第 3 章

进阶推荐算法

本章会介绍如今深度学习时代下热门的几个经典算法，以及如何推导深度学习推荐算法模型。笔者认为推荐算法工程师切勿生搬硬套前沿论文的模型。一定要有自己对于目前推荐场景的思路，学习他人的算法模型是为了帮助自己更好地建立思路，而不是单纯地搬运他人算法在自己的场景中使用。

因为对于推荐算法而言，至少现在并不存在一个万金油的模型可以覆盖一切场景，而是需要推荐算法工程师结合当前场景及当前的数据，结合前沿及经典算法的思路推导出最合适的推荐算法。

身处当代的大家，是否觉得目前正是推荐算法百家争鸣的年代呢？网上随便一查就可以查到诸多推荐算法，但其实如果纵观历史，每个年代当时的学问无不处于百家争鸣的状态，而之所以会对当代百家争鸣的状态更有印象，正是因为我们是局中人。

3.1 神经网络推荐算法推导范式

1992 年，协同过滤问世之后，直到 2003 年才出现 ItemCF 算法，这 11 年间没有别的推荐算法吗？当然不是，只是经过多年的沉淀，成为经典的只有协同过滤而已，而今天众多的推荐算法也一样，20 年后能流传下去的也不会太多，所以对于学习算法的大家，一定要在基础巩固的前提下，形成推导算法的范式。

自深度学习问世以来，神经网络的概念也随之而来。神经网络复杂吗？当然复杂，因为神经网络直接将算法的层次拉深且拉宽，动辄就会有好几个层级，以及无数个神经元。神经网络简单吗？其实简单无比，了解了基本网络层的组成后，就会发现深度学习神经网络像搭积木一样。

3.1.1 ALS＋MLP

先从最基础的推荐算法 ALS 结合最基础的深度学习网络 MLP 开始讲解。首先，如果把 ALS 的模型结构画成神经网络图，则如图 3-1 所示。

多层感知机(MultiLayer Perceptron，MLP)[2]是深度学习的开端，简单理解是在最终计算前，使向量经过一次或多次的线性投影及非线性激活从而增加模型的拟合度。ALS 结合 MLP 的神经网络如图 3-2 所示。

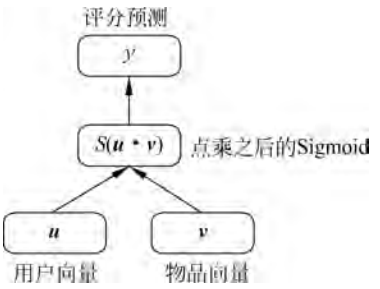


图 3-1 ALS 神经网络示意图

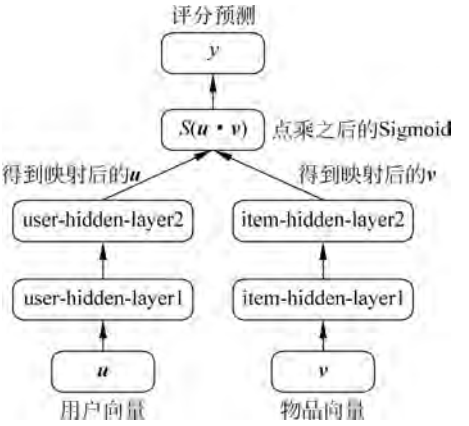


图 3-2 ALS-MLP 示意图

其中,每个隐藏层(Hidden Layer)都由一个线性层和非线性激活层组成。线性层是 $y=w \cdot x$ 的一个线性方程的形式,非线性激活层类似于 Sigmoid、ReLU 和 Tanh 等激活函数。线性层加非线性激活层的组合又被称为全连接层(Dense Layer)。

基础知识——非线性激活层的意义

这里顺便提一下,如果没有非线性激活单元,则多层的线性层是没有意义的。假设有 l 个隐藏线性层,则 x 经过多层投影得到第 l 层输出的这一过程可被描述为式(3-1)。

$$\begin{cases} h^1 = w^0 \cdot x \\ h^2 = w^1 \cdot h^1 \\ \cdots \\ h^l = w^{l-1} \cdot h^{l-1} \end{cases} \tag{3-1}$$

如果将此公式稍微改变一下形式,则可得

$$h^l = w^{l-1} \cdots w^1 w^0 x \tag{3-2}$$

所以可以发现,中间隐藏层的输出全部都没有意义,$w^{l-1} \cdots w^1 w^0$ 这些计算最终只是输出一个第 l 层的权重 w^l,因为永远进行的是线性变换,所以多次线性变换完全可由一次合适的线性变化代替,所以这个多层的神经网络与只初始化一个 w^l 的单层神经网络其实并无差别,而加入非线性激活单元就不一样了,通常由 $\sigma(\cdot)$ 表示一次非线性激活计算,所以 l 个隐藏层的公式就必须写成:

$$\begin{cases} h^1 = \sigma(w^0 \cdot x) \\ h^2 = \sigma(w^1 \cdot h^1) \\ \cdots \\ h^l = \sigma(w^{l-1} \cdot h^{l-1}) \end{cases} \tag{3-3}$$

如此一来,多层的神经网络就变得有意义了。

接下来是 ALS+MLP 的核心代码部分，代码如下：

```
#代码的地址：recbyhand\chapter3\s11a_ALS_MLP.py
class ALS_MLP (nn.Module):
    def __init__(self, n_users, n_items, dim):
        super(ALS_MLP, self).__init__()
        '''
        :param n_users: 用户数量
        :param n_items: 物品数量
        :param dim: 向量维度
        '''
        #随机初始化用户的向量
        self.users = nn.Embedding( n_users, dim, max_norm = 1 )
        #随机初始化物品的向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )

        #初始化用户向量的隐含层
        self.u_hidden_layer1 = self.dense_layer(dim, dim //2)
        self.u_hidden_layer2 = self.dense_layer(dim//2, dim //4)

        #初始化物品向量的隐含层
        self.i_hidden_layer1 = self.dense_layer(dim, dim //2)
        self.i_hidden_layer2 = self.dense_layer(dim//2, dim //4)

        self.sigmoid = nn.Sigmoid()

    def dense_layer(self,in_features,out_features):
        #每个 MLP 单元包含一个线性层和非线性激活层，当前代码中非线性激活层采取 Tanh 双曲
        #正切函数
        return nn.Sequential(
            nn.Linear(in_features, out_features),
            nn.Tanh()
        )

    def forward(self, u, v, isTrain = True):
        '''
         :param u: 用户索引 id shape:[batch_size]
         :param i: 用户索引 id shape:[batch_size]
         :return: 用户向量与物品向量的内积 shape:[batch_size]
         '''
        u = self.users(u)
        v = self.items(v)
        u = self.u_hidden_layer1(u)
        u = self.u_hidden_layer2(u)
        v = self.i_hidden_layer1(v)
        v = self.i_hidden_layer2(v)
```

```
        #训练时采取 DropOut 来防止过拟合
        if isTrain:
            u = F.DropOut(u)
            v = F.DropOut(v)

        uv = torch.sum( u * v, axis = 1)
        logit = self.sigmoid(uv * 3)
        return logit
```

深度学习拟合度高,所以更要注意过拟合的问题,最常用且有效的手段是在适当的位置放 DropOut 操作,例如上面代码倒数第 5 行和第 6 行。DropOut 是将向量随机丢弃若干个值,默认的比例是 0.5,即 50%的元素会被丢弃,所谓丢弃是将数值设为 0。在训练时采取 DropOut 有增添噪声的效果,会让预测不容易接近真实值,所以可以防止过拟合,而验证时不需要这个操作。

上述的操作是将用户与物品向量分别进行几个隐藏层的映射之后最后进行点乘计算。也可以将用户与物品向量拼接之后再进行 MLP 的传播,如图 3-3 所示。

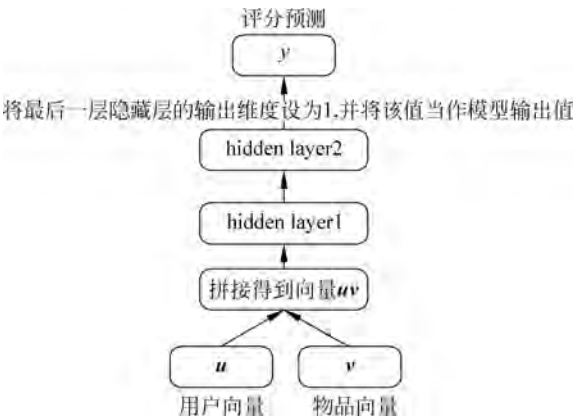


图 3-3 向量拼接之后的 MLP

该结构的代码如下:

```
#recbyhand\chapter3\s11b_ALS_CONCAT.py
class ALS_MLP ( nn.Module ):
    def __init__( self, n_users, n_items, dim ):
        super( ALS_MLP, self).__init__()
        '''
        :param n_users: 用户数量
        :param n_items: 物品数量
        :param dim: 向量维度
        '''
        #随机初始化用户的向量
```

```
        self.users = nn.Embedding( n_users, dim, max_norm = 1 )
        #随机初始化物品的向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )
        #第一层的输入的维度是向量维度乘以 2,因为用户与物品拼接之后的向量维度自然是原
        #来的 2 倍
        self.denseLayer1 = self.dense_layer( dim * 2, dim )
        self.denseLayer2 = self.dense_layer( dim , dim //2 )
        #最后一层的输出维度是 1,该值经 Sigmoid 激活后即为模型输出
        self.denseLayer3 = self.dense_layer( dim //2, 1 )
        self.sigmoid = nn.Sigmoid()

    def dense_layer(self, in_features, out_features):
        #每个 MLP 单元包含一个线性层和非线性激活层,当前代码中非线性激活层采取 Tanh 双曲
        #正切函数
        return nn.Sequential(
            nn.Linear(in_features, out_features),
            nn.Tanh()
        )

    def forward(self, u, v, isTrain = True):
        '''
         :param u: 用户索引 id shape:[batch_size]
         :param i: 用户索引 id shape:[batch_size]
         :return: 用户向量与物品向量的内积 shape:[batch_size]
         '''
        #[batch_size, dim]
        u = self.users( u )
        v = self.items( v )
        #[batch_size, dim * 2]
        uv = torch.cat([ u, v ], dim = 1)
        #[batch_size, dim]
        uv = self.denseLayer1( uv )
        #[batch_size, dim//2]
        uv = self.denseLayer2( uv )
        #[batch_size,1]
        uv = self.denseLayer3( uv )
        #训练时采取 DropOut 来防止过拟合
        if isTrain:uv = F.DropOut( uv )
        #[batch_size]
        uv = torch.squeeze( uv )
        logit = self.sigmoid( uv )
        return logit
```

这种先拼接后传播的方式其实在大多数情况下比先传播后点乘的方式好,其原因也不难理解,因为向量拼接后一起做线性投影会更充分地将向量之间(用户和物品之间)的交互关系学到,但是用户物品各自先经过 MLP 的传递,之后点乘的这种做法也有它的好处,其

实这种结构被称为“双塔模型”,涉及召回层和粗排序层等概念,这一概念会在第 6 章详细说明。在本节想告诉大家的是,神经网络的构造可以结合数理及深度学习的基础知识在任意位置像搭积木一样添加网络层和神经元等。

16min

3.1.2 特征向量+MLP

以上的 ALS 算法仅考虑了用户与物品交互的情况,如果每个用户都很活跃,并且每个物品都很热门,则这样的 ALS 自然就会学得很好,但是在实际工作中通常不会存在这么理想的数据,另一个策略是通过活跃用户的数据学到用户特征与物品特征之间对应用户物品交互情况的模型,从而可以通过特征泛化到非活跃用户。

总而言之,如果加入了特征模型该怎么做? 其实也非常简单,如图 3-4 所示。

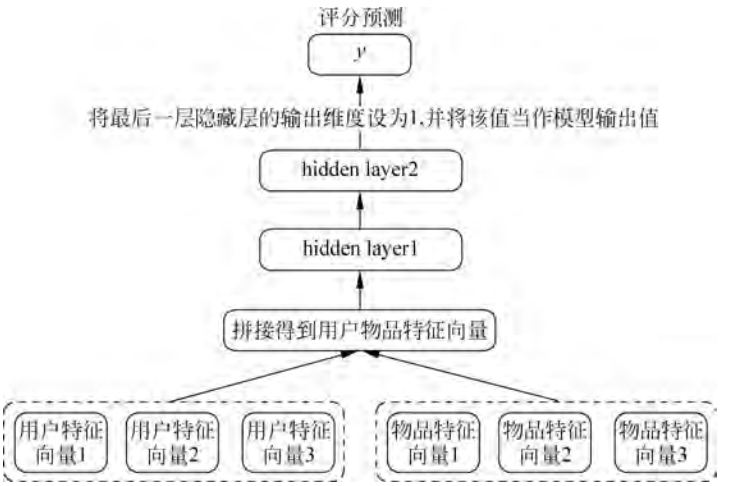


图 3-4 用户物品特征向量拼接之后的 MLP

相比图 3-3 的结构,图 3-4 是在最底下一层由原来的用户向量及物品向量变为用户的特征向量及物品的特征向量。

在实际操作时,需先将用户及物品的特征编码,此处仅需硬索引编码,如图 3-5 所示的用

	C1	C2	C3	C4
1	<null>	age	gender	occupation
2	1	2	8	29
3	2	5	9	23
4	3	2	8	30
5	4	2	8	29
6	5	3	9	23
7	6	4	8	16
8	7	5	8	10
9	8	3	8	10
10	9	2	8	28
11	10	5	8	19

图 3-5 用户特征索引

户特征索引。第一栏的数字是用户 id,其余每个数字都代表某个值所在模型中 Embedding 层的索引。

本节的代码地址为 recbyhand\chapter3\s12_Embedding_mlp.py,其中的核心部分代码如下:

```
#recbyhand\chapter3\s12_Embedding_mlp.py,
class embedding_mlp( nn.Module ):

    def __init__( self, n_user_features, n_item_features, user_df, item_df, dim = 128 ):
        super( embedding_mlp, self ).__init__()
        #随机初始化所有特征的特征向量
        self.user_features = nn.Embedding( n_user_features, dim, max_norm = 1 )
        self.item_features = nn.Embedding( n_item_features, dim, max_norm = 1 )
        #记录好用户和物品的特征索引
        self.user_df = user_df
        self.item_df = item_df

        #得到用户和物品特征的数量的和
        total_neighbours = user_df.shape[1] + item_df.shape[1]

        #定义 MLP 传播的全连接层
        self.dense1 = self.dense_layer( dim * total_neighbours, dim * total_neighbours//2 )
        self.dense2 = self.dense_layer( dim * total_neighbours//2 , dim )
        self.dense3 = self.dense_layer( dim, 1)
        self.sigmoid = nn.Sigmoid()

    def dense_layer(self,in_features,out_features):
        return nn.Sequential(
            nn.Linear(in_features, out_features),
            nn.Tanh()
        )

    def forward(self, u, i, isTrain = True):
        user_ids = torch.LongTensor(self.user_df.loc[u].values)
        item_ids = torch.LongTensor(self.item_df.loc[i].values)
        #[batch_size, user_neighbours, dim]
        user_features = self.user_features(user_ids)
        #[batch_size, item_neighbours, dim]
        item_features = self.item_features(item_ids)

#将用户和物品特征向量拼接起来
        #[batch_size, total_neighbours, dim]
        uv = torch.cat( [user_features, item_features] ,dim = 1)

        #将向量平铺以方便后续计算
```

```
        #[batch_size, total_neighbours * dim]
        uv = uv.reshape((len(u), -1))

    #开始 MLP 的传播
        #[batch_size, total_neighbours * dim//2]
        uv = self.dense1(uv)
        #[batch_size, dim]
        uv = self.dense2(uv)
        #[batch_size, 1]
        uv = self.dense3(uv)
        #训练时采取 DropOut 来防止过拟合
        if isTrain: uv = F.DropOut(uv)
        #[batch_size]
        uv = torch.squeeze(uv)
        logit = self.sigmoid(uv)
        return logit
```

完整代码中会有训练及测试过程。大家也可尝试着去改变一下模型结构或调整一些超参来观察评估指标的变化。

3.1.3 结合 CNN 的推荐

3.1.2 节中有一步向量拼接的操作,假设某用户特征向量与某物品特征向量的总数量为 n,每个特征的维度为 k,拼接之后可以得到一个维度为 $n \times k$ 的一维向量。该向量似乎显得有点长,是否有更美观的特征向量聚合方式呢? 当然有且有很多。其中最简单的自然是卷积神经网络。

此次将特征向量拼接成一个形状为 $n \times k$ 的二维矩阵取代之前的一维长向量,这样就可对该矩阵进行卷积操作,从而达到特征向量聚合的效果,然后继续经过几层全连接层最后得到模型输出,该过程如图 3-6 所示。

本节代码的地址为 recbyhand\chapter3\s13 _CNN_rec.py,核心部分代码如下:

```
#recbyhand\chapter3\s13 _CNN_rec.py
class embedding_CNN( nn.Module ):

    def __init__( self, n_user_features, n_item_features, user_df, item_df, dim = 128 ):
        super( embedding_CNN, self ).__init__()
        #随机初始化所有特征的特征向量
        self.user_features = nn.Embedding( n_user_features, dim, max_norm = 1 )
        self.item_features = nn.Embedding( n_item_features, dim, max_norm = 1 )
        #记录好用户和物品的特征
        self.user_df = user_df
        self.item_df = item_df
```

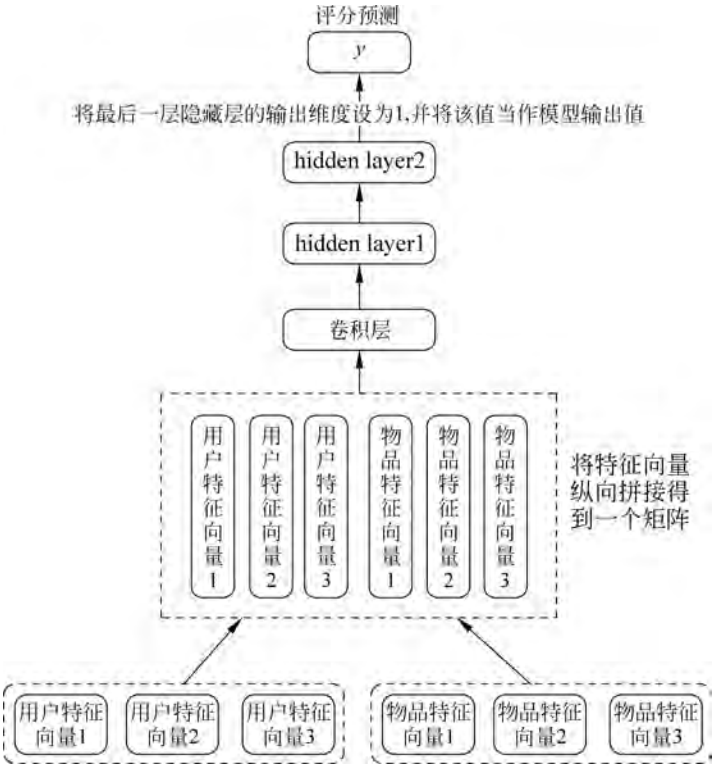


图 3-6　加入对向量拼接矩阵进行卷积操作的神经网络

```
        #得到用户和物品特征的数量的和
        total_neighbours = user_df.shape[1] + item_df.shape[1]

        self.Conv = nn.Conv1d( in_channels = total_neighbours, out_channels = 1, Kernel_size = 3 )

        #定义 MLP 传播的全连接层
        self.dense1 = self.dense_layer( dim - 2, dim//2 )
        self.dense2 = self.dense_layer( dim//2 , 1 )

self.sigmoid = nn.Sigmoid()

    def dense_layer( self, in_features, out_features ):
        return nn.Sequential(
            nn.Linear( in_features, out_features ),
            nn.Tanh()
        )

    def forward(self, u, i, isTrain = True):
        user_ids = torch.LongTensor(self.user_df.loc[u].values)
```

```
        item_ids = torch.LongTensor(self.item_df.loc[i].values)
        #[batch_size, user_neighbours, dim]
        user_features = self.user_features(user_ids)
        #[batch_size, item_neighbours, dim]
        item_features = self.item_features(item_ids)

        #将用户和物品特征向量拼接起来
        #[batch_size, total_neighbours, dim]
        uv = torch.cat( [user_features, item_features] ,dim = 1)

        #[batch_size, 1, dim + 1 - Kernel_size]
        uv = self.Conv(uv)
        #[batch_size, dim + 1 - Kernel_size]
        uv = torch.squeeze(uv)

        #开始 MLP 的传播
        #[batch_size, dim//2]
        uv = self.dense1(uv)
        #[batch_size, 1]
        uv = self.dense2(uv)

        #训练时采取 DropOut 来防止过拟合
        if isTrain: uv = F.DropOut(uv)
        #[batch_size]
        uv = torch.squeeze(uv)
        logit = self.sigmoid(uv)
        return logit
```

其实以上代码是在 3.1.3 节代码的基础上改动了特征向量拼接的部分及加入了一个卷积层。

相对于提取平铺拼接后的特征向量,对特征进行 CNN 卷积提取的优势在于能保留更多方向的信息。

3.1.4 结合 RNN 的推荐

14min

有了 CNN,自然就会想到 RNN。RNN 的优势在于它是一个序列模型,最直观的感受是能够很好地利用时间上的信息。图 3-7 是一个 RNN 的基本示意图。

13min

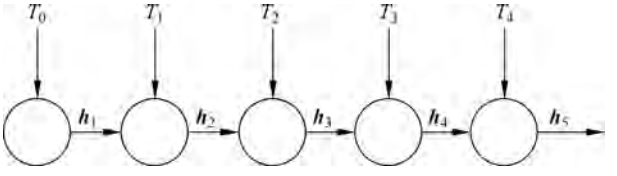


图 3-7 RNN 的基本示意图

每个 RNN 节点的公式如下[3]：

$$\boldsymbol{h}_t = \tanh(\boldsymbol{w}_{ih}\boldsymbol{x}_t + \boldsymbol{b}_{ih} + \boldsymbol{w}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hh}) \tag{3-4}$$

其中，$\boldsymbol{w}_{ih}\boldsymbol{x}_t + \boldsymbol{b}_{ih}$ 是一个基本的线性回归方程，$\boldsymbol{x}_t$ 是 t 时刻的输入，假设训练序列数据是用户对于物品的历史观看记录，则输入是 t 时刻的物品。RNN 节点比全连接层多出的内容是 $\boldsymbol{w}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hh}$，其中 $\boldsymbol{h}_{t-1}$ 是上一个 RNN 节点的输出。如此一来直到输入最后一个物品，都不会丢失前面所有物品的信息，并且先后顺序的信息也保留了。PyTorch 中 RNN 的代码如下：

```
import torch
from torch import nn

rnn = nn.RNN( input_size = 12, hidden_size = 6, batch_first = True)
input = torch.randn(24, 5, 12)
outputs, hn = rnn(input)
```

其中几个重要的参数如下。

input_size：输入特征维度，也是公式中 $\boldsymbol{x}$ 的向量维度。

hidden_size：隐含层特征维度，也是公式中 $\boldsymbol{h}$ 的维度。有了这两个维度后，公式中 $\boldsymbol{w}_{ih}$ 和 $\boldsymbol{w}_{hh}$ 等的形状都能自动计算得到。

batch_first：是否第一维表示 batch_size。这个要结合下面的 input 细说，input 在这里随机初始化了一个形状为(24,5,12) 的张量，batch_first＝True 意味着第 1 个数字代表 batch_size，此时 input 的意义是一批数据中有 24 个序列样本，每个序列样本有 5 个物品，每个物品的特征向量维度为 12。在 nn.RNN 这种方法中 batch_first 的默认值为 False，它默认会把输入张量的第 2 个数字当作 batch_size，而将第 1 个数字当作序列长度，这是要注意的参数。

可以看到代码中 rnn()有两个输出，一个是 outputs，另一个是 hn，hn 其实是第 n 层的输出也是最后一层的输出，形状是(1，batch_size，hidden_size)，当前数据环境中是(1，24，6)，而 outputs 其实是每个 RNN 节点的输出，形状是(batch_size，序列长度，hidden_size)，当前的数据环境下是(24,5,6)。

如果把 batch_first 设为 False，则 outputs 的形状是(序列长度，batch_size，hidden_size)这么做的好处是和 hn 有更统一的形状，因为 hn 其实是 outputs 中的最后一个序列向量，如果 batch_first＝False，则 hn＝outputs[－1]。否则 hn 要和 outputs[－1]对等就要调整形状。当然这得看个人习惯，本书的示例代码会将 batch_first 设为 True，因为毕竟将张量理解为 batch_size 更符合大多数情况的习惯认知。

MovieLens 的数据集有时间戳的信息，所以可以将数据整理成序列的形式，以便下游的任务使用此数据，示例数据如下：

```
[1973,5995,560,5550,6517,4620,1],
[5995,560,5550,6517,4620,4563,1],
[560,5550,6517,4620,4563,1314,1],
[2439,1600,7999,1743,8282,8204,0]
```

每一行的序列由 6 个物品 id 及一个 0 或 1 的标注组成。其中前 5 个物品 id 代表用户最近历史单击的物品 id,第 6 个物品 id 代表用户当前观看的物品,第 7 位的标注代表用户对第 6 个物品 id 真实喜欢的情况,1 为喜欢,0 为不喜欢。根据 MovieLens 原始数据得到序列的脚本地址为 recbyhand\chapter3\s14_RNN_data_prepare.py。

利用 RNN 做推荐算法的思路如图 3-8 所示。

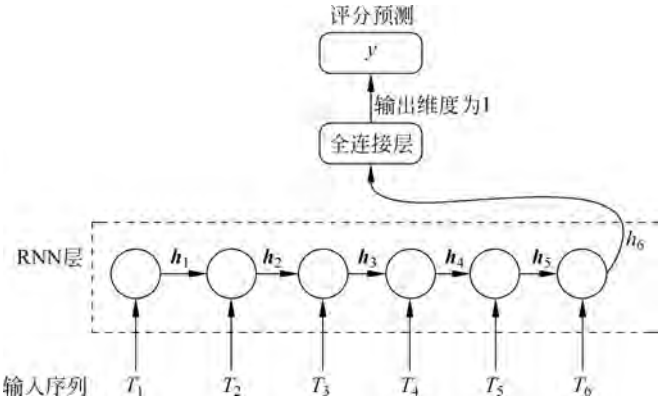


图 3-8 结合 RNN 推荐的基础思路

取 RNN 最后一个节点的输出向量(如前文实例代码中的 hn),作为下一个全连接层的输入,之后就是 MLP 的模式了。图 3-8 中虽然只有一个全连接层,但是在实际工作中大家可以多加几层尝试一下。

基于 RNN 的推荐模型的核心代码如下:

```
#recbyhand\chapter3\s14_RNN_rec.py
class RNN_rec( nn.Module ):

    def __init__( self, n_items, hidden_size = 64, dim = 128):
        super( RNN_rec, self ).__init__()
        #随机初始化所有物品向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )
        self.rnn = nn.RNN( dim, hidden_size, batch_first = True )
        self.dense = self.dense_layer( hidden_size, 1 )
        self.sigmoid = nn.Sigmoid()

    #全连接层
    def dense_layer(self,in_features,out_features):
```

```
        return nn.Sequential(
          nn.Linear(in_features, out_features),
          nn.Tanh())

    def forward(self, x, isTrain = True):
        #[batch_size, len_seqs, dim]
        item_embs = self.items(x)
        #[1, batch_size, hidden_size]
        _,h = self.rnn(item_embs)
        #[batch_size, hidden_size]
        h = torch.squeeze(h)
        #[batch_size, 1]
        out = self.dense(h)
        #训练时采取 DropOut 来防止过拟合
        if isTrain: out = F.DropOut(out)
        #[batch_size]
        out = torch.squeeze(out)
        logit = self.sigmoid(out)
        return logit
```

大家也许会有一个疑问，这里仅仅将用户的历史物品序列通过 RNN 网络训练了一遍，对推荐能有效果吗？当然有，RNN 的优势就在于在消息传递的过程中能保留顺序信息，所以把 RNN 层当作聚合用户的历史物品序列信息的函数会更好理解，也就是说 RNN 最后一层的输出向量可以认为是用户向量与物品向量的拼接。如果还是不太理解，则 3.1.5 节有助于更好地理解 RNN 对推荐作用的原理。

3.1.5　ALS 结合 RNN

大家对 ALS 已经再熟悉不过了，中心思想是求用户向量与物品向量的点积。求点积不是重点，重点是用户向量与物品向量这个概念。别忘了 ALS 还有另一个名字叫作 LFM，即隐因子模型，用向量表示用户或者物品才是这个思路的核心思想。

10min

在 RNN 的计算环境中，可以将用户历史交互的物品序列当作用户本身。以此聚合那些物品序列的向量，然后与目标物品的向量表示进行点积运算，从而建立损失函数，该过程如图 3-9 所示。

与 3.1.4 节不同的是，这次的结构是将目标物品单提出后作为 ALS 思路上所谓的物品向量，而前 5 个物品组成的序列经 RNN 层消息传递后，得到的最后一层输出当作用户向量。之后将得到的用户向量与物品进行点积等操作即可，代码如下：

```
#recbyhand\chapter3\s15_RNN_rec_ALS.py
class RNN_ALS_rec( nn.Module ):
```

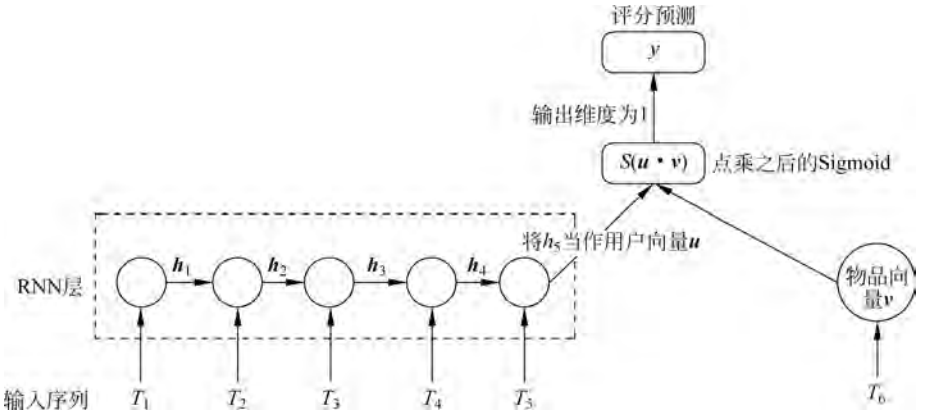


图 3-9　隐因子模型思路下的 RNN 推荐算法

```
def __init__( self, n_items, dim = 128):
    super( RNN_ALS_rec, self ).__init__()
    #随机初始化所有物品的特征向量
    self.items = nn.Embedding( n_items, dim, max_norm = 1 )
    #因为要进行向量点积运算,所以 RNN 层的输出向量维度也需与物品向量一致
    self.rnn = nn.RNN( dim, dim, batch_first = True )
    self.sigmoid = nn.Sigmoid()

def forward(self, x, item):
    #[batch_size, len_seqs, dim]
    item_embs = self.items(x)
    #[1, batch_size, dim]
    _,h = self.rnn(item_embs)
    #[batch_size, dim]
    h = torch.squeeze(h)
    #[batch_size, dim]
    one_item = self.items(item)
    #[batch_size]
    out = torch.sum(h * one_item, dim = 1)
    logit = self.sigmoid(out)
    return logit
```

该做法虽然从结果上比 3.1.4 节的算法更容易理解,但其实效果是差不多的。因为 3.1.4 节的 RNN 层也是消息聚合的作用而已,但就算仅仅这“而已”的作用在通常情况下推荐的帮助也会大于 CNN。

CNN 在处理计算机视觉领域很优秀,因为图像数据本身是一张张二维的矩阵,而 CNN 对二维矩阵的特征提取能力相当优秀,所以在推荐领域如果有本身是二维矩阵信息的数据,则 CNN 自然也会有很优秀的作用,但现在演示的仅仅是把特征拼接成二维矩阵再由 CNN 进行特征提取。

而在推荐领域中能体现 RNN 优势的数据就多了，因为用户与物品发生交互时总有先后顺序，自然总能形成序列。RNN 的潜力还不仅如此，经 nn.rnn()函数传播后还会有一个 outputs 输出，对于这个 outputs 该怎么使用，可参看 3.1.6 节。

3.1.6　联合训练的 RNN

好多读者是通过语言模型(Language Model)了解的 RNN。通过输入很多语料，让 RNN 模型学习预测输入单词的下一个单词的能力。例如输入“今天下雨了”这句话，则通过“今”可预测“天”，通过“天”可预测“下”，通过“下”可预测“雨”，以此类推，经 RNN 的传播如图 3-10 所示。

25min

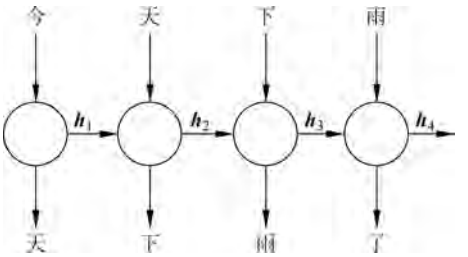


图 3-10　RNN 语言模型示意图

单纯的一个 RNN 节点输出的只是一个隐藏向量 $\boldsymbol{h}$，如要完成一个字的预测还需要接一个全连接层，这个全连接层的输出维度是类别数，在语言模型中类别数是所有词或所有字的个数。在推荐场景中类别数是所有候选物品的数量。

将输入的向量再进行 Softmax 激活一下，然后与真实的类别建立交叉熵损失函数，以便进行多分类预测，公式如下：

$$\begin{cases} \hat{y}_t = \text{Softmax}(\boldsymbol{w}_{hn}\boldsymbol{h}_t + b_{hn}) \\ \text{loss} = \sum_{0}^{t} \text{crossEntropyLoss}(\hat{y}_t, y_t) \end{cases} \tag{3-5}$$

在当前场景中，用“今天下雨”这一条序列，预测了“天下雨了”这一序列，其中一共有 4 个时刻的预测值，损失值是 4 个损失函数的叠加。

在推荐场景中该怎么用呢？假设用户历史物品交互序列是：[物品 1，物品 2，物品 3，物品 4，物品 5]，则完全可以建立一个 RNN 网络并输入 [物品 1，物品 2，物品 3，物品 4]，以便预测 [物品 2，物品 3，物品 4，物品 5]。这么做的意义在于可以产生一个序列预测序列的损失函数，从而辅助推荐算法，其实利用这一过程辅助推荐算法的方式有很多，但是为了保持本章简单的宗旨，不介绍太多容易混淆且不好理解的内容，在此介绍一个最基础的思想，即联合训练的 RNN。

输入物品序列，预测错开一位的物品序列这一过程本身是一个模型，而妙就妙在该 RNN 网络最后一个节点的输出可以当作聚合后的用户 Embedding 与候选物品 Embedding 去进行 CTR 预估。因为反向传播序列预测序列的模型与反向传播 CTR 预估的模型迭代

的是同一个 RNN 网络，于是自然会有相辅相成的效果，该过程如图 3-11 所示。

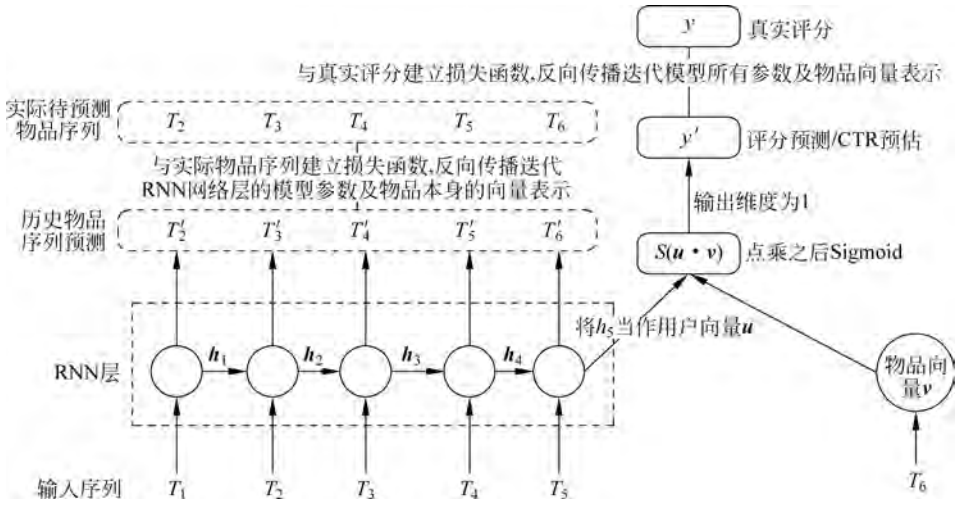


图 3-11　联合训练的 RNN

核心代码如下：

```
#recbyhand\chapter3\s16_RNN_rec_withPredictHistorySeq.py
class RNN_rec( nn.Module ):

    def __init__( self, n_items, dim = 128):
        super( RNN_rec, self ).__init__()
        self.n_items = n_items
        #随机初始化所有特征的特征向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )
        self.rnn = nn.RNN( dim, dim, batch_first = True )

        #初始化历史序列预测的全连接层及损失函数等
        self.ph_dense = self.dense_layer( dim, n_items )
        self.Softmax = nn.Softmax()
        self.crossEntropyLoss = nn.CrossEntropyLoss()

        #初始化推荐预测损失函数等
        self.sigmoid = nn.Sigmoid()
        self.BCELoss = nn.BCELoss()

    #全连接层
    def dense_layer(self,in_features,out_features):
        return nn.Sequential(
            nn.Linear(in_features, out_features),
            nn.Tanh())
```

```
    #历史物品序列预测的前向传播
    def forwardPredHistory( self, outs, history_seqs ):
        outs = self.ph_dense( outs )
        outs = self.Softmax( outs )
        outs = outs.reshape( -1, self.n_items )
        history_seqs = history_seqs.reshape( -1 )
        return self.crossEntropyLoss( outs, history_seqs )

    #推荐 CTR 预测的前向传播
    def forwardRec(self, h, item, y):
        h = torch.squeeze( h )
        one_item = self.items( item )
        out = torch.sum( h * one_item, dim = 1 )
        logit = self.sigmoid( out )
        return self.BCELoss( logit, y )

    #整体前向传播
    def forward( self, x, history_seqs, item, y ):
        '''
        :param x: 输入序列
        :param history_seqs: 要预测的序列,其实是与 x 错开一位的历史记录
        :param item: 候选物品序列
        :param y: 0 或 1 的标注
        :return: 联合训练的总损失函数值
        '''
        item_embs = self.items(x)
        outs, h = self.rnn(item_embs)
        hp_loss = self.forwardPredHistory(outs, history_seqs)
        rec_loss = self.forwardRec(h, item, y)
        return hp_loss + rec_loss

    #因为模型中 forward 函数输出的是损失函数值,所以另定义一个预测函数以方便预测及评估
    def predict( self, x, item ):
        item_embs = self.items( x )
        _, h = self.rnn( item_embs )
        h = torch.squeeze( h )
        one_item = self.items( item )
        out = torch.sum( h * one_item, dim = 1 )
        logit = self.sigmoid( out )
        return logit
```

这次代码有点长,在书中去掉了一些过程中张量形状的注释,当然本书配套代码中的注释是完整的。另外值得注意的是,这次代码模型的整体前向传播方式输出的是联合训练的整体损失函数,所以另定义了一个预测函数输出预测值以方便评估模型。

3.1.7 小节总结

3min

大家应该会发现每个模型结构是在前面模型的基础上改动或增加了某些元素。本书希望通过这个过程使大家可以了解到深度学习神经网络推荐算法推导的范式思想,其实算法推导很简单,使用一些基础的计算形式在不同的场景下发挥出不同的运用。相信大家都能举一隅而以三隅反,例如最后在联合训练 RNN 的基础上,再加上运用用户特征或者物品特征向量进一步泛化算法。大家也可以尝试一下 LSTM 或者 GRU 等在 RNN 基础上衍生出来的神经网络结构。

有了这些算法的推导能力后,再学习前沿的算法可以说是轻松愉悦,并且大家也应该很容易推导出自己的推荐算法。

3.2 FM 在深度学习中的应用

正如在前文所说,推荐系统有两大基石,一个是 ALS,而另一个是 FM。本章就来介绍一下由 FM 衍生出的深度学习模型。

3.2.1 FNN

4min

FNN 全称 Factorisation-machine supported Neural Networks[4],于 2016 年被提出。名字直译为用 FM 支持神经网络。通常认为 FNN 是学术界发表的第一个将 FM 运用在深度学习的模型,所以将 FNN 作为学习 FM 在深度学习中应用的入门模型很适合。原始的 FNN 思路很简单,即用 FM 得到的隐向量去初始化深度学习神经网络的 Embedding 输入。模型结构如图 3-12 所示。

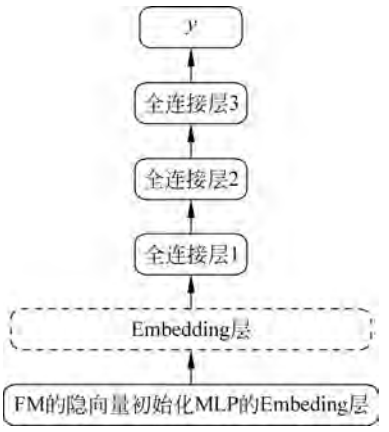


图 3-12 原始的 FNN 模型

以下是 FM 的公式:

$$\hat{y}=\sigma\left(w0+\sum_{i=1}^{n}w1_i x_i+\sum_{i=1}^{n}\sum_{j=i+1}^{n}(\boldsymbol{v}_i\cdot\boldsymbol{v}_j)x_i x_j\right) \tag{3-6}$$

经过 FM 训练后可以得到 $\boldsymbol{w}0$,$\boldsymbol{w}1$ 和 $\boldsymbol{v}$ 等模型参数,将这些模型参数初始化一个 MLP 神经网络的 Embedding 输入,是 FNN 的做法。MLP 的 Embedding 层不去迭代更新,而是利用 FM 的先验知识去训练 MLP 中那些全连接层的模型参数。反过来也可以说是利用 MLP 多层的网络结构进一步提高 FM 对数据的拟合度。

所以原始的 FNN 并不是端到端的训练,但是可以在 FNN 的基础上改进一下,使其变为端到端的训练。

3.2.2 改进后的 FNN

10min

本节的目的是要开发一个能够同时用到 FM 及 MLP 的端到端模型。首先来回顾一下

2.7.5 节中端到端训练下的 FM 二次项简化公式。

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}(\boldsymbol{v}_i \cdot \boldsymbol{v}_j)x_i x_j = \frac{1}{2}\sum_{f=1}^{k}\left(\left(\sum_{i=1}^{n_{\text{single}}^{(j)}} v_{i,f}^{(j)}\right)^2 - \sum_{i=1}^{n_{\text{single}}^{(j)}} v_{i,f}^{(j)\,2}\right) \tag{3-7}$$

该公式等号右边的 $\sum_{f=1}^{k}(\cdot)$ 是指将 k 个括号内的值累加，而如果不进行累加这一步，则括号内得到的值是一个维度为 k 的向量。括号内的计算具备了特征交叉的信息，外层的累加可以当作是给该向量做了一次求和池化，所以其实完全可以跳过累加这一步，直接将这个 k 维向量输入 MLP 网络中进行传播，最终得到预测值。

该操作被称为 FM 聚合层，记作：

$$\text{agg}_{\text{FM}}(x) = \left(\sum_{i=1}^{n_{\text{single}}} \boldsymbol{v}_i\right)^2 - \sum_{i=1}^{n_{\text{single}}} \boldsymbol{v}_i^2 \tag{3-8}$$

其中，n_{single} 是一次数据的特征数量，$\boldsymbol{v}_i$ 是 i 的特征向量，假设传递的数据是一个形状为[batch size，特征数量 n，特征维度 dim]的张量，则经过 FM 聚合层的传递之后就得到了[batch size，特征维度 dim] 的张量，该过程的代码如下：

```
#recbyhand\chapter3\s22_FNN_plus.py
def FMaggregator(self, feature_embs):
    #feature_embs:[batch_size, n_features, dim]
    #[batch_size, dim]
    square_of_sum = torch.sum( feature_embs, dim = 1) ** 2
    #[batch_size, dim]
    sum_of_square = torch.sum( feature_embs ** 2, dim = 1)
    #[batch_size, dim]
    output = square_of_sum - sum_of_square
    return output
```

整个改进后的 FNN 模型结构如图 3-13 所示。

图 3-13　改进后的 FNN 模型结构

这样就完全是一个端到端的深度学习模型了。

本节代码的地址为 recbyhand\chapter3\s22_FNN_plus.py。书中展示一下完整的FNN_plus模型类的代码：

```
#recbyhand\chapter3\s22_FNN_plus.py
class FNN_plus( nn.Module ):

    def __init__( self, n_features, dim = 128 ):
        super( FNN_plus, self ).__init__()
        #随机初始化所有特征的特征向量
        self.features = nn.Embedding(n_features, dim, max_norm = 1)
        self.mlp_layer = self.__mlp(dim)

    def __mlp( self, dim ):
        return nn.Sequential(
            nn.Linear(dim, dim //2),
            nn.Tanh(),
            nn.Linear(dim //2, dim //4),
            nn.Tanh(),
            nn.Linear(dim //4, 1),
            nn.Sigmoid()
        )

    def FMaggregator(self, feature_embs):
        #feature_embs:[batch_size, n_features, dim]
        #[batch_size, dim]
        square_of_sum = torch.sum( feature_embs, dim = 1) ** 2
        #[batch_size, dim]
        sum_of_square = torch.sum( feature_embs ** 2, dim = 1)
        #[batch_size, dim]
        output = square_of_sum - sum_of_square
        return output

    #把用户和物品的特征合并起来
    def __getAllFeatures( self,u, i, user_df, item_df ):
        users = torch.LongTensor( user_df.loc[u].values )
        items = torch.LongTensor( item_df.loc[i].values )
        all = torch.cat( [ users, items ], dim = 1 )
        return all

    def forward(self, u, i, user_df, item_df):
        #得到用户与物品组合起来后的特征索引
        all_feature_index = self.__getAllFeatures( u, i, user_df, item_df )
        #取出特征向量
        all_feature_embs = self.features( all_feature_index )
```

```
        #[batch_size, dim]
        out = self.FMaggregator( all_feature_embs )
        #[batch_size, 1]
        out = self.mlp_layer(out)
        #[batch_size]
        out = torch.squeeze(out)
        return out
```

3.2.3　Wide & Deep

接下来是 Wide & Deep 模型，论文名叫作 Wide & Deep Learning for Recommender Systems[5]，于 2016 年由谷歌公司提出，又称作 WAD。从事推荐工作的人员多多少少听说过 FM，而听过 FM 的人员多多少少听说过 DeepFM，而 DeepFM 是由 Wide & Deep 演化而来，并且 Wide & Deep 在领域内的地位并不亚于 DeepFM，究其原因主要是谷歌在 TensorFlow 中有现成的 Wide & Deep API，所以在讲解 Deep FM 之前，Wide & Deep 还是很有必要讲解一下的。

图 3-14 展示了最原始的 Wide & Deep 模型结构图。

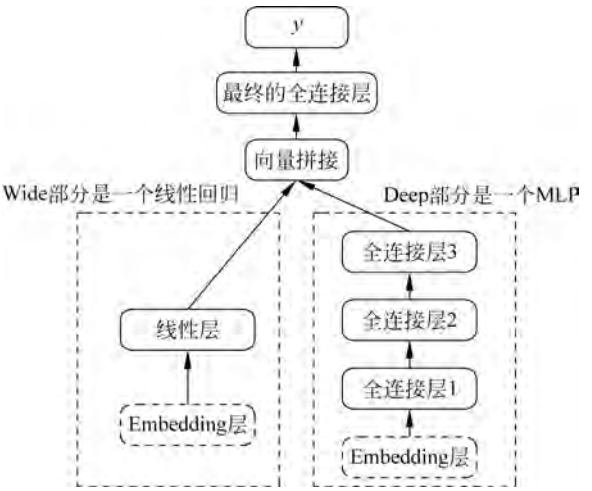


图 3-14　原始 Wide & Deep 模型结构图

Wide & Deep 是一个将基础的线性回归模型与 MLP 深度学习网络横向拼接的网络模型。这么做的好处是兼具“记忆能力”与“泛化能力”。

“记忆能力”是 Wide 部分的任务，所谓“记忆能力”是希望通过简单的操作来学到特征的表示，以此使该特征对结果的影响可以尽可能的直接。

“泛化能力”是 Deep 部分的任务，这部分功能是利用深度学习优秀的泛化能力来充分学习每个特征对结构的影响。有人会说网络越深不是越容易过拟合吗？的确如此，但一个合适的深度学习网络的泛化能力完全会高过简单的机器学习模型，更不用说有很多深度学

习常用的手段(如 DropOut)去防止过拟合。

什么特征应该更注重“记忆”,什么特征更应该注重“泛化”呢?按照最初的想法,交互性质的数据需要重点突出记忆能力,而用户物品的属性应该更需要突出泛化能力。因为交互数据往往是会动态变化的数据,需要捕捉到短期形成的记忆。例如在一个电影推荐场景中,统计用户观看最多的一个电影类型作为“用户观影类型偏好”特征,再例如将用户最近看过的五部电影作为“用户历史观影特征”,这些特征本身就强力代表了用户的兴趣取向,所以“记忆”这些特征自然对推荐更有帮助。

而用户物品本身的那些静态属性,如年龄、性别、职业等,乍看之下与推荐并不具备强相关的关系,但多多少少又感觉会有影响,所以将这些特征经神经网络泛化开来是再好不过的操作。

但是对以上那些特征例子,人为能够区分,如果碰到一些模棱两可的特征,人为很难区分应输入 Wide 部分还是 Deep 部分时该怎么办呢?其实目前业内更多的做法是直接将所有的特征同时输入 Wide 部分和 Deep 部分,让模型同时去学每个特征的“记忆”与“泛化”权重。模型当然具备这个能力,那些“记忆”要求不高的特征,它们的“记忆”权重自然不会高。

目前改进并更主流的 Wide & Deep 模型的 Wide 部分是一个特征间两两交叉相乘的计算,是 2.7.3 节中提到的 POLY2 算法,并且仅用到 POLY2 的二次项。模型结构如图 3-15 所示。

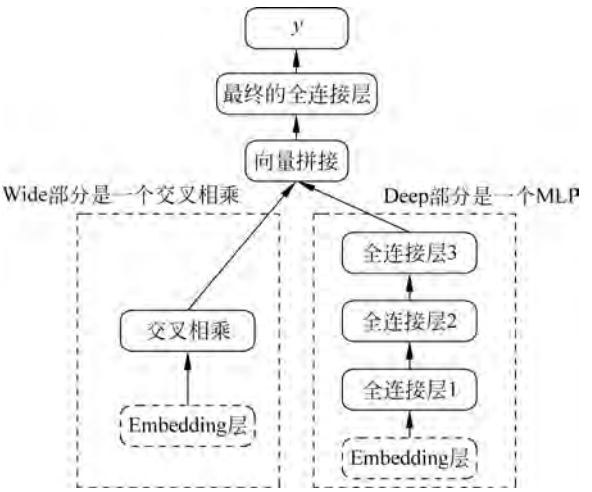


图 3-15 Wide & Deep

相比最初的 Wide & Deep,不同之处是把线性回归层换成了交叉相乘层。

既然可以用 POLY2 算法,那是不是可以把 Wide 部分直接用 FM 替换呢?当然可以,所以就形成了 DeepFM。

10min

3.2.4 DeepFM

DeepFM[6]是由华为公司在 2017 年提出的深度学习推荐算法模型,是将 FM 替换掉

Wide & Deep 的 Wide 部分。由于 FM 一贯的优越性使 DeepFM 瞬间流行起来。其实 DeepFM 的模型结构又可以视为一个横向的 FNN。模型结构如图 3-16 所示。

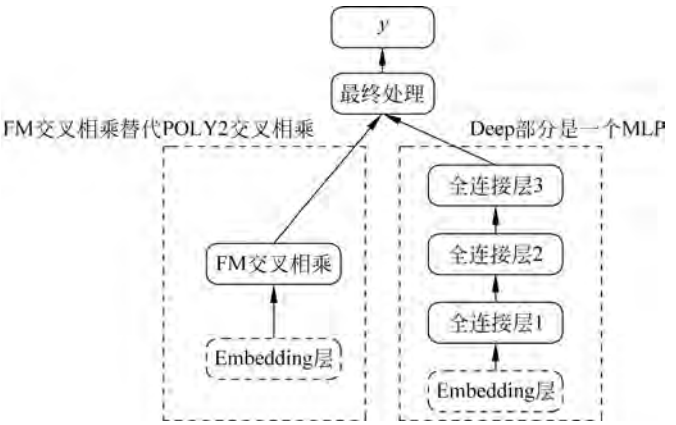


图 3-16　DeepFM 模型结构

图 3-16 中用了一个“最终处理”的方块代替原来画在 Wide & Deep 模型结构图中的向量拼接 + 最终全连接层的两个方块。因为这里的做法其实不止一个，当然向量拼接＋全连接层是做法之一，但最流行的做法其实还是将 FM 层的输出与 MLP 的输出直接相加，然后求 Sigmoid 函数。

因为 FM 二次项的公式默认输出的是一个一维标量，所以将 MLP 层中最后一个全连接层的输出维度设为 1，则两个一维标量相加求 Sigmoid 就可以作为 CTR 的预测值去与真实值建立损失函数了。

如果还是采用向量拼接的方式，FM 层则可以用前文在 FNN 章节中提到的式(3-8)计算 FM 的输出，此时 FM 的输出是一个有维度的向量，然后 MLP 最后一个全连接层的输出维度也可以设为不为 1 的值，此时将这两个输出向量拼接再进行全连接层的传递就会变得有意义了。

代码实现了前一种方法，代码的地址为 recbyhand\chapter3\s24_DeepFM.py。核心的代码如下：

```
#recbyhand\chapter3\s24_DeepFM.py
class DeepFM( nn.Module ):

    def __init__( self, n_features, user_df, item_df, dim = 128 ):
        super( DeepFM, self ).__init__( )
        #随机初始化所有特征的特征向量
        self.features = nn.Embedding( n_features, dim, max_norm = 1 )
        #记录好用户和物品的特征索引
        self.user_df = user_df
        self.item_df = item_df
```

```
        #得到用户和物品特征的数量的和
total_neigbours = user_df.shape[1] + item_df.shape[1]
        #初始化 MLP 层
self.mlp_layer = self.__mlp( dim * total_neigbours )

    def __mlp( self, dim ):
        return nn.Sequential(
nn.Linear( dim, dim //2 ),
nn.ReLU( ),
nn.Linear( dim //2, dim //4 ),
nn.ReLU( ),
nn.Linear( dim //4, 1 ),
nn.Sigmoid( ) )

    #FM 部分
def FMcross( self, feature_embs ):
        #feature_embs:[ batch_size, n_features, dim ]
        #[ batch_size, dim ]
square_of_sum = torch.sum( feature_embs, dim = 1 ) ** 2
        #[ batch_size, dim ]
sum_of_square = torch.sum( feature_embs ** 2, dim = 1 )
        #[ batch_size, dim ]
        output = square_of_sum - sum_of_square
        #[ batch_size, 1 ]
        output = torch.sum( output, dim = 1, keepdim = True )
        #[ batch_size, 1 ]
        output = 0.5 * output
        #[ batch_size ]
        return torch.squeeze( output )

    #DNN 部分
def Deep( self, feature_embs ):
        #feature_embs:[ batch_size, n_features, dim ]
        #[ batch_size, total_neigbours * dim ]
feature_embs = feature_embs.reshape( ( feature_embs.shape[0], -1 ) )
        #[ batch_size, 1 ]
        output = self.mlp_layer( feature_embs )
        #[ batch_size ]
        return torch.squeeze( output )

    #把用户和物品的特征合并起来
def __getAllFeatures( self,u, i ):
        users = torch.LongTensor( self.user_df.loc[u].values )
        items = torch.LongTensor( self.item_df.loc[i].values )
        all = torch.cat( [ users, items ], dim = 1 )
        return all
```

```
    #前向传播方法
    def forward( self, u, i ):
        #得到用户与物品组合起来后的特征索引
        all_feature_index = self.__getAllFeatures( u, i )
        #取出特征向量
        all_feature_embs = self.features( all_feature_index )
        #[batch_size]
        fm_out = self.FMcross( all_feature_embs )
        #[batch_size]
        deep_out = self.Deep( all_feature_embs )
        #[batch_size]
        out = torch.sigmoid( fm_out + deep_out )
        return out
```

这次 DeepFM 的代码是将所有特征同时传递给了 FM 部分和 Deep 部分，正如前文中所讲，这样不仅逻辑清晰，代码写起来也很省力。当然缺点是对模型的学习要求更高了，即对数据的质量要求更高。

3.2.5　AFM

16min

AFM 是浙江大学与新加坡国立大学于 2017 年发布的模型，全称为 Attentional Factorization Machines[7]，是在 FM 的基础上加入注意力机制。图 3-17 是论文中的模型结构图。

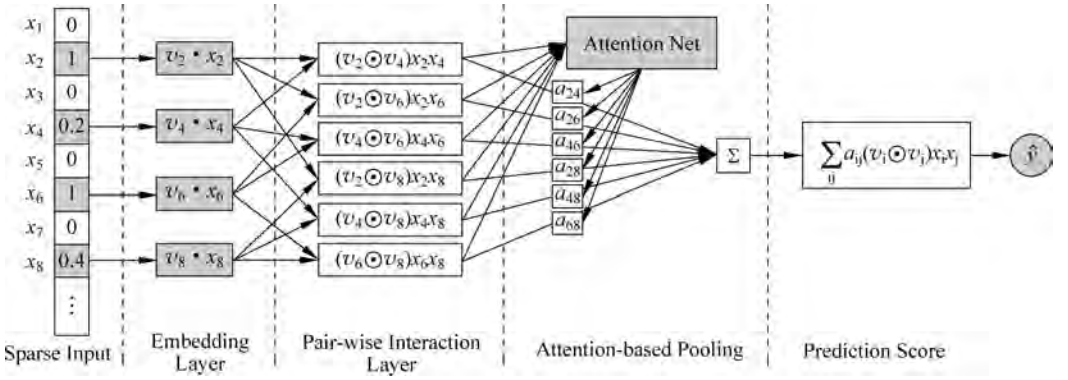


图 3-17　AFM 模型结构图[7]

先来回顾一下 FM 的标准公式：

$$\hat{y} = \sigma\left(w0 + \sum_{i=1}^{n} w1_i x_i + \sum_{i=1}^{n}\sum_{j=i+1}^{n} (\boldsymbol{v}_i \cdot \boldsymbol{v}_j) x_i x_j\right) \tag{3-9}$$

而 AFM 的完整公式如下[7]：

$$\hat{y} = \sigma\left(w0 + \sum_{i=1}^{n} w1_i x_i + p^{\mathrm{T}} \sum_{i=1}^{n}\sum_{j=i+1}^{n} a_{ij} (\boldsymbol{v}_i \odot \boldsymbol{v}_j) x_i x_j\right) \tag{3-10}$$

可以发现从 FM 到 AFM 零次项和一次项没有变,而二次项多了几个参数。首先从向量求内积变为⊙(哈达玛乘法),即全元素对应位相乘,对于一个向量而言,哈达玛积与内积的区别是少了一步累加,所以 $\sum_{i=1}^{n}\sum_{j=i+1}^{n}a_{ij}(\boldsymbol{v}_i\odot\boldsymbol{v}_j)x_ix_j$ 这一部分的输出会是一个维度为 k 的向量而不是一个标量,所以 $\boldsymbol{p}$ 也需要是一个维度为 k 的向量从而使 $\boldsymbol{p}$ 与二次项输出求内积可以得到一个标量。

a_{ij} 是特征 i 与特征 j 之间的注意力,计算过程如下[7]:

$$\begin{cases}a'_{ij}=\boldsymbol{h}^{\mathrm{T}}\mathrm{ReLU}(\boldsymbol{W}(\boldsymbol{v}_i\odot\boldsymbol{v}_j)x_ix_j+\boldsymbol{b})\\ a_{ij}=\mathrm{Softmax}(a'_{ij})=\dfrac{\exp(a'_{ij})}{\sum\limits_{(i,j)\in R_x}\exp(a'_{ij})}\end{cases}\tag{3-11}$$

$\boldsymbol{W}$ 和 $\boldsymbol{b}$ 可视为一个线性层的权重及偏置项,而这个线性层的输入是隐向量长度 k,输出是一个超参,假设是 t,所以 $\boldsymbol{W}\in\mathbf{R}^{t\times k}$,$\boldsymbol{b}\in\mathbf{R}^{t}$,$\boldsymbol{h}\in\mathbf{R}^{t}$。

AFM 的计算过程很简单,加入了注意力机制后,模型的表达能力会更优秀,并且一定程度也增加了模型的可解释性,因为可以直接提取出注意力作为每两个特征之间的权重,以便解释出推荐理由。

工程实现 AFM 的训练模型首先自然先省去 one-hot 表示,与之前一样直接将隐向量作为特征 Embedding。模型初始化参数的代码如下:

```
#recbyhand\chapter3\s25_AFM.py
class AFM( nn.Module ):

    def __init__( self, n_features, k, t ):
        super( AFM, self ).__init__( )
        #随机初始化所有特征的特征向量
        self.features = nn.Embedding( n_features, k, max_norm = 1 )
        #注意力计算中的线性层
        self.attention_liner = nn.Linear( k, t )
        #AFM 公式中的 h
        self.h = init.xavier_uniform_( Parameter( torch.empty( t, 1 ) ) )
        #AFM 公式中的 p
        self.p = init.xavier_uniform_( Parameter( torch.empty( k, 1 ) ) )
```

另外在计算二次项的时候自然要避免双重 for 循环,仍然要用到 FM 的二次项简化公式。正如前文中提过,向量间的哈达玛积与内积之间的区别是少了一步累加,所以其实 $\sum_{i=1}^{n}\sum_{j=i+1}^{n}(v_i\odot v_j)x_ix_j$ 是式(3-8)的另一种表述形式。如此一来仍可用与 FNN 同样的方式去批量计算该二次项的输出,代码如下:

```
#recbyhand\chapter3\s25_AFM.py
def FMaggregator( self, feature_embs ):
    #feature_embs:[ batch_size, n_features, k ]
    #[ batch_size, k ]
    square_of_sum = torch.sum( feature_embs, dim = 1 ) ** 2
    #[ batch_size, k ]
    sum_of_square = torch.sum( feature_embs ** 2, dim = 1 )
    #[ batch_size, k ]
    output = square_of_sum - sum_of_square
    return output
```

在计算注意力时，其中输入的 embs 是上面 FM 聚合层的输出，代码如下：

```
#recbyhand\chapter3\s25_AFM.py
#注意力计算
def attention( self, embs ):
    #embs: [ batch_size, k ]
    #[ batch_size, t ]
    embs = self.attention_liner( embs )
    #[ batch_size, t ]
    embs = torch.ReLU( embs )
    #[ batch_size, 1 ]
    embs = torch.matmul( embs, self.h )
    #[ batch_size, 1 ]
    atts = torch.Softmax( embs, dim = 1 )
    return atts
```

得到批量的注意力后，再与批量的 FM 聚合层输出进行最后的计算，代码如下：

```
#recbyhand\chapter3\s25_AFM.py
#经过 FM 层得到输出
embs = self.FMaggregator( all_feature_embs )
#得到注意力
atts = self.attention( embs )
outs = torch.matmul(atts * embs, self.p)
```

当然之后还有调整形状及求 Sigmoid 等操作，完整代码可到配套代码中查看。另外本节的示例代码省略了零次项和一次项的计算，因为那些其实不重要，而且也很简单，大家有兴趣可以自己实现。

3.2.6　小节总结

FM 衍生出的推荐算法模型还有很多，例如 FFM、PNN、NFM、ONN、xDeepFM 及与图神经网络结合的 Graph FM。本书还是那个主张，大家学习算法的时候学的是一种推导的

过程,而不要执着于算法的形式上。

例如 FNN 是 FM+MLP,DeepFM 是 FM 和 MLP 横向的结合,AFM 在计算 FM 二次项时加入了注意力机制。那些没在本书中详细介绍的算法其实无非是添砖加瓦的操作,例如 FFM 加入了一个特征域的概念,ONN 是 FFM 与 MLP 的结合,xDeepFM 看名字就知道是在 DeepFM 的基础上增加了一些新花样。

3.3 序列推荐算法

序列模型本身一直在发展,从最早的马尔可夫链到深度学习时代的 RNN,以及现在流行的 BERT。

在推荐系统里序列模型一直有着重要的地位,它的核心思想是通过用户的行为序列建立模型推荐,例如历史观看的电影序列,以及历史购买的商品序列等。因为通过统计历史序列数据可以学习到用户兴趣的变化,从而为序列中的下一个进行推荐预测。

3.3.1 基本序列推荐模型

在 3.1.4 节~3.1.6 节,从 RNN 的角度已经介绍过一些基于 RNN 的序列推荐模型,但如果将 RNN 作为最基本的序列推荐模型仍然略显复杂。当然从马尔可夫链讲起就太古老了,但是可以从序列推荐算法的核心思想出发,如图 3-18 所示。

10min

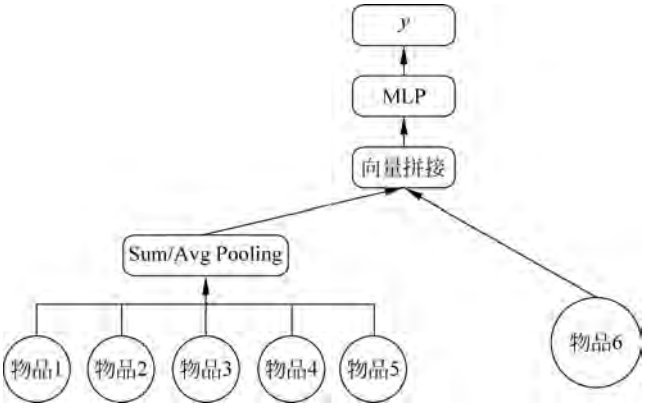


图 3-18 基本序列推荐模型

图 3-18 中的物品 1、物品 2 和物品 3 等可以认为是用户历史交互的物品序列,其实与之前 RNN 章节中的数据是一个意思。这张图显示的是将历史交互的物品序列直接进行求和或者求平均的池化操作,从而使得到的向量再与要预测的物品向量做拼接,然后经过 MLP 网络最终输出预测值。核心的代码如下:

```
#recbyhand\chapter3\s31_base_Sequential.py
class Base_Sequential( nn.Module ):
```

```
    def __init__( self, n_items, dim = 128):
        super( Base_Sequential, self ).__init__()
        #随机初始化所有物品向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )
        self.dense = self.dense_layer( dim * 2, 1 )

    #全连接层
    def dense_layer( self, in_features, out_features ):
        return nn.Sequential(
            nn.Linear( in_features, out_features ),
            nn.Tanh() )

    def forward( self, x, item, isTrain = True ):
        #[ batch_size, len_seqs, dim ]
        item_embs = self.items( x )
        #[ batch_size, dim ]
        sumPool = torch.sum( item_embs, dim = 1 )
        #[ batch_size, dim ]
        one_item = self.items( item )
        #[ batch_size, dim * 2 ]
        out = torch.cat( [ sumPool, one_item ], dim = 1)
        #[ batch_size, 1 ]
        out = self.dense( out )
        #训练时采取 DropOut 来防止过拟合
        if isTrain: out = F.DropOut( out )
        #[ batch_size ]
        out = torch.squeeze( out )
        logit = torch.sigmoid( out )
        return logit
```

当然仅仅这样做并没有把序列的信息充分利用起来，至少应该给每个物品设置不同的权重进行加权求和。这个权重该怎么设置呢？自然就应该利用注意力机制。

3.3.2 DIN 与注意力计算方式

15min

6min

深度兴趣网络（Deep Interest Network，DIN[8]）是阿里巴巴团队在 2018 年发布的模型。DIN 的中心思想是在 3.3.1 节的基本模型上添加了注意力机制。

先从一个简单的结构出发来了解 DIN，它的基本网络结构如图 3-19 所示。

其实是在基本序列模型的基础上增加了注意力机制。先用一个最简单的注意力计算过程来说明，注意力计算公式如下：

$$\begin{cases} a_i' = \boldsymbol{h}^{\mathrm{T}} \sigma(\boldsymbol{W}\boldsymbol{x}_i + \boldsymbol{b}) \\ a_i = \mathrm{Softmax}(a_i') = \dfrac{\exp(a_i')}{\sum\limits_{i \in \mathbf{R}_x} \exp(a_i')} \end{cases} \tag{3-12}$$

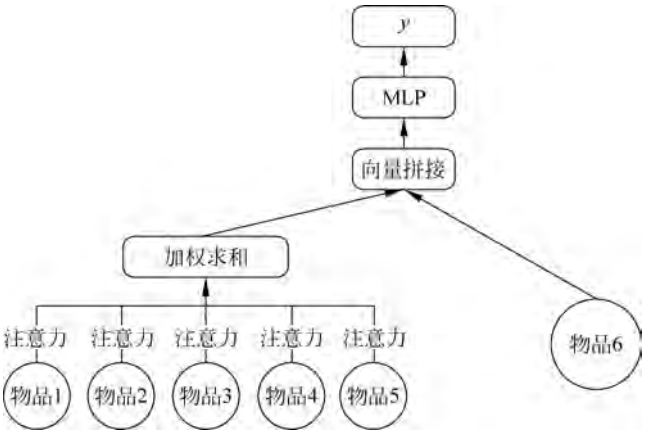


图 3-19　DIN 基本网络结构

其中，$\sigma(\cdot)$指任意激活函数，x_i 为物品 i 的 Embedding，假设维度为 k。$\boldsymbol{W}\boldsymbol{x}_i+\boldsymbol{b}$ 是一个输入维度为 k、输出维度假设为 t 的线性层。$\boldsymbol{h}$ 是一个维度为 t 的向量。在代码中 h 这个部分其实可用一个输入维度为 t、输出维度为 1 的线性层代替，效果是一样的。对每个物品向量进行计算后再进行 Softmax 激活来获得归一化的注意力权重。之后就用这些注意力权重进行加权求和，公式如下：

$$f(x)=\sum_{i\in R_x} a_i x_i \tag{3-13}$$

$i\in R_x$ 指遍历该用户的历史交互物品，得到 $f(x)$后再与要预测的目标物品的向量进行拼接，然后经 MLP 传播，最终输出预测值。

注意力的计算方式并不止一种，例如可以加入用户的 Embedding 更进一步地学习到用户对每个历史交互物品的注意力，如图 3-20 所示。

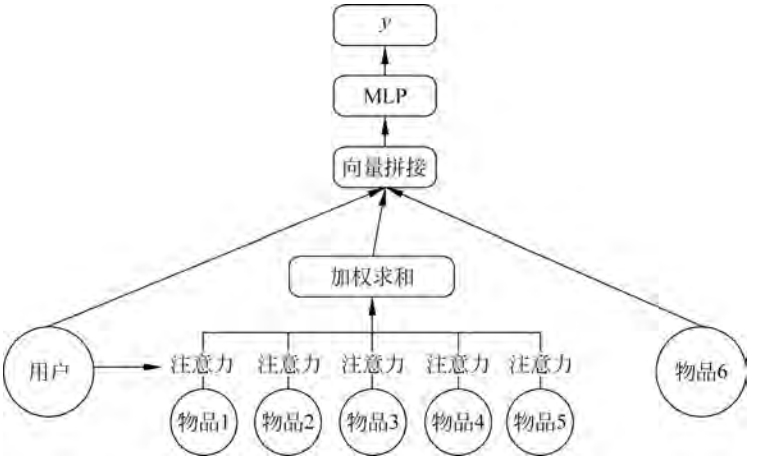


图 3-20　加入用户向量的 DIN

假设用户向量为 $\boldsymbol{u}$，第 i 个物品向量为 $\boldsymbol{x}_i$。通常最简单的注意力还是一个点乘，即 $a_i=\boldsymbol{u}\cdot\boldsymbol{x}_i$。或者 $a_i=\boldsymbol{u}\boldsymbol{W}\boldsymbol{x}_i$，其中 $\boldsymbol{W}$ 是 $\boldsymbol{u}$ 向量长度×$\boldsymbol{x}_i$ 向量长度的矩阵，但是更好的办法是用向量对应位置全元素运算的方式来计算。例如全元素相乘、全元素相加或全元素相减。全元素运算后的那个向量再经一次或几次线性变化得到的向量也可得到注意力权重。

基础知识——注意力机制基础计算方式

是时候总结一下注意力机制的计算方式了，注意力的计算其实能被看作一个小型的神经网络，任何的拼接可产生无穷多种神经网络。当然用注意力计算神经网络往往不会非常复杂，在这里就来介绍一下最基本的计算方式。

首先定义如下一个公式：

$$l(\boldsymbol{x})=\boldsymbol{w}\boldsymbol{x}+\boldsymbol{b} \tag{3-14}$$

将 $l(\boldsymbol{x})$指代对向量 $\boldsymbol{x}$ 做一次带偏置项的线性变化。

(1) 最基本的对于单个样本 i 的注意力计算公式：

$$a=\mathrm{Softmax}(l(\boldsymbol{x})) \tag{3-15}$$

可以看到此处将线性变化后得到的值进行 Softmax 归一化，注意力权重是一个一维的标量，所以 $l(\boldsymbol{x})$在这里的输出维度是 1，输入维度当然是 $\boldsymbol{x}$ 向量的维度。

(2) 单个样本 i 的注意力计算范式：

$$\begin{cases} d(\boldsymbol{x})=\sigma(l(\boldsymbol{x})) \\ a=\mathrm{Softmax}(l(d(\cdots d(x))))=\mathrm{Softmax}(\mathrm{MLP}(\boldsymbol{x})) \end{cases} \tag{3-16}$$

$\sigma(\cdot)$代表任意激活函数，$d(\boldsymbol{x})$代表一个全连接层，式子的后半部分可看作一个以 Softmax 作为最后一层全连接层激活函数的 MLP 网络，中间包含若干全连接层，输入和输出的维度都可任意指定，只要保证第一层的输入维度是 $\boldsymbol{x}$ 向量的长度，最后一层输出的维度是 1 即可。

对于单个样本，其注意力权重是针对结果而言的，即这个样本对最终模型结果的影响越大，它的注意力权重就会越大，所以 MLP 的计算意义是通过增加模型参数来放大原本对结果有较大影响的样本影响力，以及缩小原本对结果影响较小的样本影响力。

(3) 两个样本 i 和 j 的点乘注意力的计算方式：

$$a=\mathrm{Softmax}(\boldsymbol{x}_i\cdot\boldsymbol{x}_j)$$

或

$$a=\mathrm{Softmax}(\boldsymbol{x}_i\boldsymbol{W}\boldsymbol{x}_j),\quad \boldsymbol{W}\in\mathbf{R}^{|x_i|\times|x_j|} \tag{3-17}$$

两个样本间注意力的意义不仅针对最终结构，也针对彼此。例如点积代表两个向量间的相似度，即仅点积计算可以使原本相似的两个样本产生更大的注意力权重。加入一个线性变换矩阵则能增加模型的拟合度。

(4) 两个样本 i 和 j 的全元素运算注意力的计算方式：

$$a=\text{Softmax}(\text{MLP}(\boldsymbol{x}_i \circledcirc \boldsymbol{x}_j)) \tag{3-18}$$

此处用一个◎符号代表任意全元素运算，可以是加法、乘法或者减法。通常不会用除法。全元素运算相比点乘的好处在于损失的信息可以更少，并且运算后维度不变，可视为一个单独的向量再进行 MLP 的传播。

所以这么一来，多个样本的注意力计算方式应该也推导出来了。

(5) 多个样本的注意力计算方式：

$$a=\text{Softmax}(\text{MLP}(\boldsymbol{x}^0 \circledcirc \boldsymbol{x}^1 \cdots \circledcirc \boldsymbol{x}^i)) \tag{3-19}$$

将所有样本全部进行全元素计算得到的向量进行 MLP 的传递，但对于推荐系统而言，不太会出现需要计算两个以上样本注意力权重的场景，且在处理两个以上样本时，用 CNN 或 RNN 等计算方式聚合信息会比全元素相乘更好，但是在注意力层就把网络变得如此复杂是很容易过拟合的，所以并不建议去设计两个以上样本的注意力权重计算场景。

弄明白注意力的计算方式后，对于 DIN 算法基本就学会了 70%。当然商业级的 DIN 网络还会更复杂，业界通常会用物品的特征组合代替物品，所以需要学那些物品特征的向量表示，而不是每个物品的原子化向量表示。商业级的 DIN 网络结构如图 3-21 所示。

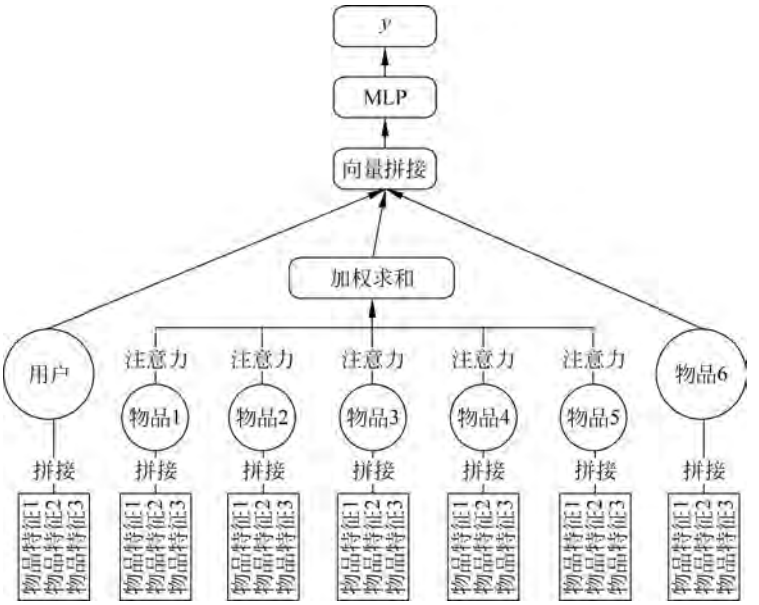


图 3-21 商业级 DIN 示意图

可以看到用户也可由用户的特征组合来指代。

示例代码实现了最基本的 DIN 模型，核心部分的代码如下：

```
#recbyhand\chapter3\s32_DIN.py
class DIN( nn.Module ):

    def __init__( self, n_items, dim = 128, t = 64 ):
        super( DIN, self ).__init__()
        #随机初始化所有物品向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )
        self.fliner = nn.Linear( dim * 2, 1 )
        #注意力计算中的线性层
        self.attention_liner = nn.Linear(dim, t)
        self.h = init.xavier_uniform_( Parameter( torch.empty( t, 1 ) ) )

        #初始化一个 BN 层,在 Dice 计算时会用到
        self.BN = nn.BatchNorm1d(1)

    #Dice 激活函数
    def Dice(self, embs, a = 0.1 ):
        prob = torch.sigmoid( self.BN( embs ) )
        return prob * embs + ( 1 - prob ) * a * embs

    #注意力计算
    def attention( self, embs ):
        #embs: [ batch_size, k ]
        #[ batch_size, t ]
        embs = self.attention_liner( embs )
        #[ batch_size, t ]
        embs = torch.ReLU( embs )
        #[ batch_size, 1 ]
        embs = torch.matmul( embs, self.h )
        #[ batch_size, 1 ]
        atts = torch.Softmax( embs, dim = 1 )
        return atts

    def forward(self, x, item, isTrain = True):
        #[ batch_size, len_seqs, dim ]
        item_embs = self.items( x )
        #[ batch_size, len_seqs, 1 ]
        atts = self.attention( item_embs )
        #[ batch_size, dim]
        sumWeighted = torch.sum( item_embs * atts, dim = 1 )
        #[ batch_size, dim]
        one_item = self.items(item)
        #[ batch_size, dim * 2 ]
        out = torch.cat( [ sumWeighted, one_item ], dim = 1 )
        #[ batch_size, 1 ]
        out = self.fliner( out )
```

```
        out = self.Dice( out )
        #训练时采取 DropOut 来防止过拟合
        if isTrain: out = F.DropOut( out )
        #[ batch_size ]
        out = torch.squeeze( out )
        logit = torch.sigmoid( out )
        return logit
```

大家发现了没有,在线性层之后紧跟着一个 Dice 激活函数。Dice 是阿里巴巴团队创新的一个激活函数。从代码上看很简单,下面就用一个小节来了解一下 Dice 激活函数。

3.3.3 从 PReLU 到 Dice 激活函数

数据相关激活函数(Data Dependent Activation Function,Dice)[8]是阿里巴巴团队伴随 DIN 算法一起发表的创新激活函数。阿里巴巴团队发表的推荐算法通常很具备工程性,并且它们自己有一套推荐算法体系,Dice 激活函数从原理上看就很显然是一个在实战中诞生的算法,但在介绍 Dice 计算原理前,得先从 ReLU 讲起。

线性修正单元(Rectified Linear Unit,ReLU)大家一定不陌生,公式如下:

$$\mathrm{ReLU}(x)=\begin{cases}x, & x\geqslant 0\\ 0, & x<0\end{cases} \tag{3-20}$$

ReLU 函数属于"非饱和激活函数",由式(3-20)可见 ReLU 是将所有负值都设为 0。相较于 Sigmoid 与 Tanh 等"饱和激活函数"的优势在于能解决梯度消失问题且可加快收敛速度。

ReLU 的优点也是它的缺点,如果大多数的参数为负值,则显然 ReLU 的激活能力会大打折扣,所以参数化线性修正单元(Parametric Rectified Linear Unit,PReLU)应运而生,PReLU 公式如下:

$$\mathrm{PReLU}(x)=\begin{cases}x, & x\geqslant 0\\ \alpha x, & x<0\end{cases} \tag{3-21}$$

PReLU 与 ReLU 不同的地方就在于在负值部分赋予了一个负值斜率 α。如此一来就不是所有负值都归为 0,而是会根据 α 的值发生变化。

当 α 很小时又可称为小线性修正单元(Leaky Rectified Linear Unit,LeakyReLU)。如果将 α 的值设定在一个范围内随机获取,则是随机线性修正单元(Randomized Rectified Linear Unit,RReLU)。

图 3-22 展示了 ReLU、LeakyReLU、PReLU 和 RReLU 的函数图像。

介绍完这么多的 ReLU 系列函数,终于要轮到 Dice 出场了。首先注意 PReLU 公式的含义是当输入大于 0 时,输出等于输入的值,而当输入小于或等于 0 时,输出是 αx。设 $p(x)$为输入值 x 大于 0 的概率,则输出 $f(x)$的期望值可表示为[8]

$$f(x)=p(x)\cdot x+(1-p(x))\cdot \alpha x \tag{3-22}$$

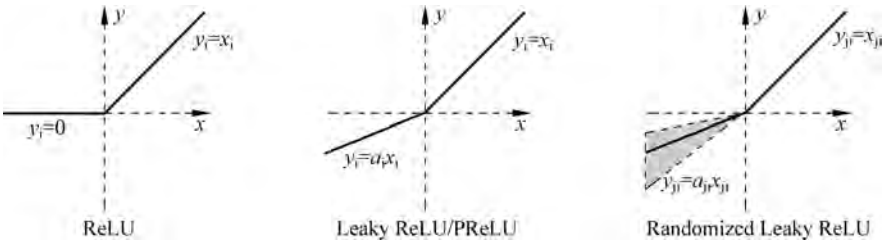


图 3-22　各个 ReLU 函数的示意图

在 Dice 中，又将 $p(x)$定义如下：

$$p(x)=\text{Sigmoid}\left(\frac{x-E(x)}{\text{Var}(x)+\varepsilon}\right) \tag{3-23}$$

$E(x)$表示样本的均值，Var(x)表示方差，ε 是噪声因子。Sigmoid 已经再熟悉不过了，是输出 0～1 的激活函数，而 Sigmoid 内部的计算过程其实是批量归一化算法。

批量归一化(Batch Normalization，BN)的计算公式为

$$\text{BN}(x)=\frac{x-E(x)}{\text{Var}(x)+\varepsilon} \tag{3-24}$$

所以 Dice 激活函数可写成：

$$\text{Dice}(x)=\text{Sigmoid}(\text{BN}(x))\cdot x+1-\text{Sigmoid}(\text{BN}(x))\cdot ax \tag{3-25}$$

这样一来是不是就显得很简单了。Batch Normalization 属于基础操作，这个大家应该不陌生，其意义主要有以下 3 个：

(1) 减缓过拟合。

(2) 在训练过程中使数据平滑从而加快训练的速度。

(3) 减缓因数据不平滑而造成的梯度消失。

所以 Dice 激活函数的意义也在于此。BN 算法在 PyTorch 中有现成的 API，所以 Dice 激活函数实现起来非常简单，代码如下：

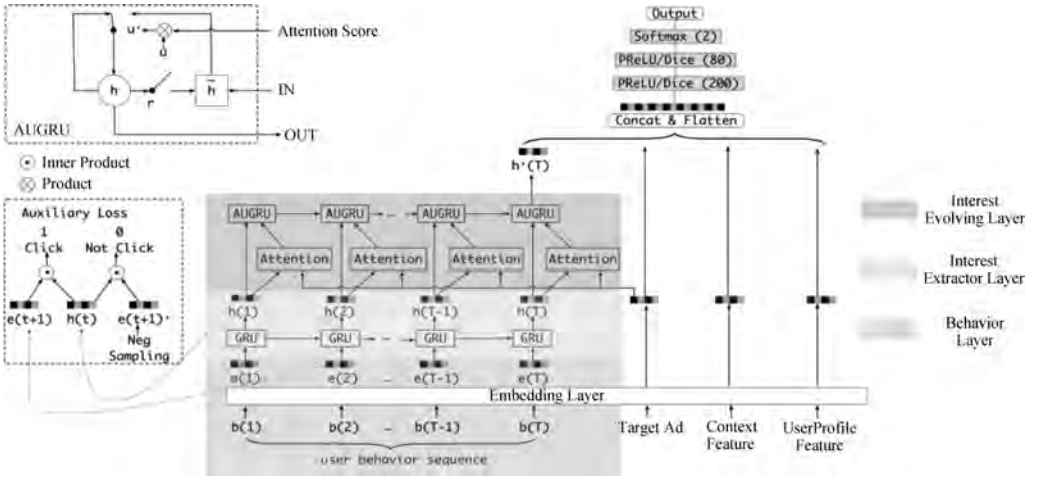
```
#recbyhand\chapter3\s32_DIN.py
def Dice( self, x, a = 0.1 ):
    BN = torch.nn.BatchNorm1d( 1 )
    prob = torch.sigmoid( BN( x ) )
    return prob * x + ( 1 - prob ) * a * x
```

3.3.4　DIEN 模拟兴趣演化的序列网络

17min

深度兴趣演化网络(Deep Interest Evolution Network，DIEN)[9] 是阿里巴巴团队在 2018 年推出的另一力作，比 DIN 多了一个 Evolution，即演化的概念。

在 DIEN 模型结构上比 DIN 复杂许多，但大家丝毫不用担心，本书会将 DIEN 拆解开来详细地说明。首先来看从 DIEN 论文中截下的模型结构图，如图 3-23 所示。



19min

图 3-23 DIEN 模型结构全图[9]

这张图初看之下很复杂,但可从简单到难一点点来说明。首先最后输出往前一段的截图如图 3-24 所示。

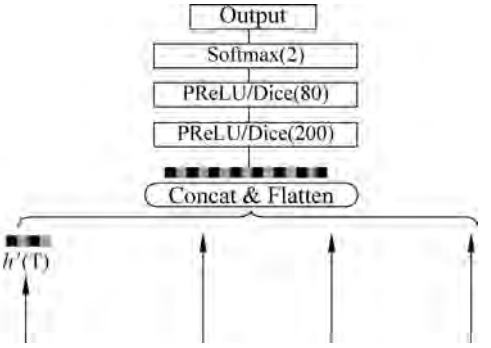


图 3-24 DIEN 模型结构局部图(1)[9]

这部分很简单,是一个 MLP,下面一些箭头表示经过处理的向量。这些向量会经一个拼接层拼接,然后经几个全连接层,全连接层的激活函数可选择 PReLU 或者 Dice。最后用了一个 Softmax(2)表示二分类,当然也可用 Sigmoid 进行二分类任务。

对输出端了解过后,再来看输入端,将输入端的部分放大后截图如图 3-25 所示。

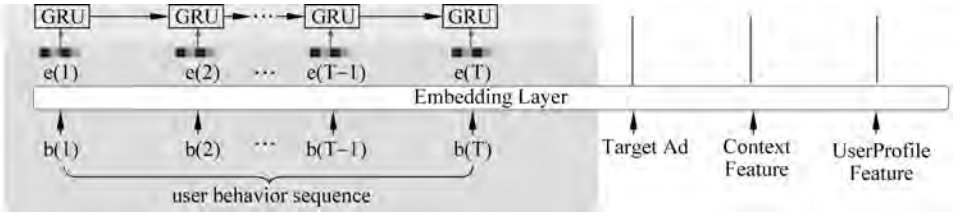


图 3-25 DIEN 模型结构局部图(2)[9]

从右往左看，UserProfile Feature 指用户特征，Context Feature 指内容特征，Target Ad 指目标物品，其实这 3 个特征表示的无非是随机初始化一些向量，或者通过特征聚合的方式量化表达各种信息。

DIEN 模型的重点就在图 3-25 的 user behavior sequence 区域。user behavior sequence 代表用户行为序列，通常利用用户历史交互的物品代替。图 3-26 展示了这块区域的全貌。

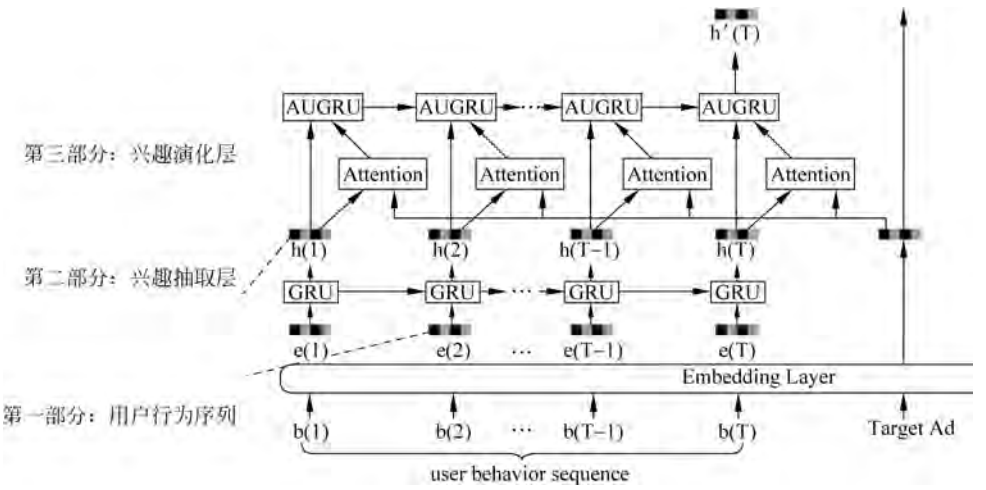


图 3-26　DIEN 模型结构局部图(3)[9]

这部分是 DIEN 算法的核心，这里直接配合公式和代码来讲解。本节代码的地址为 recbyhand\chapter3\s34_DIEN.py。

第一部分：用户行为序列，是将用户历史交互的物品序列经 Embedding 层初始化物品序列向量准备输入下一层，代码如下：

```
#recbyhand\chapter3\s34_DIEN.py
#初始化 embedding
items = nn.Embedding( n_items, dim, max_norm = 1 )
#[batch_size, len_seqs, dim]
item_embs = items(history_seqs)#history_seqs 指用户历史物品序列 id
```

所以输出的是一个[批次样本数量，序列长度，向量维度]的张量。

第二部分：兴趣抽取层，是一个 GRU 网络，将上一层的输出在这一层输入。GRU 是 RNN 的一个变种，在 PyTorch 里有现成模型，所以只有以下两行代码。

```
#recbyhand\chapter3\s34_DIEN.py
#初始化 GRU 网络，注意正式写代码时，初始化动作通常写在__init__() 方法里
GRU = nn.GRU( dim, dim, batch_first = True)
outs, h = GRU(item_embs)
```

和 RNN 网络一样，会有两个输出，一个是 outs，是每个 GRU 单元输出向量组成的序列，维度是[批次样本数量，序列长度，向量维度]，另一个 $\boldsymbol{h}$ 指的是最后一个 GRU 单元的输出向量。在 DIEN 模型中，目前位置处的 $\boldsymbol{h}$ 并没有作用，而 outs 却有两个作用。一个作用是作为下一层的输入，另一个作用是获取辅助 loss。

什么是辅助 loss，其实 DIEN 网络是一个联合训练任务，最终对目标物品的推荐预测可以产生一个损失函数，暂且称为 L_{target}，而这里可以利用历史物品的标注得到一个辅助损失函数，此处称为 L_{aux}。总的损失函数的计算公式为

$$L = L_{\text{target}} + \alpha \cdot L_{\text{aux}} \tag{3-26}$$

其中，α 是辅助损失函数的权重系数，是个超参。这里辅助损失函数的计算与 3.1.6 节中所介绍的联合训练 RNN 不同，3.1.6 节说的是多分类预测产生的损失函数，而 DIEN 给出的方法是一个二分类预测，如图 3-27 所示。

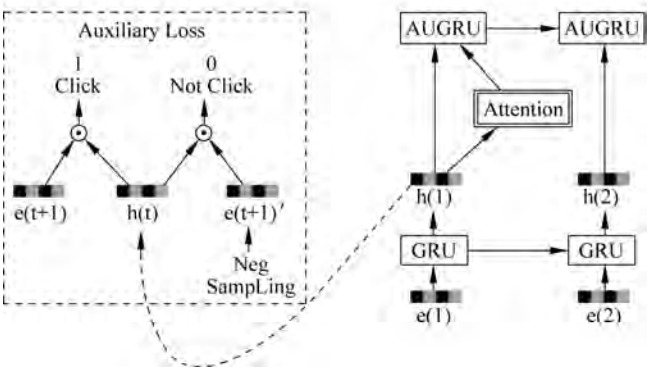


图 3-27 DIEN 模型结构局部图(4)[9]

历史物品标注指的是用户对对应位置的历史物品交互的情况，通常由 1 和 0 组成，1 表示“感兴趣”，0 则表示“不感兴趣”，如图 3-27 所示，将 GRU 网络输出的 outs 与历史物品序列的 Embedding 输入一个二分类的预测模型中即可得到辅助损失函数，代码如下：

```
#recbyhand\chapter3\s34_DIEN.py
#辅助损失函数的计算过程
def forwardAuxiliary( self, outs, item_embs, history_labels ):
    '''
    :param item_embs: 历史序列物品的向量 [ batch_size, len_seqs, dim ]
    :param outs: 兴趣抽取层 GRU 网络输出的 outs [ batch_size, len_seqs, dim ]
    :param history_labels: 历史序列物品标注 [ batch_size, len_seqs, 1 ]
    :return: 辅助损失函数
    '''
    #[ batch_size * len_seqs, dim ]
```

```
item_embs = item_embs.reshape( -1, self.dim )
#[ batch_size * len_seqs, dim ]
outs = outs.reshape( -1, self.dim )
#[ batch_size * len_seqs ]
out = torch.sum( outs * item_embs, dim = 1 )
#[ batch_size * len_seqs, 1 ]
out = torch.unsqueeze( torch.sigmoid( out ), 1 )
#[ batch_size * len_seqs,1 ]
history_labels = history_labels.reshape( -1, 1 ).float()
return self.BCELoss( out, history_labels )
```

调整张量形状后做点乘，Sigmoid 激活后与历史序列物品标注做二分类交叉熵损失函数(BCEloss)。

以上是第二部分兴趣抽取层所做的事情，最后来看最关键的第三部分。

第三部分：兴趣演化层，主要由一个叫作 AUGRU 的网络组成，AUGRU 是在 GRU 的基础上增加了注意力机制。全称叫作 GRU With Attentional Update Gate。AUGRU 的细节结构如图 3-28 所示。

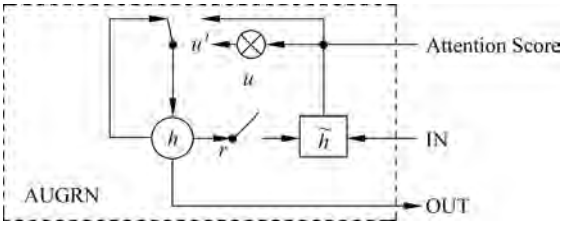


图 3-28　AUGRU 单元细节[9]

GRU 是在 RNN 的基础上增加了所谓的更新门(Update Gate) 和重置门(Reset Gate)。每个 GRU 单元的计算公式如下：

$$\begin{cases} u_t = \sigma(\boldsymbol{W}_u \boldsymbol{i}_t + \boldsymbol{U}_u \boldsymbol{h}_{t-1} + \boldsymbol{b}_u) \\ r_t = \sigma(\boldsymbol{W}_r \boldsymbol{i}_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \\ \tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_h \boldsymbol{i}_t + r_t \odot \boldsymbol{U}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h) \\ \boldsymbol{h}_t = (1 - u_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{u}_t \odot \tilde{\boldsymbol{h}}_t \end{cases} \tag{3-27}$$

其中，$\boldsymbol{u}_t$ 代表第 t 层更新门的输出向量，$\boldsymbol{r}_t$ 代表第 t 层重置门的输出向量。$\boldsymbol{i}_t$ 是序列中第 t 个物品向量，$\boldsymbol{h}_{t-1}$ 是第 $t-1$ 个 GRU 单元的输出向量。其余 $\boldsymbol{W}$、$\boldsymbol{U}$、$\boldsymbol{b}$ 等都是模型要学习的参数。$\boldsymbol{W}$ 和 $\boldsymbol{U}$ 是参数矩阵，输入维度分别对物品向量 i 和循环神经网络单元输出向量 $\boldsymbol{h}$ 的向量维度。输出则自己定义即可，参数的详细维度情况可参考本书配套的代码。

AUGRU 给更新门增添了一个注意力操作，此处用 a_t 代表每个历史序列中物品的注意力权重，所以 AUGRU 的总体计算方式如下[9]：

$$\begin{cases} \boldsymbol{u}_t = \sigma(\boldsymbol{W}_u \boldsymbol{i}_t + \boldsymbol{U}_u \boldsymbol{h}_{t-1} + \boldsymbol{b}_u) \\ \boldsymbol{r}_t = \sigma(\boldsymbol{W}_r \boldsymbol{i}_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \\ \tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_h \boldsymbol{i}_t + \boldsymbol{r}_t \odot \boldsymbol{U}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h) \\ \tilde{\boldsymbol{u}}_t = a_t \times \boldsymbol{u}_t \\ \boldsymbol{h}_t = (1 - \tilde{\boldsymbol{u}}_t) \odot \boldsymbol{h}_{t-1} + \tilde{\boldsymbol{u}}_t \odot \tilde{\boldsymbol{h}}_t \end{cases} \tag{3-28}$$

AUGRU 只是在 GRU 的基础上多了第 4 行,即用注意力权重去更新 Update Gate 输出的操作。在 DIN 模型章节中的基础知识栏目里介绍了很多注意力权重的计算方式。DIEN 论文里给出的是最基础的计算方式,公式如下:

$$a_t = \text{Softmax}(\boldsymbol{i}_t \cdot \boldsymbol{W}_a \cdot \boldsymbol{e}_{\text{tar}}) \tag{3-29}$$

其中,$\boldsymbol{e}_{\text{tar}}$ 指的是目标物品的向量,$\boldsymbol{W}_a$ 是一个线性变换矩阵,维度是 $|i| \times |e|$。

一个完整的 AUGRU 单元的代码如下:

```
#recbyhand\chapter3\s34_DIEN.py
#AUGRU 单元
class AUGRU_Cell(nn.Module):

    def __init__(self, in_dim, hidden_dim ):
        '''
        :param in_dim: 输入向量的维度
        :param hidden_dim: 输出的隐藏层维度
        '''
        super(AUGRU_Cell, self).__init__()

        #初始化更新门的模型参数
        self.Wu = init.xavier_uniform_( Parameter(torch.empty( in_dim, hidden_dim ) ) )
        self.Uu = init.xavier_uniform_( Parameter( torch.empty( in_dim, hidden_dim ) ) )
        self.bu = init.xavier_uniform_( Parameter( torch.empty( 1, hidden_dim ) ) )

        #初始化重置门的模型参数
        self.Wr = init.xavier_uniform_( Parameter( torch.empty( in_dim, hidden_dim ) ) )
        self.Ur = init.xavier_uniform_(Parameter( torch.empty( in_dim, hidden_dim ) ) )
        self.br = init.xavier_uniform_( Parameter( torch.empty( 1, hidden_dim ) ) )

        #初始化计算 h~的模型参数
        self.Wh = init.xavier_uniform_( Parameter( torch.empty( hidden_dim, hidden_dim ) ) )
        self.Uh = init.xavier_uniform_( Parameter( torch.empty( hidden_dim, hidden_dim ) ) )
        self.bh = init.xavier_uniform_( Parameter( torch.empty( 1, hidden_dim ) ) )

        #初始化注意力计算中的模型参数
        self.Wa = init.xavier_uniform_( Parameter( torch.empty( hidden_dim, in_dim ) ) )

    #注意力的计算
```

```
def attention( self, x, item ):
        '''
        :param x: 输入的序列中第 t 个向量 [ batch_size, dim ]
        :param item: 目标物品的向量 [ batch_size, dim ]
        :return: 注意力权重 [ batch_size, 1 ]
        '''
        hW = torch.matmul(x,self.Wa)
        hWi = torch.sum(hW * item,dim = 1)
        hWi = torch.unsqueeze(hWi,1)
        return torch.Softmax(hWi,dim = 1)

    def forward(self,x,h_1,item):
        '''
        :param x: 输入的序列中第 t 个物品向量 [ batch_size, in_dim ]
        :param h_1: 上一个 AUGRU 单元输出的隐藏向量 [ batch_size, hidden_dim ]
        :param item: 目标物品的向量 [ batch_size, in_dim ]
        :return: h 为当前层输出的隐藏向量 [ batch_size, hidden_dim ]
        '''
        #[ batch_size, hidden_dim ]
        u = torch.sigmoid( torch.matmul( x, self.Wu ) + torch.matmul( h_1, self.Uu ) + self.bu )
        #[ batch_size, hidden_dim ]
        r = torch.sigmoid( torch.matmul( x, self.Wr ) + torch.matmul( h_1, self.Ur ) + self.br )
        #[ batch_size, hidden_dim ]
        h_hat = torch.tanh( torch.matmul( x, self.Wh ) + r * torch.matmul( h_1, self.Uh ) + self.bh )
        #[ batch_size, 1 ]
        a = self.attention( x, item )
        #[ batch_size, hidden_dim ]
        u_hat = a * u
        #[ batch_size, hidden_dim ]
        h = ( 1 - u_hat ) * h_1 + u_hat * h_hat
        #[ batch_size, hidden_dim ]
        return h
```

完整的 AUGRU 循环神经网络的代码如下：

```
#recbyhand\chapter3\s34_DIEN.py
class AUGRU( nn.Module ):

    def __init__( self, in_dim, hidden_dim ):
        super( AUGRU, self ).__init__( )
        self.in_dim = in_dim
        self.hidden_dim = hidden_dim
        #初始化 AUGRU 单元
        self.augru_cell = AUGRU_Cell( in_dim, hidden_dim )
```

```
    def forward( self, x, item ):
        '''
        :param x: 输入的序列向量,维度为 [ batch_size, seq_lens, dim ]
        :param item: 目标物品的向量
        :return: outs: 所有 AUGRU 单元输出的隐藏向量[ batch_size, seq_lens, dim ]
        h: 最后一个 AUGRU 单元输出的隐藏向量[ batch_size, dim ]
        '''
        outs = []
        h = None
        #开始循环,x.shape[1]是序列的长度
        for i in range( x.shape[1]):
            if h == None:
                #初始化第一层的输入 h
                h = init.xavier_uniform_( Parameter( torch.empty( x.shape[0], self.hidden_dim ) ) )
            h = self.augru_cell( x[:,i], h, item )
            outs.append( torch.unsqueeze( h, dim = 1 ) )
        outs = torch.cat( outs, dim = 1 )
        return outs, h
```

至此,第三部分的兴趣演化层讲解完毕,物理上它的意义在于通过一个序列神经网络来模拟用户兴趣演化的过程。最后将 AUGRU 输出的 $\boldsymbol{h}$ 作为兴趣演化层的输出向量进行后面的运算,如图 3-29 所示。

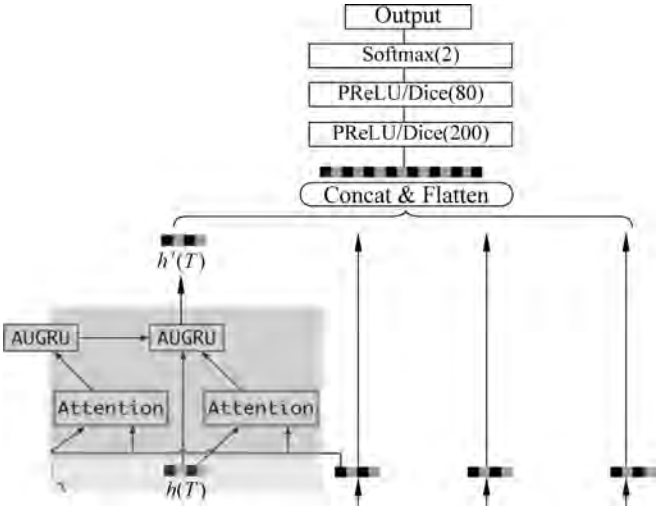


图 3-29 DIEN 模型结构局部图(5)[9]

如此一来就回到了第一张 DIEN 模型结构局部图,即将这个 $\boldsymbol{h}$ 向量经过 MLP 的传递最终输出预测值,以此完成整个 DIEN 模型的传播过程。整个 DIEN 模型的核心代码如下:

```
#recbyhand\chapter3\s34_DIEN.py
class DIEN( nn.Module ):

    def __init__( self, n_items, dim = 128, alpha = 0.2):
        super( DIEN, self ).__init__()
        self.dim = dim
        self.alpha = alpha#计算辅助损失函数时的权重
        self.n_items = n_items
        self.BCELoss = nn.BCELoss( )

        #随机初始化所有特征的特征向量
        self.items = nn.Embedding( n_items, dim, max_norm = 1 )

        #初始化兴趣抽取层的 GRU 网络,直接用 PyTorch 中现成的实现即可
        self.GRU = nn.GRU( dim, dim, batch_first = True)
        #初始化兴趣演化层的 AUGRU 网络,因无现成模型,所以需使用自己编写的 AUGRU
        self.AUGRU = AUGRU( dim, dim )

        #初始化最终 CTR 预测的 MLP 网络,激活函数采用 Dice
        self.dense1 = self.dense_layer( dim * 2, dim, Dice )
        self.dense2 = self.dense_layer( dim, dim//2, Dice )
        self.f_dense = self.dense_layer( dim//2, 1, nn.Sigmoid )

    #全连接层
    def dense_layer(self, in_features, out_features, act ):
        return nn.Sequential(
            nn.Linear( in_features, out_features ),
            act() )

    #辅助损失函数的计算过程
    def forwardAuxiliary( self, outs, item_embs, history_labels ):
        '''
        :param item_embs: 历史序列物品的向量 [ batch_size, len_seqs, dim ]
        :param outs: 兴趣抽取层 GRU 网络输出的 outs [ batch_size, len_seqs, dim ]
        :param history_labels: 历史序列物品标注 [ batch_size, len_seqs, 1 ]
        :return: 辅助损失函数
        '''
        #[ batch_size * len_seqs, dim ]
        item_embs = item_embs.reshape( -1, self.dim )
        #[ batch_size * len_seqs, dim ]
        outs = outs.reshape( -1, self.dim )
        #[ batch_size * len_seqs ]
        out = torch.sum( outs * item_embs, dim = 1 )
        #[ batch_size * len_seqs, 1 ]
        out = torch.unsqueeze( torch.sigmoid( out ), 1 )
        #[ batch_size * len_seqs,1 ]
```

```
        history_labels = history_labels.reshape( -1, 1 ).float()
        return self.BCELoss( out, history_labels )

    def __getRecLogit( self, h, item ):
        #将 AUGRU 输出的 h 向量与目标物品相拼接,之后经 MLP 传播
        concatEmbs = torch.cat([ h, item ], dim = 1)
        logit = self.dense1( concatEmbs )
        logit = self.dense2( logit )
        logit = self.f_dense( logit )
        logit = torch.squeeze( logit )
        return logit

    #推荐 CTR 预测的前向传播
    def forwardRec( self, h, item, y ):
        logit = self.__getRecLogit( h, item )
        y = y.float()
        return self.BCELoss( logit, y )

    #整体前向传播
    def forward( self, history_seqs, history_labels, target_item, target_label ):
        #[ batch_size, len_seqs, dim ]
        item_embs = self.items( history_seqs )

        outs, _ = self.GRU( item_embs )
        #利用 GRU 输出的 outs 得到辅助损失函数
        auxi_loss = self.forwardAuxiliary( outs,item_embs, history_labels )
        #[ batch_size, dim]
        target_item_embs = self.items( target_item )

        #利用 GRU 输出的 outs 与目标的向量输入兴趣演化层的 AUGRU 网络,得到最后一层的
        #输出 h
        _, h = self.AUGRU( outs, target_item_embs )

        #得到 CTR 预估的损失函数
        rec_loss = self.forwardRec( h, target_item_embs, target_label )

        #将辅助损失函数与 CTR 预估损失函数加权求和输出
        return self.alpha * auxi_loss + rec_loss

    #因为模型的 forward 函数输出的是损失函数值,所以另用一个预测函数以方便预测及评估
    def predict( self, x, item ):
        item_embs = self.items( x )
        outs, _ = self.GRU( item_embs )
        one_item = self.items( item )
        _, h = self.AUGRU ( outs, one_item )
        logit = self.__getRecLogit( h, one_item )
        return logit
```

其余代码可去本书配套的代码中详细观察。DIEN 模型到此介绍完毕，虽然复杂，但拆解开来其实也很好理解。本书希望大家不仅把 DIEN 模型学会，还要学会它产生的过程，学会它是如何利用联合训练，如何利用注意力机制，以及如何利用序列循环神经网络等。

3.4 Transformer 在推荐算法中的应用

6min

Transformer 是 2017 年谷歌大脑团队在一篇名为 *Attention Is All You Need*[10] 的论文中提出的序列模型。基于 Transformer 做推荐自然也属于序列推荐模型，但之所以将它单起一节来介绍的原因是 Transformer 近几年名气实在是太大，本书认为有必要尽可能详细地介绍它在推荐系统中的应用。

Transformer 模型原本是解决自然语言处理中机器翻译任务而提出的。本质上是对 Seq2Seq 算法的优化。Seq2Seq 是序列 to 序列，即输入一个序列，去预测另一个序列。例如输入一段英文“What a good day!”，模型的任务是要输出“多好的一天!”以完成机器翻译。Seq2Seq 也会用作聊天对话模型，即输入“问题”，输出“答案”，而 Transformer 也可以理解为一个序列到序列的模型。由编码器 Encoder 和解码器 Decoder 组成。

另外再顺便提一下 BERT，BERT 这个名号甚至比 Transformer 还要响亮。业内更有“万能的 BERT”这种称号，BERT 原名为 Pre-training of Deep Bidirectional Transformers for Language Understanding[11]，是谷歌团队在 2019 年提出的，其实 BERT 才是 Transformer 真正火起来的原因。BERT 算法的结构其实采用的是 12 层的 Transformer “编码器”，所以算法本身还是 Transformer，但 BERT 更多的重点是对预训练模型的运用，以及在预训练模型的基础上进行迁移学习或者微调。

所以伴随 BERT 的问世，谷歌还开源了若干个 BERT 预训练模型。谷歌拥有的自然语言语料的量级可想而知非常庞大。如此一个举动对于那些原本苦恼于收集语料的中小型公司而言，如久旱逢甘霖，所以这才是 BERT 能够火遍大江南北的本质原因。当然 Transformer 的算法结构本身的确也很优秀，尤其是对于自然语言处理而言。

3.4.1 从推荐角度初步了解 Transformer

6min

在介绍 Transformer 这种结构模型在推荐算法中该怎么去用之前，要告诉大家的是推荐算法理论上可以融合任何算法及数学技巧，因为推荐任务可以说完全等效于所有机器学习预测任务。

例如可以把一个机器翻译模型理解成给一个“句子”推荐与其更匹配“句子”的模型，甚至可以把一个人脸识别模型理解成给一张“图片” 推荐与其更匹配的“人名”的模型，所以既然如此，自然语言处理的算法模型当然可以用作推荐。

话说回来，也正因为推荐算法具备这样的性质，所以每当一个新的数学技巧或者某个算法在别的领域火起来之后，一定会有推荐算法的学者争先恐后地将其应用于自己的研究。Transformer 自然也不例外，所以大家不要去期待 Transformer 这种在自然语言处理领域

中的王道算法会在推荐领域中也是王道。

但是 Transformer 毕竟是 Transformer，如果想成为一个成熟的推荐算法工程师，Transformer 仍然是一门必修课。在以后的工作中一定会遇到某些特别适合 Transformer 的任务场景，并且 Transformer 中的一些结构尤其是对于注意力机制的应用很值得大家学习。

言归正传，接下来介绍 Transformer 模型结构，如图 3-30 所示。

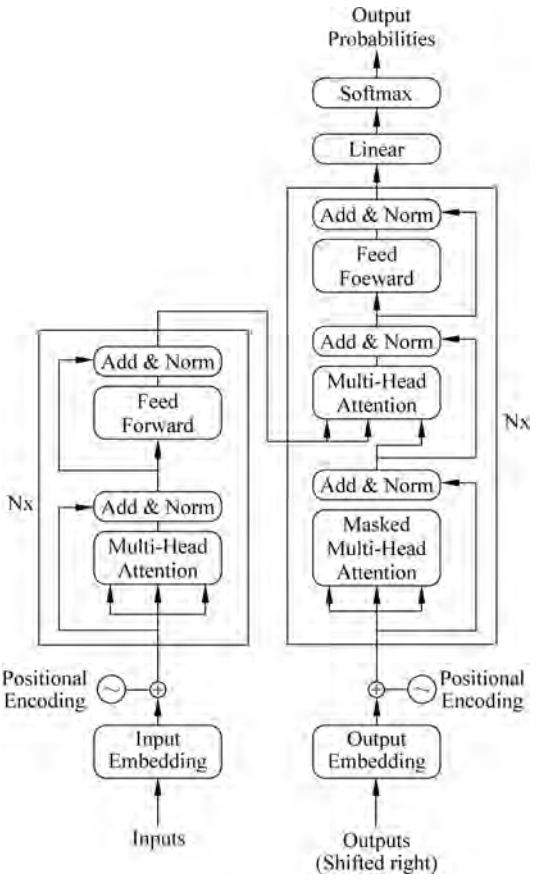


图 3-30 Transformer 模型整体结构[10]

左半边是编码器，右半边是解码器。首先把编码器或者解码器理解为处理序列的神经网络，即它的作用就像是 RNN，所以输入也跟 RNN 一样，是一个包含序列长度的张量。通常该张量的维度是[批次样本数量，序列长度，向量维度]，而输出也是相同维度的张量。

虽然编码器与 RNN 的作用类似，但 Transformer 中的编码器或者解码器并不是由 RNN 演化而来，这个需要特别注意。图 3-30 中的 N×字样表示编码器或者解码器是由 N 个编码层或者解码层组成。论文中默认为 6。现在把注意力集中在单个编码层的结构，如图 3-31 所示。

图 3-31 中有 3 种模模块，分别如下：

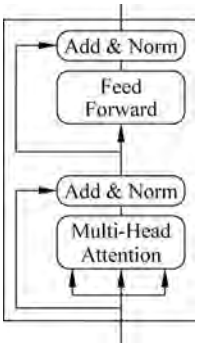


图 3-31　Transformer 编码层详细结构图[10]

(1) Multi-Head Attention,多头注意力。

(2) Add & Norm,残差与 Layer Normalization。

(3) Feed Forward,前馈神经网络。

接下来介绍这些部分分别做了什么。

3.4.2　多头注意力与缩放点乘注意力算法

Transformer 最核心也是最起作用的部分是注意力层。理解了注意力层也就相当于理解了 80%的 Transformer。

多头注意力(Multi-Head Attention),看名字就知道是由多个注意力组合成的一个大注意力,这里先介绍“一个头”的注意力如何计算。

在 3.3.2 节中介绍过很多基础的注意力权重算法,Transformer 中的注意力算法叫作缩放点乘注意力(Scaled Dot-Product Attention),公式如下:

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{3-30}$$

其中,$\boldsymbol{Q}$ 代表 Query 向量,$\boldsymbol{K}$ 代表 Key 向量,$\boldsymbol{V}$ 代表 Value 向量。在“编码器”中,$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 都由输入的序列向量得到。设 $\boldsymbol{X}$ 为输入的序列向量,则

$$\begin{cases}\boldsymbol{Q}=\boldsymbol{W}_q\boldsymbol{X}+\boldsymbol{b}_q\\ \boldsymbol{K}=\boldsymbol{W}_k\boldsymbol{X}+\boldsymbol{b}_k\\ \boldsymbol{V}=\boldsymbol{W}_v\boldsymbol{X}+\boldsymbol{b}_v\end{cases} \tag{3-31}$$

此处用三套不同的线性变化参数给输入的序列向量做线性变化,而在解码器中的某个注意力层的 $\boldsymbol{Q}$ 由来自编码器的输出向量计算而来。

前一个公式计算中的 $\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}$ 是一个点乘,点乘可以表示两个向量之间的相似度,等效于余弦相似度,所以这一步计算的意义也是如果一个 Query 与一个 Key 更相似,则该 Query 对该 Key 的影响就会越大。之后除以 $\sqrt{d_k}$ 是名字中缩放的意思,d_k 指 $\boldsymbol{K}$ 向量的维度。

$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 向量维度是一样的,所以如果它们的维度越大,则 $\boldsymbol{Q}\boldsymbol{K}$ 点乘的值就会越大,虽然

后面会做 Softmax 归一化,但在此之前缩放一下也是为了使数据平滑一点以防止梯度消失。

经过 $\text{Softmax}\left(\frac{\boldsymbol{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right)$ 这样计算后便可得到注意力权重,再由这个注意力权重乘以 V 向量就可作为该注意力层的输出。

以上是所谓"一个头"的注意力层所得到的输出,示例代码如下:

```
#recbyhand\chapter3\s47_transfermorOnlyEncoder.py
#单头注意力层
class OneHeadAttention( nn.Module ):

    def __init__( self, e_dim, h_dim ):
        '''
        :param e_dim: 输入向量维度
        :param h_dim: 输出向量维度
        '''
        super().__init__()
        self.h_dim = h_dim
        #初始化 Q、K、V 的映射线性层
        self.lQ = nn.Linear( e_dim, h_dim )
        self.lK = nn.Linear( e_dim, h_dim )
        self.lV = nn.Linear( e_dim, h_dim )

    def forward(self, seq_inputs):
        #:seq_inputs [ batch, seq_lens, e_dim ]
        Q = self.lQ( seq_inputs ) #[ batch, seq_lens, h_dim ]
        K = self.lK( seq_inputs ) #[ batch, seq_lens, h_dim ]
        V = self.lV( seq_inputs ) #[ batch, seq_lens, h_dim ]
        #[ batch, seq_lens, seq_lens ]
        QK = torch.matmul( Q,K.permute(0, 2, 1) )
        #[ batch, seq_lens, seq_lens ]
        QK /= (self.h_dim ** 0.5)
        #[ batch, seq_lens, seq_lens ]
        a = torch.Softmax( QK, dim = -1 )
        #[ batch, seq_lens, h_dim ]
        outs = torch.matmul( a, V )
        return outs
```

多头注意力其实是将这些输出的向量拼接起来。

$$\begin{aligned}&\text{MultiHead}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \text{Concat}(\text{head1},\ldots,\text{headh}) \cdot \boldsymbol{W}_O\\&\text{where headi} = \text{Attention}(\boldsymbol{Q}_i,\boldsymbol{K}_i,\boldsymbol{V}_i)\end{aligned} \tag{3-32}$$

其中,$\boldsymbol{W}_O$ 是一个线性变化矩阵,维度为[单头注意力层输出向量的长度×head 数量,多头注意力层输入向量的长度]。它的作用是将经过多头注意力操作的向量维度再调整至输入时

的维度。

完整的多头注意力层的代码如下：

```
#recbyhand\chapter3\s47_transfermorOnlyEncoder.py
#多头注意力层
class MultiHeadAttentionLayer(nn.Module):

    def __init__(self, e_dim, h_dim, n_heads):
        '''
        :param e_dim: 输入的向量维度
        :param h_dim: 每个单头注意力层输出的向量维度
        :param n_heads: 头数
        '''
        super().__init__()
        self.atte_layers = nn.ModuleList([OneHeadAttention( e_dim, h_dim ) for _ in range(n_
heads) ] )
        self.l = nn.Linear( h_dim * n_heads, e_dim)

    def forward(self, seq_inputs):
        outs = []
        for one in self.atte_layers:
            out = one(seq_inputs)
            outs.append(out)
        #[ batch, seq_lens, h_dim * n_heads ]
        outs = torch.cat(outs, dim = -1)
        #[ batch, seq_lens, e_dim ]
        outs = self.l(outs)
        return outs
```

3.4.3　残差

所谓残差是在经神经网络多层传递后加上最初的向量，该过程如图 3-32 所示。

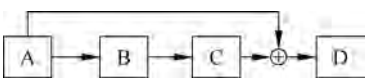


图 3-32　残差连接图

4min

A、B、C、D 是 4 个不同的网络层，A 层的输出经过 B 层和 C 层的传递后再加上 A 层原本的输出即完成残差连接，在代码中是一个加法。

残差的作用是当网络层级深时可以有效防止梯度消失。因为根据后向传播链式法则

$$\frac{\partial Y}{\partial X}=\frac{\partial Y}{\partial Z}\frac{\partial Z}{\partial X} \tag{3-33}$$

图 3-32 的传播方式用数学描述则如下：

$$D_{\text{in}} = A_{\text{out}} + C(B(A_{\text{out}})) \tag{3-34}$$

反向传播时则

$$\frac{\partial D_{\text{in}}}{\partial A_{\text{out}}} = 1 + \frac{\partial C}{\partial B}\frac{\partial B}{\partial A_{\text{out}}} \tag{3-35}$$

所以这样一来不管网络多深，梯度上都会有个 1 兜底，不会为 0 而造成梯度消失。

3.4.4 Layer Normalization

Layer Normalization (LN)[12]和 Batch Normalization(BN)类似，都是规范化数据的操作。公式看起来也和 BN 一样，LN 完整的公式如下：

$$\hat{a}^l = \frac{a^l - \mu^l}{\sqrt{(\sigma^l)^2 + \varepsilon}} \tag{3-36}$$

其中，u^l 代表第 l 个的均值，计算公式如下：

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \tag{3-37}$$

σ^l 代表第 l 个标准差，计算公式如下：

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{3-38}$$

标准差和均值的计算公式和普通的没什么区别，其实重点是什么叫作第 l 个。只要把 BN 和 LN 的区别理解了就能理解 l 的含义，如图 3-33 所示。

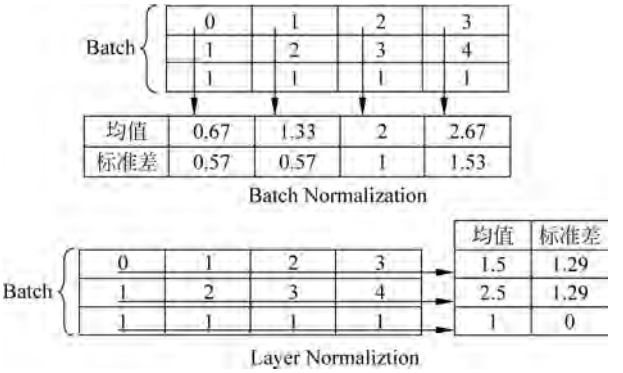


图 3-33 Batch Normalization 与 Layer Normalization 的区别

图 3-33 中三行四列的表格就代表一个张量，行数是批次数量，根据这个图 BN 与 LN 的差别就显而易见了。BN 是计算一批次向量在同一维下的均值与标准差，而 LN 计算中的均值、标准差其实与批次无关，它是计算每个向量自身的均值与标准差，所以 LN 公式中的 l 代表的是第 l 个向量。公式中的 H 代表向量维度。

LN 在 PyTorch 中也有现成的 API：

```
#传入向量维度,以便初始化 LN
ln = torch.nn.LayerNorm( e_dim )
#前向传播时直接将张量输入即可
out = ln(x)
```

3.4.5　前馈神经网络层

3min

Transformer 中所谓的前馈神经网络是 MLP 结构,非常简单,代码如下:

```
#recbyhand\chapter3\s47_transformerOnlyEncoder.py
#前馈神经网络
class FeedForward(nn.Module):

    def __init__( self, e_dim, ff_dim, drop_rate = 0.1 ):
        super().__init__()
        self.l1 = nn.Linear( e_dim, ff_dim )
        self.l2 = nn.Linear( ff_dim, e_dim )
        self.drop_out = nn.DropOut( drop_rate )

    def forward( self, x ):
        outs = self.l1( x )
        outs = self.l2( self.drop_out( torch.ReLU( outs ) ) )
        return outs
```

唯一值得一提的是在这个 MLP 中,输入向量和输出向量是一样的,中间隐藏层的维度可随意调整。

至此,单个的"编码器",即如图 3-31 所示的传播方式大家应已理解,而在讲解完整的"编码器"前,还有一个重要的内容也需要讲解,即位置编码。

3.4.6　位置编码

14min

注意在图 3-30 中,不管是左边的编码器还是右边的解码器,在输入的 Embedding 与后面的网络块之间有一步 Positional Encoding 操作,即位置编码。

为什么要进行位置编码? 因为在之后的"多头注意力层"与"前馈神经网络层"中的网络并不像 RNN 一样天生具备前后位置信息。虽然 Transformer 的作者认为 Attention Is All You Need(你仅需注意力),但是毕竟序列向量本身具备的位置信息还是很有利用价值的,所以在 Transformer 中还是引入了位置编码的操作。

位置编码究竟如何做的呢? 参看下面的 3 个公式:

$$\mathrm{emb}_{out} = \mathrm{emb}_{in} + \mathrm{PE}_{in} \tag{3-39}$$

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{d_{\mathrm{model}}}}}\right) \tag{3-40}$$

$$\mathrm{PE}(\mathrm{pos},2i+1)=\cos\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{d_{\mathrm{model}}}}}\right) \tag{3-41}$$

PE 代表该样本的位置编码向量,式(3-39)的意思是经过位置编码层的传递后输出自身加上位置编码向量的和。

在式(3-40)与式(3-41)这两个公式中 d_{model} 代表这个模型中此时输入向量的维度。pos 代表该输入的样本在序列中的位置,从 0 开始。$2i$ 和 $2i+1$ 得看作两个整体,$2i$ 代表该向量中第 $2i$ 偶数位,$2i+1$ 是第 $2i+1$ 奇数位,如图 3-34 所示。

	$2i=0$	$2i+1=1$	$2i=2$	$2i+1=3$	$2i=4$	$2i+1=5$
pos=0	x_0	x_1	x_2	x_3	x_4	x_5
pos=1	x_0	x_1	x_2	x_3	x_4	x_5
pos=2	x_0	x_1	x_2	x_3	x_4	x_5

图 3-34 位置编码中 pos 与 i 意义的示意图

假设图 3-34 中表格的序列长度为 3,每个向量长度为 6。pos 相当于它的行数,$2i$ 或者 $2i+1$ 是向量中第几位的值。这 3 个序列在推荐场景就代表 3 个用户历史交互的物品,通过 pos 自然就知道了位置的信息。

至于偶数位计算一个 sin 函数的值,奇数位计算一个 cos 函数的值的这种计算方式,是为了利用三角函数的性质,使 PE($M+N$)可由 PE(M)与 PE(N)计算得到。

以下是三角函数性质的公式:

$$\begin{aligned}\sin(\alpha+\beta)&=\sin\alpha\cos\beta+\cos\alpha\sin\beta\\ \cos(\alpha+\beta)&=\cos\alpha\cos\beta-\sin\alpha\sin\beta\end{aligned} \tag{3-42}$$

所以将 sin(*)=PE(*,$2i$)和 cos(*)=PE(*,$2i+1$)代入式(3-42)可有

$$\begin{aligned}\mathrm{PE}(M+N,2i)&=\mathrm{PE}(M,2i)\times\mathrm{PE}(N,2i+1)+\mathrm{PE}(M,2i+1)\times\mathrm{PE}(N,2i)\\ \mathrm{PE}(M+N,2i+1)&=\mathrm{PE}(M,2i+1)\times\mathrm{PE}(N,2i+1)+\mathrm{PE}(M,2i)\times\mathrm{PE}(N,2i)\end{aligned} \tag{3-43}$$

如此编码后,各个位置可以相互计算得到,所以每个向量都包含了相对位置的信息。

位置编码层的代码如下:

```
#recbyhand\chapter3\s47_transformerOnlyEncoder.py
#位置编码
class PositionalEncoding(nn.Module):

    def __init__( self, e_dim, DropOut = 0.1, max_len = 512 ):
        super().__init__()
        self.DropOut = nn.DropOut( p = DropOut )
        pe = torch.zeros( max_len, e_dim )
```

```
        position = torch.arange( 0, max_len ).unsqueeze( 1 )
        div_term = 10000.0 ** ( torch.arange( 0, e_dim, 2 ) / e_dim )
        #偶数位计算 sin,奇数位计算 cos
        pe[ :, 0::2 ] = torch.sin( position / div_term )
        pe[ :, 1::2 ] = torch.cos( position / div_term )
        pe = pe.unsqueeze(0)
        self.pe = pe

    def forward( self, x ):
        x = x + Variable( self.pe[:, : x.size( 1 ) ], requires_grad = False )
        return self.DropOut( x )
```

目前业内对位置编码存在争议,基本认为通过位置处理的信息会在之后的注意力层消失。究竟在实际应用中情况如何本书就不讨论了,有兴趣的同学可以在网上搜索相关话题。

3.4.7 Transformer Encoder

5min

Transformer Encoder(Transformer 编码器)中的所有网络块细节已经介绍完毕。完整的 Encoder 网络如图 3-35 所示。

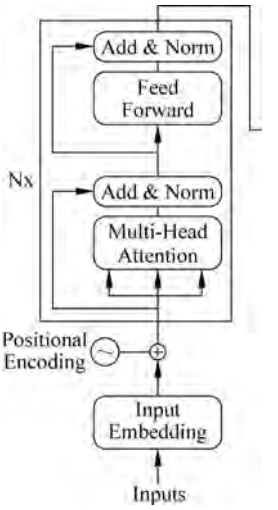


图 3-35 Transformer 编码器[10]

首先输入[批次数量,序列个数]的一个张量,经 Embedding 后得到[批次数量,序列个数,向量维度]的张量;加与位置编码,进入多头注意力层,并与多头注意力层的输出进行残差连接,之后进行 Layer Normalization 操作,然后进入前馈神经网络中传递,最后仍然是残差与 LN 操作。重复从注意力层开始的操作 N 次。最后输出的还是[批次数量,序列个数,向量维度]的张量。这是一个 Transformer 编码器的传播过程。

接上文的一些代码,给出一个编码层的代码如下:

```
#recbyhand\chapter3\s47_transformerOnlyEncoder.py
#编码层
class EncoderLayer(nn.Module):

    def __init__( self, e_dim, h_dim, n_heads, drop_rate = 0.1 ):
        '''
        :param e_dim: 输入向量的维度
        :param h_dim: 注意力层中间隐含层的维度
        :param n_heads: 多头注意力的头目数量
        :param drop_rate: drop out 的比例
        '''
```

```
        super().__init__()
        #初始化多头注意力层
        self.attention = MultiHeadAttentionLayer( e_dim, h_dim, n_heads )
        #初始化注意力层之后的 LN
        self.a_LN = nn.LayerNorm(e_dim)
        #初始化前馈神经网络层
        self.ff_layer = FeedForward(e_dim, e_dim//2)
        #初始化前馈网络之后的 LN
        self.ff_LN = nn.LayerNorm(e_dim)

        self.drop_out = nn.DropOut(drop_rate)

    def forward(self, seq_inputs ):
        #seq_inputs = [batch, seqs_len, e_dim]
        #多头注意力,输出维度[ batch, seq_lens, e_dim ]
        outs_ = self.attention( seq_inputs )
        #残差连接与 LN,输出维度[ batch, seq_lens, e_dim ]
        outs = self.a_LN( seq_inputs + self.drop_out( outs_ ) )
        #前馈神经网络,输出维度[ batch, seq_lens, e_dim ]
        outs_ = self.ff_layer( outs )
        #残差与 LN,输出维度[ batch, seq_lens, e_dim ]
        outs = self.ff_LN( outs + self.drop_out( outs_) )
        return outs
```

完整“编码器”的代码如下：

```
#recbyhand\chapter3\s47_transformerOnlyEncoder.py
class TransformerEncoder(nn.Module):

    def __init__(self, e_dim, h_dim, n_heads, n_layers, drop_rate = 0.1 ):
        '''
        :param e_dim: 输入向量的维度
        :param h_dim: 注意力层中间隐含层的维度
        :param n_heads: 多头注意力的头目数量
        :param n_layers: 编码层的数量
        :param drop_rate: drop out 的比例
        '''
        super().__init__()
        #初始化位置编码层
        self.position_encoding = PositionalEncoding( e_dim )
        #初始化 N 个编码层
        self.encoder_layers = nn.ModuleList( [EncoderLayer( e_dim, h_dim, n_heads, drop_rate )
                                              for _ in range( n_layers )] )
    def forward( self, seq_inputs ):
        '''
```

```
        :param seq_inputs: 经过 Embedding 层的张量,维度是[ batch, seq_lens, dim ]
        :return: 与输入张量维度一样的张量,维度是[ batch, seq_lens, dim ]
        '''
        #先进行位置编码
        seq_inputs = self.position_encoding( seq_inputs )
        #输入 N 个编码层中开始传播
        for layer in self.encoder_layers:
        seq_inputs = layer( seq_inputs )

        return seq_inputs
```

在讲解解码器前,可以先介绍利用 Transformer 编码器的推荐算法。本身 Transformer 编码器、解码器的结构是为了用序列预测序列任务的有效结构。对于推荐来讲,Transformer 编码器起信息聚合的作用。

3.4.8　利用 Transformer 编码器的推荐算法 BST

9min

处于序列推荐算法前沿地位的阿里巴巴自然不会错过 Transformer。2019 年阿里巴巴团队提出了算法 BST,论文名叫作 Behavior Sequence Transformer for E-commerce Recommendation in Alibaba。其中 Behavior Sequence 是行为序列的意思。图 3-36 截取自该论文,展示了 BST 模型完整的结构。

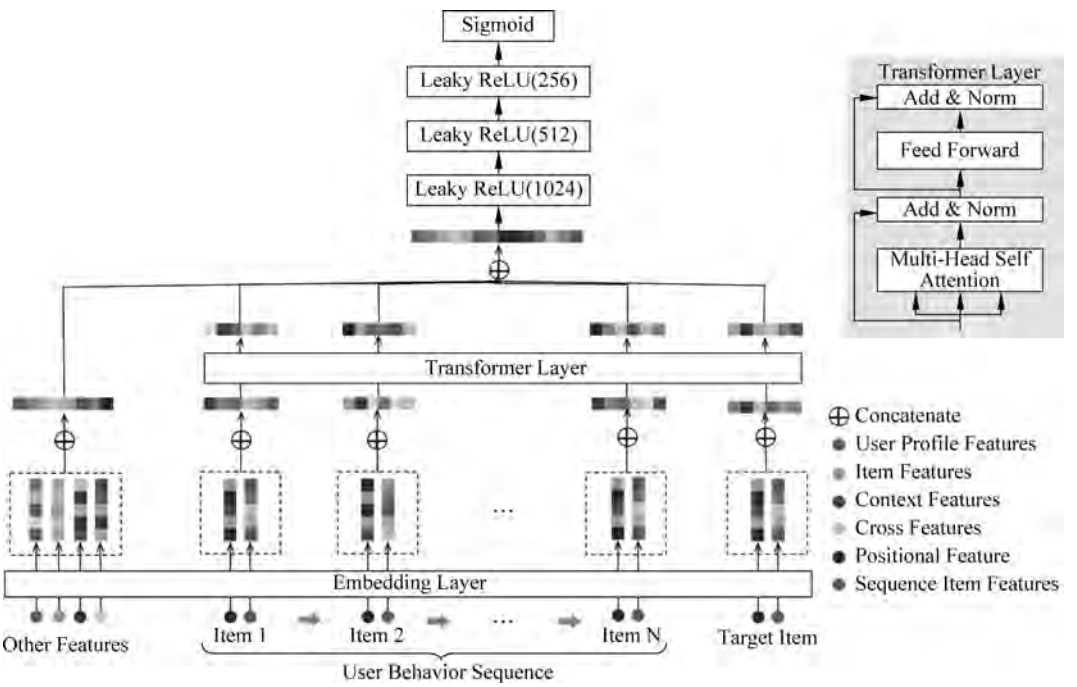


图 3-36　BST 模型结构示意图[13]

其中的 Transformer Layer 是 Transformer 的“编码器”,了解过 Transformer 编码器之后基本上应该能够比较容易地理解 BST 模型。

图 3-36 中 User Behavior Sequence 是用户历史交互物品序列,包括目标物品在内经 Transformer 传递后拼接在一起进行 MLP 的传播。最左边的 Other Features 表示拼一些其他的特征向量,这不重要,在示例代码中已将其省略。

BST 的核心代码如下:

```
#recbyhand\chapter3\s48_BST.py
from chapter3 import s47_transformerOnlyEncoder as TE

class BST( nn.Module ):

    def __init__( self, n_items, all_seq_lens, e_dim = 128, n_heads = 3, n_layers = 2 ):
        '''
        :param n_items: 总物品数量
        :param all_seq_lens: 序列总长度,包含历史物品序列及目标物品
        :param e_dim: 向量维度
        :param n_heads: Transformer 中多头注意力层的头目数
        :param n_layers: Transformer 中的 encoder_layer 层数
        '''
        super( BST, self ).__init__()
        self.items = nn.Embedding( n_items, e_dim, max_norm = 1 )
        self.transformer_encoder = TE.TransformerEncoder( e_dim, e_dim//2, n_heads,n_layers )
        self.mlp = self.__MLP( e_dim * all_seq_lens )

    def __MLP( self, dim ):
        return nn.Sequential(
            nn.Linear( dim, dim//2 ),
            nn.LeakyReLU( 0.1 ),
            nn.Linear( dim//2, dim//4 ),
            nn.LeakyReLU( 0.1 ),
            nn.Linear( dim//4, 1 ),
            nn.Sigmoid() )

    def forward(self, x, target_item):
        #[ batch_size, seqs_len, dim ]
        item_embs = self.items( x )
        #[ batch_size, 1, dim ]
        one_item = torch.unsqueeze( self.items( target_item ), dim = 1 )
        #[ batch_size, all_seqs_len, dim ]
        all_item_embs = torch.cat([ item_embs, one_item ], dim = 1 )
        #[ batch_size, all_seqs_len, dim ]
        all_item_embs = self.transformer_encoder( all_item_embs )
        #[ batch_size, all_seqs_len * dim ]
```

```
        all_item_embs = torch.flatten( all_item_embs, start_dim = 1 )
        #[ batch_size, 1 ]
        logit = self.mlp( all_item_embs )
        #[ batch_size ]
        logit = torch.squeeze(logit)
        return logit
```

因为事先已经实现了 Transformer Encoder，此处直接调用即可，所以代码写起来很简单。更多细节可参阅完整代码。

值得一提的是像这种深度序列模型，其中要学习的模型参数很多，没有足够量级的数据实际上是学不出来的，示例代码中用的 MovieLens 数据其实并不够，所以大家不要盲目地觉得只要把模型搞深搞得更复杂最终效果一定会更好，实际绝非如此。如果数据量少，那就老老实实地用最基础的 FM，效果一定远胜这些序列模型。模型的选型及改造一定要结合实际场景，不要纸上谈兵凭空构造模型。

3.4.9　Transformer Decoder

22min

接下来介绍 Transformer 的 Decoder，即解码器。解码器的完整结构如图 3-37 所示。

因为是序列预测序列的任务，所以解码器的输入部分在图 3-37 中称为 Outputs，在训练 Transformer 模型时，输入解码器的是标注样本。

从解码器的输入开始，进行 Embedding 之后加上位置编码，然后开始进入 N 个解码层传播。每个解码层与编码层差不多，但是有略微差异。

首先是 Masked Multi-Head Attention，直译叫作遮盖的多头注意力层。遮盖的意义是为了将未来信息掩盖住，使训练出来的模型更准确。从自然语言处理的角度来举个例子，例如输入“我爱吃冰棍”，假设是 RNN 模型，则当轮到要预测“冰”这个字时，模型获得的信息自然是“我爱吃”这 3 个字，但是 Transformer 的 Attention 层在做计算时将一整个序列张量输入后进行计算，所以如果不加处理，当预测“冰”这个字时，模型获得的信息将会是“我爱吃棍”这 4 个字。“棍”这个字对于“冰”来讲显然属于未来信息，因为它在实际预测生成序列时是不可能出现的，所以在训练时需要将未来信息都遮盖住。使模型在预测“冰”时，获得的信息是“我爱吃 ** ”。在预测“吃”时，获得的信息是“我爱 *** ”。

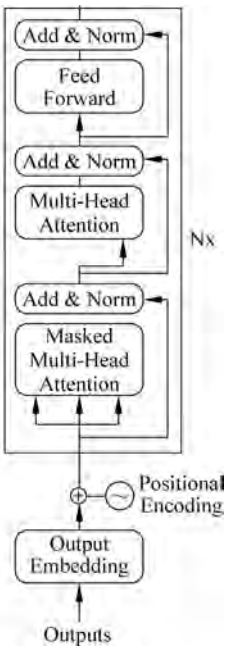


图 3-37　Transformer 解码器[10]

如果是商品序列也是一样的道理，例如模型的任务是利用用户 $t-1$ 时刻的商品序列，预测用户 t 时刻的商品序列。在训练模型时，作为 t 时刻的商

品序列假设是[商品1,商品2,商品3,商品4,商品5],则在预测序列中第3个商品也是商品3时,作为t给模型的信息应该是[商品1,商品2,*,*,*]。具体的代码如下:

```
#recbyhand\chapter3\s49_transformer.py
#生成mask序列
def subsequent_mask( size ):
    subsequent_mask = torch.triu(torch.ones( (1, size, size) )) == 0
    return subsequent_mask
```

其中,size指的是序列长度。这个函数的输出数据的形式如下:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

然后在One Head Attention的前向传播方法中加入以下代码:

```
QK = QK.masked_fill( mask == 0, -1e9 )
```

这么做的效果是将***QK***张量中与mask张量中值为0的对应位置的值给消除了。从而达到遮盖未来信息的效果。

在图3-37中,在Masked Multi-Head Attention上还有个多头注意力层,并且第2个注意力层有根线是从左边的编码器层连过来的。这个注意力层称为交互注意力层,编码器中的注意力层及解码器第1个注意力层其实叫作自注意力层,区别就在于交互注意力层负责编码器与解码器信息的传递。还记得在3.4.2节讲过解码器中注意力层的Query向量是由编码器的输出向量计算而来的吗?指的就是这个交互注意力层。

注意力层加入两个新逻辑后,代码也要相应地变化一下。变化后的注意力代码如下:

```
#recbyhand\chapter3\s49_transformer.py
#多头注意力层
class MultiHeadAttentionLayer( nn.Module ):

    def __init__( self, e_dim, h_dim, n_heads ):
        super().__init__()
        self.atte_layers = nn.ModuleList([ OneHeadAttention( e_dim, h_dim ) for _ in range( n_
        heads ) ] )
        self.l = nn.Linear( h_dim * n_heads, e_dim)

    def forward( self, seq_inputs, querys = None, mask = None ):
        outs = []
        for one in self.atte_layers:
            out = one( seq_inputs, querys, mask )
        outs.append( out )
```

```
        outs = torch.cat( outs, dim = -1 )
        outs = self.l( outs )
        return outs

#单头注意力层
class OneHeadAttention( nn.Module ):

    def __init__( self, e_dim, h_dim ):
        super().__init__()
        self.h_dim = h_dim

        #初始化 Q、K、V 的映射线性层
        self.lQ = nn.Linear( e_dim, h_dim )
        self.lK = nn.Linear( e_dim, h_dim )
        self.lV = nn.Linear( e_dim, h_dim )

    def forward( self, seq_inputs , querys = None, mask = None ):
        '''
        #如果有 Encoder 的输出,则映射该张量,否则还是自注意力的逻辑
        if querys is not None:
            Q = self.lQ( querys ) #[ batch, seq_lens, h_dim ]
        else:
            Q = self.lQ( seq_inputs ) #[ batch, seq_lens, h_dim ]
        K = self.lK( seq_inputs ) #[ batch, seq_lens, h_dim ]
        V = self.lV( seq_inputs ) #[ batch, seq_lens, h_dim ]
        QK = torch.matmul( Q,K.permute( 0, 2, 1 ) )
        QK /= ( self.h_dim ** 0.5 )

        #将对应 Mask 序列中 0 的位置变为 -1e9,意为遮盖掉此处的值
        if mask is not None:
            QK = QK.masked_fill( mask == 0, -1e9 )

        a = torch.Softmax( QK, dim = -1 )
        outs = torch.matmul( a, V )
        return outs
```

主要的变化是加入了 mask 机制及用 querys 容器作为来自编码器输出的向量。一个完整的 Transformer 解码器的代码如下：

```
#recbyhand\chapter3\s49_transformer.py
#解码层
class DecoderLayer(nn.Module):

    def __init__( self, e_dim, h_dim, n_heads, drop_rate = 0.1 ):
        '''
```

```
        :param e_dim: 输入向量的维度
        :param h_dim: 注意力层中间隐含层的维度
        :param n_heads: 多头注意力的头目数量
        :param querys: Encoder 的输出
        :param drop_rate: drop out 的比例
        '''
        super().__init__()

        self.self_attention = MultiHeadAttentionLayer( e_dim, h_dim, n_heads )
        self.sa_LN = nn.LayerNorm( e_dim )
        self.interactive_attention = MultiHeadAttentionLayer( e_dim, h_dim, n_heads )
        self.ia_LN = nn.LayerNorm (e_dim )
        self.ff_layer = FeedForward( e_dim, e_dim//2 )
        self.ff_LN = nn.LayerNorm( e_dim )

self.drop_out = nn.DropOut( drop_rate )

    def forward( self, seq_inputs , querys, mask ):
        '''
        :param seq_inputs: [ batch, seqs_len, e_dim ]
        :param mask: 遮盖位置的标注序列 [ 1, seqs_len, seqs_len ]
        '''
        outs_ = self.self_attention( seq_inputs , mask = mask )
        outs = self.sa_LN( seq_inputs + self.drop_out( outs_ ) )
        outs_ = self.interactive_attention( outs, querys )
        outs = self.ia_LN( outs + self.drop_out(outs_) )
        outs_ = self.ff_layer( outs )
        outs = self.ff_LN( outs + self.drop_out( outs_) )
        return outs

class TransformerDecoder(nn.Module):

    def __init__(self, e_dim, h_dim, n_heads, n_layers, drop_rate = 0.1 ):
        '''
        :param e_dim: 输入向量的维度
        :param h_dim: 注意力层中间隐含层的维度
        :param n_heads: 多头注意力的头目数量
        :param n_layers: 编码层的数量
        :param drop_rate: drop out 的比例
        '''
        super().__init__()
        self.position_encoding = PositionalEncoding( e_dim )
        self.decoder_layers = nn.ModuleList( [DecoderLayer( e_dim, h_dim, n_heads, drop_
        rate )for _ in range( n_layers )] )
    def forward( self, seq_inputs, querys ):
        '''
```

```
        :param seq_inputs: 经过 Embedding 层的张量,维度是[ batch, seq_lens, dim ]
        :return: 与输入张量维度一样的张量,维度是[ batch, seq_lens, dim ]
        '''
        seq_inputs = self.position_encoding( seq_inputs )
        mask = subsequent_mask( seq_inputs.shape[1] )
        for layer in self.decoder_layers:
            seq_inputs = layer( seq_inputs, querys, mask )
        return seq_inputs
```

在配套代码中有更多的注释及代码内容,大家可参阅。

3.4.10　结合 Transformer 解码器的推荐算法推导

12min

了解了 Transformer 解码器后,大家想到如何将其应用在推荐算法上了吗? 是的,可以设计一个联合训练的机制。既然编码器、解码器的构造本身利用序列预测序列的任务,则可以用用户的历史交互物品序列去预测错一位的历史交互物品序列,例如用[物品 1,物品 2,物品 3,物品 4,物品 5]去预测[物品 2,物品 3,物品 4,物品 5,物品 6]。就像 3.1.6 节中讲过的一样。

完整的过程如图 3-38 所示。

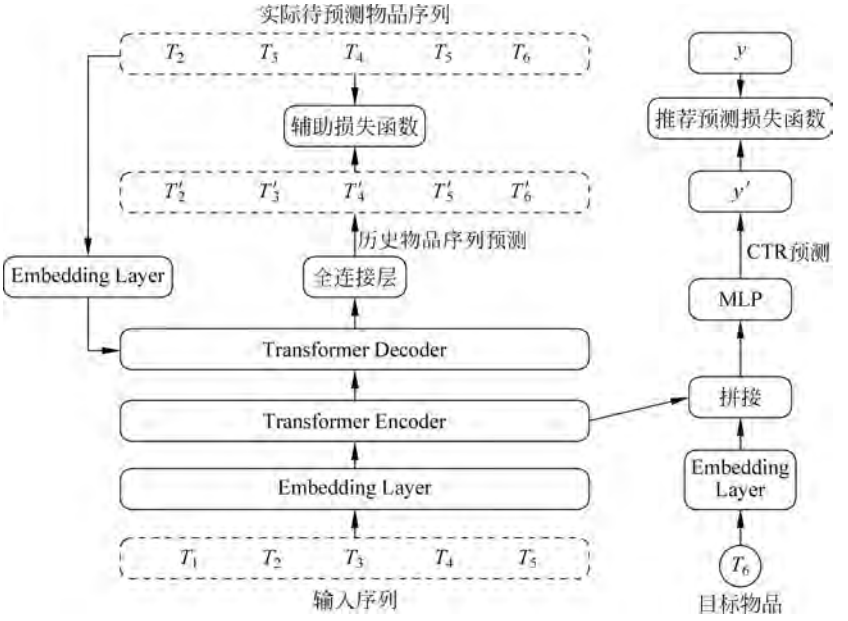


图 3-38　利用 Transformer Encoder 和 Decoder 联合训练

这次编码器的输出张量不仅与目标物品向量拼接,从而预测最终的 CTR,也会同时传递给解码器。训练时,被预测的序列也经 Embedding 之后传入解码器,与编码器的输出进行 Transformer 解码器内部的传递,将输出的张量经全连接层传递后再与要被预测的序列

建立交叉熵损失函数作为辅助损失函数。

将推荐预测的损失函数与辅助损失函数相加得到最终损失函数,中间可给辅助损失函数添加一个权重来调整序列预测所占整体损失函数的比重。

本节代码的地址为 recbyhand\chapter3\s410_transformer_rec.py。

核心代码如下:

```
#recbyhand\chapter3\s410_transformer_rec.py
class Transformer4Rec( nn.Module ):

    def __init__( self, n_items, all_seq_lens, e_dim = 128, n_heads = 3, n_layers = 2 ,
alpha = 0.2):
        '''
        :param n_items: 总物品数量
        :param all_seq_lens: 序列总长度,包含历史物品序列及目标物品
        :param e_dim: 向量维度
        :param n_heads: Transformer 中多头注意力层的头目数
        :param n_layers: Transformer 中的 encoder_layer 层数
        :param alpha: 辅助损失函数的计算权重
        '''
        super( Transformer4Rec, self ).__init__()
        self.items = nn.Embedding( n_items, e_dim, max_norm = 1 )
        self.encoder = TE.TransformerEncoder( e_dim, e_dim//2, n_heads, n_layers )
        self.mlp = self.__MLP(e_dim * all_seq_lens)
        self.BCEloss = nn.BCELoss()

        self.decoder = TE.TransformerDecoder( e_dim, e_dim//2, n_heads, n_layers )
        self.auxDense = self.__Dense4Aux(e_dim, n_items)
        self.crossEntropyLoss = nn.CrossEntropyLoss( )
        self.alpha = alpha
        self.n_items = n_items

    def __MLP( self, dim ):
        return nn.Sequential(
            nn.Linear( dim, dim//2 ),
            nn.LeakyReLU( 0.1 ),
            nn.Linear( dim//2, dim//4 ),
            nn.LeakyReLU( 0.1 ),
            nn.Linear( dim//4, 1 ),
            nn.Sigmoid( ) )

    def __Dense4Aux( self, dim, n_items ):
        return nn.Sequential(
            nn.Linear( dim, n_items ),
            nn.Softmax( ) )
```

```
    #历史物品序列预测的前向传播
    def forwardPredHistory( self, outs, history_seqs ):
        history_seqs_embds = self.items(history_seqs)
        outs = self.decoder(history_seqs_embds, outs)
        outs = self.auxDense( outs )
        outs = outs.reshape( -1, self.n_items )
        history_seqs = history_seqs.reshape( -1 )
        return self.alpha * self.crossEntropyLoss( outs, history_seqs )

    #推荐预测的前向传播
    def forwardRec(self,item_embs,target_item,target_label):
        logit = self.__getReclogit(item_embs,target_item)
        return self.BCEloss(logit,target_label)

    def __getReclogit(self,item_embs,target_item):
        one_item = torch.unsqueeze(self.items(target_item), dim=1)
        all_item_embs = torch.cat([item_embs, one_item], dim=1)
        all_item_embs = torch.flatten(all_item_embs, start_dim=1)
        logit = self.mlp(all_item_embs)
        logit = torch.squeeze(logit)
        return logit

    def forward( self, x, history_seqs, target_item , target_label ):
        item_embs = self.items( x )
        item_embs = self.encoder( item_embs )
        recLoss = self.forwardRec( item_embs,target_item,target_label )
        auxLoss = self.forwardPredHistory( item_embs, history_seqs )
        return recLoss + auxLoss
```

配套代码里有更详细的注释，大家结合本节配图及代码的注释一定能看懂代码。

该算法的确将 Transformer 的 Decoder 也利用了起来，但是也正因如此要学习的模型参数量会变得更多，所以在数据量不大时还应慎用此算法。另外也可将序列预测序列的辅助任务改进成通过序列去预测该物品序列对应的历史交互标注序列，等于做成了二分类的预测，这样可以大大减少模型学习的难度。

3.5　本章总结

经过本章的学习，大家应该已经具备了用深度学习的方式搭建出自己的推荐算法。CNN、RNN、联合训练、注意力机制等的用法应该已经掌握，而 FM 可以演化出很多算法，也可在某些大型的神经网络中插入 FM 结构。

模型参数越多拟合能力越强，但也越容易过拟合，所以像序列推荐这个系列的模型尽可

能在数据量大的场景使用才会有效。尤其在用到注意力机制甚至是多头注意力时,需要更多的数据才能学出效果。

正如本章开头所讲,除了书中的这些算法外,实际上还有很多推荐算法,包括后两个章节要介绍的图神经网络推荐算法与知识图谱推荐算法。FM 演化出的算法至今仍然在更新,当然序列推荐算法系列也没有停止更新的迹象,大家有兴趣可以自行查阅最前沿的推荐算法参考学习,而 3.1.1 节的内容可能反而更为关键,因为当你有推导算法的思路时,阅读前沿的论文学习他人的算法也会变得非常简单。

参考文献