1 循环神经网络的基本网络结构

循环神经网络简介

在前面学习的传统神经网络或卷积神经网络，其输入是向量或多维张量且是一次性输入到网络中的，并不考虑先后顺序。但在很多实际问题中，存在着很多序列型的数据（文本、语音，股票以及视频等）。什么是序列信息呢？通俗理解就是一段连续的信息，前后信息之间是有关系的，必须将不同时刻的信息放在一起理解。而且网络的输出也是和多个时刻的输入（甚至整个输入）都是有关系的。

但卷积神经网络并不能处理好这种关联性，不同时刻的输入之间是没有关联的，没有记忆能力，所以前面时刻的输出不能传递到后面的时刻。

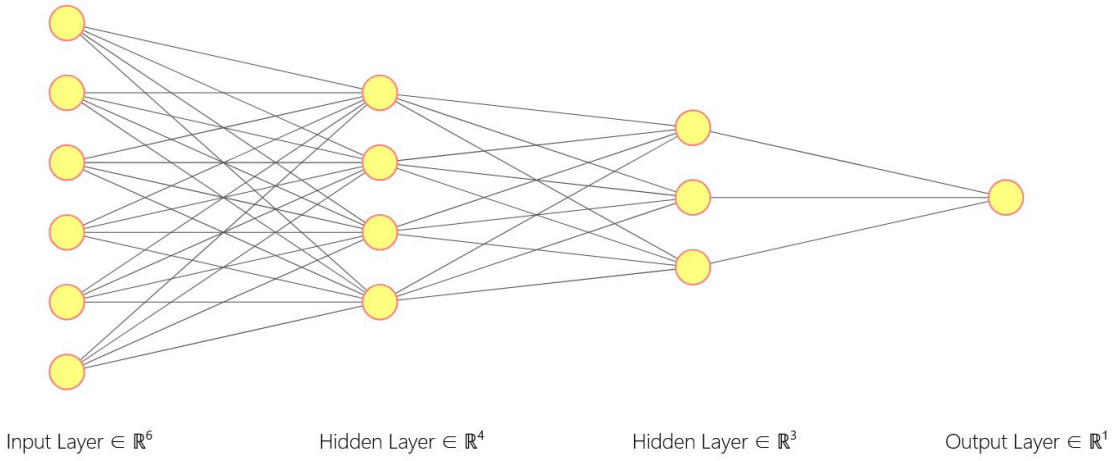
因此，就有了循环神经网络（Recurrent Neural Network，RNN），其本质是：当前的输出同时依赖于当前的输入和之前的输入（可以以不同的形式记忆下来），也就意味着拥有记忆的能力。

比如：我喜欢吃 X，吃是一个动词，按照语法规则，它后面接名词的概率就比较大，在预测 X 是什么的时候就要考虑前面的动词吃的信息，如果没考虑上下文信息而预测 X 是一个动词的话，动词+动词，很大概率是不符合语言逻辑的。

RNN 相比卷积神经网络，为了适应能够记忆前面时序的内容，其结构有诸多变化，往往让初学者感到困惑，下面我们首先详细对比介绍一下 RNN 的网络结构。

前面我们讲到，神经网络的各种结构都是为了挖掘变换数据特征的，所以下面我们也将结合数据特征的维度来对比介绍一下 RNN 的网络结构。

传统神经网络结构



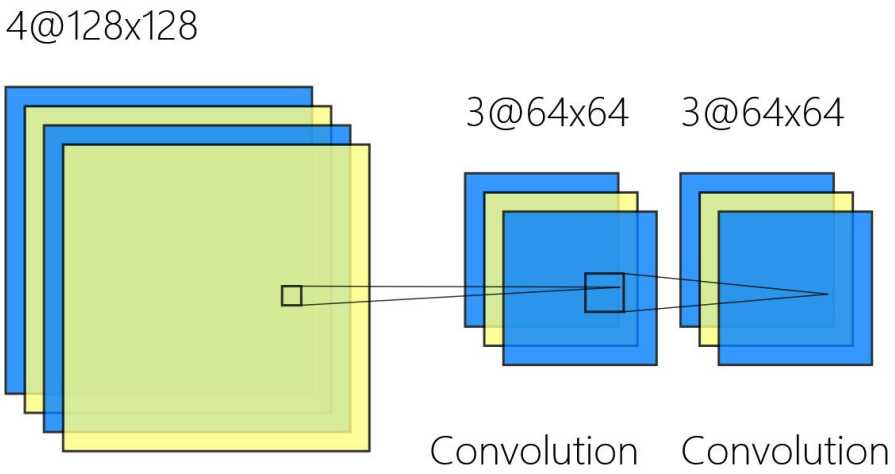
从特征角度考虑：

输入特征：是 n*1 的单维向量（这也是为什么卷积神经网络在 linear 层前要把所有特征层展平），

隐藏层特征：中间的变换特征，即利用参数矩阵将前层输入的特征根据隐藏层神经元的数量 m 表示成 m*1 的单维向量，可以设置多个隐藏层；

输出特征：最终根据输出层的神经元数量 y 输出 y*1 的单维向量。

卷积神经网络结构



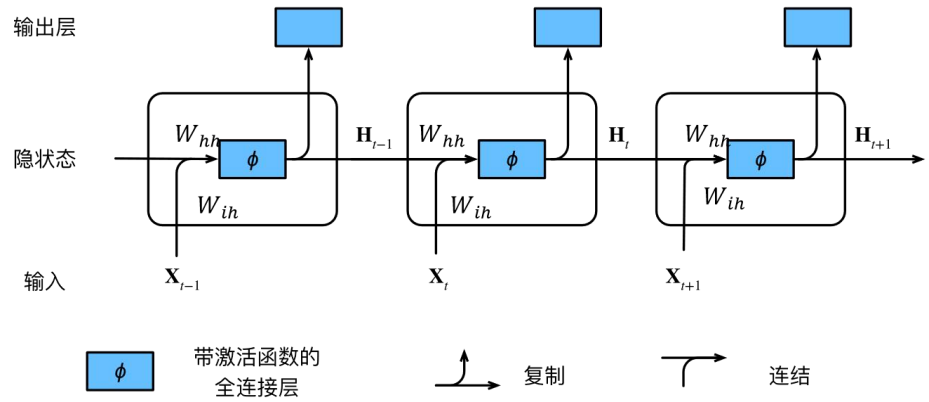
从特征角度考虑：

输入特征：是(batch)*channel*width*height 的张量，

卷积层（等）：然后根据输入通道 channel 的数量 c_in 和输出通道 channel 的数量 c_out 会有 c_out*c_in*k*k 个卷积核将前层输入的特征进行卷积（对特征进行了提取变换，k 为卷积核尺寸），卷积核的大小和数量 c_out*c_in*k*k 就代表网络参数，可以设置多个卷积层；每一个 channel 都代表提取某方面的一种特征，该特征用 width*height 的二维张量表示，不同特征层之间是相互独立的（可以进行融合）。

输出特征：根据场景的需要设置后面的输出，可以是多分类的单维向量等等。

循环神经网络 RNN 结构



从特征角度考虑：

输入特征：是(batch)*T_seq*feature_size 的张量（T_seq 代表序列长度，注意不是 batch_size）.

我们来详细对比一下卷积神经网络的输入特征，

(batch)*T_seq*feature_size

(batch)*channel*width*height，

逐个进行分析，RNN 系列的基础输入特征表示是 feature_size*1 的单维向量，比如一个单词的词向量，比如一个股票价格的影响因素向量，而 CNN 系列的基础输入特征是 width*height 的二维张量；

再来看一下序列 T_seq 和通道 channel，RNN 系列的序列 T_seq 是指一个连续的输入，比如一句话，一周的股票信息，而且这个序列是有时间先后顺序且互相关联的，而 CNN 系列的通道 channel 则是指不同角度的特征，比如彩色图像的 RGB 三色通道，过程中每个通道代表提取了每个方面的特征，不同通道之间是没有强相关性的，不过也可以进行融合。

最后就是 batch，两者都有，在 RNN 系列，batch 就是有多个句子，在 CNN 系列，就是有多张图片（每个图片可以有多个通道）

隐藏层：隐藏层有 T_seq 个隐状态 H_t（和输入序列长度相同），每个隐状态 H_t 是 hidden_size*1 的单维向量，表示着该时刻 t 原来的输入特征从 feature_size*1 的向量变换为了 hidden_size*1 的特征向量，其本质也是数据的一种特征。所以一个隐含层是 T_seq*hidden_size 的张量；

如图中所示，同一个隐含层不同时刻的参数 W_ih 和 W_hh 是共享的；隐藏层可以有 num_layers 个（图中只有 1 个）

以 t 时刻具体阐述一下：

X_t 是 t 时刻的输入，是一个 feature_size*1 的向量

W_ih 是输入层到隐藏层的权重矩阵

H_t 是 t 时刻的隐藏层的值,是一个 hidden_size*1 的向量

W_hh 是上一时刻的隐藏层的值传入到下一时刻的隐藏层时的权重矩阵

Ot 是 t 时刻 RNN 网络的输出

从结构图中我们可以发现 H_t 并不单单只是由 Xt 决定，还与 t-1 时刻的隐藏层的值 H_t-1 有关。

怎么理解这个参数共享呢？

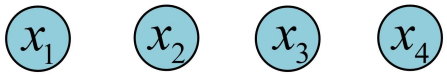
虽然说 X{t-1}，X{t}，X{t+1}是表示不同时刻的输入，但是他们输入到 RNN 网络中的时候并不是作为单独的向量一个一个输入地，而是组合在一起形成一个矩阵输入，然后这个矩阵再通过权重矩阵 U 的变化，其实是同一时刻输入地，只是计算的先后顺序不同。因此同一个隐藏层中，不同时刻的输入他们的 W，V，U 参数是共享地。

输出特征：最终的输出可以根据需要，保留所有隐状态或只保留最后时刻的隐状态。

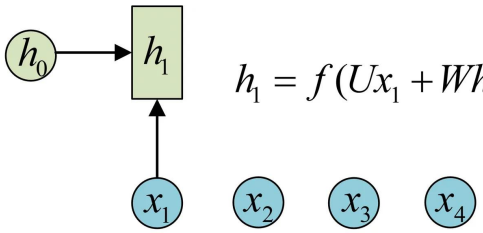
RNN 网络结构的数学公式

前面从概念上介绍了 RNN 网络结构和输入输出隐藏层，接下来具体探索一下其用数学公式的表达。

首先给定一个序列输入句子，假设这个句子有 4 个词语，定义为 x1、x2、x3、x4，



然后我们依次看一下隐状态是如何计算的，先从 h1 的计算开始看：



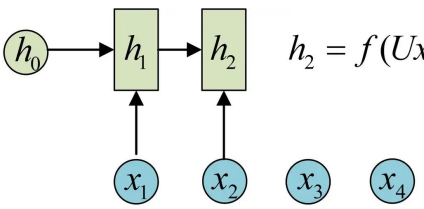
$$h_1 = f(Ux_1 + Wh_0 + b)$$

这里的 h0 是假设给定的先验值，也是需要训练优化的。

图示中记号的含义是：

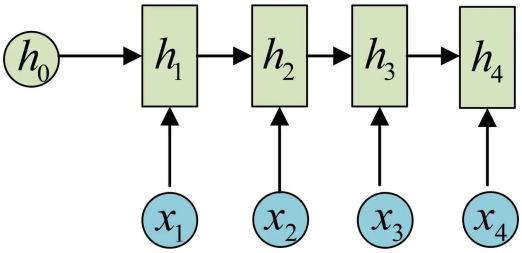
- 圆圈或方块表示的是向量。
- 一个箭头就表示对该向量做一次变换。如上图中 h0 和 x1 分别有一个箭头连接，就表示对 h0 和 x1 各做了一次变换。

h2 的计算和 h1 类似。要注意的是，在计算时，**每一步使用的参数 U、W、b 都是一样的，也就是说每个步骤的参数都是共享的，这是 RNN 的重要特点，一定要牢记。**



$$h_2 = f(Ux_2 + Wh_1 + b)$$

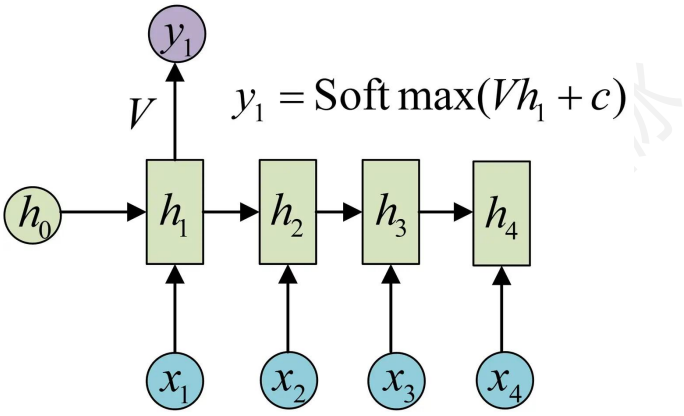
依次计算剩下来的（使用相同的参数 U、W、b）：



这里为了方便起见，只画出序列长度为 4 的情况，实际上，这个计算过程可以无限地持续下去。

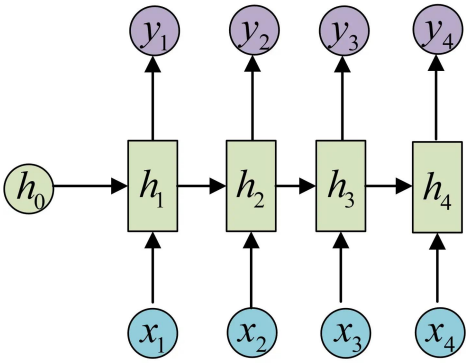
然后这只是计算了一个隐藏层，事实上可以像 CNN 那样，继续把隐藏层的状态作为新的输入特征，继续传递到下一个隐藏层挖掘转换特征。

目前的 RNN 还没有输出，得到输出值的方法就是直接通过 h 进行计算：



正如之前所说，**一个箭头就表示对对应的向量做一次类似于 f(Wx+b)的变换，这里的这个箭头就表示对 h1 进行一次变换，得到输出 y1。**

剩下的输出类似进行（使用和 y1 同样的参数 V 和 c）：



这就是最经典的 RNN 结构，它的输入是 x1, x2, ...xn，输出为 y1, y2, ...yn，也就是说，**输入和输出序列必须要是等长的**。

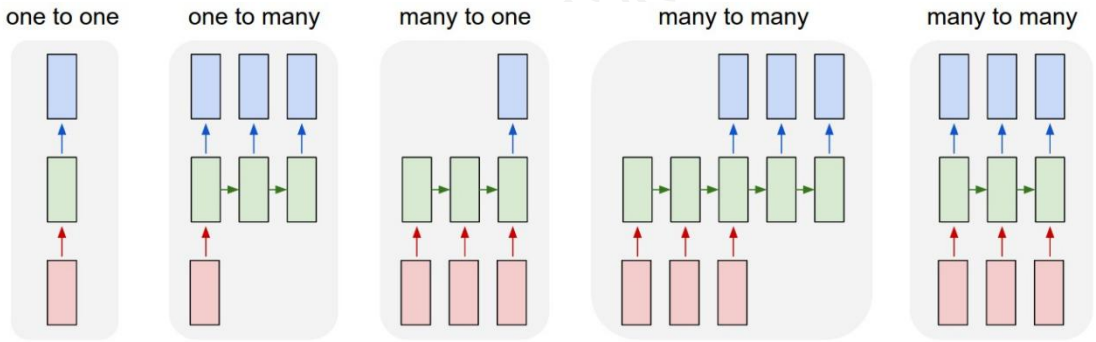
由于这个限制的存在，经典 RNN 的适用范围比较小，但也有一些问题适合用经典的 RNN 结构建模，如：

- 计算视频中每一帧的分类标签。因为要对每一帧进行计算，因此输入和输出序列等长。
- 输入为字符，输出为下一个字符的概率。这就是著名的 Char RNN

- （详细介绍请参考：The Unreasonable Effectiveness of Recurrent Neural Networks，Char RNN 可以用来生成文章，诗歌，甚至是代码，非常有意思）。

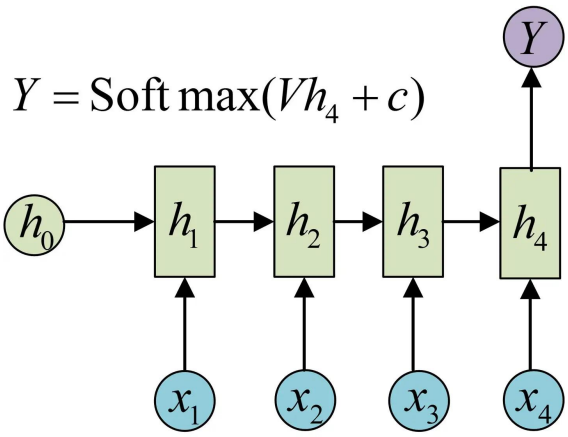
2 RNN 结构分类

前面在介绍 RNN 经典网络结构的时候，我们对比分析了输入输出的基本特征是什么样的，而且输入输出是等长的。在实际应用中，根据输入输出长度的不同，RNN 又分为几种不同的形式，概括起来如下图所示：



N VS 1

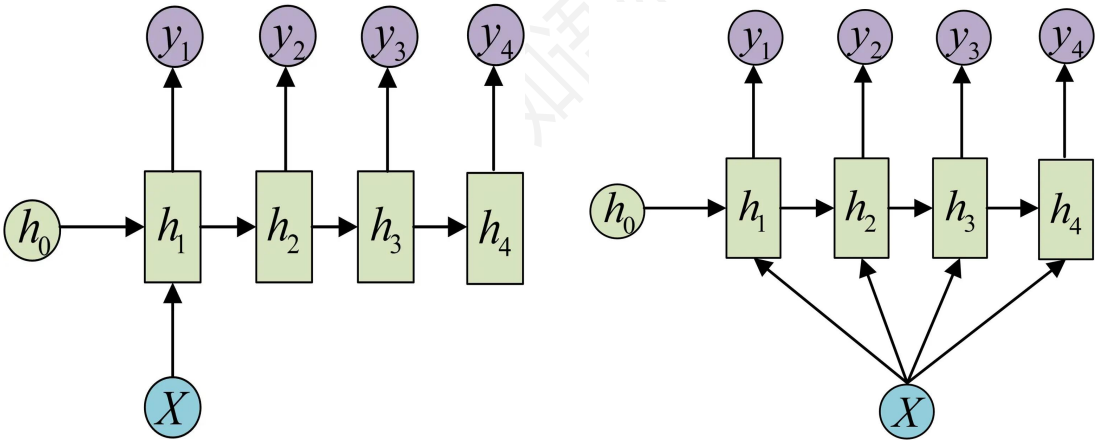
有的时候，我们要处理的问题输入是一个序列，输出是一个单独的值而不是序列，应该怎样建模呢？实际上，我们只在最后一个 h 上进行输出变换就可以了：



这种结构通常用来处理序列分类问题。如输入一段文字判别它所属的类别，输入一个句子判断其情感倾向，输入一段视频并判断它的类别等等。

1 VS N

输入不是序列而输出为序列的情况怎么处理？我们可以只在序列开始进行输入计算，如左图，还有一种结构是把输入信息 X 作为每个阶段的输入，如右图：



这种 1 VS N 的结构可以处理的问题有：

- 从图像生成文字（image caption），此时输入的 X 就是图像的特征，而输出的 y 序列就是一段句子
- 从类别生成语音或音乐等

N vs M

下面先简单介绍 RNN 最重要的一个变种：N vs M。这种结构又叫 Encoder-Decoder 模型，也可以称之为 Seq2Seq 模型。

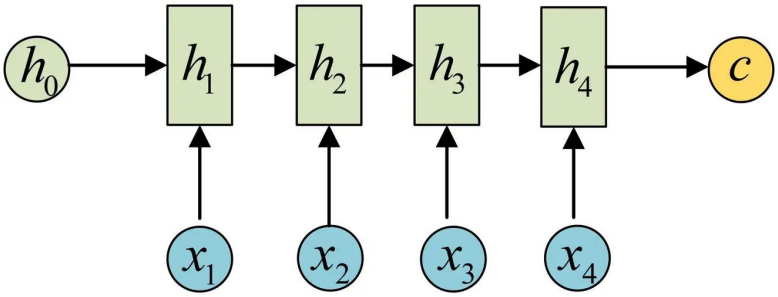
经典的 N vs N RNN 要求序列等长，然而我们遇到的大部分问题序列都是不等长的，如机器翻译中，源语言和目标语言的句子往往并没有相同的长度。

为此，Encoder-Decoder 结构先将输入数据编码成一个上下文中间向量 c：

$$（1）\ c = h_4$$

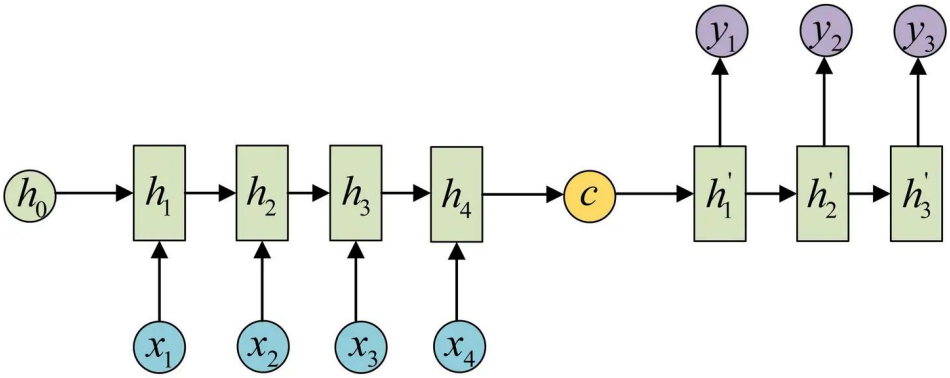
$$（2）\ c = q(h_4)$$

$$（3）\ c = q(h_1, h_2, h_3, h_4)$$

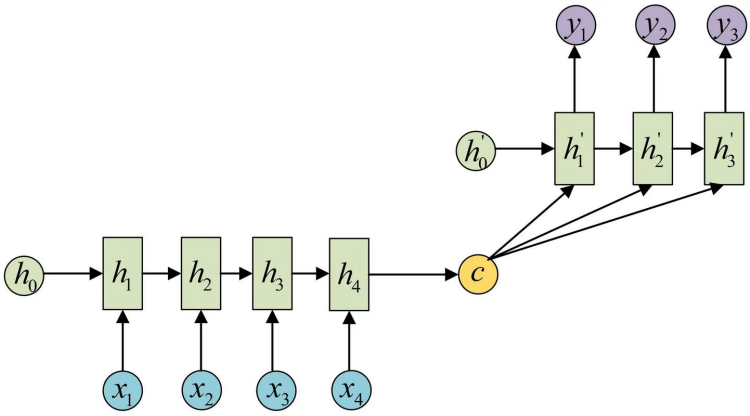


得到 c 有多种方式，最简单的方法就是把 Encoder 的最后一个隐状态赋值给 c，还可以对最后的隐状态做一个变换得到 c，也可以对所有的隐状态做变换。

拿到 c 之后，就用另一个 RNN 网络对其进行解码，这部分 RNN 网络被称为 Decoder。具体做法就是将 c 当做之前的初始状态 h0 输入到 Decoder 中：



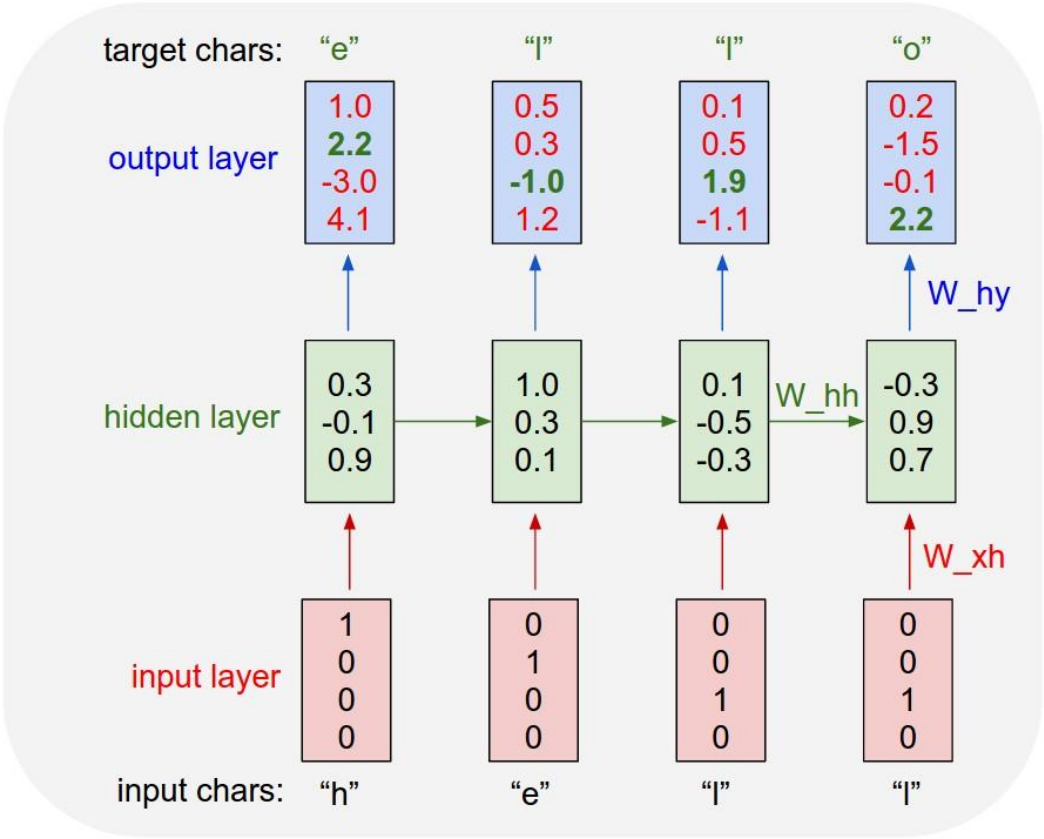
还有一种做法是将 c 当做每一步的输入：



由于这种 Encoder-Decoder 结构不限制输入和输出的序列长度，因此应用的范围非常广泛，比如：

- 机器翻译。Encoder-Decoder 的最经典应用，事实上这一结构就是在机器翻译领域最先提出的
- 文本摘要。输入是一段文本序列，输出是这段文本序列的摘要序列。
- 阅读理解。将输入的文章和问题分别编码，再对其进行解码得到问题的答案。
- 语音识别。输入是语音信号序列，输出是文字序列。
- ……

3 代码



模型里使用两种方法创建，一种是直接调用 pytorch 框架里的模型，熟悉参数及调用流程，另一种是手写了 rnn 函数的具体计算过程，学习内部结构是什么样的。

model.py

```
import torch
import torch.nn as nn

import random
#直接使用 pytorch 自带的 RNN 类
#可以看到，RNN 网络也是继承自 nn.Module 的
class RNN(nn.Module):
    #这里的输入参数包括最开始输入特征“词向量”维度，隐藏层的每个隐状态的特征维度，隐藏层数量，输出层的特征维度（一般和隐状态特征维度一致）
    def __init__(self,feature_size,hidden_size,num_layers,output_size):
        super(RNN,self).__init__()

        self.rnn=nn.RNN(
            input_size=feature_size,hidden_size=hidden_size,
```

```
            num_layers=num_layers,batch_first=True
        )
        #参数初始化，
        for k in self.rnn.parameters():
            nn.init.normal_(k,mean=0.0,std=0.001)
        #linear 层的输入和输出的维度可以是任意的,只需要保证最后一个维度是特征维度 in_features&out_features 就行
        # #- Input: :math:`(*, H_{in})` where :math:`*` means any number of
        #     dimensions including none and :math:`H_{in} = \text{in\_features}`.
        # - Output: :math:`(*, H_{out})` where all but the last dimension
        #     are the same shape as the input and :math:`H_{out} = \text{out\_features}`.
        #
        # Examples::
        # >>> m = nn.Linear(20, 30)
        # >>> input = torch.randn(128, 20)
        # >>> output = m(input)
        # >>> print(output.size())
        # torch.Size([128, 30])
        self.linear=nn.Linear(hidden_size,output_size)
        self.hidden_size=hidden_size

    def forward(self,x,hidden_prev):
        #每一次调用 rnn 层返回的就是输出层和隐状态值，隐状态又是下一循环的上一状态值，所以用 hidden_prev 表示
        out,hidden_prev=self.rnn(x,hidden_prev)
        print("out1&hidden_prev.shape",out.shape,hidden_prev.shape)
        #view()相当于 reshape、resize，重新调整 PyTorch 中的 Tensor 形状，若非 Tensor 类型，可使用 data = torch.tensor(data)来进行转换。
        # out=out.view(-1,self.hidden_size)
        print("out2.shape", out.shape)
        out=self.linear(out)
        print("out3.shape", out.shape)
        # out=out.unsqueeze(0)
        print("out4.shape", out.shape)
        #输出的维度是 batch_size*T_seq*hidden_size
        return out,hidden_prev

#自己实现一个 RNN 函数
#这里的函数参数需要手动给定网络结构参数，
def rnn_forward(input,weight_ih,weight_hh,bias_ih,bias_hh,h_prev):
    #input 的 shape 就是 batch_size*T_seq*feature_size(设置 batch_first=TRUE)
    batch_size,T_seq,feature_size=input.shape
    hidden_size=weight_ih.shape[0]
```

```
    h_out=torch.zeros(batch_size,T_seq,hidden_size)
    for t in range(T_seq):
        x=input[:,t,:].unsqueeze(2)
      #   print("xt.shape",x.shape)
            #unsqueeze，在给定维度上（从 0 开始）扩展 1 个维度，负数代表从后开始数
        #具体到下面，就是先在第 0 维度上扩展成 1*hidden_size*feature_size；
        # 然后.tile 就是在第 0 维度复制 batch_size 次，变成 batch_size*hidden_size*feature_size
        weight_ih_batch=weight_ih.unsqueeze(0).tile(batch_size,1,1)
      #   print("weight_ih_batch.shape", weight_ih_batch.shape)
        weight_hh_batch=weight_hh.unsqueeze(0).tile(batch_size,1,1)
      #   print("weight_hh_batch.shape", weight_hh_batch.shape)

        #计算两个 tensor 的矩阵乘法，torch.bmm(a,b),tensor a  的 size 为(b,h,w),tensor b 的 size 为(b,w,m)
        # 也就是说两个 tensor 的第一维是相等的，然后第一个数组的第三维和第二个数组的第二维度要求一样，其实就是第一维不变，后面二维张量相乘，h*w*w*m=h*m
        # 对于剩下的则不做要求，输出维度 （b,h,m）
        # weight_ih_batch=batch_size*hidden_size*feature_size
        #x=batch_size*feature_size*1
        #w_times_x=batch_size*hidden_size*1
        ##squeeze，在给定维度（维度值必须为 1）上压缩维度，负数代表从后开始数

        w_times_x=torch.bmm(weight_ih_batch,x).squeeze(-1)#
      #   print("w_times_x.shape", w_times_x.shape)

        w_times_h=torch.bmm(weight_hh_batch,h_prev.unsqueeze(2)).squeeze(-1)
      #   print("w_times_h.shape", w_times_h.shape)

        h_prev=torch.tanh(w_times_x+bias_ih+w_times_h+bias_hh)
        print("h_prev.shape", h_prev.shape)
        h_out[:,t,:]=h_prev
        print("h_out.shape", h_out.shape)

    return h_out,h_prev.unsqueeze(0)

if __name__=="__main__":
    # input=torch.randn(batch_size,T_seq,feature_size)
    # h_prev=torch.zeros(batch_size,hidden_size)

    # rnn=nn.RNN(input_size,hidden_size,batch_first=True)
```

```
    # output,state_final=rnn(input,h_prev.unsqueeze(0))

    # print(output)
    # print(state_final)
    batch_size, T_seq =10, 30   # 批大小，输入序列长度
    feature_size, hidden_size = 5, 8   #
    num_layers, output_size=1,3
    input = torch.randn(batch_size, T_seq, feature_size)
    h_prev = torch.zeros(1,batch_size, hidden_size)#.unsqueeze(0)
    # my_rnn=RNN(feature_size,hidden_size,num_layers,output_size)
    rnn=nn.RNN(feature_size,hidden_size,batch_first=True)
    # rnn_output, state_final = rnn(input, h_prev.unsqueeze(0))
    # for k,v in rnn.named_parameters():
    #         print(k,v.shape)

my_rnn_output,my_state_final=rnn_forward(input,rnn.weight_ih_l0,rnn.weight_hh_l
0,

rnn.bias_ih_l0,rnn.bias_hh_l0,h_prev)
    print(my_rnn_output.shape)
    print(my_state_final.shape)
```


train.py

```
import torch
import torch.nn as nn
import numpy as np
#从自己创建的 models 库里导入 RNN 模块
#import RNN  仅仅是把 RNN.py 导入进来,当我们创建 RNN 的实例的时候需要通
过指定 RNN.py 中的具体类.
#例如:我的 RNN.py 中的类名是 RNN,则后面的模型实例化 RNN 需要通过
**RNN.RNN()**来操作
#还可以通过  from  还可以通过  from RNN import *  直接把 RNN.py 中除了以  _
开头的内容都导入
from models.nlp import RNN
from models.nlp.RNN import *

import datetime
import torch.optim as optim
#导入画图的库，后面将主要学习使用 axes 方法来画图
from matplotlib import pyplot as plt
```

```
batch_size=2#批大小
T_seq=30#输入序列长度(时间步）
feature_size=3#输入特征维度

hidden_size=5#隐含层维度
output_size=4#输出层维度

num_layers=1
lr_rate=0.001
epoch=1000
#input 即 RNN 网络的输入，维度应该为(T_seq, batch_size, input_size)。如果设置 batch_first=True，输入维度则为(batch, seq_len, input_size)
input=torch.randn(batch_size,T_seq,feature_size)

def train(input):
    model=RNN(feature_size,hidden_size,num_layers,output_size)
    print("model:\n",model)
    # 设置损失函数
    loss_fn=nn.MSELoss()
    # 设置优化器
    optimizer=optim.Adam(model.parameters(),lr_rate)
    # 初始化 h_prev，它和输入 x 本质是一样的，hidden_size 就是它的特征维度
    #维度应该为(num_layers * num_directions, batch, hidden_size)。num_layers 表示堆叠的 RNN 网络的层数。
    # 对于双向 RNNs 而言 num_directions= 2，对于单向 RNNs 而言，num_directions= 1
    hidden_prev=torch.zeros(1,batch_size,hidden_size)
    loss_plt=[]
    #开始训练
    for iter in range(epoch):
        x = input
        print("x:", x.shape)

        output,hidden_prev=model(x,hidden_prev)
        print("output_size:",output.shape)
        #注意这里的标签，在实际任务的训练中标签往往是下一时刻的数据
        y = torch.randn(batch_size,T_seq,output_size)
        print("y:", y.shape)
        #返回一个新的 tensor，从当前计算图中分离下来的，但是仍指向原变量的存放位置,
        # 不同之处只是 requires_grad 为 false，得到的这个 tensor 永远不需要计算其梯度，不具有 grad。
        hidden_prev=hidden_prev.detach()
```

```
            loss=loss_fn(output,y)
            model.zero_grad()
            loss.backward()
            optimizer.step()
            if iter%100==0:
                print("iteration:{} loss {}".format(iter,loss.item()))
                loss_plt.append(loss.item())

    fig,ax=plt.subplots(1,1)
    ax.plot(loss_plt, 'r')
    ax.set_xlabel('epcoh')
    ax.set_ylabel('loss')
    ax.set_title('RNN-train-loss')

    return hidden_prev, model

if __name__ == '__main__':
    #计算训练时间，结束时间减去开始时间
    start_time = datetime.datetime.now()
    hidden_pre, model = train(input)
    end_time = datetime.datetime.now()
    print('The training time: %s' % str(end_time - start_time))
    plt.show()
```


参考资料

https://zh-v2.d2l.ai/chapter_recurrent-neural-networks/rnn.html
https://zhuanlan.zhihu.com/p/28054589
https://www.bilibili.com/video/BV13i4y1R7jB/?spm_id_from=333.788&vd_source=cf7630d31a6ad93edecfb6c5d361c659
http://karpathy.github.io/2015/05/21/rnn-effectiveness/