学习FM算法之前，作为小白，在面对一堆带有FM名字的算法时总是一脸茫然，FFM（Field-aware Factorization Machines）就是其中让我头疼的一个。但是静下心来，认真学习FFM之后，反而觉得并不是想像中的那么晦涩难懂。这篇文章就记录一下我对FFM算法的理解，但是个人水平十分有限，有不对的地方，还请大家指出，不胜感激！

1 FFM简介

1.1 提出的动机

通过上一篇文章（[因子分解机（FM）简介及实践]），我们了解到FM通过两个向量对原始的二阶特征组合权重矩阵$w_{i,j}$进行分解，进而缓解了稀疏数据对权重更新的影响，如公式1所示：

$$\hat{y}(x) := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} < v_i, v_j > x_i x_j. \tag{1}$$

假设现有一组人工构造的CTR数据，其大致格式如下图所示，其中“+”代表该广告在展示过程中被点击的次数，“-”代表没有被点击的次数，“Publisher”列代表的是发放广告的平台，而“Advertiser”列代表的是不同的广告主。

		Publisher	Advertiser
+80	−20	ESPN	Nike
+10	−90	ESPN	Gucci
+0	−1	ESPN	Adidas
+15	−85	Vogue	Nike
+90	−10	Vogue	Gucci
+10	−90	Vogue	Adidas
+85	−15	NBC	Nike
+0	−0	NBC	Gucci
+90	−10	NBC	Adidas

那么对于一条数据：

Clicked	Publisher (P)	Advertiser (A)	Gender (G)
Yes	ESPN	Nike	Male

FM算法在进行预测时，它的二阶项可以表示为：

$$w_{ESPN} \cdot w_{Nike} + w_{ESPN} \cdot w_{Male} + w_{Nike} \cdot w_{Male} \tag{2}$$

从中可以看出，每项特征都通过一个隐向量来与其他特征的隐向量进行组合，进而实现特征与特征之间的组合关系。以ESPN为例，在组合另外两个特征时均以相同的一个权重w_{ESPN}与其他两个特征的权重w_{Nike}和w_{Male}进行组合。这里需要说明一点的是，因为"Publisher"特征中包含三个取值，即“ESPN”、“Vogue”和“NBC”，在实际做的时候会通过one-hot编码将类别变量“Publisher”编码出一个3维的特征向量，特征向量中的每一维对应一个特征的具体取值，即使用三个类别中的哪一个。因此，在公式2中，出现的是w_{ESPN}而不是$w_{Publisher}$。

FM算法实际上是并没有考虑到每个特征所属类别这一信息的，它会使用相同的一个隐向量w_{ESPN}来计算$w_{ESPN} \cdot w_{Nike}$和$w_{ESPN} \cdot w_{Male}$，而特征"Nike"和“Male”原本是属于两类特征的，即“Advertiser”类和“Gender”类。

那么倘若我们使用两种不同的$w_{ESPN,A}$和$w_{ESPN,G}$分别用来与w_{Nike}和w_{Male}进行计算，那么结果显然是不会比原先使用统一的w_{ESPN}要差的。因为使用统一的w_{ESPN}只是使用不同的w_{ESPN}的一项特例，比如我们可以令$w_{ESPN,A} = w_{ESPN,G} = w_{ESPN}$。

1.2 具体内容

FFM便是在FM的基础之上，加上了每个特征所属领域对模型的影响，其表达式可以写为公式3的形式：

$$\hat{y}(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{j_1=1}^{n} \sum_{j_2=j_1+1}^{n} (w_{j_1,f_2} \cdot w_{j_2,f_1}) x_{j_1} x_{j_2}, \tag{3}$$

其中f_1和f_2分别代表的是特征j_1和j_2所属的领域。也就是说原先只有一个w_{j_1}来应对其他所有的特征，而现在是有多个$w_{j_1,f}$来应对不同领域的特征，假设领域一共有f个，那么原先只有1个参数，现在变为了$f-1$个。根据上一篇介绍FM算法的文章，我们知道FM算法的时间复杂度是$O(kn)$，k代表隐向量维度，n代表特征个数，那么由此可知FFM算法的时间复杂度便是$O(knf)$。当领域种类$f=1$时，即所有特征都属于同一个类别，那么FFM算法会退化为FM算法，而当领域数目$f=n$时，就是每个特征都属于不同的类别，那么FFM算法的时间复杂度会增加到$O(kn^2)$。

那么在更新FFM模型时，对于常数项有：

$$\frac{\partial \hat{y}(x)}{\partial w_0} = 1. \tag{4}$$

对于一阶项有：

$$\frac{\partial \hat{y}(x)}{\partial w_i} = x_i. \tag{5}$$

而对于二阶项的更新，我们先通过一个例子来说明，对于下面的一条数据，一共有3个特征类别，8个特征以及1个label：

	Publisher(1)			Advertiser(2)			Gender(3)	
label	1	2	3	4	5	6	7	8
Clicked	ESPN	Vogue	NBC	Nike	Gucci	Adidas	Male	Female
Yes	1	0	0	1	0	0	1	0

那么，二阶项可以写为：

$$\begin{aligned} w_{1,2} \cdot w_{4,1} x_1 x_4 + w_{1,2} \cdot w_{5,1} x_1 x_5 + w_{1,2} \cdot w_{6,1} x_1 x_6 + w_{1,3} \cdot w_{7,1} x_1 x_7 + w_{1,3} \cdot w_{8,1} x_1 x_8 \\ + w_{4,3} \cdot w_{7,2} x_4 x_7 + w_{4,3} \cdot w_{8,2} x_4 x_8 \end{aligned} \tag{3}$$

其中$w_{1,2}$中的1代表的是第1个特征ESPN，2代表的是第2个类别Advertiser，以此类推其他项。那么对于$w_{1,2}$的更新，我们可以写为：

$$\frac{\partial \hat{y}(x)}{\partial w_{1,2}} = w_{4,1} x_1 x_4 + w_{5,1} x_1 x_5 + w_{6,1} x_1 x_6. \tag{6}$$

因此可以得到二阶项的梯度更新公式：

$$\frac{\partial \hat{y}(x)}{\partial w_{j_1,f_2}} = \sum_{j_2 \in f_2} w_{j_2,f_1} x_{j_1} x_{j_2}. \tag{7}$$

当然公式6和公式7都可以再进一步简化，因为在同一个类别中，只有一个特征值为1，其余都是0。比如公式6可以写为：

$$\frac{\partial \hat{y}(x)}{\partial w_{1,2}} = w_{4,1} x_1 x_4. \tag{8}$$

那么公式7就会变为：

$$\frac{\partial \hat{y}(x)}{\partial w_{j_1,f_2}} = w_{j_2,f_1} x_{j_1} x_{j_2}, j_2 \in f_2 \text{ and } x_{j_2} \neq 0. \tag{9}$$

2 FFM算法tensorflow实践

在这一部分中，我使用tensorflow来实现一下FFM算法，但是这里的实现仅仅是为了理解算法流程而做的一个小例子，相比于FFM作者提供的开源包肯定是有很多不完备的地方的。

2.1 数据集介绍及预处理

因为FFM算法是专门考虑了特征所属领域对模型预测的影响，因此如果仍旧使用以往没有特征类别的数据集，我感觉是无法体会到FFM算法的作用的。所以在这里采用Avazu提供的Kaggle比赛数据集（下载地址：[Click-Through Rate Prediction](#)），由于原始训练数据集规模较大（解压后5.87GB），所以在这里我们只采用前500条数据（大小为77KB）作为示范，这个小数据集可以在我的github上找到，当然最后模型训练的结果是无法与FFM作者在这一比赛中的优异表现所能对比的。

在原始数据集上，除去类标click（0代表用户没点击，1代表用户点击了）之外，一共有23个特征，包括用户id、网站类别、用户设备等等。在这里我们随机选取了11个特征进行后续处理，因为如果对23个特征均进行one-hot编码的话，特征维度会比较大，代码在我这台破电脑上运行速度会非常慢（当然也可能是我代码写的不够高效）。

经过one-hot编码，11个类别总共产生66个特征，每个类别包含的特征数目分别对应如下：

特征类别	特征数量
C1	3
banner_pos	2
site_category	8
app_domain	7
app_category	6
device_id	24
device_type	3
device_conn_type	3
C15	3
C16	3
C18	4

2.2 FFM算法核心代码

这里实现的FFM算法和原始FFM论文有略微的差别，即原始FFM算法论文的损失是对数指数损失，而在这里我采用的是交叉熵损失。当然，我这里的实现也加上了正则化项。

略有不足的地方是在于计算FFM特征组合时，使用了两层的for循环，这样下来，FFM程序运行是比较慢的。因为原始论文并没有化简公式3，不像FM运行那么快捷。

需要提一点的是，训练数据x里面是带有每列的列名的，函数 **self._get_field**是用来计算当前的列属于哪一类特征。

```
class FFM:
    def __init__(self, x, y, field_name, lbd = 0.01, learning_rate = 2e-3,
batch_size = 16, epoch = 50):
        #篇幅原因，这里省略，详见Github

    def fit(self):
        x_input = tf.placeholder(dtype = tf.float32, shape = [None,
self.fea_num], name = "x_input")
        y_input = tf.placeholder(dtype = tf.float32, shape = [None, 1], name =
"y_input")
#         print(x_input[:,1])
        biases = tf.Variable(tf.random_normal(shape = [1], mean = 0, stddev =
1), name = 'biases')
        linear_w = tf.Variable(tf.random_normal(shape = [self.fea_num, 1], mean
= 0, stddev = 1), name = "linear_weights")
        complex_w = tf.Variable(tf.random_normal(shape = [self.fea_num,
self.fie_num, self.k], mean = 0, stddev = 1), name = "complex_weights")
        part1 = biases
        part2 = tf.matmul(x_input, linear_w)
        # first method to calculate the part3
        part3 = tf.Variable(tf.random_normal(shape = [1], mean = 0, stddev = 1),
name = 'biases')
        for i in range(self.fea_num - 1):
            f1 = self._get_field(i)
            for j in range(i+1, self.fea_num):
                f2 = self._get_field(j)
                part3 += tf.reduce_sum(tf.multiply(complex_w[i][f2],
complex_w[j][f1])) * tf.multiply(x_input[:,i],x_input[:,j])
        part3 = tf.reshape(part3,[-1, 1])
        y_predicted = tf.add(tf.add(part1, part2), part3)
        y_predicted2 = tf.nn.sigmoid(y_predicted)
        loss1 = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(labels =
y_input, logits = y_predicted))
        loss2 = tf.nn.l2_loss(complex_w)
        loss = loss1 + self.lbd * loss2
        optimizer = tf.train.AdamOptimizer(learning_rate = self.learning_rate)
        train_op = optimizer.minimize(loss)

        saver = tf.train.Saver(max_to_keep = 1)
        tf.add_to_collection('y_p', y_predicted2)
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            for i in range(self.epoch):
                for batch_x, batch_y in self._batch():
                    sess.run(train_op, feed_dict = {x_input:batch_x,
y_input:batch_y})
```

```
                if i % 50 == 0:
                    train_loss = sess.run(loss, feed_dict = {x_input: self.x,
y_input: self.y})
                    print("epoch" + str(i) + "_loss: ", int(train_loss))
                    saver.save(sess, "models/m")
            sess.close()

    def predict(self, x_test):
        #篇幅原因，这里省略，详见Github
    def score(self, x_test, y_test):
        #篇幅原因，这里省略，详见Github
    def _batch(self):
        #篇幅原因，这里省略，详见Github

    def _get_field(self, i):
        cur = self.x.columns[i]
        for index, element in enumerate(self.field_name):
            if len(cur) > len(element) and cur[:len(element)] == element and
cur[len(element)] == "_":
                return index
```

训练过程中产生的loss变化如下

```
epoch0_loss:  494
epoch50_loss:  54
epoch100_loss:  50
epoch150_loss:  50
epoch200_loss:  49
epoch250_loss:  49
epoch300_loss:  49
```

最后模型在测试集上的准确率为0.65，对于二分类问题来说效果不是很好，这可能因为我们训练的样本数特别少，而且特征是随机选择的。当然我写代码主要是理解一下FFM模型的训练过程，也就不继续深究模型性能如何提升了。^_^