推荐系统遇上深度学习(七)--NFM模型理论和实践

原创 文文 小小挖掘机 2018-05-05

收录于话题
#推荐系统遇上深度学习

92个

推荐系统遇上深度学习系列：

推荐系统遇上深度学习(一)--FM模型理论和实践

推荐系统遇上深度学习(二)--FFM模型理论和实践

推荐系统遇上深度学习(三)--DeepFM模型理论和实践

推荐系统遇上深度学习(四)--多值离散特征的embedding解决方案

推荐系统遇上深度学习(五)--Deep&Cross Network模型理论和实践

推荐系统遇上深度学习(六)--PNN模型理论和实践

1、引言

在CTR预估中，为了解决稀疏特征的问题，学者们提出了FM模型来建模特征之间的交互关系。但是FM模型只能表达特征之间两两组合之间的关系，无法建模两个特征之间深层次的关系或者说多个特征之间的交互关系，因此学者们通过Deep Network来建模更高阶的特征之间的关系。

因此 FM和深度网络DNN的结合也就成为了CTR预估问题中主流的方法。有关FM和DNN的结合有两种主流的方法，并行结构和串行结构。两种结构的理解以及实现如下表所示：

结构	描述	常见模型
并行结构	FM部分和DNN部分分开计算，只在输出层进行一次融合得到结果	DeepFM，DCN，Wide&Deep
串行结构	将FM的一次项和二次项结果(或其中之一)作为DNN部分的输入，经DNN得到最终结果	PNN,NFM,AFM

今天介绍的NFM模型(Neural Factorization Machine)，便是串行结构中一种较为简单的网络模型。

2、NFM模型介绍

我们首先来回顾一下FM模型，FM模型用n个隐变量来刻画特征之间的交互关系。这里要强调的一点是，n是特征的总数，是one-hot展开之后的，比如有三组特征，两个连续特征，一个离散特征有5个取值，那

么n=7而不是n=3.

$$\hat{y}_{FM(x)} = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \boldsymbol{v}_i^T \boldsymbol{v}_j \cdot x_i x_j$$

小小挖掘机

顺便回顾一下化简过程：

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

小小挖掘机

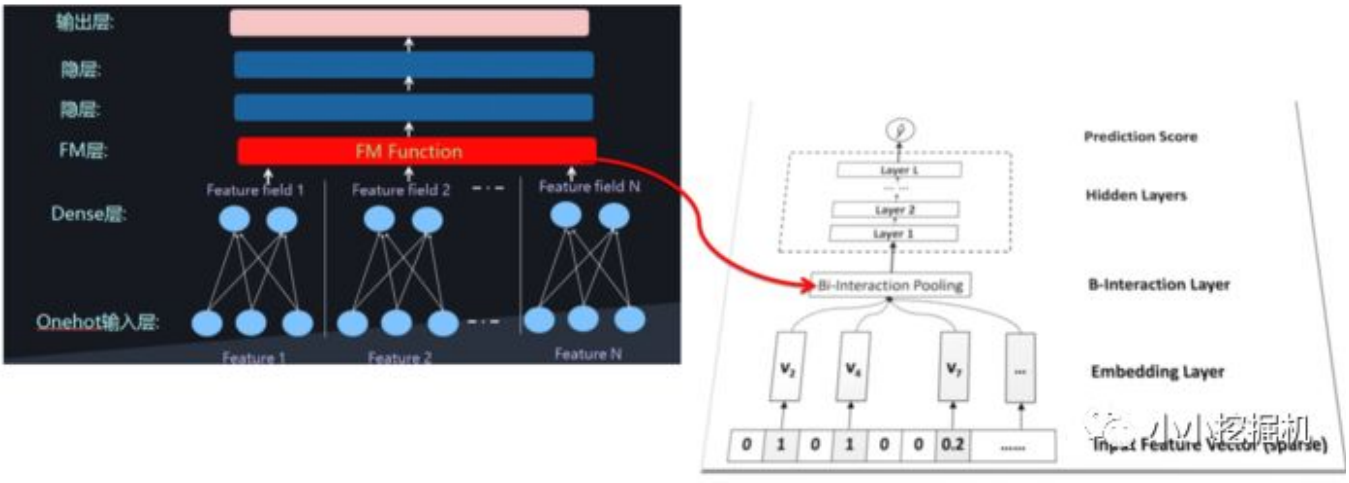
可以看到，不考虑最外层的求和，我们可以得到一个K维的向量。

对于NFM模型，目标值的预测公式变为：

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

小小挖掘机

其中，f(x)是用来建模特征之间交互关系的多层前馈神经网络模块，架构图如下所示：



Embedding Layer和我们之间几个网络是一样的，embedding 得到的vector其实就是我们在FM中要学习的隐变量v。

Bi-Interaction Layer名字挺高大上的，其实它就是计算FM中的二次项的过程，因此得到的向量维度就是我们的Embedding的维度。最终的结果是：

$$f_{BI}(\mathcal{V}_x) = \frac{1}{2}[\left(\sum_{i=1}^{n} x_i \boldsymbol{v}_i\right)^2 - \sum_{i=1}^{n} (x_i \boldsymbol{v}_i)^2]$$

Hidden Layers就是我们的DNN部分，将Bi-Interaction Layer得到的结果接入多层的神经网络进行训练，从而捕捉到特征之间复杂的非线性关系。

在进行多层训练之后，将最后一层的输出求和同时加上一次项和偏置项，就得到了我们的预测输出：

$$\hat{y}_{NFM}(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \mathbf{h}^T \sigma_L(\mathbf{W}_L(...\sigma_1(\mathbf{W}_1 f_{BI}(\mathcal{V}_x) + \mathbf{b}_1)...) + \mathbf{b}_L)$$

是不是很简单呢，哈哈。

3、代码实战

终于到了激动人心的代码实战环节了，本文的代码有不对的的地方或者改进之处还望大家多多指正。

本文的github地址为：
https://github.com/princewen/tensorflow_practice/tree/master/recommendation/Basic-NFM-Demo

本文的代码根据之前DeepFM的代码进行改进，我们只介绍模型的实现部分，其他数据处理的细节大家可以参考我的github上的代码.

模型输入

模型的输入主要有下面几个部分:

```
self.feat_index = tf.placeholder(tf.int32,
                                  shape=[None,None],
                                  name='feat_index')
self.feat_value = tf.placeholder(tf.float32,
                                 shape=[None,None],
                                 name='feat_value')

self.label = tf.placeholder(tf.float32,shape=[None,1],name='label')
```

```
self.dropout_keep_deep  =  tf.placeholder(tf.float32,shape=[None],name='dropout_deep_deep')
```

feat_index是特征的一个序号，主要用于通过embedding_lookup选择我们的embedding。feat_value是对应的特征值，如果是离散特征的话，就是1，如果不是离散特征的话，就保留原来的特征值。label是实际值。还定义了dropout来防止过拟合。

权重构建

权重主要分以下几部分，偏置项，一次项权重，embeddings，以及DNN的权重

```
def _initialize_weights(self):
    weights = dict()

    #embeddings
    weights['feature_embeddings'] = tf.Variable(
        tf.random_normal([self.feature_size,self.embedding_size],0.0,0.01),
        name='feature_embeddings')
     weights['feature_bias'] = tf.Variable(tf.random_normal([self.feature_size,1],0.0,1.0),name='feature_bias')
    weights['bias'] = tf.Variable(tf.constant(0.1),name='bias')

    #deep layers
    num_layer = len(self.deep_layers)
    input_size = self.embedding_size
    glorot = np.sqrt(2.0/(input_size + self.deep_layers[0]))

    weights['layer_0'] = tf.Variable(
         np.random.normal(loc=0,scale=glorot,size=(input_size,self.deep_layers[0])),dtype=np.float32
    )
    weights['bias_0'] = tf.Variable(
         np.random.normal(loc=0,scale=glorot,size=(1,self.deep_layers[0])),dtype=np.float32
    )


    for i in range(1,num_layer):
        glorot = np.sqrt(2.0 / (self.deep_layers[i - 1] + self.deep_layers[i]))
        weights["layer_%d" % i] = tf.Variable(
             np.random.normal(loc=0, scale=glorot, size=(self.deep_layers[i - 1], self.deep_layers[i])),
            dtype=np.float32)  # layers[i-1] * layers[i]
        weights["bias_%d" % i] = tf.Variable(
            np.random.normal(loc=0, scale=glorot, size=(1, self.deep_layers[i])),
            dtype=np.float32)  # 1 * layer[i]

    return weights
```

Embedding Layer

这个部分很简单啦，是根据feat_index选择对应的weights['feature_embeddings']中的embedding值，然后再与对应的feat_value相乘就可以了：

```
# Embeddings
self.embeddings = tf.nn.embedding_lookup(self.weights['feature_embeddings'],self.fea
t_index) # N * F * K
feat_value = tf.reshape(self.feat_value,shape=[-1,self.field_size,1])
self.embeddings = tf.multiply(self.embeddings,feat_value) # N * F * K
```

Bi-Interaction Layer

我们直接根据化简后的结果进行计算，得到一个K维的向量：

```
# sum-square-part
self.summed_features_emb = tf.reduce_sum(self.embeddings, 1)  # None * k
self.summed_features_emb_square = tf.square(self.summed_features_emb)  # None * K

# squre-sum-part
self.squared_features_emb = tf.square(self.embeddings)
self.squared_sum_features_emb = tf.reduce_sum(self.squared_features_emb, 1)  # None
 * K

# second order
self.y_second_order = 0.5 * tf.subtract(self.summed_features_emb_square, self.square
d_sum_features_emb)
```

Deep Part

将Bi-Interaction Layer层得到的结果经过一个多层的神经网络，得到交互项的输出：

```
self.y_deep = self.y_second_order
for i in range(0, len(self.deep_layers)):
    self.y_deep = tf.add(tf.matmul(self.y_deep, self.weights["layer_%d" % i]), self.
weights["bias_%d" % i])
    self.y_deep = self.deep_layers_activation(self.y_deep)
    self.y_deep = tf.nn.dropout(self.y_deep, self.dropout_keep_deep[i + 1])
```

得到预测输出

为了得到预测输出，我们还需要两部分，分别是偏置项和一次项：

```
# first order term
self.y_first_order = tf.nn.embedding_lookup(self.weights['feature_bias'], self.feat_
index)
self.y_first_order = tf.reduce_sum(tf.multiply(self.y_first_order, feat_value), 2)

# bias
self.y_bias = self.weights['bias'] * tf.ones_like(self.label)
```

而我们的最终输出如下：

```
# out
self.out = tf.add_n([tf.reduce_sum(self.y_first_order,axis=1,keep_dims=True),
```

```
                tf.reduce_sum(self.y_deep,axis=1,keep_dims=True),
                self.y_bias])
```

剩下的代码就不介绍啦！
好啦，本文只是提供一个引子，有关NFM的知识大家可以更多的进行学习呦。

4、小结

NFM模型将FM与神经网络结合以提升FM捕捉特征间多阶交互信息的能力。根据论文中实验结果，NFM的预测准确度相较FM有明显提升，并且与现有的并行神经网络模型相比，复杂度更低。

NFM本质上还是基于FM，FM会让一个特征固定一个特定的向量，当这个特征与其他特征做交叉时，都是用同样的向量去做计算。这个是很不合理的，因为不同的特征之间的交叉，重要程度是不一样的。因此，学者们提出了AFM模型（Attentional factorization machines），将attention机制加入到我们的模型中，关于AFM的知识，我们下一篇来一探究竟。

推荐阅读：强化学习系列

实战深度强化学习DQN-理论和实践

DQN三大改进(一)-Double DQN

DQN三大改进(二)-Prioritised replay

DQN三大改进(三)-Dueling Network

深度强化学习-Policy Gradient基本实现

深度强化学习-Actor-Critic算法原理和实现

深度强化学习-DDPG算法原理和实现

对抗思想与强化学习的碰撞-SeqGAN模型原理和代码解析

有关作者：

石晓文，中国人民大学信息学院在读研究生，美团外卖算法实习生

简书ID：石晓文的学习日记(https://www.jianshu.com/u/c5df9e229a67)

天善社区：https://www.hellobi.com/u/58654/articles

腾讯云：https://cloud.tencent.com/developer/user/1622140