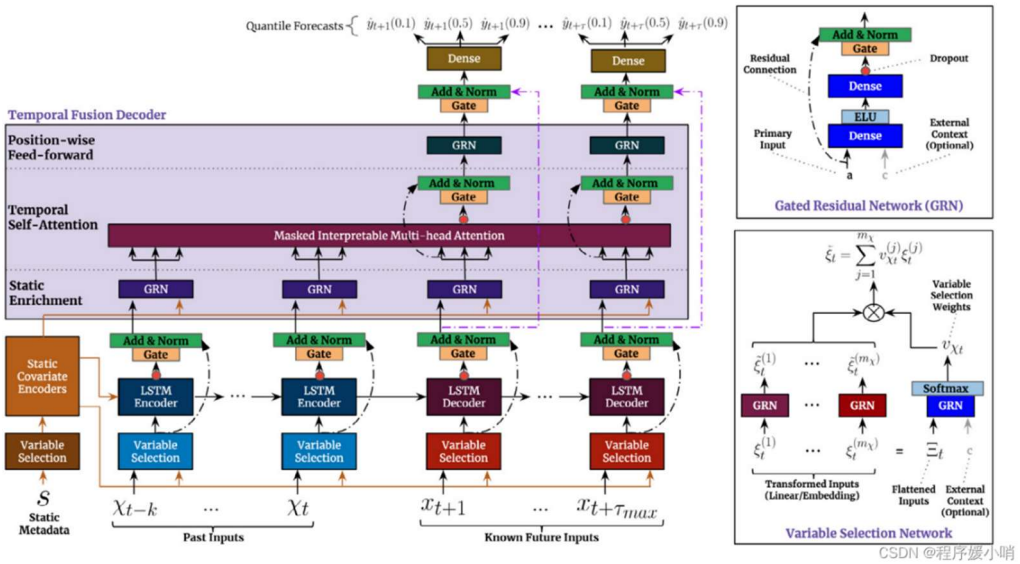
Temporal Fusion Transformer (TFT) 各模块功能和代码解析(pytorch)

文章目录



GLU(Gated Linear Unit)模块

该模块包括一个前向传播函数 forward 和一个参数初始化函数 init_weights。GLU 是一种多层感知器（MLP）结构，由两个子层组成，分别为全连接层和 sigmoid 门。

全连接层有两个，一个作为输入门，另一个作为忘记门。sigmoid 门负责控制两个门的权重，使得在两个门中仅有一个门起作用。

如果激活函数 activation 不是 None，则在输出门之前添加一个激活函数。

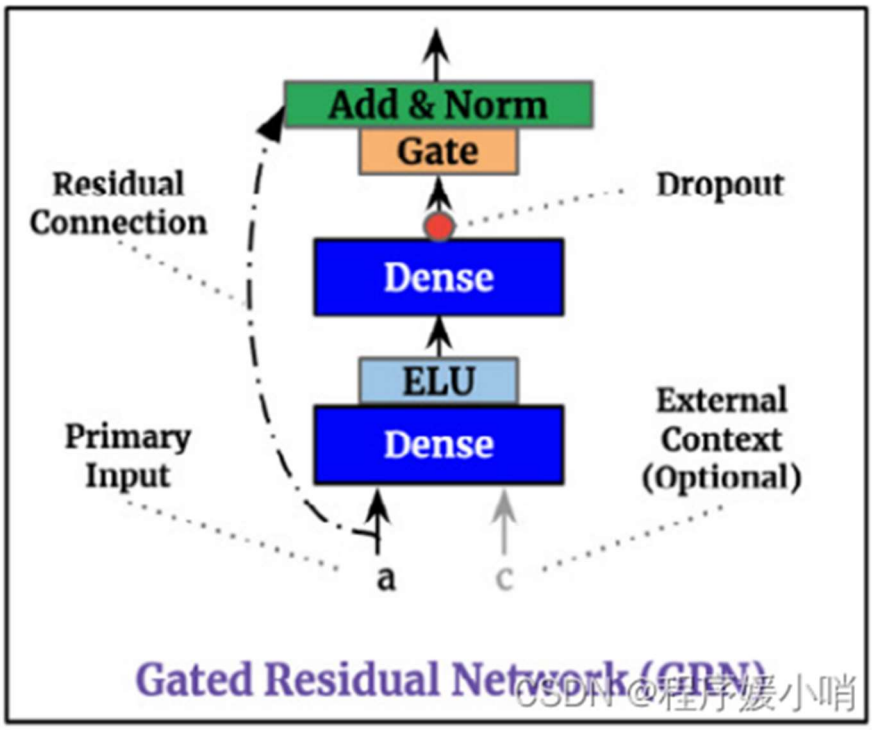
在 init_weights 函数中，使用了 Xavier 均匀分布初始化权重和零初始化偏置。

```
class GatedLinearUnit(nn.Module):
    def __init__(self, input_size,
                 hidden_layer_size,
                 dropout_rate,
                 activation = None):

        super(GatedLinearUnit, self).__init__()

        self.input_size = input_size
        self.hidden_layer_size = hidden_layer_size
        self.dropout_rate = dropout_rate
        self.activation_name = activation

        if self.dropout_rate:
            self.dropout = nn.Dropout(p=self.dropout_rate)

        self.W4 = torch.nn.Linear(self.input_size,
self.hidden_layer_size)
        self.W5 = torch.nn.Linear(self.input_size,
self.hidden_layer_size)

        if self.activation_name:
            self.activation = getattr(nn, self.activation_name)()

        self.sigmoid = nn.Sigmoid()

        self.init_weights()

    def init_weights(self):
        for n, p in self.named_parameters():
            if 'bias' not in n:
                torch.nn.init.xavier_uniform_(p)
#                 torch.nn.init.kaiming_normal_(p, a=0,
mode='fan_in', nonlinearity='sigmoid')
            elif 'bias' in n:
                torch.nn.init.zeros_(p)

    def forward(self, x):

        if self.dropout_rate:
            x = self.dropout(x)

        if self.activation_name:
```

```
            output = self.sigmoid(self.W4(x)) *
self.activation(self.W5(x))
        else:
            output = self.sigmoid(self.W4(x)) * self.W5(x)

        return output
```

GRN(Gated Residual Network)门控残差网络

这段代码实现了一个门控残差网络（Gated Residual Network），它是Temporal Fusion Transformer中的一个组成部分。



该类的初始化方法包含了多个参数：

hidden_layer_size：隐层的维度大小。
input_size：输入向量的维度大小。默认为None，若未指定则使用hidden_layer_size作为输入向量的维度大小。
output_size：输出向量的维度大小。默认为None，若未指定则输出向量的维度大小与隐层的维度大小相同。
dropout_rate：dropout概率。
additional_context：附加的上下文信息。默认为None。
return_gate：是否返回门控向量。默认为False。

在初始化方法中，首先定义了该类所需要用到的线性层，其中包括 self.W1、self.W2 和 self.W3（若有附加的上下文信息）。接着，根据是否指定了输出向量的维度大小，定义了一个门控加和归一化网络（GateAddNormNetwork），其输入向量的维度大小取决于 self.hidden_layer_size 和 self.output_size 的大小关系。最后，对模型中的参数进行了初始化。

在该类的前向传播方法中，如果有附加的上下文信息，则将输入向量和上下文信息分别输入 self.W2 和 self.W3 中，并将它们的输出进行求和，并通过激活函数（这里使用了 ELU）进行激活。接着，将该结果输入 self.W1 中，并将其输出通过门控加和归一化网络进行处理。如果指定了输出向量的维度大小，则将输入向量通过 self.skip_linear 进行线性转换，并将结果与门控加和归一化网络的输出相加后得到最终的输出向量；否则，直接将输入向量与门控加和归一化网络的输出相加后得到最终的输出向量。最后返回输出向量。

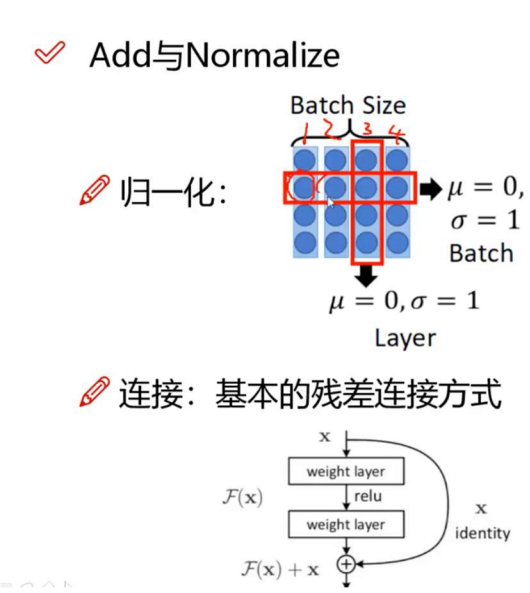
```
class GatedResidualNetwork(nn.Module):
    def __init__(self,
                 hidden_layer_size,
                 input_size = None,
                 output_size = None,
                 dropout_rate = None,
                 additional_context = None,
                 return_gate = False):

        super(GatedResidualNetwork, self).__init__()

        self.hidden_layer_size = hidden_layer_size
        self.input_size = input_size if input_size else
self.hidden_layer_size
        self.output_size = output_size
        self.dropout_rate = dropout_rate
        self.additional_context = additional_context
        self.return_gate = return_gate

        self.W1 = torch.nn.Linear(self.hidden_layer_size,
self.hidden_layer_size)
        self.W2 = torch.nn.Linear(self.input_size,
self.hidden_layer_size)

        if self.additional_context:
            self.W3 = torch.nn.Linear(self.additional_context,
self.hidden_layer_size, bias = False)


        if self.output_size:
```

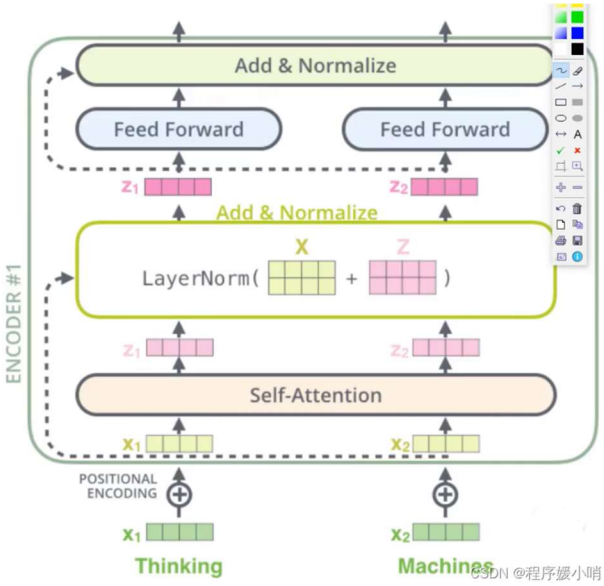
```
            self.skip_linear = torch.nn.Linear(self.input_size,
self.output_size)
            self.glu_add_norm =
GateAddNormNetwork(self.hidden_layer_size, self.output_size,
                                    self.dropout_rate)
        else:
            self.glu_add_norm =
GateAddNormNetwork(self.hidden_layer_size, self.hidden_layer_size,
                                               self.dropout_rate)

        self.init_weights()

    def init_weights(self):
        for name, p in self.named_parameters():
            if ('W2' in name or 'W3' in name) and 'bias' not in name:
                torch.nn.init.kaiming_normal_(p, a=0, mode='fan_in',
nonlinearity='leaky_relu')
            elif ('skip_linear' in name or 'W1' in name) and 'bias'
not in name:
                torch.nn.init.xavier_uniform_(p)
#                 torch.nn.init.kaiming_normal_(p, a=0,
mode='fan_in', nonlinearity='sigmoid')
            elif 'bias' in name:
                torch.nn.init.zeros_(p)

    def forward(self, x):

        if self.additional_context:
            x, context = x
            #x_forward = self.W2(x)
            #context_forward = self.W3(context)
            #print(self.W3(context).shape)
            n2 = F.elu(self.W2(x) + self.W3(context))
        else:
            n2 = F.elu(self.W2(x))

        #print('n2 shape {}'.format(n2.shape))

        n1 = self.W1(n2)

        #print('n1 shape {}'.format(n1.shape))

        if self.output_size:
            output = self.glu_add_norm(n1, self.skip_linear(x))
```

```
        else:
            output = self.glu_add_norm(n1, x)

        #print('output shape {}'.format(output.shape))

        return output
```

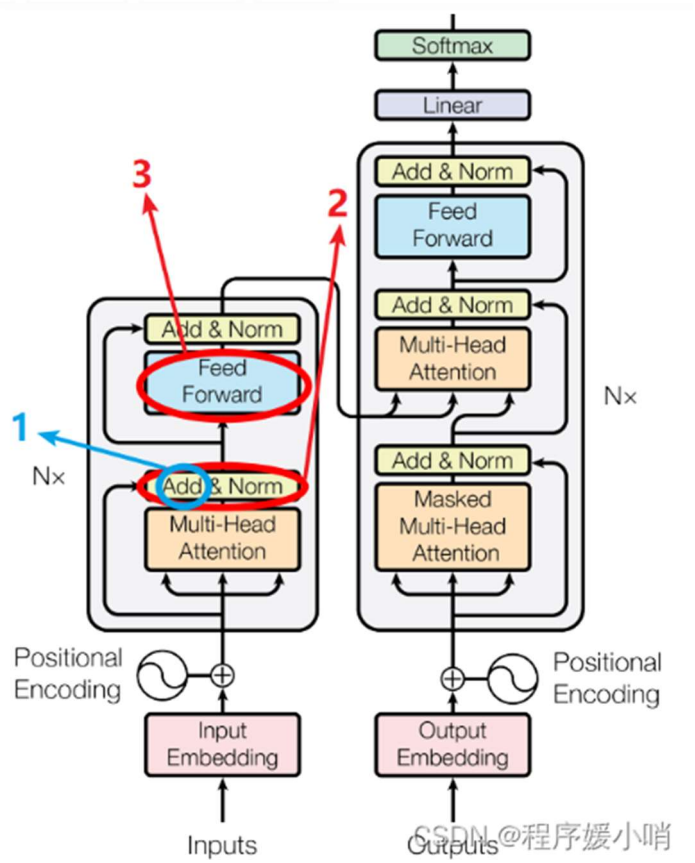
Transformer 经典模块

Add&Normalize 模块

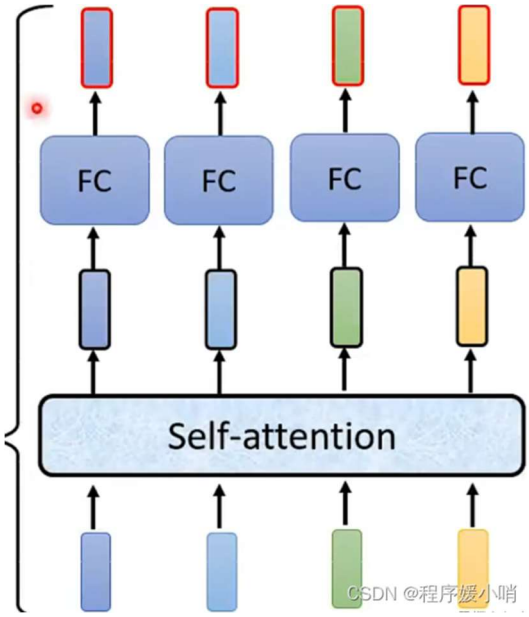




transformer 在进行 self-Attention 之后会进行一个 layerNormalization 【将数据统一到固定区间内】
其中又分为 batchNormalization 和 layerNormalization，
batchNormalization 即按照 batch 维度化成均值为 0 标准差为 1 的数据
Layer 则是纵向将每次的向量数据进行归一化
残差作用：加入未学习的原向量使得到的结果的效果至少不弱于原来的结果

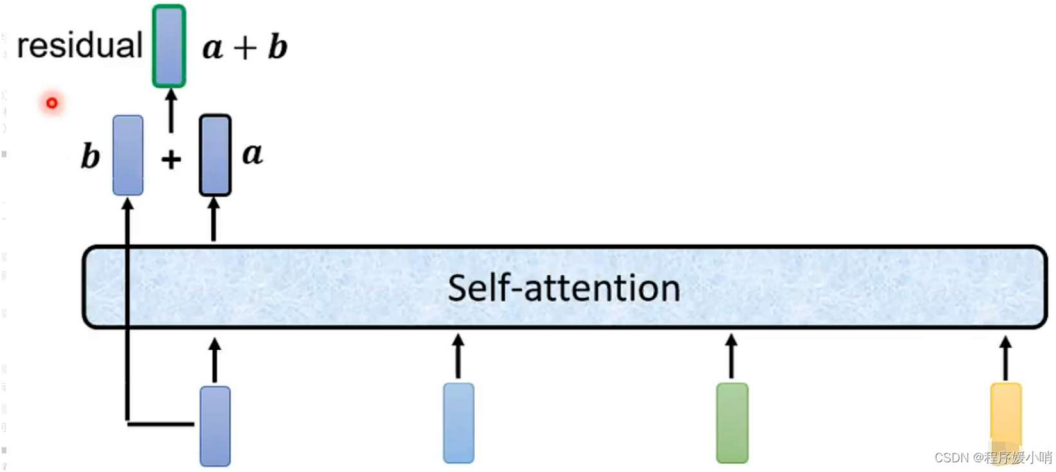


蓝色圆圈标注的“add”是什么呢？好像初始注意力机制（如下图）中并没有 add 呀？？



如图所示，在 self-attention 中求出 $Z(i)=\sum_{j=1}^{T}a_{ij}v^{\left(j \right)}$ Z(i)=∑j=1Taijv(j)后直接将 Z 放入全连接层中，而 transformer 又经过了 add&Norm 步骤 ，这个 add 是什么呢？？？

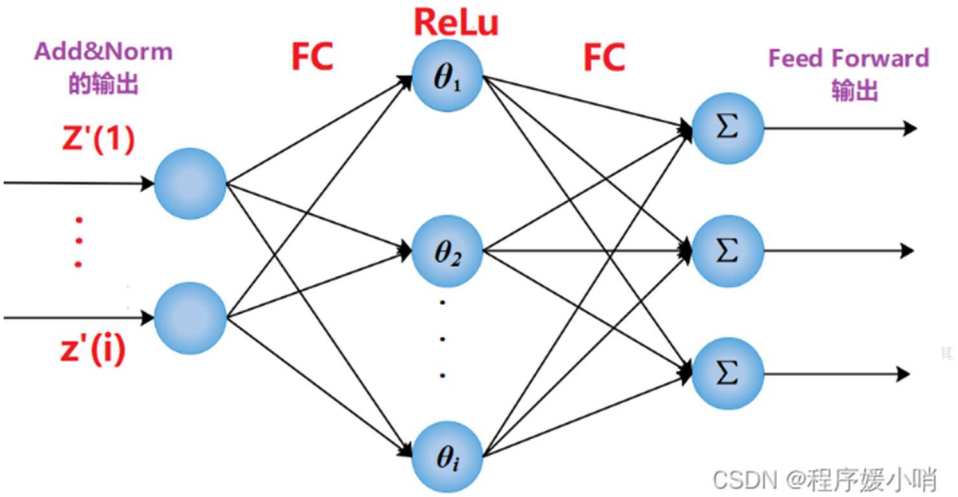
其实 add 就是在 Z 的基础上加了一个残差块 X，加入残差块 X 的目的是为了防止在深度神经网络训练中发生退化问题，退化的意思就是深度神经网络通过增加网络的层数，Loss 逐渐减小，然后趋于稳定达到饱和，然后再继续增加网络层数，Loss 反而增大。这个 Add 具体操作如下：其实就是将 self-attention 的 output+input 作为新的 output，如下图：



关于首图所标记的 2，Add&Norm 这一过程指什么呢？
就是指对新的 output 做标准化，也就是上图的，对 a+b 做标准化

FeedForward：FeedForward 又是什么呢？好像图二（注意力机制）中这里应该是全连接层（FC）呀？**？？？**

先说一下 FeedForward 是什么？其实 FeedForward 是由全连接层（FC）与激活 ReLu 组成的结构，其实和 bp 神经网络结构差不多，输入：Add&Norm 的输出（记作：Z’（i））， FC:全连接层



接下来再说一下为什么要用 FeedForward 呢？不用单纯的 FC 呢？

其实主要还是想提取更深层次的特征，在 Multi-Head Attention 中，主要是进行矩阵乘法，即都是线性变换，而线性变换的学习能力不如非线性变换的学习能力强，我们希望通过引入 ReLu 激活函数，使模型增加非线性成分，强化学习能力。

回到这个模块来，它的输入是 x 和 skip 两个张量，其中 x 是输入张量，skip 是跳跃连接（skip connection），用于保留之前层的信息，防止信息丢失。

在这个模块中，先通过 Gated Linear Unit（GLU）对 x 进行变换，然后将变换后的结果和 skip 进行加和操作，再通过 Layer Normalization 进行归一化，得到输出张量。

GLU 的作用是将 x 变换为与之相同维度的张量，并同时生成一个门控向量。门控向量的每个元素都是介于 0 和 1 之间的实数，用于控制该元素对最终结果的贡献程度。这里的 GLU 实现了类似于 Highway Network 的效果，即能够选择性地保留或抛弃某些信息。

Layer Normalization 的作用是对张量的每个特征维度进行归一化，使得每个特征维度上的均值为 0，方差为 1，从而加快收敛速度、提高泛化能力。

最终，这个门控加和归一化网络将 x 和 skip 合并为一个输出张量，其中 skip 是经过跳跃连接保留下来的信息， x 是经过 GLU 变换后的信息，经过 Layer Normalization 归一化后得到的输出张量。

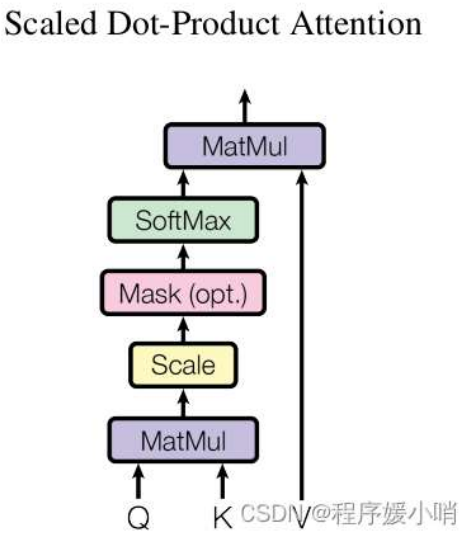
```
class GateAddNormNetwork(nn.Module):
    def __init__(self, input_size,
                 hidden_layer_size,
                 dropout_rate,
                 activation = None):

        super(GateAddNormNetwork, self).__init__()

        self.input_size = input_size
        self.hidden_layer_size = hidden_layer_size
        self.dropout_rate = dropout_rate
        self.activation_name = activation

        self.GLU = GatedLinearUnit(self.input_size,
                                   self.hidden_layer_size,
                                   self.dropout_rate,
                                   activation = self.activation_name)

        self.LayerNorm = nn.LayerNorm(self.hidden_layer_size)
```

```
    def forward(self, x, skip):
        output = self.LayerNorm(self.GLU(x) + skip)
        return output
```

Scaled Dot-Product Attention 模块

在实际应用中，经常会用到 Attention 机制，其中最常用的是 Scaled Dot-Product Attention，它是通过计算 query 和 key 之间的点积 来作为 之间的相似度。

Scaled 指的是 Q 和 K 计算得到的相似度 再经过了一定的量化，具体就是 除以 根号下 K_dim;
Dot-Product 指的是 Q 和 K 之间 通过计算点积作为相似度;
Mask 可选择性 目的是将 padding 的部分 填充负无穷，这样算 softmax 的时候这里就 attention 为 0，从而避免 padding 带来的影响.



在 TFT 里面套用原始 Transformer 架构，实现了一个 Scaled Dot-Product Attention。
该类定义了一个基于 nn.Module 的自定义 PyTorch 模块，包含了初始化函数和前向传递函数。S

caledDotProductAttention 中的主要计算过程是计算 query 和 key 之间的点积，然后进行 softmax 归一化，最后加权求和得到输出。具体地，该类的初始化函数接受两个参数：dropout 和 scale。
dropout 参数指定了在 softmax 之前要应用多少 dropout，
而 scale 参数指定是否应该缩放点积结果。
该类的前向传递函数接受四个参数：q，k，v 和 mask。
其中，q，k 和 v 是查询、键和值的输入向量，mask 是一个可选的屏蔽向量，用于屏蔽输入中的某些位置。

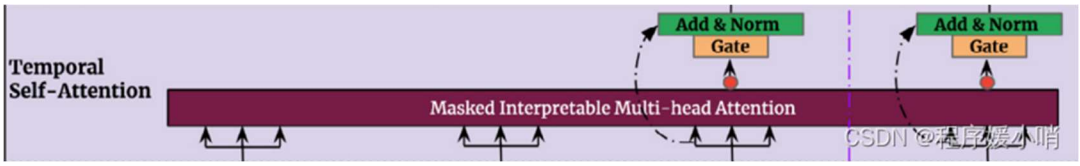
在前向函数中，首先计算 q 和 k 的点积得到注意力矩阵 attn。
如果模块被初始化为缩放模式，则在计算点积时会将结果除以 k 向量长度的平方根，以缩小点积结果的值域，避免梯度消失或梯度爆炸的问题。
如果 mask 不为空，则在 attn 中填充 -1e9，以便在 softmax 操作中产生 0 的概率。
然后，通过 softmax 函数将注意力矩阵归一化为概率分布，并应用 dropout 操作以防止过拟合。最后，通过点积计算得到输出向量 output。

在输出时，函数返回输出向量 output 和注意力矩阵 attn。

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, dropout = 0, scale = True):
        super(ScaledDotProductAttention, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        self.softmax = nn.Softmax(dim = 2)
        self.scale = scale

    def forward(self, q, k, v, mask = None):
        #print('---Inputs----')
        #print('q: {}'.format(q[0]))
        #print('k: {}'.format(k[0]))
        #print('v: {}'.format(v[0]))

        attn = torch.bmm(q, k.permute(0,2,1))
        #print('first bmm')
        #print(attn.shape)
        #print('attn: {}'.format(attn[0]))

        if self.scale:
            dimention = torch.sqrt(torch.tensor(k.shape[-
1]).to(torch.float32))
            attn = attn / dimention
        #    print('attn_scaled: {}'.format(attn[0]))

        if mask is not None:
            #fill = torch.tensor(-1e9).to(DEVICE)
            #zero = torch.tensor(0).to(DEVICE)
            attn = attn.masked_fill(mask == 0, -1e9)
        #    print('attn_masked: {}'.format(attn[0]))

        attn = self.softmax(attn)
        #print('attn_softmax: {}'.format(attn[0]))
        attn = self.dropout(attn)
```

```
        output = torch.bmm(attn, v)

        return output, attn
```

InterpretableMultiHeadAttention 可解释的多头注意力机制



TFT 参照原始 Transformer 架构，实现了一个可解释的多头注意力机制，即 Interpretable Multi-Head Attention。

在初始化函数中，定义了注意力头的个数 n_head、输入特征的维度 d_model 和临时的 q、k、v 的维度 d_k、d_q、d_v。同时，通过 nn.Linear 函数定义了 q、k、v 的线性层和 v 的线性层的组合，还定义了缩放点积注意力的 ScaledDotProductAttention 层以及输出的全连接层 w_h，并初始化了权重。

在前向函数中，首先对输入的 q、k、v 分别通过线性层得到 q_i、k_i、v_i，再利用 ScaledDotProductAttention 层计算出每个注意力头对应的权重矩阵和输出矩阵，并对输出矩阵进行 dropout 操作。然后，将多个头的输出按照最后一个维度进行拼接，再通过均值池化得到最终的输出矩阵，最后通过全连接层和 dropout 操作得到最终的输出和注意力权重矩阵，并将它们返回。

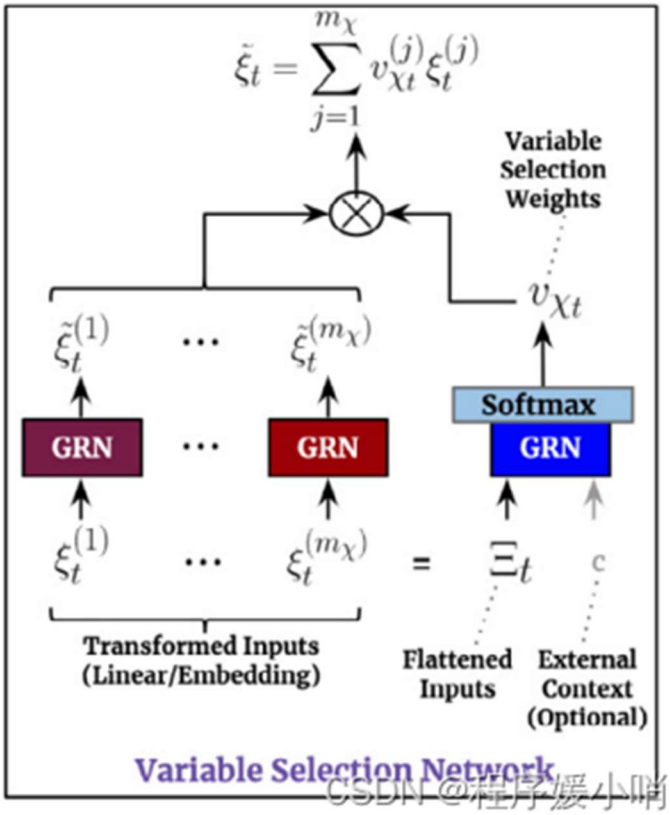
需要注意的是，在计算每个注意力头的权重矩阵时，利用了缩放点积注意力。该注意力的计算过程包括三个步骤：1）计算 q 和 k 的点积，即 attn = torch.bmm(q, k.permute(0,2,1))；2）如果需要进行缩放，则除以根号下 d_k；3）将得到的结果进行 softmax 操作得到注意力矩阵。同时，在计算注意力矩阵时，还可以通过 mask 参数实现对无效位置的屏蔽。

```
class InterpretableMultiHeadAttention(nn.Module):
    def __init__(self, n_head, d_model, dropout):
        super(InterpretableMultiHeadAttention, self).__init__()

        self.n_head = n_head
        self.d_model = d_model
        self.d_k = self.d_q = self.d_v = d_model // n_head
        self.dropout = nn.Dropout(p=dropout)

        self.v_layer = nn.Linear(self.d_model, self.d_v, bias =
False)
```

```
        self.q_layers = nn.ModuleList([nn.Linear(self.d_model,
self.d_q, bias = False)
                                       for _ in range(self.n_head)])
        self.k_layers = nn.ModuleList([nn.Linear(self.d_model,
self.d_k, bias = False)
                                       for _ in range(self.n_head)])
        self.v_layers = nn.ModuleList([self.v_layer for _ in
range(self.n_head)])
        self.attention = ScaledDotProductAttention()
        self.w_h = nn.Linear(self.d_v, self.d_model, bias = False)

        self.init_weights()

    def init_weights(self):
        for name, p in self.named_parameters():
            if 'bias' not in name:
                torch.nn.init.xavier_uniform_(p)
#                 torch.nn.init.kaiming_normal_(p, a=0,
mode='fan_in', nonlinearity='sigmoid')
            else:
                torch.nn.init.zeros_(p)

    def forward(self, q, k, v, mask = None):

        heads = []
        attns = []
        for i in range(self.n_head):
            qs = self.q_layers[i](q)
            ks = self.k_layers[i](k)
            vs = self.v_layers[i](v)
            #print('qs layer: {}'.format(qs.shape))
            head, attn = self.attention(qs, ks, vs, mask)
            #print('head layer: {}'.format(head.shape))
            #print('attn layer: {}'.format(attn.shape))
            head_dropout = self.dropout(head)
            heads.append(head_dropout)
            attns.append(attn)

        head = torch.stack(heads, dim = 2) if self.n_head > 1 else
heads[0]
        #print('concat heads: {}'.format(head.shape))
        #print('heads {}: {}'.format(0, head[0,0,Ellipsis]))
        attn = torch.stack(attns, dim = 2)
        #print('concat attn: {}'.format(attn.shape))
```

```
        outputs = torch.mean(head, dim = 2) if self.n_head > 1 else
head
        #print('outputs mean: {}'.format(outputs.shape))
        #print('outputs mean {}: {}'.format(0,
outputs[0,0,Ellipsis]))
        outputs = self.w_h(outputs)
        outputs = self.dropout(outputs)

        return outputs, attn
```

Variable Selection Network 变量选择网络



这段代码是 Temporal Fusion Transformer 中 VariableSelectionNetwork 的前向传递方法。

VariableSelectionNetwork 的目的是选择哪些变量对于当前时间步骤的预测是最重要的。这里有两种情况：使用静态上下文（只有输入嵌入）和使用非静态上下文（输入嵌入和静态上下文）。

在这个方法中，如果使用了非静态上下文，输入会分成嵌入和静态上下文两个部分。

首先，使用 GatedResidualNetwork 对嵌入进行非线性转换，
然后使用 Softmax 函数将其转换为稀疏权重。
接着，对于每个要预测的特征，使用 GatedResidualNetwork 对嵌入进行非线性转换，将其与稀疏权重相乘，以生成一组经过转换的特征，最后将这些特征相加以生成最终的时间上下文。

如果只有静态上下文，则直接对输入嵌入进行非线性转换，将其与稀疏权重相乘，以生成一组转换后的特征，最后将这些特征相加以生成最终的时间上下文。
在这两种情况下，该方法返回时间上下文和稀疏权重。

```
class VariableSelectionNetwork(nn.Module):
    def __init__(self, hidden_layer_size,
                 dropout_rate,
                 output_size,
                 input_size = None,
                 additional_context = None):
        super(VariableSelectionNetwork, self).__init__()

        self.hidden_layer_size = hidden_layer_size
        self.input_size = input_size
        self.output_size = output_size
        self.dropout_rate = dropout_rate
        self.additional_context = additional_context

        self.flattened_grn =
GatedResidualNetwork(self.hidden_layer_size, input_size =
self.input_size, output_size = self.output_size, dropout_rate =
self.dropout_rate, additional_context=self.additional_context)

        self.per_feature_grn =
nn.ModuleList([GatedResidualNetwork(self.hidden_layer_size,
dropout_rate=self.dropout_rate)  for i in range(self.output_size)])
    def forward(self, x):
        # Non Static Inputs
        if self.additional_context:
            embedding, static_context = x
            #print('static_context')
            #print(static_context.shape)

            time_steps = embedding.shape[1]
            flatten = embedding.view(-1, time_steps,
self.hidden_layer_size * self.output_size)
            #print('flatten')
```

```
            #print(flatten.shape)

            static_context = static_context.unsqueeze(1)
            #print('static_context')
            #print(static_context.shape)

            # Nonlinear transformation with gated residual network.
            mlp_outputs = self.flattened_grn((flatten,
static_context))
            #print('mlp_outputs')
            #print(mlp_outputs.shape)

            sparse_weights = F.softmax(mlp_outputs, dim = -1)
            sparse_weights = sparse_weights.unsqueeze(2)
            #print('sparse_weights')
            #print(sparse_weights.shape)

            trans_emb_list = []
            for i in range(self.output_size):
                e = self.per_feature_grn[i](embedding[Ellipsis, i])
                trans_emb_list.append(e)
            transformed_embedding = torch.stack(trans_emb_list,
axis=-1)
            #print('transformed_embedding')
            #print(transformed_embedding.shape)

            combined = sparse_weights * transformed_embedding
            #print('combined')
            #print(combined.shape)

            temporal_ctx = torch.sum(combined, dim = -1)
            #print('temporal_ctx')
            #print(temporal_ctx.shape)

        # Static Inputs
        else:
            embedding = x
            #print('embedding')
            #print(embedding.shape)

            flatten = torch.flatten(embedding, start_dim=1)
            #flatten = embedding.view(batch_size, -1)
            #print('flatten')
            #print(flatten.shape)
```

```
            # Nonlinear transformation with gated residual network.
            mlp_outputs = self.flattened_grn(flatten)
            #print('mlp_outputs')
            #print(mlp_outputs.shape)

            sparse_weights = F.softmax(mlp_outputs, dim = -1)
            sparse_weights = sparse_weights.unsqueeze(-1)
#             print('sparse_weights')
#             print(sparse_weights.shape)

            trans_emb_list = []
            for i in range(self.output_size):
                #print('embedding for the per feature static grn')
                #print(embedding[:, i:i + 1, :].shape)
                e = self.per_feature_grn[i](embedding[:, i:i + 1, :])
                trans_emb_list.append(e)
            transformed_embedding = torch.cat(trans_emb_list, axis=1)
#             print('transformed_embedding')
#             print(transformed_embedding.shape)

            combined = sparse_weights * transformed_embedding
#             print('combined')
#             print(combined.shape)

            temporal_ctx = torch.sum(combined, dim = 1)
#             print('temporal_ctx')
#             print(temporal_ctx.shape)

        return temporal_ctx, sparse_weights
```

QuantileLoss 分位数损失函数

$$\mathcal{L}(\Omega, \boldsymbol{W}) = \sum_{y_t \in \Omega} \sum_{q \in \mathcal{Q}} \sum_{\tau=1}^{\tau_{max}} \frac{QL\left(y_t,\ \hat{y}(q, t-\tau, \tau),\ q\right)}{M\tau_{max}} \qquad (24)$$

$$QL(y,\ \hat{y},\ q) = q(y-\hat{y})_+ + (1-q)(\hat{y}-y)_+, \qquad (25)$$

这段代码定义了两个类，QuantileLossCalculator 和 NormalizedQuantileLossCalculator，用于计算量化损失（quantile loss）和标准化量化损失（normalized quantile loss）。

QuantileLossCalculator 类中的 **init** 方法初始化了计算器的属性 quantiles 和 output_size。quantiles 是一个浮点数列表，代表要计算的分位数，例如 [0.1, 0.5, 0.9] 表示要计算 10%、50% 和 90% 的分位数。output_size 是输出序列的长度。

QuantileLossCalculator 类中的 quantile_loss 方法计算单个分位数的量化损失。该方法使用了标准的量化损失公式，计算方法可以参考 Temporal Fusion Transformer 论文中的 “Training Procedure” 章节。该方法接受三个参数：y 是目标序列（也称为标签），y_pred 是预测序列，quantile 是分位数。该方法首先检查分位数是否在 0 和 1 之间，然后计算预测序列与目标序列之间的差值，并根据分位数计算加权误差。最后，该方法返回一个张量，表示量化损失。

QuantileLossCalculator 类中的 apply 方法计算所有指定的分位数的量化损失之和。该方法接受两个参数：a 是目标序列，b 是预测序列。该方法首先创建一个空列表 loss，然后对于每个指定的分位数，使用 quantile_loss 方法计算量化损失，并将结果添加到 loss 列表中。最后，该方法使用 torch.mean 计算 loss 列表中所有张量的平均值，表示所有指定的分位数的量化损失之和。返回的值是一个标量张量，表示总的量化损失。

NormalizedQuantileLossCalculator 类中的 **init** 方法和 apply 方法与 QuantileLossCalculator 类相同，不同之处在于 NormalizedQuantileLossCalculator 类计算的是标准化量化损失。标准化量化损失的计算方法与量化损失的计算方法类似，但是在计算加权误差时，使用的权重是相对于整个序列绝对误差的比例。最后，标准化量化损失通过总的标准化误差进行归一化，使得它们可以与其他模型的损失进行比较。返回的值是一个标量张量，表示标准化量化损失。

```
import torch

class QuantileLossCalculator():
    """Computes the combined quantile loss for prespecified
quantiles.
    Attributes:
      quantiles: Quantiles to compute losses
    """

    def __init__(self, quantiles, output_size):
        """Initializes computer with quantiles for loss calculations.
            Args:
            quantiles: Quantiles to use for computations.
        """
        self.quantiles = quantiles
        self.output_size = output_size
```

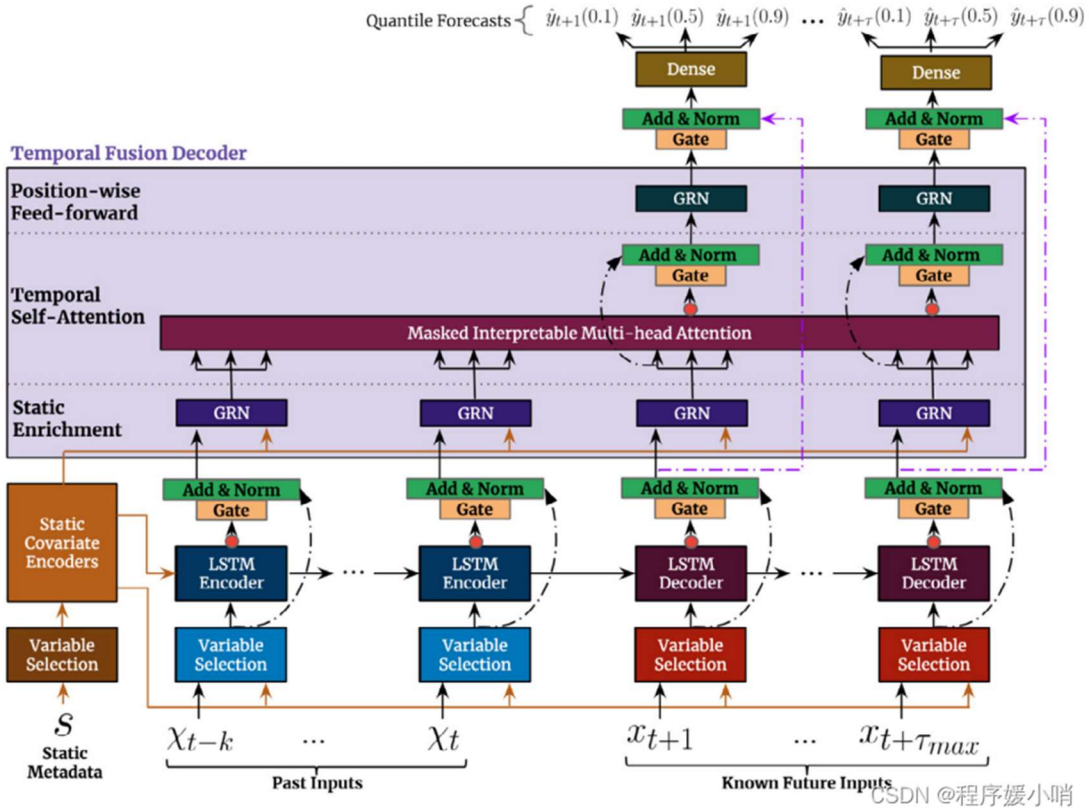
```
    # Loss functions.
    def quantile_loss(self, y, y_pred, quantile):
        """ Computes quantile loss for pytorch.
            Standard quantile loss as defined in the "Training
Procedure" section of
            the main TFT paper
            Args:
            y: Targets
            y_pred: Predictions
            quantile: Quantile to use for loss calculations (between
0 & 1)
            Returns:
            Tensor for quantile loss.
        """

        # Checks quantile
        if quantile < 0 or quantile > 1:
            raise ValueError(
                'Illegal quantile value={}! Values should be between
0 and 1.'.format(quantile))

        prediction_underflow = y - y_pred
#         print('prediction_underflow')
#         print(prediction_underflow.shape)
        q_loss = quantile * torch.max(prediction_underflow,
torch.zeros_like(prediction_underflow)) + \
                 (1. - quantile) * torch.max(-prediction_underflow,
torch.zeros_like(prediction_underflow))

#         print('q_loss')
#         print(q_loss.shape)

#         loss = torch.mean(q_loss, dim = 1)
#         print('loss')
#         print(loss.shape)
#         return loss

#         return torch.sum(q_loss, dim = -1)
        return q_loss.unsqueeze(1)

    def apply(self, b, a):
        """Returns quantile loss for specified quantiles.
            Args:
            a: Targets
```

```
            b: Predictions
        """
        quantiles_used = set(self.quantiles)

        loss = []
#         loss = 0.
        for i, quantile in enumerate(self.quantiles):
            if quantile in quantiles_used:
                #print(a[Ellipsis, self.output_size *
i:self.output_size * (i + 1)].shape)
#                 loss += self.quantile_loss(a[Ellipsis,
self.output_size * i:self.output_size * (i + 1)],
#                                            b[Ellipsis,
self.output_size * i:self.output_size * (i + 1)],
#                                            quantile)
                #print(a[Ellipsis, self.output_size * i].shape)
                #loss += self.quantile_loss(a[Ellipsis,
self.output_size * i],
                #                           b[Ellipsis,
self.output_size * i],
                #                           quantile)

#                 loss.append(self.quantile_loss(a[Ellipsis,
self.output_size * i:self.output_size * (i + 1)],
#                                                b[Ellipsis,
self.output_size * i:self.output_size * (i + 1)],
#                                                quantile))

                loss.append(self.quantile_loss(a[Ellipsis, i],
                                               b[Ellipsis, i],
                                               quantile))

#         loss_computed = torch.cat(loss, axis = -1)
#         loss_computed = torch.sum(loss_computed, axis = -1)
#         loss_computed = torch.sum(loss_computed, axis = 0)

        loss_computed = torch.mean(torch.sum(torch.cat(loss, axis =
1), axis = 1))

        return loss_computed
#         return loss

class NormalizedQuantileLossCalculator():
```

```
    """Computes the combined quantile loss for prespecified
quantiles.
    Attributes:
      quantiles: Quantiles to compute losses
    """

    def __init__(self, quantiles, output_size):
        """Initializes computer with quantiles for loss calculations.
            Args:
            quantiles: Quantiles to use for computations.
        """
        self.quantiles = quantiles
        self.output_size = output_size

    # Loss functions.
    def apply(self, y, y_pred, quantile):
        """ Computes quantile loss for pytorch.
            Standard quantile loss as defined in the "Training
Procedure" section of
            the main TFT paper
            Args:
            y: Targets
            y_pred: Predictions
            quantile: Quantile to use for loss calculations (between
0 & 1)
            Returns:
            Tensor for quantile loss.
        """

        # Checks quantile
        if quantile < 0 or quantile > 1:
            raise ValueError(
                'Illegal quantile value={}! Values should be between
0 and 1.'.format(quantile))

        prediction_underflow = y - y_pred
#         print('prediction_underflow')
#         print(prediction_underflow.shape)
        weighted_errors = quantile * torch.max(prediction_underflow,
torch.zeros_like(prediction_underflow)) + \
                (1. - quantile) * torch.max(-prediction_underflow,
torch.zeros_like(prediction_underflow))

        quantile_loss = torch.mean(weighted_errors)
```

```
        normaliser = torch.mean(torch.abs(quantile_loss))
        return 2 * quantile_loss / normaliser
```

Temporal Fusion Transformer 模型构建



利用上述组件构建模型

```
class TemporalFusionTransformer(pl.LightningModule):
    def __init__(self, hparams):
        super(TemporalFusionTransformer, self).__init__()

        self.hparams = hparams

        self.name = self.__class__.__name__

        # Data parameters
        self.time_steps =
int(hparams.total_time_steps)#int(params['total_time_steps'])
        self.input_size =
int(hparams.input_size)#int(params['input_size'])
        self.output_size =
int(hparams.output_size)#int(params['output_size'])
```

```
        self.category_counts =
json.loads(str(hparams.category_counts))#json.loads(str(params['categ
ory_counts']))
        self.num_categorical_variables = len(self.category_counts)
        self.num_regular_variables = self.input_size -
self.num_categorical_variables
        self.n_multiprocessing_workers =
int(hparams.multiprocessing_workers)
#int(params['multiprocessing_workers'])

        # Relevant indices for TFT
        self._input_obs_loc =
json.loads(str(hparams.input_obs_loc))#json.loads(str(params['input_o
bs_loc']))
        self._static_input_loc =
json.loads(str(hparams.static_input_loc))#json.loads(str(params['stat
ic_input_loc']))
        self._known_regular_input_idx =
json.loads(str(hparams.known_regular_inputs))#json.loads(str(params['
known_regular_inputs']))
        self._known_categorical_input_idx =
json.loads(str(hparams.known_categorical_inputs))#json.loads(str(para
ms['known_categorical_inputs']))

        self.num_non_static_historical_inputs =
self.get_historical_num_inputs()
        self.num_non_static_future_inputs =
self.get_future_num_inputs()

        self.column_definition = [
                                  ('id', DataTypes.REAL_VALUED,
InputTypes.ID),
                                  ('hours_from_start',
DataTypes.REAL_VALUED, InputTypes.TIME),
                                  ('power_usage',
DataTypes.REAL_VALUED, InputTypes.TARGET),
                                  ('hour', DataTypes.REAL_VALUED,
InputTypes.KNOWN_INPUT),
                                  ('day_of_week',
DataTypes.REAL_VALUED, InputTypes.KNOWN_INPUT),
                                  ('hours_from_start',
DataTypes.REAL_VALUED, InputTypes.KNOWN_INPUT),
                                  ('categorical_id',
DataTypes.CATEGORICAL, InputTypes.STATIC_INPUT),
```

```
                                    ]

        # Network params
        self.quantiles = [0.1, 0.5, 0.9]
#         self.use_cudnn = use_cudnn  # Whether to use GPU optimised
LSTM
        self.hidden_layer_size =
int(hparams.hidden_layer_size)#int(params['hidden_layer_size'])
        self.dropout_rate =
float(hparams.dropout_rate)#float(params['dropout_rate'])
        self.max_gradient_norm =
float(hparams.max_gradient_norm)#float(params['max_gradient_norm'])
        self.learning_rate =
float(hparams.learning_rate)#float(params['learning_rate'])
        self.minibatch_size =
int(hparams.minibatch_size)#int(params['minibatch_size'])
        self.num_epochs =
int(hparams.num_epochs)#int(params['num_epochs'])
        self.early_stopping_patience =
int(hparams.early_stopping_patience)#int(params['early_stopping_patie
nce'])

        self.num_encoder_steps =
int(hparams.num_encoder_steps)#int(params['num_encoder_steps'])
        self.num_stacks =
int(hparams.stack_size)#int(params['stack_size'])
        self.num_heads =
int(hparams.num_heads)#int(params['num_heads'])

        # Serialisation options
#         self._temp_folder = os.path.join(params['model_folder'],
'tmp')
#         self.reset_temp_folder()

        # Extra components to store Tensorflow nodes for attention
computations
        self._input_placeholder = None
        self._attention_components = None
        self._prediction_parts = None

        print('*** {} params ***'.format(self.name))
        for k in vars(hparams):
            print('# {} = {}'.format(k, vars(hparams)[k]))
```

```
        self.train_criterion = QuantileLossCalculator(self.quantiles,
self.output_size)
        self.test_criterion =
NormalizedQuantileLossCalculator(self.quantiles, self.output_size)

        # Build model
        ## Build embeddings
        self.build_embeddings()

        ## Build Static Contex Networks
        self.build_static_context_networks()

        ## Building Variable Selection Networks
        self.build_variable_selection_networks()

        ## Build Lstm
        self.build_lstm()

        ## Build GLU for after lstm encoder decoder and layernorm
        self.build_post_lstm_gate_add_norm()

        ## Build Static Enrichment Layer
        self.build_static_enrichment()

        ## Building decoder multihead attention
        self.build_temporal_self_attention()

        ## Building positionwise decoder
        self.build_position_wise_feed_forward()

        ## Build output feed forward
        self.build_output_feed_forward()

        ## Initializing remaining weights
        self.init_weights()

    def init_weights(self):
        for name, p in self.named_parameters():
            if ('lstm' in name and 'ih' in name) and 'bias' not in
name:
                #print(name)
                #print(p.shape)
                torch.nn.init.xavier_uniform_(p)
```

```
#                 torch.nn.init.kaiming_normal_(p, a=0,
mode='fan_in', nonlinearity='sigmoid')
            elif ('lstm' in name and 'hh' in name) and 'bias' not in
name:

                torch.nn.init.orthogonal_(p)

            elif 'lstm' in name and 'bias' in name:
                #print(name)
                #print(p.shape)
                torch.nn.init.zeros_(p)

    def get_historical_num_inputs(self):

        obs_inputs = [i for i in self._input_obs_loc]

        known_regular_inputs = [i for i in
self._known_regular_input_idx
                                if i not in self._static_input_loc]

        known_categorical_inputs = [i for i in
self._known_categorical_input_idx
                                    if i + self.num_regular_variables
not in self._static_input_loc]

        wired_embeddings = [i for i in
range(self.num_categorical_variables)
                            if i not in
self._known_categorical_input_idx
                            and i not in self._input_obs_loc]

        unknown_inputs = [i for i in
range(self.num_regular_variables)
                          if i not in self._known_regular_input_idx
                          and i not in self._input_obs_loc]


        return
len(obs_inputs+known_regular_inputs+known_categorical_inputs+wired_em
beddings+unknown_inputs)

    def get_future_num_inputs(self):

        known_regular_inputs = [i for i in
self._known_regular_input_idx
```

```
                                    if i not in self._static_input_loc]

        known_categorical_inputs = [i for i in
self._known_categorical_input_idx
                                    if i + self.num_regular_variables
not in self._static_input_loc]

        return len(known_regular_inputs + known_categorical_inputs)

    def build_embeddings(self):
        self.categorical_var_embeddings =
nn.ModuleList([nn.Embedding(self.category_counts[i],

self.hidden_layer_size)
                                                    for i in
range(self.num_categorical_variables)])

        self.regular_var_embeddings = nn.ModuleList([nn.Linear(1,

self.hidden_layer_size)
                                                   for i in
range(self.num_regular_variables)])

    def build_variable_selection_networks(self):

        self.static_vsn = VariableSelectionNetwork(hidden_layer_size
= self.hidden_layer_size,
                                                   input_size =
self.hidden_layer_size * len(self._static_input_loc),
                                                   output_size =
len(self._static_input_loc),
                                                   dropout_rate =
self.dropout_rate)

        self.temporal_historical_vsn =
VariableSelectionNetwork(hidden_layer_size = self.hidden_layer_size,

input_size = self.hidden_layer_size *

self.num_non_static_historical_inputs,

output_size = self.num_non_static_historical_inputs,

dropout_rate = self.dropout_rate,
```

```
additional_context=self.hidden_layer_size)

        self.temporal_future_vsn =
VariableSelectionNetwork(hidden_layer_size = self.hidden_layer_size,

input_size = self.hidden_layer_size *

self.num_non_static_future_inputs,

output_size = self.num_non_static_future_inputs,

dropout_rate = self.dropout_rate,

additional_context=self.hidden_layer_size)

    def build_static_context_networks(self):

        self.static_context_variable_selection_grn =
GatedResidualNetwork(self.hidden_layer_size,

dropout_rate=self.dropout_rate)

        self.static_context_enrichment_grn =
GatedResidualNetwork(self.hidden_layer_size,

dropout_rate=self.dropout_rate)

        self.static_context_state_h_grn =
GatedResidualNetwork(self.hidden_layer_size,

dropout_rate=self.dropout_rate)

        self.static_context_state_c_grn =
GatedResidualNetwork(self.hidden_layer_size,

dropout_rate=self.dropout_rate)

    def build_lstm(self):
        self.historical_lstm = nn.LSTM(input_size =
self.hidden_layer_size,
                                       hidden_size =
self.hidden_layer_size,
                                       batch_first = True)
```

```
        self.future_lstm = nn.LSTM(input_size =
self.hidden_layer_size,
                                   hidden_size =
self.hidden_layer_size,
                                   batch_first = True)

    def build_post_lstm_gate_add_norm(self):
        self.post_seq_encoder_gate_add_norm =
GateAddNormNetwork(self.hidden_layer_size,

self.hidden_layer_size,

self.dropout_rate,

activation = None)

    def build_static_enrichment(self):
        self.static_enrichment =
GatedResidualNetwork(self.hidden_layer_size,
                                                    dropout_rate =
self.dropout_rate,

additional_context=self.hidden_layer_size)

    def build_temporal_self_attention(self):
        self.self_attn_layer = InterpretableMultiHeadAttention(n_head
= self.num_heads,

d_model = self.hidden_layer_size,

dropout = self.dropout_rate)

        self.post_attn_gate_add_norm =
GateAddNormNetwork(self.hidden_layer_size,

self.hidden_layer_size,

self.dropout_rate,
                                                        activation
= None)

    def build_position_wise_feed_forward(self):
        self.GRN_positionwise =
GatedResidualNetwork(self.hidden_layer_size,
```

```
                                                        dropout_rate =
self.dropout_rate)

        self.post_tfd_gate_add_norm =
GateAddNormNetwork(self.hidden_layer_size,

self.hidden_layer_size,

self.dropout_rate,
                                                            activation =
None)

    def build_output_feed_forward(self):
        self.output_feed_forward =
torch.nn.Linear(self.hidden_layer_size,
                                                    self.output_size *
len(self.quantiles))

    def get_decoder_mask(self, self_attn_inputs):
        """Returns causal mask to apply for self-attention layer.
        Args:
        self_attn_inputs: Inputs to self attention layer to determine
mask shape
        """
        len_s = self_attn_inputs.shape[1]
        bs = self_attn_inputs.shape[0]
        mask = torch.cumsum(torch.eye(len_s), 0)
        mask = mask.repeat(bs,1,1).to(torch.float32)

        return mask.to(DEVICE)

    def get_tft_embeddings(self, regular_inputs, categorical_inputs):
        # Static input
        if self._static_input_loc:
            static_regular_inputs =
[self.regular_var_embeddings[i](regular_inputs[:, 0, i:i + 1])
                                    for i in
range(self.num_regular_variables)
                                    if i in self._static_input_loc]
            #print('static_regular_inputs')
            #print([print(emb.shape) for emb in
static_regular_inputs])
```

```
            static_categorical_inputs =
[self.categorical_var_embeddings[i](categorical_inputs[Ellipsis,
i])[:,0,:]
                                        for i in
range(self.num_categorical_variables)
                                        if i +
self.num_regular_variables in self._static_input_loc]
            #print('static_categorical_inputs')
            #print([print(emb.shape) for emb in
static_categorical_inputs])
            static_inputs = torch.stack(static_regular_inputs +
static_categorical_inputs, axis = 1)
        else:
            static_inputs = None

        # Target input
        obs_inputs =
torch.stack([self.regular_var_embeddings[i](regular_inputs[Ellipsis,
i:i + 1])
                                    for i in self._input_obs_loc],
axis=-1)

        # Observed (a prioir unknown) inputs
        wired_embeddings = []
        for i in range(self.num_categorical_variables):
            if i not in self._known_categorical_input_idx \
            and i not in self._input_obs_loc:
                e =
self.categorical_var_embeddings[i](categorical_inputs[:, :, i])
                wired_embeddings.append(e)

        unknown_inputs = []
        for i in range(self.num_regular_variables):
            if i not in self._known_regular_input_idx \
            and i not in self._input_obs_loc:
                e =
self.regular_var_embeddings[i](regular_inputs[Ellipsis, i:i + 1])
                unknown_inputs.append(e)

        if unknown_inputs + wired_embeddings:
            unknown_inputs = torch.stack(unknown_inputs +
wired_embeddings, axis=-1)
        else:
            unknown_inputs = None
```

```
        # A priori known inputs
        known_regular_inputs =
[self.regular_var_embeddings[i](regular_inputs[Ellipsis, i:i + 1])
                                for i in
self._known_regular_input_idx
                                if i not in self._static_input_loc]
        #print('known_regular_inputs')
        #print([print(emb.shape) for emb in known_regular_inputs])

        known_categorical_inputs =
[self.categorical_var_embeddings[i](categorical_inputs[Ellipsis, i])
                                    for i in
self._known_categorical_input_idx
                                    if i + self.num_regular_variables
not in self._static_input_loc]
       #print('known_categorical_inputs')
       #print([print(emb.shape) for emb in known_categorical_inputs])

        known_combined_layer = torch.stack(known_regular_inputs +
known_categorical_inputs, axis=-1)

        return unknown_inputs, known_combined_layer, obs_inputs,
static_inputs

    def forward(self, all_inputs):

        regular_inputs =
all_inputs[:, :, :self.num_regular_variables].to(torch.float)
        #print('regular_inputs')
        #print(regular_inputs.shape)
        categorical_inputs = all_inputs[:, :,
self.num_regular_variables:].to(torch.long)
        #print('categorical_inputs')
        #print(categorical_inputs.shape)

        unknown_inputs, known_combined_layer, obs_inputs,
static_inputs \
            = self.get_tft_embeddings(regular_inputs,
categorical_inputs)

        # Isolate known and observed historical inputs.
        if unknown_inputs is not None:
              historical_inputs = torch.cat([
```

```
                    unknown_inputs[:, :self.num_encoder_steps, :],

known_combined_layer[:, :self.num_encoder_steps, :],
                    obs_inputs[:, :self.num_encoder_steps, :]
                ], axis=-1)
            else:
                historical_inputs = torch.cat([

known_combined_layer[:, :self.num_encoder_steps, :],
                    obs_inputs[:, :self.num_encoder_steps, :]
                ], axis=-1)

            #print('historical_inputs')
            #print(historical_inputs.shape)

            # Isolate only known future inputs.
            future_inputs = known_combined_layer[:,
self.num_encoder_steps:, :]
            #print('future_inputs')
            #print(future_inputs.shape)

            #print('static_inputs')
            #print(static_inputs.shape)

            static_encoder, sparse_weights =
self.static_vsn(static_inputs)

            #print('static_encoder')
            #print(static_encoder.shape)

            #print('sparse_weights')
            #print(sparse_weights.shape)

            static_context_variable_selection =
self.static_context_variable_selection_grn(static_encoder)
            #print('static_context_variable_selection')
            #print(static_context_variable_selection.shape)
            static_context_enrichment =
self.static_context_enrichment_grn(static_encoder)
            #print('static_context_enrichment')
            #print(static_context_enrichment.shape)
            static_context_state_h =
self.static_context_state_h_grn(static_encoder)
            #print('static_context_state_h')
```

```
        #print(static_context_state_h.shape)
        static_context_state_c =
self.static_context_state_c_grn(static_encoder)
        #print('static_context_state_c')
        #print(static_context_state_c.shape)

        historical_features, historical_flags \
        = self.temporal_historical_vsn((historical_inputs,

static_context_variable_selection))
        #print('historical_features')
        #print(historical_features.shape)
        #print('historical_flags')
        #print(historical_flags.shape)

        future_features, future_flags \
        = self.temporal_future_vsn((future_inputs,

static_context_variable_selection))
        #print('future_features')
        #print(future_features.shape)
        #print('future_flags')
        #print(future_flags.shape)

        history_lstm, (state_h, state_c) \
        = self.historical_lstm(historical_features,
                               (static_context_state_h.unsqueeze(0),
                                static_context_state_c.unsqueeze(0)))
        #print('history_lstm')
        #print(history_lstm.shape)
        #print('state_h')
        #print(state_h.shape)
        #print('state_c')
        #print(state_c.shape)

        future_lstm, _ = self.future_lstm(future_features,
                                          (state_h,
                                           state_c))
        #print('future_lstm')
        #print(future_lstm.shape)

        # Apply gated skip connection
        input_embeddings = torch.cat((historical_features,
future_features), axis=1)
```

```
        #print('input_embeddings')
        #print(input_embeddings.shape)

        lstm_layer = torch.cat((history_lstm, future_lstm), axis=1)
        #print('lstm_layer')
        #print(lstm_layer.shape)

        temporal_feature_layer =
self.post_seq_encoder_gate_add_norm(lstm_layer, input_embeddings)
        #print('temporal_feature_layer')
        #print(temporal_feature_layer.shape)

        # Static enrichment layers
        expanded_static_context =
static_context_enrichment.unsqueeze(1)

        enriched = self.static_enrichment((temporal_feature_layer,
expanded_static_context))
        #print('enriched')
        #print(enriched.shape)

        # Decoder self attention
        #self.mask = self.get_decoder_mask(enriched)
        #print('enriched')
        #print(enriched.shape)
        x, self_att = self.self_attn_layer(enriched,
                                           enriched,
                                           enriched,
                                           mask =
self.get_decoder_mask(enriched))
        #print('x')
        #print(x.shape)
        #print('self_att')
        #print(self_att.shape)

        x = self.post_attn_gate_add_norm(x, enriched)
        #print('x')
        #print(x.shape)

        # Nonlinear processing on outputs
        decoder = self.GRN_positionwise(x)
        #print('decoder')
        #print(decoder.shape)
```

```
        # Final skip connection
        transformer_layer = self.post_tfd_gate_add_norm(decoder,
temporal_feature_layer)
        #print('transformer_layer')
        #print(transformer_layer.shape)

        outputs =
self.output_feed_forward(transformer_layer[Ellipsis,
self.num_encoder_steps:, :])
        #print('outputs')
        #print(outputs.shape)

        #ipdb.set_trace()

        return outputs

    def loss(self, y_hat, y):
        return self.train_criterion.apply(y_hat, y)

    def test_loss(self, y_hat, y):
        return self.test_criterion.apply(y_hat, y, self.quantiles[1])

    def training_step(self, batch, batch_nb):
        x, y, _ = batch

        x = x.to(torch.float)
        y = y.to(torch.float)
#         print('y')
#         print(y.shape)
        y_hat = self.forward(x)
#         print('y_hat')
#         print(y_hat.shape)
        loss = self.loss(y_hat, torch.cat([y, y, y], dim = -1))
        #print(loss.shape)
        tensorboard_logs = {'train_loss': loss}
        return {'loss': loss, 'log': tensorboard_logs}

    def validation_step(self, batch, batch_nb):
        x, y, _ = batch
        x = x.to(torch.float)
        y = y.to(torch.float)
        y_hat = self.forward(x)
        #print(y_hat.shape)
        #print(torch.cat([y, y, y], dim = -1).shape)
```

```
        loss = self.loss(y_hat, torch.cat([y, y, y], dim = -1))
        #print(loss)
        return {'val_loss': loss}

    def validation_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in
outputs]).mean()
        tensorboard_logs = {'val_loss': avg_loss}
        return {'avg_val_loss': avg_loss, 'log': tensorboard_logs}

    def test_step(self, batch, batch_idx):
        # OPTIONAL
        x, y, _ = batch
        x = x.to(torch.float)
        y = y.to(torch.float)
        y_hat = self.forward(x)
        return {'test_loss': self.test_loss(y_hat[Ellipsis, 1],
y[Ellipsis, 0])}

    def test_end(self, outputs):
        # OPTIONAL
        avg_loss = torch.stack([x['test_loss'] for x in
outputs]).mean()
        tensorboard_logs = {'test_loss': avg_loss}
        return {'avg_test_loss': avg_loss, 'log': tensorboard_logs}

    def configure_optimizers(self):
        # REQUIRED
        # can return multiple optimizers and learning_rate schedulers
        # (LBFGS it is automatically supported, no need for closure
function)
        return [torch.optim.Adam(self.parameters(),
lr=self.learning_rate)]

    def plot_grad_flow(self, named_parameters):
        ave_grads = []
        layers = []
        for name, p in named_parameters:
            if p.grad is not None:
                if (p.requires_grad) and ("bias" not in name):
                    layers.append(name)
                    ave_grads.append(p.grad.abs().mean())
                    self.logger.experiment.add_histogram(tag=name,
values=p.grad,
```

```
global_step=self.trainer.global_step)
            else:
                 print('{} - {}'.format(name, p.requires_grad))

        plt.plot(ave_grads, alpha=0.3, color="b")
        plt.hlines(0, 0, len(ave_grads), linewidth=1, color="k" )
        plt.xticks(list(range(0,len(ave_grads), 1)), layers,
rotation='vertical')
        plt.xlim(left=0, right=len(ave_grads))
        plt.xlabel("Layers")
        plt.ylabel("average gradient")
        plt.title("Gradient flow")
        plt.grid(True)
        plt.rcParams["figure.figsize"] = (20, 5)

    def on_after_backward(self):
        # example to inspect gradient information in tensorboard
        if self.trainer.global_step % 25 == 0:
            self.plot_grad_flow(self.named_parameters())

    def train_dataloader(self):
        # REQUIRED
        return DataLoader(train_dataset, batch_size =
self.minibatch_size, shuffle=True, drop_last=True, num_workers=1)

    def val_dataloader(self):
        # OPTIONAL
        return DataLoader(valid_dataset, batch_size =
self.minibatch_size, shuffle=True, drop_last=True, num_workers=1)

    def test_dataloader(self):
        # OPTIONAL
        return DataLoader(test_dataset, batch_size =
self.minibatch_size, shuffle=True, drop_last=True, num_workers=1)
```