**基于Transformer模型的机器翻译实践**

1. 任务介绍

本次实践将手动实现一个Transformer模型，完成一个简单的翻译任务，以更直观地理解Transformer的计算过程。考虑到自然语言的复杂性， 而且目标是要理解Transformer的计算过程，所以不会涉及太复杂的任务，而是完成一个尽量简单的任务，以更清晰地观察到ransformer的计算过程。接下来介绍本次任务中原文和译文的生成策略，原文的生成策略如图1所示。



图1 原文的生成策略

在本次任务中，原文的生成策略是从一个有限词表中按一定的概率随机采样，生成随机长度的原文。这并不是自然语言，但可以模拟一句自然语言而且足够简单，便于我们观察Transformer的计算过程。译文的生成策略如图2所示。



图2 译文的生成策略

出于简单起见，译文的生成策略也简单清晰，译文中第1个词双写，剩下的词和原文的词一一对应，如果是数字，则取9以内的互补数，如果是字母，则取大写，并且顺序是原文的整体逆序。首字母双写的目的一方面是为了增加对应复杂度，让这个任务不至于过分简单，另一方面是为了让译文比原文多一位，这会给后续的计算提供方便。

和大多数的NLP任务一样，在本次任务中也会对文本进行预处理，会对每一句文本添加首尾标识符，此外由于文本是随机长度的，所以文本是长短不一的，为了更方便地处理这些文本，我们会把文本都补充到固定的长度，如图3所示。



图3 文本数据预处理

出了方便叙述，后将使用和表示原文和译文。

2. 定义数据集

首先把本次任务要使用的字典定义出来，代码如下：

|  |
| --- |
| vocab\_x = '<SOS>,<EOS>,<PAD>,0,1,2,3,4,5,6,7,8,9,q,w,e,r,t,y,u,i,o,p,a,s,d,f,g,h,j,k,l,z,x,c,v,b,n,m'  vocab\_x = {word: i for i, word in enumerate(vocab\_x.split(','))}  vocab\_xr = [k for k, v in vocab\_x.items()]  vocab\_y = {k.upper(): v for k, v in vocab\_x.items()}  vocab\_yr = [k for k, v in vocab\_y.items()]  print('vocab\_x=', vocab\_x)  print('vocab\_y=', vocab\_y)  #在这段代码中分别定义了x和y的字典. |

vocab\_x和vocab\_y分别代表了x和y的字典，字典内容就是简单的每个词和某个数字的对应关系，包括特殊符号。为了方便以后打印预测结果，还构建了两个逆字典，也就是通过数字找到对应的原文。

接下来定义生成数据的函数，代码如下：

|  |
| --- |
| import random  import numpy as np  import torch  def get\_data():  #定义词集合  words = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'q', 'w', 'e', 'r','t', 'y', 'u', 'i', 'o', 'p', 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k','l', 'z', 'x', 'c', 'v', 'b', 'n', 'm']  #定义每个词被选中的概率  p = np.array([  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,  13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26  ])  p = p / p.sum()  #随机选n个词  n = random.randint(30, 48)  x = np.random.choice(words, size=n, replace=True, p=p)  #采样的结果就是x  x = x.tolist()  #y是对x的变换得到的  #字母大写,数字取9以内的互补数  def f(i):  i = i.upper()  if not i.isdigit():  return i  i = 9 - int(i)  return str(i)  y = [f(i) for i in x]  #逆序  y = y[::-1]  #y中的首字母双写  y = [y[0]] + y  #加上首尾符号  x = ['<SOS>'] + x + ['<EOS>']  y = ['<SOS>'] + y + ['<EOS>']  #补pad到固定长度  x = x + ['<PAD>'] \* 50  y = y + ['<PAD>'] \* 51  x = x[:50]  y = y[:51]  #编码成数据  x = [vocab\_x[i] for i in x]  y = [vocab\_y[i] for i in y]  #转tensor  x = torch.LongTensor(x)  y = torch.LongTensor(y)  return x, y get\_data() #这个函数每调用一次就按照之前所说的策略生成一对x、y. |

接下来可以定义数据集及数据集加载器，代码如下：

|  |
| --- |
| #定义数据集 class Dataset(torch.utils.data.Dataset):  def \_\_init\_\_(self):  super(Dataset, self).\_\_init\_\_()  def \_\_len\_\_(self):  return 100000  def \_\_getitem\_\_(self, i):  return get\_data() #数据加载器 loader = torch.utils.data.DataLoader(dataset=Dataset(),  batch\_size=8,  drop\_last=True,  shuffle=True,  collate\_fn=None) #查看数据样例 for i, (x, y) in enumerate(loader):  break x.shape, y.shape |

在数据集中，数据的总量定义为10万条，事实上由于数据是随机生成的，其实数据有无穷多条，但PyTorch在定义数据集时需要有一个明确的数量，所以此处定义为10万条。每次获取数据时，就调用定义好的生成数据函数，生成一对x、y即可。数据集加载器定义了每个批次中包括8对x和y。在代码的最后获取了一批x、y，并输出了形状。可以观察到y的长度比x多一位，这是故意为之的设计，以便于后续的计算。

3. 定义MASK函数

接下来定义两个MASK函数，先定义PAD MASK函数，代码如下：

|  |
| --- |
| def mask\_pad(data):  #b句话,每句话50个词,这里是还没embed的  #data = [b, 50]  #判断每个词是不是<PAD>  mask = data == vocab\_x['<PAD>']  #[b, 50] -> [b, 1, 1, 50]  mask = mask.reshape(-1, 1, 1, 50)  #在计算注意力时,是计算50个词和50个词相互之间的注意力,所以是个50\*50的矩阵  #是pad的列是true,意味着任何词对pad的注意力都是0  #但是pad本身对其他词的注意力并不是0  #所以是pad的行不是true  #复制n次  #[b, 1, 1, 50] -> [b, 1, 50, 50]  mask = mask.expand(-1, 1, 50, 50)  return mask mask\_pad(x[:1]) |

在这段代码中，根据输入的句子中的每个词是否是PAD，选择是否MASK某一列，最终的输出形状是b×1×50×50，其中b表示一个批次数据的数量，50表示句子的词数量，在本次任务中，每个句子的长度都是固定的50。

接下来定义上三角MASK，代码如下：

|  |
| --- |
| def mask\_tril(data):  #b句话,每句话50个词,这里是还没embed的  #data = [b, 50]  #50\*50的矩阵表示每个词对其他词是否可见  #上三角矩阵,不包括对角线,意味着,对每个词而言,他只能看到他自己,和他之前的词,而看不到之后的词  #[1, 50, 50]  """  [[0, 1, 1, 1, 1],  [0, 0, 1, 1, 1],  [0, 0, 0, 1, 1],  [0, 0, 0, 0, 1],  [0, 0, 0, 0, 0]]"""  tril = 1 - torch.tril(torch.ones(1, 50, 50, dtype=torch.long))  #判断y当中每个词是不是pad,如果是pad则不可见  #[b, 50]  mask = data == vocab\_y['<PAD>']  #变形+转型,为了之后的计算  #[b, 1, 50]  mask = mask.unsqueeze(1).long()  #mask和tril求并集  #[b, 1, 50] + [1, 50, 50] -> [b, 50, 50]  mask = mask + tril  #转布尔型  mask = mask > 0  #转布尔型,增加一个维度,便于后续的计算  mask = (mask == 1).unsqueeze(dim=1)  return mask mask\_tril(x[:1]) |

在这段代码中，首先生成了一个上三角MASK，之后以输入文本中的每个词是否是PAD来生成PAD MASK，最后把两个MASK合并。最终输出的形状和PAD MASK函数相同，也是b×1×50×50。

4. 定义Transformer工具子层

接下来定义注意力计算层，代码如下：

|  |
| --- |
| def attention(Q, K, V, mask):  #b句话,每句话50个词,每个词编码成32维向量,4个头,每个头分到8维向量  #Q、K、V = [b, 4, 50, 8]  #[b, 4, 50, 8] \* [b, 4, 8, 50] -> [b, 4, 50, 50]  #Q、K矩阵相乘,求每个词相对其他所有词的注意力  score = torch.matmul(Q, K.permute(0, 1, 3, 2))  #除以每个头维数的平方根,做数值缩放  score /= 8\*\*0.5  #mask遮盖,mask是true的地方都被替换成-inf,这样在计算softmax的时候,-inf会被压缩到0  #mask = [b, 1, 50, 50]  score = score.masked\_fill\_(mask, -float('inf'))  score = torch.softmax(score, dim=-1)  #以注意力分数乘以V,得到最终的注意力结果  #[b, 4, 50, 50] \* [b, 4, 50, 8] -> [b, 4, 50, 8]  score = torch.matmul(score, V)  #每个头计算的结果合一  #[b, 4, 50, 8] -> [b, 50, 32]  score = score.permute(0, 2, 1, 3).reshape(-1, 50, 32)  return score  attention(torch.randn(8, 4, 50, 8), torch.randn(8, 4, 50, 8),  torch.randn(8, 4, 50, 8), torch.zeros(8, 1, 50, 50)).shape |

该处的计算过程完全是理论部分的实现，只是把其中的部分数字替换成了实际情况中的数字。需要注意的是，在该函数中计算的已经是多头注意力，为了计算简便，这里把多组Q、K、V组成了一个矩阵输入注意力函数中，再在函数中拆分成多组Q、K、V，最后通过矩阵计算的形式计算多头注意力。

接下来要定义多头注意力计算层，在该层中需要使用批量标准化层，在PyTorch当中主要提供了两种批量标准化的网络层，分别是BatchNorm和LayerNorm，其中BatchNorm按照处理的数据维度分为BatchNorm1d、BatchNorm2d、BatchNorm3d。由于本次的任务是自然语言处理任务，属于一维的数据，所以应该使用BatchNorm1d。

BatchNorm1d和LayerNorm之间的区别，在于BatchNorm1d是取不同样本做标准化，而LayerNorm是取不同通道做标准化，可通过如下代码验证。

|  |
| --- |
| #标准化之后,均值是0,标准差是1 #BN是取不同样本做标准化 #LN是取不同通道做标准化 #affine=True,elementwise\_affine=True,指定标准化后,再计算一个线性映射 norm = torch.nn.BatchNorm1d(num\_features=4, affine=True) print(norm(torch.arange(32, dtype=torch.float32).reshape(2, 4, 4))) norm = torch.nn.LayerNorm(normalized\_shape=4, elementwise\_affine=True) print(norm(torch.arange(32, dtype=torch.float32).reshape(2, 4, 4))) |

从结果很显然能够看出，两个标准化层的计算输出虽然都是标准的正态分布，但是BatchNorm1d计算后的数据两个样本的均值都不是0，前一个样本的均值显然小于0，后一个样本的均值显然大于0。 相比较之下，LayerNorm计算后的两个样本均值都在0附近，对于本次的任务而言，选择使用LayerNorm更适合。

明确了要使用的标准化层实现以后，接下来就可以定义多头注意力计算层了，代码如下：

|  |
| --- |
| class MultiHead(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.fc\_Q = torch.nn.Linear(32, 32)  self.fc\_K = torch.nn.Linear(32, 32)  self.fc\_V = torch.nn.Linear(32, 32)  self.out\_fc = torch.nn.Linear(32, 32)  self.norm = torch.nn.LayerNorm(normalized\_shape=32,  elementwise\_affine=True)  self.dropout = torch.nn.Dropout(p=0.1)  def forward(self, Q, K, V, mask):  #b句话,每句话50个词,每个词编码成32维向量  #Q、K、V = [b, 50, 32]  b = Q.shape[0]  #保留下原始的Q,后面要做短接用  clone\_Q = Q.clone()  #标准化  Q = self.norm(Q)  K = self.norm(K)  V = self.norm(V)  #线性运算,维度不变  #[b, 50, 32] -> [b, 50, 32]  K = self.fc\_K(K)  V = self.fc\_V(V)  Q = self.fc\_Q(Q)  #拆分成多个头  #b句话,每句话50个词,每个词编码成32维向量,4个头,每个头分到8维向量  #[b, 50, 32] -> [b, 4, 50, 8]  Q = Q.reshape(b, 50, 4, 8).permute(0, 2, 1, 3)  K = K.reshape(b, 50, 4, 8).permute(0, 2, 1, 3)  V = V.reshape(b, 50, 4, 8).permute(0, 2, 1, 3)  #计算注意力  #[b, 4, 50, 8] -> [b, 50, 32]  score = attention(Q, K, V, mask)  #计算输出,维度不变  #[b, 50, 32] -> [b, 50, 32]  score = self.dropout(self.out\_fc(score))  #短接  score = clone\_Q + score  return score  MultiHead()(torch.randn(8, 50, 32), torch.randn(8, 50, 32),  torch.randn(8, 50, 32), torch.zeros(8, 1, 50, 50)).shape |

这里使用多组WQ、WK、WV矩阵对词向量进行投影，得到多组Q、K、V向量，只是为了便于计算，这里把多组WQ、WK、WV矩阵进行了合并，使用矩阵运算也能提高计算的效率。在这段代码中，首先对词向量进行了标准化计算，这和论文的实现不一致，在Transformer原始论文中的计算顺序是先计算自注意力，再进行短接，然后进行标准化计算。此处把标准化的计算提前了，这样做的原因是因为经过了广泛的实验论证，从实际效果来看标准化前置能更好地保证数值的稳定性，能帮助模型更好地收敛，所以此处选择标准化前置的计算方法，这是一种对Transformer原有模型的改进。

接下来定义位置编码层，代码如下：

|  |
| --- |
| import math class PositionEmbedding(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  #pos是第几个词,i是第几个维度,d\_model是维度总数  def get\_pe(pos, i, d\_model):  d = 1e4\*\*(i / d\_model)  pe = pos / d  if i % 2 == 0:  return math.sin(pe)  return math.cos(pe)  #初始化位置编码矩阵  pe = torch.empty(50, 32)  for i in range(50):  for j in range(32):  pe[i, j] = get\_pe(i, j, 32)  pe = pe.unsqueeze(0)  #定义为不更新的常量  self.register\_buffer('pe', pe)  #词编码层  self.embed = torch.nn.Embedding(39, 32)  #初始化参数  self.embed.weight.data.normal\_(0, 0.1)  def forward(self, x):  #[8, 50] -> [8, 50, 32]  embed = self.embed(x)  #词编码和位置编码相加  #[8, 50, 32] + [1, 50, 32] -> [8, 50, 32]  embed = embed + self.pe  return embed PositionEmbedding()(torch.ones(8, 50).long()).shape |

在这段代码中包括一个内嵌函数get\_pe()，使用该函数计算出位置编码矩阵，位置编码矩阵的尺寸是0×32，因为在本次任务中，文本的长度是50个词，每个词编码成32维的向量。位置编码矩阵本身是一个不更新的常量，所以使用register\_buffer()函数定义为常量。位置编码层的计算过程和理论保持一致，先把每个词编码成普通的词向量，再和位置编码矩阵相加作为最终的词向量编码。

接下来定义全连接输出层，代码如下：

|  |
| --- |
| class FullyConnectedOutput(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.fc = torch.nn.Sequential(  torch.nn.Linear(in\_features=32, out\_features=64),  torch.nn.ReLU(),  torch.nn.Linear(in\_features=64, out\_features=32),  torch.nn.Dropout(p=0.1),  )  self.norm = torch.nn.LayerNorm(normalized\_shape=32,  elementwise\_affine=True)  def forward(self, x):  #保留下原始的x,后面要做短接用  clone\_x = x.clone()  #标准化  x = self.norm(x)  #线性全连接运算  #[b, 50, 32] -> [b, 50, 32]  out = self.fc(x)  #做短接  out = clone\_x + out  return out FullyConnectedOutput()(torch.randn(8, 50, 32)).shape #这里同样使用了标准化层前置的计算方法。 |

5. 定义Transformer模型

做完以上准备工作，现在可以定义编码器层和解码器层了。先看编码器，代码如下：

|  |
| --- |
| #编码器层 class EncoderLayer(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.mh = MultiHead()  self.fc = FullyConnectedOutput()  def forward(self, x, mask):  #计算自注意力,维度不变  #[b, 50, 32] -> [b, 50, 32]  score = self.mh(x, x, x, mask)  #全连接输出,维度不变  #[b, 50, 32] -> [b, 50, 32]  out = self.fc(score)  return out  #编码器 class Encoder(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.layer\_1 = EncoderLayer()  self.layer\_2 = EncoderLayer()  self.layer\_3 = EncoderLayer()  def forward(self, x, mask): Encoder()(torch.randn(8, 50, 32), torch.ones(8, 1, 50, 50)).shape |

在这段代码中，定义了编码器层和编码器，编码器由3层编码器层组成，和理论部分一致，3层编码器是前后串联的关系。编码器层本身的计算是用x同时作为Q、K、V向量计算自注意力，计算得到的注意力分数再输入全连接输出层计算输出。

接下来看解码器的实现，代码如下：

|  |
| --- |
| #解码器层 class DecoderLayer(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.mh1 = MultiHead()  self.mh2 = MultiHead()  self.fc = FullyConnectedOutput()  def forward(self, x, y, mask\_pad\_x, mask\_tril\_y):  #先计算y的自注意力,维度不变  #[b, 50, 32] -> [b, 50, 32]  y = self.mh1(y, y, y, mask\_tril\_y)  #结合x和y的注意力计算,维度不变  #[b, 50, 32],[b, 50, 32] -> [b, 50, 32]  y = self.mh2(y, x, x, mask\_pad\_x)  #全连接输出,维度不变  #[b, 50, 32] -> [b, 50, 32]  y = self.fc(y)  return y #解码器 class Decoder(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.layer\_1 = DecoderLayer()  self.layer\_2 = DecoderLayer()  self.layer\_3 = DecoderLayer()  def forward(self, x, y, mask\_pad\_x, mask\_tril\_y):  y = self.layer\_1(x, y, mask\_pad\_x, mask\_tril\_y)  y = self.layer\_2(x, y, mask\_pad\_x, mask\_tril\_y)  y = self.layer\_3(x, y, mask\_pad\_x, mask\_tril\_y)  return y Decoder()(torch.randn(8, 50, 32), torch.randn(8, 50, 32), *torch.ones(8, 1, 50, 50), torch.ones(8, 1, 50, 50)).shape* |

解码器和编码器的计算过程大致相同，第1步是以y同时作为Q、K、V向量计算自注意力。接下来就是解码器和编码器计算的不同点，多了一层编解码注意力层的计算。这一层的计算也是多头注意力的计算，只是入参的Q向量替换成了上一步计算得到的y的自注意力分数，K和V向量则是使用从编码器那里获得的x的注意力分数。

最后把编解码注意力层计算得到的注意力分数输入全连接输出层计算输出。有了编码器和解码器就可以定义Transformer主模型了，代码如下：

|  |
| --- |
| #定义主模型 class Transformer(torch.nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.embed\_x = PositionEmbedding()  self.embed\_y = PositionEmbedding()  self.encoder = Encoder()  self.decoder = Decoder()  self.fc\_out = torch.nn.Linear(32, 39)  def forward(self, x, y):  #[b, 1, 50, 50]  mask\_pad\_x = mask\_pad(x)  mask\_tril\_y = mask\_tril(y)  #编码,添加位置信息  #x = [b, 50] -> [b, 50, 32]  #y = [b, 50] -> [b, 50, 32]  x, y = self.embed\_x(x), self.embed\_y(y)  #编码层计算  #[b, 50, 32] -> [b, 50, 32]  x = self.encoder(x, mask\_pad\_x)  #解码层计算  #[b, 50, 32],[b, 50, 32] -> [b, 50, 32]  y = self.decoder(x, y, mask\_pad\_x, mask\_tril\_y)  #全连接输出,维度不变  #[b, 50, 32] -> [b, 50, 39]  y = self.fc\_out(y)  return y model = Transformer() model(torch.ones(8, 50).long(), torch.ones(8, 50).long()).shape |

在主模型中，初始化了两个位置编码层，分别用来编码x和y，计算的流程如下：

(1)获取一批x和y之后，对x计算PAD MASK，对y计算上三角MASK。

(2)对x和y分别编码。

(3)把x输入编码器计算输出。

(4)把编码器的输出和y同时输入解码器计算输出。

(5)将解码器的输出输入全连接输出层计算输出。

从上面的叙述可以看出，Transformer主模型的计算需要同时输入x和y。原因是我们要使用强制教学的方法训练Transformer模型，所以在计算时需要同时输入x和y。 但是在预测时只有x数据，没有y数据，所以需要定义一个额外的预测函数，这个函数不使用强制教学，所以不需要y，而是使用Transformer本身的能力预测句子，代码如下：

|  |
| --- |
| #定义预测函数 def predict(x):  #x = [1, 50]  model.eval()  #[1, 1, 50, 50]  mask\_pad\_x = mask\_pad(x)  #初始化输出,这个是固定值  #[1, 50]  #[[0,2,2,2...]]  target = [vocab\_y['<SOS>']] + [vocab\_y['<PAD>']] \* 49  target = torch.LongTensor(target).unsqueeze(0)  #x编码,添加位置信息  #[1, 50] -> [1, 50, 32]  x = model.embed\_x(x)  #编码层计算,维度不变  #[1, 50, 32] -> [1, 50, 32]  x = model.encoder(x, mask\_pad\_x)  #遍历生成第1个词到第49个词  for i in range(49):  #[1, 50]  y = target  #[1, 1, 50, 50]  mask\_tril\_y = mask\_tril(y)  #y编码,添加位置信息  #[1, 50] -> [1, 50, 32]  y = model.embed\_y(y)  #解码层计算,维度不变  #[1, 50, 32],[1, 50, 32] -> [1, 50, 32]  y = model.decoder(x, y, mask\_pad\_x, mask\_tril\_y)  #全连接输出,39分类  #[1, 50, 32] -> [1, 50, 39]  out = model.fc\_out(y)  #取出当前词的输出  #[1, 50, 39] -> [1, 39]  out = out[:, i, :]  #取出分类结果  #[1, 39] -> [1]  out = out.argmax(dim=1).detach()  #以当前词预测下一个词,填到结果中  target[:, i + 1] = out  return target predict(torch.ones(1, 50).long()) |

如理论部分所描述，在预测函数中，Transformer模型将一个词一个词地预测输出，每预测一个词，就作为下一个词预测的输入使用。

6. 训练和测试

现在Transformer模型已经定义完毕，并且也有了预测函数，现在可以开始训练Transformer模型了，代码如下：

|  |
| --- |
| #定义训练函数 def train():  loss\_func = torch.nn.CrossEntropyLoss()  optim = torch.optim.Adam(model.parameters(), lr=2e-3)  sched = torch.optim.lr\_scheduler.StepLR(optim, step\_size=3, gamma=0.5)  for epoch in range(10):  for i, (x, y) in enumerate(loader):  #x = [8, 50]  #y = [8, 51]  #在训练时,是拿y的每一个字符输入,预测下一个字符,所以不需要最后一个字  #[8, 50, 39]  pred = model(x, y[:, :-1])  #[8, 50, 39] -> [400, 39]  pred = pred.reshape(-1, 39)  #[8, 51] -> [400]  y = y[:, 1:].reshape(-1)  #忽略pad  select = y != vocab\_y['<PAD>']  pred = pred[select]  y = y[select]  loss = loss\_func(pred, y)  optim.zero\_grad()  loss.backward()  optim.step()  if i % 2000 == 0:  #[select, 39] -> [select]  pred = pred.argmax(1)  correct = (pred == y).sum().item()  accuracy = correct / len(pred)  lr = optim.param\_groups[0]['lr']  print(epoch, i, lr, loss.item(), accuracy)  sched.step() train() |

在这段代码中，定义了学习率衰减器，每训练3个轮次，则学习率减半，但是由于本次要训练的任务复杂度太低，所以只需训练1个轮次就可以了，没有机会应用到学习率衰减。 在每次计算时，把y的最后一个词切除，因为在Transformer中计算时，根据y的前一个词预测下一个词，所以不需要最后一个词。这也是在设计数据时，故意让y多一个词的原因，这样在切除一个词以后， 长度刚好和x的长度相等。

而在计算loss 和正确率时， 需要把y 的第1 个词切除， 因为Transformer根据y的前一个词预测下一个词，所以Transformer并没有预测y当中的第1个词，而在设计数据时，y当中的第1个词是确定的 ，这个词也确实没有预测的必要。该过程如图4所示。



图4 y和预测结果的对应关系

训练结束后，可以使用模型预测，代码如下：

|  |
| --- |
| #测试 def test():  for i, (x, y) in enumerate(loader):  break  for i in range(8):  print(i)  print(''.join([vocab\_xr[i] for i in x[i].tolist()]))  print(''.join([vocab\_yr[i] for i in y[i].tolist()]))  print(''.join(  [vocab\_yr[i] for i in predict(x[i].unsqueeze(0))[0].tolist()])) test() |