1.学习打印规则的过程。

- rule_print()函数：
  是rrl类里面的一个函数。
  变量解释：

```
def rule_print(self, feature_name, label_name, train_loader, file=sys.stdout, mean=None, std=None, display=True):
            #self参数 feature_name是特征每一列的实际意义，y_name是标签的实际意义。file是输出位置。mean是平均值，std是标准差。
```

  为什么传入整个train_loder：要为了后面的检测dead_node提供数据。

- 检测死结点：

> 目的是为了简化模型：
> 如果有结点从来没有被激活过；或者激活的次数等于输入的样本个数(说明所有的样本输入都激活了这个结点)。
>
> 上述两种情况如果发生，对应的结点可以直接删除。因为删除之后，不影响整个模型的分类结果。

  调用函数检测：

```
328         if self.net.layer_list[1].node_activation_cnt is None:    #上面统计了多少次，之后，
329             self.detect_dead_node(train_loader)
```

  当第一个层次(二值化层次)的node_activation_cnt为空，说明之前没有检测过，调用detect_dead_node函数进行检测。

  - detect_dead_node(train_loader)函数：

2.学习detect_dead_node函数：

函数代码：

```
def detect_dead_node(self, data_loader=None):
        with torch.no_grad():
            for layer in self.net.layer_list[:-1]:
                layer.node_activation_cnt = torch.zeros(layer.output_dim, dtype=torch.double, device=self.device_id) #node_activation_cnt就是记录结点被激活多少次。
                layer.forward_tot = 0

            for x, y in data_loader:
                x_bar = x.cuda(self.device_id)
                self.net.bi_forward(x_bar, count=True)
```

函数参数，data_loader是整个数据集。只有重新再跑一次，并且在跑的时候计算`激活情况`，才可以检测dead_node。

- 详细解释每一块代码的作用：
  - for layer in self .net.layer_list[:-1]:
    遍历所有的layer，除了最后一个layer。因为最后一个layer就是用来分类的，这一层次的结点

是一定不会删除的。

`layer.node_activation_cnt` 是一个形状为 `layer.output_dim` 的张量。也就是每一个layer的结点个数。(二值化层次的之后再讨论)

- for x ,y in data_loader:

  `x_bar = x.cuda(self.device_id)` 将输入数据移动到指定的设备（这里是GPU）。

  `self.net.bi_forward(x_bar, count=True)` 是网络的前向传播过程，其中 `count=True` 表示要在前向传播中

  统计节点的激活次数。

- 实际统计激活次数用的是**bi_forward**。具体过程：

```
1         def bi_forward(self, x, count=False):
2               for layer in self.layer_list:
3                     if layer.conn.skip_from_layer is not None:
4                           x = torch.cat((x, layer.conn.skip_from_layer.x_res), dim=1)
5                           del layer.conn.skip_from_layer.x_res
6                     x = layer.binarized_forward(x)
7                     if layer.conn.is_skip_to_layer:
8                           layer.x_res = x
9                     if count and layer.layer_type != 'linear':
10                          layer.node_activation_cnt += torch.sum(x, dim=0)
11                          layer.forward_tot += x.shape[0]
12              return x
```

详细解释每一行代码的作用：(这里没有用到skip)

- `x = layer.binarized_forward(x)`：

  该层的前向传播操作。这个时候，所有的权重都进行了二值化。

- 当count == True 并且 不是最后的 `线性层` 就进行：

  layer.node_activation_cnt += torch.sum(x, dim=0)；layer.forward_tot += x.shape[0]

- **以二值化层次说明下面的过程：**

`binarized_forward` 的过程：

```
73      @torch.no_grad()
74      def binarized_forward(self, x):
75          return self.forward(x)
```

forward过程：

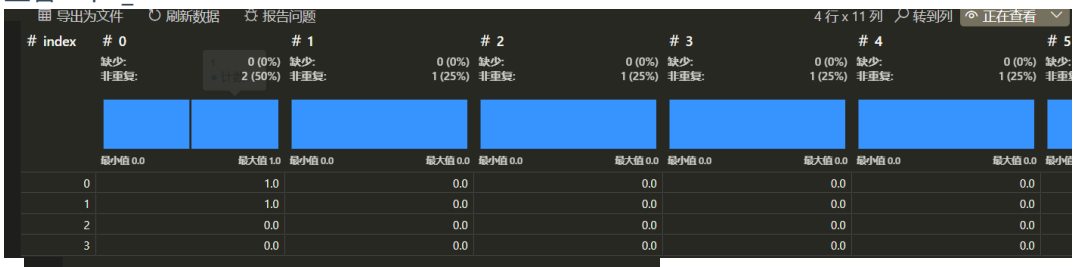
```
61      def forward(self, x):
62          if self.input_dim[1] > 0:
63              x_disc, x = x[:, 0: self.input_dim[0]], x[:, self.input_dim[0]:]
64              x = x.unsqueeze(-1)
65              if self.use_not:
66                  x_disc = torch.cat((x_disc, 1 - x_disc), dim=1)
67              binarize_res = Binarize.apply(x - self.cl.t()).view(x.shape[0], -1)
68              return torch.cat((x_disc, binarize_res, 1. - binarize_res), dim=1)
69          if self.use_not:
70              x = torch.cat((x, 1 - x), dim=1)
71          return x
```

- 详细解释forward过程：（**这里只是二值化层次，每一个具体的层次之间(binarize 和 unionlayer)还会有差别**）

  `if self.input_dim[1] > 0:` #如果有连续数据。

  `x_disc, x = x[:, 0: self.input_dim[0]], x[:, self.input_dim[0]:]` #x_disc存储所有的离散的特征值。x只存储连续特征值。

查看一下x_disc:



x:

`x = x.unsqueeze(-1)` 将连续输入 `x` 添加一个新的维度

- 如果 `self.use_not` 为 `True`，则将离散输入 `x_disc` 与其“取反”的版本拼接在一起

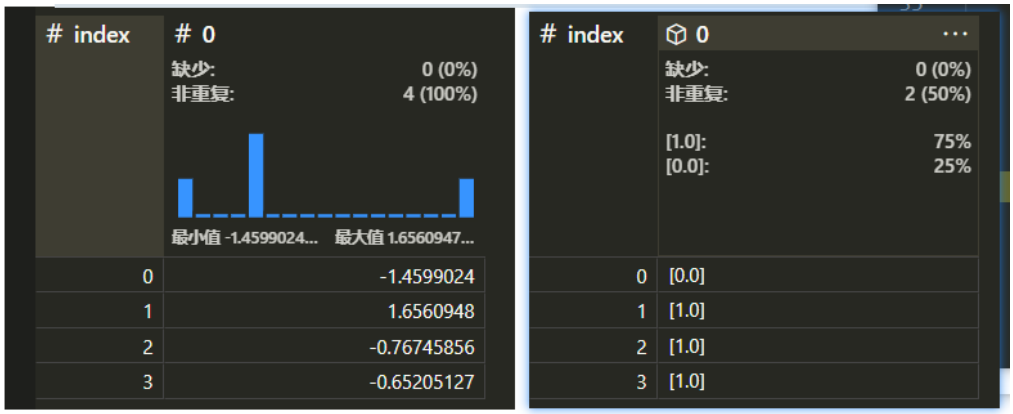
```
1    if self.use_not: #如果 self.use_not 为 True，则将离散输入 x_disc 与其“取反”的版本拼接在一起
2                            x_disc = torch.cat((x_disc, 1 - x_disc), dim=1)
```

`torch.cat((x_disc, 1 - x_disc), dim=1)` 表示在特征维度上拼接原始的离散输入和其“取反”版本，增加了特征的维度。

- binarize_res = Binarize.apply(x - self .cl.t()).view(x .shape[0], -1):
  `Binarize.apply` 是一个自定义的二值化操作。它将 `x` 与某些阈值 `self.cl.t()` 进行操作，然后执行 `view(x.shape[0], -1)`，将结果重塑为一个二维张量，形状为 (批量大小 × 特征数量)。

  `self.cl.t()` 是某种阈值张量，二值化操作的目标是将输入 `x` 与阈值比较，并将结果转化为离散的二值输出（如 `0` 或 `1`）。

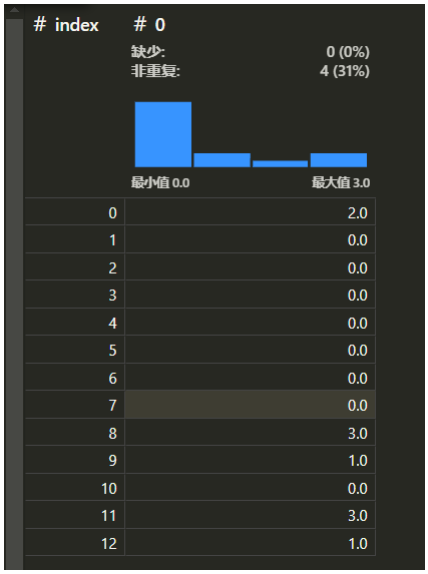
  其实就是将连续特征(用x存储)进行了二值化操作。
  结果是：



左图是原先的连续特征，右图是处理完之后的样子.
右图就是现在代码里面的 `binarize_res` 。

    - return torch.cat((x_disc, binarize_res, 1. - binarize_res), dim =1)
      返回 离散特征 + binarize_res (就是处理了的连续特征) + (1.0 - binarize_res)。
      为什么要 1 — 处理好的连续。因为处理好的连续只是按照阈值大小进行0 1判断。
      我们对于连续特征的处理就是会**用两个结点**表示，一个结点表示大于阈值，另外一个结点表示小于等于。两者一定一个是0，一个是1.

- 上面已经处理完了二值化层次，然后：
  `layer.node_activation_cnt += torch.sum(x, dim=0)` 调用这个函数就实现了将每一个结点是否激活，激活的次数加给了当前层次的node_activation_cnt张量。
  在处理了四个数据之后的情况：



0	2.0
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0
7	0.0
8	3.0
9	1.0
10	0.0
11	3.0
12	1.0

> 总结一下，detect_dead_node函数会通过 `dataloader` 遍历所有的数据，让每一个数据都在已经训练好的模型里面再跑一遍 `调用biforward再跑一遍所有的数据` ，然后将每一个层次的激活情况存储下来。存储在layer.node_activation_cnt张量里面。

3.学习get_bound_name函数。

> self .net.layer_list[0].get_bound_name(feature_name ， mean ， std)
> 这里的self.net.layer_list[0]是binarizelayer。

- 函数的参数：
  1.feature_name是样本里面每一个特征的名称。
  2.mean和std分别是平均值和标准差
  3.self就是binarizelayer
- 声明了 `bound_name` 列表

- 对于离散数据的处理：
  将所有的离散特征的 `feature_name` 添加到bound_name里面。
  bound_name结果：

# index	凸 bound_name ···
	缺少: 0 (0%) 非重复: 11 (100%) workclass_ Self-emp-not-inc: 9% workclass_ State-gov: 9% education_ 7th-8th: 9% 其他: 73%
0	workclass_ Self-emp-not-inc
1	workclass_ State-gov
2	education_ 7th-8th
3	education_ 9th
4	education_ Assoc-acdm
5	education_ Assoc-voc
6	education_ Bachelors
7	education_ Bachelors
8	education_ HS-grad
9	education_ Masters
10	education_ Some-college

- 对于连续数据的处理：

```
if self.input_dim[1] > 0: #如果input_dim[1]>0说明有连续的特征
    for c, op in [(self.cl, '>'), (self.cl, '<=')]:
        c = c.detach().cpu().numpy()  #将 c（即 self.cl）从 GPU 转移到 CPU，并转换为 NumPy 数组，以便于后续的处理和访问。
        for i, ci in enumerate(c.T):  #对 c 的转置进行遍历，ci 是当前特征的阈值。i 是特征的索引。
            fi_name = feature_name[self.input_dim[0] + i]
            for j in ci:
                if mean is not None and std is not None:
                    j = j * std[fi_name] + mean[fi_name]
                bound_name.append('{} {} {:.3f}'.format(fi_name, op, j))
self.rule_name = bound_name
```

  self.cl是当前层次里面的 `所有阈值` 。
  fi_name就是对应特征的 `实际意义的名称` ，这里就是"age"。
  每一个阈值都会有 大于 和 小于等于 两种情况。
  同时每一个阈值都会进行阈值 $*\ std + avg$ 的变化。

  上述完成之后bound_name后面又添加了：

```
11 = 'age > 27.006'
12 = 'age <= 27.006'
```

  之后把整个bound_name赋值给self.rule_name变量。
  就成功将这一个层次的 `第i个结点` 的实际意义在 `self.rule_name[i]` 里面成功存储。

4.继续回到rule_print函数：

```
335         for i in range(1, len(self.net.layer_list) - 1): #
336             layer = self.net.layer_list[i]  #先把这一个层次的结构给layer
337             layer.get_rules(layer.conn.prev_layer, layer.conn.skip_from_layer)  #第一个参数是上一个层次是什么，这里是对于unionlayer处理，
338             skip_rule_name = None if layer.conn.skip_from_layer is None else layer.conn.skip_from_layer.rule_name
339             wrap_prev_rule = False if i == 1 else True  # do not warp the bound_name
340             layer.get_rule_description((skip_rule_name, layer.conn.prev_layer.rule_name), wrap=wrap_prev_rule)
```

- 从索引1开始循环遍历网络的层列表 `self.net.layer_list` ，遍历的范围是从第2层到倒数第2层（因为范围为 `1` 到 `len(self.net.layer_list) - 2` ）。第一层和最后一层不在这个处理的逻辑里面。
- 从层列表中取出当前索引 `i` 对应的层，赋值给变量 `layer` 。
- 调用当前层的 `get_rules` 方法，传入两个参数： `layer.conn.prev_layer` 和 `layer.conn.skip_from_layer` ，分别表示上一层（ `prev_layer` ）和跳过的层（ `skip_from_layer` ）。
  这里上一层是 `binarizelayer` ，跳过的层是none。

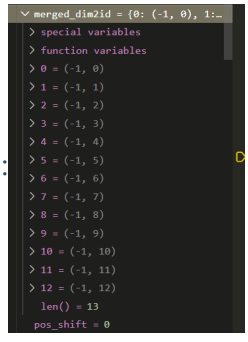
- 处理跳过的层的规则名。(因为后面的get_rule_description函数需要这个参数)如果跳过的层 `skip_from_layer` 为空，则 `skip_rule_name` 为 `None`，否则取跳过层的规则名 `rule_name`。
  这里就是none。
- `wrap_prev_rule` 决定是否要包装上一层的规则名。根据逻辑，只有在处理第2层（索引为1）时，不包装规则名，其他情况下需要包装。
  这里的 "wrap" 可能是指是否对规则名进行某种变换或处理（例如加上某种符号或结构）。

  > 这里的wrap等到后面研究了get_rule_description函数之后再补充。

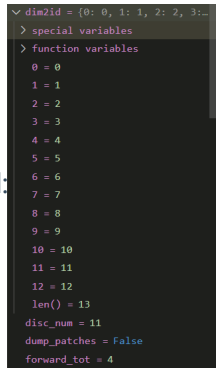
- 最后调用get_rule_description函数，两个参数：
  第一个参数是(跳过的规则的名称，上一层次的`每一个结点的实际意义`)
  第二个参数就是wrap。不过这里是false。

5.学习extract_rules函数

- dim2id变量：
  是一个字典，实际意义是 维度到编号的映射。
  声明：`dim2id = defaultdict(lambda : -1)`。
  defaultdict是一个`允许给没有声明value的key添加默认值。`这里添加的是-1.
  就类似于之前的map<int,int>mp里面，如果一个没有声明过的变量mp[x]的默认值 就是 0 。
  声明为defaultdict作用：当某个维度没有定义规则 ID 时（比如该维度可能是无效维度或未被激活），代码希望返回一个默认值 `-1`。
- `Wb = (layer.W.t() > 0.5).type(torch.int).detach().cpu().numpy()`：
  权重二值化，大于0.5是1，否则为0。保存为numpy数组。
  此时权重的维度是(16,13)
  只是con层次，有16个结点，13是上一个层次的输出维度。
- `merged_dim2id = prev_dim2id = {k: (-1, v) for k, v in *prev_layer*.dim2id.items()}`：
  从前一层的 `dim2id` 生成新的字典 `prev_dim2id`，其中的值变为 `(-1, rule_id)`，表示规则来源于前一层。
  运行之后结果：
  merged_dim2id:



  二值化层次的dim2id:



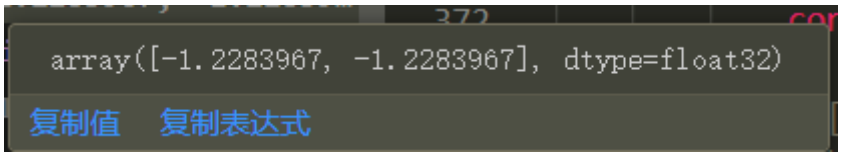
- `for ri, row in enumerate(Wb):`
  ri是遍历Wb的行，从1到最后一行。
  row是一个array，是Wb一行里面所有的权值。
  Wb的每一行`对应`层次里面的一个结点。
  - 当这个结点是`死结点`，赋值dim2id[当前结点编号] = -1.
    同时进行conintue，不进行下面的部分。
  - 这里有一个官方说明：

```
# rule[i] = (k, rule_id):
#             k == -1: connects to a rule in prev_layer,
#             k ==  1: connects to a rule in prev_layer (NOT),
#             k == -2: connects to a rule in skip_connect_layer,
#             k ==  2: connects to a rule in skip_connect_layer (NOT).
```

  - rule 和dound 两个列表的声明：
    字典`rule`，用于存储当前节点的规则。
    `bound`，用于处理在某些层（如`binarization`层）中的特定维度的界限约束
  - 如果前一层是`二值化层次`，同时有连续的特征值：

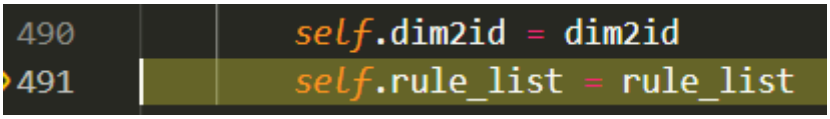
```
if prev_layer.layer_type == 'binarization' and prev_layer.input_dim[1] > 0:
                    c = torch.cat((prev_layer.cl.t().reshape(-1),
prev_layer.cl.t().reshape(-1))).detach().cpu().numpy()
```

    也就是把每一个连续特征的阈值存储下来，存储两遍。
    现在的结果是：



- 对extract_rules结果的说明：(虽然这个过程并没有完全弄懂，但是可以学习一下最后的结果，帮助搞懂这个过程)。

  最后主要生成了两个值：

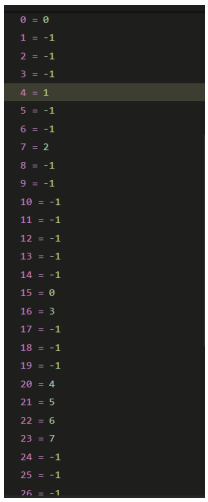


  dim2id。是一个字典，dim2id[key] = value。
  key的取值是[0,31]，每一个都代表一个结点。
  value的取值是[0,9]。因为通过extract_rules之后，一共得到了10种状态。(这个状态的意思是，这个结点是对应哪些的1传过来的)。
  10种里面，3种是con，7种是dis析取。

0	((-1, 12),)
1	((-1, 0), (-1, 11))
2	((-1, 8),)

0	((-1, 0), (-1, 6))
1	((-1, 8), (-1, 12))
2	((-1, 0), (-1, 10))
3	((-1, 0), (-1, 5), (-1, 10))
4	((-1, 0), (-1, 5), (-1, 6), (-1, 10))
5	((-1, 5), (-1, 9))
6	((-1, 6), (-1, 9), (-1, 10))

dim2id的结果：



> 解释：
> 比如对于第23个结点，对应的dim2id[22] = 6。
> 说明对应的就是rule_list里面的下标为6的元素(其实是第七个)，也就是((-1,0),(-1,5),(-1,10)).
> 也就是这一层次的这个结点的含义是 从上一层的第0，5，10个点一起指向自己。

6.学习get_rule_description

> 总结get_rule_description函数，就是 把上面的所有的rule_list都转换为对应的具体的实际意义的集合。