推荐系统系列（六）：Wide&Deep理论与实践

原创 默存 SOTA Lab 2019-11-17

背景

在CTR预估任务中，线性模型仍占有半壁江山。利用手工构造的交叉组合特征来使线性模型具有“记忆性”，使模型记住共现频率较高的特征组合，往往也能达到一个不错的baseline，且可解释性强。但这种方式有着较为明显的缺点：首先，特征工程需要耗费太多精力。其次，因为模型是强行记住这些组合特征的，所以对于未曾出现过的特征组合，权重系数为0，无法进行泛化。

为了加强模型的泛化能力，研究者引入了DNN结构，将高维稀疏特征编码为低维稠密的Embedding vector，这种基于Embedding的方式能够有效提高模型的泛化能力。但是，现实世界是没有银弹的。基于Embedding的方式可能因为数据长尾分布，导致长尾的一些特征值无法被充分学习，其对应的Embedding vector是不准确的，这便会造成模型泛化过度。

2016年，Google提出Wide&Deep模型，将线性模型与DNN很好的结合起来，在提高模型泛化能力的同时，兼顾模型的记忆性。Wide&Deep这种线性模型与DNN的并行连接模式，后来成为推荐领域的经典模式。今天与大家一起分享这篇paper，向经典学习。

分析

1. Motivation

在这篇论文中，主要围绕模型的两部分能力进行探讨：Memorization与Generalization。原文定义如下 [1]：

> Memorization can be loosely defined as learning the frequent co-occurrence of items or features and exploiting the correlation available in the historical data. Generalization, on the other hand, is based on transitivity of correlation and explores new feature combinations that have never or rarely occurred in the past.

模型能够从历史数据中学习到高频共现的特征组合的能力，这是模型的Memorization。而Generalization代表模型能够利用相关性的传递性去探索历史数据中从未出现过的特征组合。

广义线性模型能够很好地解决Memorization的问题，但是在Generalization方面表现不足。基于Embedding的DNN模型在Generalization表现优异，但在数据分布较为长尾的情况下，对于长尾数据的处理能力较弱，容易造成过度泛化。

能否将二者进行结合，取彼之长补己之短？使得模型同时兼顾Memorization与Generalization。为此，作者提出二者兼备的Wide&Deep模型，并在Google Play store的场景中成功落地。

2. 模型结构

模型结构示意图如下：

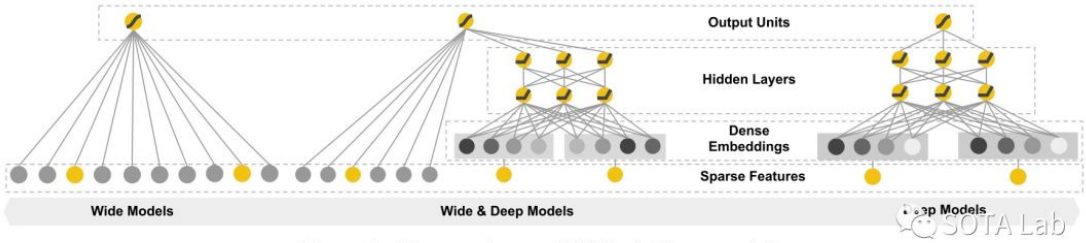

Figure 1: The spectrum of Wide & Deep models.

示意图中最左边便是模型的Wide部分，这个部分可以使用广义线性模型来替代，如LR便是最简单的一种。由此可见，Wide&Deep是一类模型的统称，将LR换成FM同样也是一个Wide&Deep模型（与DeepFM的差异见后续博文）。模型的Deep部分是一个简单的基于Embedding的全连接网络，结构与FNN一致 [2]。

2.1 Wide part

这部分是一个广义线性模型，即 $y = W^T[X, \phi(X)] + b$ 。其中，$X = [x_1, x_2, \cdots, x_d]$ 是 d 维特征向量。$\phi(X) = [\phi_1(X), \phi_2(X), \cdots, \phi_k(X)]$ 是 k 维特征转化函数向量。

最常用的特征转换函数便是特征交叉函数，定义为 $\phi_k(X) = \prod_{i=1}^{d} x_i^{c_{ki}}, c_{ki} \in \{0, 1\}$ ，当且仅当 x_i 是第 k 个特征变换的一部分时，$c_{ki} = 1$ 。否则为0。

举例来说，对于二值特征，一个特征交叉函数为$And(gender = female, language = en)$ ，这个函数中只涉及到特征 $female$ 与 en ，所以其他特征值对应的 $c_{ki} = 0$ ，即可忽略。当样本中 $female$ 与 en 同时存在时，该特征交叉函数为1，否则为0。这种特征组合可以为模型引入非线性。

2.2 Deep part

Deep侧是简单的全连接网络：$a^{(l+1)} = f(W^{(l)}a^{(l)} + b^{(l)})$ ，其中 $a^{(l)}, b^{(l)}, W^{(l)}, f$ 分别代表第 l 层的输入、偏置项、参数项与激活函数。

2.3 Output part

Wide与Deep侧都准备完毕之后，对两部分输出进行简单 **加权求和** 即可作为最终输出。对于简单二分类任务而言可以定义为：

$$P(Y=1|X)=\sigma(W_{wide}^T[X,\phi(X)]+W_{deep}^T a^{(l_f)}+b)$$

其中，$W_{wide}^T[X,\phi(X)]$ 为Wide输出结果，W_{deep} 为Deep侧作用到最后一层激活函数输出的参数，Deep侧最后一层激活函数输出结果为 $a^{(l_f)}$ ，b 为全局偏置项，σ 为 $sigmoid$ 激活函数 。

将Wide与Deep侧进行联合训练，需要注意的是，因为Wide侧的数据是高维稀疏的，所以作者使用了 $FTRL$ 算法优化，而Deep侧使用的是 $AdaGrad$ 。

3. 工程实现

Google使用的pipeline如下，共分为三个部分：Data Generation、Model Training与Model Serving。

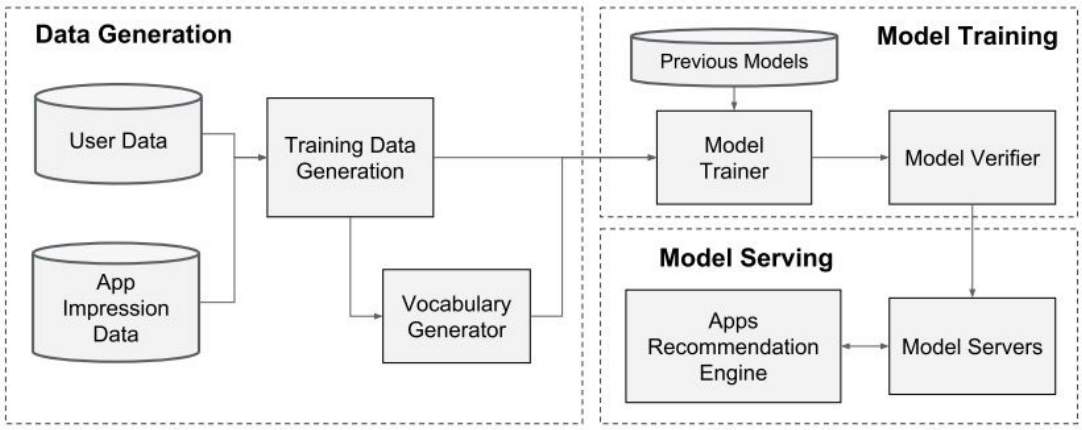

Figure 3: Apps recommendation pipeline overview.

3.1 Data Generation

本阶段负责对数据进行预处理，供给到后续模型训练阶段。其中包括用户数据收集、样本构造。对于类别特征，首先过滤掉低频特征，然后构造映射表，将类别字段映射为编号，即token化。对于连续特征可以根据其分布进行离散化，论文中采用的方式为等分位数分桶方式，然后再放缩至[0,1]区间。

3.2 Model Training

针对Google paly场景，作者构造了如下结构的Wide&Deep模型。在Deep侧，连续特征处理完之后直接送入全连接层，对于类别特征首先输入到Embedding层，然后再连接到全连接层，与连续特征向量拼接。在Wide侧，作者仅使用了用户历史安装记录与当前候选app作为输入。

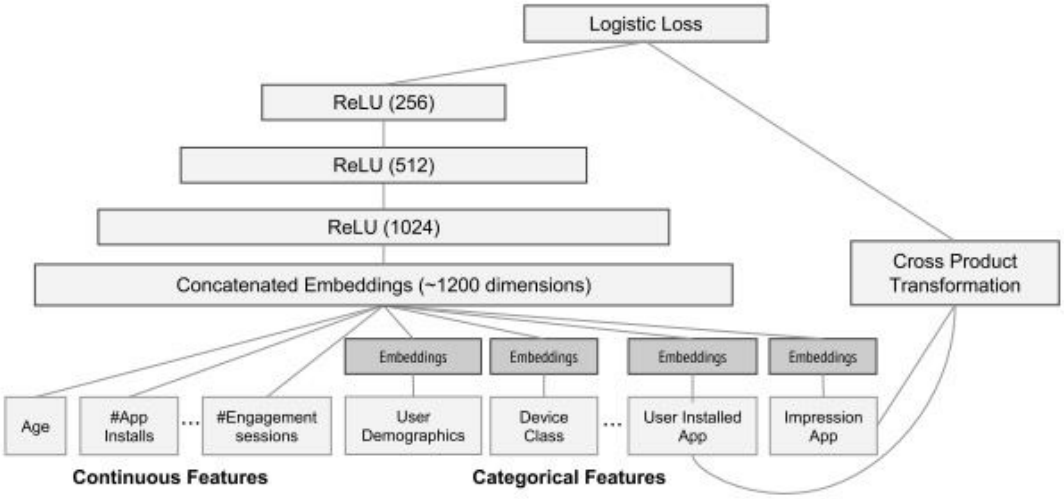

Figure 4: Wide & Deep model structure for apps recommendation.

作者采用这种“重Deep，轻Wide”的结构完全是根据应用场景的特点来的。Google play因为数据长尾分布，对于一些小众的app在历史数据中极少出现，其对应的Embedding学习不够充分，需要通过Wide部分Memorization来保证最终预测的精度。

作者在训练该模型时，使用了5000亿条样本（惊呆），这也说明了Wide&Deep并没有那么容易训练。为了避免每次从头开始训练，每次训练都是先load上一次模型的得到的参数，然后再继续训练。有实验说明，类似于FNN使用预训练FM参数进行初始化可以加速Wide&Deep收敛。

3.3 Model Serving

在实际推荐场景，并不会对全量的样本进行预测。而是针对召回阶段返回的一小部分样本进行打分预测，同时还会采用多线程并行预测，严格控制线上服务时延。

4. 实验结果

作者在线上线下同时进行实验，线上使用A/B test方式运行3周时间，对比收益结果如下。Wide&Deep线上线下都有提升，且提升效果显著。

Table 1: Offline & online metrics of different models. Online Acquisition Gain is relative to the control.

Model	Offline AUC	Online Acquisition Gain
Wide (control)	0.726	0%
Deep	0.722	+2.9%
Wide & Deep	0.728	+3.9%

5. 优缺点分析

优点：

- 简单有效。结构简单易于理解，效果优异。目前仍在工业界广泛使用，也证明了该模型的有效性。
- 结构新颖。使用不同于以往的线性模型与DNN串行连接的方式，而将线性模型与DNN并行连接，同时兼顾模型的Memorization与Generalization。

缺点：

- Wide侧的特征工程仍无法避免。

实践

依旧使用 $MovieLens100K dataset$ ，核心代码如下。其中需要注意的是，针对Wide部分采用了 $FTRL$ 优化器，Deep部分使用了 $Adam$ 优化器。

```
class WideDeep(object):
    def __init__(self, vec_dim=None, field_lens=None, dnn_layers=None, wide_lr=None, l1_reg=None, d
        self.vec_dim = vec_dim
        self.field_lens = field_lens
        self.field_num = len(field_lens)
        self.dnn_layers = dnn_layers
        self.wide_lr = wide_lr
        self.l1_reg = l1_reg
        self.deep_lr = deep_lr

        assert isinstance(dnn_layers, list) and dnn_layers[-1] == 1
        self._build_graph()

    def _build_graph(self):
        self.add_input()
        self.inference()

    def add_input(self):
        self.x = [tf.placeholder(tf.float32, name='input_x_%d'%i) for i in range(self.field_num)]
        self.y = tf.placeholder(tf.float32, shape=[None], name='input_y')
        self.is_train = tf.placeholder(tf.bool)

    def inference(self):
        with tf.variable_scope('wide_part'):
            w0 = tf.get_variable(name='bias', shape=[1], dtype=tf.float32)
```

```
        linear_w = [tf.get_variable(name='linear_w_%d'%i, shape=[self.field_lens[i]], dtype=tf.
        wide_part = w0 + tf.reduce_sum(
            tf.concat([tf.reduce_sum(tf.multiply(self.x[i], linear_w[i]), axis=1, keep_dims=Tru
            axis=1, keep_dims=True) # (batch, 1)
    with tf.variable_scope('dnn_part'):
        emb = [tf.get_variable(name='emb_%d'%i, shape=[self.field_lens[i], self.vec_dim], dtype
        emb_layer = tf.concat([tf.matmul(self.x[i], emb[i]) for i in range(self.field_num)], ax
        x = emb_layer
        in_node = self.field_num * self.vec_dim
        for i in range(len(self.dnn_layers)):
            out_node = self.dnn_layers[i]
            w = tf.get_variable(name='w_%d' % i, shape=[in_node, out_node], dtype=tf.float32)
            b = tf.get_variable(name='b_%d' % i, shape=[out_node], dtype=tf.float32)
            in_node = out_node
            if out_node != 1:
                x = tf.nn.relu(tf.matmul(x, w) + b)
            else:
                self.y_logits = wide_part + tf.matmul(x, w) + b

    self.y_hat = tf.nn.sigmoid(self.y_logits)
    self.pred_label = tf.cast(self.y_hat > 0.5, tf.int32)
    self.loss = -tf.reduce_mean(self.y*tf.log(self.y_hat+1e-8) + (1-self.y)*tf.log(1-self.y_hat

    # set optimizer
    self.global_step = tf.train.get_or_create_global_step()

    wide_part_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='wide_part')
    dnn_part_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='dnn_part')

    wide_part_optimizer = tf.train.FtrlOptimizer(learning_rate=self.wide_lr, l1_regularization_
    wide_part_op = wide_part_optimizer.minimize(loss=self.loss, global_step=self.global_step, v

    dnn_part_optimizer = tf.train.AdamOptimizer(learning_rate=self.deep_lr)
    # set global_step to None so only wide part solver gets passed in the global step;
    # otherwise, all the solvers will increase the global step
    dnn_part_op = dnn_part_optimizer.minimize(loss=self.loss, global_step=None, var_list=dnn_pa

    self.train_op = tf.group(wide_part_op, dnn_part_op)
```

reference