6.11 机器翻译和数据集

机器翻译（MT）：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。 输出序列的长度可能与源序列的长度不同。

In [28]:

```
import os
os.listdir('/home/kesci/input/')
```

Out[28]:

```
['fraeng6506', 'd2l9528', 'd2l6239']
```

In [1]:

```
import sys
sys.path.append('/home/kesci/input/d2l9528/')
import collections
import d2l
import zipfile
from d2l.data.base import Vocab
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils import data
from torch import optim
```

数据预处理

将数据集清洗、转化为神经网络的输入minbatch

In [2]:

```
with open('/home/kesci/input/fraeng6506/fra.txt', 'r') as f:
      raw_text = f.read()
print(raw_text[0:1000])
```

字符在计算机里是以编码的形式存在，我们通常所用的空格是 \x20 ，是在标准ASCII可见字符 0x20~0x7e 范围内。 而 \xa0 属于 latin1 （ISO/IEC_8859-1）中的扩展字符集字符，代表不间断空白符nbsp(non-breaking space)，超出gbk编码范围，是需要去除的特殊字符。再数据预处理的过程中，我们首先需要对数据进行清洗。

```
Go.	Va !	CC-BY 2.0 (France) Attribution: tatoeba.org #28772
72 (CM) & #1158250 (Wittydev)
Hi.	Salut !	CC-BY 2.0 (France) Attribution: tatoeba.org #5381
23 (CM) & #509819 (Aiji)
Hi.	Salut.	CC-BY 2.0 (France) Attribution: tatoeba.org #5381
23 (CM) & #4320462 (gillux)
Run!	Cours!	CC-BY 2.0 (France) Attribution: tatoeba.org #90632
8 (papabear) & #906331 (sacredceltic)
Run!	Courez!	CC-BY 2.0 (France) Attribution: tatoeba.or
g #906328 (papabear) & #906332 (sacredceltic)
Who?	Qui ?	CC-BY 2.0 (France) Attribution: tatoeba.org #2083
030 (CK) & #4366796 (gillux)
Wow!	Ça alors!	CC-BY 2.0 (France) Attribution: tatoeba.or
g #52027 (Zifre) & #374631 (zmoo)
Fire!	Au feu !	CC-BY 2.0 (France) Attribution: tatoeba.o
rg #1829639 (Spamster) & #4627939 (sacredceltic)
Help!	À l'aide!	CC-BY 2.0 (France) Attribution: tatoeba.or
g #435084 (lukaszpp) & #128430 (sysko)
Jump.	Saute.	CC-BY 2.0 (France) Attribution: tatoeba.org #6310
38 (Shishir) & #2416938 (Phoenix)
Stop!	Ça suffit!	CC-BY 2.0 (France) Attribution: tato
```

In [3]:

预处理第一步需要去掉空格，这儿空格是是拉丁文中的字符而不是我们平常用的空格，这是超出GBK编码范围的，需要替换它。

① 去空格 ② 转小写 ③ 在单词和标点间加空格

```python
def preprocess_raw(text):
    text = text.replace('\u202f', ' ').replace('\xa0', ' ')
    out = ''
    for i, char in enumerate(text.lower()):
        if char in (',', '!', '.') and i > 0 and text[i-1] != ' ':
            out += ' '
        out += char
    return out

text = preprocess_raw(raw_text)
print(text[0:1000])
```

```
go .	va !	cc-by 2 .0 (france) attribution: tatoeba .org #287
7272 (cm) & #1158250 (wittydev)
hi .	salut ! cc-by 2 .0 (france) attribution: tatoeba .org #53
8123 (cm) & #509819 (aiji)
hi .	salut . cc-by 2 .0 (france) attribution: tatoeba .org #53
8123 (cm) & #4320462 (gillux)
run !	cours ! cc-by 2 .0 (france) attribution: tatoeba .org #90
6328 (papabear) & #906331 (sacredceltic)
```

```
run !   courez !        cc-by 2 .0 (france) attribution: tatoeba
.org #906328 (papabear) & #906332 (sacredceltic)
who?    qui ?   cc-by 2 .0 (france) attribution: tatoeba .org #20
83030 (ck) & #4366796 (gillux)
wow !   ça alors !      cc-by 2 .0 (france) attribution: tatoeba
.org #52027 (zifre) & #374631 (zmoo)
fire !  au feu !        cc-by 2 .0 (france) attribution: tatoeba
.org #1829639 (spamster) & #4627939 (sacredceltic)
help !  à l'aide !      cc-by 2 .0 (france) attribution: tatoeba
.org #435084 (lukaszpp) & #128430 (sysko)
jump .  saute . cc-by 2 .0 (france) attribution: tatoeba .org #63
1038 (shishir) & #2416938 (phoenix)
stop !  ça suffit !     cc-b
```

字符在计算机里是以编码的形式存在，我们通常所用的空格是 \x20 ，是在标准ASCII可见字符 0x20~0x7e 范围内。 而 \xa0 属于 latin1 （ISO/IEC_8859-1）中的扩展字符集字符，代表不间断空白符nbsp(non-breaking space)，超出gbk编码范围，是需要去除的特殊字符。再数据预处理的过程中，我们首先需要对数据进行清洗。

分词

字符串→单词组成的列表

In [4]:

这两个空列表放英语、法语

切分每一行(以'\n'区分)

切分 go . \t va !，以'\t'区分

每个词与标点切分.

```python
num_examples = 50000
source, target = [], []
for i, line in enumerate(text.split('\n')):
    if i > num_examples:
        break
    parts = line.split('\t')
    if len(parts) >= 2:
        source.append(parts[0].split(' '))
        target.append(parts[1].split(' '))

source[0:3], target[0:3]
```

Out[4]:

一句话

一个词

英语→

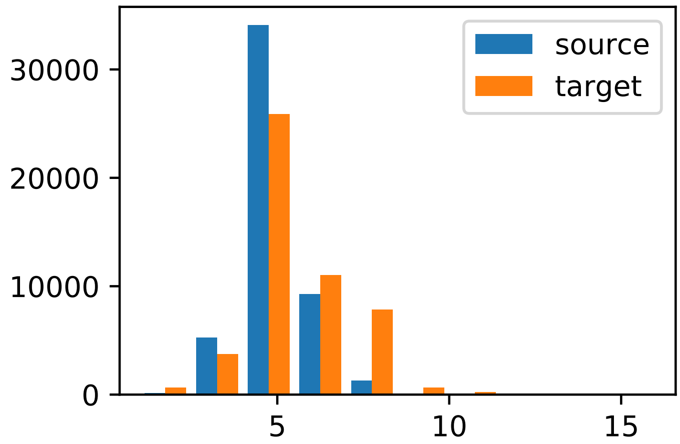
法语→

```
([['go', '.'], ['hi', '.'], ['hi', '.']],
 [['va', '!'], ['salut', '!'], ['salut', '.']])
```

In [5]:

统计句长：

```
d2l.set_figsize()
d2l.plt.hist([[len(l) for l in source], [len(l) for l in target]]
,label=['source', 'target'])
d2l.plt.legend(loc='upper right');
```



建立词典 为英语和法语建立词典

收录在数据集中出现过的所有英语、法语单词。

单词组成的列表-->单词id组成的列表

In [6]:

```
def build_vocab(tokens):
    tokens = [token for line in tokens for token in line]
    return d2l.data.base.Vocab(tokens, min_freq=3, use_special_to
kens=True)

src_vocab = build_vocab(source)
len(src_vocab)
```

tokenize的过程：

调用vocab类：

Out[6]:

英文单词数：3789

```
class Vocab(object): # This class is saved in d2l.
                          所有单词组成的列表      最小词频
  def __init__(self, tokens, min_freq=0, use_special_tokens=False):
    # sort by frequency and token
    counter = collections.Counter(tokens) # 统计词频
    token_freqs = sorted(counter.items(), key=lambda x: x[0])
    token_freqs.sort(key=lambda x: x[1], reverse=True)
    if use_special_tokens:
      # padding, begin of sentence, end of sentence, unknown
      self.pad, self.bos, self.eos, self.unk = (0, 1, 2, 3) # 单词表的0~3先赋予特殊字符
      tokens = ['<pad>', '<bos>', '<eos>', '<unk>']
    else:
      self.unk = 0
      tokens = ['<unk>']
    tokens += [token for token, freq in token_freqs if freq >= min_freq]
    建立id<—>tokens的双向映射
    self.idx_to_token = []
    self.token_to_idx = dict()
    for token in tokens:
      self.idx_to_token.append(token)
      self.token_to_idx[token] = len(self.idx_to_token) - 1
建立内置函数:

  def __len__(self): # 单词表类vocab的长度
    return len(self.idx_to_token)

魔法函数，用vocab[] 直接调用
  def __getitem__(self, tokens): # 输入一个单词的列表tokens，返回对应的id。列表中有几个token，就返回几个id。
    if not isinstance(tokens, (list, tuple)):
      return self.token_to_idx.get(tokens, self.unk)
    else:
返回一个列表: return [self.__getitem__(token) for token in tokens]
```

载入数据集

In [7]:

```
                    每个batch最长的长度
统一化句子的长度: def pad(line, max_len, padding_token):
    if len(line) > max_len: # 如果这句话比规定的max_len长，则切去
        return line[:max_len]
    否则，用padding_token补足句子，补到max_len长度。
    return line + [padding_token] * (max_len - len(line))
pad(src_vocab[source[0]], 10, src_vocab.pad)
```

Out[7]:

```
[38, 4, 0, 0, 0, 0, 0, 0, 0, 0]
```

所有句子在处理之前，都要变成id的形式，如上面的样子，‘go .’变成38，4……。

In [8]:

```
                                          定是英语还是法语
def build_array(lines, vocab, max_len, is_source):
                      line为一串单词列表，传入line，返回一串id列表
    lines = [vocab[line] for line in lines]
    if not is_source:
        lines = [[vocab.bos] + line + [vocab.eos] for line in lines]
                              为每段line都padding
    array = torch.tensor([pad(line, max_len, vocab.pad) for line in lines])
                      即保留句子真实的长度，比如"go."padding后的长度是10，真实长度是2 。
确定有效长度 valid_len = (array != vocab.pad).sum(1) #第一个维度
    return array, valid_len
             由id组成的tensor
```

```
[docs]class TensorDataset(Dataset):
    r"""Dataset wrapping tensors.

    Each sample will be retrieved by indexing tensors along the first dimension.

    Arguments:
        *tensors (Tensor): tensors that have the same size of the first dimension.
    """

    def __init__(self, *tensors):
        assert all(tensors[0].size(0) == tensor.size(0) for tensor in tensors)
        self.tensors = tensors

    def __getitem__(self, index):
        return tuple(tensor[index] for tensor in self.tensors)

    def __len__(self):
        return self.tensors[0].size(0)
```

In [9]:

```
def load_data_nmt(batch_size, max_len): # This function is saved in d2l.
生成词典: src_vocab, tgt_vocab = build_vocab(source), build_vocab(target)
生成id列表 src_array, src_valid_len = build_array(source, src_vocab, max_len, True)
```

```
_len, True)
    tgt_array, tgt_valid_len = build_array(target, tgt_vocab, max
_len, False)
    train_data = data.TensorDataset(src_array, src_valid_len, tgt
_array, tgt_valid_len)
    train_iter = data.DataLoader(train_data, batch_size, shuffle=
True)
    return src_vocab, tgt_vocab, train_iter
```

确定四个参数的第0维是否一一对应，即每个array样本有一个有效长度，都有一个对应的法语标签，法语有效长度；用一个assert判断，返回一个元组。

训练数据生成器

In [10]:

```
src_vocab, tgt_vocab, train_iter = load_data_nmt(batch_size=2, ma
x_len=8)
for X, X_valid_len, Y, Y_valid_len, in train_iter:
    print('X =', X.type(torch.int32), '\nValid lengths for X =',
X_valid_len,
          '\nY =', Y.type(torch.int32), '\nValid lengths for Y =',
Y_valid_len)
    break
```

因为batch_size设为2，所以每次取2个句子，最长长度为8

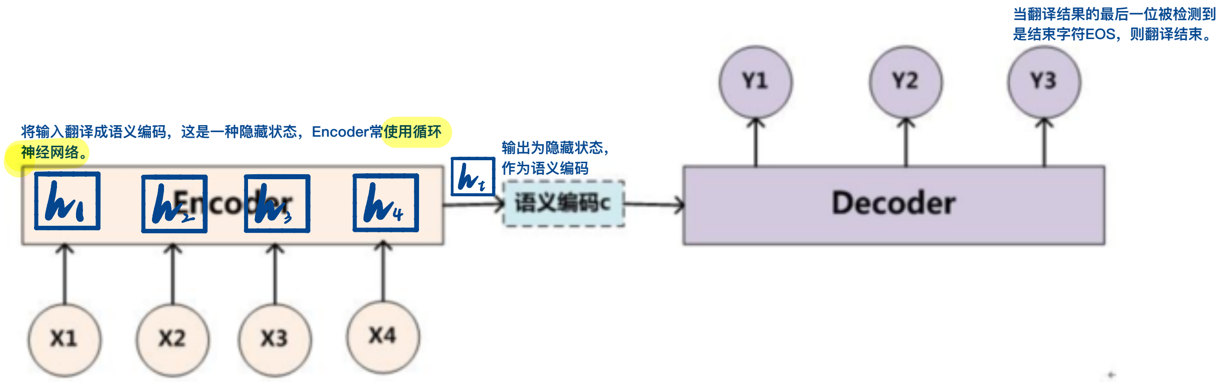
```
X = tensor([[    5,    24,     3,     4,     0,     0,     0,     0],
        [   12, 1388,     7,     3,     4,     0,     0,     0]], dtype=
torch.int32)
Valid lengths for X = tensor([4, 5])
Y = tensor([[    1,    23,    46,     3,     3,     4,     2,     0],
        [    1,    15,   137,    27, 4736,     4,     2,     0]], dtype=
torch.int32)
Valid lengths for Y = tensor([7, 7])
```

Encoder-Decoder

翻译的输入输出是不等长的，针对输入输出不等价，引入的Encoder-Decoder

encoder：输入到隐藏状态

decoder：隐藏状态到输出



In [11]:

继承

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

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

In [12]:

```
class Decoder(nn.Module):
    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
```

In [13]:

```
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)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

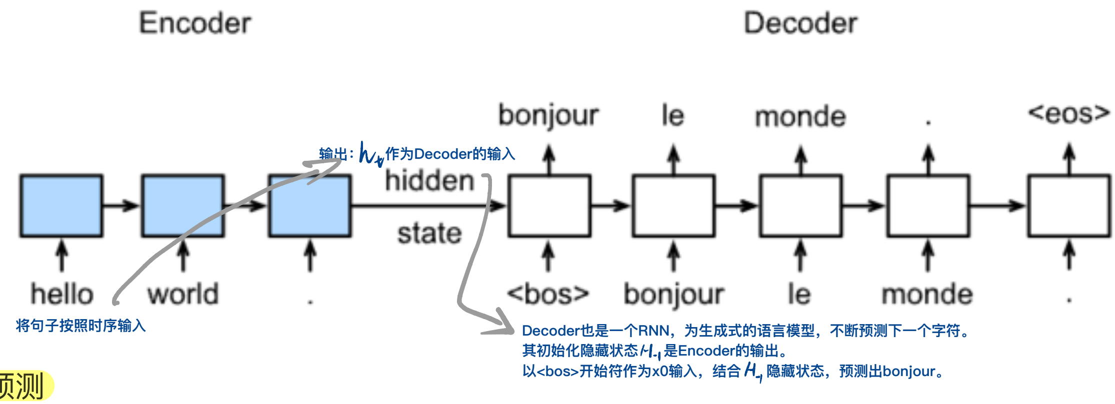
可以应用在对话系统、生成式任务中。

机器翻译中用到的一种Encoder-Decoder结构为

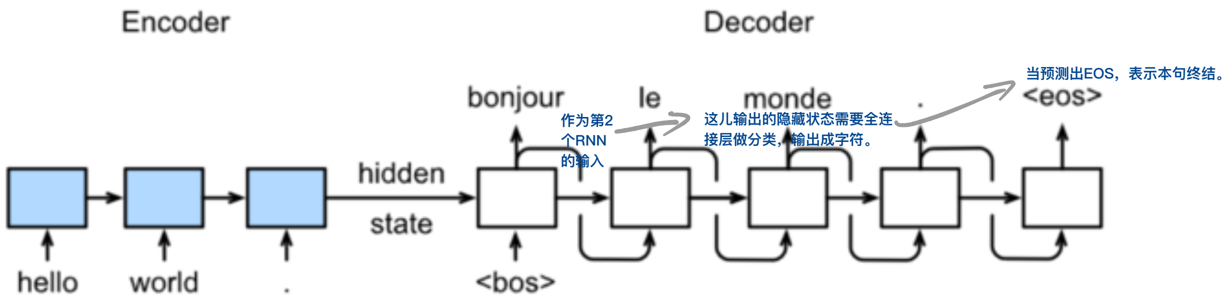
Sequence to Sequence模型

模型：

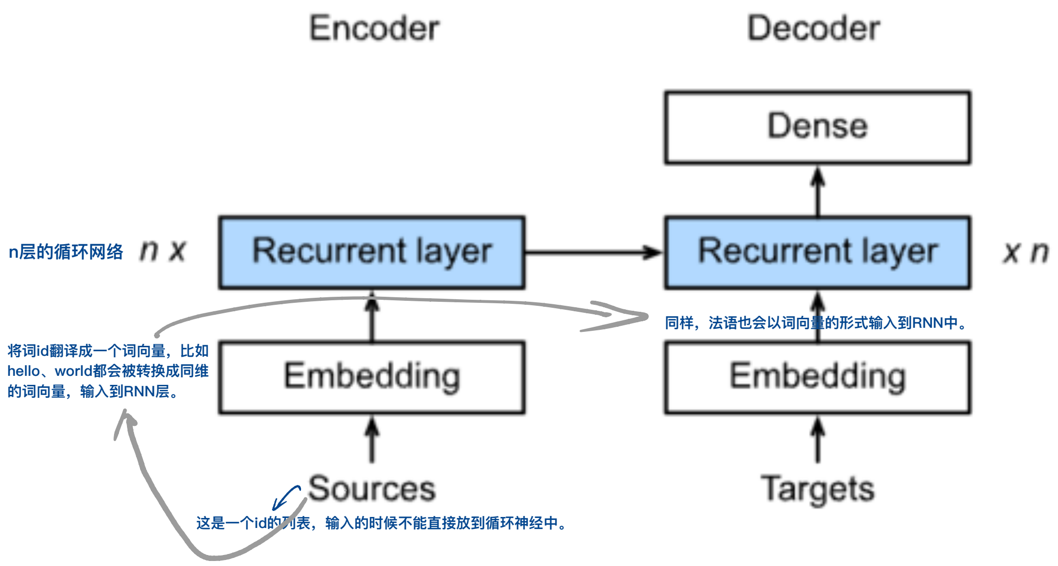
训练



预测



具体结构：



Encoder

In [14]:

```
class Seq2SeqEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_l
ayers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        self.num_hiddens=num_hiddens
        self.num_layers=num_layers
                                                    词向量的纬度
  生成词向量 self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.LSTM(embed_size,num_hiddens, num_layers, dr
opout=dropout)

初始化函数 def begin_state(self, batch_size, device):
        return [torch.zeros(size=(self.num_layers, batch_size, se
lf.num_hiddens),  device=device),
                torch.zeros(size=(self.num_layers, batch_size, se
lf.num_hiddens),  device=device)]
    def forward(self, X, *args):
                                       有几句话        一句话有几个单词   每个单词是几维的词向量
        X = self.embedding(X) # X shape: (batch_size, seq_len, em
bed_size)
        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) RNN按照时序输入，时序作为第0个维度。反映到NLP中，句子顺序要作为第0个维度，所以把batch_size和seq_len调换。
        # 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 is (num_layers, batch_
size, num_hiddens)
                   每个时序的输出
        return out, state
                        最后的语义编码，包含隐藏层状态和记忆细胞的状态
```

In [15]:

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8,num_hiddens=16, num_layers=2)  2层RNN
X = torch.zeros((4, 7),dtype=torch.long)
output, state = encoder(X)
output.shape, len(state), state[0].shape, state[1].shape
```

构造一个Encoder

每个词用一个8维的向量表示

隐藏层神经元的个数是16

构造一个输入X

输入4句话，每句话7个单词

Out[15]:

记忆细胞+隐层2部分

```
(torch.Size([7, 4, 16]), 2, torch.Size([2, 4, 16]), torch.Size([2, 4, 16]))
```

Decoder

In [16]:

```
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)
        self.dense = nn.Linear(num_hiddens,vocab_size)# 每个RNN单元都会预测一个词出来，预测的结果是一个隐藏状态，需要将这个隐藏状态转换成单词字符，所以需要全连接层做分类输出。

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1] # 含记忆细胞和隐藏层状态

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

把batch_size和seq_len点到颠倒的维度在转回来。

In [17]:

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

```
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, state[1].shape
```

Out[17]:

字典大小是10，所以每个RNN单元的输出都会在这10个词中选一个概率最高的词作输出。

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

损失函数

损失函数计算要针对句子的有效长度，所以要先提出padding的部分。

In [18]:

```
def SequenceMask(X, X_len,value=0):
    maxlen = X.size(1)
    mask = torch.arange(maxlen)[None, :].to(X_len.device) < X_len
[:, None]
    X[~mask]=value
    return X
```

要把arange放到和X_len同一个GPU上计算；
因为X_len已经在GPU里了，可以直接调用它的一个属性.device获取其设备

In [19]:

例子，下面的输入X有两句话，“1，2，3”和“4，5，6”。

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

第2个参数显示，第1句话有效长度为1，第二句有效长度为2 。

Out[19]:

```
tensor([[1, 0, 0],
        [4, 5, 0]])
```

按照有效长度对原句作剔除。

In [20]:

```
X = torch.ones((2,3, 4))
SequenceMask(X, torch.tensor([1,2]),value=-1)
```

用-1填充

Out[20]:

```
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.]]])
```

In [21]:

计算损失函数

继承交叉熵损失函数，只要重写forward函数，在其中调用SequenceMask函数

```
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
    # pred shape: (batch_size, seq_len, vocab_size) pred：是decoder的输出，即out，每个decoder隐藏层会输出每个单词的概率
    # label shape: (batch_size, seq_len) label：概率最高的那个单词的id
    # valid_length shape: (batch_size, ) valid_length：每个句子的有效长度
    def forward(self, pred, label, valid_length):
        # the sample weights shape should be (batch_size, seq_len
)
先初始化weights的尺寸 weights = torch.ones_like(label)
        weights = SequenceMask(weights, valid_length).float() # 会把有效长度以外的部分置为0；后面的计算中，就可以将weights乘以output，将padding部分计算的损失变成0，保留有效长度部分的损失值。
        self.reduction='none'
        output=super(MaskedSoftmaxCELoss, self).forward(pred.tran
spose(1,2), label) # 调用交叉熵损失函数的forward方法
        return (output*weights).mean(dim=1)
                                在第1维取平均，第0维是batch_size，第1维是每个单词
```

In [22]:

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

Out[22]:

```
tensor([2.3026, 1.7269, 0.0000])
```

训练

In [23]:

```
                    是Encoder-Decoder
                    模型的结构
def train_ch7(model, data_iter, lr, num_epochs, device):  # Saved
in d2l
    model.to(device) # to(device)这一步是针对GPU的，方向传播的计算都要放到同一个GPU中。若用CPU，这一步可忽略。
    optimizer = optim.Adam(model.parameters(), lr=lr)
    loss = MaskedSoftmaxCELoss()
    tic = time.time() # 计时
    for epoch in range(1, num_epochs+1):
    本epoch的loss的总和    本epoch的tokens的总和
        l_sum, num_tokens_sum = 0.0, 0.0
        for batch in data_iter:
            optimizer.zero_grad() # 优化器的梯度置0
```

英语句子　有效长度　标签：法语句子，内容格式：BOS words...... EOS

这些参数都是从x.to(device)获取的，x.to(device)已经在GPU中了，所以这些参数不用再显式地放入GPU。

```
            X, X_vlen, Y, Y_vlen = [x.to(device) for x in batch]
            Y_input, Y_label, Y_vlen = Y[:,:-1], Y[:,1:], Y_vlen-
1
```

即Decoder的输入，它是不需要EOS的。　标签，即正确的输出，它不需要BOS的。　即label的有效长度，所以减去BOS这个字符

预测的输出

```
            Y_hat, _ = model(X, Y_input, X_vlen, Y_vlen)
            l = loss(Y_hat, Y_label, Y_vlen).sum()计算交叉熵
            l.backward()

            with torch.no_grad():# 梯度裁剪
                d2l.grad_clipping_nn(model, 5, device)
            num_tokens = Y_vlen.sum().item()
            optimizer.step()
            l_sum += l.sum().item()
            num_tokens_sum += num_tokens
        if epoch % 50 == 0:
            print("epoch {0:4d},loss {1:.3f}, time {2:.1f} sec".f
ormat(
                  epoch, (l_sum/num_tokens_sum), time.time()-tic)
)
            tic = time.time()
```

In [24]:

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_examples, max_len = 64, 1e3, 10
lr, num_epochs, ctx = 0.005, 300, d2l.try_gpu()
src_vocab, tgt_vocab, train_iter = d2l.load_data_nmt(
    batch_size, max_len,num_examples)
encoder = Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
model = d2l.EncoderDecoder(encoder, decoder)
train_ch7(model, train_iter, lr, num_epochs, ctx)
```

```
epoch   50,loss 0.093, time 38.2 sec
epoch  100,loss 0.046, time 37.9 sec
epoch  150,loss 0.032, time 36.8 sec
epoch  200,loss 0.027, time 37.5 sec
epoch  250,loss 0.026, time 37.8 sec
epoch  300,loss 0.025, time 37.3 sec
```

测试

In [25]:

```
                                                  输入的一句话（字符串）
def translate_ch7(model, src_sentence, src_vocab, tgt_vocab, max_len, device):
tokenization过程: src_tokens = src_vocab[src_sentence.lower().split(' ')]
获取有效长度src_len = len(src_tokens)
oadding:  if src_len < max_len:
        src_tokens += [src_vocab.pad] * (max_len - src_len)
把enc_X(id列表)和有效长度变成神经
网络能够接受的输入，tensor的形式
    enc_X = torch.tensor(src_tokens, device=device)
    enc_valid_length = torch.tensor([src_len], device=device)
    # use expand_dim to add the batch_size dimension. 因为encoder的输入还有batch_size维度，所以需要添加一个维度
                                                该函数用于在dim=0维增加一个维度
作为decoder的初始化隐层 enc_outputs = model.encoder(enc_X.unsqueeze(dim=0), enc_valid_length)
                                   比如，enc_X是一个长度为10的tensor，这步函数后，就将它变成一个1*10的tensor
初始化的状态 dec_state = model.decoder.init_state(enc_outputs, enc_valid_length)
decoder的第一个输入，BOS dec_X = torch.tensor([tgt_vocab.bos], device=device).unsqueeze(dim=0)
    predict_tokens = []
    for _ in range(max_len):
                                                     BOS          初始化的隐藏层状态
        Y, dec_state = model.decoder(dec_X, dec_state)
        # The token with highest score is used as the next time step input.
                     找到概率最高的那个单词，比如预测出是bonjour，这就作为下一个RNN的输入
获取下一个RNN的输入 dec_X = Y.argmax(dim=2)
        py = dec_X.squeeze(dim=0).int().item() # py为当前获取的单词
        if py == tgt_vocab.eos: #如果碰到EOS，本句结束，跳出循环。
            break
        predict_tokens.append(py)
    return ' '.join(tgt_vocab.to_tokens(predict_tokens))
```

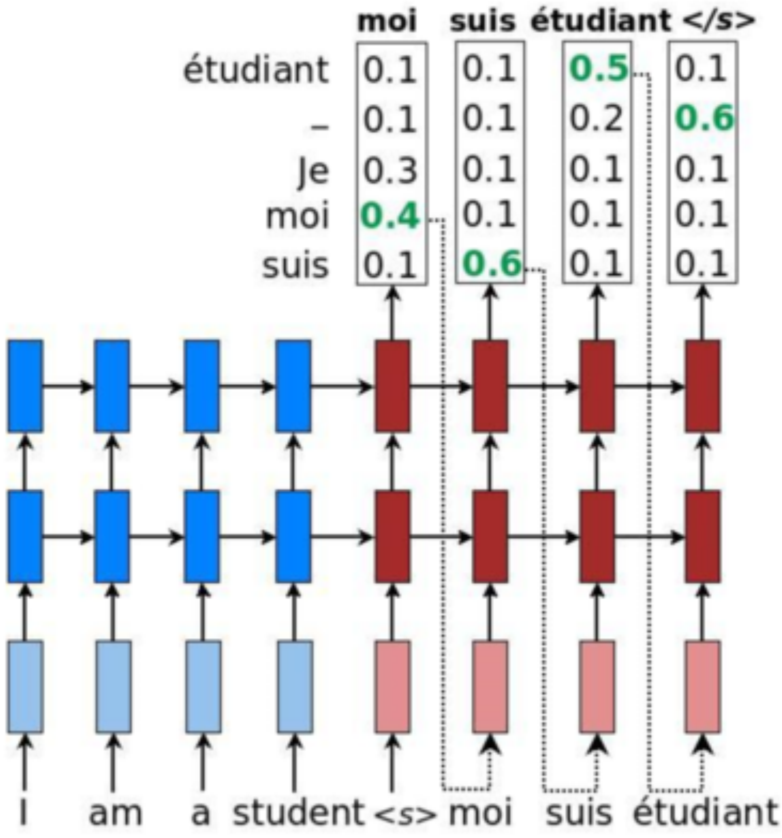
In [26]:

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + translate_ch7(
        model, sentence, src_vocab, tgt_vocab, max_len, ctx))
```

```
Go . => va !
Wow ! => <unk> !
I'm OK . => ça va .
I won ! => j'ai gagné !
```

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：