实验总结:

实验实现了机器翻译相关代码。成功实现将文本序列从一种语言自动翻译成另一种语言。实验过程中:

实验使用单词级词元化时的词表大小, 将明显大于使用字符级词元化时的词表大小。为了缓解这一问题将低频词元视为相同的未知词元。

实验通过截断和填充文本序列，保证所有的文本序列都具有相同的长度，以便以小批量的方式加载。

为了处理长度可变的序列作为的输入和输出，实验使用“编码器 – 解码器”架构，因此适用于机器翻译等序列转换问题。其中，编码器将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。解码器将具有固定形状的编码状态映射为长度可变的序列。

最后根据“编码器 – 解码器”架构，使用两个循环神经网络来设计一个序列到序列学习的模型。其中在实现编码器和解码器时，使用到多层循环神经网络。同时还使用遮蔽来过滤不相关的计算，例如在计算损失时。最后在“编码器 – 解码器”训练中，强制教学方法将原始输出序列（而非预测结果）输入解码器。

。

9.5 机器翻译与数据集

机器翻译（machine translation）指将序列从一种语言自动翻译成另一种语言。

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(Brown et al., 1990, Brown et al., 1988)。

基于神经网络的方法通常被称为 神经机器翻译（neural machine translation）。

本书的关注点是神经网络机器翻译方法，强调的是端到端的学习。

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，将预处理后的数据加载到小批量中用于训练。

In [62]:
```
import os
import torch
from d2l import torch as d2l
```

9.5.1 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 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 [63]:
```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip', '94646ad1522d915e7b0f9296181140edcf86a4f5')  # 将数据集链接和

#@save
def read_data_nmt():  # 载入“英语－法语”数据集
    data_dir = d2l.download_extract('fra-eng')  # 获取数据集存储路径
    with open(os.path.join(data_dir, 'fra.txt'), 'r',encoding='utf-8') as f:  # 打开文件，读取数据
        return f.read()

raw_text = read_data_nmt()  # 读取原始文本数据
print(raw_text[:75])  # 打印前75个字符
```

```
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 [64]:
```
#@save
def preprocess_nmt(text):
    """预处理“英语－法语”数据集"""
    def no_space(char, prev_char):
        # 检查当前字符是否是标点符号，并且前面的字符不是空格
        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)]  # 如果当前字符是标点符号（逗号，句号，问号，感叹号）且前一个字符不是空格，则在当前字符
    return ''.join(out)  # 将处理后的文本合并成一个字符串形式返回

text = preprocess_nmt(raw_text)  # 调用preprocess_nmt函数对原始文本进行预处理
print(text[:80])  # 打印输出文本的前80个字符
```

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

9.5.2. 词元化

在机器翻译中，我们更喜欢单词级词元化。

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。

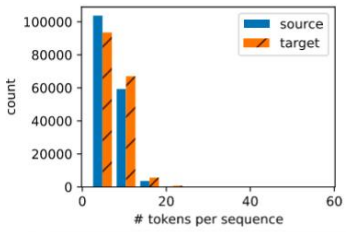
此函数返回两个词元列表：source和target： source[i]是源语言（这里是英语）第个文本序列的词元列表， target[i]是目标语言（这里是法语）第个文本序列的词元列表。

In [65]:
```
#@save
def tokenize_nmt(text, num_examples=None):
    """词元化“英语－法语”数据数据集"""
    source, target = [], []  # 定义两个列表用于存储源和目标序列
    for i, line in enumerate(text.split('\n')):  # 遍历每一行
        if num_examples and i > num_examples:  # 如果指定了num_examples参数，且当前行号大于num_examples，则跳出循环
            break
        parts = line.split('\t')  # 将每行按制表符分割成两部分
        if len(parts) == 2:  # 如果分割出来的部分长度为2，则说明该行包含源序列和目标序列
            source.append(parts[0].split(' '))  # 将源序列以空格为分割符进行分词，并存储到source列表中
            target.append(parts[1].split(' '))  # 将目标序列以空格为分割符进行分词，并存储到target列表中
    return source, target  # 返回源序列和目标序列两个列表

source, target = tokenize_nmt(text)  # 将预处理后的文本进行词元化
source[:6], target[:6]  # 输出前6个源语言列表和前6个目标语言列表的结果
```

Out[65]:
```
([['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

In [66]:
```
#@save
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()  # 设置图的大小
    _, _, patches = d2l.plt.hist([[len(l) for l in xlist], [len(l) for l in ylist]])
    # 使用 d2l.plt.hist() 函数来绘制两个列表的长度对的直方图，其中第一个参数是一个由两个列表组成的列表，第二个参数是要绘制的直方图的数
    # 设置X轴和Y轴的标签
    d2l.plt.xlabel(xlabel)
    d2l.plt.ylabel(ylabel)

    for patch in patches[1].patches:
        patch.set_hatch('/')  # 设置第二个列表的直方图填充模式
    d2l.plt.legend(legend)  # 设置图例

show_list_len_pair_hist(['source', 'target'], '# tokens per sequence','count', source, target);  # 调用该函数，并提供它
```



9.5.3. 词表

由于机器翻译数据集由语言对组成， 因此我们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，这里我们将出现次数少于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”）。

In [67]:
```
c_vocab = d2l.Vocab(source, min_freq=2,reserved_tokens=['<pad>', '<bos>', '<eos>'])  # 定义了一个源语言的词表src_vocab,
n(src_vocab)  # 输出src_vocab中单词的数量。
```

Out[67]:
```
10012
```

9.5.4 加载数据集

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，我们仍然可以通过截断（truncation）和 填充（padding）方式实现一次只处理一个小批量的文本序列。

语言模型中的序列样本都有一个固定的长度，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“”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 [68]:
```
#@save
def truncate_pad(line, num_steps, padding_token):  #     line：待截断或填充的文本序列；num_steps：指定的序列长度；padding_t
    """截断或填充文本序列"""
    if len(line) > num_steps:  # 如果 line 的长度大于 num_steps
        return line[:num_steps]  # 截断
    return line + [padding_token] * (num_steps - len(line))  # 填充（在 line 末尾添加若干个 padding_token，使得 line 的长

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])  # 调用函数truncate_pad截断或填充文本序列
```

Out[68]:
```
[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 [69]:
```
#@save
def build_array_nmt(lines, vocab, num_steps):  # lines：机器翻译的文本序列。vocab：词汇表。num_steps：每个句子被截断或填充后的
    """将机器翻译的文本序列转换成小批量"""
    lines = [vocab[l] for l in lines]  # 将 lines 中的每个句子转换为相应的词汇 id 列表
    lines = [l + [vocab['<eos>']] for l in lines]  # 在每个序列结尾添加了 <eos> 标记

    array = torch.tensor([truncate_pad(l, num_steps, vocab['<pad>']) for l in lines])  # 用 truncate_pad 函数将每个序列
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)  # 将处理后的文本序列转换成一个张量，再计算每个序列的有效长

    return array, valid_len  # 返回转换后的文本序列张量和有效长度张量
```

9.5.5. 训练模型

最后，我们定义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 [70]:
```
#@save
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
```

读出"英语－法语"数据集中的第一个小批量数据。

In [71]:
```
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
```

```
    break
```

```
X: tensor([[  7,  62,   4,   3,   1,   1,   1,   1],
        [118,  55,   4,   3,   1,   1,   1,   1]], dtype=torch.int32)
X的有效长度: tensor([4, 4])
Y: tensor([[6, 7, 0, 4, 3, 1, 1, 1],
        [0, 4, 3, 1, 1, 1, 1, 1]], dtype=torch.int32)
Y的有效长度: tensor([5, 3])
```

9.6. 编码器-解码器架构

正如在 9.5节中所讨论的， 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， 我们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。

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。

In [72]:
```
from torch import nn


#@save
class Encoder(nn.Module):  # 定义了一个名为 Encoder 的类，该类继承自 nn.Module 类。
    """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)  # 使用 super() 函数调用父类的构造函数。

    def forward(self, X, *args):
        raise NotImplementedError  # 前向传播函数，接收输入数据 X 和一些额外的参数 *args，返回输出结果。
```

9.6.2. 解码器

在下面的解码器接口中，我们新增一个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

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 [73]:
```
#@save
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):  # enc_outputs是编码器最后一个时间步的输出，*args为所需的额外的参数
        raise NotImplementedError
```

```
    def forward(self, X, state):  # X解码器输入  state解码器状态
        raise NotImplementedError
        # 当子类继承了某个基类，并且需要实现基类中的方法时，如果还没有实现该方法，则可以使用raise NotImplementedError语句引发此异常。
```

9.6.3. 合并编码器和解码器

“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 [74]:
```
#@save
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  # 编码器
        self.decoder = decoder  # 解码器

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)  # 对编码器输入enc_X进行编码
        dec_state = self.decoder.init_state(enc_outputs, *args)  # 初始化解码器的状态
        return self.decoder(dec_X, dec_state)  # 返回解码结果
```

正如我们在 9.5节中看到的， 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，我们在 9.6节中 设计了一个通用的”编码器－解码器“架构。

本节，我们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

9.7. 序列到序列学习（seq2seq）

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。

训练允许标签成为原始的输出序列， 从源序列词元“”“Ils”“regardent”“.” 到新序列词元 “Ils”“regardent”“.”“”来移动预测的位置。

In [75]:
```
import collections
import math
import torch
from torch import nn
from d2l import torch as d2l
```

9.7.1. 编码器

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， 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。

实现循环神经网络编码器，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， 嵌入层获取权重矩阵的第行（从开始）以返回其特征向量。

另外，本文选择了一个多层门控循环单元来实现编码器。

In [76]:
```
#@save
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)  # 是一个矩阵，其中每一行是一个标记的向量表示。
        # num_hiddens表示的是隐藏单元的个数，一般选择的较大，即为词向量的维度
        # num_layers为循环神经网络的层数
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers, dropout=dropout)
        # 将输入词嵌入序列编码成隐藏状态序列，最终的输出包括所有时间步的隐藏状态和最后一个时间步的隐藏状态，即output和state。

    def forward(self, X, *args):
        # 输出'X'的形状：(batch_size,num_steps,embed_size)
        X = self.embedding(X)  # 将输入序列 X 中的每个元素替换成对应的向量表示，这个过程叫做嵌入
        # 在循环神经网络模型中，第一个轴对应于时间步（对输入张量进行维度的转换，将形状为 (num_steps, batch_size, embed_size) 的输入
        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的形状:(num_layers,batch_size,num_hiddens)
        return output, state  # 返回输出output和最终时间步的隐藏状态state
```

In [77]:
```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
num_layers=2) # 创建一个编码器实例
encoder.eval() # 将模型切换为评估模式
X = torch.zeros((4, 7), dtype=torch.long) # 构造输入数据，大小为(4,7)
output, state = encoder(X) # 前向计算，返回输出（时间步数，批量大小，隐藏单元数）和最终时间步的隐藏状态
output.shape # 输出结果形状为（时间步数，批量大小，隐藏单元数）
```

Out[77]:
```
torch.Size([7, 4, 16])
```

In [78]:
```
state.shape  # 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）
```

Out[78]:
```
torch.Size([2, 4, 16])
```

9.7.2. 解码器

当实现解码器时， 我们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 [79]:
```
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)
        '''
        将vocab_size个词嵌入到embed_size维度的向量空间中。
        在训练中，每个词都将被替换为其对应的向量表示。
        例如，如果一个句子是“hello world”，那么它会被转换成一个整数序列（每个整数对应于一个单词），然后使用nn.Embedding将其转换成一个
        '''
        self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens, num_layers,dropout=dropout)
        '''
        定义了一个由num_layers层循环神经网络组成的 GRU 模型，其中每一层的输入由 embed_size + num_hiddens 维的向量组成，
        它是当前时间步输入的词向量与上一时间步的隐藏状态在特征维上的连结，输出由 num_hiddens 维的隐藏状态组成。
        dropout 是一个 dropout 层，它的输出将随机的丢弃一些元素，以防止过拟合。
        '''
        self.dense = nn.Linear(num_hiddens, vocab_size)
        '''
        一个线性变换层 (Linear)，
        将输入的 num_hiddens 维的隐藏状态映射到一个 vocab_size 维的输出，用于生成词汇表中的单词的概率分布。
        '''

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]  # 直接返回编码器最终时间步的隐藏状态


    def forward(self, X, state):
        X = self.embedding(X).permute(1, 0, 2) # 将序列数据进行嵌入并调整形状，将时间步放在第一维
        context = state[-1].repeat(X.shape[0], 1, 1) # 将编码器最终时间步的隐藏状态进行复制，广播到所有时间步上
        X_and_context = torch.cat((X, context), 2) # 将解码器的输入与复制后的上下文变量进行拼接
        output, state = self.rnn(X_and_context, state) # 将拼接后的输入和状态传入循环神经网络
        output = self.dense(output).permute(1, 0, 2) # 将输出数据进行全连接层处理，并调整形状，将时间步放在第一维
        return output, state # 返回解码器的输出和最终时间步的隐藏状态
```

In [80]:
```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
num_layers=2)  # 初始化一个Seq2SeqDecoder对象decoder
decoder.eval()   # 将模型切换为评估模式
state = decoder.init_state(encoder(X))  # 先用encoder来对输入序列进行编码，并初始化decoder的隐状态，将编码器输出的隐状态作为初始
output, state = decoder(X, state)  # 然后将初始解码器隐状态和输入序列传递给解码器，生成输出序列。
output.shape, state.shape  # 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。
```

Out[80]:
```
(torch.Size([4, 7, 10]), torch.Size([2, 4, 16]))
```

9.7.3. 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。

9.5节中， 特定的填充词元被添加到序列的末尾， 因此不同长度的序列可以以相同形状的小批量加载。 但是，我们应该将填充词元的预测排除在损失函数的计算之外。

为此，我们可以使用下面的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。

In [81]:
```
#@save
def sequence_mask(X, valid_len, value=0):  # X表示需要被屏蔽的序列，valid_len表示每个序列的有效长度
    # 在序列中屏蔽不相关的项

    maxlen = X.size(1)  # X的第2个维度的长度，即序列的最大长度
    # 创建一个二维的掩码矩阵mask，对于每个序列，mask中对应有效长度以后的元素都被赋值为0，其余元素为1
    mask = torch.arange((maxlen), dtype=torch.float32,device=X.device)[None, :] < valid_len[:, None]

    X[~mask] = value  # 使用掩码矩阵将X中无效的元素（即有效长度以后的元素）屏蔽掉，将这些元素的值设为value
    return X  # 返回屏蔽后的序列。

X = torch.tensor([[1, 2, 3], [4, 5, 6]])
sequence_mask(X, torch.tensor([1, 2]))  # 使用sequence_mask函数对输入张量进行了屏蔽
```

Out[81]:
```
tensor([[1, 0, 0],
        [4, 5, 0]])
```

In [82]:
```
# 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。
X = torch.ones(2, 3, 4)
sequence_mask(X, torch.tensor([1, 2]), value=-1)
```

Out[82]:
```
tensor([[[ 1.,  1.,  1.,  1.],
```

Out[82]:
```
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，我们可以通过扩展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 [83]:
```python
#@save
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)  # 初始化权重张量为一个与标签值张量label同样形状的全一张量
        weights = sequence_mask(weights, valid_len)  # 根据有效长度valid_len对权重张量进行遮蔽，以屏蔽掉无效项。

        self.reduction='none'  # 将损失函数的reduction属性设置为none，以便返回每个样本的未加权损失。
        unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
        pred.permute(0, 2, 1), label)  # 调用nn.CrossEntropyLoss类的forward函数计算未加权损失。

        weighted_loss = (unweighted_loss * weights).mean(dim=1)  # 对未加权损失进行加权，以便屏蔽掉无效项，然后计算加权损失的均

        return weighted_loss  # 返回加权损失的均值作为结果。
```

In [84]:
```python
loss = MaskedSoftmaxCELoss()

'''
传入了三个参数。其中第一个参数 torch.ones(3, 4, 10) 表示模型预测出的类别概率值，其形状为 (batch_size, num_steps, vocab_size)

第二个参数 torch.ones((3, 4), dtype=torch.long) 表示真实的标签序列，其形状为 (batch_size, num_steps)

第三个参数 torch.tensor([4, 2, 0]) 表示每个序列有效长度，其形状为 (batch_size,)

'''
loss(torch.ones(3, 4, 10), torch.ones((3, 4), dtype=torch.long),torch.tensor([4, 2, 0]))
```

Out[84]:
```
tensor([2.3026, 1.1513, 0.0000])
```

9.7.4. 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。

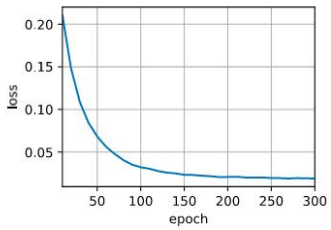
或者，将来自上一个时间步的预测得到的词元作为解码器的当前输入。

In [85]:
```python
#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device): # 训练序列到序列模型
    '''
    net：待训练的序列到序列模型
    data_iter：数据迭代器，用于读取训练数据
    lr：学习率
    num_epochs：训练迭代次数
    tgt_vocab：目标语言词典
    device：指定使用的设备，如 CPU 或 GPU
    '''
    def xavier_init_weights(m):
        # 模型初始化的辅助函数
        if type(m) == nn.Linear:  # 当输入是 nn.Linear类型时
            nn.init.xavier_uniform_(m.weight)  # 一种初始化方法，这种方法可以有效地减少神经网络在训练过程中的梯度消失或梯度爆炸问
        if type(m) == nn.GRU:  # 当输入是 nn.GRU类型时,针对 nn.GRU 层的权重进行 Xavier 初始化
            for param in m._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(m._parameters[param])

    net.apply(xavier_init_weights)  # 对模型参数进行Xavier初始化
    net.to(device)  # 将模型移动到指定设备
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)  # 创建Adam优化器
    loss = MaskedSoftmaxCELoss()  # 定义带遮蔽的softmax交叉熵损失函数
    net.train()  # 设置模型为训练模式
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[10, num_epochs])  # 创建动画

    for epoch in range(num_epochs):
        timer = d2l.Timer() # 计时器，记录每个epoch的训练时间
        metric = d2l.Accumulator(2) # Accumulator用于记录训练损失总和，词元数量
        for batch in data_iter:
            optimizer.zero_grad() # 优化器梯度清零
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch] # 将数据移动到指定设备
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0], device=device).reshape(-1, 1) # 获取<BOS>的id并构造
            dec_input = torch.cat([bos, Y[:, :-1]], 1) # 构造解码器输入，通过将<BOS>和去掉最后一个位置的标签拼接
```

```
            Y_hat, _ = net(X, dec_input, X_valid_len) # 模型预测
            l = loss(Y_hat, Y, Y_valid_len) # 计算损失
            l.sum().backward() # 损失函数的标量进行"反向传播"
            d2l.grad_clipping(net, 1) # 梯度裁剪，限制梯度的大小，避免梯度爆炸
            num_tokens = Y_valid_len.sum() # 一个batch中标签的总数
            optimizer.step() # 根据反向传播计算的梯度更新模型参数
            with torch.no_grad():
                metric.add(l.sum(), num_tokens) # 更新metric的值，累加每个batch的损失和标签总数
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],)) # 用animator记录训练过程中的损失值，每10个epoch记录一次


    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')
    # metric[0] / metric[1]：表示平均损失，其中 metric[0] 是训练期间的总损失，metric[1] 是训练期间处理的词元总数；
    # metric[1] / timer.stop()：表示平均每秒处理的词元数量，其中 timer.stop() 返回自训练开始以来的总秒数。
```

In [86]:
```
# 定义模型参数
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()

# 加载数据
train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)

# 定义编码器和解码器
encoder = Seq2SeqEncoder(len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)

# 构建seq2seq模型
net = d2l.EncoderDecoder(encoder, decoder)

# 训练seq2seq模型
train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.019, 6031.3 tokens/sec on cpu
```



9.7.5. 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。

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 [87]:
```
#@save
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()
    # 将源语言句子转换成token
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    # 通过长度截断或填充源语言句子
    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)
    # 使用编码器编码源语言句子
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    # 初始化解码器的隐藏状态
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # 将目标语言句子的第一个词元设置为<bos>
    dec_X = torch.unsqueeze(torch.tensor(
        [tgt_vocab['<bos>']], dtype=torch.long, device=device), dim=0)
    # 初始化输出序列和注意力权重序列
    output_seq, attention_weight_seq = [], []
```

```
    for _ in range(num_steps):
        # 解码器解码
        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)
    # 将输出序列转换成文本
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

9.7.6. 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。

BLEU的评估都是这个n元语法是否出现在标签序列中。

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。

In [88]:
```
def bleu(pred_seq, label_seq, k): #@save
    # 计算BLEU
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')  # 分割句子为单词序列

    len_pred, len_label = len(pred_tokens), len(label_tokens)  # 计算预测序列和标签序列的长度

    score = math.exp(min(0, 1 - len_label / len_pred))  # 计算长度惩罚项，这是机器翻译中的一种技术，旨在惩罚生成的句子长度过长或

    for n in range(1, k + 1):
        # num_matches 是一个计数器，表示在当前 n-gram 下，预测序列和标签序列中匹配的 n-gram 的数量。
        # label_subs 是一个字典，存储了标签序列中每个 n-gram 的出现次数
        num_matches, label_subs = 0, collections.defaultdict(int)

        # 记录长度为n的子序列在label_tokens中出现的次数
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1

        # 遍历预测序列中所有长度为n的n-gram，并检查它们是否在目标序列的n-gram字典中出现过
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
            # 接着对于预测序列中的每个n-gram，如果其在label_subs中出现的次数大于0，则说明该n-gram匹配成功，将其计数，并将label_sub
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1

        # 计算BLEU中的精确匹配项的权重
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
        '''
        这行代码计算了当前n-gram的precision，然后对precision进行加权。

        precision的计算为num_matches / (len_pred - n + 1)，
        其中num_matches为当前预测序列和参考序列中n-gram相同的数量，len_pred - n + 1为当前预测序列的n-gram数量。
        其中，len_pred - n + 1是denominator是考虑到每个位置的n-gram只能与后面的len_pred - n + 1个位置的n-gram匹配。

        这个precision会被加权，权重是0.5的n次方，这个值是为了惩罚较高的n，因为更高的n的匹配数量在总体分数中应该占较小的比例。

        最后，对于每个n，它们的加权precision的乘积作为BLEU分数。
        '''
    return score  # 返回BLEU分数
```

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 [89]:
```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']  # 英语句子列表engs
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']  # 法语句子列表fras

for eng, fra in zip(engs, fras):
    translation, attention_weight_seq = predict_seq2seq(net, eng, src_vocab, tgt_vocab, num_steps, device)  # 对英语句
    print(f'{eng} => {translation}, bleu {bleu(translation, fra, k=2):.3f}')  # BLEU分数，并打印结果
```

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

根据结果可以看出，对于简单的短句来说，模型表现很好，BLEU分数是1.0。但是对于更复杂的短语和句子，模型表现不佳，BLEU分数较低，甚至有一些错误的翻译。这可能是由于模型结构和数据集的限制导致的。