基于深度学习的推荐(三)：基于attention的AFM模型

原创 如雨星空 推荐算法工程师 2019-09-15

公众号

输入"进群",加入交流群,和小伙伴们一起讨论机器学习,深度学习,推荐算法.

前言

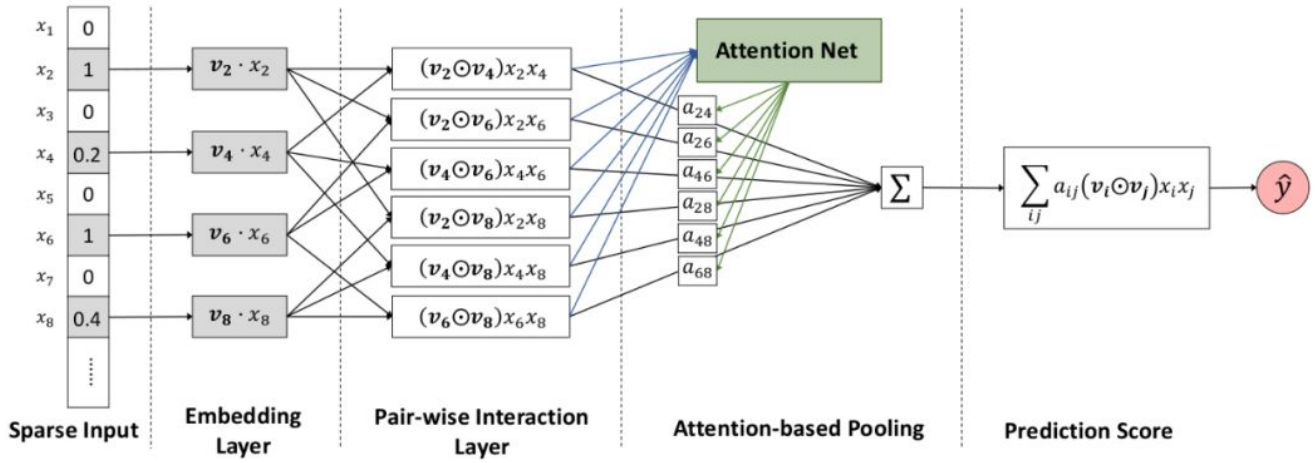
从这周开始,我们关注基于attention机制的推荐模型,首先看下较早提出的AFM(Attentional Factorization Machines)模型.

论文作者认为,并非所有的特征交互都包含有效信息,因此那些"less userful features"应该赋予更低的权重.很明显,当无效信息和有效信息权重相同时,相当于引入了噪声.而FM缺乏区分不同特征重要性的能力,可能会获得次优解.

怎么赋予权重呢?神经网络中最常见的就是映射,加个权重矩阵(wx+b),但这种线性的权重,有时不能满足需要.论文中使用了一层映射+sum pooling+sotfmax构建出了非线性权重系数,作者称之为"Attention-based Pooling".

论文链接:https://www.ijcai.org/proceedings/2017/0435.pdf

模型介绍



从整体上看,AFM就是FM+Attention.,前面一部分embeding和pair-wise和FM模型的是类似的,然后后面加了个attention机制,就是AFM模型.

首先看前面一部分.FM中特征的交互是'inner product',两个向量做内积,结果是一个值.而AFM中特征交互是'element-wise product',两个向量对应元素相乘,结果是向量:

$$f_{PI}(\mathcal{E}) = \{(\mathbf{v}_i \odot \mathbf{v}_j)x_i x_j\}_{(i,j)\in\mathcal{R}_x},$$

然后给向量一个映射矩阵,对向量元素求和,得到交叉特征部分的预测结果,就是论文中的'sum pooling':

$$\hat{y} = \mathbf{p}^T \sum_{(i,j)\in\mathcal{R}_x} (\mathbf{v}_i \odot \mathbf{v}_j)x_i x_j + b,$$

这就是AFM和FM在特征交互不同的地方,p可以看成是sum pooling,也可以认为是embedding特征的权重矩阵,加上后文交互特征的attention权重,可以说比FM多了两层权重,一层区分交互特征重要性,一层区分embedding各维特征的重要性.从而大大提高了模型的拟合能力.attention系数可以这样加上去:

$$f_{Att}(f_{PI}(\mathcal{E})) = \sum_{(i,j)\in\mathcal{R}_x} a_{ij}(\mathbf{v}_i \odot \mathbf{v}_j)x_i x_j,$$

aij可以直接通过最小化损失函数求得,但是当xi,xj没有共同出现时,是求不出对应的aij的.为了获得更加通用的模型,作者将attention score--aij进行参数化(有没有似曾相识,想想FM),并使用如下公式来定义:

$$a'_{ij} = \mathbf{h}^T ReLU(\mathbf{W}(\mathbf{v}_i \odot \mathbf{v}_j)x_i x_j + \mathbf{b}),$$

$$a_{ij} = \frac{\exp(a'_{ij})}{\sum_{(i,j)\in\mathcal{R}_x} \exp(a'_{ij})},$$

所以,最终模型的预测值可以表示为:

$$\hat{y}_{AFM}(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \mathbf{p}^T \sum_{i=1}^{n} \sum_{j=i+1}^{n} a_{ij}(\mathbf{v}_i \odot \mathbf{v}_j) x_i x_j,$$

代码实战

代码通俗易懂,数据也很小,我们着重看一下attention部分怎么实现的,代码格式容易乱,直接截个图吧:

```
with tf.name_scope('attention_net'):
    glorot = np.sqrt(2.0 / (self.attention_size + self.embedding_size))
    weights['attention_w'] = tf.Variable(
        np.random.normal(loc=0, scale=glorot, size=(self.embedding_size, self.attention_size)),
        dtype=tf.float32, name='attention_w')
    biases['attention_b'] = tf.Variable(np.random.normal(loc=0, scale=glorot, size=(1, self.attention_size)),
                                        dtype=tf.float32, name='attention_b')
    weights['attention_h'] = tf.Variable(np.random.normal(loc=0, scale=1, size=(1, self.attention_size)),
                                         dtype=tf.float32, name='attention_h')
    weights['attention_p'] = tf.Variable(np.random.normal(loc=0, scale=1, size=(self.embedding_size, 1)),
                                         dtype=tf.float32, name='attention_p') # 若p全为1, 则直接表示FM二阶项

    num_interactions = self.pair_wise_product.shape.as_list()[1]
    # w*x + b
    self.attention_wx_plus_b = tf.add(tf.matmul(tf.reshape(self.pair_wise_product, shape=[-1, self.embedding_size]), weights['attention_w']), biases['attention_b'])
    self.attention_wx_plus_b = tf.reshape(self.attention_wx_plus_b,shape=[-1, num_interactions, self.attention_size]) # [None, field_size*(field_size - 1)/2, attention_size]

    # relu(w*x + b)
    self.attention_relu_wx_plus_b = tf.nn.relu(self.attention_wx_plus_b) # [None, field_size*(field_size - 1)/2, attention_size]

    # h*relu(w*x + b)
    self.attention_h_mul_relu_wx_plus_b = tf.multiply(self.attention_relu_wx_plus_b, weights['attention_h']) # [None, field_size*(field_size - 1)/2, attention_size]

    # exp(h*relu(w*x + b))
    self.attention_exp = tf.exp(tf.reduce_sum(self.attention_h_mul_relu_wx_plus_b, axis=2, keep_dims=True)) # [None, field_size*(field_size - 1)/2, 1]

    # sum(exp(h*relu(w*x + b)))
    self.attention_exp_sum = tf.reduce_sum(self.attention_exp, axis=1, keep_dims=True) # [None, 1, 1]

    # exp(h*relu(w*x + b)) / sum(exp(h*relu(w*x + b)))
    self.attention_out = tf.div(self.attention_exp, self.attention_exp_sum, name='attention_out')  # [None, field_size*(field_size - 1)/2, 1]

    # attention*Pair-wise
    self.attention_product = tf.multiply(self.attention_out, self.pair_wise_product) # [None, field_size*(field_size - 1)/2, embedding_size]

    self.attention_product = tf.reduce_sum(self.attention_product, axis=1) # [None, embedding_size]

    # p*attention*Pair-wise
    self.attention_net_out = tf.matmul(self.attention_product, weights['attention_p']) # [None, 1]
```


实际就是根据公式一步步计算的,清晰易懂,变量名称和论文都是保持一致的.最后的输出结果就是线性拟合和特征交叉这部分之和:

```
with tf.name_scope('out'):
    self.out = tf.add_n([self.w0, self.linear_out, self.attention_net_out]) # # yAFM = w0 + wx + attention(x)
```

这份代码简单易懂,从前到后逻辑很清晰,可以拿来练习tensorflow.

原代码:源代码出处我修改了一些细节,完整代码和数据