学习笔记-4: Task 4

10 - 机器翻译及相关技术

机器翻译和数据集

机器翻译

将一段文本从一种语言自动翻译为另一种语言，用神经网络解决这个问题通常称为神经机器翻译 (NMT)
主要特征：输出是单词序列而不是单个单词
主要困难：输出序列的长度可能与源序列的长度不同

数据预处理

将数据集清洗、转化为神经网络的输入mini-batch

分词

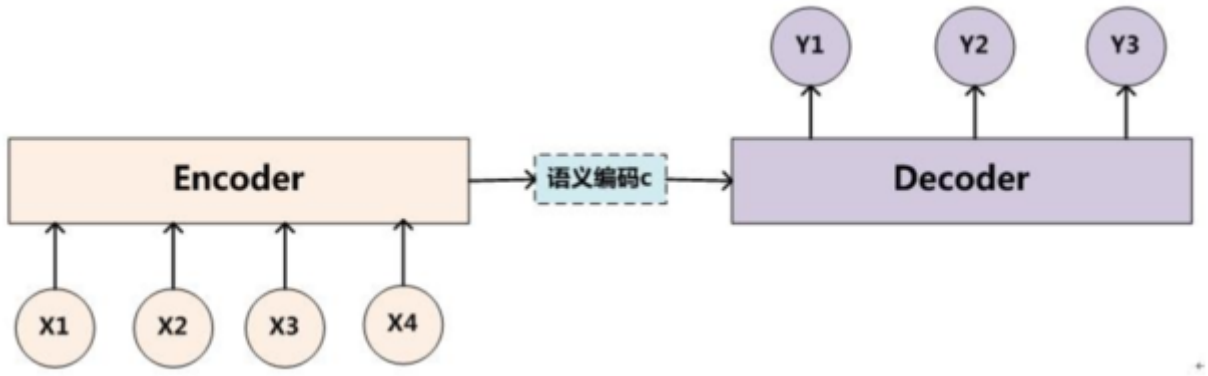
字符串 ⟶ 单词组成的列表

建立词典

单词组成的列表 ⟶ 单词id组成的列表

Encoder-Decoder

Encoder：输入到隐藏状态
Decoder：隐藏状态到输出
通过Encoder，将输入转化成为一个隐藏状态，并把它作为语义编码的输入 解决输入输出序列长度不同的问题



代码实现

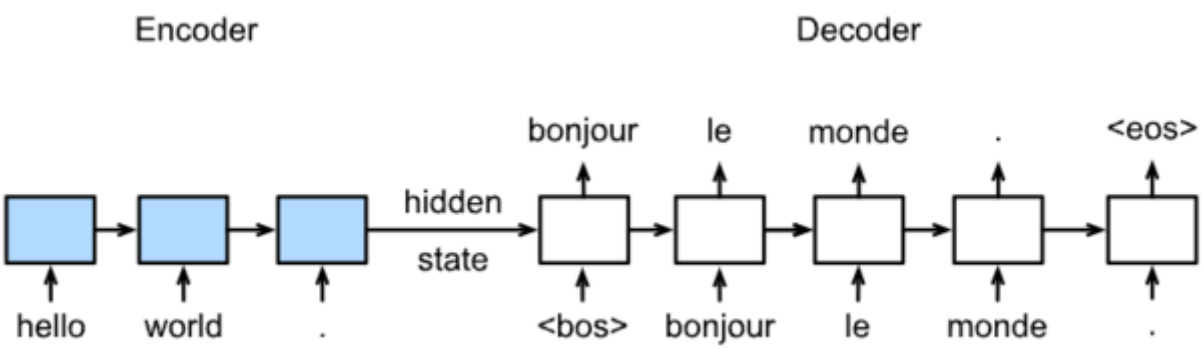
```
class Encoder(nn.Module):
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
    """
```

```
        预留forward方法接口不实现，在其子类中实现；
        并且要求其子类一定要实现，不实现的时候会导致问题；
        这里产生的问题是NotImplementedError。
        """
            raise NotImplementedError

    class Decoder(nn.Module):
        def __init__(self, **kwargs):
            super(Decoder, self).__init__(**kwargs)

        # 这里raise Error的目的和Encoder中的相同
        def init_state(self, enc_outputs, *args):
            raise NotImplementedError

        def forward(self, X, state):
            raise NotImplementedError

    class EncoderDecoder(nn.Module):
        def __init__(self, encoder, decoder, **kwargs):
            super(EncoderDecoder, self).__init__(**kwargs)
            self.encoder = encoder
            self.decoder = decoder

        def forward(self, enc_X, dec_X, *args):
            enc_outputs = self.encoder(enc_X, *args)
            # 需要在Decoder的子类中定义init_state方法，才能调用decoder实例的init_state方
    法
            dec_state = self.decoder.init_state(enc_outputs, *args)
            return self.decoder(dec_X, dec_state)
```

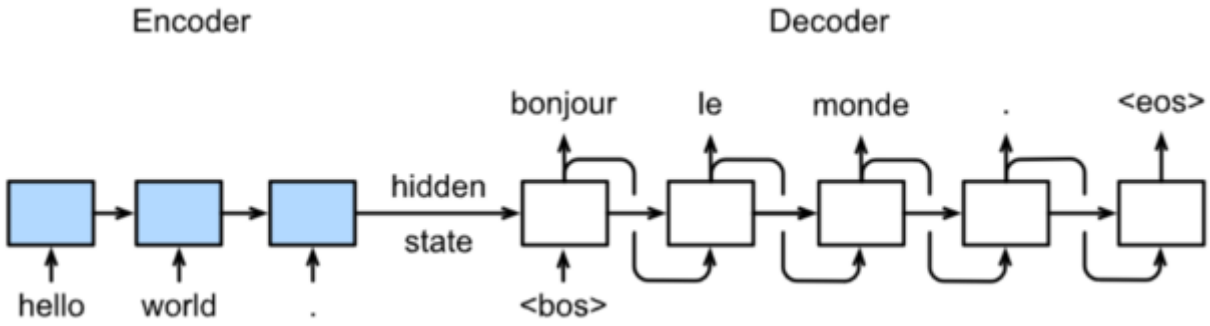
Sequence to Sequence模型

模型：

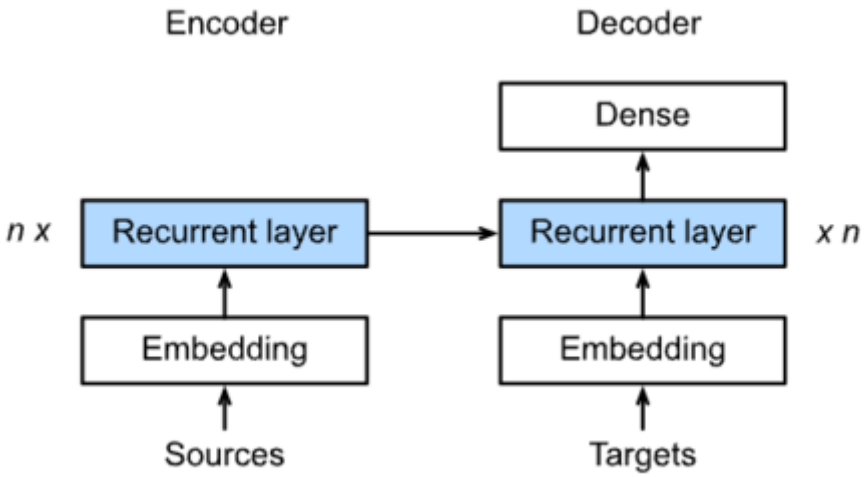
训练



预测



具体结构：



Encoder

Sources是一个列表，经过Embedding层之后得到词向量

```
class Seq2SeqEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.LSTM(embed_size,num_hiddens, num_layers, dropout=dropout)

    def begin_state(self, batch_size, device):
        return [torch.zeros(size=(self.num_layers, batch_size, self.num_hiddens),
device=device),
                torch.zeros(size=(self.num_layers, batch_size, self.num_hiddens),
device=device)]
    def forward(self, X, *args):
        X = self.embedding(X) # X shape: (batch_size, seq_len, embed_size)
        # RNN的输入必须是时序的tensor，X的第一个维度应该是句子中单词的顺序，即seq_len
        X = X.transpose(0, 1)  # RNN needs first axes to be time
        # state = self.begin_state(X.shape[1], device=X.device)
```

```
        out, state = self.rnn(X)
        """
        The shape of out is (seq_len, batch_size, num_hiddens);
        state contains the hidden state and the memory cell
        of the last time step,
        the shape of state is (num_layers, batch_size, num_hiddens).
        """
        return out, state
```

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
num_layers=2)
X = torch.zeros((4, 7), dtype = torch.long)  # (batch_size, seq_len)
output, state = encoder(X)

output.shape
>>> torch.Size([7, 4, 16])  # (seq_len, batch_size, num_hiddens)

len(state)
>>> 2  # 因为state包含hidden state和memory cell两项

state[0].shape
>>> torch.Size([2, 4, 16])  # (num_layers, batch_size, num_hiddens)

state[1].shape
>>> torch.Size([2, 4, 16])  # (num_layers, batch_size, num_hiddens)
```

Decoder

```
class Seq2SeqDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.LSTM(embed_size,num_hiddens, num_layers, dropout=dropout)
        # dense层，把num_hiddens大小的向量转化为vocab_size的向量，以便输出预测的单词
        self.dense = nn.Linear(num_hiddens,vocab_size)

    def init_state(self, enc_outputs, *args):
        # decoder初始状态设置为encoder返回的state，包含最后一步的hidden state和memory
cell
        return enc_outputs[1]

    def forward(self, X, state):
        # 转置，目的同encoder中的X.transpose(0, 1)
        X = self.embedding(X).transpose(0, 1)
        out, state = self.rnn(X, state)
        # Make the batch to be the first dimension to simplify loss computation.
        out = self.dense(out).transpose(0, 1)  # 刚刚转置过，再颠倒回来，现在out是
```

```
(batch_size, seq_len, vocab_size)
        return out, state
```

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8,num_hiddens=16, num_layers=2)
state = decoder.init_state(encoder(X))  # X: (batch_size, seq_len)
out, state = decoder(X, state)

out.shape
>>> torch.Size([4, 7, 10])  # (batch_size, seq_len, vocab_size)

len(state)
>>> 2  # 因为state包含hidden state和memory cell两项

state[0].shape
>>> torch.Size([2, 4, 16])  # (num_layers, batch_size, num_hiddens)

state[1].shape
>>> torch.Size([2, 4, 16])  # (num_layers, batch_size, num_hiddens)
```

损失函数

计算损失的时候要屏蔽之前padding的部分，通过SequenceMask函数实现

```
def SequenceMask(X, X_len, value = 0):
    """
    X: 待处理的tensor
    X_len: 该tensor的有效长度
    value: 用来替换padding部分的值, 默认为0
    """
    maxlen = X.size(1)
    # [None, :]和[:, None]都是用来扩展tensor维度的
    mask = torch.arange(maxlen)[None, :].to(X_len.device) < X_len[:, None]
    X[~mask] = value
    return X
```

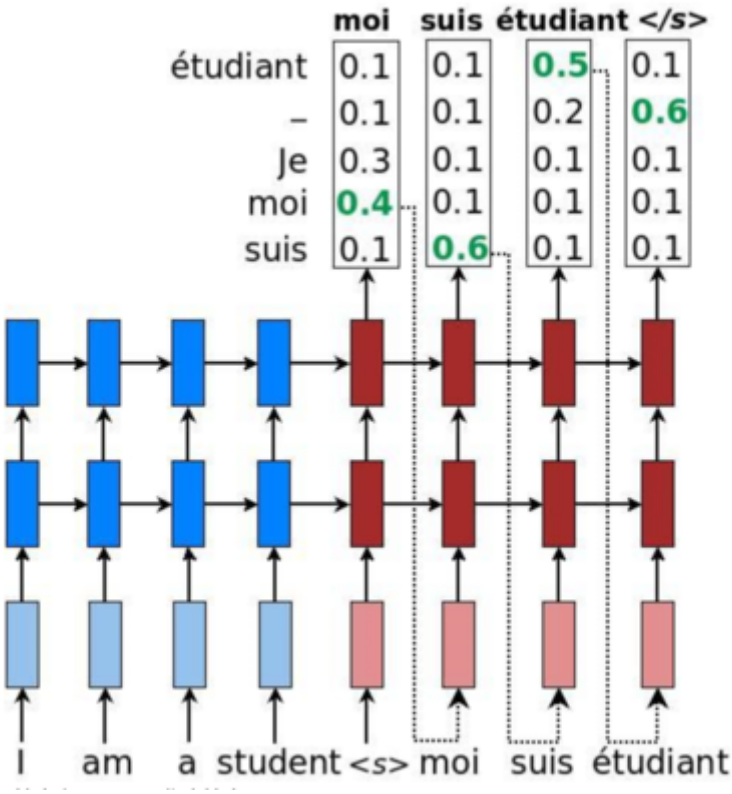
```
X = torch.tensor([[1,2,3], [4,5,6]])
SequenceMask(X, torch.tensor([1,2]))
>>> tensor([[1, 0, 0],
        [4, 5, 0]])
```

```
# 改写nn.CrossEntropyLoss的forward函数, 加入Mask操作
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
    # pred shape: (batch_size, seq_len, vocab_size)
    # label shape: (batch_size, seq_len)
```

```
    # valid_length shape: (batch_size, )
    def forward(self, pred, label, valid_length):
        # the sample weights shape should be (batch_size, seq_len)
        weights = torch.ones_like(label)
        weights = SequenceMask(weights, valid_length).float()
        # 现在weights是一个只有有效位置为1，其余位置均为0，形状和label一致的tensor
        self.reduction = 'none'
        output = super(MaskedSoftmaxCELoss, self).forward(pred.transpose(1,2),
label)
        return (output*weights).mean(dim=1)
```

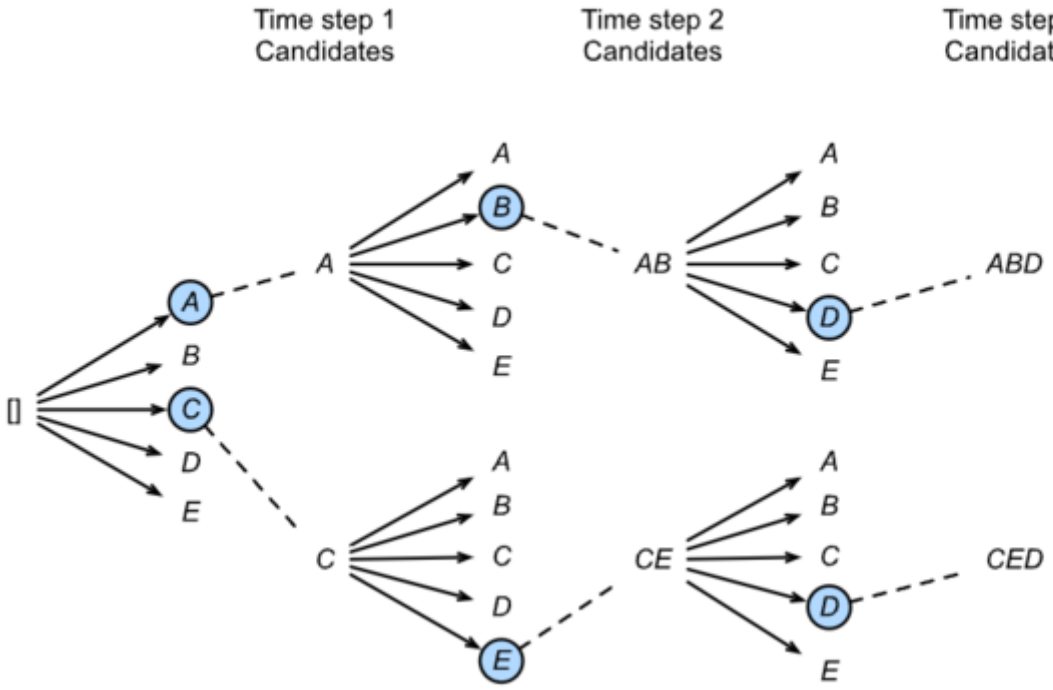
Beam Search

简单greedy search：



维特比算法：选择整体分数最高的句子 (搜索空间太大)

集束搜索：



上图中 `beam_size = 2`

11 - 注意力机制和Seq2seq模型

注意力机制

编码器—解码器(seq2seq)结构存在着问题，尤其是RNN机制实际中存在长程梯度消失的问题，对于较长的句子，我们很难寄希望于将输入的序列转化为定长的向量而保存所有的有效信息，所以随着所需翻译句子的长度的增加，这种结构的效果会显著下降。

与此同时，解码的目标词语可能只与原输入的部分词语有关，而并不是与所有的输入有关。例如，当把“Hello world”翻译成“Bonjour le monde”时，“Hello”映射成“Bonjour”，“world”映射成“monde”。在Seq2seq模型中，解码器只能隐式地从编码器的最终状态中选择相应的信息。然而，注意力机制可以将这种选择过程显式地建模。

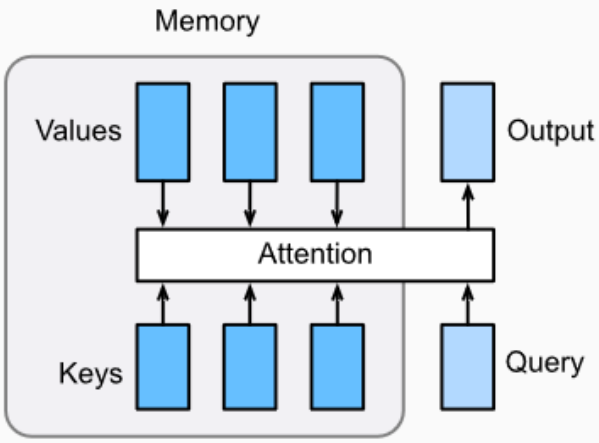


Fig. 10.1.1 The attention layer returns an output based on the input query and its memory.

注意力机制框架

Attention是一种通用的带权池化方法，输入由两部分构成：询问(query)和键值对(key-value pairs)。$\mathbf{k}_i \in \mathbb{R}^{d_k}, \mathbf{v}_i \in \mathbb{R}^{d_v}, \mathbf{q} \in \mathbb{R}^{d_q}$，attention layer得到输出与value的维度一致 $\mathbf{o} \in \mathbb{R}^{d_v}$。对于一个query来说，attention layer会对每一个key计算注意力分数并进行权重的归一化，输出的向量$\boldsymbol{o}$则是value的加权求和，而每个key计算的权重与value一一对应。

为了计算输出，我们首先假设有一个函数α用于计算query和key的相似性，然后可以计算所有的attention scores $a_1, \ldots, a_n$ by

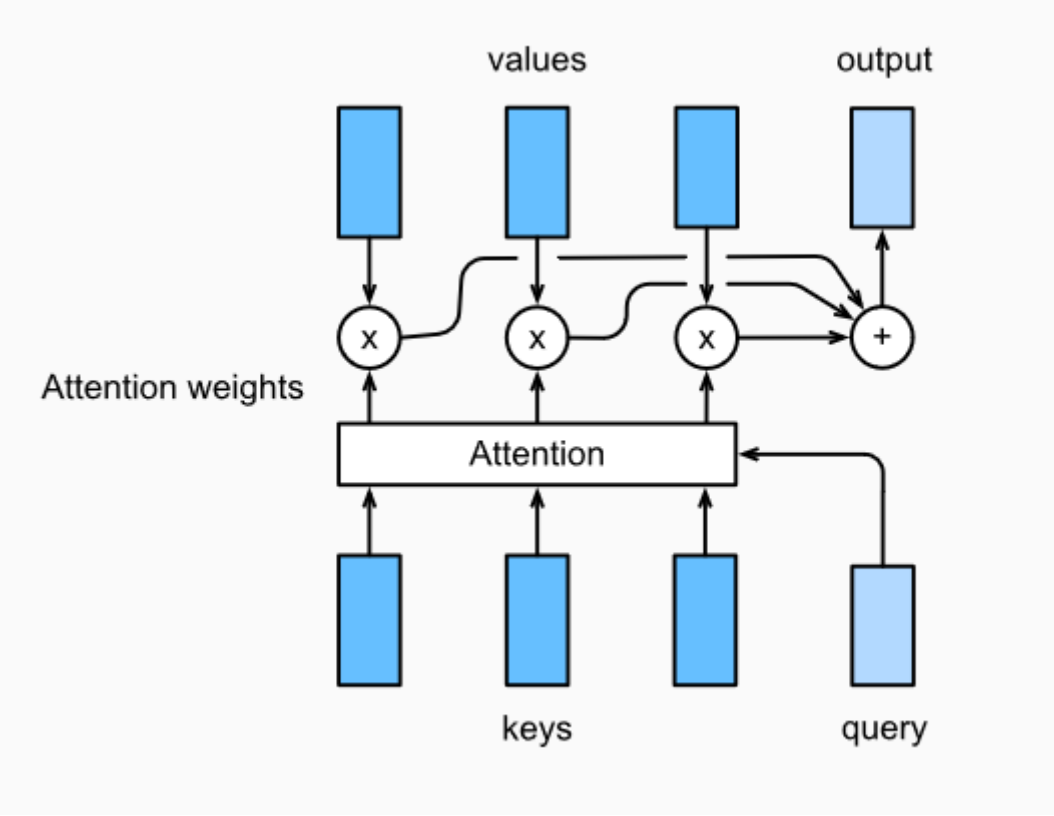
$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i)$$

我们使用softmax函数获得注意力权重：

$$b_1, \ldots, b_n = \text{softmax}(a_1, \ldots, a_n)$$

最终的输出就是value的加权求和：

$$\mathbf{o} = \sum_{i=1}^{n} b_i \mathbf{v}_i$$



不同的attetion layer选择的score函数不同

超出2维矩阵的乘法

X 和 Y 是维度分别为 (b, n, m) 和 (b, m, k) 的张量，进行 b 次二维矩阵乘法后得到 Z, 维度为 (b, n, k)

$$Z[i,:,:] = dot(X[i,:,:], Y[i,:,:]) \qquad for \quad i = 1, 2, \cdots n$$

点积注意力

The dot product 假设query和keys有相同的维度, 即 $\forall i, \mathbf{q}, \mathbf{k}_i \in \mathbb{R}_d$ 。通过计算query和key转置的乘积来计算attention score，通常还会除去 $\sqrt{d}$ 减少计算出来的score对维度d的依赖性，如下

$$\alpha(\mathbf{q}, \mathbf{k}) = \langle \mathbf{q}, \mathbf{k} \rangle / \sqrt{d}$$

假设 $\mathbf{Q} \in \mathbb{R}^{m \times d}$ 有 m 个query，$\mathbf{K} \in \mathbb{R}^{n \times d}$ 有 n 个keys。我们可以通过矩阵运算的方式计算所有 mn 个score：

$$\alpha(\mathbf{Q}, \mathbf{K}) = \mathbf{Q}\mathbf{K}^T / \sqrt{d}$$

多层感知机注意力

在多层感知器中，我们首先将query and key投影到 $\mathbb{R}^h$。为了更具体，我们将可以学习的参数做如下映射

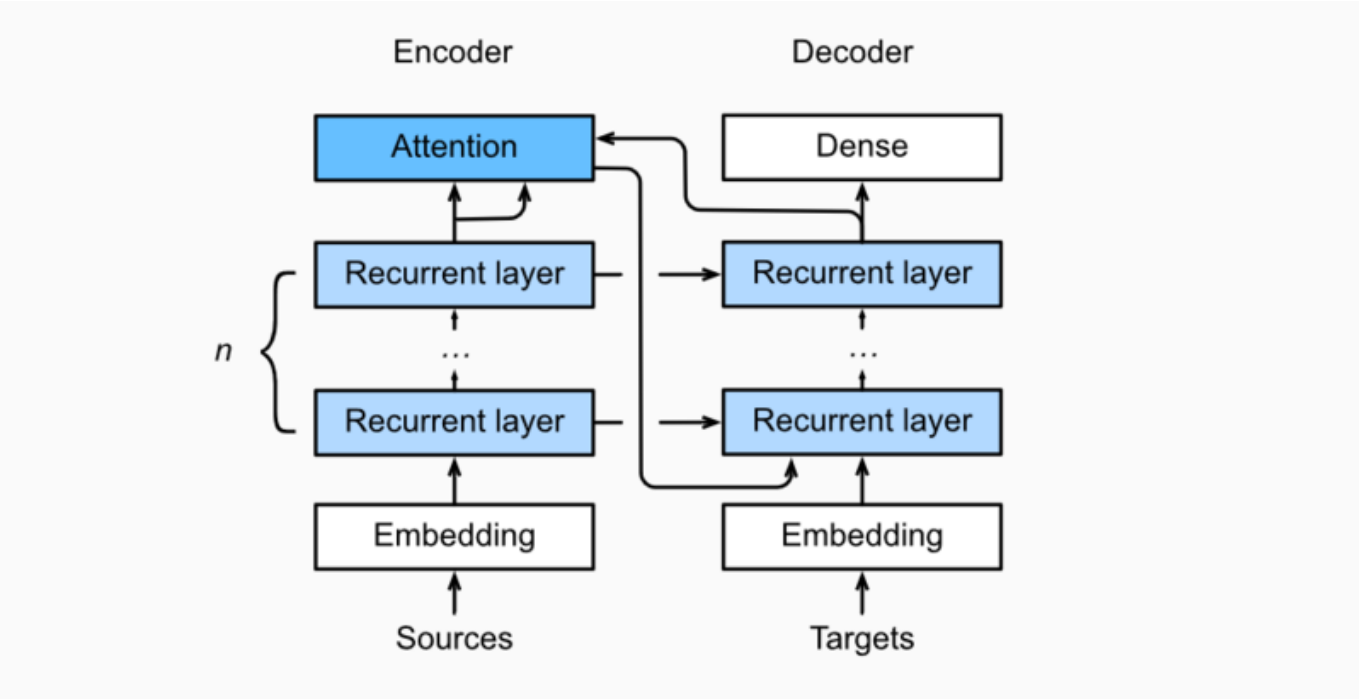
$$\mathbf{W}_k \in \mathbb{R}^{h \times d_k}, \ \mathbf{W}_q \in \mathbb{R}^{h \times d_q}, \ \mathbf{v} \in \mathbb{R}^h$$

将score函数定义为

$$\alpha(\mathbf{k}, \mathbf{q}) = \mathbf{v}^T tanh(\mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q})$$

然后将key和query在特征的维度上合并(concatenate)，然后送至 a single hidden layer perceptron，这层中hidden layer为 h，输出的size为 1。隐层激活函数为tanh，无偏置。

引入注意力机制的Seq2seq模型



带有注意力机制的Seq2seq的编码器与之前章节中的Seq2SeqEncoder相同；解码器中添加了一个MLP注意层(MLPAttention)，它的隐藏大小与解码器中的LSTM层相同。
以下三个参数用来初始化解码器的状态:

- the encoder outputs of all timesteps：encoder输出的各个状态，被用于attetion layer的memory部分，有相同的key和value
- the hidden state of the encoder's final timestep：编码器最后一个时间步的隐藏状态，被用于初始化decoder的hidden state

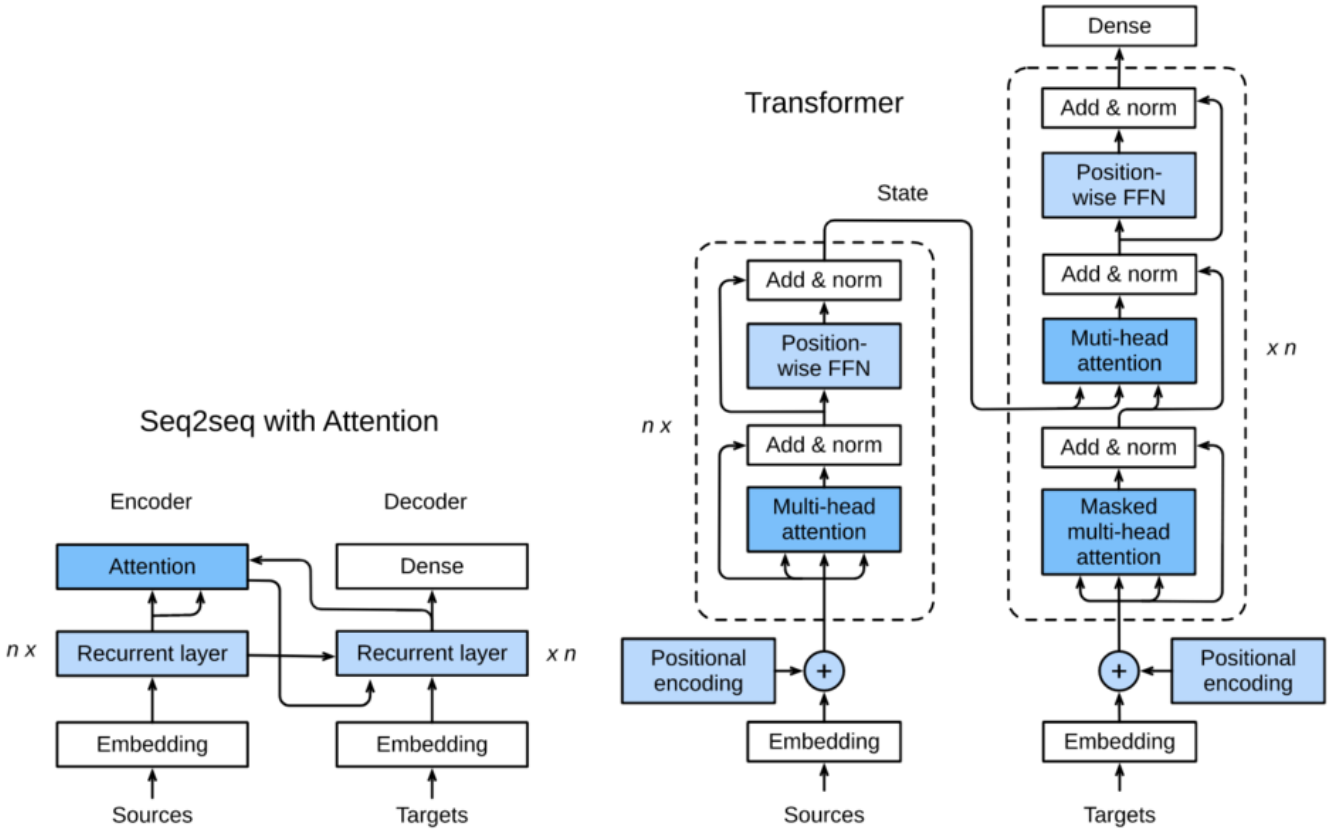
- the encoder valid length: 编码器的有效长度，借此，注意层不会考虑编码器输出中的`paddings` 在解码的每个时间步，使用解码器的最后一个`RNN`层的输出作为注意层的`query`。然后，将注意力模型的输出`context vector`与输入的嵌入向量 D_t 连接起来，输入到`RNN`层。

12 - Transformer

之前介绍的主流神经网络架构CNN和RNN具有如下的优势和不足：

- CNNs 易于并行化，却不适合捕捉变长序列内的依赖关系
- RNNs 适合捕捉长距离变长序列的依赖，但是却难以实现并行化处理序列

为了整合CNN和RNN的优势，Vaswani et al., 2017 创新性地使用注意力机制设计了`Transformer`模型。该模型利用`attention`机制实现了并行化捕捉序列依赖，并且同时处理序列的每个位置的`tokens`，上述优势使得`Transformer`模型在性能优异的同时大大减少了训练时间。

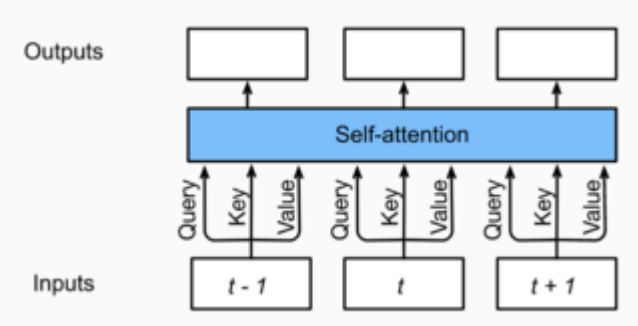


上图展示了`Transformer`模型的架构，与`Seq2seq`模型相似，`Transformer`同样基于编码器-解码器架构，其区别主要在于以下三点：

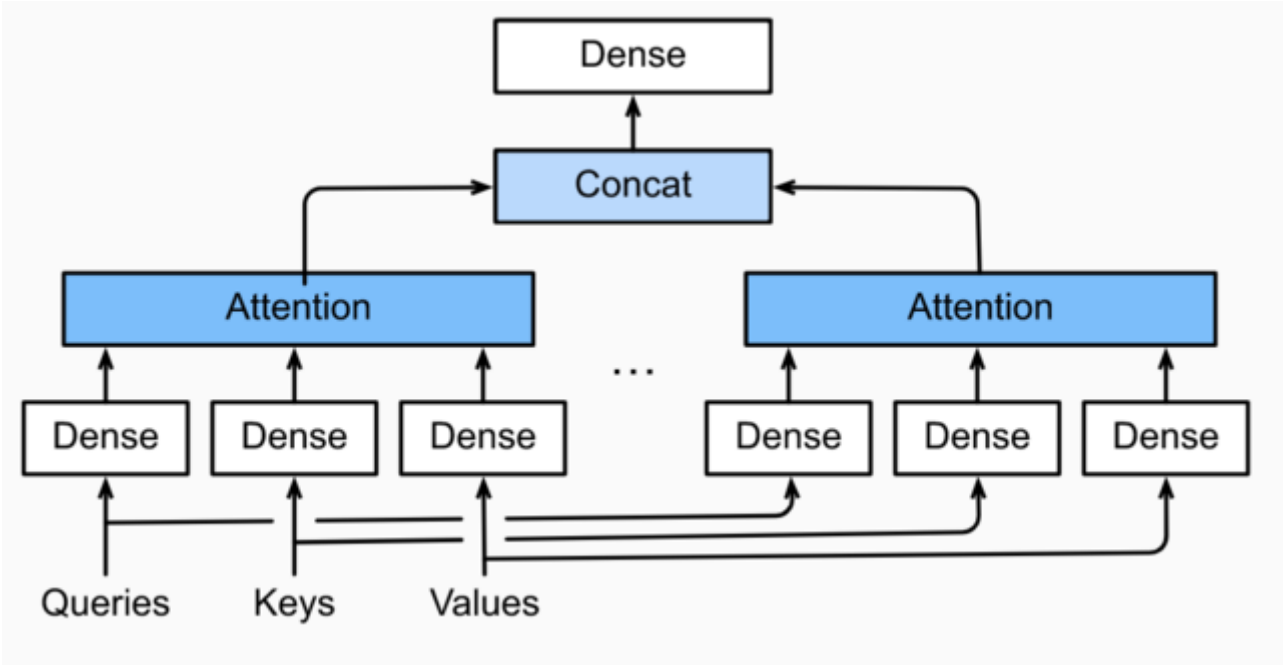
1. `Transformer blocks`：将`Seq2seq`模型重的循环网络替换为了`Transformer Blocks`，该模块包含一个多头注意力层（Multi-head Attention Layers）以及两个`position-wise feed-forward networks`(FFN)。对于解码器来说，另一个多头注意力层被用于接受编码器的隐藏状态。
2. `Add and norm`：多头注意力层和前馈网络的输出被送到两个`add and norm`层进行处理，该层包含残差结构以及层归一化。
3. `Position encoding`：由于自注意力层并没有区分元素的顺序，所以一个位置编码层被用于向序列元素里添加位置信息。

多头注意力层

自注意力模型是一个正规的注意力模型，序列的每一个元素对应的key，value，query是完全一致的。如下图所示，自注意力输出了一个与输入长度相同的表征序列，与循环神经网络相比，自注意力对每个元素输出的计算是并行的。



多头注意力层包含h个并行的自注意力层，每一个这种层被成为一个head。对每个头来说，在进行注意力计算之前，将query、key和value用三个线性层进行映射，这h个注意力头的输出将会被拼接之后输入最后一个线性层进行整合。



假设query，key和value的维度分别是d_q、d_k和d_v。那么对于每一个头$i=1,\dots,h$，我们可以训练相应的模型权重$W_q^{(i)}\in\mathbb{R}^{p_q\times d_q}$、$W_k^{(i)}\in\mathbb{R}^{p_k\times d_k}$和$W_v^{(i)}\in\mathbb{R}^{p_v\times d_v}$，以得到每个头的输出：

$$o^{(i)}=attention(W_q^{(i)}q,W_k^{(i)}k,W_v^{(i)}v)$$

这里的attention可以是任意的attention function，比如前一节介绍的dot-product attention以及MLP attention。之后将所有head对应的输出拼接起来，送入最后一个线性层进行整合，这个层的权重可以表示为$W_o\in\mathbb{R}^{d_0\times hp_v}$

$$o=W_o[o^{(1)},\dots,o^{(h)}]$$

接下来实现多头注意力，假设有h个头，隐藏层维度hidden size$=p_q=p_k=p_v$与query，key，value经过Dense层之后输出的维度一致。除此之外，因为多头注意力层保持输入与输出张量的维度不变，所以输出feature的维度也设置为$d_0=$hidden size。

```
class MultiHeadAttention(nn.Module):
    def __init__(self, input_size, hidden_size, num_heads, dropout, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = DotProductAttention(dropout)
        self.W_q = nn.Linear(input_size, hidden_size, bias=False)
        self.W_k = nn.Linear(input_size, hidden_size, bias=False)
        self.W_v = nn.Linear(input_size, hidden_size, bias=False)
        self.W_o = nn.Linear(hidden_size, hidden_size, bias=False)

    def forward(self, query, key, value, valid_length):
        # query, key, and value shape: (batch_size, seq_len, dim),
        # where seq_len is the length of input sequence
        # valid_length shape is either (batch_size, )
        # or (batch_size, seq_len).

        # Project and transpose query, key, and value from
        # (batch_size, seq_len, hidden_size * num_heads) to
        # (batch_size * num_heads, seq_len, hidden_size).

        query = transpose_qkv(self.W_q(query), self.num_heads)
        key = transpose_qkv(self.W_k(key), self.num_heads)
        value = transpose_qkv(self.W_v(value), self.num_heads)

        if valid_length is not None:
            # Copy valid_length by num_heads times
            device = valid_length.device
            valid_length = valid_length.cpu().numpy() if valid_length.is_cuda else
valid_length.numpy()
            if valid_length.ndim == 1:
                valid_length = torch.FloatTensor(np.tile(valid_length,
self.num_heads))
            else:
                valid_length = torch.FloatTensor(np.tile(valid_length,
(self.num_heads,1)))

            valid_length = valid_length.to(device)

        output = self.attention(query, key, value, valid_length)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)

def transpose_qkv(X, num_heads):
    # Original X shape: (batch_size, seq_len, hidden_size * num_heads),
    # -1 means inferring its value, after first reshape, X shape:
    # (batch_size, seq_len, num_heads, hidden_size)
    X = X.view(X.shape[0], X.shape[1], num_heads, -1)

    # After transpose, X shape: (batch_size, num_heads, seq_len, hidden_size)
    X = X.transpose(2, 1).contiguous()

    # Merge the first two dimensions. Use reverse=True to infer shape from
    # right to left.
```

```
    # output shape: (batch_size * num_heads, seq_len, hidden_size)
    output = X.view(-1, X.shape[2], X.shape[3])
    return output

# 还原成类似初始的维度
def transpose_output(X, num_heads):
    # A reversed version of transpose_qkv
    X = X.view(-1, num_heads, X.shape[1], X.shape[2])
    X = X.transpose(2, 1).contiguous()
    return X.view(X.shape[0], X.shape[1], -1)
```

基于位置的前馈网络

Transformer模块另一个非常重要的部分就是基于位置的前馈网络(FFN)，它接受一个形状为(batch_size, seq_length, feature_size)的三维张量。Position-wise FFN由两个全连接层组成，它们作用在最后一维上。因为序列的每个位置的状态都会被单独地更新，所以我们称它为position-wise，这等效于一个1x1的卷积。

```
class PositionWiseFFN(nn.Module):
    def __init__(self, input_size, ffn_hidden_size, hidden_size_out, **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.ffn_1 = nn.Linear(input_size, ffn_hidden_size)
        self.ffn_2 = nn.Linear(ffn_hidden_size, hidden_size_out)

    def forward(self, X):
        return self.ffn_2(F.relu(self.ffn_1(X)))
```

与多头注意力层相似，FFN层同样只会对最后一维的大小进行改变；除此之外，对于两个完全相同的输入，FFN层的输出也将相等。

Add and Norm

除了上面两个模块之外，Transformer还有一个重要的相加归一化层，它可以平滑地整合输入和其他层的输出，因此在每个多头注意力层和FFN层后面都添加一个含残差连接的Layer Norm层。这里Layer Norm与Batch Norm很相似，唯一的区别在于Batch Norm是对于batch size这个维度进行计算均值和方差的，而Layer Norm则是对最后一维进行计算。层归一化可以防止层内的数值变化过大，从而有利于加快训练速度并且提高泛化性能 (ref)。

```
class AddNorm(nn.Module):
    def __init__(self, hidden_size, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm(hidden_size)
    def forward(self, X, Y):
        return self.norm(self.dropout(Y) + X)
```

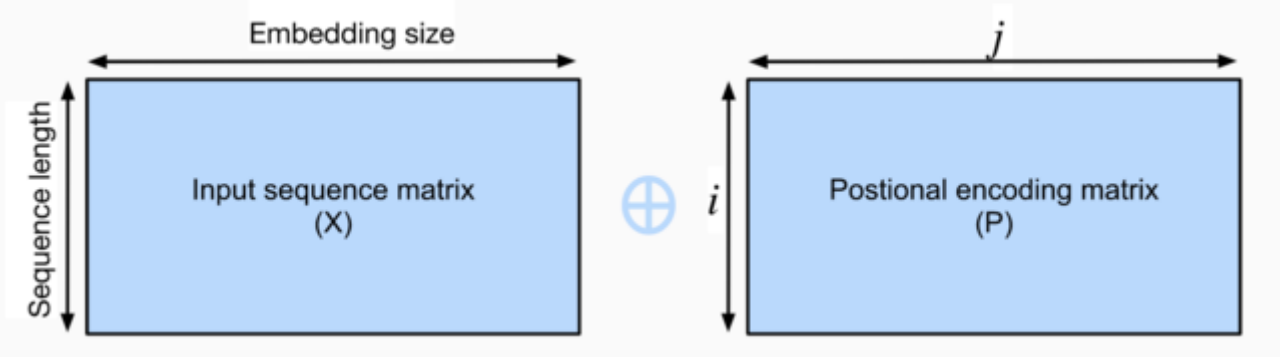
位置编码

与循环神经网络不同，无论是多头注意力网络还是前馈神经网络都是独立地对每个位置的元素进行更新，这种特性帮助我们实现了高效的并行，却丢失了重要的序列顺序的信息。为了更好的捕捉序列信息，Transformer模型引入了位置编码去保持输入序列元素的位置。
假设输入序列的嵌入表示 $X \in \mathbb{R}^{l \times d}$, 序列长度为$l$嵌入向量维度为$d$，则其位置编码为$P \in \mathbb{R}^{l \times d}$ ，输出的向量就是二者相加 $X + P$。
位置编码是一个二维的矩阵，i对应着序列中的顺序，j对应其embedding vector内部的维度索引。可以通过以下等式计算位置编码：

$$P_{i,2j} = sin(i/10000^{2j/d})$$

$$P_{i,2j+1} = cos(i/10000^{2j/d})$$

$$for\ i = 0, \dots, l-1 \qquad j = 0, \dots, \lfloor (d-1)/2 \rfloor$$




```
class PositionalEncoding(nn.Module):
    def __init__(self, embedding_size, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        self.P = np.zeros((1, max_len, embedding_size))
        X = np.arange(0, max_len).reshape(-1, 1) / np.power(
            10000, np.arange(0, embedding_size, 2)/embedding_size)
        self.P[:, :, 0::2] = np.sin(X)
        self.P[:, :, 1::2] = np.cos(X)
        self.P = torch.FloatTensor(self.P)

    def forward(self, X):
        if X.is_cuda and not self.P.is_cuda:
            self.P = self.P.cuda()
        X = X + self.P[:, :X.shape[1], :]
        return self.dropout(X)
```

编码器

现在已经有了组成Transformer的各个模块，可以开始搭建了！编码器包含一个多头注意力层，一个position-wise FFN，和两个Add and Norm层。对于attention模型以及FFN模型，输出维度都是与embedding维度一致的，这是由残差连接天生的特性导致的，因为要将前一层的输出与原始输入相加并归一化。

```
class EncoderBlock(nn.Module):
    def __init__(self, embedding_size, ffn_hidden_size, num_heads,
                 dropout, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = MultiHeadAttention(embedding_size, embedding_size,
num_heads, dropout)
        self.addnorm_1 = AddNorm(embedding_size, dropout)
        self.ffn = PositionWiseFFN(embedding_size, ffn_hidden_size,
embedding_size)
        self.addnorm_2 = AddNorm(embedding_size, dropout)

    def forward(self, X, valid_length):
        Y = self.addnorm_1(X, self.attention(X, X, X, valid_length))
        return self.addnorm_2(Y, self.ffn(Y))
```

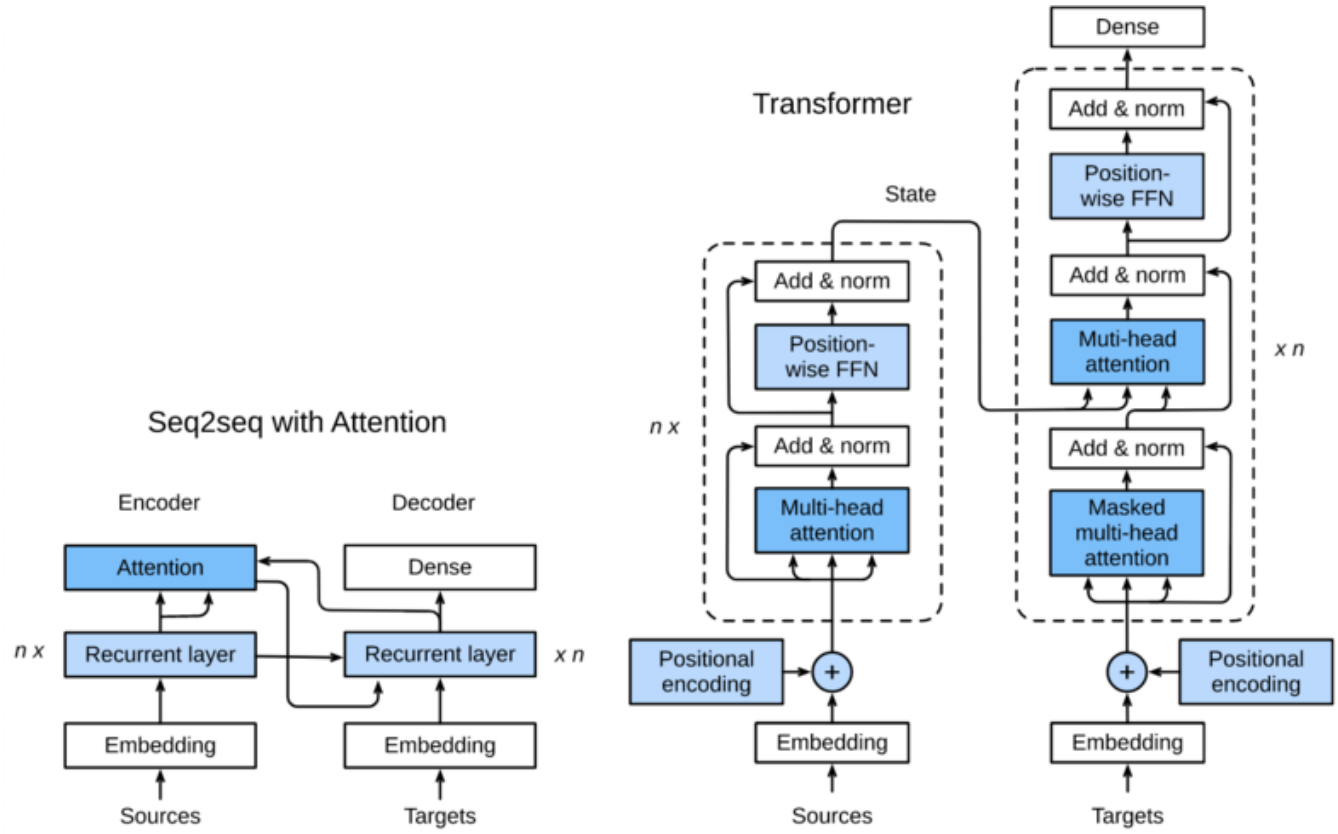
现在来实现整个Transformer编码器模型，整个编码器由 n 个刚刚定义的Encoder Block堆叠而成，因为残差连接的缘故，中间状态的维度始终与嵌入向量的维度 d 一致；同时我们把嵌入向量乘以 $\sqrt{d}$ 以防止其值过小。

```
class TransformerEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embedding_size, ffn_hidden_size,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.embedding_size = embedding_size
        self.embed = nn.Embedding(vocab_size, embedding_size)
        self.pos_encoding = PositionalEncoding(embedding_size, dropout)
        self.blks = nn.ModuleList()
        for i in range(num_layers):
            self.blks.append(
                EncoderBlock(embedding_size, ffn_hidden_size,
                             num_heads, dropout))

    def forward(self, X, valid_length, *args):
        X = self.pos_encoding(self.embed(X) * math.sqrt(self.embedding_size))
        for blk in self.blks:
            X = blk(X, valid_length)
        return X
```

解码器

Transformer模型的解码器与编码器结构类似，然而，除了上述几个模块之外，编码器部分有另一个子模块。该模块也是多头注意力层，接受编码器的输出作为key和value，decoder的状态作为query。与编码器部分相类似，解码器同样是使用了add and norm机制，用残差和层归一化将各个子层的输出相连。

仔细来讲，在第t个时间步，当前输入x_t是query，那么self attention接受了第t步以及前$t-1$步的所有输入$x_1, \ldots, x_{t-1}$。在训练时，由于第t位置的输入可以观测到全部的序列，这与预测阶段的情形项矛盾，所以要将第t个时间步所对应的可观测长度设置为t，以消除不需要看到的未来的信息。




```
# 一个DecoderBlock的实现
class DecoderBlock(nn.Module):
    def __init__(self, embedding_size, ffn_hidden_size,
num_heads,dropout,i,**kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention_1 = MultiHeadAttention(embedding_size, embedding_size,
num_heads, dropout)
        self.addnorm_1 = AddNorm(embedding_size, dropout)
        self.attention_2 = MultiHeadAttention(embedding_size, embedding_size,
num_heads, dropout)
        self.addnorm_2 = AddNorm(embedding_size, dropout)
        self.ffn = PositionWiseFFN(embedding_size, ffn_hidden_size,
embedding_size)
        self.addnorm_3 = AddNorm(embedding_size, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_length = state[0], state[1]

        # state[2][self.i] stores all the previous t-1 query state of layer-i
        # len(state[2]) = num_layers

        # If training:
        #     state[2] is useless.
        # If predicting:
        #     In the t-th timestep:
        #         state[2][self.i].shape = (batch_size, t-1, hidden_size)
        # Demo:
        # love dogs ! [EOS]
```

```
        #  |    |   |   |
        #   Transformer
        #    Decoder
        #  |   |   |   |
        #  I love dogs !

        if state[2][self.i] is None:
            key_values = X
        else:
            # shape of key_values = (batch_size, t, hidden_size)
            key_values = torch.cat((state[2][self.i], X), dim=1)
        state[2][self.i] = key_values

        if self.training:
            batch_size, seq_len, _ = X.shape
            # Shape: (batch_size, seq_len), the values in the j-th column are j+1
            valid_length = torch.FloatTensor(np.tile(np.arange(1, seq_len+1),
(batch_size, 1)))
            valid_length = valid_length.to(X.device)
        else:
            valid_length = None

        X2 = self.attention_1(X, key_values, key_values, valid_length)
        Y = self.addnorm_1(X, X2)
        Y2 = self.attention_2(Y, enc_outputs, enc_outputs, enc_valid_length)
        Z = self.addnorm_2(Y, Y2)
        return self.addnorm_3(Z, self.ffn(Z)), state
```

对于Transformer解码器来说，构造方式与编码器一样，除了最后一层添加一个dense layer以获得输出的置信度分数。下面实现Transformer Decoder，除了常规的超参数例如vocab_size，embedding_size之外，解码器还需要编码器的输出enc_outputs和句子有效长度enc_valid_length。

```
class TransformerDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embedding_size, ffn_hidden_size,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.embedding_size = embedding_size
        self.num_layers = num_layers
        self.embed = nn.Embedding(vocab_size, embedding_size)
        self.pos_encoding = PositionalEncoding(embedding_size, dropout)
        self.blks = nn.ModuleList()
        for i in range(num_layers):
            self.blks.append(
                DecoderBlock(embedding_size, ffn_hidden_size, num_heads,
                             dropout, i))
        self.dense = nn.Linear(embedding_size, vocab_size)

    def init_state(self, enc_outputs, enc_valid_length, *args):
        return [enc_outputs, enc_valid_length, [None]*self.num_layers]

    def forward(self, X, state):
```

```
        X = self.pos_encoding(self.embed(X) * math.sqrt(self.embedding_size))
        for blk in self.blks:
            X, state = blk(X, state)
        return self.dense(X), state
```