第5章

注意力模型

深度学习中的注意力机制(Attention Mechanism)是从人类注意力机制中获取的灵感。人的大脑在接收外界多种多样的信息中,只关注重要的信息,而忽略无关紧要的信息,这就是注意力的体现。注意力机制能帮助神经网络选择关键重要的信息进行处理,不仅能减小神 经网络中的计算量,而且使模型能作出更加准确的预测。本章将首先介绍注意力机制的原理,接着介绍基于自注意力机制的 Transformer 模型及其实现,最后详细介绍 PaddlePaddle 在基于 seq2seq 的对联生成问题中的实际应用。学习本章,希望读者能够:

- 理解注意力机制的基本原理;
- 掌握自注意力机制的基本原理及 Transformer 模型结构;
- 使用 PaddlePaddle 搭建简单的 Transformer 模型。

5.1 任务简介

古诗和对联是中国文化的精髓。古诗一般被用来歌颂英雄人物、美丽的风景、爱情、友谊等。古诗被分为很多类,例如,唐诗、宋词、元曲等,每种古诗都有自己独特的结构、韵律。表 5-1 展示了一种中国古代最流行的古诗体裁——唐诗绝句。绝句在结构和韵律上具有严格的规则:每首诗由 4 行组成,每一行有 5 个或者 7 个汉字(5 个汉字称为五言绝句,7 个汉字称为七言绝句);每个汉字音调要么是平,要么是仄;诗的第二行和最后一行的最后一个汉字必须同属一个韵部。正因为绝句在结构和韵律上具有严格的限制,所以好的绝句朗诵起来具有很强的节奏感。

表 5-1　唐诗绝句《望庐山瀑布》

绝　　句	韵　　律
日照香炉生紫烟	（仄仄平平平仄平）
遥看瀑布挂前川	（平仄仄仄仄平平）
飞流直下三千尺	（平平平仄平平仄）
疑是银河落九天	（平仄平平仄仄平）

对联一般在春节、婚礼、贺岁等场合下写于红纸贴于门墙上，代表人们对美好生活的祝愿。表 5-2 展示了一副中国对联。对联分为上联和下联，上下联具有严格的约束，在结构上要求长度一致，语义上要求词性相同，音调上要求仄起平落。如表 5-2 中的对联长度一致，即汉字个数相同；语义相对，“一帆风顺”对“万事如意”，“年年好”对“步步高”；在最后一个字符的音调上仄起平落，“好”是仄，“高”是平，因此好的对联读起来会感觉朗朗上口。

表 5-2　中国对联

对　　联	韵　　律
一帆风顺年年好	（仄平平仄平平仄）
万事如意步步高	（仄仄平仄仄仄平）

在自然语言处理中，古诗和对联的自动生成一直是研究的热点。近几年，古诗和对联的自动生成研究得到了学术界的广泛关注。科研工作者们采用了各种方法研究古诗和对联的生成，包括采用规则和模板的方式、采用文本生成算法、采用自动摘要的方法、采用统计机器翻译的方法等。最近，深度学习方法被广泛地应用于古诗和对联生成任务上，并取得了很大成效。主要采用序列到序列循环神经网络和卷积神经网络模型来生成古诗和对联，此类方法在古诗和对联生成任务上取得了很大的进步，但也存在着一定的问题：如采用的单任务模型，则泛化能力低；在古诗生成上如输入现代词汇，则系统就会出现问题。此外，此类方法在生成时，需要限制用户的输入，当输入符合条件时才能创作，如此增加了用户使用的难度。基于卷积或循环网络的序列是一种局部的编码方式，只建模了输入信息的局部依赖关系.虽然循环网络在理论上可以建立长距离依赖关系，但是由于信息传递的容量以及梯度消失问题，实际上也只能建立短距离的依赖关系。如果要建立输入序列之间的长距离依赖关系，可以使用注意力机制来解决问题。5.2 节对将注意力机制进行介绍，5.3 节给出基于注意力机制的古诗和对联任务的设计方案。

5.2　注意力机制

为了解决序列到序列模型记忆长序列能力不足的问题（如机器翻译问题），一个非常直观的想法是当生成一个目标语言单词时，不光考虑前一个时刻的状态和已经生成的单词，还考虑当前要生成的单词和源语言句子中的那些单词，即更关注源语言的那些词，这种做法叫作注意力机制。注意力模型已被广泛地应用在自然语言处理、语音识别、图像识别等任务中，本节从注意力机制原理、自注意力机制、多头注意力机制和 Transformer 模型四个方面介绍相关理论。

5.2.1 注意力机制原理

在人类认识事物和阅读文本的过程中，总会有选择性地关注全局的部分信息，获得需要重点关注的目标区域，而抑制其他无用信息，这种方式是注意力机制在认知科学中的体现。深度学习中的注意力机制与之相似，目标是从全部信息中选择对当前任务更关键的信息。

假设 N 个输入信息 $\boldsymbol{X}=[\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_N]$，给定查询向量 $\boldsymbol{q}$，定义选择第 i 个输入信息的概率注意力分布为 α_i。如图 5-1 所示，注意力机制分布的计算可以分为：

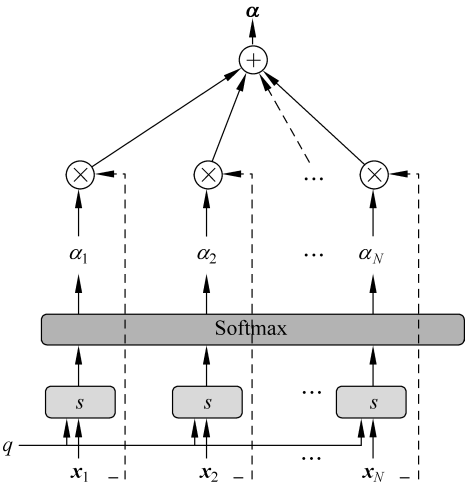


图 5-1 注意力分布计算

(1) 注意力分布计算，公式如下

$$\alpha_i=\mathrm{Softmax}(\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q}))=\frac{\exp(\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q}))}{\sum_{j=1}^{N}\exp(\boldsymbol{s}(\boldsymbol{x}_j,\boldsymbol{q}))} \tag{5-1}$$

其中，$\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q})$为注意力打分函数，不同的注意力打分函数如表 5-3 所示。

(2) 输入信息的加权平均计算，公式如下

$$r=\sum_i \alpha_i h_i \tag{5-2}$$

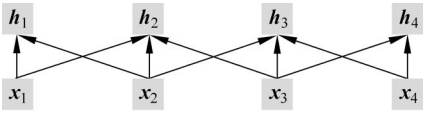
式(5-2)获得输入向量的重要性得分 α_i 后，将 h_i 和 α_i 相乘，体现更重要性的输入部分在整个输出向量的学习中贡献更大。

表 5-3 注意力打分函数

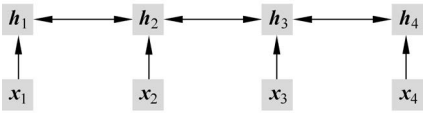
模　　型	函数表达式
加权模型	$\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q})=\boldsymbol{v}^{\mathrm{T}}\tanh(\boldsymbol{W}\boldsymbol{x}_i+\boldsymbol{U}\boldsymbol{q})$
点积模型	$\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q})=\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{q}$
缩放点积模型	$\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q})=\frac{\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{q}}{\sqrt{d}}$
双线性模型	$\boldsymbol{s}(\boldsymbol{x}_i,\boldsymbol{q})=\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{W}\boldsymbol{q}$

5.2.2　自注意力机制

当使用神经网络来处理一个变长的向量序列时，通常可以使用卷积网络或循环网络进行编码，来得到一个相同长度的输出向量序列，如图 5-2 所示。



(a) 卷积网络



(b) 双向循环网络

图 5-2　基于卷积网络和循环网络的变长序列编码

基于卷积或循环网络的序列编码都是一种局部的编码方式，只建模了输入信息的局部依赖关系。虽然循环网络理论上可以建立长距离依赖关系，但是由于信息传递的容量以及梯度消失问题，实际上也只能建立短距离依赖关系。如果要建立输入序列之间的长距离依赖关系，可以使用以下两种方法：一种方法是增加网络的层数，通过一个深层网络来获取远距离的信息交互；另一种方法是使用全连接网络. 全连接网络是一种非常直接的建模远距离依赖的模型，但是无法处理变长的输入序列。不同的输入长度，其连接权重的大小也是不同的。这时就可以利用注意力机制来“动态”地生成不同连接的权重，这就是自注意力模型(Self-Attention Model)，自注意力也称为内部注意力(Intra-attention)。

1. 数学定义

为了提高模型能力，自注意力模型经常采用查询-键-值(Query-Key-Value，QKV)模式，其计算过程如图 5-3 所示，其中浅色字母表示矩阵的维度。

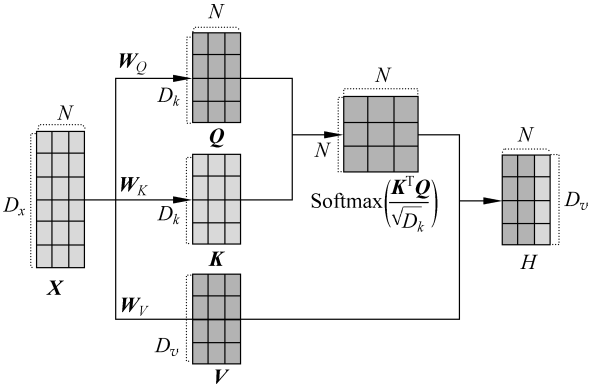


图 5-3　自注意力模型的计算过程

假设输入序列为 $\boldsymbol{X}=[\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_N]\in\mathbb{R}^{D_x\times N}$，输出序列为 $\boldsymbol{H}=[\boldsymbol{h}_1,\boldsymbol{h}_2,\cdots,\boldsymbol{h}_N]\in\mathbb{R}^{D_v\times N}$，自注意力模型的具体计算过程如下：对于每个输入 $\boldsymbol{x}_i$，我们首先将其线性映射到三个不同的空间，得到查询向量 $\boldsymbol{q}_i\in\mathbb{R}^{D_k}$、键向量 $\boldsymbol{k}_i\in\mathbb{R}^{D_k}$ 和值向量 $\boldsymbol{v}_i\in\mathbb{R}^{D_v}$。对于整个输入序列 $\boldsymbol{X}$，线性映射过程可以简写为 $\boldsymbol{Q}=\boldsymbol{X}\boldsymbol{W}_Q$、$\boldsymbol{K}=\boldsymbol{X}\boldsymbol{W}_K$ 和 $\boldsymbol{V}=\boldsymbol{X}\boldsymbol{W}_V$。

如果使用如表 5-3 所示的缩放点积来作为注意力打分函数，输出向量序列可以简写为：

$$\boldsymbol{H}=\boldsymbol{V}\mathrm{Softmax}\left(\frac{\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Q}}{\sqrt{D_k}}\right) \tag{5-3}$$

2. 计算过程

上面介绍了自注意力的数学原理，接下来以实例说明其计算方法：

(1) 从每个编码器的输入向量创建三个向量：**Query** 向量、**Key** 向量、**Value** 向量，如图 5-4 所示。注意：在原论文中，这三个向量尺寸(64 维)小于嵌入向量或者输入维数(512 维)，目的是可以完成多头注意力计算。

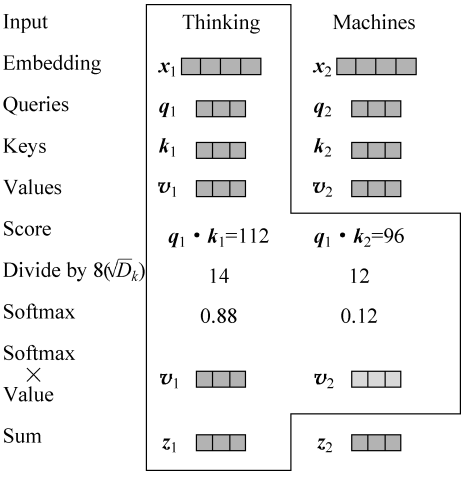


图 5-4　自注意力计算过程

(2) 计算第一个单词“Thinking”的打分数值 score。该数值 score 的计算方式是 $\boldsymbol{q}$ 与 $\boldsymbol{k}$ 向量的点积，即先计算 $\boldsymbol{q}_1\cdot\boldsymbol{k}_1$，再计算 $\boldsymbol{q}_1\cdot\boldsymbol{k}_2$。

(3) 将 score 除以 8(论文中使用的 $\boldsymbol{k}$ 向量维数 64 的算术平方根，这使得模型具有更稳定的梯度。这里可能存在其他合理的值，但默认采用刚刚的计算方法)，然后将结果传入 Softmax 操作。Softmax 将分数标准化，从而使得它们都是正数并且累加和为 1。

(4) 将 $\boldsymbol{v}$ 向量与对应的 Softmax 值相乘，以便基于打分值抽取相应的信息，即保持关注的单词不变的情况下，过滤掉不相关的词汇。

(5) 产生累加求和项：$\boldsymbol{z}=0.88\times\boldsymbol{v}_1+0.12\times\boldsymbol{v}_2$。

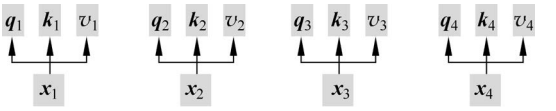
3. 矩阵形式

上述过程阐述了自注意力的总体计算流程，但在实际使用过程中，为便于编程和加速计算等需求，常常以矩阵运算实现自注意力的计算。

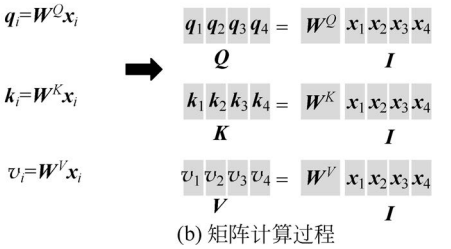
1) $\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}$ 三向量

将输入变为行向量 $\boldsymbol{X}(\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{x}_3,\boldsymbol{x}_4)$，乘以对应的权重矩阵 $\boldsymbol{W}^Q,\boldsymbol{W}^K,\boldsymbol{W}^V$，得到行向量 $\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}$，其详细推导过程如图 5-5 所示。详细过程如下：

- 如图 5-5(a)所示，$\boldsymbol{x}_1$、$\boldsymbol{x}_2$、$\boldsymbol{x}_3$、$\boldsymbol{x}_4$ 分别乘 $\boldsymbol{W}^q$ 得到 $\boldsymbol{q}_1$、$\boldsymbol{q}_2$、$\boldsymbol{q}_3$、$\boldsymbol{q}_4$，然后利用线性代数知识将 $\boldsymbol{x}_1$、$\boldsymbol{x}_2$、$\boldsymbol{x}_3$、$\boldsymbol{x}_4$ 拼接成矩阵 $\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3\boldsymbol{x}_4$(图 5-5(b))。其中 $\boldsymbol{I}$ 表示输入，X 为标量，权重矩阵 $\boldsymbol{W}^Q$、$\boldsymbol{W}^K$、$\boldsymbol{W}^V$ 在模型训练时通过学习获得。



(a) 图示计算过程



(b) 矩阵计算过程

图 5-5　获得 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 三向量过程

- $\boldsymbol{I}$ 乘以 $\boldsymbol{W}^q$ 就得到另外一个矩阵 $\boldsymbol{Q}$，$\boldsymbol{Q}$ 由 $\boldsymbol{q}_1$ 到 $\boldsymbol{q}_4$ 四个向量拼接而成。
- $\boldsymbol{K},\boldsymbol{V}$ 的计算过程如(1)和(2)相同。

2) 打分矩阵

如图 5-6 给出了列向量 $\boldsymbol{K}^{\mathrm{T}}$ 与行向量 $\boldsymbol{Q}$ 相乘得到打分矩阵 $\boldsymbol{A}$ 的过程，具体如下：

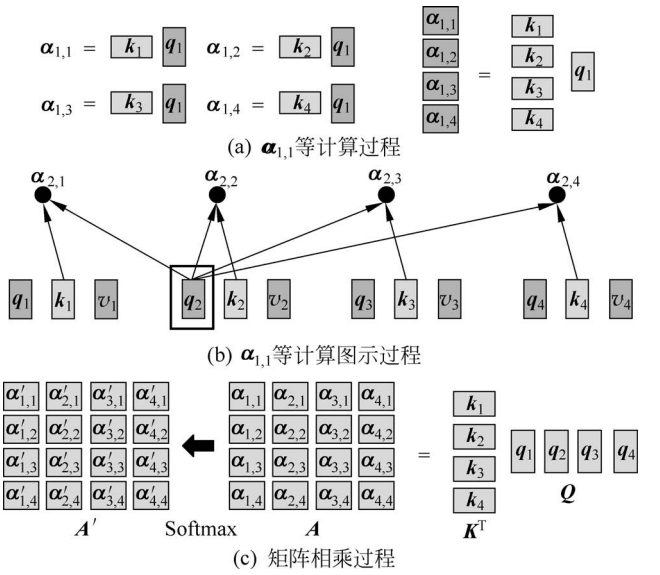
- 如图 5-5(a)所示，$\boldsymbol{q}_1$ 跟 $\boldsymbol{k}_1$ 缩放点积运算得到 $\boldsymbol{\alpha}_{1,1}$ 和 $\boldsymbol{\alpha}_{1,1}$。同理得到 $\boldsymbol{\alpha}_{1,2}$、$\boldsymbol{\alpha}_{1,3}$、$\boldsymbol{\alpha}_{1,4}$。对于以上四个步骤的操作，可以利用线性代数知识将 $\boldsymbol{k}_1$、$\boldsymbol{k}_2$、$\boldsymbol{k}_3$、$\boldsymbol{k}_4$ 按行拼接成矩阵 $\boldsymbol{k}_1\boldsymbol{k}_2\boldsymbol{k}_3\boldsymbol{k}_4$，然后该矩阵跟向量 $\boldsymbol{q}_1$ 相乘得到列向量 $\boldsymbol{\alpha}_{1,1}\boldsymbol{\alpha}_{1,2}\boldsymbol{\alpha}_{1,3}\boldsymbol{\alpha}_{1,4}$。如图 5-5(b)给出图示计算 $\boldsymbol{\alpha}_{2,1}\boldsymbol{\alpha}_{2,2}\boldsymbol{\alpha}_{2,3}\boldsymbol{\alpha}_{2,4}$ 过程。
- 重复上述过程，可以获得打分矩阵 $\boldsymbol{A}$，经过 Softmax 处理后得到打分矩阵 $\boldsymbol{A}'$。

3) 输出向量 $\boldsymbol{H}$

如图 5-7 所示，与 1)和 2)类似可得到输出向量 $\boldsymbol{H}$，这里不再重复介绍。

4. 多头注意力机制

自注意力模型可以作为神经网络中的一层来使用，既可以用来替换卷积层和循环层，也可以和它们一起交替使用(比如 $\boldsymbol{X}$ 可以是卷积层或循环层的输出)。自注意力模型计

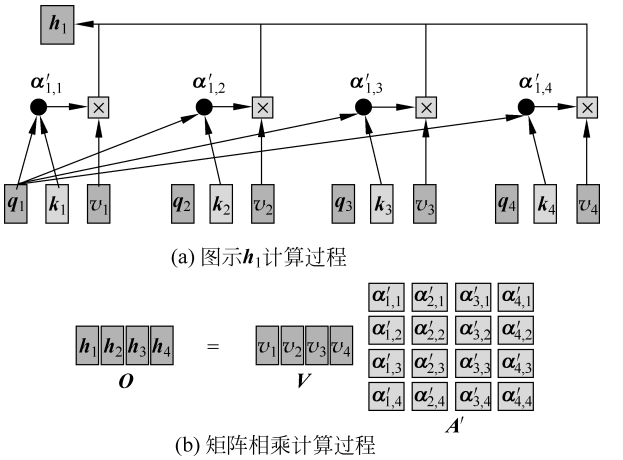


(a) $\boldsymbol{\alpha}_{1,1}$等计算过程

(b) $\boldsymbol{\alpha}_{1,1}$等计算图示过程

(c) 矩阵相乘过程

图 5-6　打分矩阵



(a) 图示$\boldsymbol{h}_1$计算过程

(b) 矩阵相乘计算过程

图 5-7　输出向量 $\boldsymbol{H}$ 计算过程

算的权重 α_{ij} 只依赖于 $\boldsymbol{q}_i$ 和 $\boldsymbol{k}_j$ 的相关性，而忽略了输入信息的位置信息。因此在单独使用时，自注意力模型一般需要加入位置编码信息来进行修正。自注意力模型可以扩展为多头自注意力(Multi-Head Self-Attention)模型，在多个不同的投影空间中捕捉不同的交互信息，即利用多个查询 $Q=[q_1,q_2,\cdots,q_M]$，来并行地从输入信息中选取多组信息. 每个注意力关注输入信息的不同部分，如图 5-8 所示。

5.2.3　Transformer 模型

广义的 Transformer 是一种基于注意力机制的前馈神经网络，主要由编码器(Encoder)和解码器(Decoder)两部分组成。在 Transformer 的原论文中，编码器和解码

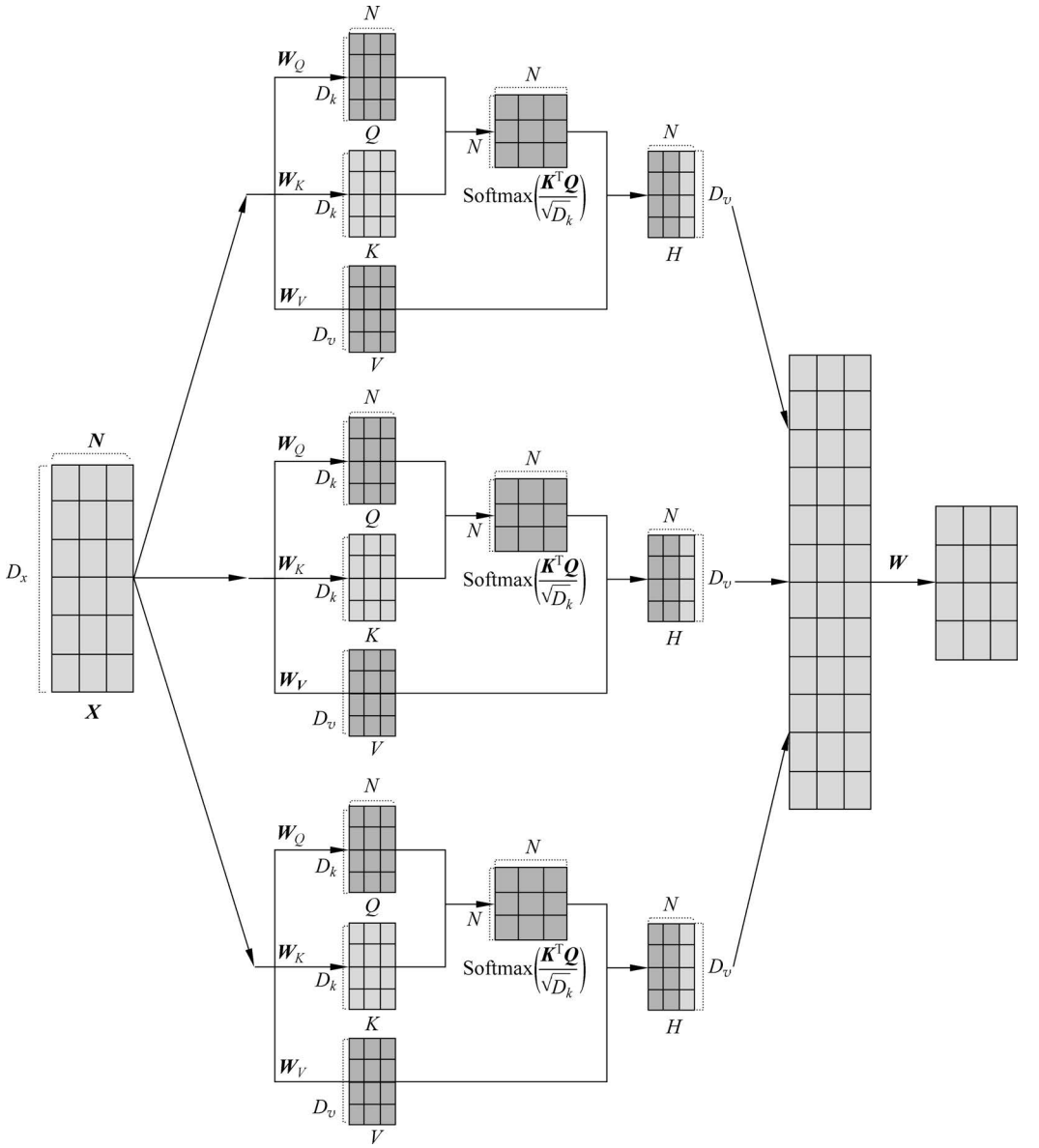


图 5-8　多头注意力机制

器均由 6 个编码器子层(Encoder Layer)和解码器子层(Decoder Layer)组成,该子层通常称之为 Transformer 块(Block),具体网络结构如图 5-9 所示。

1. 编码器

首先,模型需要对输入的数据进行 Embedding 操作,Embedding 结束之后,输入到编码器子层,自注意力处理完数据后把数据送给前馈神经网络,得到的输出会输入到下一个 Transformer 块,具体如图 5-10 所示。

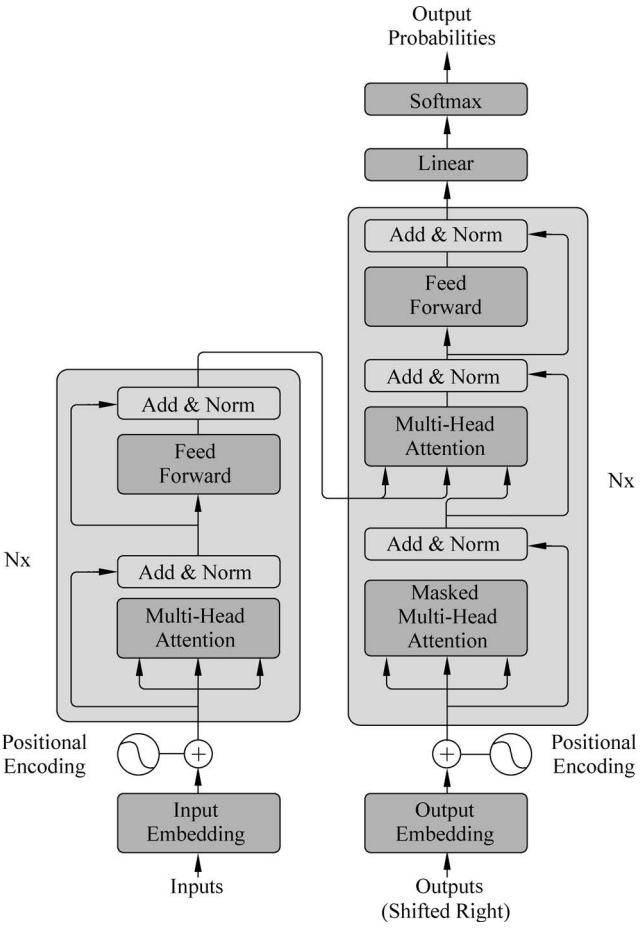


图 5-9 Transformer 模型网络结构

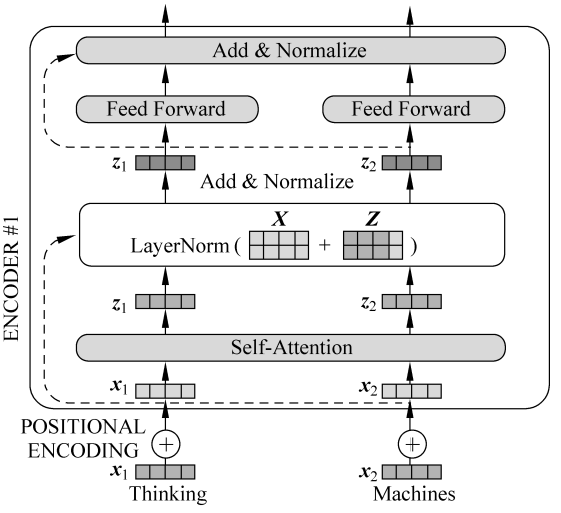


图 5-10 编码器①

① 图片来源：http://jalammar.github.io/illustrated-transformer/

编码器由 $N=6$ 个相同的 Transformer 块堆叠在一起组成。每块由多头注意力机制和全连接的前馈网络两个子层构成。其中，每个子层都加了残差连接和层归一化，具体过程如下：

(1) 输入部分主要完成输入 $\boldsymbol{x}_1$ 和 $\boldsymbol{x}_2$ 及其位置信息的编码，如下所示：

$$\boldsymbol{X}=\text{EmbeddingLookup}(\boldsymbol{X})+\text{PositionEncoding}(\boldsymbol{X}) \tag{5-4}$$

其中，查找表函数 EmbeddingLookup($\boldsymbol{X}$)是获得输入序列的词向量序列，位置编码 PositionEncoding($\boldsymbol{X}$)计算输入词的位置信息，若输入词在偶数位置，使用正弦编码 $\text{PE}(\text{pos},2i)=\sin(\text{pos}/10000^{2i}/d_{\text{model}})$，否则使用余弦编码 $\text{PE}(\text{pos},2i+1)=\cos(\text{pos}/10000^{2i}/d_{\text{model}})$。

(2) 多层注意力处理后获得 $\boldsymbol{z}_1$ 和 $\boldsymbol{z}_2$，其公式如下

$$\boldsymbol{Z}=\text{selfAttention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) \tag{5-5}$$

式(5-5)具体实现过程见 5.2.2 节。

(3) 残差连接与层归一化，其公式如下

$$\boldsymbol{Z}=\text{LayerNorm}(\boldsymbol{X}+\boldsymbol{Z}) \tag{5-6}$$

(4) 两层线性映射并用激活函数激活，其公式如下

$$\boldsymbol{Z}_{\text{hidden}}=\text{Linear}(\text{ReLU}(\text{Linear}(\boldsymbol{Z})) \tag{5-7}$$

(5) 残差连接与层归一化，其公式如下

$$\boldsymbol{Z}=\boldsymbol{Z}+\boldsymbol{Z}_{\text{hidden}} \tag{5-8}$$

$$\boldsymbol{X}_{\text{hidden}}=\text{LayerNorm}(\boldsymbol{Z}) \tag{5-9}$$

2. 解码器

解码器和编码器有类似的结构，也是 $N=6$ 个相同的层堆叠在一起组成。相比于编码器，输入层中多了个掩码多头注意力子层。同时，如图 5-11 所示中间位置，在解码器中每块的查询向量 $\boldsymbol{Q}$，会同编码器提供的记忆信息($\boldsymbol{Q}$,$\boldsymbol{V}$)作自注意力计算，称为交叉注意力(Cross Attention)。总之，在整个 Transformer 结构中有三种注意力机制：多头注意力机制、掩码多头注意力机制、交叉注意力机制，请读者区分其出现的位置以及差异之处。如图 5-11 以翻译为例展示了解码器的解码过程，解码器中的字符预测完之后，会当成输入预测下一个字符，直到遇见终止符号为止。

5.2.4　模型实现

PaddlePaddle 框架已实现了 Transformer 模型。其中，nn. TransformerEncoder 实现了编码模型，其模型由多个 Transformer 编码器层(TransformerEncoderLayer)叠加组成。Transformer 编码器层由两个子层组成：多头自注意力机制和前馈神经网络。如果 normalize_before 为 True，则对每个子层的输入进行层标准化(Layer Normalization)，对每个子层的输出进行 dropout 和残差连接(Residual Connection)；否则(即 normalize_before 为 False)，则对每个子层的输入不进行处理，只对每个子层的输出进行 dropout、残差连接和层标准化(Layer Normalization)。下面代码演示其具体过程。

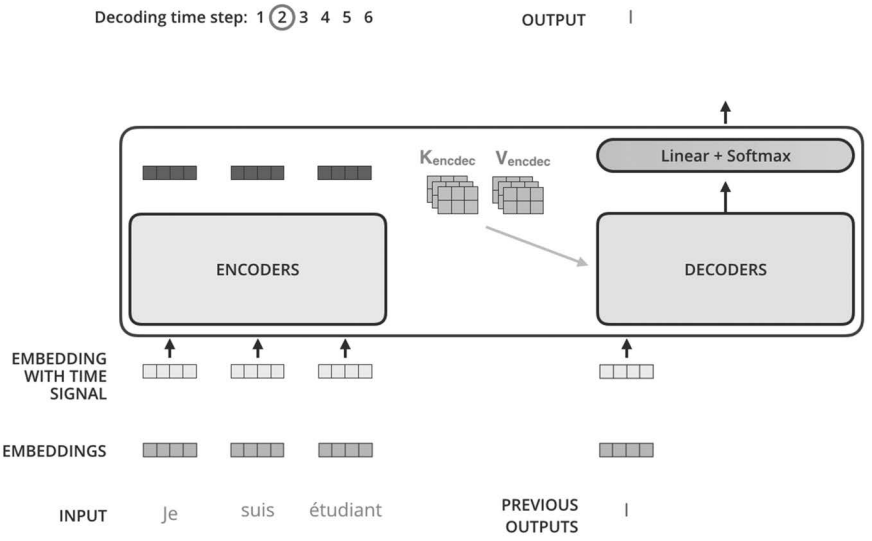


图 5-11 解码器解码过程

```
01. import paddle
02. #创建一个 Transformer 块,每个输入向量、输出向量的维度为 4,头数为 2,前馈神经网络中
    隐藏层的大小为 128
03. encoder_layer = paddle.nn.TransformerEncoderLayer(d_model = 4, nhead = 2, dim_
    feedforward = 128)
04. #输入一个随机张量,批次为 2,每批次 3 个数据,每个数据维数 4
05. src = paddle.randn((2, 3, 4))
06. out1 = encoder_layer(src)
07. print("输出结果 1: ",out1)
```

运行结果如下。

```
输出结果: Tensor(shape = [2, 3, 4], dtype = float32, place = CUDAPlace(0),
top_gradient = False,
    [[[ - 1.12602544, 0.86143279, 1.12097585, - 0.85638326],
      [ - 0.02759376, - 0.73740852, - 0.87503952, 1.64004183],
      [ 1.60045302, - 0.48906180, - 1.09501755, - 0.01637374]],

     [[ 0.84246475, - 0.50896662, 1.42670989, - 1.26020789],
      [ - 0.91479391, 1.07564783, 0.91786611, - 1.07872021],
      [ 0.87792587, - 0.82447380, 1.10199046, - 1.15544271]]])
```

然后,可以将多个 Transformer 块堆叠起来,构成一个完整的 nn.TransformerEncoder。代码如下所示。

```
01. transformer_encoder = paddle.nn.TransformerEncoder(encoder_layer,num_layers = 6)
02. out2 = transformer_encoder(src)
03. print("输出结果 2:",out2)
```

运行结果如下。

```
输出结果 2: Tensor(shape = [2, 3, 4], dtype = float32, place = CUDAPlace(0),
top_gradient = False,
    [[[ 0.21231177, -1.26825130, 1.47372913, -0.41778976],
      [-0.59615535, -1.29232252, 0.63498265, 1.25349522],
      [ 0.22930427, 1.17037129, -1.59372377, 0.19404820]],

    [[ 0.55925030, -1.09686399, 1.35037553, -0.81276196],
     [ 0.23460560, -1.24863398, 1.47441125, -0.46038279],
     [ 0.22599012, -1.38127530, 1.40638554, -0.25110036]]])
```

解码模块也有上述类似结构，TranformerDecoderLayer 定义了一个解码模型的 Transformer 层，通过多层堆叠构成了 nn.TransformerDecoder。下面代码演示其具体调用方式。

```
01. memory = transformer_encoder(src)
02. decoder_layer = paddle.nn.TransformerDecoderLayer(d_model = 4, nhead = 2, dim_
    feedforward = 128)
03. transformer_decoder = paddle.nn.TransformerDecoder(decoder_layer,num_layers = 6)
04. out_part = paddle.randn((2, 3, 4))
05. out3 = transformer_decoder(out_part,memory)
06. print("输出结果 3:",out3)
```

运行结果如下。

```
输出结果 3: Tensor(shape = [2, 3, 4], dtype = float32, place = CUDAPlace(0),
top_gradient = False,
    [[[ 0.89957052, 0.20762412, 0.57216758, -1.67936206],
      [-0.08860647, 0.01838757, -1.37721026, 1.44742906],
      [ 1.30570745, -0.09988701, 0.27973360, -1.48555410]],

    [[ 0.92537606, -1.49781322, 0.89299929, -0.32056224],
     [ 0.65954101, -1.44866693, 1.15448976, -0.36536375],
     [ 0.66365826, -1.37158740, 1.19885933, -0.49093029]]])
```

5.2.5　自注意力模型与全连接、卷积、循环、图神经网络的不同

自注意力模型不仅适用于本章的自然语言处理任务，目前正进一步扩展到语音识别、图像识别以及生成式对抗网络(Generative Adversarial Networks，GAN)等。本节以下

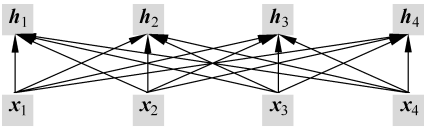
主要介绍自注意力模型与全连接、卷积、循环神经网络的不同，即自注意力模型在遵循一定约束条件下，可以转化为以下神经网络。

1. 自注意力模型与全连接神经网络

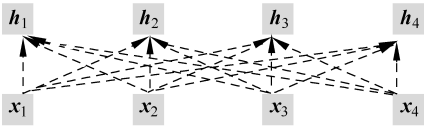
图 5-12 给出了全连接神经网络模型和自注意力模型的对比，其中实线表示可学习的权重，虚线表示动态生成的权重。由于自注意力模型的权重是动态生成的，因此可以处理变长的信息序列。

2. 自注意力模型与卷积神经网络模型

如图 5-13 所示，如果用自注意力机制处理图片，右下角像素(0)为 query，图片内其他像素为 key，我们能够将得到该像素与图中其他像素相关性的全局信息；如果用 CNN 处理图片，在感受野(Receptive Field)范围内将获得图片局部的信息。因此，我们可以得到以下比较结果：CNN 可以看作是一种简化版的 Self-Attention，因为 CNN 感受的是图片的局部信息，而 Self-Attention 获得整张图片的全局信息；反过来说，Self-Attention 是一个复杂化的 CNN。



(a) 全连接神经网络模型



(b) 自注意力模型

图 5-12　全连接神经网络模型和自注意力模型对比

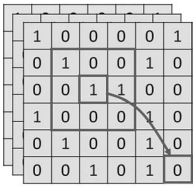


图 5-13　卷积神经网络模型和自注意力模型对比

两者另外一个区别是：CNN 感受野(卷积核)大小是人决定的，而 Self-Attention 的“全局感受野”是机器自动学习出来的。文献 *On the Relationship, between Self-Attention and Convolutional Layers* 用数学的方式证明了 CNN 就是 Self-Attention 的特例，Self-Attention 只要设定合适的参数，它可以做到跟 CNN 一模一样的事情。但 Self-Attention 与 CNN 相比，训练需要更多的样本数据，否则容易过拟合。

3. 自注意力模型与循环神经网络模型

如图 5-14 所示，RNN 接受输入是通过左边的 Memory 开始，从左到右串行得到其左边传来的信息，直到最右边才能得到整个输入序列的信息；而 Self-Attention 没有 RNN 这种问题，直接可以并行得到整个输入的全局信息。因此，近年来 RNN 正逐步被 Self-Attention 所取代。

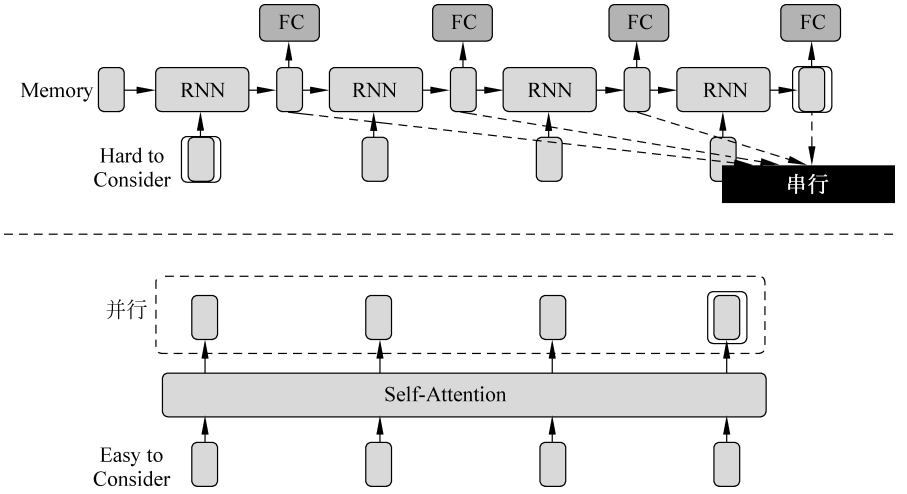


图 5-14　循环神经网络模型和自注意力模型对比

4. 自注意力模型与图神经网络模型

如图 5-15 所示，图神经网络的节点可以看成是输入向量，节点之间的边可认为是不同网络层次间的权重系数矩阵。利用 Self-Attention 的相关性在以上几个比较中是学习出来的，而对图其相关性体现在边上已经指定。因此，在 Self-Attention 的相关性矩阵中只要考虑相连节点的连接情况。例如，图中节点 1 和节点 8 有相连，那就只需要计算节点 1 和节点 8 两个向量之间的 Attention 分数（浅色球）；节点 7 和节点 8 果之间没有相连，说明两个节点之间没有关系，其 Attention 分数设置为 0。

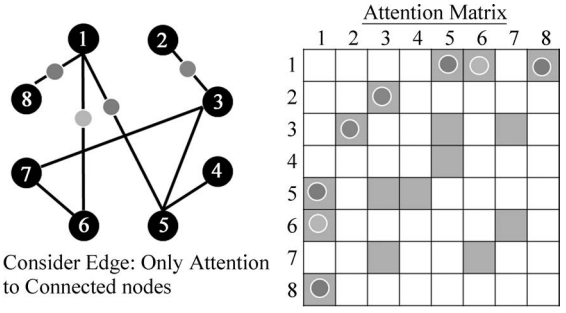


图 5-15　图神经网络模型和自注意力模型对比

5.3　案例：基于 seq2seq 的对联生成

对联，是汉族传统文化之一，是写在纸、布上或刻在竹子、木头、柱子上的对偶语句。对联对仗工整，平仄协调，是一字一音的汉语独特的艺术形式，是中国传统文化瑰宝。

对联生成是一个典型的序列到序列（sequence2sequence，seq2seq）建模的场景，编码器-解码器（Encoder-Decoder）框架是解决 seq2seq 问题的经典方法，它能够将一个任意长

度的源序列转换成另一个任意长度的目标序列：编码阶段将整个源序列编码成一个向量，解码阶段通过最大化预测序列概率，从中解码出整个目标序列。

5.3.1 方案设计

本案例的实现方案如图 5-16 所示，模型输入是对联上文文本，模型输出是对联下联文本。在模型构建时，需要先对输入的对联文本进行数据处理，生成规整的文本序列数据，包括语句分词、将词转换为 id、过长文本截断、过短文本填充等；然后使用双向 LSTM 对文本序列进行编码，获得文本的语义向量表示；然后使用带有 Attention 机制的双向 LSTM 对文本序列进行解码；最后经过全连接层和 Softmax 处理，得到文本下联的概率。

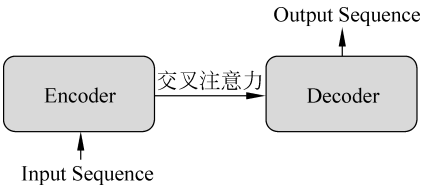


图 5-16 Encoder-Decoder 示意图

5.3.2 数据预处理

1. 数据集介绍


数据集网址

该案例采用开源的对联数据集 couplet-clean-dataset，该数据集过滤了 couplet-dataset 中的低俗、敏感内容。该数据集包含 70 万多条训练样本，1000 条验证样本和 1000 条测试样本。

下面列出一些训练集中对联样例：

上联：晚风摇树树还挺，下联：晨露润花花更红。

上联：愿景天成无墨迹，下联：万方乐奏有于阗。

上联：丹枫江冷人初去，下联：绿柳堤新燕复来。

上联：闲来野钓人稀处，下联：兴起高歌酒醉中。

2. 加载数据集

paddlenlp 中内置了对联数据集 couplet。获取该数据集可以调用 paddlenlp.datasets.load_dataset，传入 splits（“train”，“dev”，“test”），即可获取对应的 train_ds、dev_ds 和 test_ds。其中，train_ds 为训练集，用于模型训练；dev_ds 为开发集，也称验证集，用于模型参数调优；test_ds 为测试集，用于评估算法的性能，但不会根据测试集上的表现再去调整模型或参数。代码如下所示。

```
01. import io
02. import os
03. from functools import partial
04. import numpy as np
05. import paddle
06. import paddle.nn as nn
07. import paddle.nn.functional as F
```

```
from paddlenlp.data import Vocab, Pad
from paddlenlp.metrics import Perplexity
from paddlenlp.datasets import load_dataset

train_ds, test_ds = load_dataset('couplet', splits=('train', 'test'))

print(len(train_ds), len(test_ds))
for i in range(5):
  print(train_ds[i])

vocab = Vocab.load_vocabulary(**train_ds.vocab_info)
trg_idx2word = vocab.idx_to_token
vocab_size = len(vocab)

pad_id = vocab[vocab.eos_token]
bos_id = vocab[vocab.bos_token]
eos_id = vocab[vocab.eos_token]
print(pad_id, bos_id, eos_id)
```

3. 数据集文本转成 id

想将数据集文本转成 id(如图 5-17),需要实现一个 convert_example 函数,然后传入 map 函数,用 map 将带有文本的数据集转成带 id 的数据集。代码如下所示。

```
def convert_example(example, vocab):
  pad_id = vocab[vocab.eos_token]
  bos_id = vocab[vocab.bos_token]
  eos_id = vocab[vocab.eos_token]
  source = [bos_id] + vocab.to_indices(example['first'].split('\x02')) + [eos_id]
  target = [bos_id] + vocab.to_indices(example['second'].split('\x02')) + [eos_id]
  return source, target

trans_func = partial(convert_example, vocab=vocab)
train_ds = train_ds.map(trans_func, lazy=False)
test_ds = test_ds.map(trans_func, lazy=False)
```

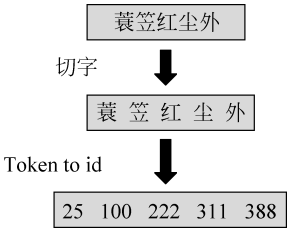


图 5-17　token to id 示意图

4. 构造 dataloder

模型训练前最后一个步骤是定义 create_data_loader()函数,实现数据成批次加载。其中 paddle.io.DataLoader 来创建训练和预测时所需要的 DataLoader 对象。paddle.io.DataLoader 返回一个迭代器,该迭代器根据 batch_sampler 指定的顺序迭代返回 dataset 数据。支持单进程或多进程加载数据,其函数参数如下:

- batch_sampler:批采样器实例,用于在 paddle.io.DataLoader 中迭代式获取 mini-batch 的样本下标数组,数组长度与 batch_size 一致。
- collate_fn:指定如何将样本列表组合为 mini-batch 数据。传给它参数需要是一个 callable 对象,需要实现对组建的 Batch 的处理逻辑,并返回每个 Batch 的数据。在这里传入的是 prepare_input 函数,对产生的数据进行 pad 操作,并返回实际长度等。

代码如下所示。

```
01. def create_data_loader(dataset):
02.   data_loader = paddle.io.DataLoader(
03.       dataset,
04.       batch_sampler=None,
05.       batch_size = batch_size,
06.       collate_fn=partial(prepare_input, pad_id=pad_id))
07.   return data_loader
08.
09. def prepare_input(insts, pad_id):
10.   src, src_length = Pad(pad_val=pad_id, ret_length=True)([inst[0] for inst in
   insts])
11.   tgt, tgt_length = Pad(pad_val=pad_id, ret_length=True)([inst[1] for inst in
   insts])
12.   tgt_mask = (tgt[:, :-1] != pad_id).astype(paddle.get_default_dtype())
13.   return src, src_length, tgt[:, :-1], tgt[:, 1:, np.newaxis], tgt_mask
14.
15. device = "gpu" # or cpu
16. device = paddle.set_device(device)
17.
18. batch_size = 128
19. num_layers = 2
20. dropout = 0.2
21. hidden_size =256
22. max_grad_norm = 5.0
23. learning_rate = 0.001
24. max_epoch = 20
25. model_path = './couplet_models'
26. log_freq = 200
27.
28. # Define dataloader
29. train_loader = create_data_loader(train_ds)
```

```
30. test_loader = create_data_loader(test_ds)
31.
32. print(len(train_ds), len(train_loader), batch_size)
33. # 702594 5490 128 共 5490 个 Batch
34.
35. for i in train_loader:
36.   print (len(i))
37.   for ind, each in enumerate(i):
38.       print (ind, each.shape)
39.   break
```

5.3.3　模型构建

1. 模型设计

图 5-18 是带有 Attention 的 Seq2Seq 模型结构。下面分别定义网络的每个部分，最后构建 Seq2Seq 主网络。

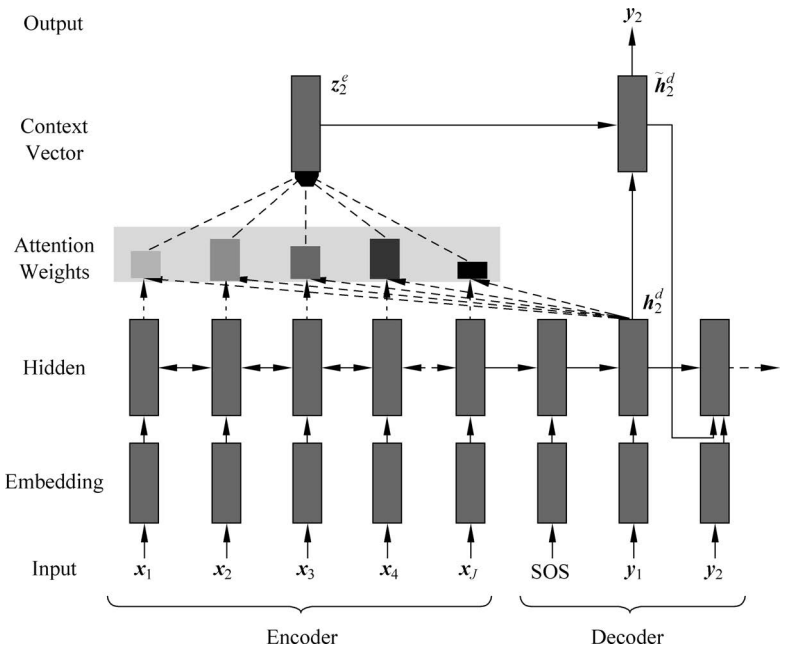


图 5-18　带有 Attention 机制的 Encoder-Decoder 原理示意图

2. 定义 Encoder

Encoder 部分非常简单，可以直接利用 PaddlePaddle2.0 提供的 RNN 系列 API：

(1) nn.Embedding：该接口用于构建 Embedding 的一个可调用对象，根据输入的 size (vocab_size, embedding_dim) 自动构造一个二维 Embedding 矩阵，用于 table-lookup。查表过程如图 5-19。

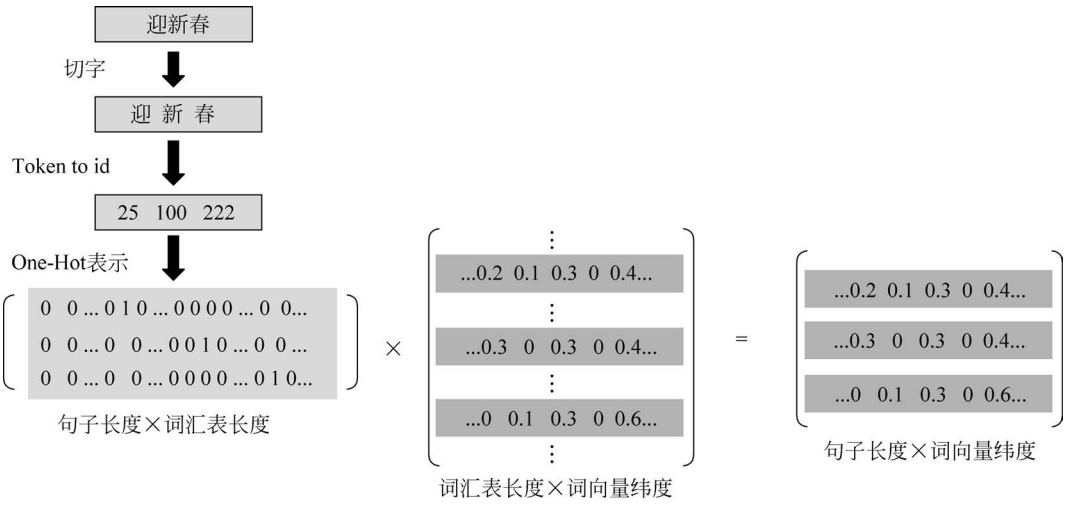


图 5-19 token to id 及查表获取向量示意图

(2) nn.LSTM：提供 LSTM 序列模型，得到 encoder_output 和 encoder_state。其输入和输出参数：

- input_size (int)输入的大小。
- hidden_size (int) -隐藏状态大小。
- num_layers (int,可选)-网络层数。默认为 1。
- direction (str,可选)-网络迭代方向，可设置为 forward 或 bidirect(或 bidirectional)。默认为 forward。
- time_major (bool,可选)-指定 input 的第一个维度是否是 time steps。默认为 False。
- dropout (float,可选)-dropout 概率，指的是出第一层外每层输入时的 dropout 概率。默认为 0。
- outputs (Tensor) -输出，由前向和后向 cell 的输出拼接得到。如果 time_major 为 True，则 Tensor 的形状为[time_steps,batch_size,num_directions * hidden_size]，如果 time_major 为 False，则 Tensor 的形状为[batch_size,time_steps,num_directions * hidden_size]，当 direction 设置为 bidirectional 时，num_directions 等于 2，否则等于 1。
- final_states (tuple) -最终状态，一个包含 $\boldsymbol{h}$ 和 $\boldsymbol{c}$ 的元组。形状为[num_lauers * num_directions, batch_size, hidden_size]，当 direction 设置为 bidirectional 时，num_directions 等于 2，否则等于 1。

代码如下所示。

```
01. class Seq2SeqEncoder(nn.Layer):
02.   def __init__(self, vocab_size, embed_dim, hidden_size, num_layers):
03.       super(Seq2SeqEncoder, self).__init__()
04.       self.embedder = nn.Embedding(vocab_size, embed_dim)
```

```
        self.lstm = nn.LSTM(
            input_size = embed_dim,
            hidden_size = hidden_size,
            num_layers = num_layers,
            dropout = 0.2 if num_layers > 1 else 0)

    def forward(self, sequence, sequence_length):
        inputs = self.embedder(sequence)
        encoder_output, encoder_state = self.lstm(
            inputs, sequence_length = sequence_length)

        # encoder_output [128, 18, 256] [batch_size, time_steps, hidden_size]
        # encoder_state (tuple) - 最终状态,一个包含 h 和 c 的元组.[2, 128, 256] [2, 128, 256] [num_layers * num_directions, batch_size, hidden_size]
        return encoder_output, encoder_state
```

3. 定义 AttentionLayer

AttentionLayer 层定义如下：

- nn.Linear 线性变换层传入 2 个参数：in_features (int)(线性变换层输入单元的数目)、out_features (int)(线性变换层输出单元的数目)。
- paddle.matmul 用于计算两个 Tensor 的乘积，遵循完整的广播规则。其参数定义如下：x(Tensor)：输入变量，类型为 Tensor，数据类型为 float32、float64；y(Tensor)：输入变量，类型为 Tensor，数据类型为 float32、float64；
- transpose_x (bool，可选)：相乘前是否转置 $\boldsymbol{x}$，默认值为 False；
- transpose_y (bool，可选)：相乘前是否转置 $\boldsymbol{y}$，默认值为 False。
- paddle.unsqueeze 用于向输入 Tensor 的 Shape 中一个或多个位置(axis)插入尺寸为 1 的维度
- paddle.add 逐元素相加算子，输入 $\boldsymbol{x}$ 与输入 $\boldsymbol{y}$ 逐元素相加，并将各个位置的输出元素保存到返回结果中。

具体代码实现如下所示。

```
class AttentionLayer(nn.Layer):
    def __init__(self, hidden_size):
        super(AttentionLayer, self).__init__()
        self.input_proj = nn.Linear(hidden_size, hidden_size)
        self.output_proj = nn.Linear(hidden_size + hidden_size, hidden_size)

    def forward(self, hidden, encoder_output, encoder_padding_mask):
        encoder_output = self.input_proj(encoder_output)
        attn_scores = paddle.matmul(
            paddle.unsqueeze(hidden, [1]), encoder_output, transpose_y = True)
        # print('attention score', attn_scores.shape) #[128, 1, 18]

```

```
        if encoder_padding_mask is not None:
            attn_scores = paddle.add(attn_scores, encoder_padding_mask)

        attn_scores = F.Softmax(attn_scores)
        attn_out = paddle.squeeze(
            paddle.matmul(attn_scores, encoder_output), [1])
        # print('1 attn_out', attn_out.shape) #[128, 256]

        attn_out = paddle.concat([attn_out, hidden], 1)
        # print('2 attn_out', attn_out.shape) #[128, 512]

        attn_out = self.output_proj(attn_out)
        # print('3 attn_out', attn_out.shape) #[128, 256]
        return attn_out
```

4. 定义 Seq2SeqDecoder 解码器

首先，由于 Decoder 部分是带有 attention 的 LSTM，不能复用 nn.LSTM，所以需要定义 Seq2SeqDecoderCell，其中 nn.LayerList 用于保存子层列表。其次，在构建 Seq2SeqDecoder 时，paddle.nn.RNN 是循环神经网络（RNN）的封装，将输入的 Seq2SeqDecoderCell 封装为带注意力机制的双向长短记忆神经网络。代码如下所示。

```
class Seq2SeqDecoderCell(nn.RNNCellBase):
    def __init__(self, num_layers, input_size, hidden_size):
        super(Seq2SeqDecoderCell, self).__init__()
        self.dropout = nn.Dropout(0.2)
        self.lstm_cells = nn.LayerList([
            nn.LSTMCell(
                input_size=input_size + hidden_size if i == 0 else hidden_size,
                hidden_size=hidden_size) for i in range(num_layers)
        ])

        self.attention_layer = AttentionLayer(hidden_size)

    def forward(self,
                step_input,
                states,
                encoder_output,
                encoder_padding_mask=None):
        lstm_states, input_feed = states
        new_lstm_states = []
        step_input = paddle.concat([step_input, input_feed], 1)
        for i, lstm_cell in enumerate(self.lstm_cells):
            out, new_lstm_state = lstm_cell(step_input, lstm_states[i])
            step_input = self.dropout(out)
            new_lstm_states.append(new_lstm_state)
        out = self.attention_layer(step_input, encoder_output,
```

```
                            encoder_padding_mask)
        return out, [new_lstm_states, out]

class Seq2SeqDecoder(nn.Layer):
    def __init__(self, vocab_size, embed_dim, hidden_size, num_layers):
        super(Seq2SeqDecoder, self).__init__()
        self.embedder = nn.Embedding(vocab_size, embed_dim)
        self.lstm_attention = nn.RNN(
            Seq2SeqDecoderCell(num_layers, embed_dim, hidden_size))
        self.output_layer = nn.Linear(hidden_size, vocab_size)

    def forward(self, trg, decoder_initial_states, encoder_output,
                encoder_padding_mask):
        inputs = self.embedder(trg)

        decoder_output, _ = self.lstm_attention(
            inputs,
            initial_states=decoder_initial_states,
            encoder_output=encoder_output,
            encoder_padding_mask=encoder_padding_mask)
        predict = self.output_layer(decoder_output)

        return predict
```

5. 构建基于 seq2seq 的对联生成模型

根据以上步骤的定义，最后构建基于 seq2seq 的对联生成模型如下代码所示。

```
class Seq2SeqAttnModel(nn.Layer):
    def __init__(self, vocab_size, embed_dim, hidden_size, num_layers,
                 eos_id=1):
        super(Seq2SeqAttnModel, self).__init__()
        self.hidden_size = hidden_size
        self.eos_id = eos_id
        self.num_layers = num_layers
        self.INF = 1e9
        self.encoder = Seq2SeqEncoder(vocab_size, embed_dim, hidden_size,
                                      num_layers)
        self.decoder = Seq2SeqDecoder(vocab_size, embed_dim, hidden_size,
                                      num_layers)

    def forward(self, src, src_length, trg):
        # encoder_output 各时刻的输出 h
        # encoder_final_state 最后时刻的输出 h,和记忆信号 c
        encoder_output, encoder_final_state = self.encoder(src, src_length)
        print('encoder_output shape', encoder_output.shape) # [128, 18, 256] [batch_
    size,time_steps,hidden_size]
```

```
        print('encoder_final_states shape', encoder_final_state[0].shape, encoder_final_state[1].shape) #[2, 128, 256] [2, 128, 256] [num_lauers * num_directions, batch_size, hidden_size]

        # Transfer shape of encoder_final_states to [num_layers, 2, batch_size, hidden_size]
        encoder_final_states = [
          (encoder_final_state[0][i], encoder_final_state[1][i])
           for i in range(self.num_layers)
      ]
        print('encoder_final_states shape', encoder_final_states[0][0].shape, encoder_final_states[0][1].shape) #[128, 256] [128, 256]


        # Construct decoder initial states: use input_feed and the shape is
        # [[h,c] * num_layers, input_feed], consistent with Seq2SeqDecoderCell.states
        decoder_initial_states = [
           encoder_final_states,
           self.decoder.lstm_attention.cell.get_initial_states(
              batch_ref = encoder_output, shape = [self.hidden_size])
      ]

        # Build attention mask to avoid paying attention on padddings
        src_mask = (src != self.eos_id).astype(paddle.get_default_dtype())
        print ('src_mask shape', src_mask.shape) #[128, 18]
        print(src_mask[0, :])

        encoder_padding_mask = (src_mask - 1.0) * self.INF
        print ('encoder_padding_mask', encoder_padding_mask.shape) #[128, 18]
        print(encoder_padding_mask[0, :])

        encoder_padding_mask = paddle.unsqueeze(encoder_padding_mask, [1])
        print('encoder_padding_mask', encoder_padding_mask.shape) #[128, 1, 18]

        predict = self.decoder(trg, decoder_initial_states, encoder_output,
                               encoder_padding_mask)
        print('predict', predict.shape)   #[128, 17, 7931]

        return predict
```

5.3.4 训练配置和训练

1. 定义损失函数

本项目的交叉熵损失函数需要将 padding 位置的 Loss 置为 0，因此需要在损失函数中引入 trg_mask 参数，由于 PaddlePaddle 框架提供的 paddle.nn.CrossEntropyLoss 没有 trg_mask 参数，因此需要重新定义。代码如下所示。

```
class CrossEntropyCriterion(nn.Layer):
  def __init__(self):
      super(CrossEntropyCriterion, self).__init__()

  def forward(self, predict, label, trg_mask):
      cost = F.Softmax_with_cross_entropy(
         logits = predict, label = label, soft_label = False)
      cost = paddle.squeeze(cost, axis = [2])
      masked_cost = cost * trg_mask
      batch_mean_cost = paddle.mean(masked_cost, axis = [0])
      seq_cost = paddle.sum(batch_mean_cost)

      return seq_cost
```

2. 模型训练

本节使用高层 API 执行训练，需要调用 prepare 函数和 fit 函数。在 prepare 函数中，需配置优化器、损失函数，以及评价指标。其中，评价指标采用的是 PaddleNLP 提供的困惑度计算 API 函数（paddlenlp.metrics.Perplexity）。

如果安装了 VisualDL，可以在 fit 中添加一个 callbacks 参数使用 VisualDL 观测训练过程，代码如下所示。

```
model.fit(train_data = train_loader,
epochs = max_epoch,
          eval_freq = 1,
          save_freq = 1,
          save_dir = model_path,
          log_freq = log_freq,
          callbacks = [paddle.callbacks.VisualDL('
./log')])
```

在这里，由于对联生成任务没有明确的评价指标，因此可以在保存的多个模型中，通过人工评判生成结果选择最好的模型。本项目中，为了便于演示，已经将训练好的模型参数载入模型，并省略了训练过程。读者自己实验的时候，可以尝试自行修改超参数，调用下面被注释掉的 fit 函数，重新进行训练。如果读者想要在更短的时间内得到效果不错的模型，可以尝试在模型前使用词向量技术。

```
model = paddle.Model(
  Seq2SeqAttnModel(vocab_size, hidden_size, hidden_size,
                       num_layers, pad_id))

optimizer = paddle.optimizer.Adam(
  learning_rate = learning_rate, parameters = model.parameters())
ppl_metric = Perplexity()
```

```
model.prepare(optimizer, CrossEntropyCriterion(), ppl_metric)

model.fit(train_data = train_loader,
          epochs = max_epoch,
          eval_freq = 1,
          save_freq = 1,
          save_dir = model_path,
          log_freq = log_freq)
```

5.3.5 模型推理

1. 定义预测网络 Seq2SeqAttnInferModel

根据对联生成模型 Seq2SeqAttnModel，可实现定义子类 Seq2SeqAttnInferModel，实现代码如下所示。

```
class Seq2SeqAttnInferModel(Seq2SeqAttnModel):
  def __init__(self,
               vocab_size,
               embed_dim,
                hidden_size,
                num_layers,
                bos_id = 0,
                eos_id = 1,
                beam_size = 4,
                max_out_len = 256):
      self.bos_id = bos_id
      self.beam_size = beam_size
      self.max_out_len = max_out_len
      self.num_layers = num_layers
      super(Seq2SeqAttnInferModel, self).__init__(
          vocab_size, embed_dim, hidden_size, num_layers, eos_id)

      # Dynamic decoder for inference
      self.beam_search_decoder = nn.BeamSearchDecoder(
          self.decoder.lstm_attention.cell,
          start_token = bos_id,
          end_token = eos_id,
          beam_size = beam_size,
          embedding_fn = self.decoder.embedder,
          output_fn = self.decoder.output_layer)

  def forward(self, src, src_length):
      encoder_output, encoder_final_state = self.encoder(src, src_length)

      encoder_final_state = [
         (encoder_final_state[0][i], encoder_final_state[1][i])
```

```
            for i in range(self.num_layers)
        ]

        # Initial decoder initial states
        decoder_initial_states = [
            encoder_final_state,
            self.decoder.lstm_attention.cell.get_initial_states(
                batch_ref=encoder_output, shape=[self.hidden_size])
        ]
        # Build attention mask to avoid paying attention on paddings
        src_mask = (src != self.eos_id).astype(paddle.get_default_dtype())

        encoder_padding_mask = (src_mask - 1.0) * self.INF
        encoder_padding_mask = paddle.unsqueeze(encoder_padding_mask, [1])

        # Tile the batch dimension with beam_size
        encoder_output = nn.BeamSearchDecoder.tile_beam_merge_with_batch(
            encoder_output, self.beam_size)
        encoder_padding_mask = nn.BeamSearchDecoder.tile_beam_merge_with_batch(
            encoder_padding_mask, self.beam_size)

        # Dynamic decoding with beam search
        seq_output, _ = nn.dynamic_decode(
            decoder=self.beam_search_decoder,
            inits=decoder_initial_states,
            max_step_num=self.max_out_len,
            encoder_output=encoder_output,
            encoder_padding_mask=encoder_padding_mask)
        return seq_output
```

2. 解码部分

常规的搜索方法有贪心(Greedy Search)、穷举(Exhaustive Search)和束搜索(Beam Search)。

- 穷举：穷举所有可能的输出结果。例如，输出序列长度为3，候选项为4，那么就有$4^3=64$种可能，当输出序列长度为10时，就会有4^{10}种可能，这种幂级的增长对于计算机性能的要求是极高的，耗时耗力。
- 贪心：每次选择概率最大的候选者作为输出。搜索空间小，以局部最优解期望全局最优解，无法保证最终结果是做优的，但是效率高。
- 束搜索：束搜索可以看作是穷举和贪心的折中方案。需要设定一个束宽（Beam Size），当设为1时即为贪心，当设为候选项的数量时即为穷举。

束搜索是一种启发式图搜索算法，具有更大的搜索空间，可以减少遗漏隐藏在低概率单词后面的高概率单词的可能性，他会在每步保持最可能的束宽个假设，最后选出整体概

率最高的假设。图 5-20 以束宽 2 为例说明了其搜索过程。从图 5-20 中可以看到,在第一步的时候,我们除了选择概率最高的“机”字以外,还保留了概率第二高的“桨”字。在第二步的时候两个 beam 分别选择了“起”和“框”。这时我们发现“飞机快”这一序列的概率为 0.2,而“飞桨框”序列的概率为 0.32。我们找到了整体概率更高的序列。在我们这个示例中继续解下去,得到的最终结果为“飞桨框架”。代码如下所示。

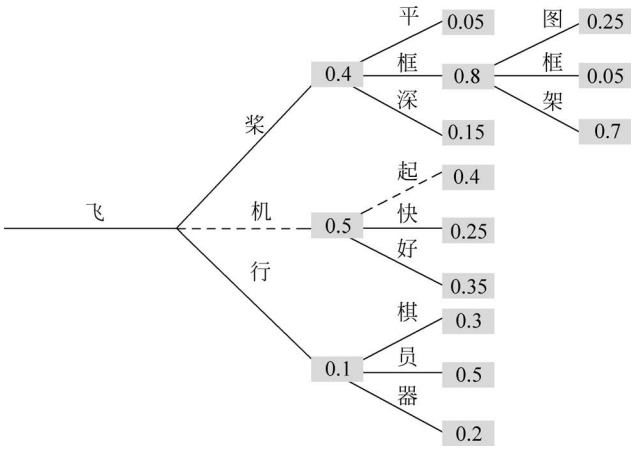


图 5-20　束搜索选择过程

```
def post_process_seq(seq, bos_idx, eos_idx, output_bos = False, output_eos = False):
    """
    Post - process the decoded sequence.
    """
    eos_pos = len(seq) - 1
    for i, idx in enumerate(seq):
        if idx == eos_idx:
            eos_pos = i
            break
    seq = [
        idx for idx in seq[:eos_pos + 1]
        if (output_bos or idx != bos_idx) and (output_eos or idx != eos_idx)
    ]
    return seq

beam_size = 10
model = paddle.Model(
    Seq2SeqAttnInferModel(
        vocab_size,
        hidden_size,
        hidden_size,
        num_layers,
        bos_id = bos_id,
        eos_id = eos_id,
```

```
        beam_size = beam_size,
        max_out_len = 256))

model.prepare()
```

3. 预测下联

在预测之前，我们需要将训练好的模型参数使用 load 方法输入到预测网络，之后就可以根据对联的上联生成对联的下联。代码如下所示。

```
model.load('couplet_models/model_18')

idx = 0
for data in test_loader():
    inputs = data[:2]
    finished_seq = model.predict_batch(inputs = list(inputs))[0]
    finished_seq = finished_seq[:, :, np.newaxis] if len(
        finished_seq.shape) == 2 else finished_seq
    = np.transpose(finished_seq, [0, 2, 1])
   for ins in finished_seq:
        for beam in ins:
            id_list = post_process_seq(beam, bos_id, eos_id)
            word_list_f = [trg_idx2word[id] for id in test_ds[idx][0]][1:-1]
            word_list_s = [trg_idx2word[id] for id in id_list]
            sequence = "上联: " + "".join(word_list_f) + "\t 下联: " + "".join(word_
list_s) + "\n"
            print(sequence)
            idx += 1
            break
```

5.4　本章小结

本章首先介绍了对联生成任务，并说明了对联生成任务的现状以及难点。其次，介绍了注意力机制、自注意力机制和 Transformer 模型。最后，给出实践案例说明了使用 PaddlePaddle 构建基于注意力机制的对联生成模型，并实现对联生成。请读者关注 5.2.5 节的自注意力模型与全连接、卷积、循环、图神经网络的不同，并能够融会贯通地理解前几章介绍的模型。