本文介绍 attention 机制的一些实现技巧和在不同场景的变种。

1. Transformer 中的 mask 机制

我们知道 transformer 中的 self-attention、context-attention（即 encoder 和 decoder 之间的 attention）都是基于 multi-head attention，而这个 multi-head 的每一个“头”指的是同一份数据经过不同的线性映射后，分别做 scaled dot-product attention。所以总结来说，整个 transformer 都在反复调用 dot-product attention 这一“粒度”最小的结构。看下实现：

```
Args:
        q: Queries张量，形状为[B, L_q, D_q]
        k: Keys张量，形状为[B, L_k, D_k]
        v: Values张量，形状为[B, L_v, D_v]，一般来说就是k
        scale: 缩放因子，一个浮点标量
        attn_mask: Masking张量，形状为[B, L_q, L_k]

Returns:
        上下文张量和attetention张量
"""
attention = torch.bmm(q, k.transpose(1, 2))
if scale:
        attention = attention * scale
if attn_mask:
        # 给需要mask的地方设置一个负无穷
        attention = attention.masked_fill_(attn_mask, -np.inf)
        # 计算softmax
attention = self.softmax(attention)
        # 添加dropout
attention = self.dropout(attention)
        # 和V做点积
context = torch.bmm(attention, v)
return context, attention
```

注意到，在计算 attention score(L_q * L_k)之后，进行了一个 masking 的操作：将 attn_mask 中标为 1 的位置对应的 attention score 置为负无穷，这样经过 softmax 之后计算出的 weight 就接近 0，从而实现掩盖的目的。那么 mask 是怎样产生的，又是用于什么场景呢？

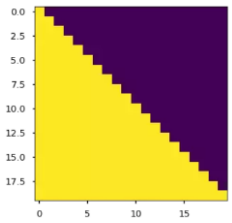
在 encoder、decoder 等所有 attention 都会使用的一种 mask 叫做 padding mask， 作用是使模型能并行处理不同长度的序列，对于一个 batch 的数据，我们通过加 padding 使其对齐，而这些填充的位置不应该获得注意力，我们通过如下代码产生 mask：

```
def padding_mask(seq_k, seq_q):
        # seq_k和seq_q的形状都是[B,L]
    len_q = seq_q.size(1)
    # `PAD` is 0
    pad_mask = seq_k.eq(0)
    pad_mask = pad_mask.unsqueeze(1).expand(-1, len_q, -1)  # shape [B, L_q, L_k]
    return pad_mask
```

在 decoder 的 self-attention 中我们还会再加上另外一种 mask 叫 sequence mask。decoder 会直接将 ground truth 作为输入，但是在前面的时刻预测时，不应该看得到之后时刻的信息，为此我们需要将每个当前 query 后面的序列遮起来，这可以通过一个上三角矩阵实现，上三角的值全

为 1，下三角和对角线的值全为 0。把这个矩阵作用在每一个序列上即可。

```
def sequence_mask(seq):
    batch_size, seq_len = seq.size()
    mask = torch.triu(torch.ones((seq_len, seq_len), dtype=torch.uint8),
                      diagonal=1)
    mask = mask.unsqueeze(0).expand(batch_size, -1, -1)  # [B, L, L]
    return mask
```



2. Scheduled Sampling for Transformers

transformer 和 rnn 的一个最大的区别是前者是并行处理数据的，而后者哪怕以 gt 作为输入，也只能串行。并行给 transformer 带来了更大的感受野和更快的训练速度，但是 nlp 中常见的一个 trick：schedules sampling 却难以用进来，从而会出现 exposure bias 的问题。为了在并行的前提下可以引入这个 trick，acl19 workshop 的一个工作提出了一种 two-pass 的解决方案：

（1）在每个 training-steps，第一趟先利用 teacher-forcing 技术，计算出当前句子中每个解码位置所有单词的分数（logits）

（2）根据一定的概率 p，选择第二趟解码时，是否用第一趟生成的单词作为 decode 输入，还是沿用 ground-truth 作为输入（只有第二趟解码会进行 back-propogation）

这样在保证并行的同时，也能够利用了上一次参数下得到的输出作为输入。

3. Soft & hard attention

这个名字会给人造成一个误解，好像 soft attention 是对所有的 value 的加权，而 hard 是加权的概率向量收敛到了 one hot（前面介绍的 dot product attention 的 scale 操作就是为了避免变成 one hot）。其实 hard attention 是指利用这个概率向量采样一个 value，作为 attention 的结果。而引入随机性会导致梯度无法回传，解决的方法是类似于 REINFORCE 的方法。

具体来说，我们将每次选新的单词 s 视为从 multinouilli 分布中采样的过程：

$$p(s_{t,i} = 1 \mid s_{j<t}, \mathbf{a}) = \alpha_{t,i}$$

$$\hat{\mathbf{z}}_t = \sum_i s_{t,i}\mathbf{a}_i.$$

根据琴森不等式，可以得到最大似然的下限，然后去优化这个下限即可，如果将单词序列表示为 y，则得到下式：

$$\begin{aligned} L_s &= \sum_s p(s \mid \mathbf{a}) \log p(\mathbf{y} \mid s, \mathbf{a}) \\ &\leq \log \sum_s p(s \mid \mathbf{a}) p(\mathbf{y} \mid s, \mathbf{a}) \\ &= \log p(\mathbf{y} \mid \mathbf{a}), \end{aligned}$$

$$\frac{\partial L_s}{\partial W} = \sum_s p(s \mid \mathbf{a}) \left[\frac{\partial \log p(\mathbf{y} \mid s, \mathbf{a})}{\partial W} + \log p(\mathbf{y} \mid s, \mathbf{a}) \frac{\partial \log p(s \mid \mathbf{a})}{\partial W} \right].$$

不同于 em 算法，引入一个未知分布 q(s|a)再去趋近后验分布 p(s|a,y)，这里就是直接用了先验分布 p(s|a)，最后得到的式子，直观上来看就是将原本的最大似然变成了在这个先验分布下，也就是当下模型产生单词的概率下的最大似然。这里看起来有点奇怪，因为 q 取任何一个分布都会是原本似然概率的下限，所以取先验分布也是可以的，但是不能保证这个下限很接近真实的似然概率。在求得梯度之后，用蒙特卡洛采样代替期望。

$$\frac{\partial L_s}{\partial W} \approx \frac{1}{N}\sum_{n=1}^{N}\left[\frac{\partial \log p(\mathbf{y} \mid \tilde{s}^n, \mathbf{a})}{\partial W} + \log p(\mathbf{y} \mid \tilde{s}^n, \mathbf{a})\frac{\partial \log p(\tilde{s}^n \mid \mathbf{a})}{\partial W}\right]$$

4. Global&local attention

对于长序列的生成，如果我们仍然对全局做 attention，会导致计算量的增长。文章" Effective Approaches to Attention-based Neural Machine Translation" 提出 local attention 的方法，只对一个小的窗口做 attention，可以极大的减小计算量。本工作做的是机翻的任务，为了找到这个窗口，先利用当前生成单词的 position t 得到一个源语言句子的 position p，然后在 p 上下文取一个窗口。那么要怎么找到这个位置 p，使得我们想要关注的内容在其上下文中呢?

有两种方法，第一种叫 monotonic alignment，即取 p=t，这种方法是无参的，也可以引申为其他无参方法，比如如果是同一种语言的任务（如摘要），可以找与当前词最相似源句中的词等。第二种是需要训练的，利用 ht 进行预测:

$$p_t = S \cdot \text{sigmoid}(\boldsymbol{v}_p^\top \tanh(\boldsymbol{W_p h}_t))$$

得到位置 p 之后，以高斯概率从这个位置向两边衰减，表现为一个权重，加在 attention score 上:

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \exp\left(-\frac{(s-p_t)^2}{2\sigma^2}\right)$$