特征重要性分析——现有方法总结

特征的重要性一直缺少直观的评估方法。有时精心设计的slot带来的收益却不明显。能否给出一个高效的特征选择方案，对现有模型的冗余特征进行删减，加快推理和训练速度，减小对硬件资源的需求，并对新特征的设计提供一个相对可靠的指导。

1 第一类方法

这类方法主通过自定义评价指标，来判断特征重要性。

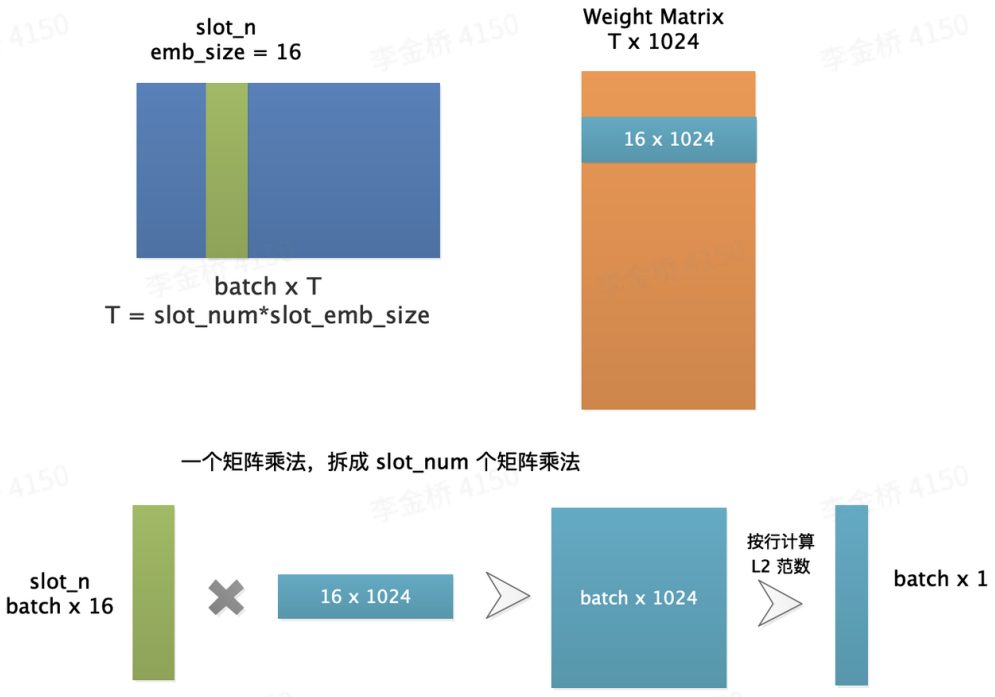
1.1 最初的尝试

原文档：特征重要性分析

1.1.1 基本思路

整体流程：定义了一个slot重要性的评价指标，根据这个重要性指标来删除部分不重要的slot，然后再重新训练。如果重新训练后，AUC并没有下降，则说明被筛除的slot确实不重要。

slot重要性指标：计算每个slot embedding在第一个NN层中的L2范数，其值越大，则认为越重要。



如图，正常的流程中，我们会把slot拼接起得到embedding矩阵，然后输入到一个全连接层中。embedding的shape为(batch_size, T)，全连接层的参数矩阵W的shape为(T, 1024)，前向传播就是将

这两个矩阵相乘得到shape为(batch_size, 1024)的矩阵，后续再加上bias以及通过激活函数。

在这个方法中，如果想要度量某个slot的重要性，就在embedding矩阵中将该slot对应的embeeding截取出来（绿色区域），并将其乘以参数矩阵W中对应的切片（青色区域），得到临时矩阵，其shape为(batch_size, 1024)。之后再按行计算L2范数，其shape是(batch_size, 1)。那么这里这个L2范数向量就是该slot在这个batch上的重要性度量。紧接着可以直接在batch维度求和，然后不同的batch的结果求平均，最终每个slot计算出一个重要度（标量）。

个人理解：从矩阵相乘的规则上来看，正常流程的计算结果其实就等价于把各个slot切片计算得到临时矩阵（shape为`(batch_size, 1204)`）按照元素相加，哪个slot的临时矩阵的值越大，对最终的结果的贡献就越大。那么求L2范数其实就相当于去衡量临时矩阵的大小——即对第一层NN的激活值的贡献。

1.1.2 实现代码

将重要性指标的计算嵌入到前向传播中，可以在训练结束后直接得到各个slot的重要性。实现方式是修改原先`sail.layers.dense`类的`call`函数，下面是原本的代码：

Python

```python
def call(self, inputs):
    output = S.dot(inputs, self.kernel)
    if self.use_bias:
        output = S.bias_add(output, self.bias)
    if self.activation is not None:
        output = self.activation(output)
    self._register_for_debug('output_for_layer_{}'.format(self.name), output)
    return output
```

`call`函数修改如下，其中`input_info`中包含了每个slot对应的维度以及起始下标。

Python

```python
def call(self, inputs, **kwargs):
    inputs_info = kwargs['inputs_info']
    slot_outputs = []
    for slot_id, slot_dim, start_pos in inputs_info:
        slot_emb    = tf.slice(inputs, [0, start_pos], [-1, slot_dim])
        slot_weight = tf.slice(self.kernel, [start_pos, 0], [slot_dim, int(self.kernel.shape[1])])
        slot_output = S.dot(slot_emb, slot_weight)
        slot_energy = tf.norm(slot_output, ord=2, axis=1)
        tf.summary.histogram('slot_{}_energy_dim_{}'.format(slot_id, slot_dim), slot_energy)
        slot_outputs.append(slot_output)
    output = tf.add_n(slot_outputs)
    if self.use_bias:
        output = S.bias_add(output, self.bias)
    if self.activation is not None:
        output = self.activation(output)
    self._register_for_debug('output_for_layer_{}'.format(self.name), output)
    return output
```

注意：由于slot的数量可能较大，这样把一个大矩阵相乘改为多个小矩阵相乘，会导致forward巨慢。这里该作者做了一个折衷，并不按照slot来切分矩阵，而是将同种类型的slot合并在一起，将其作为一个slot_section，按照slot_section来对矩阵进行切分，减小切分数量，加快前向传播的速度。

1.1.3 效果展示

该作者在头条 cvr 模型（172346，去掉了 recall info）上进行了实验：FS result - V2 。作者定义了29个slot_section，并计算了每个slot_section的重要度。进一步的，**头条 cvr 模型特征筛选** 和 特征重要性验证 显示，去除掉了NN 结构中24%的维度，离线指标和实验指标基本无损，online ps 减少 16%，相同 QPS 情况下，pilot 的计算资源占用节省 4.5%。

1.1.4 不足之处

1. 需要修改模型代码，矩阵相乘被分解为多个矩阵相乘相加，前向传播的效率低下；如果使用slot_section，又不能细分到slot粒度；
2. 定义的重要性指标对覆盖率低的特征不公平；那些出现次数较少的特征，可能多次计算出的L2范数为0，从而拉低平均值即重要度；（也不一定是缺点）
3. 评估完成后的特征选择仍然是个问题。该方法需要确定一个重要度阈值，不重要的特征将被删除，然后又重新train模型看实际的AUC，这个过程需要多次尝试以确定较好的特征集，周期仍然很长。

1.2 工程上的改进——工具的构建

Sparse Model 特征重要性分析 与1.1在算法的思路上基本是一致的，只不过在具体的实现上进行了进一步的改进，提升了分析的效率。

具体来说，他们实现了一套离线的工具，不再需要修改模型代码。可以直接在本地物理机上download训练好的模型和少量的数据（1个切片），然后在这些少量的数据上评估各个slot的重要性。

1. 使用训练好的模型，并且无需跑完整个forward即可得到slot的重要度，无需再做slot_section，能够做到slot粒度；
2. 能够根据concat节点的name，自动向前追溯到对应的slot，不需要人为指定slot对应的位置；

1.2.1 使用方法

该作者已经实现了一种快速的离线评测工具，并且该工具已经嵌入到了Lagrange Lite的SDK中，可以以工具脚本的形式进行使用。具体使用方法参考 Sparse Model 特征重要性分析 。这里只是简单说一下需要准备哪些东西。

1. `unique_model_name`，即存放在haven上的模型的名字，是已经训练好的。在评价slot重要性的时候，模型中的任何权重参数都是不会被更新的。
2. `train_paths`，即数据路径，需要先通过hadoop命令将hdfs上的数据拿下来（少部分），如：

```shell
hadoop fs -text hdfs://haruna/ss_ml/union_ad/ug_deep/ug_arpu_pb_ins_daily_v11/20200831/part-00000.pb.snappy >part-00000.pb
```

3. 使用工具时需要传入embedding拼接层（ConcatV2操作）的节点名`concated_tensor_name`，该工具需要根据这个节点名来找到计算重要度需要的参数或者节点。可以通过在代码的model_fn中打印Tensor名，也可以通过TensorBoard工具来找到这个节点对应的名字。

1.2.2 源码逻辑

通过看该工具的源码（demo和API），发现其具体实现的逻辑如下：

1. 根据`unique_model_name`从hdfs上下载模型到本地，并建立模型结构图；
2. 根据`train_paths`构建输入数据（之前已经下载到本地）；
3. 根据`concated_tensor_name`找到concat层之后的Dense层的参数矩阵；

4. 根据`concated_tensor_name`尽可能从concat层开始向前追溯，找到每个slot节点的名字及其维度dim，如果concat之前有类似FM的交叉操作导致无法回溯，则记录其tensor名。
5. 循环n个batch，每个batch内：
   a. 取得这个batch对应的输入数据；
   b. 按照1.1中提到的重要性度量方式，计算每个slot的重要性并记录下来。（如果无法回追溯到slot，则记录tensor的重要性）
6. 多个batch上slot的重要性取均值。

1.2.3 不足之处

1. 该工具暂时只支持Lite模型，不支持sail模型；
2. 如果有FM等交叉操作，还是不能追溯到对应的slot，仅给出交叉后tensor的重要性；对于LR部分的sum(bias)操作，不能再向前追溯；
3. 定义的重要性指标对覆盖率低的特征不公平；
4. 评估完成后的特征选择仍然是个问题。

1.3 工程上的进一步改进——平台化

原文档：FeatureInsight：特征重要性分析平台化 。

算法的基本思想大体不变，但在工程方面，对1.2进行了改进，最终融入到**tensorsharp**中。

1. 设计新的LR部分的slot bias的重要性评价指标。（平方）
2. 提供离线与训练伴生两种使用方式，可以根据线上剩余资源灵活选择。
3. 支持oracle-tf、sail和Lite模型。

1.3.1 使用方法

使用方法有两种：**离线工具**与**训练伴生**。详细的使用方法请参考 FeatureInsight：特征重要性分析平台化 和 Tensorsharp操作手册 ，下面仅做简单梳理。

离线使用：

如果离线使用，则需要先在物理机上安装tensorsharp，然后按照如下流程执行命令（需要添加参数）。

```shell
---1. dump需要的model，传入model-name和zk-path---
tensorsharp dump

---2. fetch需要的model input data---
tensorsharp fetch

---3. 获得指定concat节点的值，对于vector就是进入NN第一层MatMul节点的输入Tensor，对于bias就是concatenate_tensor_from_xxx_bias。可以指定多个Tensor name---
tensorsharp run

---4. 重要性分析，输出xxx_slot.csv/xxx_bias.csv---
tensorsharp slot-embedding-measure
```

注意，由于第三步可以指定多个concat节点，最后给出的结果中每个concat节点对应一组结果。

在forge平台训练伴生：

目前还有一定的局限性：仅支持batch训练模式，且用于训练伴生系统的数据无法再进行训练，推荐用于评估的数据集大小为10000。

在【训练模式配置】模块中【启用Tensorsharp训练伴生测试】，且配置参数如下：

`--concat-tensor-name=concat_1:0` NN的input tensor，如果需要评估多个需要用','隔开；

`--hdfs-path=/path/to/your/hdfs/` HDFS_PATH，用于存储评估结果csv文件；

`--slot-interval=60` 进行评估的间隔时间为（slot-interval）单位为秒；

`--slot-nr-instance=10000` 用于评估的数据集大小；

`--runstep=predict_online` 用于评估的计算图的名称（如果仅设置了一个inference runstep那么可以不设置该参数）

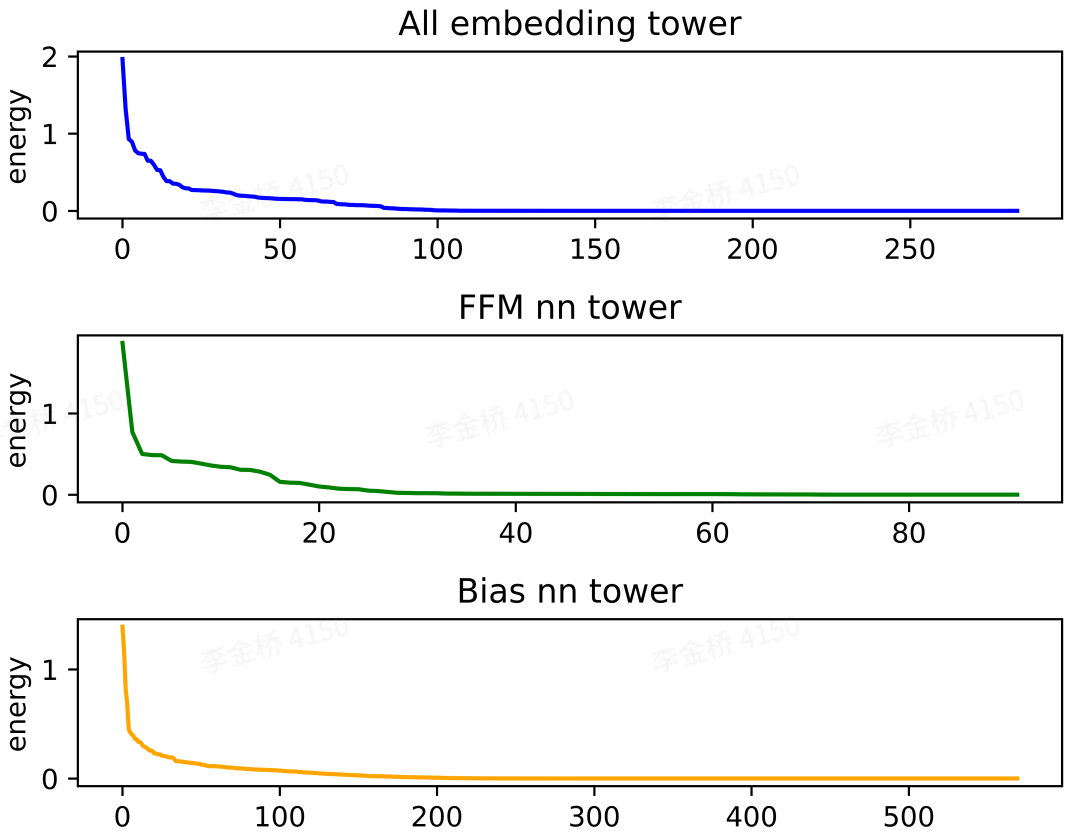
注意，使用forge上的Tensorsharp训练伴生，也可以load一个训练好的模型，只评估特征重要性而不进行训练。

1.3.2 不足之处

1. 如果有FM等交叉操作，还是不能追溯到对应的slot，仅给出交叉后tensor的重要性；
2. 定义的重要性指标对覆盖率低的特征不公平；
3. 评估完成后的特征选择仍然是个问题。需要多次尝试以确定较好的特征，这个过程的周期仍然很长。

1.3.3 初步尝试

尝试使用这个方法，分析aweme_anchor_click_mt_8targets_v256_base_r1025691_0模型所用到的特征的重要性。使用的是20210615/23_task_dorado_103967659_4_8559.1623772431810.pb.zstd分片的10000条数据。最终重要性结果为：aweme_slot.xlsx 。



---

2 第二类方法

这类方法不定义重要性评价指标，而是通过“去掉特征后看AUC降幅”来间接判断特征的重要性。

2.1 Mask Embedding

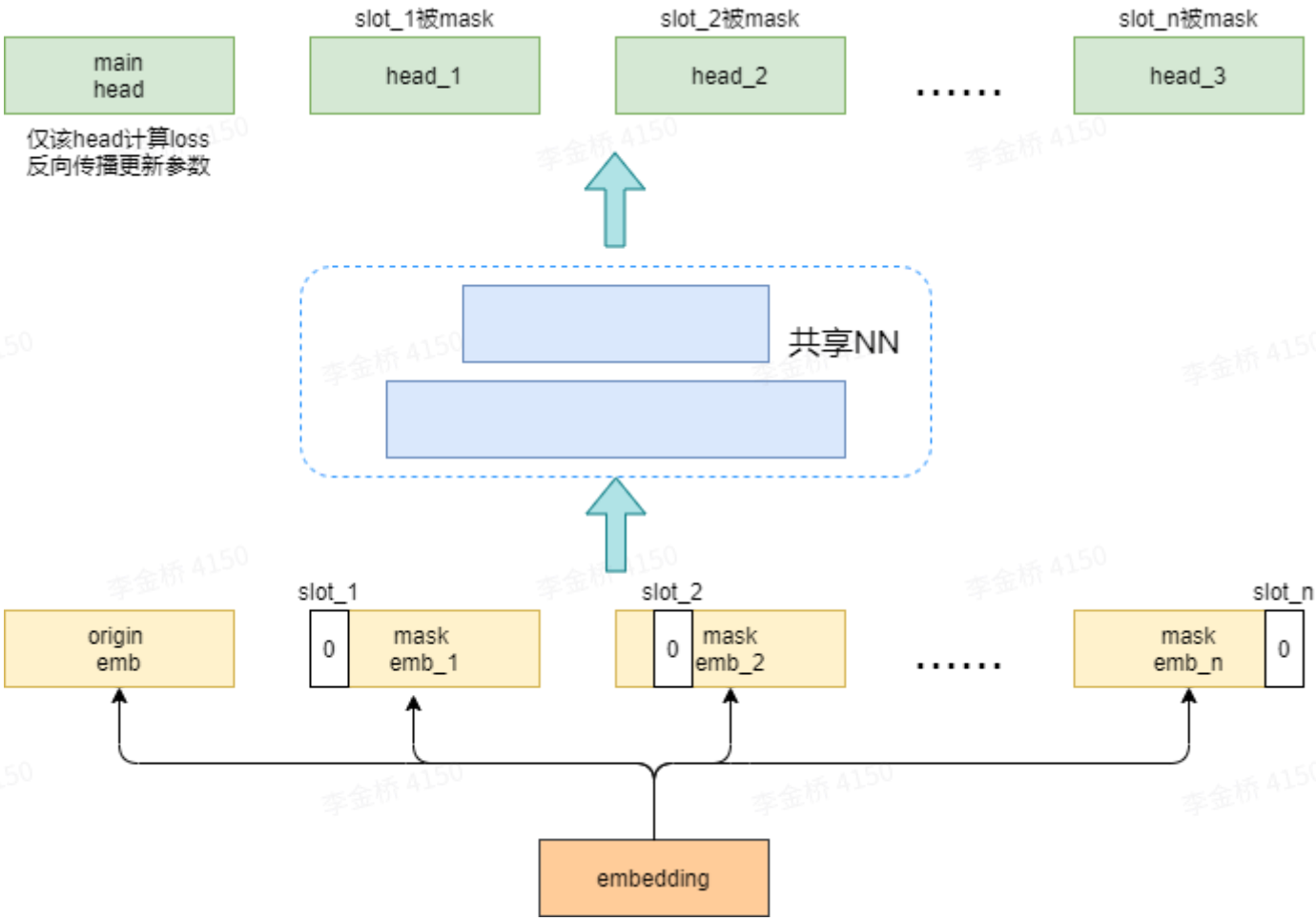
原文档：Feature selection using per slot auc drop 。

2.1.1 基本思路

首先从一个最直观、最暴力的角度去想，我们要判断一个slot是否重要，那么就直接删掉它，然后整个模型重新训练，去看AUC等指标是否会下降太多。这样得到的结果是相对最准确的，但是过于耗时。

因此该方法做了一个近似，在一个流程中批量地评估了所有slot的重要性。具体的操作如图所示：首先预加载之前某个时间点的模型参数，用main head来进行与线上模型一样的训练任务（与之前的训练保持一致）。每一个需要评估重要性的slot都增加一个head，所有head共享embedding以及NN层参数。而每一个head对应的输入有所不同，main head对应原始的embedding，而其他新增的head，其对应的slot会被mask掉，也就是变成0向量。这些新增的head相当于就是在计算某个slot不存在的情况下的预估结果。并且，仅有main head会计算损失和回传梯度，其他head仅预估。

最终，再通过1~2天（时间可调）的训练，就能够得到每个head的AUC。统计main head与其他各个head的AUC的差异，然后删掉那些AUC降幅低于一定的阈值的特征（不重要的特征）。最后还可以再重新train以得到实际的AUC。



2.1.2 实现代码

具体实现可以参考作者提供的demo，下面仅对关键步骤作说明。

1. 首先定义好select_slots，即需要衡量重要性的slot的id列表；
2. 然后构造mask_embedding（一个embedding扩充为n+1个embedding），被mask的slot对应的维度会被乘以0；

3. 添加main head和其他head，main head需要计算loss并回传梯度，其prediction由original embedding计算；其他各个head，不需要计算loss，其prediction由对应的mask embedding计算。

Python

```
----1. 定义好select_slots，其内容为所有需要评估的slot id----
select_slots = [1, 2, 3, 4, ...]
select_slots = [0] + select_slots    # id为0表示使用original embedding

----2. 对应slot的embedding进行mask----
mask_embedding = {}
    for slot in select_slots:
        mask_embedding[slot] = []
        for key in all_embedding_slots:
            if slot == key:
                # 对应slot的embedding需要mask
                mask_embedding[slot].append((all_embedding[key][0]*0, all_embedding[key][1]))
                continue
            mask_embedding[slot].append((all_embedding[key][0], all_embedding[key][1]))

----3. 添加main head和mask slot head----
if slot == 0:
    run_train.add_feeds(M.get_all_input_tensors()).add_gradients()
    run_train.add_head(name='cvr_head_'+str(slot),
                               prediction=y_pred,    # 由original emb计算
                               label=M.get_label('CVR_LABEL', 0),
                               loss=loss,
                               sample_rate=M.get_sample_rate())
else:
     run_train.add_head(name='cvr_head_'+str(slot),
                               prediction=y_pred,    # 由mask emb计算
                               label=M.get_label('CVR_LABEL', 0),
                               loss=None,
                               sample_rate=M.get_sample_rate())
```

2.1.3 效果展示

这里是别人文档中的，我们的模型正在跑；

操作：

- SG CVR模型的207个特征被删减69个，特征数减少33.3%；
- all embedding层维度从5096降到3880，减小23.8%；
- ps内存从630G降到479G，减小23.9%。

效果：

- 重train后流式AUC +0.02%，持平。
- 广告主价值+0.82%，Cost+2.79%，Send+2.14%，Convert-0.31%，打平Send广告主价值持平。

2.1.4 不足之处

1. 如果有FM等交叉操作，只能mask交叉后tensor，不能追溯到slot层面；
2. 虽然相对暴力方法（每个slot删掉后训练一次）效率高的很多，但是如果添加的head过多，效率还是比较低；（有待验证）
3. 一次仅去掉一个slot，大多数head的AUC变化可能不明显；（有待验证）
4. 评估完重要性后，特征选择仍然是个问题。

---

3 第三类方法

这类方法主要是结合了第一类和第二类方法，使用自定义的评价指标来衡量特征的重要性，并基于mask的思想来快速地在线上验证结果。

3.1 FeatureInsight 1.0

原文档：FeatureInsight 1.0 特征选择方案 。

3.1.1 基本思路

之前的方法在评估完重要性后的特征选择阶段效率比较低，通常都先根据阈值做特征选择，并重新训练一个新的模型来离线对比AUC，并在线上开实验对比效果，周期较长。

因此该作者开发及验证了一套低成本、准确率高的pipeline：

1. 基于第一类方法（1.3的tensorsharp）进行特征重要性分析；
2. 基于第二类方法中mask的思想（mask的方式不一样），直接在线上对第1步的slot重要性结果做校验，进行特征选择；
3. 重训模型（NNwarmup或者完全重训），并最终线上实验，完成模型替换。

3.1.2 具体实现

pipeline的第一步是使用1.3的方法来进行特征重要性分析，这里不再赘述。作者做了一个补充的实验，发现在采样**0.1%**（50w样本）的情况下就能够获得比较稳定的评估结果。

pipeline的第二步是基于mask的思想，在线上用**少量流量**快速验证第1步的评估结果，并进行特征选择。这里使用mask的方式与2.1不完全一样，2.1需要添加多个head，而这里仍然只有一个head，仅仅将部分slot的embedding或者bias置零了（第一步认为不重要的slot），并且直接在线上开**AB实验对比效果**。因此，如果置零一部分特征后各类指标基本持平，说明这些特征确实不重要。总的来说，就是说一开始并不真正地删除slot，而是置零，并直接进行线上实验，省略离线训练的过程。从而快速进行特征选择，并且能够减小线下和线上的GAP。

Python

```python
# 示例，在compile之前，添加mask的代码;
mask_slot_list = [319, 53, 12, 561, 6]    # 需要被mask的slot
for slot_id in mask_slot_list:
    slice_num = len(M.get_feature_column(slot_id)._fc_vector_slices)
    for i in range(slice_num):
        # 直接sail里面提供的set_slice_mask接口
        M.get_feature_column(slot_id).set_slice_mask(i)

# 修改完成之后，加载之前的模型参数，直接进行线上的实验;
```

pipeline的第三步是基于第二步选择的特征集合训练模型，可以通过NNwarmup的方式（2.5天）或者完全重新训练模型（11天），然后得到最终的AUC等指标。

3.1.3 效果展示

在头条CTR模型上，选择 20个(14%) slot embedding 和 40个(9%) slot bias 进行删，embedding的dim从2226到1990，在AB实验中核心指标持平。

3.1.4 不足之处

1. 通过mask来置零而非直接删除slot，不训练新的模型而是直接在线上进行实验，只能说是一种近似和折衷；
2. 短时间的上线可能会波动，导致评测不准确；长时间的上线又会导致效率过低（文档中说一般要观察2-3天）；
3. 使用的仍然是1.1的重要性评价指标，缺点也就继承了。

---

4 第四类方法

这类方法通过训练模型来进行特征选择，分为：

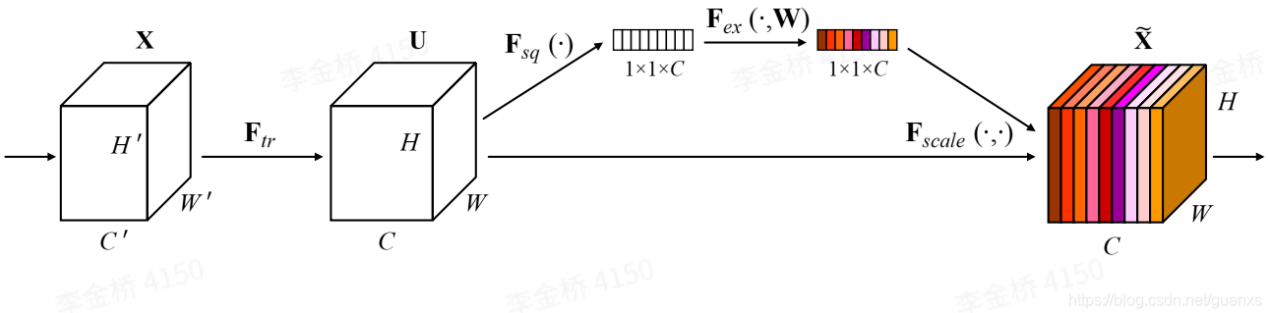
- 单阶段，将特征选择嵌入到模型中，如注意力机制；
- 两阶段，预训练一个简单模型来指导特征的选择，之后再使用选择后的特征训练真正的模型。

4.1 添加SENet结构

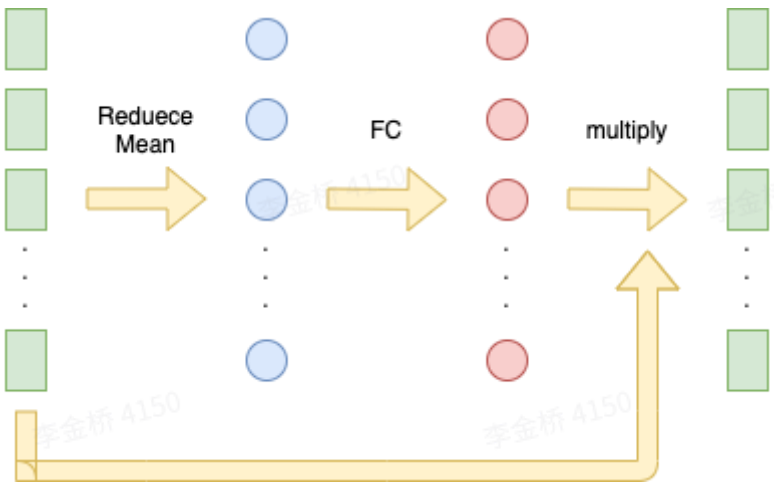
头条CVR 模型加 SENET 结构

4.1.1 基本思路

SENet引入了通道间的注意力机制，为每个通道计算一个权重，每个通道得到不同程度的激励后，再进行后续的操作。简单来说，首先shape为(H, W, C)的tensor进行全局平均池化，得到(1, 1, C)大小的tensor，然后通过全连接层对各个通道的重要性进行预测，此时每个通道就代表了其重要性，跟原来的tensor乘起来之后就得到了激励后的tensor。



借鉴这种attention的思想，n个slot的embedding各自做reduce_mean（对应SENet的池化操作），然后过全连接层得到一个n维的权重向量（slot之间传递信息），然后slot乘以各自的权重得到激励后的slot embedding。



4.1.2 不足之处

1. 模型发生了改变，需要重新训练；
2. 增加了额外的开销；

4.2 使用第二个网络

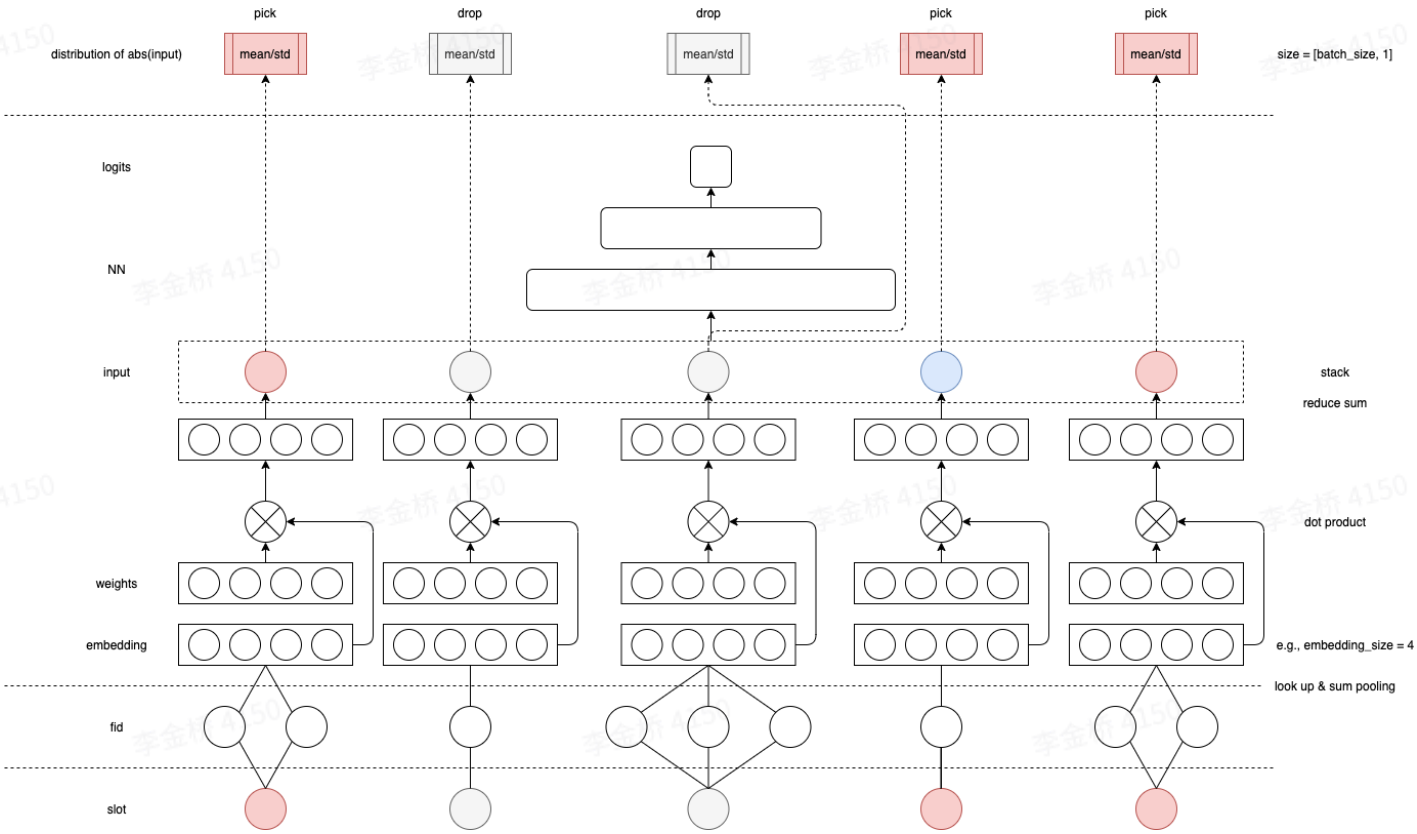
穿山甲付费率模型特征选择

4.2.1 基本思路

另外训练一个结构简单的模型，利用这个模型来衡量特征的重要性，指导特征选择。

如图：每个slot固定是4维的embedding，每个slot各自点乘一个相同维度的weight向量，然后将每个slot再reduce_sum成一个标量，得到一个维度为slot数量的vector，接一个Dense Tower进行训练。

训练完成后，跑一部分的数据，查看input层，其每一个维度就对应了一个slot。计算其mean/std。选择mean大于一定阈值且std大于一定阈值的slot（方差太小，说明不同样本对应的取值一样，没有意义）。



除此以外，[20200805] 穿山甲UG ARPU模型结构调整及特征选择 使用weights层来评价slot的重要性，即看slot对应的weight的L1范数、L2范数和L无穷范数。并且还做了集成，即训练多个用来评估slot重要性的模型（区别在于深度、宽度、激活函数等），如果一个slot在多个模型中重要性都比较高，那么就认为其重要度确实高。

4.2.2 不足之处

1. 训练另外一个模型来评估特征重要性，不需要修改本来模型的代码，但是有一定的GAP；（集成能够一定程度上减小GAP）
2. 也是自定义了一个重要性评价指标，跟第一类方法很类似，但是不如方法一来的直接。

---

5 现有方法的问题以及难点

1. 有FM交叉的情况不好处理；
2. 自定义的特征重要性评价指标不一定准确；（要么使用结果导向，要么使用多个指标做集成）
3. 周期还是比较长；（这一点感觉很难避免）
4. 因为模型不同、数据集不同，现有的不同方法之间很难进行直接的比较。（只能我们自己实验）

核心诉求：

尽量不修改模型，快速、准确地衡量特征重要性，并且尽量减小与线上的GAP。

---

6 其他特征选择的方法

1. 过滤式（仅利用样本，不跑模型）
   a. 方差法
   b. 卡方检验
   c. 相关系数
   d. Relief
   e. ……
2. 包裹式（需要跑模型，根据最终效果选择）
   a. 方法2.1就是包裹式的折衷
3. 嵌入式（特征选择与模型训练同时进行）
   a. L1正则
   b. 树模型
   c. attention某种意义上也算

---

7 下一步的计划

1. 先在我们的模型上尝试已有的方法：
    a. 按照3.1 的方法，剔除不重要特征之后，通过AB实验验证效果；
    b. 完善整个流程的代码和脚本，自动化或者半自动化进行特征选择；
2. 然后再看能否改进现有的方法，或者提出一个新的靠谱的方法。（初步的想法是集成）