寻找一个头实体一同输入模型中才能有效预测出结果。解决方案是用该物品同时作为头实体输进去，所以如果需要预测，需要在 MKR 的类中再定义一个预测的方法，代码如下：

```
#recbyhand\chapter5\s34_MKR.py
#测试时用
def predict( self, u, v, isTrain = False ):
    u = self.u_emb( u )
    v = self.e_emb( v )
    u = self.user_dense1( u, isTrain )
    u = self.user_dense2( u, isTrain )
    u = self.user_dense3( u, isTrain )
    #第一层输入 C 单元的 KG 头实体，即物品自身
    v, h = self.cc_unit1( v, v )
    v, h = self.cc_unit2( v, h )
    v, h = self.cc_unit3( v, h )
    return torch.sigmoid( torch.sum( u * v, dim = 1 ) )
```

最后还有值得一提的是在同时采样知识图谱数据和推荐三元组数据时 batch_size 必须一致，因为 C 单元中物品与头实体的计算过程是相互干涉的，所以要求张量维度一致。

代码的其余部分是些常规操作，大家可自行查看配套代码。

5.3.5　MKR 扩展

8min

实际工作中的调参自然会把原型变得五花八门，这里介绍一个针对 MKR 比较有效且也比较实用的扩展，如图 5-20 所示。

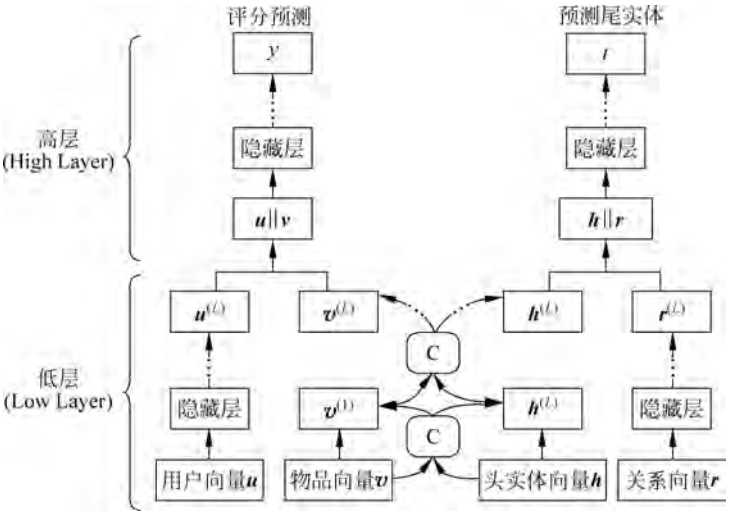


图 5-20　MKR 高低层

将模型结构分为高低两层,低层与原型一样具有若干个(可自由设置个数)的隐藏层,即全连接层及C单元,不同的是引入了高层的概念。

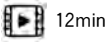
先看左半边,高层中的第一层是将低层输出的用户向量和物品向量拼接(‖是向量拼接的符号)起来,然后又经一个多层感知机的网络,当然层数可自由设置,这里最后一个全连接层输入的维度可以是1,也是直接取那个值与真实的评分建立损失函数。当然这么做的好处无非是进一步提高模型的泛化能力。

右半边的知识图谱如果感觉翻译距离系列的算法不太方便,则可以采取语义匹配模型的算法思想。例如将低层的头实体向量与关系向量都输出到高层,然后将它们拼接起来,同样经一套多层感知机的网络,而此时最后一个全连接层的输出向量应该等于尾实体向量,然后将最后一层的输出向量作为尾实体的预测归一化后与真实的尾实体求单击相似度,从而建立损失函数。

当然并不存在什么真实的尾实体向量,所谓真实的尾实体向量也许是前几轮迭代过的头实体向量或者从没迭代过的实体向量,所以为了使模型的效果更好,务必注意知识图谱不可以被做成有向无环图,或者有向无环的节点要尽可能少,因为这种节点如果只作为尾实体,则无法迭代更新。

另外还要强调的是,不管采取什么知识图谱嵌入的方法,都要采取负例采样。这是新手甚至有的老手也常会忽略的事情。其实左半边的用户物品评分预测也有负例采样,在训练集中一定有某个用户与某个物品的评分是0,如果全部都是正例,则该模型预测所有用户物品评分时都会输出1。知识图谱嵌入训练也是一样的道理,如果全部都是正例,则最后预测实体向量与真实实体向量的点积相似度就只会为1,而用户物品交互数据集中天生具备着负例样本,所以大家不太需要去手动地人造负例样本,但知识图谱训练集中并不存在负例,所以必须人造一些负例样本,当遇到负例时,让它们的点积相似度为0,这样模型才可以训练起来。

5.3.6 针对更新频率很快的新闻场景知识图谱推荐算法:DKN



如果被推荐物品来不及建立知识图谱,例如实时更新速度很快的新闻那该怎么利用知识图谱信息呢?

虽然这样的物品无法及时建立知识图谱,但是新闻所包含的内容可以事先建立知识图谱,所以处理的思路是先对新闻内容或标题进行实体识别,然后将这些实体用KGE的方式基于事先建立好的实体知识图谱学出向量表示(Embedding),将这些Embedding做拼接等操作代替被推荐新闻的Embedding进行后道计算,图5-21展现了这段话的思路。

上海交通大学与微软亚洲研究中心在2018年针对新闻场景提出了专门的推荐算法,DKN:Deep Knowledge-Aware Network for News Recommendation[10],其网络结构如图5-22所示。

每条新闻都用一个名为KCNN的单元做Embedding处理,KCNN的全称为Knowledge-aware CNN。一个单层的KCNN是将新闻内容中包含的每个实体的向量拼接

成实体数量×向量维度形状的矩阵之后，进行卷积和池化得到一个向量的操作，如图 5-23 所示。

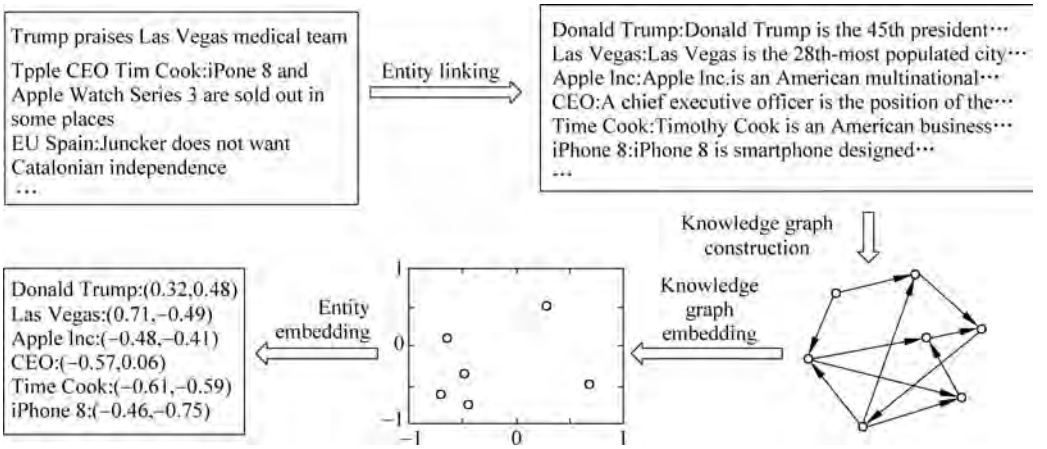


图 5-21　提取新闻标题关键字特征向量[10]

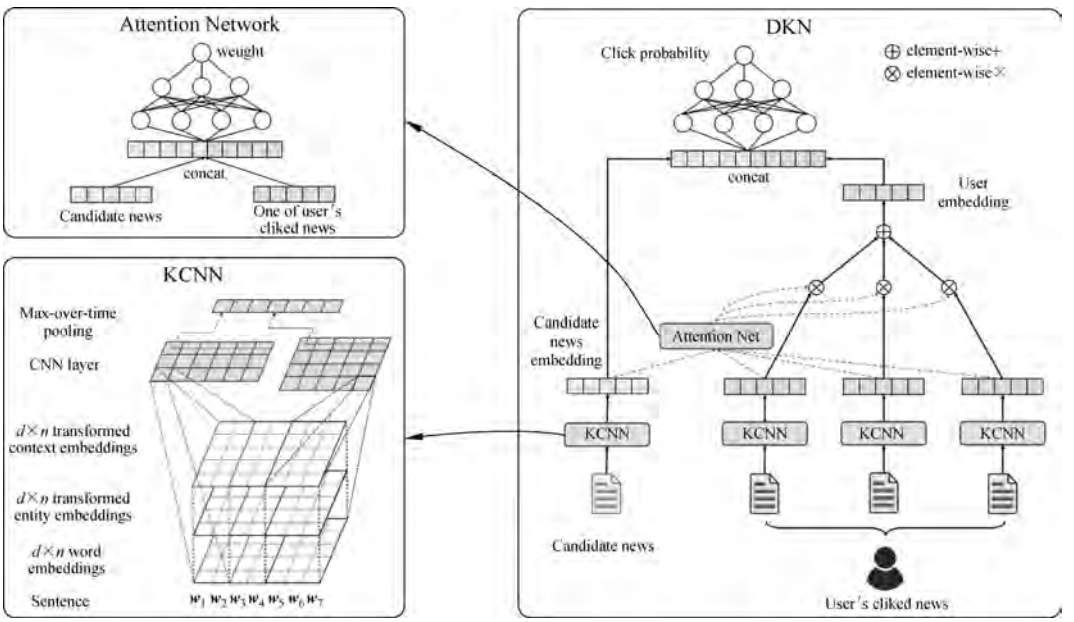


图 5-22　DKN 网络结构[10]

代表实体的 Embedding 可以用不同的 KGE 方式计算组成多通道的三维张量再进行卷积，就像图 5-22 的左下模块所示。且不仅是知识图谱嵌入得到的 Embedding，用其他手段，例如基础 Graph Embedding 甚至是自然语言处理（NLP）Embedding 的方式得到的多种 Embedding 都可以拼接成多通道的张量，然后进行 CNN 的消息传递。

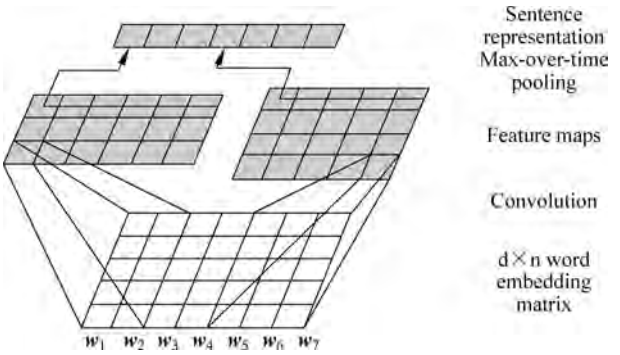


图 5-23 单层 KCNN 示意图[10]

所以通过 KCNN 的方式可以聚合出目标物品,即那个新闻的 Embedding,用户 Embedding 是由用户单击过的新闻经过 KCNN 处理后加权求和得到。权重是单击历史新闻各自与目标新闻经注意力机制计算得到的注意力。

最后目标新闻 Embedding 与用户 Embedding 拼接后经过几轮全连接层最终得到该用户对该新闻的单击预测,这是全部的 DKN 算法的过程。

5.4 基于知识图谱路径的推荐算法

路径是图论的概念,所以基于知识图谱路径的推荐算法是将知识图谱当作图来对待了,但是知识图谱不是简单的图,而是属于比较复杂的异构图。异构图指节点类型+边类型>2的图,所谓节点类型在知识图谱中是实体的类型,例如用户实体、电影实体、演员实体等。边类型指关系的类型,例如"用户—电影"关系,"演员 — 电影"关系。这些类型的数量自然非常繁多。

因为有着各种类型的节点及边,也就意味着连接着不同类型节点和边的路径似乎也可以当作所谓不同类型的路径来对待,所以在学习基于知识图谱推荐算法之前,得先了解一个异构图的基础概念,即元路径。

8min

5.4.1 元路径

元路径(Meta-Path)[12]可以理解为连接不同类型节点的一条路径,不同的元路径会有不同的路径类型,而所谓路径类型通常是用节点类型路径来表示的。节点类型路径是指由节点类型作为节点的普通同构图路径。例如"用户→电影→演员"是指连接用户节点、电影节点和演员节点的元路径类型。

下面结合图 5-24 来说明源路径。

如图 5-24 所示,假如要给用户推荐电影,则元路径类型可以分为以下几种。

电影→题材→电影：给用户推荐同题材的电影。

电影→导演→电影：给用户推荐同导演的电影。

电影→演员→电影：给用户推荐同演员的电影。

电影→角色→电影：给用户推荐同角色的电影，例如两部电影中都有孙悟空这样的角色。

电影→用户→电影：给用户推荐相似用户看过的电影，相当于协同过滤。

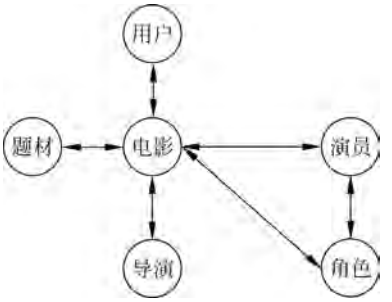


图 5-24　元路径示意图

甚至还可以包括以下这些更长的路径。

电影→角色→演员→电影：例如该用户看过的电影《大话西游》中有个角色是孙悟空，演过该角色的演员还有六小龄童，于是给用户推荐六小龄童出演的《财迷》电影。

电影→演员→角色→演员→电影：例如该用户看过的电影《大话西游》中有个演员是周星驰，周星驰还演过角色唐伯虎，演过唐伯虎的还有演员黄晓明，于是给用户推荐黄晓明主演的电影《风声》。

当然元路径越长推荐影响的比重一定会越低，有的模型可以学习出用户更偏爱哪条元路径的推荐，但不会设置较长的元路径去影响模型的效率，一般情况下，选择长度为 3 的对称元路径即可。

对称元路径指的是例如电影→题材→电影和电影→演员→角色→演员→电影这种按中间节点对称的路径。非对称情况则例如电影→演员→角色和电影→角色→演员。

5.4.2　路径相似度（PathSim）


27min

在对称元路径的前提下，可以求出头尾节点的路径相似度（PathSim）。该任务可被描述为例如求《功夫》和《功夫熊猫》在元路径“电影→演员→电影”下的路径相似度。

路径相似度的公式如下[13]：

$$s(x,y)=\frac{2\times|\{p_{x\to y}:p_{x\to y}\in P\}|}{|\{p_{x\to x}:p_{x\to x}\in P\}|+|\{p_{y\to y}:p_{y\to y}\in P\}|} \tag{5-24}$$

其中，$s(x,y)$指节点 x 与节点 y 之间的路径相似度。分子上的$|\{p_{x\to y}:p_{x\to y}\in P\}|$代表在元路径 P 的情况下，节点 x 与节点 y 之间的路径数量，所以也就是说节点间路径数量越多，代表它们越相似，而分母上的那两项，是节点 x 和节点 y 在元路径 P 的情况下回到自身的总路径数量，将这两项放在分母的位置相当于是在做归一约束。

以元路径“导演→电影→题材→电影→导演”来举个例子，目标是求取导演间的相似度。假设有导演张三、李四、王五、赵六。题材有悬疑、喜剧、爱情。电影是什么不重要，只需记录导演在某个题材上拍过的电影数量，假设每个导演与每个题材间的电影数量如表 5-3 所示。

表 5-3 路径相似度说明表

导 演	悬 疑	喜 剧	爱 情
张三	5	2	0
李四	2	3	5
王五	0	0	10
赵六	5	2	0

可以将表格中的数据做成如图 5-25 所示的图谱。

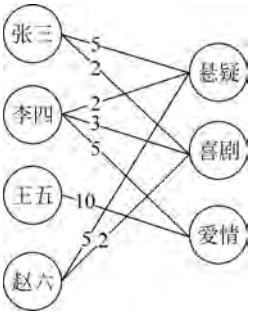


图 5-25 路径相似度图例

由图 5-25 可以很方便地看到,张三有 5 条路径通往悬疑,李四有 2 条路径通往悬疑,所以张三和李四之间的有关悬疑的总路径数是 5×2= 10 条。以此类推,张三与李四有关喜剧的路径数有 2×3=6 条,有关爱情的路径数为 0×5=0 条,则分子上的 $|\{p_{x\to y}:p_{x\to y}\in P\}|$ 这一栏是 10+6+0=16。

张三回到张三自身的路径数共 5×5+2×2=29 条。李四同理。把所有的计算写下来,张三和李四间的相似度是:

$$s(\text{张三},\text{李四})=\frac{2\times(5\times2+2\times3+0\times5)}{(5\times5+2\times2)+(2\times2+3\times3+5\times5)}\approx0.478$$

代入公式可以将所有导演间的两两相似度求出来。

如果这样一个一个算嫌麻烦,则可将张三看作向量[5,2,0]。将李四看作向量[2,3,5],则 $p_{\text{张三}\to\text{李四}}$ 就是两个向量间的点乘。$p_{\text{张三}\to\text{张三}}$ 也可以视为是张三向量自己与自己的点乘。

所以导演和题材之间的共现表格可以看作一个矩阵。记作:

$$\boldsymbol{M}=\begin{bmatrix}5&2&0\\2&3&5\\0&0&10\\5&2&0\end{bmatrix}$$

用这个矩阵点乘它的转置可以得到一个交换矩阵(Commuting Matrix)记作:

$$\mathbf{CM}=\boldsymbol{M}\cdot\boldsymbol{M}^{\mathrm{T}}=\begin{bmatrix}5&2&0\\2&3&5\\0&0&10\\5&2&0\end{bmatrix}\cdot\begin{bmatrix}5&2&0&5\\2&3&0&2\\0&5&10&0\end{bmatrix}=\begin{bmatrix}29&16&0&29\\16&38&50&16\\0&50&100&0\\29&16&0&29\end{bmatrix}\tag{5-25}$$

有了这个交换矩阵 **CM**,计算路径相似度就会变得相当方便,实体 x 和实体 y 之间的路径相似度的公式可以用另一种形式表达,公式如下:

$$s(x,y)=\frac{2\times\mathbf{CM}_{xy}}{\mathbf{CM}_{xx}+\mathbf{CM}_{yy}}\tag{5-26}$$

本节代码的地址为 recbyhand\chapter5\s42_pathSim.py

利用交换矩阵得到两个实体 e1 和 e2 之间的相似度的代码如下:

```
#recbyhand\chapter5\s42_pathSim.py
import numpy as np

M = np.mat([[5,2,0],
          [2,3,5],
          [0,0,10],
          [5,2,0]])
#根据共现矩阵求两个实体间的路径相似度
def getPathSimFromCoMatrix(e1,e2,M):
    CM = np.array(M.dot(M.T))#得到交换矩阵
    return 2 * CM[e1][e2]/(CM[e1][e1] + CM[e2][e2])
```

若需要节省内存而不需要使用交换矩阵的计算形式，则代码如下：

```
#recbyhand\chapter5\s42_pathSim.py
#根据点乘的方法求两个实体间的路径相似度
def getPathSimFromMatrix(e1,e2,M):
    up = 2 * M[e1].dot(M[e2].T)
    down = M[e1].dot(M[e1].T) + M[e2].dot(M[e2].T)
    return float(up/down)
```

得到所有实体间的两两相似度矩阵的代码如下：

```
#recbyhand\chapter5\s42_pathSim.py
#根据共现矩阵得到所有实体的相似度矩阵
def getSimMatrixFromCoMatrix(M):
    CM = M.dot(M.T)
    a = np.diagonal(CM)
    nm = np.array([a + i for i in a])
    return 2 * CM/nm
```

当然如果硬件条件不好，别说交换矩阵了，共现矩阵也没有办法加载进内存里，则只能先得到邻接表（共现矩阵也可视为邻接矩阵），假设有三元组数据如下：

```
triples =[[0,5,0],
       [0,2,1],
       [1,2,0],
       [1,3,1],
       [1,5,2],
       [2,10,2],
       [3,5,0],
       [3,2,1]]
```

则得到邻接表的代码如下：

```
#recbyhand\chapter5\s42_pathSim.py
import collections
#根据三元组得到邻接表
def getAdjacencyListByTriples( triples ):
    al = collections.defaultdict( dict )
    for h,r,t in triples:
        al[h][t] = r
    return al
```

邻接表如下：

```
{0: {0: 5, 1: 2}, 1: {0: 2, 1: 3, 2: 5}, 2: {2: 10}, 3: {0: 5, 1: 2}}
```

此邻接表是{实体 id,{实体 id,关系权重}}这样的形式。
通过邻接表得到两个实体间路径相似度的代码如下：

```
#recbyhand\chapter5\s42_pathSim.py
#得到自元路径数量
def getSelfMetaPathCount(e,al):
    return sum(al[e][i] ** 2 for i in al[e])

#得到两个实体间的元路径数
def getMetaPathCountBetween(e1,e2,al):
    return sum([al[e1][i] * al[e2][i] for i in set(al[e1]) & set(al[e2])])

#求两个实体间的路径相似度
def getPathSimFromAl(e1,e2,al):
    up = getMetaPathCountBetween(e1,e2,al)
    s1 = getSelfMetaPathCount(e1,al)
    s2 = getSelfMetaPathCount(e2,al)
    down = s1 + s2
    return 2 * up/down
```

通过邻接表得到所有实体路径相似度矩阵的代码如下：

```
#recbyhand\chapter5\s42_pathSim.py
#根据邻接表求所有实体间的路径相似度
def getSimMatrixFromAl(al,n_e):
    selfMPC = {}
    for e in al:
        selfMPC[e] = getSelfMetaPathCount(e,al)
    simMatrix = np.zeros((n_e,n_e))
    for e1 in al:
        for e2 in al:
```

```
            simMatrix[e1][e2] = 2 * getMetaPathCountBetween(e1,e2,al)\
                                    /(selfMPC[e1] + selfMPC[e2])
        return simMatrix
```

5.4.3　学习元路径的权重：PER

PER 全称为 Personalized Entity Recommendation[14]，又名 HeteRec，是最基础的基于元路径的推荐方式，提出于 2014 年。

首先作者定义了一个名为用户偏好扩散(User Preference Diffusion)的概念，设推荐基于元路径的格式为用户→物品→*→物品，前半部分的用户→物品是用户与物品的历史交互数据，可被理解为用户偏好。后半部分物品→*→物品表示物品与其他物品间的路径相似度，可被理解为用户交互过的物品在知识图谱中沿着某条元路径的偏好扩散。

假设要得到用户 u，与物品 v 基于某条元路径 P 情况下的偏好扩散分数，则基础的公式如下[14]：

$$s(u,v \mid P)=\sum_{i}^{I|u} r(u,v_i)\times \mathrm{PathSim}(v_i,v) \tag{5-27}$$

其中，$\mathrm{PathSim}(v_i,v)$是物品 v_i 和 v 之间的路径相似度。$I|u$ 表示用户 u 交互过的数据。$r(u,v_i)$表示用户与该物品的交互情况，如果能判断为喜爱，则 $r(u,v_i)=1$，如果不喜爱，则 $r(u,v_i)=0$。

例如在元路径为用户→电影→演员→电影的情况下，如图 5-26 所示。

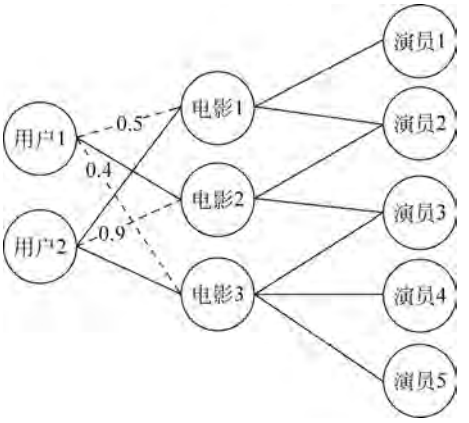


图 5-26　用户→电影→演员元路径示意图

图 5-26 中的虚线代表用户与电影间的偏好扩散预测。例如在预测用户 1 与电影 1 的偏好扩散分数时，可以观察到在三部电影中，用户 1 只有与电影 2 有正向交互，而电影 2 与电影 1 通过演员 2 互通。电影 1 通过演员有 2 条自回路径，电影 2 也有 2 条自回路径，所以电影 1 与电影 2 的路径相似度是：

$$\text{PathSim}(\text{电影 1},\text{电影 2})=\frac{2\times 1}{2+2}=0.5$$

再将用户与电影的交互数据代入式(5-27),最终可得用户 1 与电影 1 基于元路径用户→电影→演员→电影的偏好扩散分数等于 0.5。

由于已经有方式得到某一条元路径下的用户偏好分数,所以之后可用标注数据将用户针对某条元路径的偏好权重学出来,公式如下[14]:

$$r^{\text{pred}}(u,v)=\sum_{P}^{L}w_P s(u,v\mid P) \tag{5-28}$$

其中 L 代表所有元路径,w_P 是元路径 P 对应的权重。

值得一提的是,作者在论文中还提到可用矩阵分解的方法来分解每条元路径下的用户偏好扩散矩阵。所谓用户偏好扩散矩阵指遍历所有用户用式(5-27)计算得到在元路径 P 下,它们对每个物品的偏好扩散分数。所有的扩散分数可形成一个用户数量×物品数量维度的用户偏好扩散矩阵,记作 $\boldsymbol{M}_P$。

对 $\boldsymbol{M}_P$ 进行矩阵分解后,得到用户隐向量矩阵 $\boldsymbol{U}^P\in\mathbf{R}^{m\times d}$ 与物品隐向量矩阵 $\boldsymbol{V}^P\in\mathbf{R}^{n\times d}$。其中 m 是用户数量,n 是物品数量,d 是隐向量维度,则式(5-28)可修改为[14]

$$r^{\text{pred}}(u,v)=\sum_{P}^{L}w_P\boldsymbol{U}_u^P\cdot\boldsymbol{V}_v^P \tag{5-29}$$

这么做的好处是提高了模型的泛化能力。

接下来查看 PER 的代码,代码的地址为 recbyhand\chapter5\s43_per.py。

首先定义一种方法,将知识图谱三元组事实数据按照关系分开,每种关系代表不同的元路径类型,代码如下:

```
#recbyhand\chapter5\s43_per.py
#将事实按照关系分开,代表不同的元路径
def splitTriples( kgTriples, movie_set ):
    '''
    :param kgTriples: 知识图谱三元组
    :param movie_set: 包含所有电影的
    '''
    metapath_triples = {}
    for h, r, t in tqdm( kgTriples ):
        if h in movie_set:
            h, r, t = int( h ), int( r ), int( t )
            if r not in metapath_triples:
                metapath_triples[ r ] = [ ]
            metapath_triples[ r ].append( [ h, 1, t ] )
    return metapath_triples
```

因为 PER 需要实体间的路径相似度,为了避免内存溢出,所以可以用邻接表代替实体邻接矩阵,所以先用三元组数据组合成邻接表,然后用邻接表取得相似度矩阵。代码中的

s42_pathsim 是 5.4.3 节中路径相似度的代码，具体的代码如下：

```
#recbyhand\chapter5\s43_per.py
from chapter5 import s42_pathSim
#得到所有元路径下的实体邻接表
def getAdjacencyListOfAllRelations( metapath_triples ):
    print('得到所有元路径下的实体邻接表...')
    r_al = {}
    for r in tqdm(metapath_triples):
        r_al[ r ] = s42_pathSim.getAdjacencyListByTriples( metapath_triples[ r ] )
    return r_al

#得到所有元路径下的电影实体相似度矩阵
def getSimMatrixOfAllRelations( metapath_al, movie_set ):
    print( '计算实体相似度矩阵...' )
    metapath_simMatrixs = { }
    for r in tqdm(metapath_al):
        metapath_simMatrixs[r] = \
            s42_pathSim.getSimMatrixFromAl( metapath_al[r], max(movie_set) + 1 )
    return metapath_simMatrixs
```

有了实体间的路径相似度矩阵，就可以按照 PER 的方法计算用户偏好扩散矩阵了。首先定义一个 PER 的类，在 init()方法中的代码如下：

```
#recbyhand\chapter5\s43_per.py
class PER(nn.Module):

    def __init__(self,kgTriples, user_set, movie_set, recTriples, dim = 8):
        super( PER, self ).__init__( )
        #以不同关系作为不同的元路径切分知识图谱三元组事实
        metapath_triples = splitTriples( kgTriples, movie_set )
        #根据切分好的三元组数据得到各个元路径下的邻接表
        metapath_al = getAdjacencyListOfAllRelations( metapath_triples )
        #根据邻接表得到各个元路径下的路径相似度矩阵
        metapath_simMatrixs = getSimMatrixOfAllRelations( metapath_al, movie_set )

        print("计算用户偏好扩散矩阵...")
        sortedUserItemSims, self.metapath_map = self.init_userItemSims( user_set, recTriples,
metapath_simMatrixs )

        print( '初始化用户物品在每个元路径下的 Embedding...' )
        self.Embeddings = self.init_Embedding( dim, sortedUserItemSims, self.metapath_map )

        #用一个线性层加载每个 metapath 所带的权重
        self.metapath_linear = nn.Linear( len( self.metapath_map ), 1 )
```

其中,计算用户偏好扩散矩阵的 init_userItemSims()方法传入的是用户集、推荐三元组数据及物品在每个元路径下的路径相似度矩阵。该方法除了可以得到用户偏好扩散矩阵外,还可以顺便得到元路径索引的映射 map,具体的代码如下:

```
#recbyhand\chapter5\s43_per.py
#初始化用户偏好扩散矩阵
def init_userItemSims( self, user_set, recTriples, metapath_simMatrixs ):
    #根据推荐三元组数据得到用户物品邻接表
    userItemAl = s42_pathSim.getAdjacencyList( recTriples, r_col = 2 )

    userItemSims = collections.defaultdict(dict)
    for metapath in metapath_simMatrixs:
        for u in userItemAl:
            userItemSims[metapath][u] = \
                np.sum( metapath_simMatrixs[metapath][[ i for i in userItemAl[u] if
userItemAl[u][i] == 1]], axis = 0 )

    userItemSimMatrixs = { }
    for metapath in tqdm( userItemSims ):
        userItemSimMatrix = []
        for u in user_set:
            userItemSimMatrix.append( userItemSims[metapath][int(u)].tolist() )
        userItemSimMatrixs[metapath] = np.mat( userItemSimMatrix )

    metapath_map = { k: v for k, v in enumerate(sorted([metapath for metapath in
userItemSims])) }
    return userItemSimMatrixs, metapath_map
```

初始化用户物品 Embedding 的方法的代码如下:

```
#recbyhand\chapter5\s43_per.py
#初始化用户物品在每个元路径下的 Embedding
def init_Embedding( self, dim, sortedUserItemSims, metapath_map ):
    Embeddings = collections.defaultdict( dict )
    for metapath in metapath_map:
        #根据 NFM 矩阵分解的方式得到用户特征表示及物品特征表示
user_vectors, item_vectors = \
            self.__init_one_pre_emd( sortedUserItemSims[metapath_map[metapath]], dim )
        #分别用先验的用户与物品的向量初始化每个元路径下代表表示用户及表示物品的
#Embedding 层
        Embeddings[metapath]['user'] = \
        nn.Embedding.from_pretrained( user_vectors,max_norm = 1 )
        Embeddings[metapath]['item'] = \
        nn.Embedding.from_pretrained( item_vectors,max_norm = 1 )
    return Embeddings
```

```
#根据 NFM 矩阵分解的方式得到用户特征表示及物品特征表示
def __init_one_pre_emd( self, mat, dim):
u_vectors, i_vectors = getNFM( mat, dim )
    returntorch.FloatTensor(u_vectors),torch.FloatTensor(i_vectors)
```

其中 getNFM()方法利用 Sklearn 库中的 NFM()方法做非负矩阵分解,代码如下:

```
#recbyhand\chapter5\s43_per.py
from sklearn.decomposition import NMF

#NFM 矩阵分解
def getNFM( m, dim):
nmf = NMF( n_components = dim )
user_vectors = nmf.fit_transform( m )
item_vectors = nmf.components_
    return user_vectors,item_vectors.T
```

前向传播 forward 的方法定义如下:

```
#recbyhand\chapter5\s43_per.py
def forward( self, u, v ):
metapath_preds = [ ]
    for metapath in self.metapath_map:
        #[ batch_size, dim ]
        metapath_embs = self.Embeddings[ metapath ]
        #[ batch_size, 1 ]
        metapath_pred = \
            torch.sum( metapath_embs['user'](u) *
                       metapath_embs['item'](v),
                       dim = 1, keepdim = True )
        metapath_preds.append( metapath_pred )
    #[ batch_size, metapath_number ]
    metapath_preds = torch.cat( metapath_preds, 1 )
    #[ batch_size, 1 ]
    metapath_preds = self.metapath_linear( metapath_preds )
    #[ batch_size ]
    logit = torch.sigmoid( metapath_preds ).squeeze( )
    return logit
```

其余外部调用该模型类如何训练的代码在此就不列出了,是常规的 PyTorch 操作。若有兴趣,则可到配套代码中详查。

可以发现主要训练 Linear 层中的那几个权重,而那几个权重对应的便是每条元路径的权重。其余一大堆的操作全是为了利用知识图谱数据生成用户偏好矩阵作为 PER 模型先验知识。

但是目前代码只训练了元路径在综合情况下的权重,也就是说按现在的做法假设了每

个用户对所有元路径的偏好权重都是一样的，这显然并不准确，所以需要给每个用户分配一个不同的权重 w_p^u，但是在实际可统计的数据中，用户与物品的交互数据呈幂律分布，所以大部分用户没有足够的数据去学习权重参数。为此需要先将用户聚类，所以作者提出用矩阵分解得到用户物品隐向量的真正用意其实是为了方便聚类。

对所有元路径下的用户隐向量利用余弦相似度作为度量的 K-Means 算法进行聚类。最终的推荐预估函数的公式如下[14]：

$$r^{\text{pred}}(u,v)=\sum_{P}^{L}\sum_{k}^{C}\cos(C_k^P,u)w_k^P U_u^P \cdot V_u^P \tag{5-30}$$

其中，$\cos(C_k^P,u)$表示用户 u 与该簇中心的余弦相似度，这样做的好处是不只考虑了用户属于的那个簇，而是整合了多个相关簇的信息。

所以最终要学习的权重数量是聚类数量×路径个数。比原来多出了聚类数量倍数的权重个数。这样便具备了个性化推荐的效果，聚类的数量需要大家在实际工作中调参。

鉴于本节主要为了利用 PER 这个比较初级的基于图路径的知识图谱推荐算法，来让大家对利用知识图谱路径做推荐算法有个初步的了解，所以本节代码没有加入聚类之后的内容，否则会更复杂。大家有兴趣可以自己实现一下。

5.4.4 异构图的图游走算法：MetaPath2Vec

在第 4 章中学习过同构图的图游走算法 DeepWalk 与 Node2vec，图游走算法的任务是得到节点的向量表示，所以如果能够对知识图谱使用图游走算法，则路径提供的信息处理起来就没有那么麻烦了，但是知识图谱属于异构图，如果直接采用 DeepWalk，则在随机游走生成序列时，组成元路径数量多的节点会被高概率提取出来，这会导致组成出现频率低的元路径节点没有办法充分学到，所以 MetaPath2Vec[15]的算法被提出。

MetaPath2Vec 是微软研究中心在 2017 年提出的算法，具体怎么做呢？首先假设有一张图数据，如图 5-27 所示。

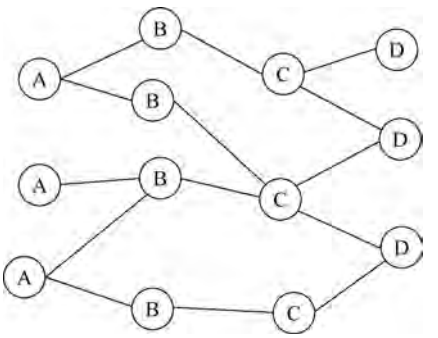


图 5-27 MetaPath2Vec 演示图

图 5-27 中有 4 个节点类型，分别为 A、B、C、D，所以元路径的类型可以定义为 ABA、ABCBA、CDC 等。与 DeepWalk 或者 Node2vec 不同的是 MetaPath2Vec 需要限定游走的序列在指定的元路径类型内。

例如将元路径指定为 ABA。则生成的序列可以是：[A1，B2，A3，B1，A2，B6]。因为 ABA 是对称元路径，所以在生成序列时可以无限循环下去。如果要指定非对称元路径，则该元路径最好较长，因为非对称元路径无法首尾循环游走，所以生成序列的最大长度是指定的元路径长度，如果序列长度不够长，则会影响后续 Word2Vec 生成的 Embedding 效果。

原理很简单，但是一条元路径的 MetaPath2Vec 的效果反而不如 DeepWalk，这里需要

指定多种元路径生成不同的序列再将这些序列一起输入 Word2Vec，以便生成 Embedding，这个操作被称为 Multi-MetaPath2Vec。

下面来看代码，代码的地址为 recbyhand\chapter5\s44_MetaPath2Vec.py。在实战中根据指定元路径生成序列这一操作写起来可能有点麻烦，本次代码会采用一种简便方法，即根据元路径生成不同的子图，而某个子图中只包含某种元路径的节点，所以这样一来后续的操作就跟 DeepWalk 一样了，代码如下：

```
#recbyhand\chapter5\s44_MetaPath2Vec.py
import networkx as nx

#将事实按照关系分开,代表不同的元路径
def splitTriples( kgTriples ):
    '''
    :param kgTriples: 知识图谱三元组
    '''
    metapath_pairs = {}
    for h, r, t in tqdm( kgTriples ):
        if r not in metapath_pairs:
            metapath_pairs[ r ] = [ ]
        metapath_pairs[ r ].append( [ h, t ] )
    return metapath_pairs

#根据边集生成 networkx 的有向图图
def get_graph( pairs ):
    G = nx.Graph( )
    G.add_edges_from( pairs ) #通过边集加载数据
    return G

def fromTriplesGeneralSubGraphSepByMetaPath( triples ):
    '''
    :param triples: 知识图谱三元组信息
    :return: 各个元路径的 networkx 子图
    '''
    metapath_pairs = splitTriples( triples )
    graphs = []
    for metapath in metapath_pairs:
        graphs.append( get_graph( metapath_pairs[metapath] ) )
    return graphs
```

然后将 graphs 传入 multi_metaPath2vec( )的函数中进行不同元路径下的随机游走，代码如下：

```
#recbyhand\chapter5\s44_MetaPath2Vec.py
def multi_metaPath2vec( graphs ,dim = 16, walk_length = 12, num_walks = 256, min_count = 3 ):
    seqs = []
    for g in graphs:
```

```
        #将不同元路径随机游走生成的序列合并起来
        seqs.extend( getDeepwalkSeqs( g, walk_length, num_walks ) )
    model = word2vec.Word2Vec( seqs, size = dim, min_count = min_count )
    return model
```

其中 getDeepwalkSeqs()函数是随机游走生成序列的函数,具体的代码如下:

```
#recbyhand\chapter5\s44_MetaPath2Vec.py
#随机游走生成序列
def getDeepwalkSeqs( g, walk_length, num_walks ):
    seqs = []
    for _ in tqdm(range(num_walks)):
        start_node = np.random.choice(g.nodes)
        w = walkOneTime(g,start_node, walk_length)
        seqs.append(w)
    return seqs

#一次随机游走
def walkOneTime( g, start_node, walk_length ):
    walk = [ str( start_node ) ]        #初始化游走序列
    for _ in range( walk_length ):      #最大长度范围内进行采样
        current_node = int( walk[-1] )
        neighbors = list( g.neighbors( current_node ) ) #获取当前节点的邻居
        if len( neighbors ) > 0:
            next_node = np.random.choice( neighbors, 1 )
            walk.extend( [ str(n) for n in next_node ] )
    return walk
```

外部调用的代码如下:

```
#recbyhand\chapter5\s44_MetaPath2Vec.py
if __name__ == '__main__':
    #读取知识图谱数据
    _, _, triples = dataloader4kge.readKGData()
    graphs = fromTriplesGeneralSubGraphSepByMetaPath( triples )
    model = multi_metaPath2vec( graphs )
    print( model.wv.most_similar( '259', topn = 3 ) ) #观察与节点 259 最相近的 3 个节点
    model.wv.save_Word2Vec_format( 'e.emd' ) #可以把 emd 储存下来,以便下游任务使用
    model.save( 'm.model' ) #可以把模型储存下来,以便下游任务使用
```

本次代码写到生成 Embedding 为止,利用这些 Embedding 可以进行多种多样的后续任务。这些 Embedding 包含了知识图谱路径信息,相比 PER 算法,MetaPath2Vec 显然更简单。相比 KGE 系列的算法,MetaPath2Vec 的劣势是没有学到边或者说关系的 Embedding,而优势是对于实体 Embedding 生成的效果,MetaPath2Vec 的效果普遍来讲高于简单的 KGE。这是因为 MetaPath2Vec 是将知识图谱当成异构图来对待,具备了图的优势。一些复杂的 KGE 也许在某些场景效果会高于 MetaPath2Vec,但是相对复杂 KGE 来讲 MetaPath2Vec 的优势又是简单。

5.4.5　MetaPath2Vec 的扩展

3min

MetaPath2Vec 会有一个很好的优化点，即在 Word2Vec 的负例采样环节限定仅采取指定元路径下的节点。因为默认的负例采样会在全部的节点中采集，这样同样会有不同元路径节点出现频率不平衡等问题。

将 Word2Vec 部分中负例采样环节也限定在指定或者与正例相匹配的元路径下的做法被称为 MetaPath2Vec++。这种做法的缺点是无法用现成的 Word2Vec 方法，必须自己实现，所以在实际工作中如果在技术可行性评估或者模型选型等探索阶段，则没必要用 MetaPath2Vec++，MetaPath2Vec++可作为后续优化时的一个优化点。

5.5　知识图谱嵌入结合图路径的推荐 RippLeNet

本节介绍 KGE 与图路径结合的知识图谱推荐算法，而 RippLeNet[16] 在这一类的推荐算法中是最为典型且效果也非常优秀的。

5.5.1　RippLeNet 基础思想

5min

水波网络(RippLeNet)由上海交通大学和微软亚洲在 2018 年提出。RippLeNet 有效地结合了知识图谱嵌入与知识图谱图路径提供的信息。效果很好，模型可解释性也很便于理解。该算法现在是最热门的知识图谱推荐算法之一。

它的基础思想是利用物品的知识图谱数据一层一层地往外扩散后提取节点，然后聚合 Embedding，每一层的物品会影响到在它之后的所有层，并且越往外对结果的影响就越小，就像水波一样，如图 5-28 所示。

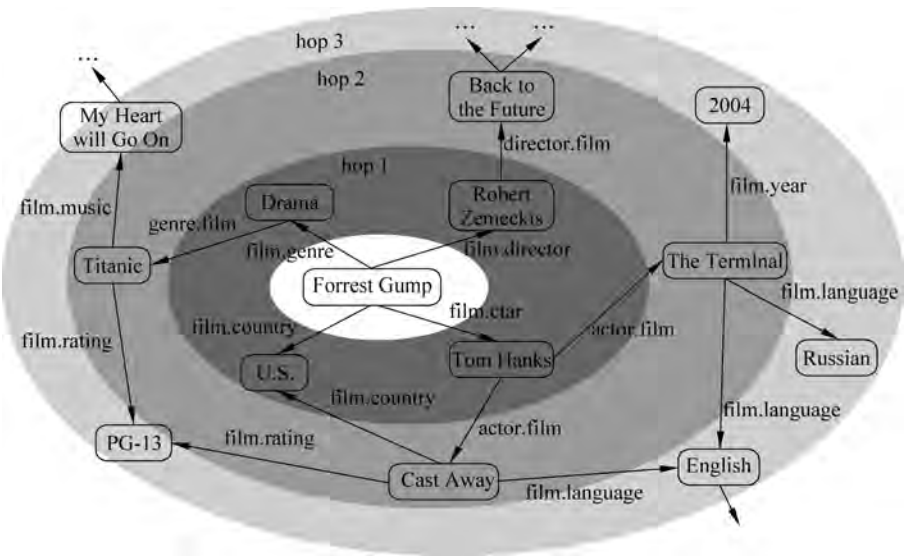


图 5-28　RippLeNet 水波扩散示意图[16]

这听起来很像是一种图采样。虽然说 RippLeNet 的提出年份在 GCN 与 GraphSAGE 之后，但是从 RippLeNet 的论文中可以推理出，作者当时似乎并不是从图神经网络的思想出发，所以 RippLeNet 相当于从侧面碰撞到了图神经网络。

10min

5.5.2 RippLeNet 计算过程

首先参看 RippLeNet 模型的计算总览图，如图 5-29 所示。

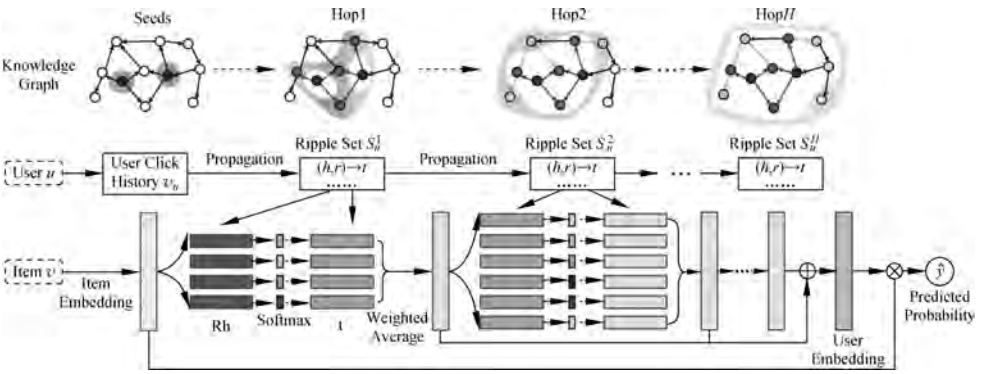


图 5-29 RippLeNet 计算总览图[16]

这张图初看之下有点复杂，为了方便理解，先把注意力集中在 Item Embedding User Embedding 和 Predicted Probability 这三项中，所以先把其余部分遮盖掉，如图 5-30 所示。

图 5-30 RippLeNet 图解(1)[16]

将其余部分先视作黑匣子。这样一来可以理解为，经过一通操作，最后将得到的 User Embedding 与 Item Embedding 做某种计算后预测出该 User 对该 Item 的喜爱程度。如何计算有很多方法，这里就先用论文中给出的最简单的计算方式，其实是在最常用的求内积之后套个 Sigmoid，公式如下：

$$y=\sigma(\boldsymbol{U}^{\mathrm{T}}\boldsymbol{V}) \tag{5-31}$$

$\boldsymbol{V}$ 代表物品向量，随机初始化即可，而用户向量 $\boldsymbol{U}$ 的计算方式如下[16]：

$$\boldsymbol{U}=\boldsymbol{o}_u^1+\boldsymbol{o}_u^2+\cdots+\boldsymbol{o}_u^H \tag{5-32}$$

公式中的 $\boldsymbol{o}_u^H$ 代表第 H 波的输出向量。即图 5-31 中用方框标记出来的长条所代表的向量。

所以关键是如何得到 $\boldsymbol{o}$ 向量，先说第 1 个 $\boldsymbol{o}$ 向量，参看以下公式组[16]：

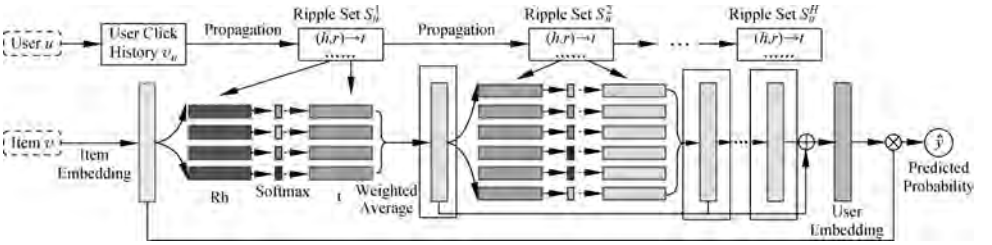


图 5-31　RippLeNet 图解(2)[16]

$$\boldsymbol{o}_u^1 = \sum_{(\boldsymbol{h}_i,\boldsymbol{R}_i,\boldsymbol{t}_i)\in S_u^1} p_i \boldsymbol{t}_i$$

$$p_i = \text{Softmax}(\boldsymbol{V}^T \boldsymbol{R}_i \boldsymbol{h}_i) = \frac{\exp(\boldsymbol{V}^T \boldsymbol{R}_i \boldsymbol{h}_i)}{\sum_{(\boldsymbol{h}_j,\boldsymbol{R}_j,\boldsymbol{t}_j)\in S_u^1} \exp(\boldsymbol{V}^T \boldsymbol{R}_j \boldsymbol{h}_j)}$$

$$\boldsymbol{V},\boldsymbol{h},t \in \mathbf{R}^d \quad \boldsymbol{R} \in \mathbf{R}^{d\times d} \tag{5-33}$$

$\boldsymbol{V}$ 是 Item 向量，$\boldsymbol{t}$ 是 Tail 向量，$\boldsymbol{h}$ 是 Head 向量，$\boldsymbol{r}$ 是 Relation 映射矩阵，这些都是模型要学的 Embedding。式(5-33)表示的计算过程如图 5-32 所示。

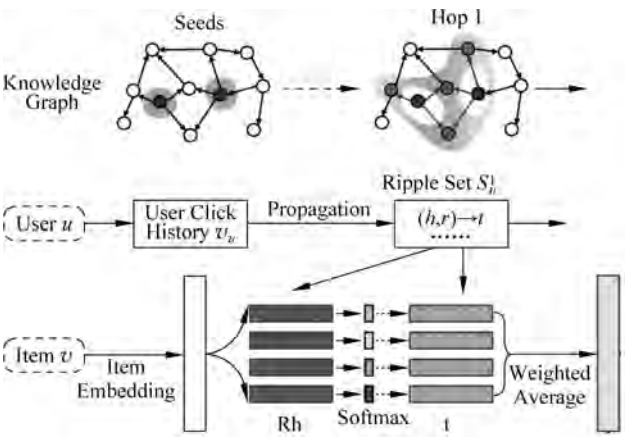


图 5-32　RippLeNet 图解(3)[16]

式(5-33)中的 S_u^1 代表 User 的第一层 Ripple Set(第一层水波集，在图 5-32 中表示为 Hop 1)。首先取一定数量的用户历史交互 Item，然后由这些 Item 作为 Head 实体通过它们的关系 r 找到 Tail 实体，记作$(h,r)\to t$，所以第二层水波(在图 5-29 中表示为 Hop 2)是将第一层水波得到的 Tail 实体作为这一层的 Head 实体从而找到本层对应的 Tail 实体，以此类推。

公式所表达的含义相当于是对每个 Hop 的 Tail 做一个注意力操作，而权重由 Head 和 Relation 得到。

之后将上述步骤得到的 $\boldsymbol{o}$ 向量作为下一层水波的 $\boldsymbol{V}$ 向量，然后重复进行式(5-33) 的计

算。重复 H 次后,就可以得到 H 个 $\boldsymbol{o}$ 向量,然后代入式(5-32)得到用户向量,最后通过与物品向量点积得到预测值。

5.5.3 水波图采样

在水波网络中所谓每一层向外扩散的操作,实际上每一次都会产生笛卡儿乘积数量级的新实体,所以实际操作时需要设定一个值来限定每一次扩散取得新实体的数量上限,假设这个值为 n_memory,若某层中(尤其是初始层)的实体数量不足 n_memory,则在候选实体中进行有返回地重复采样,以此补足 n_memory 个新实体。

水波采样与 GraphSage 略有不同。最普通的 GraphSage 通常会限定每个节点采样的邻居数量,假设这个值为 n,则经过 3 层采样,则总共采集到的节点数量为 n^3,而水波网络限定的是每一层采样的邻居总数量为 n,即经过 3 层采样后采集到的节点数量为 $3\times n$。对于利用物品图谱作推荐的模型,水波采样的方式有下几点优势:

(1) 统一了每一层中实体的数量,便于模型训练时的 batch 计算。

(2) 在物品图谱中不需要对每个节点平等对待,位于中心的物品节点及周围的一阶邻居采样被采集的概率较高,而越外层的节点被采集到的概率越低,这反而更合理。水波模型中心节点是该用户历史最近交互的物品实体,而越往外扩散自然应该像水波一般慢慢稀释后续实体的权重。

水波采样的实际操作还需注意的是,训练时每一次迭代应该重新进行一次随机采样,增加训练数据的覆盖率。在做评估时同样也需重新进行一次随机采样,而不是用训练时已经采样好的水波集作为评估时的水波集。

在做预测时,理论上讲用户的 Embedding 也会根据其历史交互的正例物品通过随机采样的方式进行水波扩散而得到,所以预测时也有随机因子,用户相同的请求也会得到略有不同的推荐列表,但是如果不想有随机因子,则可采取训练时最后采样得到的水波集聚合出的用户 Embedding 直接用作后续计算。

另外水波采样的层数也有讲究,水波采样的层数最好为最小对称元路径阶数的倍数,下面来慢慢讲解这句话。

元路径的阶数指的是元路径包含的节点类型数量。例如:

元路径为电影→演员→电影,则阶数为 3。

元路径为电影→角色→演员→角色→电影,则阶数为 5。

水波采样最好通过候选物品的知识图谱扩散出去找到相关的其他目标物品,从而挖掘出推荐物品。假设目前知识图谱的最小对称元路径是电影→演员→电影,即路径阶数为 3。如果水波采样层数仅为 2,则代表每次迭代根本就没有挖掘出其他电影,而仅采集到演员,所以采样的水波层数需要不小于最小对称元路径阶数。

至于为什么最好是最小对称元路径路径阶数的倍数倒并不是很关键,为倍数的优势是因为这样大概率能在最终一层的水波集节点停留在与候选物品同样实体类型的实体上,实测后得知这对模型的训练有正向影响,但更关键的是要保证水波采样的层数要大于或等于

最小对称元路径阶数。

5.5.4 RippLeNet 实际操作时的注意事项与代码范例

本次代码的地址为 recbyhand\chapter5\dataloader4KGNN.py（负责与读取数据相关的操作）与 recbyhand\chapter5\s54_rippLeNet.py（RippLeNet 主脚本）。代码量有点多，所以此处就不全列出来了，大家去配套代码中查看即可。在此介绍一下实际操作 RippLeNet 时应注意的事项及一些关键的代码。

43min

（1）用户向量的得到除了计算过程中式(5-32)的方式外，还有另一种方法，即直接取最后一层的输出 $\boldsymbol{o}$。后者的优势是模型收敛得会更快，但是稳健性不如前者。

在代码中的体现如下：

```
#recbyhand\chapter5\s54_rippLeNet.py
def _get_user_Embeddings( self, o_list ):
    '''
    :param o_list: 每个 hop 得到的 o 向量集
    :return: 用户向量
    '''
    user_embs = o_list[ - 1]
    #选择是否使用全部的 o 向量相加作为用户向量，否则仅用最后一层的 o 向量
    if self.using_all_hops:
        for i in range(self.n_hop - 1):
            user_embs += o_list[i]
    return user_embs
```

（2）在实际操作中最后与用户向量计算的 Item 向量也可以经过一些迭代更新，作者提出了如下 4 种更新方式。

replace：直接用新一波预测的物品向量（$\boldsymbol{o}$ 向量）替代，如果用户向量获取策略采取的是最后一层 $\boldsymbol{o}$ 向量，则此方法不适用。

plus：与 $t-1$ 波次的物品向量对应位相加，如果用户向量获取策略采取的是将所有 $\boldsymbol{o}$ 向量相加，则此方法不适用。

replace_transform：用一个映射矩阵将预测的物品向量映射一下。

plus_transform：用一个映射矩阵将预测的物品向量映射一下后与 $t-1$ 波次的物品向量对应位相加。

在代码中的体现如下：

```
$ recbyhand\chapter5\s54_rippLeNet.py
#迭代物品的向量
def _update_item_Embedding( self, item_Embeddings, o ):
    '''
    :param item_Embeddings: 上一个 hop 的物品向量 #[ batch_size, dim ]
```

```
        :param o: 当前 hop 的 o 向量 #[ batch_size, dim ]
        :return: 当前 hop 的物品向量 #[ batch_size, dim ]
        '''
        if self.item_update_mode == "replace":
            item_Embeddings = o
        elif self.item_update_mode == "plus":
            item_Embeddings = item_Embeddings + o
        elif self.item_update_mode == "replace_transform":
            item_Embeddings = self.transform_matrix( o )
        elif self.item_update_mode == "plus_transform":
            item_Embeddings = self.transform_matrix( item_Embeddings + o )
        else:
            raise Exception( "位置物品更新 mode: " + self.item_update_mode )
        return item_Embeddings
```

(3) 因为每一层水波中都包含$(\boldsymbol{h},\boldsymbol{r})\rightarrow\boldsymbol{t}$的映射关系,所以训练过程中可利用这些数据使用一些知识图谱嵌入(KGE)的方式产生一个额外的损失函数与主模型中的损失函数相加后一起优化。

在代码中的体现如下:

```
$ recbyhand\chapter5\s54_rippLeNet.py
#生成 kge loss 来联合训练
def _get_kg_loss( self, h_emb_list, r_emb_list, t_emb_list):
    '''
    h_emb_list,r_emb_list,t_emb_list 是水波采样的实体与关系集,三者间位置对应
    '''
    kge_loss = 0
    for hop in range( self.n_hop ):
        #[batch size, n_memory, 1, dim]
        h_expanded = torch.unsqueeze( h_emb_list[hop], dim = 2 )
        #[batch size, n_memory, dim, 1]
        t_expanded = torch.unsqueeze( t_emb_list[hop], dim = 3 )
        #[batch size, n_memory, dim, dim]
        hRt = torch.squeeze(
            torch.matmul( torch.matmul( h_expanded, r_emb_list[ hop ] ), t_expanded ))
        kge_loss += torch.sigmoid( hRt ).mean( )
    kge_loss = -self.kge_weight * kge_loss
    return kge_loss
```

本次代码中采用的 KGE 方法是 RESCAL,在上面倒数第二行的代码中之所以加入负号实际上是一种技巧。因为省略掉了负例采样,而单用点乘后 Sigmoid 的值做评分是越高越好的,所以在联合训练时将点乘后的结果减去自然意味着点乘得到的值越高则损失函数会越低。为什么说这是一种技巧?因为如果推荐预测的损失函数的绝对值小于 KGE 损失函数的绝对值,则整体的损失函数就会呈现负数,模型就无法收敛,所以为了避免整体损失

函数出现负数，kge_weight 实际上要设得足够小，例如 0.01。

综合来讲，在联合训练中 KGE 的确可以省略负例采样，而省略后的注意事项是前段话中的那些内容。

如果大家已经具备图神经网络的知识，则一定会发现本次 RippLeNet 的代码有很多可优化的部分，而示例代码之所以没有写得像图神经网络那样是为了尽可能与 RippLeNet 作者发表的源码相似，其实仅仅在源码的基础上做了略微改进。这么做的原因是因为一来比较原汁原味的 RippLeNet 能更加有助于大家理解知识图谱的发展路线，从而能够更好地往下理解后续知识图谱与图神经网络相结合的算法。二来可以让大家在自行搜索源码比对学习时不至于太混乱。

5.6 图神经网络与知识图谱

终于到了讲解图神经网络结合知识图谱的算法了，图神经网络的算法用到知识图谱推荐领域要注意的重点是要将知识图谱的头实体(Head)、关系(Relation)和尾实体(Tail)一同考虑进去。尤其是 Relation，在图论中是边，在普通 GNN 算法中不会给边附加一个特征向量，而如果忽略知识图谱中边对整个数据的影响，则知识图谱也就退化成了一个普通图，所以既然说要利用知识图谱做推荐模型，则设计算法时对于边不容忽视。

5.6.1 最基础的基于图神经网络的知识图谱推荐算法 KGCN



13min

首先介绍 KGCN[17]，KGCN 是知识图谱与图神经网络结合的典型案例，也是最基础的案例。提出于 2019 年，KGCN 的作者也是 RippLeNet 的作者。其中心思想是利用图神经网络的消息传递机制与基本推荐思想结合训练。



14min

虽然名字中有 GCN，但其实主要还是以计算注意力的方式进行消息传递。这个注意力可被理解为该关系影响用户行为的偏好程度。例如用户是更喜欢通过相似演员还是更喜欢通过相似题材寻找喜欢的电影。

首先设用户是 U，用户向量表示为 $\boldsymbol{u}$，物品为 V，物品向量表示为 $\boldsymbol{v}$。若要计算用户 U 对物品 V 的评分预测，最基础的公式如下：

$$\hat{y}_{UV} = f(\boldsymbol{u}, \boldsymbol{v}) \tag{5-34}$$

其中，$f(\cdot)$是任意函数，例如求内积，即点乘，$\hat{y}$ 是预测的值。

假设图 5-33 所示的是一次图采样得到的子图。中间的节点指的是要预测的目标物品 V。N_i 表示 V 的邻居。R_i 则表示关系。直接将关系向量 $\boldsymbol{r}_i$ 用作消息传递的权重可以吗？可以是可以，但效果不如以下做法，详见图 5-34。

如图 5-34 所示，每条边用一个权重 w 来表示，并令

$$w_{R_i}^{U} = g(\boldsymbol{u}, \boldsymbol{r}_i) \tag{5-35}$$

其中，$\boldsymbol{u}$ 是用户的向量，$\boldsymbol{r}_i$ 是连接第 i 个邻居的关系向量。$g(\cdot)$是任意函数，例如求内积。

$w_{R_i}^U$ 表示目标用户 U 对关系 R_i 的偏好程度，即经过 R_i 边时消息传递的权重。因为每次消息传递都加入了用户向量，所以结果自然比直接取关系向量 $\boldsymbol{r}_i$ 更能体现出用户 U 的注意力。

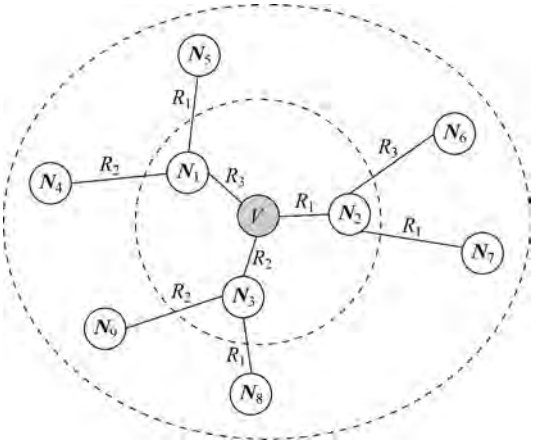


图 5-33　KGCN 流程图(1)

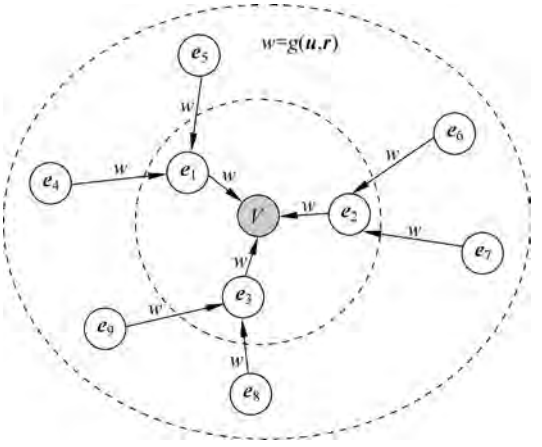


图 5-34　KGCN 流程图(2)

与所有注意力机制一样，接下来对 $w_{R_i}^U$ 做一次 Softmax 操作来归一化。

$$\tilde{\boldsymbol{w}}_{R_i}^U = \mathrm{Softmax}_j(w_{R_i}^U) = \frac{\exp(w_{R_i}^U)}{\sum\limits_{j \in N_{(V)}} \exp(w_{R_j}^U)} \tag{5-36}$$

其中，$N_{(V)}$ 代表节点 V 的一阶邻居集，然后进行一次加权求和操作得到特征向量 $\tilde{\boldsymbol{v}}$，公式如下[17]：

$$\tilde{\boldsymbol{v}} = \sum_{i \in N_{(V)}} \tilde{\boldsymbol{w}}_{R_i}^U \cdot \boldsymbol{e}_i \tag{5-37}$$

其中，$\boldsymbol{e}_i$ 代表第 i 个邻居的特征向量。根据图采样的思想，消息传递是由外而内进行的，所以在将图中的节点 $\boldsymbol{e}_4$ 和 $\boldsymbol{e}_5$ 传递到 $\boldsymbol{e}_1$ 时。则此时表示 $\boldsymbol{e}_1$ 的向量就相当于式(5-37)中的 $\tilde{\boldsymbol{v}}$。计算得到 $\boldsymbol{e}_1$ 消息聚合后的特征向量后。再将 $\boldsymbol{e}_1$ 代入式(5-37)中 $\boldsymbol{e}_i$ 的位置，以此类推，最终传递给中心位置 V，得到中心节点 V 的特征向量 $\tilde{\boldsymbol{v}}$。

细心的读者一定会发现，$\boldsymbol{v}$ 的头上有一个波浪线。也就是说 $\tilde{\boldsymbol{v}}$ 还不是最终代表目标物品 V 的特征向量 $\boldsymbol{v}$。因为 $\tilde{\boldsymbol{v}}$ 到 $\boldsymbol{v}$ 其实还可以进行另一次消息聚合，以及经一次或多次全连接层的操作。该过程的公式如下[17]：

$$\boldsymbol{v}=\sigma(\boldsymbol{W}\cdot \mathrm{agg}(\tilde{\boldsymbol{v}},\boldsymbol{e})+\boldsymbol{b}) \tag{5-38}$$

其中，$\sigma(\cdot)$是非线性激活函数，如 ReLU、Sigmoid 等。$\boldsymbol{W}$ 是线性变换矩阵，$\boldsymbol{b}$ 是偏置项。这些都是一个全连接层的基本元素，而 $\mathrm{agg}(\tilde{\boldsymbol{v}},e)$ 表示对物品 V 再做一次消息聚合(Aggregator)，不带下标的字母 $\boldsymbol{e}$ 表示物品 V 初始的特征向量，抑或是物品 V 在前一轮迭代更新产生的向量。作者在论文中提到了 3 种聚合方式。

(1) 求和聚合(Sum)：即将 $\tilde{\boldsymbol{v}}$ 与 $\boldsymbol{e}$ 对应的元素位相加，公式如下[17]：

$$\mathrm{agg}_{\mathrm{sum}}(\tilde{\boldsymbol{v}},\boldsymbol{e})=\tilde{\boldsymbol{v}}+\boldsymbol{e} \tag{5-39}$$

(2) 拼接聚合(Concat)：即将向量 $\tilde{\boldsymbol{v}}$ 与向量 $\boldsymbol{e}$ 拼接起来，如果原本它们的维度都为 F，则拼接后的向量维度是 2F，所以使用拼接聚合时要注意式(5-38)中的线性变化矩阵 $\boldsymbol{W}$ 和偏置项 $\boldsymbol{b}$ 的维度也需相应地变化。拼接聚合的公式如下[17]：

$$\mathrm{agg}_{\mathrm{concat}}(\tilde{\boldsymbol{v}},\boldsymbol{e})=\tilde{\boldsymbol{v}}\parallel\boldsymbol{e} \tag{5-40}$$

(3) 邻居聚合(Neighbor)：所谓邻居聚合是直接采用 $\tilde{\boldsymbol{v}}$ 当作本层输出向量，之所以取名为邻居聚合，是因为得到的 $\tilde{\boldsymbol{v}}$ 的计算过程本身也能称为邻居聚合，具体公式如下[17]：

$$\mathrm{agg}_{\mathrm{neighbor}}(\tilde{\boldsymbol{v}},\boldsymbol{e})=\tilde{\boldsymbol{v}} \tag{5-41}$$

至此，将聚合好的向量代入式(5-38)，即可得到这一轮表示物品 V 的向量 $\boldsymbol{v}$，然后代入式(5-34)，即可得到预测值 $\hat{y}_{\mathrm{UV}}$，然后与真实值 y_{UV} 建立损失函数，公式如下：

$$\mathrm{loss}=L(y_{\mathrm{UV}},\hat{y}_{\mathrm{UV}}) \tag{5-42}$$

$L(\cdot)$表示一个损失函数，例如如果是 CTR 预估，则是 BCE 损失函数，如果是评分预测，则可以是平方差损失函数。

接下来看代码，本次代码的地址为 recbyhand\chapter5\s61_KGCN.py。书中直接从前向传播方法切入，代码如下：

```
#recbyhand\chapter5\s61_KGCN.py
def forward( self, users, items, is_evaluate = False ):
    user_Embeddings = self.user_Embedding( users)
    item_Embeddings = self.entity_Embedding( items )
    #得到邻居实体和连接它们关系的 Embedding
    neighbor_entitys, neighbor_Relations = self.get_neighbors( items )
    #得到 v 波浪线
```

```
        neighbor_vectors = self.__get_neighbor_vectors( neighbor_entitys, neighbor_Relations,
        user_Embeddings )
        #聚合得到物品向量
        out_item_Embeddings = self.aggregator( item_Embeddings, neighbor_vectors, is_evaluate )

        out = torch.sigmoid( torch.sum( user_Embeddings * out_item_Embeddings, dim = -1 ) )

        return out
```

传入的 users 和 items 自然是一批次采样的用户与物品索引。is_evaluate 用于判断是否是评估的 flag,主要是为了在训练时某些位置采取 DropOut,而评估时则可取消 DropOut 的操作。前向传播中的一些方法如__get_neighbor_vectors() 是实现式(5-35)到式(5-37)的计算过程,aggregator()方法是实现式(5-38)到式(5-41)的计算过程,这些就不在书中展开讲解了。这里重点要讲解 get_neighbors()方法,详细代码如下:

```
#recbyhand\chapter5\s61_KGCN.py
#得到邻居的节点 Embedding 和关系 Embedding
    def get_neighbors( self, items ):
    e_ids = [self.adj_entity[ item ] for item in items ]
    r_ids = [ self.adj_Relation[ item ] for item in items ]
    e_ids = torch.LongTensor( e_ids )
    r_ids = torch.LongTensor( r_ids )
    neighbor_entities_embs = self.entity_Embedding( e_ids )
    neighbor_Relations_embs = self.Relation_Embedding( r_ids )
    return neighbor_entities_embs, neighbor_Relations_embs
```

这是一个通过物品 id 得到其邻居及连接它们的关系的 id,然后通过邻居和关系 id 得到邻居实体与关系的 Embedding 的方法,而中间的 adj_entity 与 adj_Relation 分别是图采样后得到的指定度数的邻接列表。数据的形式是二维列表,例如:[[1,2,3,4,5],[2,3,4,5,6]…],外层列表的索引对应的是实体索引。因为数据经处理后实体的所有索引是按从 0 到"实体数量-1"的顺序排列的整数,所以可直接用列表索引指代实体的索引。adj_Relation 是关系的邻接列表,外层列表的索引同样对应的是实体索引。假设 adj_entity=[[1,2,3]],adj_relation=[[1,0,2]],则实体与关系的排列如图 5-35 所示。

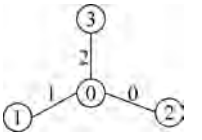


图 5-35 KGCN 图采样说明图

生成它们的方式的代码如下:

```
#recbyhand\chapter5\dataloader4KGNN.py
#根据 kg 邻接列表,得到实体邻接列表和关系邻接列表
def construct_adj( neighbor_sample_size, kg_indexes, entity_num ):
    print('生成实体邻接列表和关系邻接列表')
    adj_entity = np.zeros([ entity_num, neighbor_sample_size ], dtype = np.int64 )
```

```
    adj_Relation = np.zeros([ entity_num, neighbor_sample_size ], dtype = np.int64 )
    for entity in range( entity_num ):
        neighbors = kg_indexes[ str( entity ) ]
        n_neighbors = len( neighbors )
        if n_neighbors >= neighbor_sample_size:
            sampled_indices = np.random.choice( list(range(n_neighbors ) ),
                                        size = neighbor_sample_size, replace = False )
        else:
            sampled_indices = np.random.choice( list(range(n_neighbors ) ),
                                        size = neighbor_sample_size, replace = True )
        adj_entity[ entity ] = np.array( [ neighbors[i][0] for i in sampled_indices ] )
        adj_Relation[ entity] = np.array( [ neighbors[i][1] for i in sampled_indices ] )
    return adj_entity, adj_Relation
```

其中 kg_indexes 的数据形式是一个字典，生成的代码如下：

```
#recbyhand\chapter5\dataloader4KGNN.py
def getKgIndexsFromKgTriples( kg_triples ):
    kg_indexs = collections.defaultdict( list )
    for h, r, t in kg_triples:
        kg_indexs[ str( h ) ].append([ int( t ), int( r ) ])
    return kg_indexs
```

其中 kg_triples 是知识图谱三元组数据，到这应该没问题了，所以这一整串事情其实也是图采样的一种实现方式。代码的其余部分相对来讲比较简单，大家可自行列配套代码中对应的位置查看。

5.6.2　KGCN 的扩展 KGNN-LS

KGNN-LS[18]是 KGCN 的作者在同年提出的一个对 KGCN 有效优化的方式。其中的 LS 表示标签平滑正则化(Label Smoothness Regularization)，主要是为了防止过拟合。

首先在数据预处理的时候，用无监督的方式计算出原本无标注节点的标注，公式如下[18]：

$$\hat{y}_{UV}^{LS}=\frac{1}{|N_{(V)}|U|}\sum_{i\in N_{(V)}|U}\tilde{w}_{R_{Vi}}^{U}y_{Ui} \tag{5-43}$$

其中，y_{Ui} 代表用户 U 对节点 i 原来的标注。例如在一个电影推荐场景，节点 i 是一部电影，若用户喜欢该电影，则 y_{Ui} 为 1，如果不喜欢，则为 0。$\hat{y}_{UV}^{LS}$ 是要预测的节点 V 的标注，可能节点 V 是用户没看过的一部电影，也可能是导演或演员等实体；上标 LS 代表是 LS 部分的预测标注，用以区分 KGCN 的预测。$N_{(V)}|U$ 代表与用户 U 有交互的节点 V 的一阶邻居集，$|N_{(V)}|U|$ 代表节点 V 在这种情况下的邻居数量，即度。$\tilde{w}_{R_{Vi}}^{U}$ 是连接节点 V 与节点 i 那条边的权重，由式(5-36)得到。

在训练过程中,遍历所有标注的节点,遮盖住目标节点的标注,用它邻居的标注预测出目标节点的标注,预测的方式如式(5-43)所示,然后与它真实的节点建立损失函数。该过程的公式可描述为

$$\text{loss}_{\text{LS}} = L(y_{\text{UV}}, \hat{y}_{\text{UV}}^{\text{LS}}) \tag{5-44}$$

其中,$L(\cdot)$表示一个损失函数,目标节点的邻居节点标注是上文提到的在数据预处理时无监督计算得到的标注,也有原本是真实的标注,注意计算 $\hat{y}_{\text{UV}}^{\text{LS}}$ 时仅需遮盖目标节点自身的真实标注,无须遮盖目标节点的邻居节点的真实标注。

这是 LS 正则化部分损失函数的计算过程。如果用 $\text{loss}_{\text{KGCN}}$ 代表 KGCN 部分的损失函数,则整个 KGNN — LS 的损失函数的公式如下[18]:

$$\text{loss} = \text{loss}_{\text{KGCN}} + \gamma \text{loss}_{\text{LS}} = L(y_{\text{UV}}, \hat{y}_{\text{UV}}) + \gamma \cdot L(y_{\text{UV}}, \hat{y}_{\text{UV}}^{\text{LS}}) \tag{5-45}$$

其中,γ 是 LS 正则项的系数,加入 LS 正则项后,训练的时间复杂度会提高很多,但是的确能提高 KGCN 的效果,虽然提高的效果相较提高的训练时间而言显得不是很划算,但是在实际工作中 KGNN-LS 也可作为需要精益求精场景的优化手段之一。

5.6.3 图注意力网络在知识图谱推荐算法中的应用 KGAT

知识图谱注意力网络(Knowledge Graph Attention Network,KGAT)[19]是由新加坡国立大学与中国科学技术大学于 2019 年提出。中心思想是利用图注意力网络进行消息传递,从而聚合出物品向量之后与用户向量进行计算得到预测值。

这个算法中注意力是由头实体 $\boldsymbol{h}$、关系 $\boldsymbol{r}$ 与尾实体 $\boldsymbol{t}$ 的 3 个向量计算而来,公式如下[19]:

$$a(\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t}) = \text{Softmax}((\boldsymbol{W}_r \boldsymbol{e}_t)^{\text{T}} \tanh(\boldsymbol{W}_r \boldsymbol{e}_h + \boldsymbol{e}_r)) \tag{5-46}$$

其中,$\boldsymbol{e}_h$ 和 $\boldsymbol{e}_t$ 是头尾实体向量,假设维度是 e_dim,$\boldsymbol{e}_r$ 是关系向量,另一维度假设为r_dim,$\boldsymbol{W}_r$ 是关系变换矩阵,则维度相应为 e_dim×r_dim。关系变换矩阵的意义与 TransR 中的关系变换矩阵一样,也是将头尾实体向量映射到 $\boldsymbol{r}$ 向量空间后进行运算。其优势是 TransH 中提到的优势(TransR 可视为 TransH 的简化)。

然后在头实体的位置初步聚合,公式如下[19]:

$$\boldsymbol{e}_{\text{Nh}} = \sum_{(\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t}) \in \text{Nh}} a(\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t}) \times \boldsymbol{e}_t \tag{5-47}$$


11min

其中,$(\boldsymbol{h}, \boldsymbol{r}, \boldsymbol{t}) \in \text{Nh}$ 指遍历 $\boldsymbol{h}$ 实体的邻居尾实体和连接它们的关系,而 $\boldsymbol{h}$ 在计算过程中将被广播。

式(5-46)与式(5-47)的消息传递过程如图 5-36 所示。

得到初步的聚合向量 $\boldsymbol{e}_{\text{Nh}}$ 后,有 3 种进一步聚合的方式,主要是 $\boldsymbol{e}_{\text{Nh}}$ 与原始头实体的向量 $\boldsymbol{e}_h$ 以不同方式进行消息聚合,分别如下[19]。

(1) 求和聚合(Sum):

$$\text{agg}_{\text{sum}} = \text{LeakyReLU}(\boldsymbol{W}(\boldsymbol{e}_h + \boldsymbol{e}_{Nh})) \tag{5-48}$$

(2) 拼接聚合(Concat):

$$\text{agg}_{\text{concat}} = \text{LeakyReLU}(\boldsymbol{W}(\boldsymbol{e}_h \parallel \boldsymbol{e}_{Nh})) \tag{5-49}$$

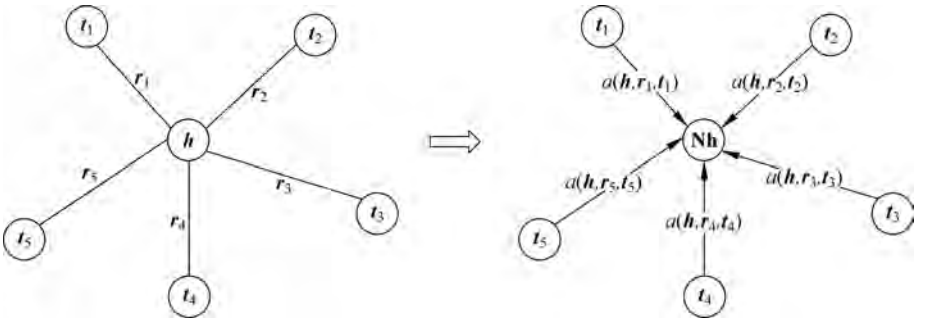


图 5-36　KGAT 消息传递示意图

（3）双相互作用聚合(Bi-Interaction)：

$$\text{agg}_{\text{bi-interaction}} = \text{LeakyReLU}(\boldsymbol{W}_1(\boldsymbol{e}_h + \boldsymbol{e}_{\text{Nh}})) + \text{LeakyReLU}(\boldsymbol{W}_2(\boldsymbol{e}_h \odot \boldsymbol{e}_{\text{Nh}})) \quad (5\text{-}50)$$

其中，$\boldsymbol{W}$ 是线性变换矩阵，在代码中可以直接声明一个线性层来代替。注意输入和输出即可，输入是根据头实体向量与聚合向量相应的维度而来，输出则要与用户向量对应。因为下一步是与用户向量进行计算得到预测值，物理意义是该用户对该头实体或者某个物品的兴趣程度，当然一般的计算方式是一个点乘套 Sigmoid 即可，最后再与真实的标注建立损失函数。

本节示例代码的地址为 recbyhand\chapter5\s63_KGAT.py。

下面拆解来看 KGAT 的主要核心代码，首先初始化 KGAT 类的代码如下：

```
#recbyhand\chapter5\s63_KGAT.py
class KGAT( nn.Module ):

    def __init__( self, n_users, n_entitys, n_Relations, e_dim, r_dim,
adj_entity, adj_Relation ,agg_method = 'Bi-Interaction'):
        super( KGAT, self ).__init__( )

        self.user_embs = nn.Embedding( n_users, e_dim, max_norm = 1 )
        self.entity_embs = nn.Embedding( n_entitys, e_dim, max_norm = 1 )
        self.Relation_embs = nn.Embedding( n_Relations, r_dim, max_norm = 1 )

        self.adj_entity = adj_entity                #节点的邻接列表
        self.adj_Relation = adj_Relation            #关系的邻接列表

        self.agg_method = agg_method                #聚合方法

        #初始化计算注意力时的关系变换线性层
        self.Wr = nn.Linear( e_dim, r_dim )

        #初始化最终聚合时所用的激活函数
        self.leakyReLU = nn.LeakyReLU( negative_slope = 0.2 )
```

```
        #初始化各种聚合时所用的线性层
        if agg_method == 'concat':
            self.W_concat = nn.Linear( e_dim * 2, e_dim )
        else:
            self.W1 = nn.Linear( e_dim, e_dim )
            if agg_method == 'Bi-Interaction':
                self.W2 = nn.Linear( e_dim, e_dim )
```

消息传递后,初步聚合的函数的代码如下:

```
#recbyhand\chapter5\s63_KGAT.py
#GAT 消息传递
def GATMessagePass( self, h_embs, r_embs, t_embs ):
    '''
    :param h_embs: 头实体向量[ batch_size, e_dim ]
    :param r_embs: 关系向量[ batch_size, n_neibours, r_dim ]
    :param t_embs: 尾实体向量[ batch_size, n_neibours, e_dim ]
    '''
    #将 h 张量广播,维度扩散为 [ batch_size, n_neibours, e_dim ]
    h_broadcast_embs = torch.cat( [ torch.unsqueeze( h_embs, 1 ) for _ in range( t_embs.shape
[ 1 ] ) ], dim = 1 )
    #[ batch_size, n_neibours, r_dim ]
    tr_embs = self.Wr( t_embs )
    #[ batch_size, n_neibours, r_dim ]
    hr_embs = self.Wr( h_broadcast_embs )
    #[ batch_size, n_neibours, r_dim ]
    hr_embs = torch.tanh( hr_embs + r_embs)
    #[ batch_size, n_neibours, 1 ]
    atten = torch.sum( hr_embs * tr_embs,dim = -1 ,keepdim = True)
    atten = torch.Softmax( atten, dim = -1 )
    #[ batch_size, n_neibours, e_dim ]
    t_embs = t_embs * atten
    #[ batch_size, e_dim ]
    return torch.sum( t_embs, dim = 1 )
```

该函数返回的是本节公式中的 $\boldsymbol{e}_{\mathrm{Nh}}$。

下面是用 $\boldsymbol{e}_{\mathrm{Nh}}$ 与原始的头实体向量 $\boldsymbol{e}_h$ 进行进一步聚合的函数,有 3 种聚合方法可供选择,具体的代码如下:

```
#recbyhand\chapter5\s63_KGAT.py
#消息聚合
def aggregate( self, h_embs, Nh_embs, agg_method = 'Bi-Interaction' ):
    '''
    :param h_embs: 原始的头实体向量 [ batch_size, e_dim ]
```

```
        :param Nh_embs: 消息传递后头实体位置的向量 [ batch_size, e_dim ]
        :param agg_method: 聚合方式,总共有 3 种,分别是'Bi - Interaction'、'concat'、'sum'
        '''
        if agg_method == 'Bi - Interaction':
            return self.leakyReLU( self.W1( h_embs + Nh_embs ) )\
                    + self.leakyReLU( self.W2( h_embs * Nh_embs ) )
        elif agg_method == 'concat':
            return self.leakyReLU( self.W_concat( torch.cat([ h_embs,Nh_embs ], dim = -1 ) ) )
        else: #sum
            return self.leakyReLU( self.W1( h_embs + Nh_embs ) )
```

模型的前向传播方法,代码如下:

```
#recbyhand\chapter5\s63_KGAT.py
def forward( self, u, i ):
    ##[ batch_size, n_neibours, e_dim ] and #[ batch_size, n_neibours, r_dim ]
    t_embs, r_embs = self.get_neighbors(i)
    ##[ batch_size, e_dim ]
    h_embs = self.entity_embs(i)
    ##[ batch_size, e_dim ]
    Nh_embs = self.GATMessagePass( h_embs, r_embs, t_embs )
    ##[ batch_size, e_dim ]
    item_embs = self.aggregate( h_embs, Nh_embs, self.agg_method )
    ##[ batch_size, e_dim ]
    user_embs = self.user_embs( u )
    ##[ batch_size ]
    logits = torch.sigmoid( torch.sum( user_embs * item_embs, dim = 1 ) )
    return logits
```

其中的 get_neighbors()方法其实和 KGCN 代码中的 get_neighbors()方法相同。图采样的方式也和 KGCN 代码中图采样的方式相同。其余的内容大家可到示例代码中详查。

另外在实际工作中注意力的使用也可采取多头注意力,以及与所有知识图谱推荐算法一样,可以与知识图谱嵌入任务做联合训练。

5.6.4　GFM 与知识图谱的结合 KGFM

8min

又到了用所学的知识自己推演算法的时候了,结合 4.4.3 节中 GFM 的思路,在知识图谱的推荐算法中可以将 FM 的操作插入某个位置来学实体交叉后的信息,FM 这一步的计算公式如下:

$$\boldsymbol{e}_{\mathrm{Nh}} = \left(\sum_{i=1}^{n} \boldsymbol{x}_i\right)^2 - \sum_{i=1}^{n} \boldsymbol{x}_i^2 \tag{5-51}$$

其中,n 的数量是实体 h 此次采样总共得到的三元组数量。这意味着与实体 h 相连的尾实体 t 有 n 个,关系 r 也有 n 条,所以 $\boldsymbol{x}_i$ 代表由第 i 组 h、r 和 t 得到的向量。$\boldsymbol{x}_i$ 的计算公式

如下：

$$\boldsymbol{x}_i = f_{\text{KGAT}}(\boldsymbol{h}, \boldsymbol{r}_i, \boldsymbol{t}_i) = (\boldsymbol{W}_r \boldsymbol{e}_t^i) \odot \tanh(\boldsymbol{W}_r \boldsymbol{e}_h + \boldsymbol{e}_r^i) \tag{5-52}$$

这个公式是从 KGAT 在计算注意力权重中的公式推演而来，这种计算的意义主要还是参考了 TransR。这样使 $\boldsymbol{x}_i$ 蕴含了第 i 组 h、r 和 t 三者的信息。

然后将 x_i 代入公式(5-51)得到 $\boldsymbol{e}_{\text{Nh}}$，$\boldsymbol{e}_{\text{Nh}}$ 再与头实体原本的向量 $\boldsymbol{e}_h$ 进行进一步的聚合，聚合方式可参考 KGAT 的那 3 种聚合方式，分别是求和聚合(Sum)、拼接聚合(Concat)与双相互作用聚合(Bi－Interaction)。

后面的操作就跟 KGAT 之后的操作一样了，即将更新后的头实体向量与用户向量计算得到预测值，所以 KGFM 的代码仅需要在 KGAT 的基础上改动一处，即将 GATMessagePass()函数改为如下这个函数：

```
#recbyhand\chapter5\s64_KGFM.py
#KGAT 由来的 FM 消息传递
def FMMessagePassFromKGAT(self, h_embs, r_embs, t_embs ):
    '''
    :param h_embs: 头实体向量[ batch_size, dim ]
    :param r_embs: 关系向量[ batch_size, n_neibours, dim ]
    :param t_embs: 尾实体向量[ batch_size, n_neibours, dim ]
    '''
    ##将 h 张量广播,维度扩散为 [ batch_size, n_neibours, dim ]
    h_broadcast_embs = torch.cat( [ torch.unsqueeze( h_embs, 1 ) for _ in range( t_embs.shape
    [ 1 ] ) ], dim = 1 )
    #[ batch_size, n_neibours, dim ]
    tr_embs = self.Wr( t_embs )
    #[ batch_size, n_neibours, dim ]
    hr_embs = self.Wr( h_broadcast_embs )
    #[ batch_size, n_neibours, dim ]
    hr_embs= torch.tanh( hr_embs + r_embs)
    #[ batch_size, n_neibours, dim ]
    hrt_embs = hr_embs * tr_embs
    #[ batch_size, dim ]
    square_of_sum = torch.sum(hrt_embs, dim=1) ** 2
    #[ batch_size, dim ]
    sum_of_square = torch.sum(hrt_embs ** 2, dim=1)
    #[ batch_size, dim ]
    output = square_of_sum - sum_of_square
    return output
```

还有需要注意的是，这次将头尾实体向量与关系向量的维度设为一样了，所以相应的 $\boldsymbol{W}_r$ 是一个正方形的变换矩阵。这么做的目的是为了将 $\boldsymbol{x}_i$ 的维度与头实体的维度一样，这样经 FM 的公式之后的 $\boldsymbol{e}_{\text{Nh}}$ 可以直接与头实体向量 $\boldsymbol{e}_h$ 进行进一步聚合。当然也不是非要一样，拼接聚合就不要求它们的维度一样。抑或是在求和之前，再进行线性变化调整维度，

但代价是增加模型参数了，但是学习用的示例代码还是简单为妙，直接限定 h、r 和 t 的维度一样即可。

再回过头想一想 KGCN，KGCN 的注意力计算方式在大多数情况下比 KGAT 更加优秀，因为它计算的是用户相对于指定关系的偏好权重并以此作为注意力权重，以 KGCN 形式计算的 $\boldsymbol{x}_i$ 的公式如下：

$$\boldsymbol{x}_i = f_{\mathrm{KGCN}}(\boldsymbol{u}, \boldsymbol{r}_i, \boldsymbol{t}_i) = \mathrm{Sotfmax}(\boldsymbol{u}\boldsymbol{r}_i) \times \boldsymbol{t}_i \tag{5-53}$$

这样计算之后的 $\boldsymbol{x}_i$ 实际上是经过注意力权重更新过后的尾实体向量，然后代入式(5-51)进行 FM 聚合似乎更符合 FM 特征交叉计算的本意。通过 KGCN 由来的 FM 消息传递的代码如下：

```
#recbyhand\chapter5\s64_KGFM.py
#KGCN 由来的 FM 消息聚合
def FMMessagePassFromKGCN( self, u_embs, r_embs, t_embs ):
    '''
    :param u_embs: 用户向量[ batch_size, dim ]
    :param r_embs: 关系向量[ batch_size, n_neibours, dim ]
    :param t_embs: 尾实体向量[ batch_size, n_neibours, dim ]
    '''
    #将用户张量广播,维度扩散为 [ batch_size, n_neibours, dim ]
    u_broadcast_embs = torch.cat( [ torch.unsqueeze( u_embs, 1 ) for _ in range( t_embs.shape
    [ 1 ] ) ], dim = 1 )
    #[ batch_size, n_neighbor ]
    ur_embs = torch.sum( u_broadcast_embs * r_embs, dim = 2 )
    #[ batch_size, n_neighbor ]
    ur_embs = torch.Softmax(ur_embs, dim = -1)
    #[ batch_size, n_neighbor, 1 ]
    ur_embs = torch.unsqueeze( ur_embs, 2 )
    #[ batch_size, n_neighbor, dim ]
    t_embs = ur_embs * t_embs
    #[ batch_size, dim ]
    square_of_sum = torch.sum( t_embs, dim = 1 ) ** 2
    #[ batch_size, dim ]
    sum_of_square = torch.sum( t_embs ** 2, dim = 1)
    #[ batch_size, dim ]
    output = square_of_sum - sum_of_square
    return output
```

更多的内容可查看完整 KGFM 的示例代码。

5.7　本章总结

本章学习了知识图谱嵌入的基础知识，以及一些具有代表性的知识图谱推荐算法。最后一节介绍了与图神经网络相结合的推荐算法。可以发现作者在创造这些算法时，是一个从原有算法结合一些新的知识从而推演出算法的过程。最后的 KGFM 相信应该能够给大

家积累些推演算法的经验。

当然对 KGFM 进行优化的空间还很大,例如加入与知识图谱嵌入的联合训练,甚至还可以与序列推荐联合训练。例如将用户的历史交互物品序列同时输入图消息传递层及序列神经网络层中,而图本身也能优化,例如将用户与物品的交互也作为知识图谱中的一个三元组事实,它们之间的交互是它们的关系。

相比单纯的基于图神经网络的推荐算法,基于知识图谱的推荐算法在工业界会更易落地。因为在实际工作中如果要将数据以图结构的形式存储,则通常以知识图谱的形式。基于图神经网络的推荐算法可以视为知识图谱推荐算法的基础知识,或者简化。

注意是在知识图谱的推荐算法体系中加入图神经网络的应用,而不是在图神经网络的推荐算法体系中加入知识图谱。知识图谱推荐算法的发展历史远长于图神经网络,并且某些基础知识(例如知识图谱嵌入和路径相似度等)至今仍然很实用。在基础知识扎实的前提下融入新颖的图神经网络,以此创造更有效的算法,这是本书想带给大家的思路。

参考文献