1. 机器翻译与数据集

语言模型是自然语言处理的关键， 而*机器翻译*是语言模型最成功的基准测试。 因为机器翻译正是将输入序列转换成输出序列的 *序列转换模型*（sequence transduction）的核心问题。 序列转换模型在各类现代人工智能应用中发挥着至关重要的作用。 为此，本节将介绍机器翻译问题及其后文需要使用的数据集。

机器翻译（machine translation）指的是 将序列从一种语言自动翻译成另一种语言。 事实上，这个研究领域可以追溯到数字计算机发明后不久的20世纪40年代， 特别是在第二次世界大战中使用计算机破解语言编码。 几十年来，在使用神经网络进行端到端学习的兴起之前， 统计学方法在这一领域一直占据主导地位。 因为*统计机器翻译*（statisticalmachine translation）涉及了 翻译模型和语言模型等组成部分的统计分析， 因此基于神经网络的方法通常被称为 *神经机器翻译*（neuralmachine translation）， 用于将两种翻译模型区分开来。

机器翻译的数据集是由源语言和目标语言的文本序列对组成的。 因此，需要一种完全不同的方法来预处理机器翻译数据集， 而不是复用语言模型的预处理程序。 下面看一下如何将预处理后的数据加载到小批量中用于训练。

In [1]:
```
import os
import torch
from d2l import torch as d2l
```

下载和预处理数据集

首先，下载一个由Tatoeba项目的双语句子对 (http://www.manythings.org/anki/) 组成的“英 - 法”数据集，数据集中的每一行都是制表符分隔的文本序列对， 序列对由英文文本序列和翻译后的法语文本序列组成。 请注意，每个文本序列可以是一个句子， 也可以是包含多个句子的一个段落。 在这个将英语翻译成法语的机器翻译问题中， 英语是*源语言*（source language）， 法语是*目标语言*（target language）。

In [2]:
```
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

def read_data_nmt():
    """载入“英语－法语”数据集"""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r',
              encoding='utf-8') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[:75])
```

```
Downloading ../data/fra-eng.zip from http://d2l-data.s3-accelerate.amazonaws.com/fra-eng.zip... (http://d2l-data.s3-accelerate.amazonaws.com/fra-eng.zip...)
Go.	Va !
Hi.	Salut !
Run!	Cours !
Run!	Courez !
Who?	Qui ?
Wow!	Ça alors !
```

下载数据集后，原始文本数据需要经过几个预处理步骤。 例如，用空格代替不间断空格（non-breaking space）， 使用小写字母替换大写字母，并在单词和标点符号之间插入空格。

In [3]:
```
def preprocess_nmt(text):
    """预处理“英语－法语”数据集"""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # 使用空格替换不间断空格
    # 使用小写字母替换大写字母
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # 在单词和标点符号之间插入空格
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[:80])
```

```
go .    va !
hi .    salut !
run !   cours !
run !   courez !
who ?   qui ?
wow !   ça alors !
```

词元化

与之前的中的字符级词元化不同， 在机器翻译中更需要单词级词元化 （最先进的模型可能使用更高级的词元化技术）。 下面的 `tokenize_nmt` 函数对前 `num_examples` 个文本序列对进行词元， 其中每个词元要么是一个词，要么是一个标点符号。 此函数返回两个词元列表： `source` 和 `target` ： `source[i]` 是源语言（这里是英语）第i个文本序列的词元列表， `target[i]` 是目标语言（这里是法语）第i个文本序列的词元列表。

In [4]:
```
def tokenize_nmt(text, num_examples=None):
    """词元化“英语－法语”数据数据集"""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[:6], target[:6]
```

Out[4]:
```
([['go', '.'],
  ['hi', '.'],
  ['run', '!'],
  ['run', '!'],
  ['who', '?'],
  ['wow', '!']],
 [['va', '!'],
  ['salut', '!'],
  ['cours', '!'],
  ['courez', '!'],
  ['qui', '?'],
  ['ça', 'alors', '!']])
```

绘制每个文本序列所包含的词元数量的直方图。 在这个简单的“英 - 法”数据集中，大多数文本序列的词元数量少于20个。

In [6]:
```
def show_list_len_pair_hist(legend, xlabel, ylabel, xlist, ylist):
    """绘制列表长度对的直方图"""
    d2l.set_figsize()
    _, _, patches = d2l.plt.hist(
        [[len(l) for l in xlist], [len(l) for l in ylist]])
    d2l.plt.xlabel(xlabel)
    d2l.plt.ylabel(ylabel)
    for patch in patches[1].patches:
        patch.set_hatch('/')
    d2l.plt.legend(legend)

show_list_len_pair_hist(['source', 'target'], '# tokens per sequence',
                        'count', source, target);
```

<Figure size 252x180 with 1 Axes>

词表

由于机器翻译数据集由语言对组成， 因此可以分别为源语言和目标语言构建两个词表。 使用单词级词元化时，词表大小将明显大于使用字符级词元化时的词表大小。 为了缓解这一问题，这里将出现次数少于2次的低频率词元 视为相同的未知（“<unk>”）词元。 除此之外，我们还指定了额外的特定词元， 例如在小批量时用于将序列填充到相同长度的填充词元（“<pad>”）， 以及序列的开始词元（“<bos>”）和结束词元（“<eos>”）。 这些特殊词元在自然语言处理任务中比较常用。

In [7]:
```
src_vocab = d2l.Vocab(source, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)
```

Out[7]: 10012

加载数据集

回想一下，语言模型中的**序列样本都有一个固定的长度**， 无论这个样本是一个句子的一部分还是跨越了多个句子的一个片断。 这个固定长度是由 `num_steps` （时间步数或词元数量）参数指定的。 在机器翻译中，每个样本都是由源和目标组成的文本序列对， 其中的每个文本序列可能具有不同的长度。

为了提高计算效率可以通过*截断*（truncation）和 *填充*（padding）方式实现一次只处理一个小批量的文本序列。 假设同一个小批量中的每个序列都应该具有相同的长度 `num_steps` ， 那么如果文本序列的词元数目少于 `num_steps` 时，我们将继续在其末尾添加特定的“<pad>”词元， 直到其长度达到 `num_steps` ； 反之，我们将截断文本序列时，只取其前 `num_steps` 个词元， 并且丢弃剩余的词元。这样，每个文本序列将具有相同的长度， 以便以相同形状的小批量进行加载。

如前所述，下面的 `truncate_pad` 函数将(**截断或填充文本序列**)。

In [8]:
```
def truncate_pad(line, num_steps, padding_token):
    """截断或填充文本序列"""
    if len(line) > num_steps:
        return line[:num_steps]  # 截断
    return line + [padding_token] * (num_steps - len(line))  # 填充

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])
```

Out[8]: [47, 4, 1, 1, 1, 1, 1, 1, 1, 1]

现在定义一个函数，可以将文本序列 **转换成小批量数据集用于训练。** 将特定的“<eos>”词元添加到所有序列的末尾，用于表示序列的结束。 当模型通过一个词元接一个词元地生成序列进行预测时， 生成的“<eos>”词元说明完成了序列输出工作。 此外，我们还记录了每个文本序列的长度， 统计长度时排除了填充词元， 在稍后将要介绍的一些模型会需要这个长度信息。

In [9]:
```
def build_array_nmt(lines, vocab, num_steps):
    """将机器翻译的文本序列转换成小批量"""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = torch.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
    return array, valid_len
```

训练模型

最后，定义 load_data_nmt 函数来返回数据迭代器， 以及源语言和目标语言的两种词表。

In [10]:
```
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """返回翻译数据集的迭代器和词表"""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

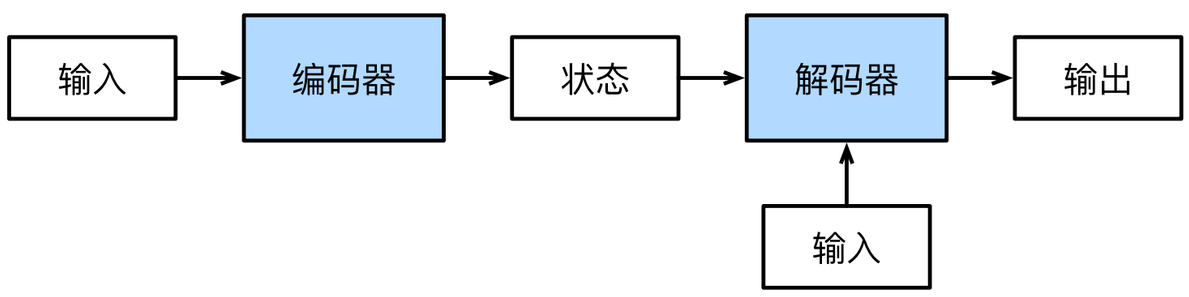
下面读出“英语 - 法语”数据集中的第一个小批量数据。

In [11]:
```
train_iter, src_vocab, tgt_vocab = load_data_nmt(batch_size=2, num_steps=8)
for X, X_valid_len, Y, Y_valid_len in train_iter:
    print('X:', X.type(torch.int32))
    print('X的有效长度:', X_valid_len)
    print('Y:', Y.type(torch.int32))
    print('Y的有效长度:', Y_valid_len)
    break
```

```
X: tensor([[  6, 119,   4,   3,   1,   1,   1,   1],
        [  0,   8,  72,   4,   3,   1,   1,   1]], dtype=torch.int32)
X的有效长度: tensor([4, 5])
Y: tensor([[ 6, 57,  4,  3,  1,  1,  1,  1],
        [ 0, 55,  4,  3,  1,  1,  1,  1]], dtype=torch.int32)
Y的有效长度: tensor([4, 4])
```

2. 编码器-解码器架构

机器翻译是序列转换模型的一个核心问题， 其输入和输出都是长度可变的序列。 为了处理这种类型的输入和输出， 可以设计一个包含两个主要组件的架构： 第一个组件是一个*编码器*（encoder）： 它接受一个长度可变的序列作为输入， 并将其转换为具有固定形状的编码状态。 第二个组件是*解码器*（decoder）： 它将固定形状的编码状态映射到长度可变的序列。 这被称为*编码器-解码器*（encoder-decoder）架构， 如下图所示。



以英语到法语的机器翻译为例： 给定一个英文的输入序列：“They”、“are”、“watching”、“.”。 首先，这种“编码器 - 解码器”架构将长度可变的输入序列编码成一个“状态”， 然后对该状态进行解码， 一个词元接着一个词元地生成翻译后的序列作为输出： “Ils”、“regordent”、“.”。 由于“编码器 - 解码器”架构是形成后续章节中不同序列转换模型的基础， 因此本节将把这个架构转换为接口方便后面的代码实现。

编码器

在编码器接口中，只指定长度可变的序列作为编码器的输入 `X` 。 任何继承这个 `Encoder` 基类的模型将完成代码实现。

In [12]:
```
from torch import nn

class Encoder(nn.Module):
    """编码器-解码器架构的基本编码器接口"""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError
```

解码器

在下面的解码器接口中，新增一个 `init_state` 函数， 用于将编码器的输出（ `enc_outputs` ）转换为编码后的状态。 注意，此步骤可能需要额外的输入，例如：输入序列的有效长度， 这在 :numref: `subsec_mt_data_loading` 中进行了解释。 为了逐个地生成长度可变的词元序列， 解码器在每个时间步都会将输入 （例如：在前一时间步生成的词元）和编码后的状态 映射成当前时间步的输出词元。

In [13]:
```
class Decoder(nn.Module):
    """编码器-解码器架构的基本解码器接口"""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

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

合并编码器和解码器

总而言之，“编码器-解码器”架构包含了一个编码器和一个解码器， 并且还拥有可选的额外的参数。 在前向传播中，编码器的输出用于生成编码状态， 这个状态又被解码器作为其输入的一部分。

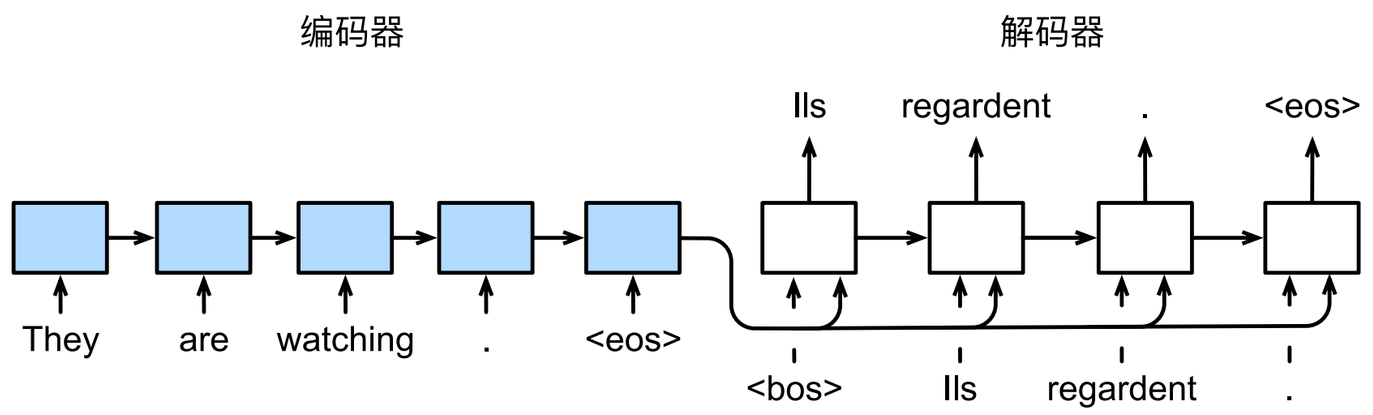
In [14]:
```
class EncoderDecoder(nn.Module):
    """编码器-解码器架构的基类"""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

3. 序列到序列学习（seq2seq）

机器翻译中的输入序列和输出序列都是长度可变的。 为了解决这类问题， 设计了一个通用的”编码器 - 解码器“架构。 下面将使用两个循环神经网络的编码器和解码器， 并将其应用于*序列到序列*（sequence to sequence，seq2seq）类的学习任务。

遵循编码器 - 解码器架构的设计原则， 循环神经网络编码器使用长度可变的序列作为输入， 将其转换为固定形状的隐状态。 换言之，输入序列的信息被*编码*到循环神经网络编码器的隐状态中。 为了连续生成输出序列的词元， 独立的循环神经网络解码器是基于输入序列的编码信息 和输出序列已经看见的或者生成的词元来预测下一个词元。 下图演示

了如何在机器翻译中使用两个循环神经网络进行序列到序列学习。



在上图中， 特定的“<eos>”表示序列结束词元。 一旦输出序列生成此词元，模型就会停止预测。 在循环神经网络解码器的初始化时间步，有两个特定的设计决定： 首先，特定的“<bos>”表示序列开始词元，它是解码器的输入序列的第一个词元。 其次，使用循环神经网络编码器最终的隐状态来初始化解码器的隐状态。 例如，在 Sutskever 等人的设计中， 正是基于这种设计将输入序列的编码信息送入到解码器中来生成输出序列的。 在其他一些设计中， 如上图所示， 编码器最终的隐状态在每一个时间步都作为解码器的输入序列的一部分。 语言模型的训练可以允许标签成为原始的输出序列， 从源序列词元“<bos>”、“Ils”、“regardent”、“.” 到新序列词元 “Ils”、“regardent”、“.”、“<eos>”来移动预测的位置。

下面构建 seq2seq 的设计， 并将基于“英 - 法”数据集来训练这个机器翻译模型。

In [15]:
```
import collections
import math
from torch import nn
```

编码器

从技术上讲，编码器将长度可变的输入序列转换成 形状固定的上下文变量$\mathbf{c}$， 并且将输入序列的信息在该上下文变量中进行编码。 可以使用循环神经网络来设计编码器。

考虑由一个序列组成的样本（批量大小是1）。 假设输入序列是$x_1, \ldots, x_T$， 其中x_t是输入文本序列中的第t个词元。 在时间步t，循环神经网络将词元x_t的输入特征向量 $\mathbf{x}_t$和$\mathbf{h}_{t-1}$（即上一时间步的隐状态） 转换为$\mathbf{h}_t$（即当前步的隐状态）。 使用一个函数f来描述循环神经网络的循环层所做的变换：

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

总之，编码器通过选定的函数q， 将所有时间步的隐状态转换为上下文变量：

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T).$$

比如，当选择$q(\mathbf{h}_1, \ldots, \mathbf{h}_T) = \mathbf{h}_T$时， 上下文变量仅仅是输入序列在最后时间步的隐状态$\mathbf{h}_T$。

到目前为止使用的是一个单向循环神经网络来设计编码器， 其中隐状态只依赖于输入子序列， 这个子序列是由输入序列的开始位置到隐状态所在的时间步的位置 （包括隐状态所在的时间步）组成。 也可以使用双向循环神经网络构造编码器， 其中隐状态依赖于两个输入子序列， 两个子序列是由隐状态所在的时间步的位置之前的序列和之后的序列（包括隐状态所在的时间步）， 因此隐状态对整个序列的信息都进行了编码。

现在**实现循环神经网络编码器。** 注意，这里使用了*嵌入层*（embedding layer） 来获得输入序列中每个词元的特征向量。 嵌入层的权重是一个矩阵， 其行数等于输入词表的大小（ `vocab_size` ）， 其列数等于特征向量的维度（ `embed_size` ）。 对于任意输入词元的索引i， 嵌入层获取权重矩阵的第i行（从0开始）以返回其特征向量。 另外选择了一个多层门控循环单元来实现编码器。

In [16]:
```python
class Seq2SeqEncoder(d2l.Encoder):
    """用于序列到序列学习的循环神经网络编码器"""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        # 嵌入层
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers,
                          dropout=dropout)

    def forward(self, X, *args):
        # 输出'X'的形状：(batch_size,num_steps,embed_size)
        X = self.embedding(X)
        # 在循环神经网络模型中，第一个轴对应于时间步
        X = X.permute(1, 0, 2)
        # 如果未提及状态，则默认为0
        output, state = self.rnn(X)
        # output的形状:(num_steps,batch_size,num_hiddens)
        # state[0]的形状:(num_layers,batch_size,num_hiddens)
        return output, state
```

下面，实例化**上述编码器的实现**： 下面使用一个两层门控循环单元编码器，其隐藏单元数为16。 给定一小批量的输入序列 X （批量大小为4，时间步为7）。 在完成所有时间步后， 最后一层的隐状态的输出是一个张量（ output 由编码器的循环层返回）， 其形状为（时间步数，批量大小，隐藏单元数）。

In [17]:
```python
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)
output, state = encoder(X)
output.shape
```

Out[17]:
```
torch.Size([7, 4, 16])
```

由于这里使用的是门控循环单元， 所以在最后一个时间步的多层隐状态的形状是 （隐藏层的数量，批量大小，隐藏单元的数量）。 如果使用长短期记忆网络， state 中还将包含记忆单元信息。

In [18]:
```python
state.shape
```

Out[18]:
```
torch.Size([2, 4, 16])
```

解码器

正如上文提到的，编码器输出的上下文变量$\mathbf{c}$ 对整个输入序列$x_1, \ldots, x_T$进行编码。 来自训练数据集的输出序列$y_1, y_2, \ldots, y_{T'}$， 对于每个时间步t'（与输入序列或编码器的时间步t不同）， 解码器输出$y_{t'}$的概率取决于先前的输出子序列 $y_1, \ldots, y_{t'-1}$ 和上下文变量$\mathbf{c}$， 即$P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$。

为了在序列上模型化这种条件概率，可以使用另一个循环神经网络作为解码器。 在输出序列上的任意时间步t'， 循环神经网络将来自上一时间步的输出$y_{t'-1}$ 和上下文变量$\mathbf{c}$作为其输入， 然后在当前时间步将它们和上一隐状态 $\mathbf{s}_{t'-1}$转换为 隐状态$\mathbf{s}_{t'}$。 因此，可以使用函数g来表示解码器的隐藏层的变换：

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}).$$

在获得解码器的隐状态之后， 我们可以使用输出层和softmax操作 来计算在时间步t' 时输出$y_{t'}$的条件概率分布 $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$。

当实现解码器时， 直接使用编码器最后一个时间步的隐状态来初始化解码器的隐状态。 这就要求使用循环神经网络实现的编码器和解码器具有相同数量的层和隐藏单元。 为了进一步包含经过编码的输入序列的信息， 上下文变量在所有的时间步与解码器的输入进行拼接（concatenate）。 为了预测输出词元的概率分布， 在循环神经网络解码器的最后一层使用全连接层来变换隐状态。

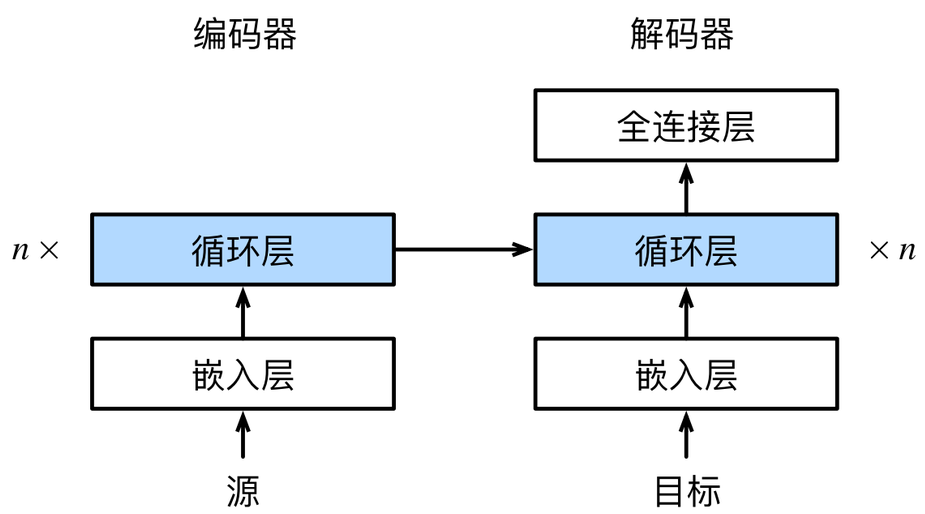
In [19]:
```python
class Seq2SeqDecoder(d2l.Decoder):
    """用于序列到序列学习的循环神经网络解码器"""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens, num_layers,
                          dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        # 输出'X'的形状：(batch_size,num_steps,embed_size)
        X = self.embedding(X).permute(1, 0, 2)
        # 广播context，使其具有与X相同的num_steps
        context = state[-1].repeat(X.shape[0], 1, 1)
        X_and_context = torch.cat((X, context), 2)
        output, state = self.rnn(X_and_context, state)
        output = self.dense(output).permute(1, 0, 2)
        # output的形状:(batch_size,num_steps,vocab_size)
        # state[0]的形状:(num_layers,batch_size,num_hiddens)
        return output, state
```

下面用与前面提到的编码器中相同的超参数来**实例化解码器**。 解码器的输出形状变为（批量大小，时间步数，词表大小）， 其中张量的最后一个维度存储预测的词元分布。

In [21]:
```python
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
decoder.eval()
state = decoder.init_state(encoder(X))
output, state = decoder(X, state)
output.shape, state.shape
```

Out[21]: (torch.Size([4, 7, 10]), torch.Size([2, 4, 16]))

总之，上述循环神经网络“编码器 - 解码器”模型中的各层如 下图所示。



损失函数

在每个时间步，解码器预测了输出词元的概率分布。 类似于语言模型，可以使用softmax来获得分布， 并通过计算交叉熵损失函数来进行优化。 因为特定的填充词元被添加到序列的末尾， 所以不同长度的序列可以以相同形状的小批量加载。 但是，应该将填充词元的预测排除在损失函数的计算之外。

为此可以使用下面的 `sequence_mask` 函数 **通过零值化屏蔽不相关的项**， 以便后面任何不相关预测的计算都是与零的乘积，结果都等于零。 例如，如果两个序列的有效长度（不包括填充词元）分别为1和2， 则第一个序列的第一项和第二个序列的前两项之后的剩余项将被清除为零。

In [22]:
```
def sequence_mask(X, valid_len, value=0):
    """在序列中屏蔽不相关的项"""
    maxlen = X.size(1)
    mask = torch.arange((maxlen), dtype=torch.float32,
                        device=X.device)[None, :] < valid_len[:, None]
    X[~mask] = value
    return X

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

Out[22]:
```
tensor([[1, 0, 0],
        [4, 5, 0]])
```

还可以使用此函数屏蔽最后几个轴上的所有项。如果愿意，也可以使用指定的非零值来替换这些项。

In [23]:
```
X = torch.ones(2, 3, 4)
sequence_mask(X, torch.tensor([1, 2]), value=-1)
```

Out[23]:
```
tensor([[[ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.],
         [-1., -1., -1., -1.]],

        [[ 1.,  1.,  1.,  1.],
         [ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.]]])
```

现在可以**通过扩展softmax交叉熵损失函数来遮蔽不相关的预测。** 最初，所有预测词元的掩码都设置为1。 一旦给定了有效长度，与填充词元对应的掩码将被设置为0。 最后，将所有词元的损失乘以掩码，以过滤掉损失中填充词元产生的不相关预测。

In [24]:
```
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
    """带遮蔽的softmax交叉熵损失函数"""
    # pred的形状：(batch_size,num_steps,vocab_size)
    # label的形状：(batch_size,num_steps)
    # valid_len的形状：(batch_size,)
    def forward(self, pred, label, valid_len):
        weights = torch.ones_like(label)
        weights = sequence_mask(weights, valid_len)
        self.reduction='none'
        unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
            pred.permute(0, 2, 1), label)
        weighted_loss = (unweighted_loss * weights).mean(dim=1)
        return weighted_loss
```

可以创建三个相同的序列来进行**代码健全性检查**， 然后分别指定这些序列的有效长度为4、2和0。 结果就是，第一个序列的损失应为第二个序列的两倍，而第三个序列的损失应为零。

In [25]:
```
loss = MaskedSoftmaxCELoss()
loss(torch.ones(3, 4, 10), torch.ones((3, 4), dtype=torch.long),
     torch.tensor([4, 2, 0]))
```

Out[25]:
```
tensor([2.3026, 1.1513, 0.0000])
```

训练

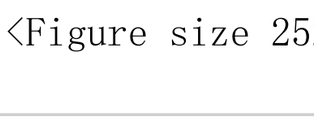
在下面的循环训练过程中， 特定的序列开始词元（“<bos>”）和 原始的输出序列（不包括序列结束词元“<eos>”） 拼接在一起作为解码器的输入。 这被称为*强制教学*（teacher forcing）， 因为原始的输出序列（词元的标签）被送入解码器。 或者，将来自上一个时间步的*预测*得到的词元作为解码器的当前输入。

In [26]:
```python
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
    """训练序列到序列模型"""
    def xavier_init_weights(m):
        if type(m) == nn.Linear:
            nn.init.xavier_uniform_(m.weight)
        if type(m) == nn.GRU:
            for param in m._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(m._parameters[param])

    net.apply(xavier_init_weights)
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    loss = MaskedSoftmaxCELoss()
    net.train()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # 训练损失总和，词元数量
        for batch in data_iter:
            optimizer.zero_grad()
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
                               device=device).reshape(-1, 1)
            dec_input = torch.cat([bos, Y[:, :-1]], 1)  # 强制教学
            Y_hat, _ = net(X, dec_input, X_valid_len)
            l = loss(Y_hat, Y, Y_valid_len)
            l.sum().backward()  # 损失函数的标量进行“反向传播”
            d2l.grad_clipping(net, 1)
            num_tokens = Y_valid_len.sum()
            optimizer.step()
            with torch.no_grad():
                metric.add(l.sum(), num_tokens)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')
```

现在，可以在机器翻译数据集上**创建和训练一个循环神经网络“编码器 - 解码器”模型**用于序列到序列的学习。

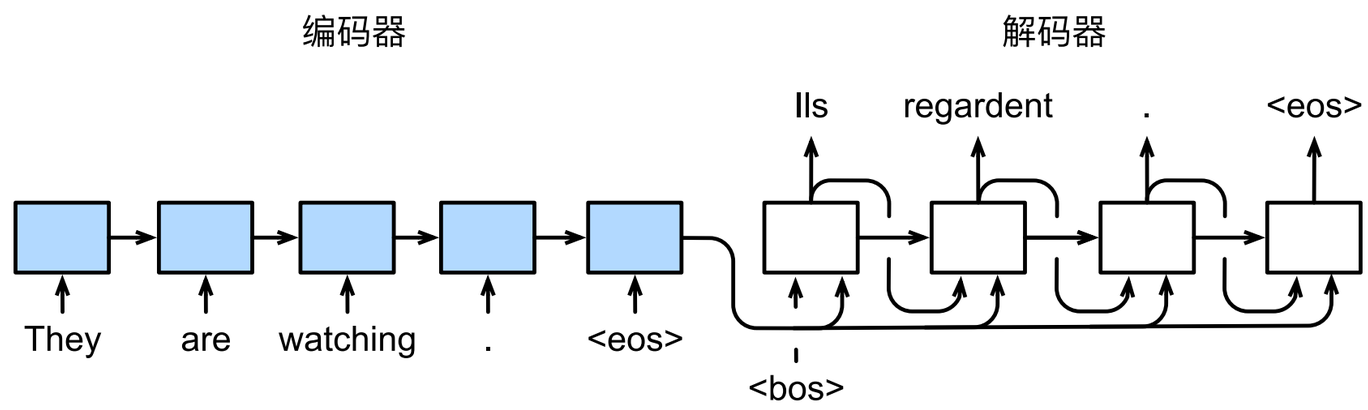
In [27]:
```python
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 300, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = Seq2SeqEncoder(len(src_vocab), embed_size, num_hiddens, num_layers,
                         dropout)
decoder = Seq2SeqDecoder(len(tgt_vocab), embed_size, num_hiddens, num_layers,
                         dropout)
net = d2l.EncoderDecoder(encoder, decoder)
train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.019, 12003.5 tokens/sec on cuda:0
```



预测

为了采用一个接着一个词元的方式预测输出序列， 每个解码器当前时间步的输入都将来自于前一时间步的预测词元。 与训练类似，序列开始词元（“<bos>”） 在初始时间步被输入到解码器中。 该预测过程如下图所示， 当输出序列的预测遇到序列结束词元（“<eos>”）时，预测就结束了。



In [28]:
```
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device, save_attention_weights=False):
    """序列到序列模型的预测"""
    # 在预测时将net设置为评估模式
    net.eval()
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # 添加批量轴
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device), dim=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # 添加批量轴
    dec_X = torch.unsqueeze(torch.tensor(
        [tgt_vocab['<bos>']], dtype=torch.long, device=device), dim=0)
    output_seq, attention_weight_seq = [], []
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        # 我们使用具有预测最高可能性的词元，作为解码器在下一时间步的输入
        dec_X = Y.argmax(dim=2)
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()
        # 保存注意力权重（稍后讨论）
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # 一旦序列结束词元被预测，输出序列的生成就完成了
        if pred == tgt_vocab['<eos>']:
            break
        output_seq.append(pred)
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

预测序列的评估

我们可以通过与真实的标签序列进行比较来评估预测序列。 虽然 Papineni 等人 提出的BLEU（bilingual evaluation understudy） 最先是用于评估机器翻译的结果， 但现在它已经被广泛用于测量许多应用的输出序列的质量。 原则上说，对于预测序列中的任意n元语法（n-grams）， BLEU的评估都是这个n元语法是否出现在标签序列中。

我们将BLEU定义为：

$$\exp\left(\min\left(0, 1 - \frac{\mathrm{len}_{\mathrm{label}}}{\mathrm{len}_{\mathrm{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n},$$

其中$\mathrm{len}_{\mathrm{label}}$表示标签序列中的词元数和 $\mathrm{len}_{\mathrm{pred}}$表示预测序列中的词元数， k是用于匹配的最长的n元语法。 另外，用p_n表示n元语法的精确度，它是两个数量的比值： 第一个是预测序列与标签序列中匹配的n元语法的数量， 第二个是预测序列中n元语法的数量的比率。 具体地说，给定标签序列A、B、C、D、E、F 和预测序列A、B、B、C、D， 我们有$p_1 = 4/5$、$p_2 = 3/4$、$p_3 = 1/3$和$p_4 = 0$。

根据上式中BLEU的定义， 当预测序列与标签序列完全相同时，BLEU为1。 此外，由于n元语法越长则匹配难度越大， 所以BLEU为更长的n元语法的精确度分配更大的权重。 具体来说，当p_n固定时，$p_n^{1/2^n}$ 会随着n的增长而增加（原始论文使用$p_n^{1/n}$）。 而且，由于预测的序列越短获得的p_n值越高， 所以上式中乘法项之前的系数用于惩罚较短的预测序列。 例如，当$k = 2$时，给定标签序列A、B、C、D、E、F 和预测序列A、B，尽管$p_1 = p_2 = 1$， 惩罚因子$\exp(1 - 6/2) \approx 0.14$会降低BLEU。

BLEU的代码实现如下。

In [29]:
```
def bleu(pred_seq, label_seq, k):  #@save
    """计算BLEU"""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

最后，利用训练好的循环神经网络“编码器 - 解码器”模型， **将几个英语句子翻译成法语**，并计算BLEU的最终结果。

In [30]:
```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, attention_weight_seq = predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device)
    print(f'{eng} => {translation}, bleu {bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu payé ., bleu 0.658
he's calm . => il est <unk> mouillé ., bleu 0.548
i'm home . => je suis chez moi bien ., bleu 0.803
```