什么是 Seq2seq

Seq2seq 就是把一段输入序列挖掘提取特征“编码”存储到中间量里，然后根据中间量，然后训练“解码”输出想要的结果。

这里举两个例子：

- 机器翻译：把一种语言翻译成另一种语言
- 语音识别：把一段语音识别出来，用文字表示

从这两个例子可以看出，输入的是一段序列（一种语言文字和一段语音），**（经过中间向量），**然后输出也是一段序列（另一种语言文字和和语音对应的文字），即 Sequence-to-sequence。

所谓的 Sequence2Sequence 任务主要是泛指一些 Sequence 到 Sequence 的映射问题，Sequence 在这里可以理解为一个字符串序列，当我们在给定一个字符串序列后，希望得到与之对应的另一个字符串序列（如 翻译后的、如语义上对应的）时，这个任务就可以称为 Sequence2Sequence 了。这种结构最重要的地方在于输入序列和输出序列的长度是可变的。

seq2seq 和 RNN 的关系是什么样的

前面介绍了 seq2seq 的任务，那它和我们之前学的 CNN 和 RNN 模型有什么关系呢？Seq2seq 是一个解决任务的框架，像 word2vec 那样，根据不同的任务可以选择不同的编码器和解码器（例如，CNN、RNN、LSTM、GRU 等）。只是在处理序列任务时，一般选用 RNN 系列模型作为 seq2seq 的组件。

编码器和解码器分别对应输入序列和输出序列的两个循环神经网络。在 Seq2Seq 结构中，编码器 Encoder 把所有的输入序列都编码成一个统一的语义向量 Context，然后再由解码器 Decoder 解码。在解码器 Decoder 解码的过程中，不断地将前一个时刻 t-1 的输出作为后一个时刻 t 的输入，循环解码，直到输出停止符为止。

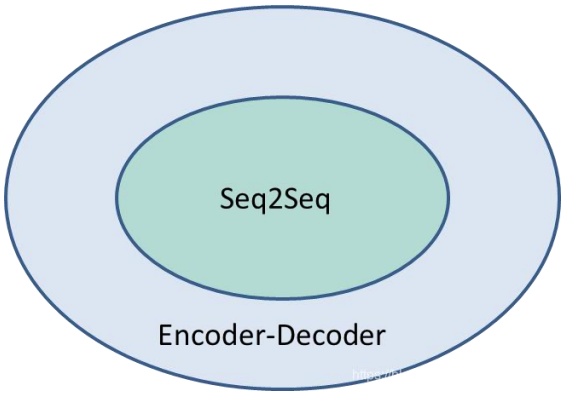
为什么 LSTM 等也可以输入输出不等长，为什么还需要 seq2seq?

Seq2seq 的框架

概述

Seq2seq 的本质是 Encoder-Decoder 结构，Encoder-Decoder 的一个显著特征就是：它是一个 end-to-end 的学习算法。只要符合这种框架结构的模型都可以统称为 Encoder-Decoder 模型。

Seq2Seq 可以看作是 Encoder-Decoder 针对某一类任务（序列任务）的模型框架，它们的范围关系如下所示：



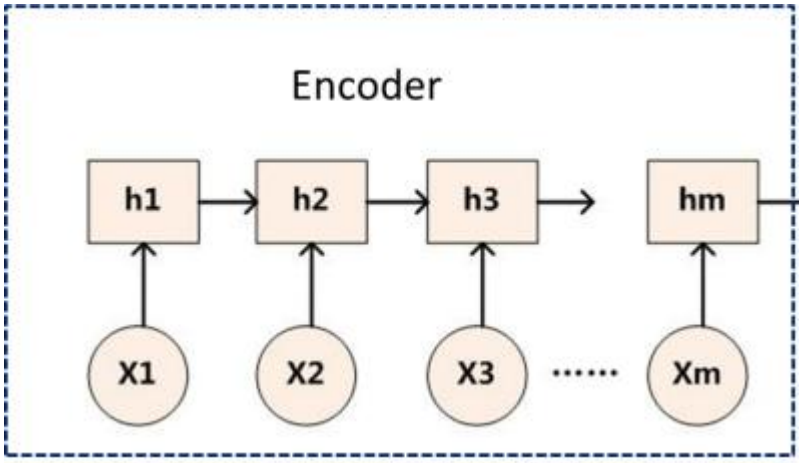
Encoder-Decoder 强调的是模型设计（编码-解码的一个过程），Seq2Seq 强调的是任务类型（序列到序列的问题）。

Seq2seq 模型由三个部分组成：编码器、中间状态向量和解码器。其中中间状态向量是编码器的输出，也是解码器的输入，如何设定中间状态向量也是学习研究的重点之一。

下面具体介绍一下 seq2seq 框架的组成部件，编码器，中间状态向量和解码器。

编码器

编码器的作用是把一个不定长的输入序列转化成一个定长的中间词向量 c0。该中间词向量包含了输入序列的信息。常用的编码器是循环神经网络 RNN 系列模型。



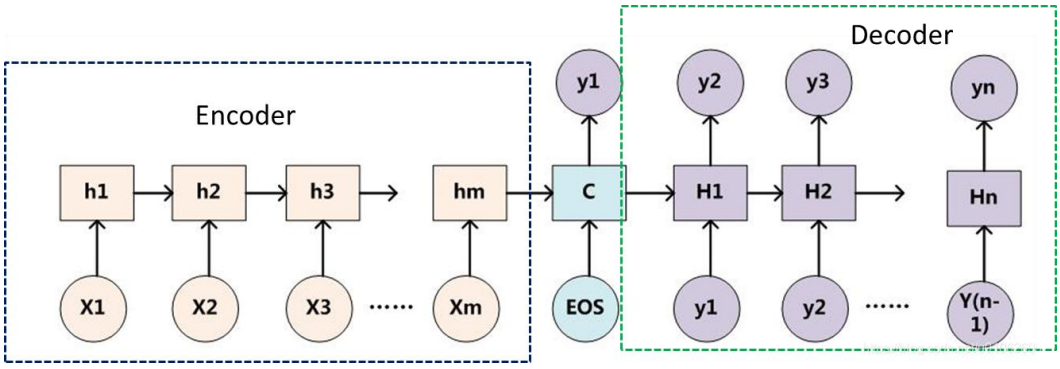
如上图所示，编码器就是一个 RNN 模型，

首先，它的输入同 RNN 系列模型一致，也需要设置输入特征维度，序列长度，批大小等超参数；

隐藏层也是相同的，每个隐藏层的隐状态数和序列长度也是一致的，隐状态特征维度，隐藏网络层数等。

输出层就是根据隐状态变换得到的中间状态向量，因此，中间状态向量也就有了不同的输出形式。

中间状态向量



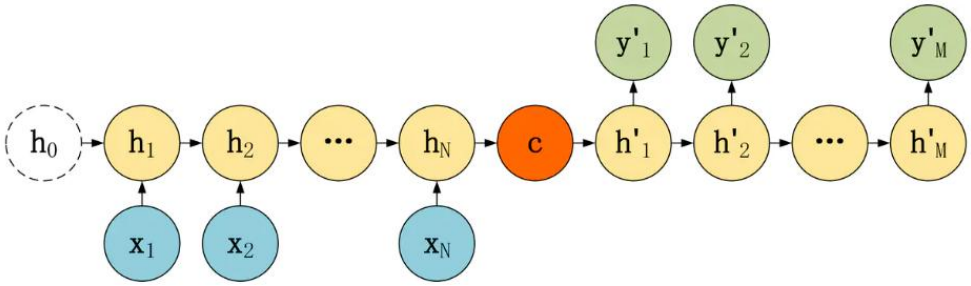
上面讲到，中间状态向量就是编码器模型的输出层，从图中可以看到，最简单的中间状态向量即是编码器的最后一个隐状态向量，当然，也可以将多个隐状态组合变换输出为中间状态向量。后续文章继续介绍如何引入 attention 进行优化。

解码器

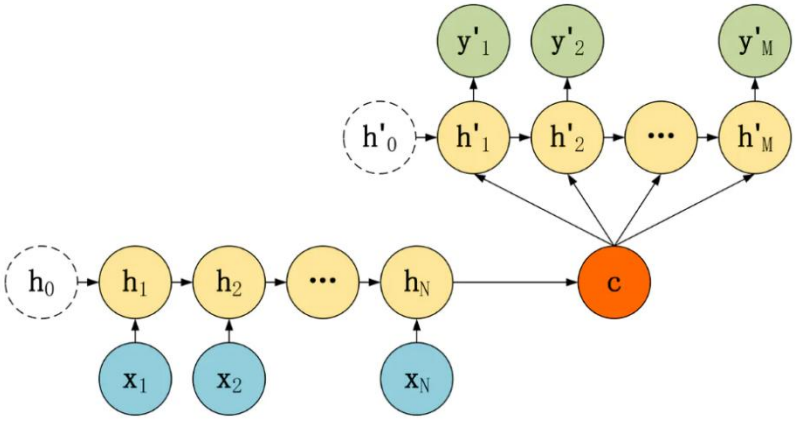
解码器和单纯的 RNN 系列模型相比有些变化，主要体现在输入的选择上，而且在训练和预测时也有不同之处。

一般来说，解码器的输入有以下三种结构：

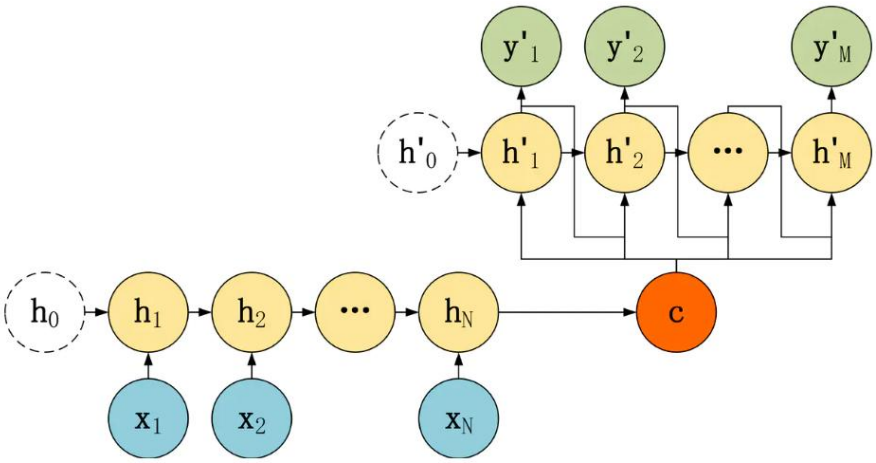
（1）中间状态向量作为最初输入，后面每一个隐状态的输入都是前一个隐状态，如下图。



（2）每一个隐状态的输入都是前一个隐状态和中间状态向量，需要手动设置一个初始隐状态，如下图。



（3）每一个隐状态的输入都是前一个隐状态和前一个输出向量以及中间状态向量，需要手动设置一个初始隐状态，如下图。

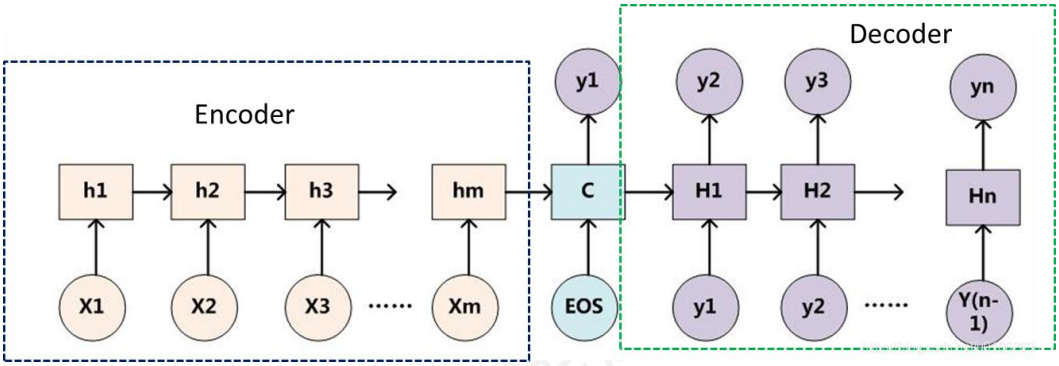


在模型训练和预测时，如果输入是有前一个输出向量的，预测时可以直接使用前一个预测出来的输出值，但是训练时一般使用的是真实文本的输出向量，不然一旦预测错误一个输出值，后面的也都将会受到影响，所谓一步错，步步错。

而使用真实的文本向量，则可以进行纠正，也叫做 teacher forcing。这种操作的目的就是为了使得训练过程更容易。但弊端就是预测时没有 teacher 纠正了，只依靠前面预测输出很容易出现错误。

所以训练时更常用的办法，是部分使用真实文本向量，部分使用模型预测的输出向量。即设置一个概率 p，每一步，以概率 p 靠模型上一步的输出作为输入来预测，以概率 1-p 根据真实文本的输入向量来预测，这种方法称为「计划采样」（scheduled sampling）

信息丢失



不论输入和输出的长度是什么，中间的“向量 c”长度都是固定的（这是它的缺陷所在）。

信息丢失的问题

通过上文可以知道编码器和解码器之间有一个共享的中间向量（上图中的向量 c）来传递信息，而且它的长度是固定的，意味着编码器要将整个序列的信息压缩进一个固定长度的向量中去。

这样做有两个弊端，一是语义向量无法完全表示整个序列的信息，二是先输入的内容携带的信息会被后输入的信息稀释覆盖掉。输入序列越长，这个现象就越严重。这就使得在解码的时候一开始就没有获得输入序列足够的信息，那么解码的准确度自然也就要打个折扣了。这个问题后续介绍注意力方法的时候会有所缓解。

便于理解，我们把“编码-解码”的过程类比为图片“压缩-解压”的过程：将一张 800X800 像素的图片压缩成 100KB，看上去还比较清晰。而将一张 3000X3000 像素的图片也压缩到 100KB，看上去就模糊了。

代码

Model.py

```
import random

import torch
import torch.nn as nn
import random

# seq2seq 框架组件之编码器。这里也会产生输出——中间状态向量。
class Encoder(nn.Module):
    # 输入和 RNN 系列模型一致，这里增加了从原始输入特征维度到词向量维度的 embedding 过程
    def __init__(self,input_dim,emb_dim,hid_dim,n_layers,dropout):
        super(Encoder,self).__init__()
        # 隐藏层的维度，即 h_t,c_t 的维度
        self.hid_dim=hid_dim
        # lstm 的层数
        self.n_layers=n_layers
        # input_dim 即输入的特征维度，emb_dim 即词向量维度，手动设置
        self.embedding=nn.Embedding(input_dim,emb_dim)
        # encoder 层真实的输入维度即词向量维度
        self.rnn=nn.LSTM(emb_dim,hid_dim,n_layers,dropout=dropout)
        self.dropout=nn.Dropout(dropout)

    def forward(self,src):
        # src = [seq_len, batch_size]
#         src = nn.tensor([[2, 2, 2, ..., 2, 2, 2],
#                          [4, 4, 4, ..., 4, 4, 4],
#                          [93, 69, 589, ..., 141, 86, 912],
#                          ...,
#                          [1, 1, 1, ..., 1, 1, 1],
#                          [1, 1, 1, ..., 1, 1, 1],
#                          [1, 1, 1, ..., 1, 1, 1]])
        # torch.Size([33, 128]) 33 为句子长度（填充后的）
        # 对输入的数据进行 embedding 操作
        # embedded = [seq_len, batch size, emb_dim] embed 后的输入张量，包括序列长度，batch 大小，嵌入后的词向量特征维度。
        embedded=self.dropout(self.embedding(src))
        # outputs = [src_len, batch_size, hid_dim * n_directions]
        # hidden(ht) = [n_layers * n_directions, batch_size, hid_dim]
```

```
        # cell(ct) = [n_layers * n_directions, batch_size, hid_dim]
        outputs,(hidden,cell)=self.rnn(embedded)
        return hidden, cell

class Decoder(nn.Module):
    def __init__(self,output_dim,emb_dim,hid_dim,n_layers,dropout):
        super(Decoder,self).__init__()
        self.output_dim=output_dim
        self.hid_dim=hid_dim
        self.n_layers=n_layers
        self.embedding=nn.Embedding(output_dim,emb_dim)
        #注意这里的 emb_dim 就是输入特征词向量处理后的维度，
        self.rnn=nn.LSTM(emb_dim,hid_dim,n_layers,dropout=dropout)
        self.fc_out=nn.Linear(hid_dim,output_dim)
        self.dropout=nn.Dropout(dropout)

    # input = [batch_size]
    # hidden = [n_layers * n_directions, batch_size, hid_dim]
    # cell = [n_layers * n_directions, batch_size, hid_dim]
    # n_directions in the decoder will both always be 1, therefore:
    # hidden = [n_layers, batch_size, hid_dim]
    # context = [n_layers, batch_size, hid_dim]
    def forward(self,input,hidden,cell):
        # input = [1, batch size]
        input=input.unsqueeze(0)
        # embedded = [1, batch size, emb dim]
        embedded=self.dropout(self.embedding(input))
        # output = [seq_len, batch_size, hid_dim * n_directions]
        # hidden = [n_layers * n_directions, batch_size, hid_dim]
        # cell = [n_layers * n_directions, batch_size, hid_dim]
        # seq len and n directions will always be 1 in the decoder, therefore:
        # output = [1, batch size, hid dim]
        # hidden = [n layers, batch size, hid dim]
        # cell = [n layers, batch size, hid dim]
        output,(hidden,cell)=self.rnn(embedded,(hidden,cell))
        # prediction = [batch size, output dim]
        prediction=self.fc_out(output.squeeze(0))
        return prediction, hidden, cell

class Seq2Seq(nn.Module):
    def __init__(self,encoder,decoder,device):
        super(Seq2Seq,self).__init__()
        self.encoder=encoder
        self.decoder=decoder
```

```
            self.device=device
            assert encoder.hid_dim == decoder.hid_dim, \
                "Hidden dimensions of encoder and decoder must be equal!"
            assert encoder.n_layers == decoder.n_layers, \
                "Encoder and decoder must have equal number of layers!"

        # src = [seq_len, batch_size]
        # trg = [trg_len, batch_size]
        # teacher_forcing_ratio is probability to use teacher forcing
        # e.g. if teacher_forcing_ratio is 0.75 we use ground-truth inputs 75% of the time
        def forward(self,src,trg,teacher_forcing_ratio=0.5):
            batch_size=trg.shape[1]
            trg_len=trg.shape[0]
            trg_vocab_size=self.decoder.output_dim
            #
            outputs=torch.zeros(trg_len,batch_size,trg_vocab_size).to(self.device)
            # last hidden state of the encoder is used as the initial hidden state of the
decoder
            hidden,cell=self.encoder(src)
            #decoder 模块的第一个输入是<sos> tokens
            input=trg[0,:]
            for t in range(1,trg_len):
                # insert input token embedding, previous hidden and previous cell
states
                # receive output tensor (predictions) and new hidden and cell states
                output,hidden,cell=self.decoder(input,hidden,cell)
                # place predictions in a tensor holding predictions for each token
                outputs[t]=output
                # decide if we are going to use teacher forcing or not
                teacher_force=random.Random()<teacher_forcing_ratio
                # get the highest predicted token from our predictions
                top1=output.argmax(1)
                #决定输入是根据预测得到的（预测可能是错误的），还是真实值
（起纠偏作用），
                input=trg[t] if teacher_force else top1
            return outputs
```

Train.py

```
import torch
import torch.nn as nn
#import seq2seq  仅仅是把 seq2seq.py 导入进来,当我们创建 seq2seq 的实例的时候
```

需要通过指定 seq2seq.py 中的具体类.
#例如:我的 seq2seq.py 中的类名是 seq2seq,则后面的模型实例化 seq2seq 需要通过 **seq2seq.seq2seq()**来操作
#还可以通过 from 还可以通过 from seq2seq import * 直接把 seq2seq.py 中除了以 _ 开头的内容都导入

```
from models.nlp import seq2seq
from models.nlp.seq2seq import *

import numpy as np
import torch.optim as optim

# S: Symbol that shows starting of decoding input
# E: Symbol that shows starting of decoding output

# P: Symbol that will fill in blank sequence if current batch data size is short than time steps, pad 补充，不够长度就 pad


##    seq_data = [['man', 'women'], ['black', 'white']]
def make_batch():
    input_batch, output_batch, target_batch = [], [], []

    for seq in seq_data:
        for i in range(2):
            seq[i] = seq[i] + 'P' * (n_step - len(seq[i]))    ### 不够长度的 补充 pad
            print(" seq[i] =", seq[i])
        input = [num_dic[n] for n in seq[0]]    ##    seq = ['manPP', 'women']
        output = [num_dic[n] for n in ('S' + seq[1])]
        # output = [num_dic[n] for n in ('S' + 'P' * n_step)]    ## test is ok ?
        target = [num_dic[n] for n in (seq[1] + 'E')]    ### 表示输出结果

        input_batch.append(np.eye(n_class)[input])    ## np.eye(n_class)[input] 生成 one-hot 词向量    5*29
        output_batch.append(np.eye(n_class)[output])
        target_batch.append(target)    # not one-hot

    # make tensor
    return torch.FloatTensor(input_batch), torch.FloatTensor(output_batch), torch.LongTensor(target_batch)


# make test batch 测试数据构建
```

```
def make_testbatch(input_word):
    input_batch, output_batch = [], []

    input_w = input_word + 'P' * (n_step - len(input_word))
    input = [num_dic[n] for n in input_w]
    output = [num_dic[n] for n in ('S' + 'P' * n_step)]

    input_batch = np.eye(n_class)[input]
    output_batch = np.eye(n_class)[output]

    return torch.FloatTensor(input_batch).unsqueeze(0),
torch.FloatTensor(output_batch).unsqueeze(0)

if __name__ == '__main__':

    n_step = 5  ##单词长度，不够的用 padding 补充
    hidden_dim = 128
    emb_dim=25
# 创建一个含有 26 个字母以及 S E P 的字母列表
    char_arr = [c for c in 'SEPabcdefghijklmnopqrstuvwxyz']
    # 创建一个字母列表的字母和下标为键值对的字典
    num_dic = {n: i for i, n in enumerate(char_arr)}
    # 数据序列是 batch_size*seq_len，句子的长度 seq_len 为 2.
    seq_data = [['man', 'man'], ['black', 'white'], ['king', 'queen'], ['girl', 'boy'], ['up',
'down'], ['high', 'low']]

    n_class = len(num_dic)
    batch_size = len(seq_data)

    encoder=Encoder(n_class,emb_dim,hidden_dim,1,0.5)
    decoder=Decoder(hidden_dim,emb_dim,hidden_dim,1,0.5)
    device='cuda' if torch.cuda.is_available() else 'cpu'

    model = Seq2Seq(encoder,decoder,device)
# 设置损失函数
    criterion = nn.CrossEntropyLoss()
    #设置优化器
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
# 预处理数据，将自然语言转换成数字
    input_batch, output_batch, target_batch = make_batch()
    input_batch=input_batch.long()

    for epoch in range(5000):
        # make hidden shape [num_layers * num_directions, batch_size, n_hidden]
```

```
            hidden = torch.zeros(1, batch_size,hidden_dim)    ##  隐层向量初始化
            # pdb.set_trace()

            optimizer.zero_grad()
            # input_batch : [batch_size, max_len(=n_step, time step), n_class]
            # output_batch : [batch_size, max_len+1(=n_step, time step) (becase of 'S'
or 'E'), n_class]
            # target_batch : [batch_size, max_len+1(=n_step, time step)], not one-hot
            print("input_batch.shape:",input_batch.shape)
            output = model(input_batch, hidden, output_batch)
            # output : [max_len+1, batch_size, n_class]
            output = output.transpose(0, 1)    # [batch_size, max_len+1(=6), n_class]
            loss = 0
            for i in range(0, len(target_batch)):
                # output[i] : [max_len+1, n_class, target_batch[i] : max_len+1]
                loss += criterion(output[i], target_batch[i])
            if (epoch + 1) % 1000 == 0:
                print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.6f}'.format(loss))
            loss.backward()
            optimizer.step()

    print(' now is starting test ....')


    # Test
    def translate(word):
            input_batch, output_batch = make_testbatch(word)

            # make hidden shape [num_layers * num_directions, batch_size, n_hidden]
            hidden = torch.zeros(1, 1, n_hidden)    ##  隐层向量初始化
            output = model(input_batch, hidden, output_batch)
            # output : [max_len+1(=6), batch_size(=1), n_class]

            predict = output.data.max(2, keepdim=True)[1]    # select n_class
dimension    get index
            decoded = [char_arr[i] for i in predict]
            end = decoded.index('E')
            translated = ''.join(decoded[:end])

            return translated.replace('P', '')


    print('test')
    print('man ->', translate('man'))
```

```
print('mans ->', translate('mans'))
print('king ->', translate('king'))
print('black ->', translate('black'))
print('ups ->', translate('ups'))
```

参考资料

https://blog.csdn.net/xziyuan/article/details/140060118

Cho et al., Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

Sutskever et al., Sequence to Sequence Leaerning with Neural Networks