TabNet[1] 的介绍

- 引言
- 决策树
  - 原理
  - 实例（XGBoost）
- TabNet
  - Tabnet 的整体思想
  - NN 构建 决策树流形
  - Tabnet 的整体网络结构
  - TabNet的各个模块

引言

神经网络（以下简称 NN）在图像、NLP以及语音识别等领域应用广泛，NN 能将它们的原始数据编码成有意义的表示，再进行分类（回归）任务，且取得了显著的效果。

然而，对于一种传统形式的数据--表格数据（tabular data），NN 的表现却不如 决策树模型（以下简称DT），且 DT 有更强的可解释性，但 NN 的端对端学习具有较大的优势。

因此人们想将 NN 和 DT 优势结合起来,设计一种专门用来处理表格数据的网络模型。

决策树

原理

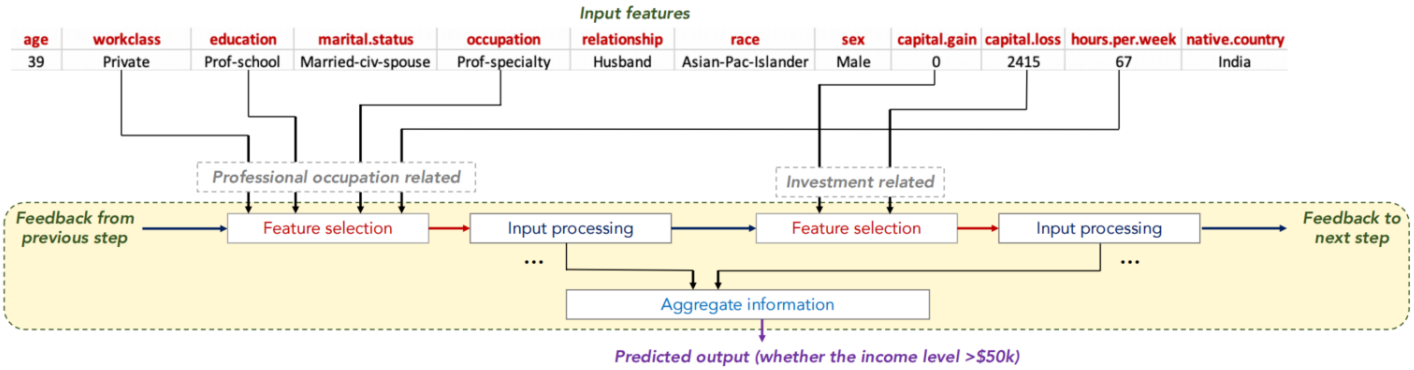
实例（XGBoost）

- **分类任务**
- **回归任务**

TabNet

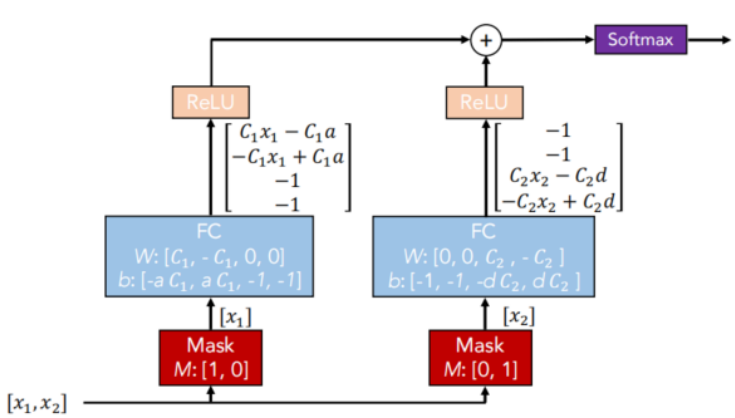
Tabnet 的整体思想

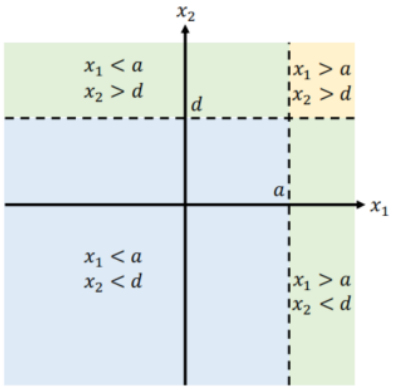
TabNet的核心就是利用 NN 去构建树模型，模拟树模型的决策方式进行决策



TabNet使用多个决策块，每个决策块专注于处理输入特征的一个子集，再将各个决策块的结果汇总，做出预测。如上图所示：两个决策块分别处理与专业（职业）、投资相关的特征，再汇总以预测收入水平。

NN 构建 决策树流形





以两个特征（x_1, x_2）为例：

- 输入特征向量 $x = [x_1, x_2]$ ，通过两个$mask$层得到： $mask\_\ x_1$ 和 $mask\_\ x_2$

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \circ \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} x_1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \circ \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & x_2 \end{bmatrix}$$

- $mask\_\ x_1$ 和 $mask\_\ x_2$ 分别通过权重（w） 和偏置（b）被专门设定过的全链接层（FC）得到：out_1 和 out_2

$$\begin{bmatrix} C_1 \\ -C_1 \\ 0 \\ 0 \end{bmatrix} x_1 + \begin{bmatrix} -aC_1 \\ aC_1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} C_1 x_1 - C_1 a \\ -C_1 x_1 + C_1 a \\ -1 \\ -1 \end{bmatrix}$$

- - - - - -

$$\begin{bmatrix} 0 \\ 0 \\ C_2 \\ -C_2 \end{bmatrix} x_2 + \begin{bmatrix} -1 \\ -1 \\ -dC_2 \\ dC_2 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ C_2x_2 - C_2d \\ -C_2x_2 + C_2d \end{bmatrix}$$

- out_1 和 out_2 再通过 $ReLU$ 激活后，可以保证输出的向量里面只有一个是正值，其余全为0，而这就对应着决策树的条件判断

$$\begin{bmatrix} C_1x_1 - C_1a \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ C_2x_2 - C_2d \\ 0 \end{bmatrix}$$

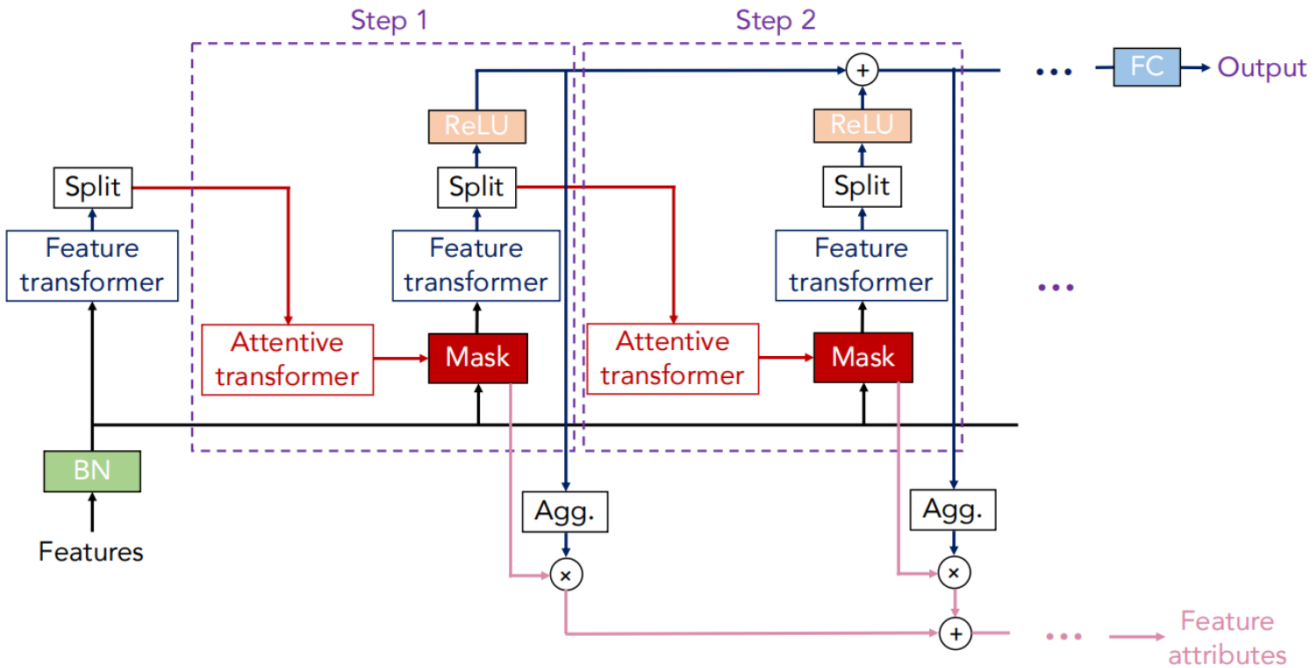
- 将激活后的 out_1 和 out_2 相加

$$\begin{bmatrix} C_1x_1 - C_1a \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ C_2x_2 - C_2d \\ 0 \end{bmatrix} = \begin{bmatrix} C_1x_1 - C_1a \\ 0 \\ C_2x_2 - C_2d \\ 0 \end{bmatrix}$$

- 最后，通过 $Softmax$ 激活后得到最终输出 $output$

Tabnet 的整体网络结构

- **TabNet encoder architecture**

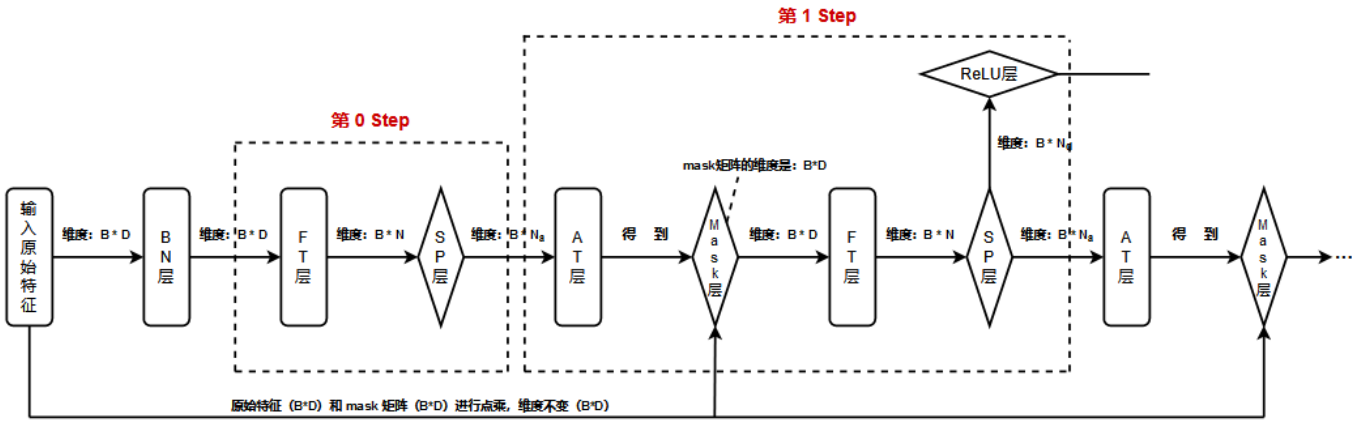


(a) TabNet encoder architecture

前文提到，TabNet 是利用不同决策模块，处理不同的特征子集，再结合各决策块的结果做出最终预测。

上图中的每个$step$代表一个决策块，每个 $step$ 处理的特征子集由 $mask$ 层筛选得到。第 i 个 $step$ 的 $mask$ 层掩码矩阵由第 $i-1$ 个 $step$ 处理后输入到第 i 个 $step$ 的信息来决定。

具体流程图如下：



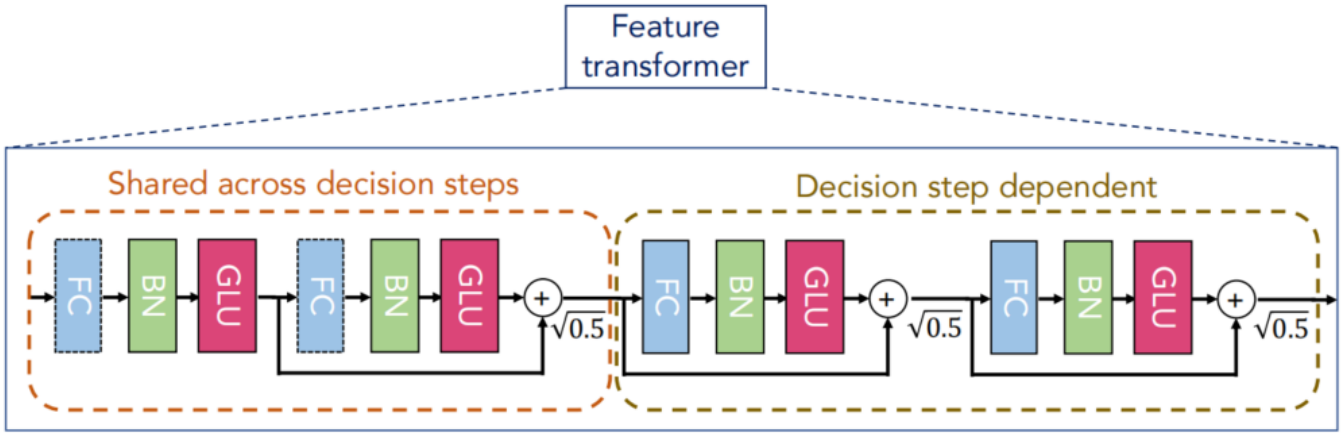
TabNet的各个模块

- **BN（BatchNormal）** [2]

对输入的$Feature$（$f \in \Re^{B\times D}$）进行 $BatchNormal$ 归一化操作，得到 $BN\_Feature$（$f \in \Re^{B\times D}$）

- **Feature Transformer**[3]

用于特征提取，提取出对样本属性更有效的信息表征，提取后得到的特征 $Feature$ （$f \in \Re^{B\times N}$）



feature transformer 模块由两部分组成。前半部分为参数共享层，其参数在所有决策步（$step$）中共享；后半部分为参数独享层，其参数只在当前决策步中使用。

两部分均采用 $FC + BN + GLU$的单元方式,每个单元之间乘以 $\sqrt{0.5}$ 进行残差连接[4]，是为了保证网络的稳定性。

其中：GLU[5]为 $x * sigmoid(x)$，BN 是 $Ghost\ BN$[6]

```
# GLU 源码
class GLU_Layer(torch.nn.Module):
    def __init__(
        self, input_dim, output_dim, fc=None, virtual_batch_size=128, momentum=0.02
    ):
        super(GLU_Layer, self).__init__()

        self.output_dim = output_dim
        if fc:
            self.fc = fc
        else:
            self.fc = Linear(input_dim, 2 * output_dim, bias=False)
        initialize_glu(self.fc, input_dim, 2 * output_dim)

        self.bn = GBN(
            2 * output_dim, virtual_batch_size=virtual_batch_size, momentum=momentum
        )

    def forward(self, x):
        x = self.fc(x)
        x = self.bn(x)
        out = torch.mul(x[:, : self.output_dim], torch.sigmoid(x[:, self.output_dim :]))
        return out


# Ghost BN 源码
class GBN(torch.nn.Module):
    """
    Ghost Batch Normalization
    https://arxiv.org/abs/1705.08741
    """

    def __init__(self, input_dim, virtual_batch_size=128, momentum=0.01):
        super(GBN, self).__init__()

        self.input_dim = input_dim
        self.virtual_batch_size = virtual_batch_size
        self.bn = BatchNorm1d(self.input_dim, momentum=momentum)

    def forward(self, x):
        chunks = x.chunk(int(np.ceil(x.shape[0] / self.virtual_batch_size)), 0)
        res = [self.bn(x_) for x_ in chunks]

        return torch.cat(res, dim=0)
```

- **Split**

  对 $Feature$（$f \in \Re^{B \times N}$）进行切分，得到$f_a \in \Re^{B \times N_a}$、$f_d \in \Re^{B \times N_d}$.

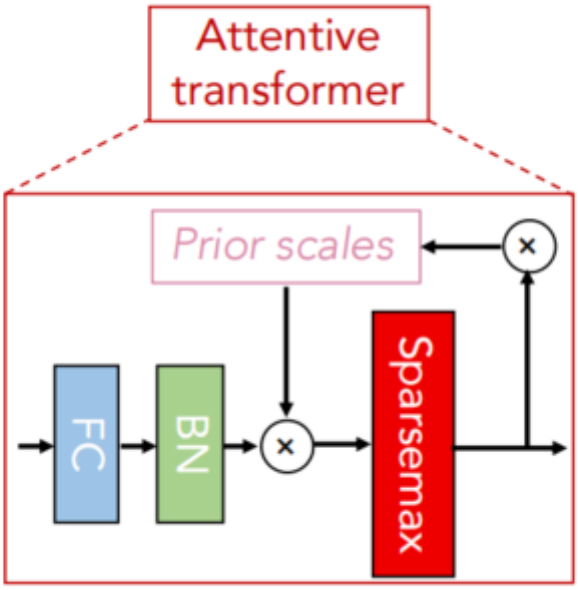
其中：f_d 用于当前决策步的输出，f_a 则作为下一决策步的输入信息。

```
# forword 源码
def forward(self, x, prior=None):
        x = self.initial_bn(x)

        if prior is None:
            prior = torch.ones(x.shape).to(x.device)

        M_loss = 0
        att = self.initial_splitter(x)[:, self.n_d :]

        steps_output = []
        for step in range(self.n_steps):
            M = self.att_transformers[step](prior, att)
            M_loss += torch.mean(
                torch.sum(torch.mul(M, torch.log(M + self.epsilon)), dim=1)
            )
            # update prior
            prior = torch.mul(self.gamma - M, prior)
            # output
            masked_x = torch.mul(M, x)
            out = self.feat_transformers[step](masked_x)
            d = ReLU()(out[:, : self.n_d])
            steps_output.append(d)
            # update attention
            att = out[:, self.n_d :]

        M_loss /= self.n_steps
        return steps_output, M_loss
```

疑问：分割 feature 的时候为什么是取 feature 的前 n_d 列，而不是按照 feature 中不同属性来分割

- **Attentive transformer**[7]

  学习每个样本中每个 $feature$ 的重要程度，得到 $mask$ 矩阵（$M \in \Re^{B \times D}$）



公式如下：

$$M\left[\ i\ \right] = Sparsemax(P\left[\ i-1\ \right] \cdot h_i(a\left[\ i-1\ \right])$$

$$Sparsemax(\boldsymbol{z}) := \underset{\boldsymbol{p}\in\Delta^{K-1}}{\arg\min}\|\boldsymbol{p}-\boldsymbol{z}\|$$

$$P\left[\ i\ \right] = \prod_{j=1}^{i}(\gamma - M\left[\ j\ \right])$$

其中：

$M\left[\ i\ \right]$：表示第 i 个 $step$ 的 $mask$ 矩阵

$Sparsemax$[8]：类比 $softmax$，可以得到更稀疏的输出结果（取值集中于0 或 1附近）

$P\left[\ i-1\ \right]$：是 $Prio\ scales$ 项，用来表示 $features$ 在之前 $step$ 中的使用程度

$a\left[\ i-1\ \right]$：是前一个 $step$ 决策中通过 $split$ 划分来的，即 $f_a \in \Re^{B\times N_a}$

$h_i(\cdot)$：代表 $FC + BN$ 层

γ：是稀疏正则项权重，用来对特征选择阶段的特征稀疏性添加约束，γ是越小，则特征选择越稀疏

```
# 公式源码：
https://github.com/dreamquark-ai/tabnet/blob/develop/pytorch_tabnet/sparsemax.py

# 公式还没有看懂
```

- **Feature attributes**

输出得到每个样本中每个 $feature$ 的重要程度。

将对应决策步 $step$ 进行 $split$ 操作后得到的 $f_d \in \Re^{B \times N_d}$ ，通过 $ReLU$ 函数，进行agg，即按行求和，得到$steps_out \in \Re^{B \times 1}$.

然后与 $mask$ 矩阵相乘，得到维度为 $\Re^{B \times D}$,接着各决策步得到的结果进行求和累加，进行归一化最终得到每个样本中每个特征的重要程度

```
# 源码
def forward(self, x):
        res = 0
        steps_output, M_loss = self.encoder(x)
        res = torch.sum(torch.stack(steps_output, dim=0), dim=0)

        if self.is_multi_task:
            # Result will be in list format
            out = []
            for task_mapping in self.multi_task_mappings:
                out.append(task_mapping(res))
        else:
            out = self.final_mapping(res)
        return out, M_loss
```


1. TabNet: Attentive Interpretable Tabular Learning ↩
2. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift ↩
3. Attention Is All You Need ↩
4. Convolutional Sequence to Sequence Learning ↩
5. Language Modeling with Gated Convolutional Networks ↩
6. Train longer, generalize better: closing the generalization gap in large batch training of neural networks ↩
7. Neural Machine Translation by Jointly Learning to Align and Translate ↩
8. From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification ↩