PNN [2016]

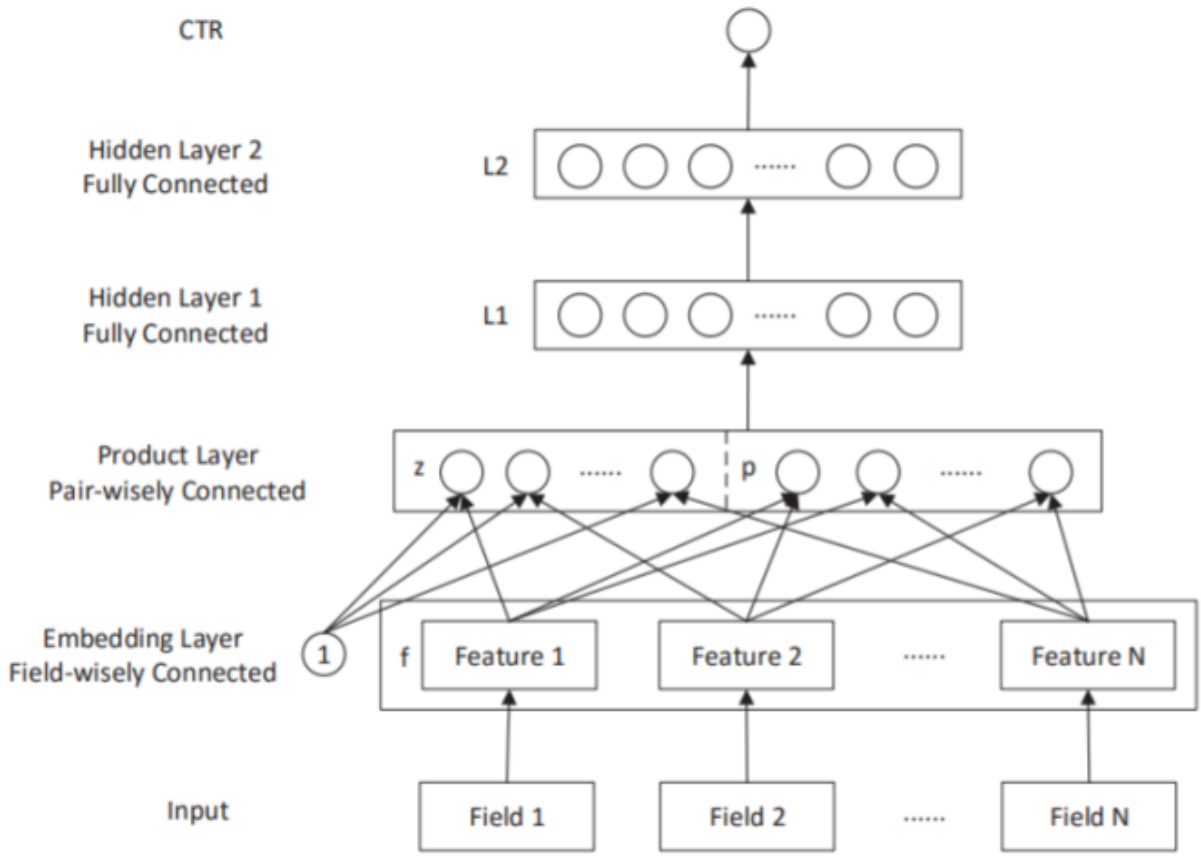
如果说FNN是在DNN时代的一个初步尝试的话，那么PNN就是真正意义上把FM融入到了DNN之中。

回忆一下FNN的操作，它是将FM预训练的embedding直接concat之后输入到DNN中，声称这样能够把握高阶交互特征。然而，DNN中的操作是“add”，而不是"product",参见下面这个图。输入向量中的蓝色、棕色、黄色分别代表三个不同特征的embedding，乘上一个矩阵之后得到转换后的向量。但是我们仔细看一下这个乘矩阵的过程：还是蓝色向量和蓝色矩阵部分相乘、棕色向量和棕色矩阵部分相乘、黄色向量和黄色矩阵部分相乘，之后再**加起来**，得到输出向量的一位。而蓝色、棕色、黄色向量本身并没有发生交互啊。

相比Embedding+MLP的传统结构，PNN在embedding层后设计了**Product Layer**，以显示捕捉基于Field的二阶特征（其实，这个product layer就是一个FM）。这是因为NN层之间的"**add** operation"不足以捕获不同的Field特征间的相关性, 而 "**product**" 相比 "add" 能更好得捕捉特征间的dependence，因此作者希望在NN中显示地引入"product"操作，从而更好地学习不同Field特征间的相关性。

> The 'add' operations of the perceptron layer might not be useful to explore the interactions of categorical data in multiple fields.

同时，PNN使用的是端到端训练，并没有使用FM预训练的embedding。



product layer分成了两个部分：

- 一个部分称为z，是linear部分(一阶项)，实际就是原始的embedding concat到一起。为了保持格式上的统一性，不妨把它当成原始embedding和1的内积。

- 另一部分称为p，就是真正的product产生的部分。product部分其实就是FM，计算所有特征的两两product，得到$n(n-1)/2$个数。计算product的时候可以选择是使用“内积”还是“外积”，也就是所谓的IPNN和OPNN。内积就和图上画的一样，直接对两个 f 做**内积**得到 p 就好了。但是外积怎么做呢？按照外积的定义：$g(f_i, f_j) = f_i f_j^T$ 这样得到的结果是一个emb_size*emb_size的矩阵，要想把它映射成一个数，就还需要一个3维的矩阵来做转换。如果每次都先算完两两的外积然后再转换成一个数，这样的复杂度是有点爆炸的。这篇文章在这里做了一个近似：先计算embedding layer的所有embedding f 的和，再做外积，即 $p = f_{\sum} f_{\sum}^T$ 。这样先计算求和的复杂度不高，最后把它转化为L_1层即可。

PNN这个模型结构相对来说是比较舒服的，每一步操作都挺自然，也不需要两阶段。

Neural Factorization Machine (NFM) [2017]:

> FM之精髓，其上在于latent embedding，有了它才能把**交互拆解**，提升泛化能力；居中在于element-wise乘，能让两个特征之间交互；其下在于**点积**，把好不容易带进来的高维信息全部压缩完了。

FM的精髓实际在于利用了隐式的embedding，但是点积实质上仍然对信息做了压缩。当我们做到element-wise这一步，其实已经把FM的优势大体上包进来了。然而如果下一步做**sum，会损失一些信息**，那么有没有办法，取FM的精髓，但又不造成信息损失呢?

1. NFM

出自论文 Neural Factorization Machines for Sparse Predictive Analytics 。在NFM这个方法中，作者认为**点积某种程度上限制了FM的能力上限**，因为embedding比较长，是包含更多信息的，而点积之后就**只剩下一个数了**。比如PNN，就是计算两两点积，然后输入到下一层DNN中。如果仅仅做到element-wise就停止，然后往下直接接DNN，这样就能够保存长的embedding，而不会导致信息丢失。那多是一件美事!

NFM的计算公式:

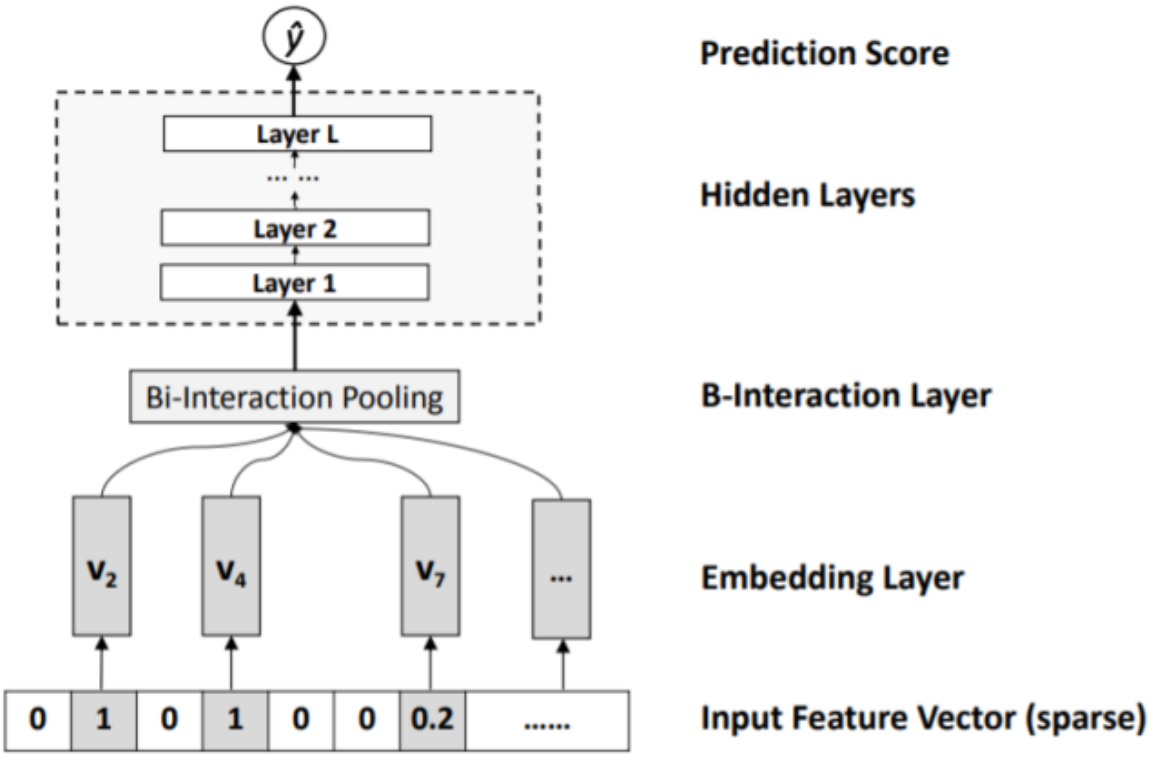
$$\hat{y}_{NFM}(x) = w_0 + \sum_{i=1}^{n} w_i x_i + f(\mathbf{x})$$

其中，x_i$是one-hot向量。

下图只表示了

$$f(\mathbf{x})$$

部分:



重要的模块是**Bi-Interaction Pooling**，用来建模二阶交叉特征。Bi-Interaction Pooling的公式：

$$f_{BI}(V_x) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} x_i v_i \odot x_j v_j \qquad (2)$$

其中 $\odot$ 表示element-wise乘法，即哈达玛积，表示两个向量对应元素相乘，其结果为一个向量。所以，Bi-Interation Layer其实就是将有值的embedding vectors 进行**两两哈达玛积**运算，然后将所有向量进行**对应元素求和**，最终 $f_{BI}(V_x)$ 为pooling之后的一个向量。

Bi-Interaction的原型还是按照原始FM来的，把所有的交叉都表示到**element-wise乘法**这一步，就停止，不再做sum了，防止信息的丢失。将长embedding pooling之后作为输入送进DNN中。我们都知道DNN具有很好的**非线性**，二阶信息叠加高度非线性，那我们其实是有理由展望一下**更高阶**的交叉**可能**存在。

2. 总结

NFM的思想：让模型在浅层尽可能包含更多的信息、减小信息丢失，来降低后续下游DNN的学习负担，这样DNN可以使用较浅的网络和较少的参数。

其实在NFM这里隐隐透露出可以做更高阶的交叉的影子了，但是为什么我们只强调是可能存在呢，因为MLP不会替你做交叉，而且这里的交叉还是不如DCN里面**形式上那么明显**，是隐式的特征交叉。NFM的主要立论点是要扩展FM的上限，强化FM的能力，其实主要目标不在高阶交叉上，而DCN这一类方法就完全是奔着高阶交叉去的了。

ONN：Operation-aware Neural Network [2019] -- FFM与NN的结合体

如果说之前的PNN是FM和DNN的结合体，那么ONN就是FFM和DNN的结合体。PNN给每个feature都给予一个embedding，然后做向量的点积。但是，ONN认为针对不同的**交叉操作**(内积or外积)、以及**不同field**的交叉，都应该用不同的Embedding。

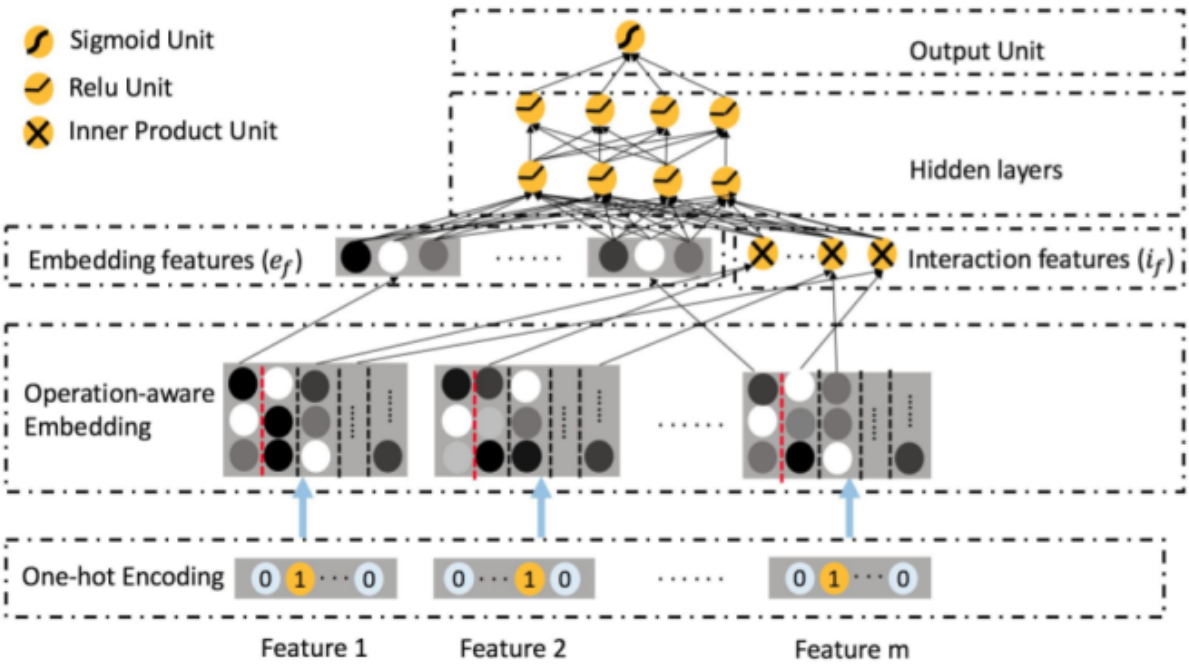
> Besides traditional fully-connected architectures, some new operations, such as Convolution operation and product operations, are proposed to learn feature interactions better. In these models, the representation is shared among different operations. However, the best representation for different operations should be different.

如果用同样的Embedding，从好处来讲，就是不同的交叉操作会对对方都有一个正则化的效果，尤其是当数据量比较少的时候，可以缓解过拟合。但是，对于CTR/CVR任务，数据量从来就不是问题！那么，还只用一个embedding的话就只能是限制模型的capacity了，就会产生FM中所说的"拉扯问题"，是sub-optimal的。

另外一个好处就是，我们可以灵活的选择特征的维度。例如对于USER ID和 EDUCATION这两个特征，显然ID应该具有更高的embedding维度，因为ID特征值很多，而EDU取值就那么几个。如果不对不同的operation选择不同的embedding，那么他俩必须是同样的embedding size，因为点积操作要求两个特征embedding大小一样。而我们使用不同的embedding之后，就可以在点积的时候让他们相等，但是在直接拼接这个操作的时候，让ID的embedding size >> EDU的embedding size。

其实，ONN的思路在本质上和FFM、AFM都有异曲同工之妙，这三个模型都是通过引入了额外的信息来区分不同特征交叉具备的信息表达。总结下来：

- FFM：引入**Field-aware**，对于field a来说，与field b交叉和field c交叉应该用不同的embedding
- AFM：引入Attention机制，a与b的交叉特征重要度与a与c的交叉重要度不同
- ONN：引入Operation-aware，field a与field b进行内积所用的embedding，不同于field a与field b进行外积用的embedding；field a与field b做内积用的embedding，亦不同于field a与field c做内积用的embedding。



上图中，一个feature有多个embedding。在图中以红色虚线为分割，第一列的embedding是feature本身的embedding，之后直接用来拼接过DNN的；从第二列开始往后是当前特征与**第n个特征**（field）以**方式m**（Operation：内积/外积）交叉所使用的embedding。