# CTR预估专栏 | 一文搞懂DeepFM的理论与实践

李宁宁 AI前线 2018-07-06

作者 | 李宁宁
来源 | 机器学习荐货情报局（ID：ML_CIA）
编辑 | Natalie

**AI 前线导读：** 认真阅读完本文，抓住 DeepFM 理论和实践的核心内容不成问题！

**更多优质内容请关注微信公众号“AI 前线”（ID：ai-front)**

## 1. CTR 预估

CTR 预估 **数据特点**：

1. 输入中包含类别型和连续型数据。类别型数据需要 one-hot, 连续型数据可以先离散化再 one-hot，也可以直接保留原值
2. 维度非常高
3. 数据非常稀疏
4. 特征按照 Field 分组

CTR 预估 **重点** 在于 **学习组合特征**。注意，组合特征包括二阶、三阶甚至更高阶的，阶数越高越复杂，越不容易学习。Google 的论文研究得出结论：高阶和低阶的组合特征都非常重要，同时学习到这两种组合特征的性能要比只考虑其中一种的性能要好。

那么关键问题转化成：**如何高效的提取这些组合特征**。一种办法就是引入领域知识人工进行特征工程。这样做的弊端是高阶组合特征非常难提取，会耗费极大的人力。而且，有些组合特征是隐藏在数据中的，即使是专家也不一定能提取出来，比如著名的“尿布与啤酒”问题。

在 DeepFM 提出之前，已有 LR，FM，FFM，FNN，PNN（以及三种变体：IPNN,OPNN,PNN*），Wide&Deep 模型，这些模型在 CTR 或者是推荐系统中被广泛使用。

## 2. 模型演进历史

### 2.1 线性模型

最开始 CTR 或者是推荐系统领域，一些线性模型取得了不错的效果。比如：LR，FTRL。**线性模型** 有个致命的缺点：**无法提取高阶的组合特征**。所以常用的做法是人为的加入 **pairwise feature interactions**。即使是这样：对于那些出现很少或者没有出现的组合特征以及高阶组合特征依旧无法提取。

LR 最大的缺点就是无法组合特征，依赖于人工的特征组合，这也直接使得它表达能力受限，基本上只能处理线性可分或近似线性可分的问题。

### 2.2 FM 模型

线性模型差强人意，直接导致了 FM 模型应运而生（在 Kaggle 上打比赛提出来的，取得了第一名的成绩）。FM 通过隐向量 **latent vector** 做内积来表示组合特征，从理论上解决了低阶和高阶组合特征提取的问题。但是实际应用中受限于计算复杂度，一般也就只考虑到 2 阶交叉特征。

后面又进行了改进，提出了 FFM，增加了 Field 的概念。

## 2.3 遇上深度学习

随着 DNN 在图像、语音、NLP 等领域取得突破，人们渐渐意识到 DNN 在特征表示上的天然优势。相继提出了使用 CNN 或 RNN 来做 CTR 预估的模型。但是，**CNN 模型的缺点是：偏向于学习相邻特征的组合特征。 RNN 模型的缺点是：比较适用于有序列（时序）关系的数据。**

FNN 的提出，应该算是一次非常不错的尝试：先使用预先训练好的 FM，得到隐向量，然后作为 DNN 的输入来训练模型。缺点在于：受限于 FM 预训练的效果。

随后提出了 PNN，PNN 为了捕获高阶组合特征，在`embedding layer`和`first hidden layer`之间增加了一个`product layer`。根据 product layer 使用内积、外积、混合分别衍生出`IPNN, OPNN, PNN*`三种类型。

无论是 FNN 还是 PNN，他们都有一个绕不过去的缺点：**对于低阶的组合特征，学习到的比较少。** 而前面我们说过，低阶特征对于 CTR 也是非常重要的。

Google 意识到了这个问题，为了同时学习低阶和高阶组合特征，提出了 **Wide&Deep 模型**。它混合了一个 **线性模型（Wide part）** 和 **Deep 模型 (Deep part)**。这两部分模型需要不同的输入，而 **Wide part** 部分的输入，依旧 **依赖人工特征工程。**

**但是**，这些模型普遍都存在两个问题：

1. 偏向于提取低阶或者高阶的组合特征。不能同时提取这两种类型的特征。
2. 需要专业的领域知识来做特征工程。

DeepFM 在 Wide&Deep 的基础上进行改进，成功解决了这两个问题，并做了一些改进，其 **优势 / 优点** 如下：

1. 不需要预训练 FM 得到隐向量
2. 不需要人工特征工程
3. 能同时学习低阶和高阶的组合特征
4. FM 模块和 Deep 模块共享 **Feature Embedding** 部分，可以更快的训练，以及更精确的训练学习

下面，就让我们走进 DeepFM 的世界，一起去看看它到底是怎么解决这些问题的！

# 3. DeepFM

DeepFM 闪亮登场！

主要做法如下：

1. **FM Component + Deep Component**。FM 提取低阶组合特征，Deep 提取高阶组合特征。但是和 Wide&Deep 不同的是，DeepFM 是端到端的训练，不需要人工特征工程。
2. **共享 feature embedding**。FM 和 Deep 共享输入和feature embedding不但使得训练更快，而且使得训练更加准确。相比之下，Wide&Deep 中，input vector 非常大，里面包含了大量的人工设计的 pairwise 组合特征，增加了他的计算复杂度。

**DeepFM 架构图：**

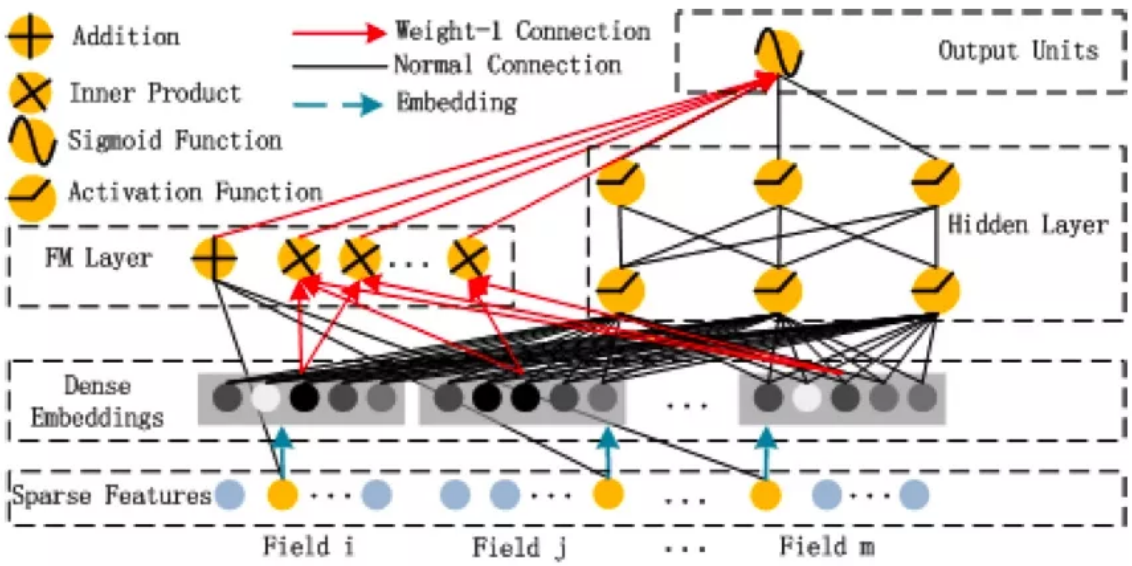



Figure 1: Wide & deep architecture of DeepFM. The wide and deep component share the same input raw feature vector, which enables DeepFM to learn low- and high-order feature interactions simultaneously from the input raw features.

## 3.1 FM Component

FM 部分的输出由两部分组成：一个 **Addition Unit**，多个 **内积单元。**

$$y_{FM} = \langle w, x \rangle + \sum_{j_1=1}^{d} \sum_{j_2=j_1+1}^{d} \langle V_i, V_j \rangle x_{j_1} \cdot x_{j_2},$$

这里的 d 是输入 one-hot 之后的维度，我们一般称之为`feature_size`。对应的是 one-hot 之前的特征维度，我们称之为`field_size`。

FM 架构图：

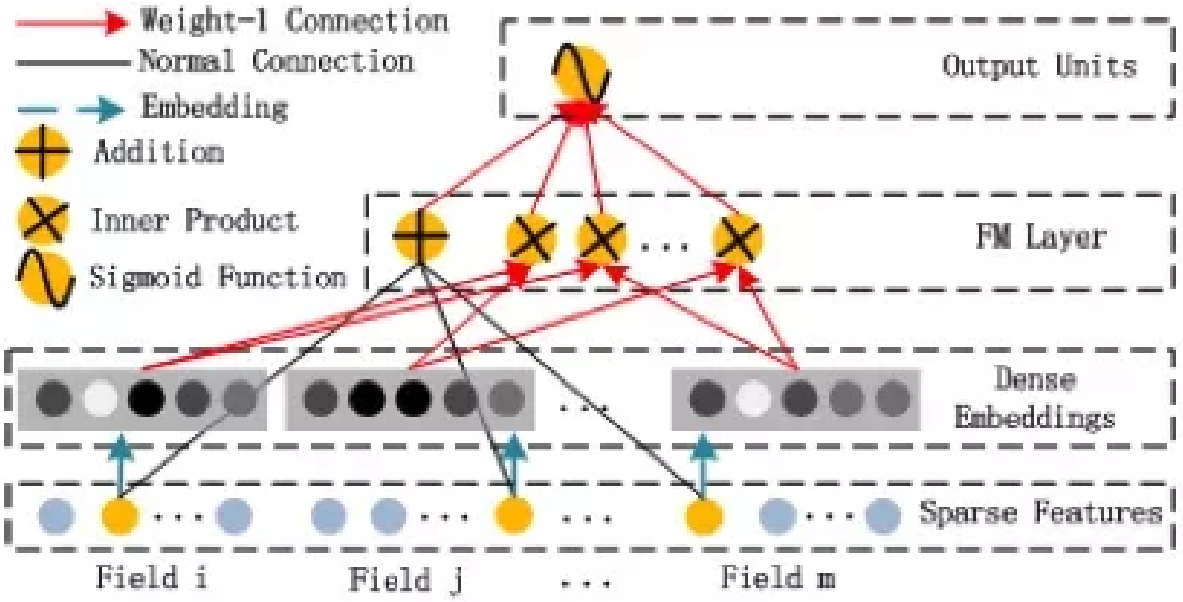



Figure 2: The architecture of FM.

**Addition Unit** 反映的是 1 阶的特征。**内积单元** 反映的是 2 阶的组合特征对于预测结果的影响。

**注意：** 虽然公式上 Y_fm 是所有部分都求和，是一个标量。但是从 FM 模块的架构图上我们可以看到，输入到输出单元的部分并不是一个标量，应该是一个向量。实际实现中采用的是 FM 化简之后的内积公式，最终的维度是：`field_size + embedding_size`

这里分别展开解释下维度的两部分是怎么来的，对于理解模型还是很重要的：

第一、`field_size` 对应的是 `<W,X>`。

这里的 X 是 one-hot 之后的，one-hot 之后，我们认为 X 的每一列都是一个单独的维度的特征。这里我们表达的是 X 的 1 阶特征，说白了就是单独考虑 X 的每个特征，他们对最终预测的影响是多少。是多少那？是 W！W 对应的就是这些维度特征的权重。假设 one-hot 之后特征数量是`feature_size`，那么 W 的维度就是 `(feature_size, 1)`。

这里<W,X>是把 X 和 W 每一个位置对应相乘相加。由于 X 是 one-hot 之后的，所以 **相当于** 是进行了一次 **Embedding**！X 在 W 上进行一次嵌入，或者说是一次选择，选择的是 W 的行，按什么选择那，按照 X 中不为 0 的那些特征对应的 index，选择 W 中 row=index 的行。

**这里解释下 Embedding**：

W 是一个矩阵，每一行对应 X 的一个维度的特征（这里是 one-hot 之后的维度，一定要注意）。W 的列数为 1，表示嵌入之后的维度是 1。W 的每一行对应一个特征，相当于是我们拿输入 X_i 作为一个 index, X_i 的任意一个 Field i 中只有 1 个为 1，其余的都是 0。哪个位置的特征值为 1，那么就选中 W 中对应的行，作为嵌入后这个 Field i 对应的新的特征表示。对于每一个 Field 都执行这样的操作，就选出来了 X_i Embedding 之后的表示。注意到，每个 Field 都 **肯定会选出且仅选出 W 中的某一行**(想想为什么？)，因为 W 的列数是固定的，每一个 Field 都选出 W.cols 作为对应的新特征。把每个 Field 选出来的这些 W 的行，拼接起来就得到了 X Embedding 后的新的表示：维度是num(Field) * num(W.cols)。虽然每个 Field 的长度可能不同，但是都是在 W 中选择一行，所以选出来的长度是相同的。这也是 Embedding 的一个特性：虽然输入的 Field 长度不同，但是 Embedding 之后的长度是相同的。

什么？Embedding 这么复杂，怎么实现的？**非常简单！直接把 X 和 W 做内积即可。** 是的，你没看错，就是这么简单（tensoflow 中封装了下，改成了 tf.nn.embedding_lookup(embeddings, index)，原理就是不做乘法，直接选出对应的行）。自己写个例子试试：X.shape=(1, feature_size), W.shape = (feture_size, embedding_size)就知道为什么选出 1 对应 W 的行和做内积结果是一样的是怎么回事了。

所以：**FM 模块图中，黑线部分是一个全连接！W 就是里面的权重。** 把输入 X 和 W 相乘就得到了输出。至于Addition Unit, 我们就不纠结了，这里并没有做什么加法，就把他当成是反应 1 阶特征对输出的影响就行了。

第二、embedding_size对应的是

$$\sum_{j_1=1}^{d}\sum_{j_2=j_1+1}^{d}\langle V_i, V_j\rangle x_{j_1}\cdot x_{j_2}$$

FM 论文中给出了化简后的公式：

$$
\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=i+1}^{n}\langle \mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j \\
=&\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\langle \mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j - \frac{1}{2}\sum_{i=1}^{n}\langle \mathbf{v}_i, \mathbf{v}_i\rangle x_i x_i \\
=&\frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{f=1}^{k} v_{i,f}\, v_{j,f}\, x_i x_j - \sum_{i=1}^{n}\sum_{f=1}^{k} v_{i,f}\, v_{i,f}\, x_i x_i\right) \\
=&\frac{1}{2}\sum_{f=1}^{k}\left(\left(\sum_{i=1}^{n} v_{i,f}\, x_i\right)\left(\sum_{j=1}^{n} v_{j,f}\, x_j\right) - \sum_{i=1}^{n} v_{i,f}^2\, x_i^2\right) \\
=&\frac{1}{2}\sum_{f=1}^{k}\left(\left(\sum_{i=1}^{n} v_{i,f}\, x_i\right)^2 - \sum_{i=1}^{n} v_{i,f}^2\, x_i^2\right)
\end{aligned}
$$

这里最后的结果中是在 [1,K] 上的一个求和。

**K 就是 W 的列数，就是 Embedding 后的维度，也就是 embedding_size**。也就是说，在 DeepFM 的 FM 模块中，最后没有对结果从[1,K]进行求和。而是把这 K 个数拼接起来形成了一个 K 维度的向量。

**FM Component 总结：**

1. FM 模块实现了对于 1 阶和 2 阶组合特征的建模。
2. 没有使用预训练
3. 没有人工特征工程
4. embedding 矩阵的大小是：特征数量 * 嵌入维度。然后用一个 index 表示选择了哪个特征。

**需要训练的有两部分：**

1. input_vector 和 Addition Unit 相连的全连接层，也就是 1 阶的 Embedding 矩阵。
2. Sparse Feature 到 Dense Embedding 的 Embedding 矩阵，中间也是全连接的，要训练的是中间的权重矩阵，这个权重矩阵也就是隐向量 V。

## 3.2 Deep Component

Deep Component 架构图：

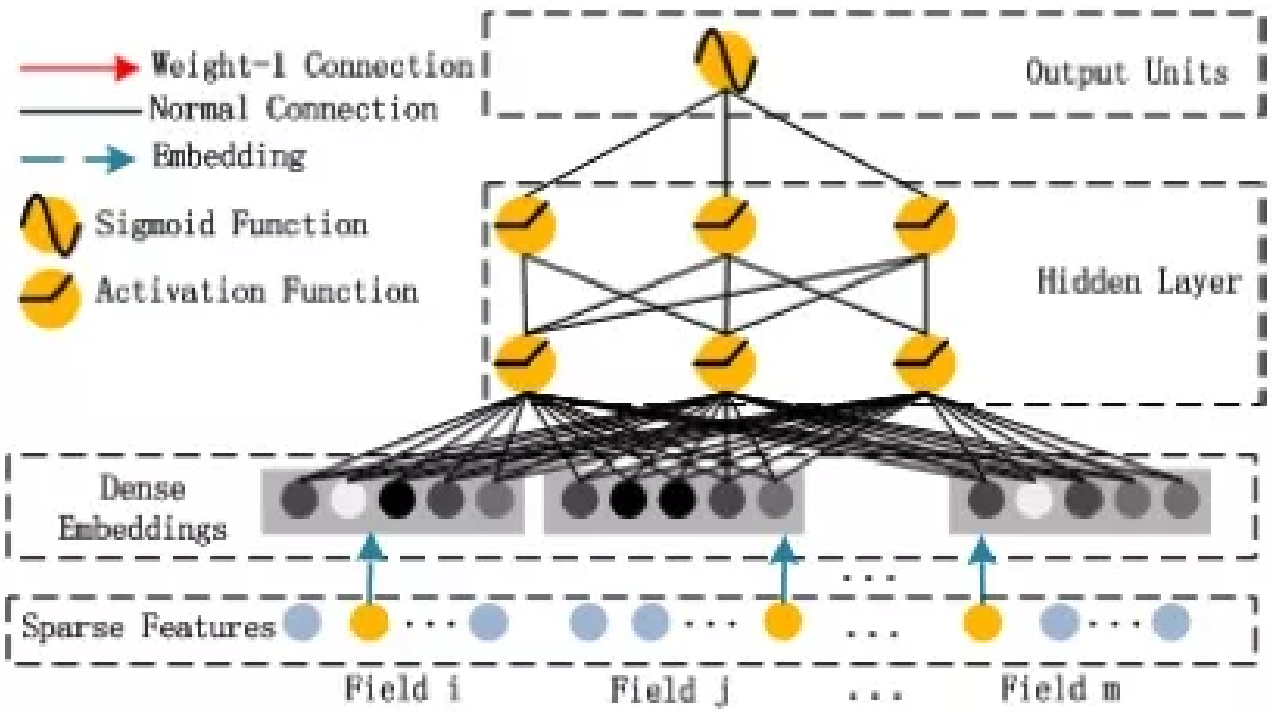



Figure 3: The architecture of DNN.

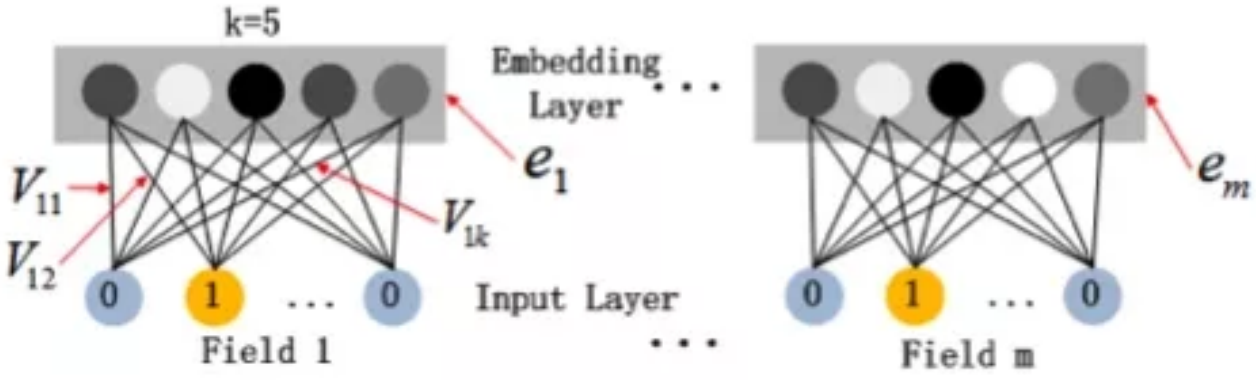
Deep Component 是用来学习 **高阶组合特征** 的。网络里面黑色的线是全连接层，参数需要神经网络去学习。

由于 CTR 或推荐系统的数据 one-hot 之后特别稀疏，如果直接放入到 DNN 中，参数非常多，我们没有这么多的数据去训练这样一个网络。所以 **增加了一个 Embedding 层，用于降低纬度。**

**这里继续补充下 Embedding 层，两个特点：**

1. 尽管输入的长度不同，但是映射后长度都是相同的.`embedding_size` 或 `k`
2. embedding 层的参数其实是全连接的 Weights，是通过神经网络自己学习到的。

Embedding 层的架构图：



值得注意的是：**FM 模块和 Deep 模块是共享 feature embedding 的（也就是 V）。**

好处：

1. 模型可以从最原始的特征中，同时学习低阶和高阶组合特征
2. 不再需要人工特征工程。Wide&Deep 中低阶组合特征就是同过特征工程得到的。

## 3.3 对比其他模型

模型图：

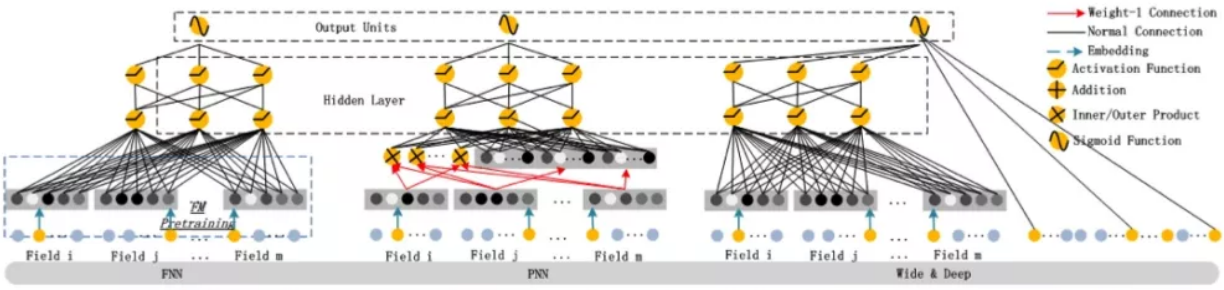



Figure 5: The architectures of existing deep models for CTR prediction: FNN, PNN, Wide & Deep Model

### FNN

FNN is a FM-initialized feedforward neural network.FNN 使用预训练的 FM 来初始化 DNN，然后只有 Deep 部分，不能学习低阶组合特征。

**FNN 缺点：**

1. Embedding 的参数受 FM 的影响，不一定准确
2. 预训练阶段增加了计算复杂度，训练效率低
3. FNN 只能学习到高阶的组合特征。模型中没有对低阶特征建模。

### PNN

PNN：为了捕获高阶特征。PNN 在第一个隐藏层和 embedding 层之间，增加了一个 product layer。

根据 product 的不同，衍生出三种 PNN：IPNN，OPNN，PNN* 分别对应内积、外积、两者混合。

作者为了加快计算，采用近似计算的方法来计算内积和外积。内积：忽略一些神经元。外积：把 m*k 维的 vector 转换成 k 维度的 vector。由于外积丢失了较多信息，所以一般没有内积稳定。

但是内积的计算复杂度依旧非常高，原因是：product layer 的输出是要和第一个隐藏层进行全连接的。

**PNN 缺点：**

1. 内积外积计算复杂度高。采用近似计算的方法外积没有内积稳定。
2. product layer 的输出需要与第一个隐藏层全连接，导致计算复杂度居高不下
3. 和 FNN 一样，只能学习到高阶的特征组合。没有对于 1 阶和 2 阶特征进行建模。

### Wide&Deep

> Wide & Deep 设计的初衷是想同时学习低阶和高阶组合特征，但是 wide 部分需要领域知识进行特征工程。

Wide 部分可以用 LR 来替换，这样的话就和 DeepFM 差不多了。但是 DeepFM 共享 feature embedding 这个特性使得在反向传播的时候，模型学习 feature embedding，而后又会在前向传播的时候影响低阶和高阶特征的学习，这使得学习更加的准确。

**Wide&Deep 缺点：** 需要特征工程提取低阶组合特征

### DeepFM

**优点：**

1. 没有用 FM 去预训练隐向量 V，并用 V 去初始化神经网络。（相比之下 FNN 就需要预训练 FM 来初始化 DNN）
2. FM 模块不是独立的，是跟整个模型一起训练学习得到的。（相比之下 Wide&Deep 中的 Wide 和 Deep 部分是没有共享的）
3. 不需要特征工程。（相比之下 Wide&Deep 中的 Wide 部分需要特征工程）
4. 训练效率高。（相比 PNN 没有那么多参数）

什么？太多了乱七八糟记不住？OK，那就记住 **最核心的**：

1. 没有预训练（no pre-training）
2. 共享 Feature Embedding，没有特征工程（no feature engineering）
3. 同时学习低阶和高阶组合特征（capture both low-high-order interaction features）

### 3.4 超参数建议

论文中还给出了一些参数的实验结果，直接给出结论，大家实现的时候可以参考下。

| 超参数 | 建议 | 备注 |
| --- | --- | --- |
| 激活函数 | 1. IPNN使用tanh 2. 其余使用ReLU | |
| 学习方法 | Adam | |
| Dropout | 0.6~0.9 | |
| 隐藏层数量 | 3~5 | 根据实际数据大小调整 |
| 神经元数量 | 200~600 | 根据实际数据大小调整 |
| 网络形状 | constant | 一共有四种：固定、增长、下降、菱形。 |

PS: constant 效果最好，就是隐藏层每一层的神经元的数量 **相同**。

模型对比图如下：

Table 1: Comparison of deep models for CTR prediction

| | No Pre-training | High-order Features | Low-order Features | No Feature Engineering |
| --- | --- | --- | --- | --- |
| FNN | × | √ | × | √ |
| PNN | √ | √ | × | √ |
| Wide & Deep | √ | √ | √ | × |
| DeepFM | √ | √ | √ | √ |

## 4. DeepFM 实践

### 4.1 变量说明

为了方便后面的阅读，我们先说明一下各个名称的含义：

1. field_size: 输入 X 在进行 one-hot 之前的特征维度
2. feature_size: 输入 X 在 one-hot 之后的特征维度， 又记作 n
3. embedding_size: one-hot 后的输入，进行嵌入后的维度，又记作 k
4. 代码中 tf 中维度 None 表示任意维度，我们用来表示输入样本的数量这一维度。

## 4.2 核心代码拆解

### 4.2.1 输入

```
1.  feat_index = tf.placeholder(dtype=tf.int32, shape=[None, config.
    field_size], name='feat_index') # [None, field_size]
2.  feat_value = tf.placeholder(dtype=tf.float32, shape=[None, None
    ], name='feat_value') # [None, field_size]
3.  label = tf.placeholder(dtype=tf.float16, shape=[None,1], name='l
    abel')
```

注意下各个输入变量的维度大小就可以了，没什么特别需要说明的。None 在 tf 里面表示任意维度，此处表示样本数量维度。

### 2.2 Embedding

```
1.  # Sparse Features -> Dense Embedding
2.  embeddings_origin = tf.nn.embedding_lookup(weights['feature_embe
    dding'], ids=feat_index) # [None, field_size, embedding_size]
```

重点来了，这里是完成Sparse Features到Dense Embedding的转换。

### 2.3 FM Component - 1 维特征

1 维特征本来是求和得到一个标量的，为了提高学习效果，我们改为 **不求和，得到一个 field_size 维度的向量。相当于是进行了一次 k=1 的一次 embedding。**

```
y_first_order = tf.nn.embedding_lookup(weights['feature_bias'],
 ids=feat_index) # [None, field_size, 1]
```

得到 W 之后，和输入 X 相乘，并缩减维度，得到最终 FM-1 维的输出：

```
w_mul_x = tf.multiply(y_first_order, feat_value_reshape) # [None, field_size, 1] Wi * Xi
y_first_order = tf.reduce_sum(input_tensor=w_mul_x, axis=2) # [None, field_size]
```

### 2.4 FM Component - 2 维组合特征

在第一节中，我们给出了 FM-2 维组合特征的化简公式。发现计算的两部分都需要计算 **vi * xi**。所以我们先把这一部分计算出来：

```
feat_value_reshape = tf.reshape(tensor=feat_value, shape=[-1, config.field_size, 1]) # -1 * field_size * 1
embeddings = tf.multiply(embeddings_origin, feat_value_reshape) # [None, field_size, embedding_size]
 multiply不是矩阵相乘，而是矩阵对应位置相乘。这里应用了broadcast机制。
```

然后分别计算这两部分：

$$\left(\sum_{i=1}^{n} v_{i,f}\, x_i\right)^2$$

```
// sum_square part 先sum，再square
summed_features_emb = tf.reduce_sum(input_tensor=embeddings, axis=1) # [None, embedding_size]
summed_features_emb_square = tf.square(summed_features_emb)
```

$$\sum_{i=1}^{n} v_{i,f}^2\, x_i^2$$

```
// square_sum part
squared_features_emb = tf.square(embeddings)
squared_features_emb_summed = tf.reduce_sum(input_tensor=squared_features_emb, axis=1) # [None, embedding_size]
```

最后得到最终 FM-2 维组合特征输出结果，维度为 embedding_size：

```
// second order
y_second_order = 0.5 * tf.subtract(summed_features_emb_square, squared_features_emb_summed)
```

### 2.5 Deep Component

网络部分比较简单，只要一层一层的前向传递就可以了。只有一个问题需要说明：

**网络部分，第一个隐藏层的输入是什么？**

我的感觉应该是原始输入嵌入后的结果：

```
// Deep Component
y_deep = tf.reshape(embeddings_origin, shape=[-1, config.field_size * config.embedding_size]) # [None, field_size * embedding_size]
for i in range(0, len(deep_layers)):
    y_deep = tf.add(tf.matmul(y_deep, weights['layer_%d' % i]), weights['bias_%d' % i])
    y_deep = config.deep_layers_activation(y_deep)
```

### 2.6 输出

把前面 FM Component 和 Deep Component 的两部分结合起来，就得到了最后输出单元的输入，经过 sigmoid 函数激活就可以得到最终结果了。

```
// output
concat_input = tf.concat([y_first_order, y_second_order, y_deep], axis=1)
out = tf.add(tf.matmul(concat_input, weights['concat_projection']), weights['concat_bias'])
out = tf.nn.sigmoid(out)
```

## 3. 完整代码：

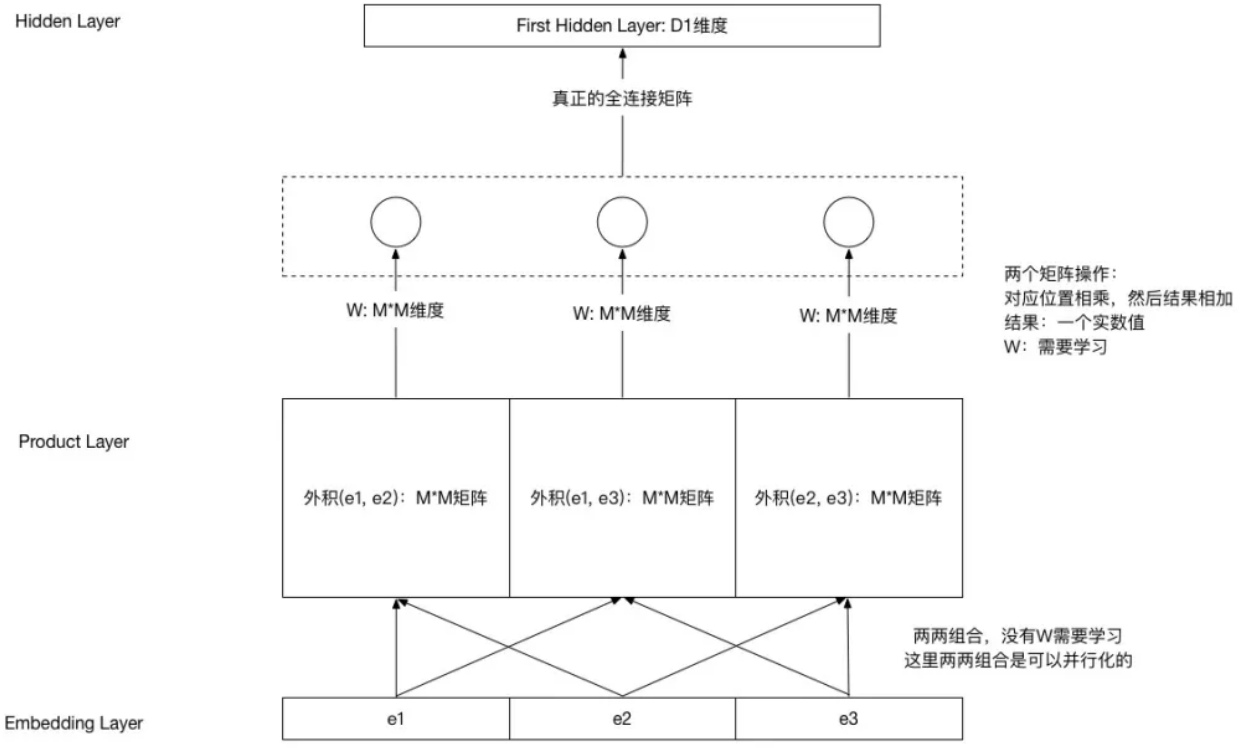
这份代码最主要的目的是学习，直接应用于工程的话还需要做一些优化。比如 batch normalization、stack train、batch train 以及代码重构等。

本代码可以帮助你快速实验，实现 DeepFM，掌握其原理，可直接运行。

Github 地址：https://github.com/gutouyu/ML_CIA

另外，附上一份整理的 DeepFM 架构图 - 实现篇 的图。主要是帮助大家理解 **各个参数的维度、权重 weights 的维度、需要学习的维度、FM Deep 两部分输出的维度。**

要想实现 DeepFM，**只要把每一部分的维度搞清楚，网络的架构搞清楚就问题不大了。**




外积 p 部分架构逻辑说明

参考资料

1. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction
2. https://github.com/ChenglongChen/tensorflow-DeepFM