

## 7.学习get\_rules函数:

在我们的层次结构里面，因为一共只有二值化、union、线性层。  
所以get\_rules函数只有第二层unionlayer使用过。

学习get\_rules函数:

- 1 `self.con_layer.forward_tot = self.dis_layer.forward_tot = self.forward_tot`  
2 `self.con_layer.node_activation_cnt = self.dis_layer.node_activation_cnt =`  
`self.node_activation_cnt`

首先将 detect\_dead\_node需要用到的 `forward_tot` 和 `node_activation_cnt` 的结果赋值给当前层次(当前层次就是unionlayer)。

- 1 `con_dim2id, con_rule_list = extract_rules(prev_layer, skip_connect_layer, self.con_layer)`  
2 `dis_dim2id, dis_rule_list = extract_rules(prev_layer, skip_connect_layer,`  
`self.dis_layer, self.con_layer.W.shape[1])`

然后会调用 `extract_rules` 函数来提取组合层和析取层的规则。该函数返回两个值:(具体函数内部的实现, 下面会讲解)。

- `dim2id`: 维度到规则 ID 的映射。
- `rule_list`: 包含相应规则的列表

- `extract_rules`里面的一些解释, 过程在下面总结了, 可以直接看`extract_rules`得到的结果, 这里继续进行`get_rules`函数剩下部分的讲解。

- 1 `con_dim2id, con_rule_list = extract_rules(prev_layer, skip_connect_layer, self.con_layer)`  
2 `dis_dim2id, dis_rule_list = extract_rules(prev_layer, skip_connect_layer, self.dis_layer,`  
`self.con_layer.W.shape[1])`  
3 `#调用 extract_rules 函数, 分别为组合层和析取层提取规则。这个函数返回两个值:`  
4 `# dim2id: 表示维度到规则 ID 的映射。`  
5 `# rule_list: 包含相应规则的列表`  
6  
7 `shift = max(con_dim2id.values()) + 1#获取组合层 con_dim2id 中最大规则 ID 值, 并加 1 赋值给`  
`shift。 这是为了确保析取层的规则 ID 与组合层不重叠, 所以在析取层的规则 ID 上增加一个偏移量`  
`shift。`  
8 `dis_dim2id = {k: (-1 if v == -1 else v + shift) for k, v in dis_dim2id.items()}`  
9 `dim2id = defaultdict(lambda: -1, {**con_dim2id, **dis_dim2id})`  
10  
11 `rule_list = (con_rule_list, dis_rule_list)`

前两行, 对`con_layer`和`dis_layer`进行规则提取, 得到`dim2id`和`rule_list`。

`shift`此时为3, 我们将两个`rule_list`合并在一起组成一个新的列表是`rule_list`。

`shift` 的作用是保证`dis_layer`的`dim2id`的映射还是正确的, `shift`就存储一个偏移量。

这里合并之后, 整个`rule_list`里面有两个元素, 一个是`con_rule_list`里面的具体值, 另外一个是一个是`dis_rule_list`里面的具体值。

## 8.学习extract\_rules函数

- 参数解释:

prev\_layer。存储上一层的有关信息。(这里是Binatizelayer)。

skip\_connect\_layer。存储从哪里skip过来。这里是none。

layer存储当前的层次信息，首先是 originalConjunctionLayer。

pos\_shift目前没有看明白是什么作用。

- 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:

```
merged_dim2id = {0: (-1, 0), 1: (-1, 1), 2: (-1, 2), 3: (-1, 3), 4: (-1, 4), 5: (-1, 5), 6: (-1, 6), 7: (-1, 7), 8: (-1, 8), 9: (-1, 9), 10: (-1, 10), 11: (-1, 11), 12: (-1, 12)}
len() = 13
pos_shift = 0
```

prev\_layer.dim2id.items():

```
dim2id = {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12}
len() = 13
disc_num = 11
dump_patches = False
forward_tot = 4
```

- `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continue，不进行下面的部分。
- 这里有一个官方说明:

```

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

```

- rule 和 bound 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

- 对于第一层的unionlayer加上特判：(如果上一层是二值化层次并且输入的时候有连续的特征)

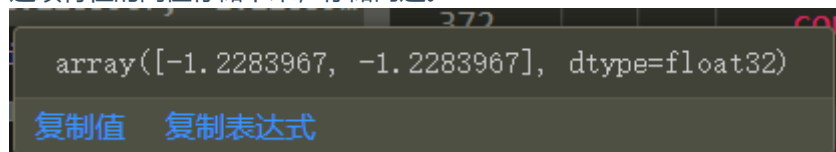
```

1  if prev_layer.layer_type == 'binarization' and prev_layer.input_dim[1] > 0:
2      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：



- 开始遍历一个row里面的权重：

```

1  for i, w in enumerate(row):
2      use_not_mul = 1
3      if layer.use_not:
4          if i >= layer.input_dim // 2:
5              use_not_mul = -1
6          i = i % (layer.input_dim // 2)  #将索引 映射到正确的 输入维度。

```

使用enumerate。i表示row行数的变化，w每一次都是第i行对应的向量。

`use_not_mul` 用1，表示当前没有用到not。这部分主要是用于`rule[i] = (k,id)`里面的k的来源，是否需要乘号。因为有无乘号都有不同的实际意义。

- 每一层次的处理：

```

1  if w > 0 and merged_dim2id[i][1] != -1:

```

- 对于i的解释：

只有满足：`if w > 0 and merged_dim2id[i][1] != -1`：才会往下进行。

- `w > 0`是因为，如果此时`w == 0`，并不需要处理这个点的规则。
- `merged_dim2id[i][1] == -1`的情况，应该是这个结点为死结点的作用。

这里的`merged_dim2id`应该是要不止一个unionlayer的时候才能明白实际的作用

- 之后分为两种：

第一种比较特别：

**(如果当前就是第一个unionlayer(上一层就是binarizelayer)，同时现在的权重连接的上一个结点是表示连续特征的某种意义)** 这个判断在下面代码的if里面判断出来：

这里对于连续特征的具体每一步是如何处理后继续看，目前没有很仔细的都看。

```

1  if prev_layer.layer_type == 'binarization' and i >= prev_layer.disc_num:
2      ci = i - prev_layer.disc_num
3      bi = ci // prev_layer.n
4      if bi not in bound:
5          bound[bi] = [i, c[ci]]

```

```

6         rule[(-1, i)] = 1 # since dim2id[i] == i in the BinarizeLayer
7     else: # merge the bounds for one feature
8         if (ci < c.shape[0] // 2 and layer.layer_type == 'conjunction') or \
9             (ci >= c.shape[0] // 2 and layer.layer_type == 'disjunction'):
10             func = max
11         else:
12             func = min
13         bound[bi][1] = func(bound[bi][1], c[ci])
14         if bound[bi][1] == c[ci]: # replace the last bound
15             del rule[(-1, bound[bi][0])]
16         rule[(-1, i)] = 1
17         bound[bi][0] = i

```

- 另外一种的情况就是**剩余的所有情况**：

```

1     else:
2         rid = merged_dim2id[i]
3         rule[(rid[0] * use_not_mul, rid[1])] = 1

```

rid就是，当前当前结点，对应的规则的表示，这里因为上一层次是二值化层次，所以一定是(-1,i)。

也就是 rid = (-1,i)。之后进行rule[(-1,i)] = 1。

- 上面两种情况处理完毕之后，使用rule = tuple(sorted(rule.keys()))。就可以将原先的rule:{(-1,0):1, (-1,11):1}转变为：{(-1,0),(-1,11)}。

```

1     if rule not in rules:
2         rules[rule] = tmp
3         rule_list.append(rule)
4         dim2id[ri + pos_shift] = tmp
5         tmp += 1
6     else:
7         dim2id[ri + pos_shift] = rules[rule]

```

如果当前的rule没有出现过，dim2id就需要新加一个元素。

某个unionlayer的dim2id[x]实际意义：就表示，**当前层次的第x结点**，表示的规则在**rule\_list**里面存储的位置。rule\_list列表，就存储一个又一个的规则，比如上文的{(-1,0), (-1,11)}就是某个节点的规则，在rule\_list里面是一个**元素**。

- **总结一下extract\_rules函数做的事情：**

是将 **当前每一个结点** 的所代表的具体的规则 保存下来。

**more formally:** 用rule\_list存储当前层次中，结点所代表的所有规则。用dim2id存储每一个结点所代表的规则在rule\_list里面的位置。

- 对extract\_rules结果的说明：

最后主要生成了两个值：

```

490 self.dim2id = dim2id
> 491 self.rule_list = rule_list

```

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。

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:

```
0 = 0
1 = -1
2 = -1
3 = -1
4 = 1
5 = -1
6 = -1
7 = 2
8 = -1
9 = -1
10 = -1
11 = -1
12 = -1
13 = -1
14 = -1
15 = 0
16 = 3
17 = -1
18 = -1
19 = -1
20 = 4
21 = 5
22 = 6
23 = 7
24 = -1
25 = -1
26 = -1
```

○ 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。

○ disj的结果: dim2id:

```
> function variables
16 = 0
17 = -1
18 = -1
19 = -1
20 = 1
21 = 2
22 = 3
23 = 4
24 = -1
25 = -1
26 = -1
27 = -1
28 = 5
29 = -1
30 = 6
31 = -1
len() = 16
i = 12
```

rule\_list:

#	index	rule_list	
		缺少:	0 (0%)
		非逻辑:	7 (100%)
		((-1, 0), (-1, 6)):	14%
		((-1, 8), (-1, 12)):	14%
		((-1, 0), (-1, 10)):	14%
		其他:	57%
	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))	

一共有七种状态。

## 9.学习get\_rule\_description

就是 把上面的所有的rule\_list都转换为对应的具体的物理意义的集合。

上面的rule\_list的具体表示已经说明过，之后进行get\_rule\_description函数:

```
1 def get_rule_description(self, input_rule_name, wrap=False):
2     """
```

```

3     input_rule_name: (skip_connect_rule_name, prev_rule_name)
4     """
5     self.rule_name = []
6     for rl, op in zip(self.rule_list, ('&', '|')):
7         for rule in rl:
8             name = ''
9             for i, ri in enumerate(rule):
10                op_str = ' {}'.format(op) if i != 0 else ''
11                layer_shift = ri[0]
12                not_str = ''
13                if ri[0] > 0: # ri[0] == 1 or ri[0] == 2
14                    layer_shift *= -1
15                    not_str = '~'
16                var_str = '({})' if (wrap or not_str == '~') else
'{}'.format(input_rule_name[2+layer_shift][ri[1]])
17                name += op_str + not_str + var_str
18            self.rule_name.append(name)

```

结果：

rule\_name的结果为：

```

> function variables
0 = 'age <= 27.006'
1 = 'workclass_ Self-emp-not-inc & age > 27.006'
2 = 'education_ HS-grad'
3 = 'workclass_ Self-emp-not-inc | education_ Bachelors'
4 = 'education_ HS-grad | age <= 27.006'
5 = 'workclass_ Self-emp-not-inc | education_ Some-college'
6 = 'workclass_ Self-emp-not-inc | education_ Assoc-voc | education_ Some-college'
7 = 'workclass_ Self-emp-not-inc | education_ Assoc-voc | education_ Bachelors | education_ Some-college'
8 = 'education_ Assoc-voc | education_ Masters'
9 = 'education_ Bachelors | education_ Masters | education_ Some-college'
len() = 10

```

具体过程就是遍历rule\_list集合，将对应的下标和bound\_name里面存储的实际意义结合。就组成了解释性良好的规则。

## 10.学习get\_rule2weights函数。

上述结点对应的规则处理完毕之后，需要知道每一个结点在最后的线性层的权重，所以调用 [线性层的get\\_rule2wieights](#) 函数实现此功能。

主要功能是根据前一层的规则和激活状态计算每个规则对应的权重。

```

1     def get_rule2weights(self, prev_layer, skip_connect_layer):
2         prev_layer = self.conn.prev_layer
3         skip_connect_layer = self.conn.skip_from_layer
4
5         always_act_pos = (prev_layer.node_activation_cnt == prev_layer.forward_tot)
6         merged_dim2id = prev_dim2id = {k: (-1, v) for k, v in prev_layer.dim2id.items()}
7         if skip_connect_layer is not None:
8             shifted_dim2id = {(k + prev_layer.output_dim): (-2, v) for k, v in
skip_connect_layer.dim2id.items()}
9             merged_dim2id = defaultdict(lambda: -1, (**shifted_dim2id, **prev_dim2id))
10            always_act_pos = torch.cat(
11                [always_act_pos, (skip_connect_layer.node_activation_cnt ==
skip_connect_layer.forward_tot)])

```

```

12
13     Wl, bl = list(self.fc1.parameters())
14     bl = torch.sum(Wl.T[always_act_pos], dim=0) + bl
15     Wl = Wl.cpu().detach().numpy()
16     self.bl = bl.cpu().detach().numpy()

```

- 传入的两个参数：  
前一层、skip\_connect\_layer（跳跃的层，此处为none）。
- always\_act\_pos数组：  
表示 prev\_layer 中哪些节点在当前前向传播中总是被激活（即激活计数等于总激活数）。
- 用merged\_dim2id存储一下上一层次的dim2id情况，结果为：

```

> 0 = (-1, 0)
> 1 = (-1, -1)
> 2 = (-1, -1)
> 3 = (-1, -1)
> 4 = (-1, 1)
> 5 = (-1, -1)
> 6 = (-1, -1)
> 7 = (-1, 2)
> 8 = (-1, -1)
> 9 = (-1, -1)
> 10 = (-1, -1)
> 11 = (-1, -1)
> 12 = (-1, -1)
> 13 = (-1, -1)
> 14 = (-1, -1)
> 15 = (-1, 0)
> 16 = (-1, 3)
> 17 = (-1, -1)
> 18 = (-1, -1)
> 19 = (-1, -1)
> 20 = (-1, 4)
> 21 = (-1, 5)
> 22 = (-1, 6)
> 23 = (-1, 7)
> 24 = (-1, -1)

```

- 使用Wl和Bl存储权重和偏置。  
当前层次有2个神经元，上一个层次有32个神经元，W的结构应该为(2,32):



b的结构(2,):



剩下一半的代码：

```

1   marked = defaultdict(lambda: defaultdict(float))
2   rid2dim = {}
3   for label_id, wl in enumerate(W1):
4       for i, w in enumerate(wl):
5           rid = merged_dim2id[i]
6           if rid == -1 or rid[1] == -1:
7               continue
8           marked[rid][label_id] += w
9           rid2dim[rid] = i % prev_layer.output_dim
10
11   self.rid2dim = rid2dim
12   self.rule2weights = sorted(marked.items(), key=lambda x: max(map(abs, x[1].values()))),
                               reverse=True)

```

- 按照结点遍历权重，也就是wl一行一行遍历。然后将每一个位置的权重w加到对应的 `marked[rid][label_id]` 里面，相当于将层次里面的权重存储下来，方便之后进行排序和输出。
- 都遍历完毕之后，成功用字典的形式将对应的规则结点，以及对应的权值存储完毕。使用sorted排序，按照权值大小进行排序。

## 11.rule\_print函数里面的打印过程：

- 经过上面的rule2weights之后，就可以打印最后的规则到rrl.txt里面。
- 打印的具体过程：

```

1   print('RID', end='\t', file=file)
2   for i, ln in enumerate(label_name):
3       print(' {} (b={:.4f})'.format(ln, layer.bl[i]), end='\t', file=file)
4   print('Support\tRule', file=file)
5   for rid, w in layer.rule2weights:
6       print(rid, end='\t', file=file)
7       for li in range(len(label_name)):
8           print(' {:.4f}'.format(w[li]), end='\t', file=file)
9       now_layer = self.net.layer_list[-1 + rid[0]]
10      print(' {:.4f}'.format((now_layer.node_activation_cnt[layer.rid2dim[rid]] /
11                              now_layer.forward_tot).item()),
12            end='\t', file=file)
13      print(now_layer.rule_name[rid[1]], end='\n', file=file)
14  print('#' * 60, file=file)
15  return layer.rule2weights

```

打印结果为：

```

1  RID class_negative(b=0.4893) class_positive(b=-0.1477) Support Rule
2  (-1, 9) -0.2856 0.3595 0.2500 education_Bachelors | education_Masters | education_Some-college
3  (-1, 0) 0.2433 -0.3457 0.2500 age <= 27.006
4  (-1, 6) -0.0789 0.3128 0.5000 workclass_Self-emp-not-inc | education_Assoc-voc | education_Some-college
5  (-1, 7) -0.2199 0.2705 0.5000 workclass_Self-emp-not-inc | education_Assoc-voc | education_Bachelors | education_Some-college
6  (-1, 5) -0.1576 0.2423 0.5000 workclass_Self-emp-not-inc | education_Some-college
7  (-1, 1) -0.1000 0.2251 0.2500 workclass_Self-emp-not-inc & age > 27.006
8  (-1, 8) -0.2193 0.1359 0.2500 education_Assoc-voc | education_Masters
9  (-1, 2) 0.1099 -0.2116 0.7500 education_HS-grad
10 (-1, 4) 0.1519 -0.1442 0.7500 education_HS-grad | age <= 27.006
11 (-1, 3) -0.1494 0.0956 0.5000 workclass_Self-emp-not-inc | education_Bachelors
12 #####
13

```

打印结果的解释：

- `print(rid, end='\t', file=file)`

打印rid，也就是上面的对应的(-1,9)，-1表示从正常连接上一层此，9代表当前结点对应的规则在rule\_list里面的第9个位置。



- `print('{:.4f}'.format(w[li]), end='\t', file=file)`  
打印权重，会按照分类类别的顺序，先后打印当前结点对分类结果的权重，这里是先输出负再输出正。分别是-0.28和3.95
- `print('{:.4f}'.format((now_layer.node_activation_cnt[layer.rid2dim[rid]] /  
now_layer.forward_tot).item()))`  
输出当前结点激活的概率，也就是结点激活次数 比上 forward\_tot。  
因为我们的样本很小，所以激活的结果是{1/4 2/4 3/4 4/4}.
- `print(now_layer.rule_name[rid[1]], end='\n', file=file)`  
输出结点对应规则。（规则之前已经get\_rule\_description里面已经存储过，这里直接输出就可以了）。

还剩下sorted里面的内容不太完整。（最后rrl里面打印的规则的依据）