

时间线：2024年11月1日 开始

【29、PyTorch RNN的原理及其手写复现】 https://www.bilibili.com/video/BV13i4y1R7jB/?share_source=copy_web&vd_source=5cbbeaf6fa2338b041c25f100ea6483

- 241107 继续
- 241108 继续 19.41done

The screenshot shows the PyTorch documentation for the `RNN` class. The URL is <https://pytorch.org/docs/stable/torch.nn.html#rnn>. The page title is "RNN". The `CLASS torch.nn.RNN(*args, **kwargs) [SOURCE]` section describes the class as applying a multi-layer Elman RNN with `tanh` or `ReLU` non-linearity to an input sequence. It details the computation function: $h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$, where h_t is the hidden state at time t , x_t is the input at time t , and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0. If `nonlinearity` is `'relu'`, then `ReLU` is used instead of `tanh`. The `Parameters` section lists the following parameters:

- `input_size` – The number of expected features in the input x
- `hidden_size` – The number of features in the hidden state h
- `num_layers` – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a stacked RNN, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- `nonlinearity` – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- `bias` – If `False`, then the layer does not use bias weights b_{ih} and b_{hh} . Default: `True`
- `batch_first` – If `True`, then the input and output tensors are provided as `(batch, seq, feature)` instead of `(seq, batch, feature)`. Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`

topic: 循环神经网络的教程、原理、及其实现

分类、模型的类别，模型的优缺点，以及应用场景四个部分进行展开

接下来展示不同类型的RNN的图示，以及应用场景的图示，再介绍pytorch中RNN的api的使用，最后通过代码验证 RNN 内部是如何计算的，通过代码来验证 pytorch的RNN的api 并对比结果。



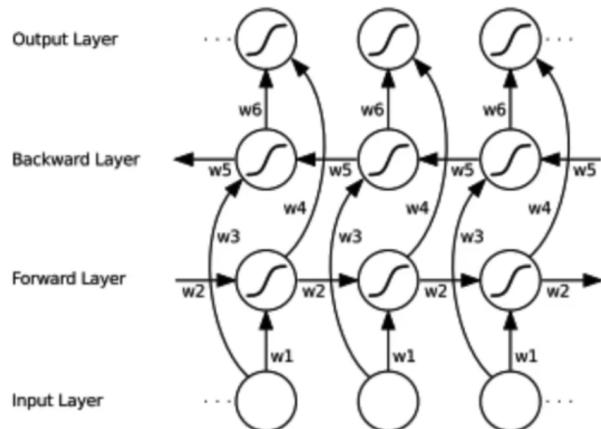
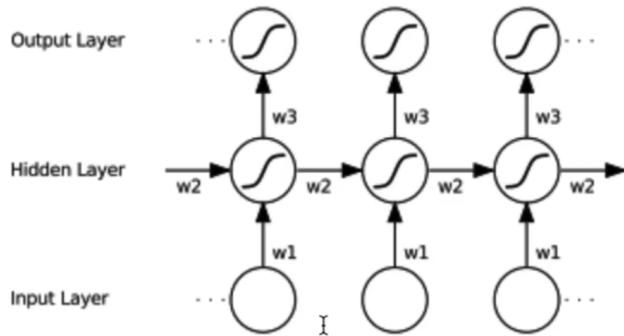
首先，循环神经网络有一个很重要的元素叫做：记忆单元。记忆单元就是存储的过去的历史信息。所谓循环神经网络就是说，在对序列进行建模的时候，在算每一时刻的表征的时候，一般考虑过去的历史信息。这个历史信息就是通过记忆单元保存的。然后每个时刻我们都会从记忆单元中获取过去的历史信息，然后辅助当前时刻做预测。关于记忆单元一般有三类，一类比如说 RNN，比如说 Simple RNN，简单的RNN 结构，等下实现的也是简单的RNN结构，另外两种是 GRU和LSTM，这两种网络的记忆性会更强一点；计算复杂度也会更高一点；使用频率也会更高一点，就是说现在很多的实际应用中，我们基本使用的是 LSTM或者GRU；但是它们都是RNN的一个变体，所以RNN是基础；

以上是记忆单元的分类，分为三种，从模型上分类，也可以分为以下三种

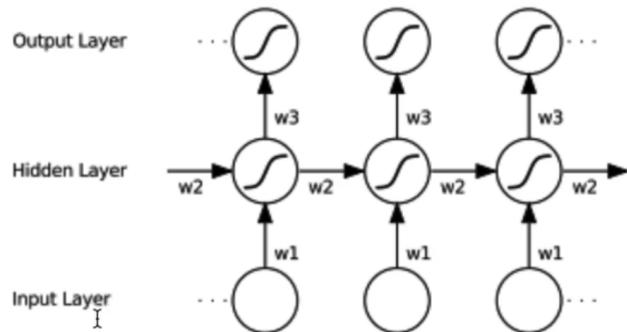


循环神经网络也可分为单向循环，所谓单向循环就是，当前时刻的预测只跟过去有关，从左到右递归的计算。也有双向循环，双向循环就是说不只有从左到右的也有从右到左的，就是说有两条链，另外一条链，在计算当前时刻的预测的时候会考虑未来信息。这个就是双向循环；那还可以把多个单向或者说多个双向叠加起来，也就是deep RNN 深度循环神经网络

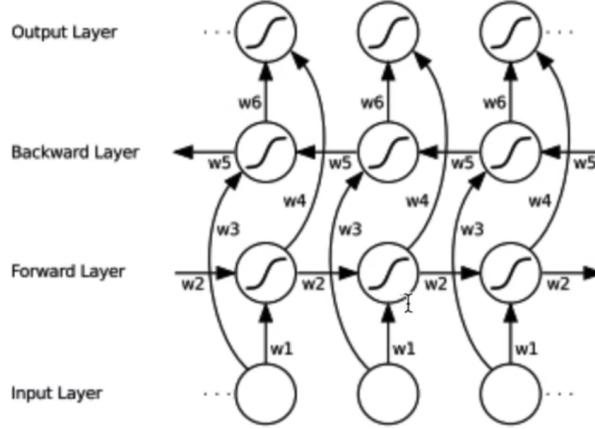
接下来，看几张图示



很好的说明了刚刚讲的几种模型类别，第一幅图就是单向的循环神经网络：



可以分为三层，最下面一层是 input layer，也就是输入层；中间是隐含层；最后是输出层；下面的输入层每一个神经元可以看做每一个时刻；也就是说每一个时刻不仅跟当前时刻的输入有关，还跟上一时刻的记忆单元有关；并且在单向循环神经网络中始终是从左到右的；就是说当前时刻的预测只跟过去的记忆单元有关，跟未来的无关的；以上是单向的 RNN 循环神经网络；下面展示的是双向的



可以看到有两条链，除了有input layer、output layer之外，还有forward layer和backward layer；所谓forward layer是从左到右的循环，意思就是说在forward layer的输出中，它的输出不仅跟当前输入有关也跟过去的记忆单元有关；那么在backward layer当中，它的当前时刻的输出不仅跟当前时刻的输入有关，还跟未来时刻的记忆单元有关，所以是从右到左的递归运算的。那么通过将forward和backward结合起来有什么好处呢？就是说既能看到过去又能看到未来

Table 5.1: **Framewise phoneme classification results on TIMIT.** The error measure is the frame error rate (percentage of misclassified frames). BLSTM results are means over seven runs \pm standard error.

Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990
RNN (delay 0)	30.1	35.5	120
LSTM (delay 0)	29.1	35.4	15
LSTM (backwards, delay 0)	29.9	35.3	15
RNN (delay 3)	29.0	34.8	140
LSTM (delay 5)	22.4	34.0	35
BLSTM (Weighted Error)	24.3	31.1	15
BRNN	24.0	31.0	170
BLSTM	22.6 ± 0.2	30.2 ± 0.1	20.1 ± 0.5
BLSTM (retrained)	21.4	29.8	17

这张表格来自某篇论文，这张表格很好的展示了RNN、LSTM、双向单向、MLP、以及是否delay等在参数数量相等的情况下在语音识别上的表现；可以看到第二列第三列分别是训练误差和测试误差；

来一个个解释通过表格可以看到不同的模型在语音识别这种序列建模，序列分类这个任务上的表现

第一行是MLP，MLP就是简单的DNN是no window的，什么意思呢？就是我们把语音分成很多帧，比方说一帧是15毫秒或者20毫秒，对于每一帧提取一个特征比如说傅里叶变换得到一个频谱特征，然后我们对每一帧进行单独建模，所谓no window就是我们不考虑周围的帧，只考虑当前这个15毫秒，然后我们把它送入DNN中，来去进行一个预测分类，这样做的话它的训练误差和测试误差大概都是在40%左右；

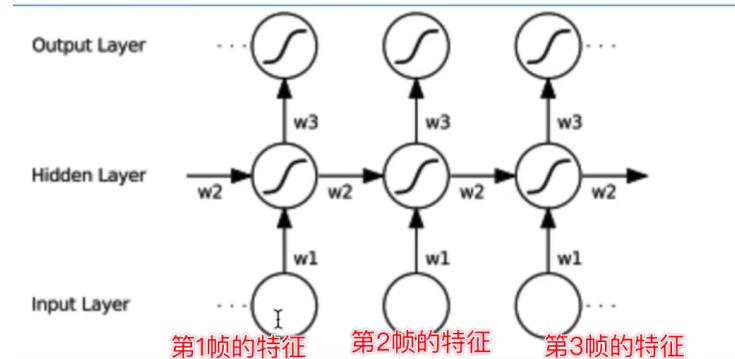
Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990

第二行 MLP 10帧作为一个窗意思是我们现在同样还是MLP，但是现在MLP的输入不仅是只有一帧的特征，而是把每10帧放到一起，那么这里是否有stride up主也不清楚，就是说这10帧到底有没有交叠并没有介绍，总之第二行这个输入比第一行覆盖的时间窗口会更大一点；那么这样可以看到这个误差，显著的从 46% 降到 32%，这个结果说明在语音识别这个序列建模任务中，当我们把上下文特征一起考虑的话效果会更好；这是第二行。

Table 5.1: **Framewise phoneme classification results on TIMIT.** The error measure is the frame error rate (percentage of misclassified frames). BLSTM results are means over seven runs \pm standard error.

Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990
RNN (delay 0)	30.1	35.5	120

第三行，将MLP换成了循环神经网络，一个简单的RNN 模型，并且括号 delay 0，等下会解释什么叫delay，这里的意思就说，就是说把每一帧特征像第一幅图一样，比如说



这里是第一帧的特征，这里是第二帧的特征，这里是第三帧的特征，我们把每一帧的特征送入到RNN中，通过中间的隐含层对历史信息进行更新，这样的网络错误率也是相比MLP更进一步，看到训练误差到30%，测试误差是35%，相比于上面 10帧的MLP，效果更好。

接下来如果我们把RNN，替换成LSTM，效果更进一步

Table 5.1: **Framewise phoneme classification results on TIMIT.** The error measure is the frame error rate (percentage of misclassified frames). BLSTM results are means over seven runs \pm standard error.

Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990
RNN (delay 0)	30.1	35.5	120
LSTM (delay 0)	29.1	35.4	15

以上都是delay 0

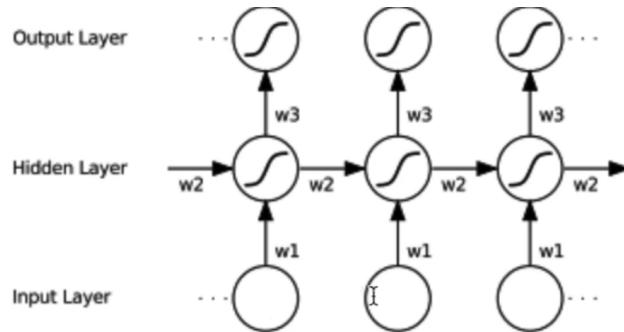
再下面一步，还是LSTM，只是把输入翻转过来，也就是把input序列倒过来，再输入到网络中，误差是差不多的，所以我们仅仅是一条链的话，所以不论是正向识别，还是反向识别其实效果是差不多的

LSTM (delay 0)	29.1	35.4	15
LSTM (backwards, delay 0)	29.9	35.3	15

再下面 我们对输入进行改造，首先可以看到 同样是用 RNN网络，这里我们对它 进行 delay 三帧，然后可以看到 它的效果 相比于原本的 RNN

Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990
RNN (delay 0)	30.1	35.5	120
LSTM (delay 0)	29.1	35.4	15
LSTM (backwards, delay 0)	29.9	35.3	15
RNN (delay 3)	29.0	34.8	140

从30% 降低到 29%，测试误差 也是从 35% 降低到 34%；那么这个 delay 3 帧是什么意思呢？



delay 3 帧的意思就是说，当我们喂入 三帧 作为输入的时候，我们前面 这三个输出，我们先不要，

我们就是说 我们先拿 三帧输入 送入到网络中 让它先对记忆单元去 更新三步，然后到第四步（帧）的 输入的时候，我们才把 输出拿出来，作为 第一帧的 预测值，这个就是delay 3 的意思【为什么有效？有什么用】

那么这样的话，好处就是说 我们如果 不做delay的话 我们在 输入 第一帧的 特征的时候，它的预测的输出 只能 看到当前的第一帧，范围就很小，但是当我们 delay 三帧的时候 我们预测第一帧的输出 其实就看到了 三帧，它看到了 第一帧、第二帧、第三帧 都进入了 记忆单元中；以上就是 delay RNN的结构；

那么很多时候 通过这种 delay 能够在 短暂的 牺牲 时延的情况下，我们知道既然有 delay 的话，那肯定在预测第一帧的输出的时候 肯定会 稍微 延迟一点，因为 如果 不做 delay 的话，我们就直接 算一步就好了，如果delay 三帧的话，那我们在预测第一帧的输出的时候，需要计算 三步，所以肯定会有一 定时延的。但是这个时延 确实能够 使得 预测的效果更好，

因为它看到的上下文会更宽一点；以上是delay的意思。

BLSTM (Weighted Error)	24.3	31.1	15
BRNN	24.0	31.0	170
BLSTM	22.6±0.2	30.2±0.1	20.1±0.5
BLSTM (retrained)	21.4	29.8	17

下面都换成了双向的LSTM、RNN

Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990
RNN (delay 0)	30.1	35.5	120
LSTM (delay 0)	29.1	35.4	15
LSTM (backwards, delay 0)	29.9	35.3	15
RNN (delay 3)	29.0	34.8	140
LSTM (delay 5)	22.4	34.0	35
BLSTM (Weighted Error)	24.3	31.1	15
BRNN	24.0	31.0	170
BLSTM	22.6±0.2	30.2±0.1	20.1±0.5
BLSTM (retrained)	21.4	29.8	17

同时可以看到 RNN delay三帧 和LSTM delay 五帧 效果都有不同程度的增加；

可以看到 双向的结果比delay 和 单向的效果都要好；训练集 错误率从29%降低到24%； 测试集错误率也是明显降低；

双向、delay; delay 3帧 虽然我们也看到了未来的信息；当我们 delay三帧的话，我们在预测第一帧的输出的时候 其实我们是看到了第二帧 第三帧 和 第四帧 指的是 看到了未来的三帧的（意义是什么？ delay的含义还没搞明白）

当我们预测 第二帧的输出的时候 同样 看到第三帧、第四帧、第五帧，这样的。虽然也看到了未来的信息，但看到未来的信息还是不够长；

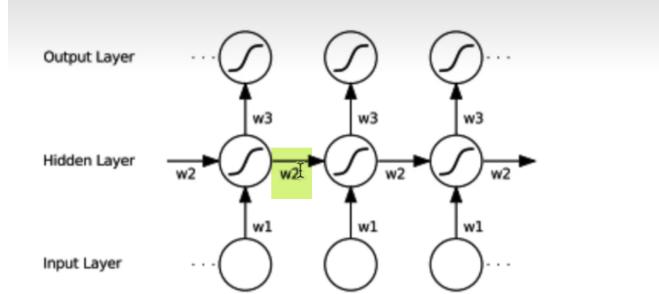
如果我们把单向换成双向的网络的话，那么整个未来的特征 和 过去的 特征，我们的网络都能看到，这就是说双向的范围更大一点；（未来 预测现在的，不明白）

但是双向也有缺点，就是说 完全的 把整个输入特征序列 送入到网络中，最后才能得到输出；而单向带时延的情况我们就不需要把整个特征 都算出来 才能预测第一帧，只要有三帧了，我们就可以预测第一帧了；所以单向带时延的，响应速度会更快；双向的响应速度 肯定是最慢的，所以在速度 和效果上 我们需要 取得一个比较好的平衡 才能满足具体的业务需求。

以上是 模型的类别。接下来，循环神经网络的优缺点

优点可以处理变长序列，这个是DNN和CNN处理不了的，比如DNN，输入的特征是固定的，而CNN的不仅和kernel size有关，还跟输入的通道数有关，所以如果CNN 输入通道数有变化的话 还需要重新搭建一个网络，而RNN 是可以处理变长序列的；

为什么RNN能处理变长序列呢？原因是因为，可以看到下图中有一个W



也就是权重，这个W在每个时刻都是相等的，正是因为所有的权重，在每一个时刻都是相等的；不论是输入跟既有单元的连接，还是历史信息跟当前的神经元的连接它的权重都是固定的，正是因为权重在每一时刻共享，所以RNN能够处理变长序列；(amazing)

一旦去掉了权重共享这个归纳偏置的话，那意思就是说，每一时刻都有一个不一样的w的话，这个时候就不能处理变长序列了，就类似position embedding一样，只要遇到了长度比训练集大的，那就处理不了了；

模型大小与序列长度无关

第二点，模型的大小与序列长度无关，这里说的是模型的大小，也就是说模型的参数数量与长度无关，等会儿可以看到模型的全部参数和序列长度都是无关的，只跟我们的输入特征和输入通道数以及RNN的隐含单元有关

计算量与序列长度呈线性增长

第三个优点就是RNN的计算量跟序列长度呈线性增长，最近我们也说过了Transformer，在原本的Transformer中最大的一个诟病的地方就是计算复杂度跟序列长度是呈一个平方关系的，但是在我们的RNN中，我们的计算量是跟长度呈现线性增长的；

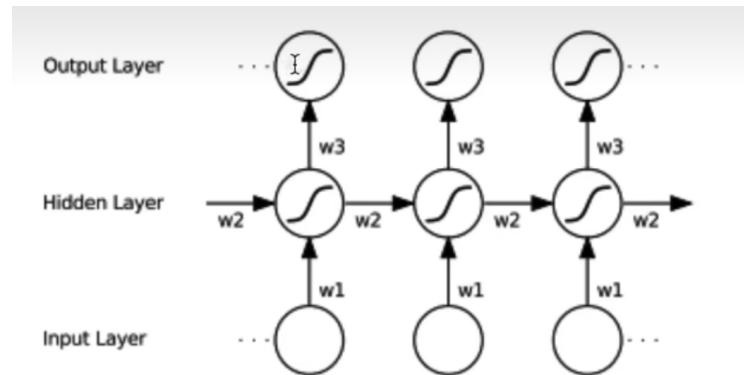
也就是说当你的序列长度为2的时候，我们的计算量可能就是一个 $2t$ （t指的是时间？总是模型固有的计算量）当序列长度为3的时候，我们的计算量就是 $3t$ ，就不是说从4变成9，呈现平方关系。在RNN中呈现线性关系；这是跟Transformer在计算量上一个明显的区别。

考虑历史信息

另外一个优点就是相比DNN而言，就是RNN是可以考虑到历史信息的，正是因为我们有一个链式的结构，我们通过隐含层来积累历史信息；

便于流式输出

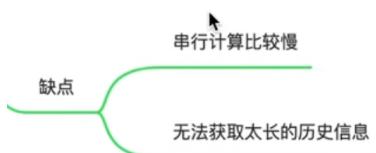
第五个优点就是便于流式的输出，这也是相比Transformer和RNN的一个优点。所谓流式输出，我们可以看到：



就是我们每计算一步，都可以得到一个输出，那么这个输出就可以直接送给用户，这就是流式的意思。但是对于Transformer而言的话，由于它是考虑到全局的信息计算一个全局的self attention，所以就不能单步的计算每一步的输出，这就是Transformer的一个缺点，不能直接的应用到流式的场景；但是在循环神经网络中，只要每算一次递归运算，就可以得到一个输出，这个输出就可以直接返回给用户，这就是流式的，也就是我们不需要把整个序列都算完才返回给用户，而是说我们每算出一个时刻都可以返回给用户

权重时不变

最后一个优点就是权重是时不变的，正是因为RNN 权重时不 变，所以RNN 可以处理变长序列；



当然RNN也有缺点，每个模型都既有优点又有缺点；

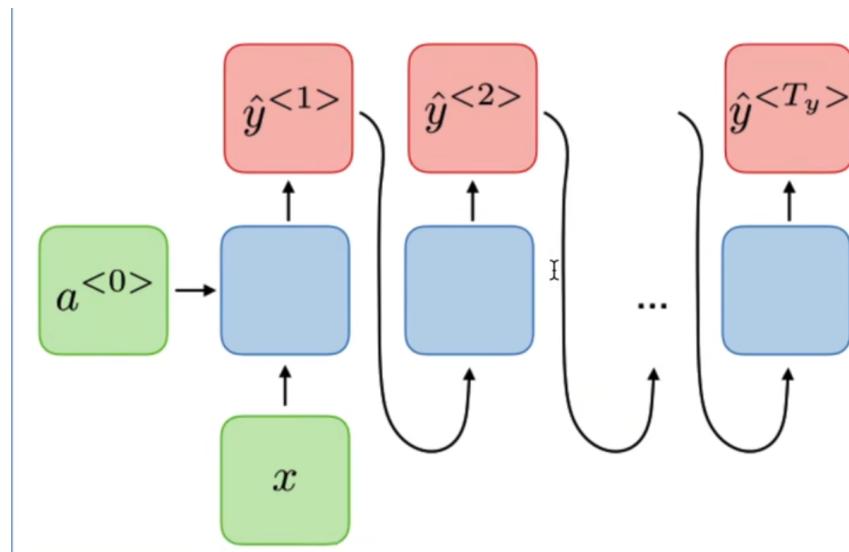
RNN的缺点就是第一个串行计算比较慢；就是说我们在算每一时刻的时候都需要等上一时刻的历史信息，等上一时刻的算出来才能算下一时刻，是一个串行的过程，比较慢；

另外呢，RNN也是无法获取太长的历史信息；也就是说从当前时刻获取很久远的信息，RNN由于梯度消失的问题，无法获得太长的历史信息。这一点正是Transformer的优点。Transformer的归纳偏置是比较弱的，然后是通过一个全局的self attention，来计算两两位置之间的一个相关性，所以Transformer是可以上下去捕捉很长的历史关联性的。

以上是RNN的优缺点，最后讲一些RNN的应用场景



RNN的应用场景是非常多的，比如生成任务中，歌词生成、对联生成、像GPT一样写小说，这种生成任务。这种生成任务，如果用一幅图来表示的话

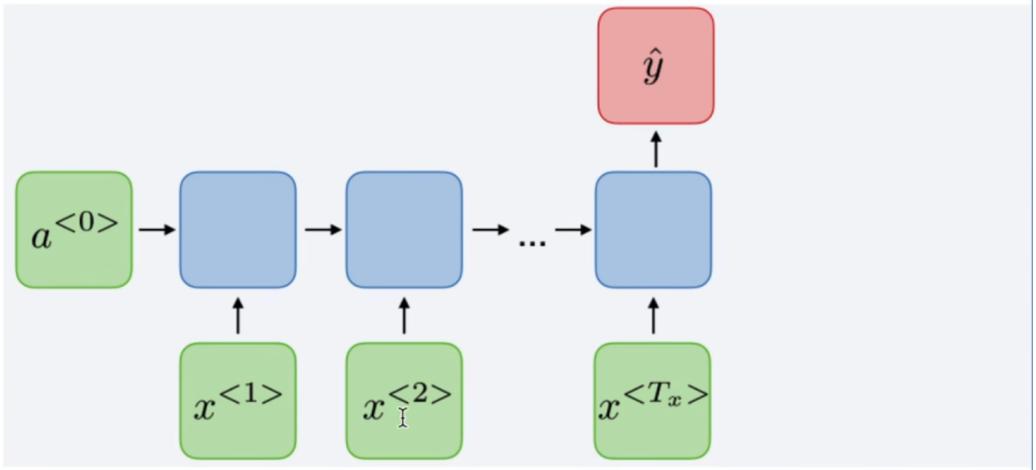


就是这样的，表示RNN在诗歌、语音、符号生成中的表示，可以认为在这类任务中，可以看成one to many的过程，也就是说只要给了一个输入，或者一个很短的输入，RNN就可以利用自己的递归机制不断的预测新的输出，就比如给出一两句话，RNN写出一段话或者一篇文章，这就是 one to many，这是RNN在生成任务上的应用。(241101)

241107

文本情感分类

另外 RNN也能做情感分类，情感分类比如说很古老的一个任务，对影评进行分类，判断一句话是正向情感还是负向情感，对于一个情感分类任务，我们可以看成many to one的任务



可以看到 输入是一段话或者说一篇文章，但是输出 只有一个，只需要对一段话预测一个类别就好了，这个就是many to one的任务，典型的应用场景就是去情感分类

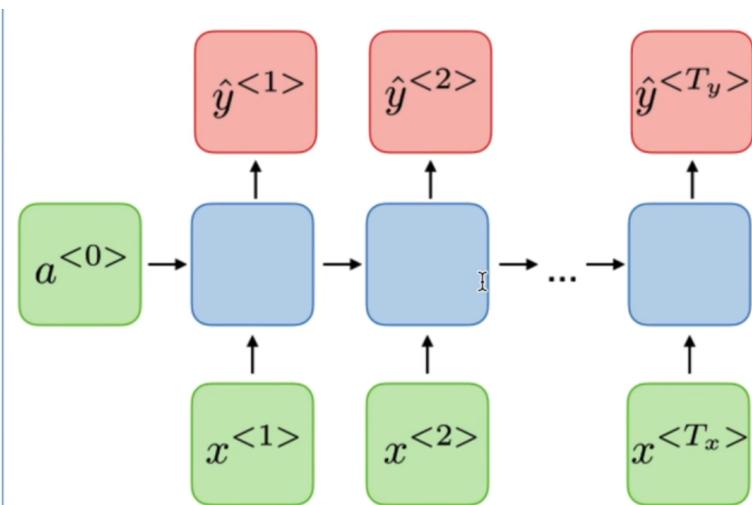


还有一些任务 many to many的

比如说词法识别 和 机器翻译； 所谓词法识别就是识别当前这个词是名词还是动词，当前这个单词多音字等等；

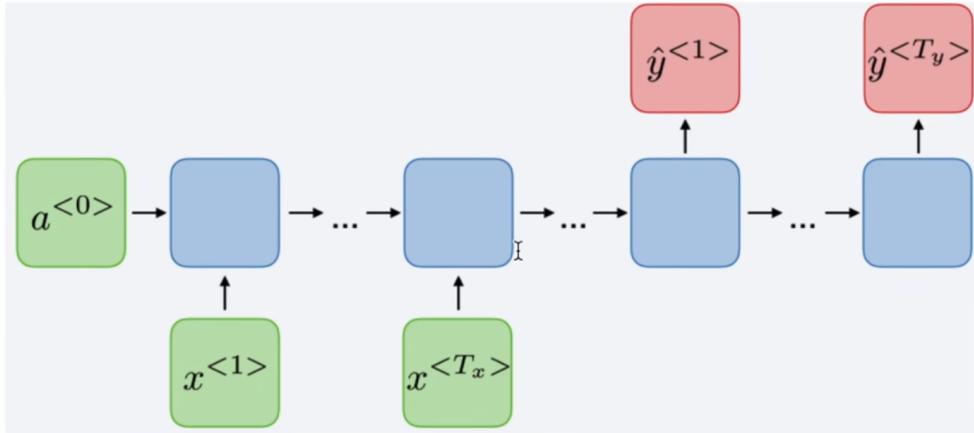
另外机器翻译，之前在Transformer中，是应用比较多的；但是这两种 many to many的模型结构还是有一些区别的；可以看到下面两幅图；

第一幅图 比如说词法识别

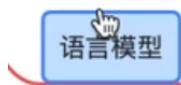


识别一句话中，每个字的拼音是什么，或者识别每个词的词性，这种就是many to many；这种属于直进直出的many to many

对于机器翻译这种，用的many to many 如图：



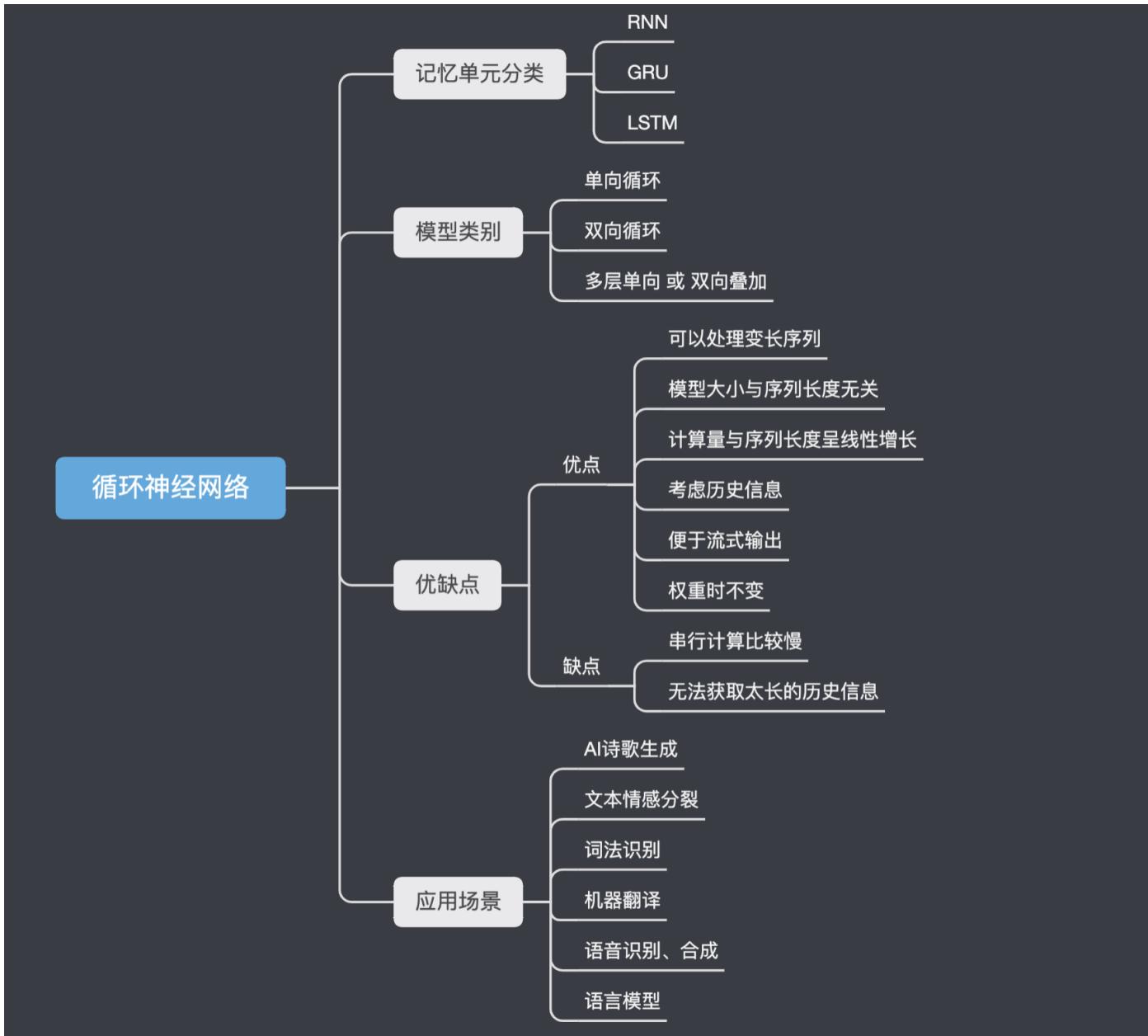
这种也就是sequence to sequence 结构；有编码器，有解码器，中间依靠注意力机制，来帮助解码器预测每一时刻的输出，这种也是many to many；比方说机器翻译或者语音合成等



语言模型 RNNLM; 总之就是

- one to one
- Many to one
- many to many

RNN主要是在NLP中 应用比较多；



以上是 RNN 基本的分类以及模型的类别；优缺点和应用场景；

接下来 pytorch 中，第一个 api `torch.nn.RNN` 的介绍

RNN

`CLASS torch.nn.RNN(*args, **kwargs) [SOURCE]`

Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

where h_t is the hidden state at time t , x_t is the input at time t , and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0 . If `nonlinearity` is `'relu'`, then ReLU is used instead of tanh.

这种 RNN 是很简单的 RNN，不像 LSTM、GRU 那种复杂一点

(讲图, 图描述)

可以用来构造一层或者多层简单的RNN结构；RNN还有另外一种结构：激活函数，可以用tanh激活函数或者ReLU激活函数，使得RNN有更强的非线性建模能力；

对于这种RNN计算公式是什么呢？

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

$$(W_{ih}x_t + b_{ih})$$

就是他每一时刻的输出，或者说每一时刻的状态；在简单RNN中，输出是等于状态的， h_t 也就是t时刻的输出，或者说t时刻RNN的状态等于tanh函数，就是非线性激活函数，里面分别是 $W_{ih} \times x_t$ 再加上 b_{ih} ，那么这里的 x_t ，就是当前时刻的输入，然后 W_{ih} ，就是在这个RNN中，它对输入的权重矩阵，就是会用这个矩阵来对权重做一个映射，然后整体上，这个东西可以看做linear层，有权重还有偏置， b_{ih} ，就是关于权重的一个偏置；

然后后面还有一项，后面一项就是跟历史状态有关的，跟 h_{t-1} 有关的，也就是说，我们需要将上一时刻的输出或者说上一时刻的隐含状态拿过来，然后对它进行一个映射，我们用 W_{hh} 的权重来进行相乘，来进行映射，然后再加上一个偏置，总体而言就是说每一时刻的输出或者说隐含状态不光跟当前时刻的输入 x_t 有关，同时也跟上一时刻的记忆单元 h_{t-1} 有关，并且都是线性组合的关系，最后通过一个非线性激活函数就能得到当前时刻的隐含状态；

Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

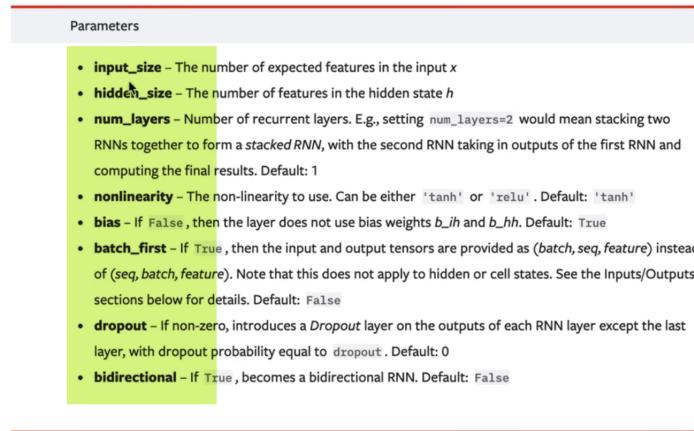
where h_t is the hidden state at time t, x_t is the input at time t, and $h_{(t-1)}$ is the hidden state of the previous layer at time t-1 or the initial hidden state at time 0. If nonlinearity is 'relu', then ReLU is used instead of tanh.

下面也解释了， h_t 是t时刻的隐含状态， x_t 是t时刻的输入，而 h_{t-1} 是t-1时刻的隐含状态，当然 h_0 表示初始时刻的隐含状态，pytorch中也提供了两种非线性激活函数：tanh和relu激活函数；默然用tanh激活函数



这是一个 class，我们在用RNN时候，首先要 实例化 这个class；实例化 class以后，得到 RNN的一个模型；然后我们再把 输入 喂入到 模型中，而不直接把 输入 喂入到模型中；一般所有模型的 class，都需要 先进行一个实例化；然后才能得到一个layer；

这里我们对RNN实例化呢，



我们需要提供这些参数

第一个参数是 `input_size`,也就是 我们输入特征的大小，也就是 x 的特征的维度

第二个参数是 `hidden_size`,`hidden_size`决定了 h_t 的大小，就是每一时刻的 h_t 就是一个向量，对于单一样本而言，每一时刻 h_t 就是一个向量，那么这个向量长度是多少呢？就是由 `hidden_size` 来决定

第三个参数 就是 `num_layers`，就是说 我们这个RNN，可以默认实例化的时候 只有一层，但是也可以改变 `num_layers`的值，变成多层，堆叠起来的结构，之前在介绍的时候也讲过，可以堆叠起来，单向的可以堆叠，双向的 也可 堆叠

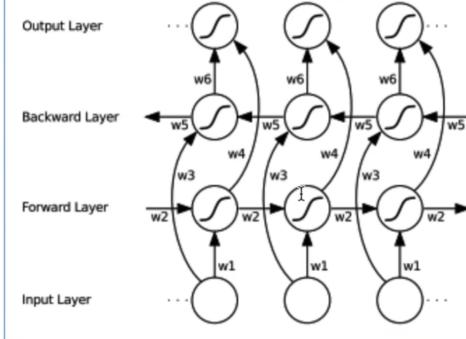
第四个参数 就是 非线性激活函数，这里默认是tanh函数，也可以改成 ReLu函数

第五个是bias 一般我们会加上 这两个bias

第六个参数是 `batch first`，这个需要注意一下，这个参数就决定了 我们的输入和输出的格式，如果我们设置 `batch first=true`的话，我们提供的输入张量 和 输出张量的 格式就是 `batch × sequence length×feature` 这样的格式，默认是`false`的，如果是 `false`的情况下，需要保证 输入的格式是 `sequence length`，也就是序列长度 在第一个维度，`batch size`在第二个维度，`feature size`在第三个维度

还有 `dropout`

最后一个参数`bidirectional`, 最后一个参数表示双向的意思, 也就是我们把这个参数设置为`true`的话, 就可以构建一个双向的RNN结构; 既然是双向RNN结构, 输出的特征大小就是 $2 \times \text{feature size}$, 就是2倍的`feature size`; 因为双向结构, 最后呢, 我们可以先看一下之前的图



这幅图就是双向的, 一旦我们把RNN设置成双向的话, 最终我们的输出是由`forward`输出和`backward`输出一起拼起来的, 所以这个输出状态是二倍的`hidden size`; 可以指定`concat`和`sum`, 一般用`concat`更多一点

也就是说如果设置`hidden size`是16的话, 那么`output layer`大小, 就是32, 如果是双向的话

以上是RNN实例化的参数讲解; 当我们实例化完以后, 我们就得到RNN的层; 然后我们就可以提供输入和初始的隐含状态, 来去递归的算出每一时刻的输入所对应的输出是什么; 所以当我们实例化完一个RNN以后呢, 我们就可以提供`input`和`h_0`, 来算出真正的输入序列,

Inputs: `input, h_0`

- input:** tensor of shape (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- h_0:** tensor of shape $(D * \text{num_layers}, N, H_{out})$ containing the initial hidden state for each element in the batch. Defaults to zeros if not provided.

where:

$N = \text{batch size}$
 $L = \text{sequence length}$
 $D = 2$ if `bidirectional=True` otherwise 1
 $H_{in} = \text{input_size}$
 $H_{out} = \text{hidden_size}$

通过RNN的模型计算以后, 所对应的输出是什么; 这里需要注意一下, 输入一般是三维的;

- input:** tensor of shape (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.

那如果我们设置的batch size first等于true的话，那对应的输入格式就是 batch size×sequence length×hidden size；反之 如果没有设置batch size等于true的话，我们提供的格式就是 sequence length×batch size×hidden size，这是输入

那 h_0 呢， h_0 的格式是 $d \times num_layers$, 然后是 N , H_{out} h_0 既然是初始状态，就是只有这一个时刻，所以我们这里，不需要考虑 sequence_length 这个维度，那这里也是三个维度，为什么呢？因为我们的 RNN 可以是多层也可以是双向，所以第一个维度 其实就是是否是双向 跟 多层 这两个因素 决定的；如果模型是一层，并且是单向的话，那么第一个维度就是 1，如果是有两层，并且是单向的话，那么就是 1×2 ，如果是双向 并且是两层的话，那就是 $2 \times 2 = 4$ ；所以这里的第一个维度 $d \times num_layers$ 由是否双向 以及 层数有关

- **h_0**: tensor of shape ($D * num_layers$, N , H_{out}) containing the initial hidden state for each element in the batch. Defaults to zeros if not provided.

where:

N = batch size
 L = sequence length
 D = 2 if bidirectional=True otherwise 1
 H_{in} = input_size
 H_{out} = hidden_size

第二个维度就是 batch size；每个样本都可以设置一个初始状态

第三个维度就是 hidden size 的大小，因为我们的初始状态就是一个向量；第三维就是向量的长度

以上是 api 的介绍；

我们可以现在 python 中 演示一下

这个RNN 是一个 class，如果我们要去实例化一个，首先 实例化一个 单向单层的RNN，这时我们可以直接 import torch.nn as nn；然后实例化 nn.RNN；传入 实例化参数；input_size=4, hidden_size 也可以随便写一个 hidden_size=3,num_layers 可以传入 1，

batch first 设置成 true，定义为single_rnn

```
import torch
import torch.nn as nn
# 1. 单向、单层RNN
single_rnn =
nn.RNN(input_size=4, hidden_size=3, num_layers=1, batch_first=True)
```

以上是单层单向RNN，接下来构建一个输入，输入的维度一般是 batch_size×sequence length×输入特征，输入特征就是RNN实例化时的 input size就是4, batch size设置成1, sequence length设置成2，特征维度是4，以上构建好了input序列

分别是：batch_size×sequence length×输入特征

```
input = torch.randn(1, 2, 4) # bs*sl*fs
```

我们把这个input作为 single_rnn的输入；也可以不传入ho,它默认以0向量填充

Inputs: input, h_0

- **input**: tensor of shape (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See [torch.nn.utils.rnn.pack_padded_sequence\(\)](#) or [torch.nn.utils.rnn.pack_sequence\(\)](#) for details.
- **h_0**: tensor of shape $(D * \text{num_layers}, N, H_{out})$ containing the initial hidden state for each element in the batch. Defaults to zeros if not provided.

where:

N = batch size
 L = sequence length
 D = 2 if `bidirectional=True` otherwise 1
 H_{in} = input_size
 H_{out} = hidden_size

同时我们也可以看看输出是什么

Outputs: output, h_n

- **output**: tensor of shape $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (h_t) from the last layer of the RNN, for each t . If a `PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n**: tensor of shape $(D * \text{num_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.

输出是两个值，一个是整个的，所有时刻的输出，另外一个输出的量就是最后一个时刻的状态；要定义变量接收输出

```
output, h_n = single_rnn(input)
```

这样整个输出就算完了，接下来，我们看一下output 和 h_n

```

>>> import torch
>>> # 单向、单层RNN
>>> import torch.nn as nn
>>> # 1. 单向、单层RNN
>>> single_rnn = nn.RNN(4, 3, 1, batch_first=True)
>>> input = torch.randn(1, 2, 4)
>>> input = torch.randn(1, 2, 4) # bs * sl * fs
>>> output, h_n = single_rnn(input)
>>> output
tensor([[-0.8492,  0.2763, -0.0716],
       [-0.6304, -0.4830, -0.4556]], grad_fn=<TransposeBackward1>)
>>> h_n
tensor([[-0.6304, -0.4830, -0.4556]], grad_fn=<StackBackward0>)

```

首先 input的形状 $1 \times 2 \times 4 = \text{batch size} \times \text{sequence length} \times \text{feature dim}$

single_rnn 的参数含义: 4,3,1=input_size,hidden_size,num_layers

output大小就是 $1 \times 2 \times 3$;

- 1表示 batch size, 输入batch size=1, 输出 batch size也是1, 没有改变
- 2是 sequence length, 序列长度, 我们喂入的输入长度是2, 所以输出的长度也是2
- 3, 第三个维度为什么是3呢? 因为我们设置的hidden size=3, 也就是说 每个输出的状态向量 长度是3

以上说明为什么output形状是 $1 \times 2 \times 3$

最后h_n就是 最后一个时刻的隐含状态; 在简单RNN中, 最后一个时刻的隐含状态等于最后时刻的输出的;

所以output最后一行的值 等于 h_n

```

>>> output
tensor([[-0.8492,  0.2763, -0.0716],
       [-0.6304, -0.4830, -0.4556]], grad_fn=<TransposeBackward1>)
>>> h_n
tensor([[-0.6304, -0.4830, -0.4556]], grad_fn=<StackBackward0>)
>>> h_n

```

以上 单向 单层RNN

接下来 双向、单层RNN

```

single_rnn =
nn.RNN(input_size=4, hidden_size=3, num_layers=1, batch_first=True)

```

input size不变; hidden size不变; num_layers不变, batch first也不变, 但是需要新增一个参数, 叫做

Parameters

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights b_{ih} and b_{hh} . Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as $(batch, seq, feature)$ instead of $(seq, batch, feature)$. Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

bidirectional, 这个参数默认是false, 我们要把它置成true

然后命名为 `bidirectional_rnn`

```
bidirectional_rnn =  
nn.RNN(input_size=4, hidden_size=3, num_layers=1, batch_first=True, bidirectional=True)
```

以上是实例化的双向RNN, 输入特征大小是4, 输出 or 隐含层大小是3, 只有一层, `batch first=True`, 并且还是双向的

同样把上面的输入送入双向RNN中, 以`input`作为输入`bidirectional_rnn(input)`, 因为无论双向、单向, 输出都是一样的, 都是`output`和`h_n`, 表示区别加前缀bi

```
bi_output, bi_h_n = bidirectional_rnn(input)
```

接下来首先打印 `output`的形状

```
bi_output.shape
```

```
>>> bi_output.shape  
torch.Size([1, 2, 6])
```

还有`h_n`的形状

```
bi_h_n.shape
```

```

>>> import torch
>>> # 单向、单层RNN
>>> import torch.nn as nn
>>> # 1. 单向、单层RNN
>>> single_rnn = nn.RNN(4, 3, 1, batch_first=True)
>>> input = torch.randn(1, 2, 4)
>>> input = torch.randn(1, 2, 4) # bs * sl * fs
>>> output, h_n = single_rnn(input)
>>> output
tensor([[-0.8492,  0.2763, -0.0716],
       [-0.6304, -0.4830, -0.4556]], grad_fn=<TransposeBackward1>)
>>> h_n
tensor([[-0.6304, -0.4830, -0.4556]], grad_fn=<StackBackward0>)
>>> # 2. 双向、单层RNN
>>> bidirectional_rnn = nn.RNN(4, 3, 1, batch_first=True, bidirectional=True)
>>> bi_output, bi_h_n = bidirectional_rnn(input)
>>> bi_output.shape
torch.Size([1, 2, 6])
>>> bi_h_n.shape
torch.Size([2, 1, 3])
>>

```

对比，把单向单层RNN的output的形状，`h_n`的形状，都打印出来

```

>>> bi_output.shape
torch.Size([1, 2, 6])
>>> bi_h_n.shape
torch.Size([2, 1, 3])
>>> output.shape
torch.Size([1, 2, 3])
>>> h_n.shape
torch.Size([1, 1, 3])
>>

```

可以看到，首先从输出上来讲；单向的输出大小是 $1 \times 2 \times 3$ 的；双向的话变成了 $1 \times 2 \times 6$ （一个batch size；2个sequence length；6个特征维度）；这是为什么呢？这是因为在双向RNN，中最后是把forward layer和backward layer两个输出拼起来；所以特征大小变成了两倍的hidden size；

最后一个时刻的状态也是不一样的，因为在双向RNN中，它的维度是 $2 \times 1 \times 3$ （前向的输出是个 1×3 ，后向的输出也是一个 1×3 ）；在单向中，维度是 $1 \times 1 \times 3$ ；为什么呢？因为双向中，其实是有两个层的最后一个时刻状态，有一个forward layer和一个backward layer；这两个状态在第一个维度上拼起来了；但是在单向中，只有一层的最后一个状态；

以上是RNN API的介绍；全部代码汇总：

```

>>> import torch
>>> # 单向、单层RNN
>>> import torch.nn as nn
>>> # 1. 单向、单层RNN
>>> single_rnn = nn.RNN(4, 3, 1, batch_first=True)
>>> input = torch.randn(1, 2, 4)
>>> input = torch.randn(1, 2, 4) # bs * sl * fs
>>> output, h_n = single_rnn(input)
>>> output
tensor([[-0.8492,  0.2763, -0.0716],
       [-0.6304, -0.4830, -0.4556]], grad_fn=<TransposeBackward1>)
>>> h_n
tensor([[-0.6304, -0.4830, -0.4556]], grad_fn=<StackBackward0>)
>>> # 2. 双向、单层RNN
>>> bidirectional_rnn = nn.RNN(4, 3, 1, batch_first=True, bidirectional=True)
>>> bi_output, bi_h_n = bidirectional_rnn(input)
>>> bi_output.shape
torch.Size([1, 2, 6])
>>> bi_h_n.shape
torch.Size([2, 1, 3])
>>

```

```
>>> bi_output.shape  
torch.Size([1, 2, 6])  
>>> bi_h_n.shape  
torch.Size([2, 1, 3])  
>>> output.shape  
.torch.Size([1, 2, 3])  
>>> h_n.shape  
torch.Size([1, 1, 3])
```

单向RNN&双向RNN 从矩阵运算的角度实现；

topic: RNN模型的原理、API及源码实现

都没设置 多层 num layers都定义的1层

首先引入库，可以使用常见的pytorch函数

```
import torch  
import torch.nn as
```

然后定义一些常量，比如batch size；序列长度，

```
bs, T = 2, 3 #批大小 和 序列长度
```

还需要定义 input size和hidden size， 分别表示输入特征大小 和 隐含层 特征大小

```
input_size, hidden_size=2, 3 #输入特征大小, 隐含层特征大小
```

有了这些量以后，我们可以去生成一个 input， 我们还是考虑batch first等于true的情况；第一个位置写batch size； 第二个位置写序列长度， 第三个位置写feature dim， 也就是input size

```
input = torch.randn(bs, T, input_size) # 随机初始化一个输入特征序列
```

我们还需要初始化一个初始的隐含状态 h_0， 初始的隐含状态一般是一个向量， 但是这里我们考虑了batch size； 所以它应该是 batch size个这样的状态， 我们也可以先写成0
h_prev=torch.zeros(bs,hidden_size) # 每一个状态向量大小是 hidden size

```
h_prev = torch.zeros(bs, hidden_size) #初始隐含状态
```

也就是在第一个时刻的时候， 我们需要一个初始的隐含状态来， 来作为第0时刻的初始状态

RNN

CLASS `torch.nn.RNN(*args, **kwargs)` [SOURCE]

Applies a multi-layer Elman RNN with `tanh` or `ReLU` non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

where h_t is the hidden state at time t , x_t is the input at time t , and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0. If `nonlinearity` is `'relu'`, then `ReLU` is used instead of `tanh`.

定义好这些以后，首先第一步调用pytorch RNN的API

还是用nn.RNN()的api，需要传入input_size,hidden size还是有batch first=True，这样我们得到一个rnn

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

我们需要把 input 传入RNN中，以及初始状态也传入RNN中，但是需要注意的是，api中初始状态是三维的

Inputs: input, h_0

- **input**: tensor of shape (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D \times \text{num_layers}, N, H_{out})$ containing the initial hidden state for each element in the batch. Defaults to zeros if not provided.

where:

$$\begin{aligned}N &= \text{batch size} \\L &= \text{sequence length} \\D &= 2 \text{ if } \text{bidirectional}=\text{True} \text{ otherwise } 1 \\H_{in} &= \text{input_size} \\H_{out} &= \text{hidden_size}\end{aligned}$$

我们刚刚初始化的是后面两维；第三维我们没有初始化，因为这里是单向的并且只有一层的，所以我们对它扩一维就好了，扩0维,得到rnn output和h_finall，最后一个时刻的状态，或者叫state_finall

```
rnn_output,state_finall = rnn(input,h_prev.unsqueeze(0))
```

这个是调用pytorch 官方的api，我们运行打印，看结果

```
In [2]: import torch
import torch.nn as nn

bs, T = 2, 3 # 批大小, 输入序列长度
input_size, hidden_size = 2, 3 # 输入特征大小, 隐含层特征大小
input = torch.randn(bs, T, input_size) # 随机初始化一个输入特征序列
h_prev = torch.zeros(bs, hidden_size) # 初始隐含状态

# step1 调用PyTorch RNN API
rnn = nn.RNN(input_size, hidden_size, batch_first=True)
rnn_output, state_final = rnn(input, h_prev.unsqueeze(0))
print(rnn_output)
print(state_final)
```

```
tensor([[[[-0.5299, -0.4027, -0.3366],
          [-0.4068, -0.5572, -0.2334],
          [-0.0404, -0.1345, -0.8417]],

         [[ 0.0229, -0.3688,  0.4001],
          [ 0.4353, -0.1760,  0.4028],
          [-0.9544, -0.7666,  0.1180]]], grad_fn=<TransposeBackward1>)
tensor([[[[-0.0404, -0.1345, -0.8417],
          [-0.9544, -0.7666,  0.1180]]], grad_fn=<StackBackward>)
```

以上实现了 单向的RNN

接下来 第二步 手写RNN forward 函数； RNN 在pytorch中的核心函数，从1.0版本以后，放到了C语言中， C语言很复杂，还有cuda语言；今天演示主要通过python语言，演示原理，知道原理就好了

定义RNN forward函数，实现RNN计算原理 def rnn_forward():，对于这个函数首先要传入参数，

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

根据公式，我们要想算出 h_t 的话，需要有 x ， x 就是输入，所以第一个参数，需要写的是 input，那还有什么呢，就是输入，输入需要一个投影矩阵，就是 W_{ih} ，我们需要一个 weight，同时还需要偏置项 b_{ih} ，

另外还有上一时刻的隐含状态也有一个投影 W_{hh} ，还有 b_{hh} 这两个参数，公式中还有 h_{t-1} 我们可以写成 h_prev ，就是前一时刻的状态，有了这些以后，就能算出RNN的输出

```
def rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev):
```

这里 input 我们默认 三维的结构，所以我们可以先把input的形状 拆解出来，形状应该是 batch size×sequence length×input size 也就是嵌入维度，就是input.shape，可以列出来

```
bs, T, input_size = input.shape
```

我们还可以知道 hidden size，也就是 h_dim ，它也就是 weight_ih，可以根据它的权重所得到，也就是weight_ih.shape，那到底是shape[0]还是 shape[1]呢？我们看

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

weight ih跟 xt 是左乘的关系，所以weight的第二个维度跟x是相同的，所以第一个维度就是隐含单元的维度，所以写成.shape[0]，得到hidden dim

```
h_dim = weight_ih.shape[0]
```

以上是得到了一些维度，接下来，可以写出 h out，首先 初始化一个输出，输出大小是 batch size×T×h dim，初始化一个输出矩阵或者状态矩阵

```
h_out = torch.zeros(bs, T, h_dim) # 初始化一个输出（状态）矩阵
```

这里 bs 跟输入是一样的，序列长度 或者叫 时间长度 也是跟 输入一样的，维度需要改成 hidden size 这个维度；接下来根据这 6 个参数，算出 h out

```
# step2 手写一个rnn_forward函数，实现RNN的计算原理
def rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim) # 初始化一个输出(状态)矩阵
```

既然叫RNN，那就是一个递归的计算，所我们需要根据x1计算h1，根据x2计算h2等等，也就是我们需要一个for循环 for t in range(T): 因为之前也说过，RNN的计算复杂度 跟序列长度 呈线性关系，所以我们需要对序列长度进行一个遍历就好了，

```
for t in range(T):
```

首先得到当前时刻的输入向量，input，因为input是三维，第一个维度是 batch size，我们全都拿出来，第二个维度是时间，就拿当前 t 时刻的输入向量，第三维是特征维度，也是全部拿出来

```
x = input[:, t, :] # 获取当前时刻输入特征，bs*input_size
```

以上是第一步：获取当前时刻的输入特征得到x

接下来 根据公式，让w跟x进行相乘，这里weight，一般默认传入 是二维的；而x的大小呢，默认是 batch size×input size

weight ih的形状是 h dim×input size；所以为了让它们进行batch维度无关的乘法运算的话，我们首先对weight ih进行一个扩充，我们把weight变成一个 batch的形式，现在 weight ih是hidden size×input size的大小，我们首先对它增加一维，batch 维度，然后我们对它进行复制，复制成跟input一样的大小，这样它的大小就变成了 batch size×h dim×input size

```
w_ih_batch = weight_ih.unsqueeze(0).tile(bs,1,1)
# bs*h_dim*input_size
```

这是 w_{ih} , 我们把它变成 batch 的形状, 同样对于 $weight_{hh}$, 也是一样的, 我们也可以转换一下, 首先对它增加一个 batch 维度, 然后把它的 batch 维度扩充成 batch size 维度大小

```
w_hh_batch = weight_hh.unsqueeze(0).title(bs,1,1)
# bs * h_dim * h_dim
```

但是这里 w_{hh} 大小就是 batch size \times h dim \times h dim, 因为它跟 hidden state 相连的, 所以它就是一个方阵

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$$

接下来, 开始计算 $w_{ih} \times x_t$ 、 $w_{hh} \times h_{t-1}$

首先计算 x , 就是计算 W_{ih} 乘以 x 这个量 w_times_x 这个量, 我们可以调用 `torch.bmm` 这个函数, batch matrix multiplication, 就是含有批大小的矩阵相乘, 与批无关的计算矩阵相乘, 第一个就是 w_{ih} batch, 第二个位置传入 x , 当前这个 x 是 batch size \times input_size 的, 为了跟 w_{ih} batch 相乘, 我们需要将它扩充一维, 扩充成 batch size \times input size \times 1 的, 所以这里我们需要对它扩充一下, 调用一下 `unsqueeze`

```
x = input[:,t,:].unsqueeze(2)
```

就在本来是二维的, 现在在第三个维度上进行扩充, 变成 batch size \times input size \times 1, 此时跟 x 相乘, 得到 batch size \times h dim \times 1, 最后 1 的维度我们去掉, 调用 `unsqueeze` 函数, 得到的结果 batch size \times h dim, 得到 $w times x$ 的结果, 偏置最后再加

```
x = input[:,t,:].unsqueeze(2) # 获取当前时刻的输入特征 bs*input_size*1
w_ih_batch = weight_ih.unsqueeze(0).tile(bs,1,1) #bs*h_dim*input_size
w_hh_batch = weight_hh.unsqueeze(0).tile(bs,1,1) #bs*h_dim*h_dim

w_times_x = torch.bmm(w_ih_batch,x).squeeze(-1) # bs*h_dm
```

第二项 w_times_h , 也就是 Whh 这个矩阵 跟上一时刻的状态, 进行相乘的一个结果, 同样调用 `torch.bmm` 函数, 带有批大小的矩阵相乘, 跟上一时刻的隐含状态进行相乘; 同样对 h_{prev} 进行扩充, $h_prev.unsqueeze(2)$, 就是把它扩充三维, 因为 h_{prev} 本来是, batch size \times hidden size, 现在变成 batch size \times hidden size \times 1, 乘出来以后是 batch size \times hidden size \times 1, 最后再把 1 去掉, 挤掉;

这里乘的 权重 是方阵, 不改变大小, 所以还是 h_{prev} 的形状

```
w_times_h = torch.bmm(w_hh_batch, h_prev.unsqueeze(2)).squeeze(-1)
```

这里调用 squeeze函数，把最后的1去掉 最后变成了 batch size× h dim

这是这两个量，最后我们把这些东西全部加起来，跟bias加起来，然后过两个tanh函数

```
import torch
import torch.nn as nn

bs, T = 2, 3 # 批大小, 输入序列长度
input_size, hidden_size = 2, 3 # 输入特征大小, 隐含层特征大小
input = torch.randn(bs, T, input_size) # 随机初始化一个输入特征序列
h_prev = torch.zeros(bs, hidden_size) # 初始隐含状态

# step1 调用PyTorch RNN API
rnn = nn.RNN(input_size, hidden_size, batch_first=True)
rnn_output, state_final = rnn(input, h_prev.unsqueeze(0))
# print(rnn_output)
# print(state_final)

# step2 手写一个rnn_forward函数, 实现RNN的计算原理
def rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim) # 初始化一个输出(状态)矩阵

    for t in range(T):
        x = input[:, t, :].unsqueeze(2) # 获取当前时刻输入特征, bs*input_size*1
        w_ih_batch = weight_ih.unsqueeze(0).tile(bs, 1, 1) # bs*h_dim*input_size
        w_hh_batch = weight_hh.unsqueeze(0).tile(bs, 1, 1) # bs*h_dim*h_dim

        w_times_x = torch.bmm(w_ih_batch, x).squeeze(-1) # bs*h_dim
        w_times_h = torch.bmm(w_hh_batch, h_prev.unsqueeze(2)).squeeze(-1) # bs*h_dim
```

首先是 w_times_x这个量 然后加上 bias ih, 最后加上 w times h, 上一时刻隐含状态相关的, 最后是bias hh, 然后过一个 tanh激活函数, 最终得到当前时刻的这一状态

```
torch.tanh(w_times_x + bias_ih + w_times_h + bias_hh)
```

定义为h_prev，因为我们现在进行的是递归的运算

```
h_prev = torch.tanh(w_times_x + bias_ih + w_times_h + bias_hh)
```

现在计算了t时刻的输出，现在我们把t时刻的输出，放入到 h out中，

怎么放，只要放到时间长度这一维，t行就好了

```
h_out[:, t, :] = h_prev
```

以上完成了递归的运算，最后返回跟 pytorch官方api一样，首先返回h_out,然后返回最后一个时刻的隐含状态，其实也就是h_prev,但是这里的h_prev是二维的，官方api是三维的，那么我们要扩一维，扩一维的原因是因为我们这里是单向的，单层的，所以我们在第0维扩充一个1就好了

```
return h_out, h_prev.unsqueeze(0)
```

以上是所有全手写的RNN forward函数，其实就是单向的RNN

补充 torch.tile函数：沿指定维度重复张量函数；例子展示用法

```
import torch

# 创建一个张量
weight_hh = torch.tensor([[1, 2], [3, 4]])

# 假设批量大小为3
bs = 3

# 使用 unsqueeze 在第0维度增加一个维度, 然后使用 tile 沿第0维度重复 bs 次
w_hh_batch = weight_hh.unsqueeze(0).tile(bs, 1, 1)

print("原始张量:")
print(weight_hh)
print("增加维度并重复后的张量:")
print(w_hh_batch)
```

在这个示例中：

1. weight_hh 是一个形状为 [2, 2] 的张量。
2. weight_hh.unsqueeze(0) 在第0维度增加一个维度，使其形状变为 [1, 2, 2]。
3. tile(bs, 1, 1) 沿第0维度重复 bs 次（这里 bs 为3），使其形状变为 [3, 2, 2]。

输出结果：

```
原始张量:
tensor([[1, 2],
        [3, 4]])
增加维度并重复后的张量:
tensor([[[1, 2],
         [3, 4]],
        [[1, 2],
         [3, 4]]])
```

这样，w_hh_batch就是一个形状为 [3, 2, 2] 的张量，其中每个批次都包含原始的weight_hh张量

接下来验证一下，怎么验证呢？就是把之前实例化的RNN网络，参数拿出来，填充到我们定义的网络中；然后算出来的结果如果是跟官方API结果一致的话，就表明我们的函数是正确的；首先我们拿出RNN的参数。

那RNN有哪些参数呢？之前我们讲过nn.Module这个类，在pytorch中所有的层，都是继承自nn.Module这个类，之前讲过nn.Module的一些函数，现在我们可以调用这个函数，叫做name.parameters这个函数，来找一下这个RNN中有哪些参数，name.parameters是一个生成器，可以用一个循环来得到 for p,n in ;p就是参数，n就是name； in rnn.named_parameters(); 就能看到这个rnn有哪些参数

```
for p,n in rnn.named_parameters():
```

可以看到RNN有哪些参数 以及 它的名称；或者改成k, v，并打印k, v

```
# 验证一下rnn_forward的正确性
for k,v in rnn.named_parameters():
    print(k, v)

weight_ih_10 Parameter containing:
tensor([[ 0.1589,  0.3338],
       [ 0.1042, -0.5644],
       [ 0.4643,  0.0864]], requires_grad=True)
weight_hh_10 Parameter containing:
tensor([[[-0.3713,  0.4136,  0.2405],
       [-0.3384,  0.1308,  0.0340],
       [-0.3676,  0.4307, -0.0852]], requires_grad=True])
bias_ih_10 Parameter containing:
tensor([-0.2768, -0.1377,  0.1113], requires_grad=True)
bias_hh_10 Parameter containing:
tensor([ 0.3048,  0.2944, -0.4984], requires_grad=True)
```

可以看到关于这个 RNN，所有的参数、名称，以及具体地张量的数值都打印出来了，一共有四个参数，分别是weight ih L0，

- 这个weight ih其实就是公式里的 wih，这个L0是什么意思呢？就是我们只有一层，层数是从0开始的，所以是从L0（那边是小写的l）
- 第二个参数就是weight hh L0,就是表示当前这一层 w hh这个参数，
- 另外两个就是偏置了，分别是bias ih和bias hh

需要注意的是，前面两个权重张量 是二维张量；后面两个偏置是一维的向量

现在我们把这些参数 代入到 我们自己写的RNN forward函数中

首先 我们先复制一下自己写的函数签名

```
rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev):
```

*input*还是我们写的*input*

*weight ih*可以改成*rnn.*,直接用*rnn.*参数名称, 就可以去访问这个参数
, rnn.weight_ih_l0

然后*weight hh*也是一样, 用*rnn.*来进行访问: *rnn.weight_hh_l0*

*bias*也是一样的 对应的是 *rnn.bias_ih_l0*

同样*hh bias*也是一样的 *rnn.bias_hh_l0*

然后这里的*h_prev*, 就是我们定义好的, 就直接用*h_prev*就好了

```
rnn_forward(input, rnn.weight_ih_l0, rnn.weight_hh_l0, rnn.bias_ih_l0, rnn.bias_hh_l0, h_prev)
```

前面写的是rnn output和state final

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

这里我们加前缀 custom, 表示自己写的

```
custom_rnn_output, custom_state_final =  
rnn_forward(input, rnn.weight_ih_l0, rnn.weight_hh_l0, rnn.bias_ih_l0, rnn.bias_hh_l0, h_prev)
```

这样我们就调用自己的写的RNN forward函数; 然后我们对比pytorch api的结果 和 自己写的结果

```
PyTorch API output:  
tensor([[-0.0851,  0.0877, -0.2537],  
      [-0.3842, -0.2042,  0.1568],  
      [-0.0530, -0.0123, -0.3959]],  
  
     [[-0.1809,  0.2473,  0.2277],  
      [-0.1150,  0.0164, -0.0249],  
      [-0.1922, -0.0139, -0.0728]]], grad_fn=<TransposeBackward1>  
tensor([[[-0.0530, -0.0123, -0.3959],  
      [-0.1922, -0.0139, -0.0728]]], grad_fn=<StackBackward>)  
  
rnn_forward function output:  
tensor([[-0.0851,  0.0877, -0.2537],  
      [-0.3842, -0.2042,  0.1568],  
      [-0.0530, -0.0123, -0.3959]],  
  
     [[-0.1809,  0.2473,  0.2277],  
      [-0.1150,  0.0164, -0.0249],  
      [-0.1922, -0.0139, -0.0728]]], grad_fn=<CopySlices>  
tensor([[[-0.0530, -0.0123, -0.3959],  
      [-0.1922, -0.0139, -0.0728]]], grad_fn=<UnsqueezeBackward0>)
```

第一个张量整体RNN 预测的输出；基本上是一样的；因为现在维度比较小，肉眼就能看一致；第二个张量是最后一个时刻的输出，官方的结果和自定义的结果一样

Step2 全部代码：

```
# step2 手写一个rnn_forward函数，实现单向RNN的计算原理
def rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim) # 初始化一个输出(状态)矩阵

    for t in range(T):
        x = input[:, t, :].unsqueeze(2) # 获取当前时刻输入特征, bs*input_size*1
        w_ih_batch = weight_ih.unsqueeze(0).tile(bs, 1, 1) # bs*h_dim*input_size
        w_hh_batch = weight_hh.unsqueeze(0).tile(bs, 1, 1) # bs*h_dim*h_dim

        w_times_x = torch.bmm(w_ih_batch, x).squeeze(-1) # bs*h_dim
        w_times_h = torch.bmm(w_hh_batch, h_prev.unsqueeze(2)).squeeze(-1) # bs*h_dim
        h_prev = torch.tanh(w_times_x+bias_ih+w_times_h+bias_hh)

        h_out[:, t, :] = h_prev

    return h_out, h_prev.unsqueeze(0)
# 验证一下rnn_forward的正确性
# for k,v in rnn.named_parameters():
#     print(k, v)
custom_rnn_output, custom_state_final = rnn_forward(input, rnn.weight_ih_10, rnn.weight_hh_10, \
                                                    rnn.bias_ih_10, rnn.bias_hh_10, h_prev)
print("\n rnn_forward function output:")
print(custom_rnn_output)
print(custom_state_final)
```

接下来验证双向RNN

$$h_t = \tanh(x_t)$$

双向的话我们可以来调用一下 单向里面的函数；双向需要注意所有的参数都double了，所有的weight和bias 都有两个

```
# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
```

双向 我们要考虑两倍的 forward函数和backward层； weight有forward层和backward层， bias也有forward层和backward层； h prev也是有两份的；第一份 就是 forward layer，还有就是 backward，我们复制一下然后改名，按照官方的名称，改成reverse就好了，也就是说这时候所有的参数都是两份的，forward一份，backward一份； RNN是比较简单的，如果是LSTM GRU 要更久

函数签名写成：

```
# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
                               weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
```

接下来，还是一样的，得到一些基本的信息，首先上面复制下来，

```
# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim) # 初始化一个输出(状态)矩阵
```

第一步 batch size, 时间 和 input size; 然后我们得到 hidden size h_dim

关于h_out, 这里batch size不变, T不变, 但是h dim要变成两倍, 因为是双向的结构;

```
h_out = torch.zeros(bs, T, h_dim*2)
# 初始化输出状态矩阵, 注意双向是两倍的特征大小
```

该定义的定义好了, 接下来 调用RNN forward函数

调用两次RNN forward函数

第一步一模一样

```
# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim*2) # 初始化一个输出(状态)矩阵, 注意双向是两倍的特征大小
```

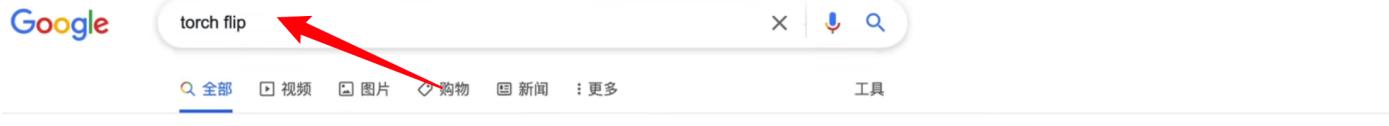
传入这些; 这是forward层的调用, 取名为 forward_output, 我们这里只取第一个返回值, 所以加个[0]

```
# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim*2) # 初始化一个输出(状态)矩阵, 注意双向是两倍的特征大小

    forward_output = rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev)[0]
```

得到 forward layer

下面 backward layer, 我们要变换一下, 除了所有的参数都用reverse版本的, 对input也要reverse一下, 就是因为如果是反向的话, 我们要保证第一个位置上, input是最后一个元素; 对input我们需要在长度这一维进行翻转,



找到约 9,630,000 条结果 (用时 0.36 秒)
<https://pytorch.org/docs/stable/generated/torch.flip.html> 翻译此页
torch.flip — PyTorch 1.10.1 documentation

Reverse the order of a n-D tensor along given axis in dims. ... `torch.flip` makes a copy of input's data. This is different from NumPy's `np.flip`, which returns a ...

torch.Tensor.flip — PyTorch 1.10.1 documentation
`torch.Tensor.flip`. `Tensor.flip(dims)` → `Tensor`. See `torch.flip()` · Next · Previous. © Copyright 2019, Torch Contributors. Built with Sphinx using a theme ...

我们可以调用 `torch.flip` api, 这个api, 对张量进行一个翻转,

The screenshot shows the PyTorch `torch.flip` API documentation. A red box highlights the "Parameters" section, which contains two bullet points: `input (Tensor) – the input tensor.` and `dims (a list or tuple) – axis to flip on`. Below this, there is an "Example:" section with the following code:

```
>>> x = torch.arange(8).view(2, 2, 2)
>>> x
tensor([[[ 0,  1],
          [ 2,  3]],

         [[ 4,  5],
          [ 6,  7]]])
>>> torch.flip(x, [0, 1])
tensor([[[ 6,  7],
          [ 4,  5]],

         [[ 2,  3],
          [ 0,  1]]])
```

有两个参数，一个是`input`，一个是`dim`，也就是说传入的是哪个 `dim`，就会对哪个 `dim` 进行翻转；就是完全相反的顺序

还是先拷贝所有的参数，然后调用 `rnn_forward` 函数

```
# step3 手写一个bidirectional_rnn_forward函数, 实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim*2) # 初始化一个输出(状态)矩阵, 注意双向是两倍的特征大小

    forward_output = rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev)[0] # forward layer
    rnn_forward(weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse)
```

然后第一个参数 `input`，进行一个翻转，调用 `torch.flip`, `flip`的第一个参数是 `input`，第二个参数是维度，维度官方api中规定

- **input** (*Tensor*) – the input tensor.
- **dims** (a *list or tuple*) - axis to flip on

要么是列表 要么是元组

我们这里的input是三维，我们要翻转的是中间这一维，T这维

```
print(custom_state_final)

# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim*2) # 初始化一个输出(状态)矩阵，注意双向是两倍的特征大小

    forward_output = rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev)[0] # forward layer
    rnn_forward(torch.flip(input, [1]), weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_rever
```

，所以我们传入一个列表，1这一维度，表示中间这一维度，进行翻转

```
rnn_forward(torch.flip(input,
[1]), weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse)
```

同样对它的调用也只取 output，定义为 backward output

```
backward_output = rnn_forward(torch.flip(input,
[1]), weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse)[0] # backward layer
```

以上我们得到了 forward output和backward output，我们为什么只保留了 h_out，没有保留 h_prev 呢？因为在RNN中，h prev可以从 h out中得到，所以为了方便我们只取了 h out

```
n_out = torch.zeros(ds, T, n_out) # 初始化一个输出(状态)矩阵

for t in range(T):
    x = input[:, t, :].unsqueeze(2) # 获取当前时刻输入特征, bs*input_size*1
    w_ih_batch = weight_ih.unsqueeze(0).tile(ds, 1, 1) # bs*h_dim*input_size
    w_hh_batch = weight_hh.unsqueeze(0).tile(ds, 1, 1) # bs*h_dim*h_dim

    w_times_x = torch.bmm(w_ih_batch, x).squeeze(-1) # bs*h_dim
    w_times_h = torch.bmm(w_hh_batch, h_prev.unsqueeze(2)).squeeze(-1) # bs*h_dim
    h_prev = torch.tanh(w_times_x+bias_ih+w_times_h+bias_hh)

    h_out[:, t, :] = h_prev

return h_out, h_prev.unsqueeze(0)
# 验证一下rnn_forward的正确性
# for k,v in rnn.named_parameters():
#     print(k, v)
custom_rnn_output, custom_state_final = rnn_forward(input, rnn.weight_ih_10, rnn.weight_hh_10, \
```

接下来，我们把 forward output 和 backward output 填充到 h_out 中，首先 h_out 是三维的，并且最后一维由 forward 和 backward 填充起来的，所以我们填充时，索引的写法：

从0到 h_dim

```
h_out[:, :, :h_dim] = forward_output
```

从 h_dim: 到最后

前向的输出，填充到前一半中，后一半的维度，我们用 backward output 填充就好了

```
h_out[:, :, h_dim:] = backward_output
```

这样我们就把前向输出和后向输出拼起来了，然后就可以返回了

同样按照官方api，我们要返回两个数，第一个数是 h_out，第二个数就是 state final

Outputs: output, h_n

- **output**: tensor of shape $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (h_t) from the last layer of the RNN, for each t . If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n**: tensor of shape $(D * num_layers, N, H_{out})$ containing the final hidden state for each element in the batch.

Sate final 我们要注意它的维度是 $D * num_layers \times N \times H_{out}$

前面表示 双向 和 层数的乘积；中间是 batch size；后面是 H_{out}

我们怎么写呢？首先肯定是要取出 h_out 的最后一个时刻，因为时刻是在中间那个维度，所用 -1 索引；

```
return h_out, h_out[:, -1, :].reshape()
```

我们先取出最后一个时刻，最后一个时刻的状态向量，现在是 batch size \times 2倍的 h_dim，我们先 reshape一下，把 2 单独拎出来，然后 reshape

Batch size 不变，2 单独拎出来，h dim 就写成 h_dim；首先把二维张量变成三维张量

```
return h_out, h_out[:, -1, :].reshape((bs, 2, h_dim))
```

然后我们再把2提到前面，根据官方api，2在前面：

- a packed sequence.
- **h_n**: tensor of shape $(D : \text{num_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.

batch size在中间，所以我们可以把2提到前面，调用一下转置函数，就是把第0维度和第一维度交换一下：

```
return h_out, h_out[:, -1, :].reshape((bs, 2, h_dim)).transpose(0, 1)
```

以上写完了双向的RNN函数

复述一遍：

```
# step3 手写一个bidirectional_rnn_forward函数，实现双向RNN的计算原理
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
                               weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):
    bs, T, input_size = input.shape
    h_dim = weight_ih.shape[0]
    h_out = torch.zeros(bs, T, h_dim*2) # 初始化一个输出(状态)矩阵，注意双向是两倍的特征大小
    forward_output = rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev)[0] # forward layer
    backward_output = rnn_forward(torch.flip(input, [1]), weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse)[0] # backward layer
    h_out[:, :, :h_dim] = forward_output
    h_out[:, :, h_dim:] = backward_output
    return h_out, h_out[:, -1, :].reshape((bs, 2, h_dim)).transpose(0, 1)
```

首先把input传入到forward layer中，然后再把input按照时间的顺序翻转一下，再传入backwardward layer中；再把forward output和backward output拼起来，形成整体的h out；最后返回序列整体的隐含状态和最后一个时刻的状态

现在验证双向rnn forward正确性

首先实例化双向RNN层；复制下来 bidirectional=True

```
# 验证一下 bidirectional_rnn_forward的正确性
bi_rnn =
nn.RNN(input_size, hidden_size, batch_first=True, bidirectional=True)
```

同样定义一个h_prev

```
h_prev = torch.zeros()
```

大小是 $2 \times \text{batch size} \times \text{hidden size}$

```
a packed sequence.  
• h_n: tensor of shape ( $D * \text{num\_layers}, N, H_{out}$ ) containing the final hidden state for each  
element in the batch.
```

```
h_prev = torch.zeros(2, bs, hidden_size)
```

我们调用这个RNN，传入input和h_prev,得到双向RNN的output，和双向state final

```
bi_rnn_output, bi_state_finall = bi_rnn(input, h_prev)
```

这是官方api的结果

```
# step3 手写一个bidirectional_rnn_forward函数, 实现双向RNN的计算原理  
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \  
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse):  
    bs, T, input_size = input.shape  
    h_dim = weight_ih.shape[0]  
    h_out = torch.zeros(bs, T, h_dim*2) # 初始化一个输出(状态)矩阵, 注意双向是两倍的特征大小  
  
    forward_output = rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev)[0] # forward layer  
    backward_output = rnn_forward(torch.flip(input, [1]), weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse)[0] # backward layer  
  
    h_out[:, :, :h_dim] = forward_output  
    h_out[:, :, h_dim:] = backward_output  
  
    return h_out, h_out[:, -1, :].reshape((bs, 2, h_dim)).transpose(0, 1)  
# 验证一下bidirectional_rnn_forward的正确性  
bi_rnn = nn.RNN(input_size, hidden_size, batch_first=True, bidirectional=True)  
h_prev = torch.zeros(2, bs, hidden_size)  
bi_rnn_output, bi_state_final = bi_rnn(input, h_prev)
```

对于这个RNN 也可以看一下参数的名字，就可以把这些参数代入到双向RNN，函数中去

```
for k,v in bi_rnn.named_parameters():  
    print(k,v)
```

```

# 验证一下bidirectional_rnn_forward的正确性
bi_rnn = nn.RNN(input_size, hidden_size, batch_first=True, bidirectional=True)
h_prev = torch.zeros(2, bs, hidden_size)
bi_rnn_output, bi_state_final = bi_rnn(input, h_prev)
for k,v in bi_rnn.named_parameters():
    print(k, v)

weight_ih_l0 Parameter containing:
tensor([[ 0.1088,  0.1269],
       [ 0.0817, -0.3656],
       [ 0.5452,  0.3497]], requires_grad=True)
weight_hh_l0 Parameter containing:
tensor([[ 0.0341,  0.2469, -0.0656],
       [-0.5707, -0.1655,  0.5743],
       [-0.2377,  0.0394, -0.3067]], requires_grad=True)
bias_ih_l0 Parameter containing:
tensor([-0.2562, -0.2534,  0.2150], requires_grad=True)
bias_hh_l0 Parameter containing:
tensor([ 0.2211, -0.4103, -0.1559], requires_grad=True)
weight_ih_l0_reverse Parameter containing:
tensor([[-0.4200, -0.1857],
       [ 0.3122, -0.2222],
       [ 0.5468, -0.1380]], requires_grad=True)
weight_hh_l0_reverse Parameter containing:
tensor([[ 0.5398,  0.5316, -0.4019],
       [ 0.1471, -0.2442, -0.0342],
       [ 0.2000, -0.0072,  0.0157]], requires_grad=True)
bias_ih_l0_reverse Parameter containing:
tensor([ 0.5492,  0.2403,  0.4561], requires_grad=True)
bias_hh_l0_reverse Parameter containing:
tensor([ 0.5348,  0.0851, -0.4330], requires_grad=True)

```

可以看到在pytorch双向RNN 中有这么多参数，有weight ih lo; weight hh lo; bias ih lo; bias hh lo; weight ih lo reverse; weight hh lo reverse; bias ih lo; bias hh lo reverse; 可以看到一共有8个参数，就是因为 forward layer有4个参数； reverse layer也有4个参数；

有了这8个参数，就可以把这个8个参数传入到双向RNN中

首先把签名 copy下来

```

# steps 步骤 / bidirectional_rnn_forward函数，实现从forward到backward
def bidirectional_rnn_forward(input, weight_ih, weight_hh, bias_ih, bias_hh, h_prev, \
    weight_ih_reverse, weight_hh_reverse, bias_ih_reverse, bias_hh_reverse, h_prev_reverse): I

```

```

bidirectional_rnn_forward(input,
                         weight_ih, weight_hh,
                         bias_ih, bias_hh, h_prev,
                         weight_ih_reverse, weight_hh_reverse,
                         bias_ih_reverse, bias_hh_reverse,
                         h_prev_reverse)

```

那input不变； weight ih改成weight ih lo

weight hh同样， weight hh lo

奥对了，前面要加上bi_rnn.，也就是说把我们实例化的RNN层 传进来

bi_rnn.bias ih lo

bi_rnn.bias hh lo

```
bi_rnn = nn.RNN(input_size,hidden_size,batch_first=True)
h_prev = torch.zeros(2,bs,hidden_size)
bi_rnn_output,bi_state_final = bi_rnn(input,h_prev)
```

`h prev`需要注意一下，是三维的，前面有个`2`我们只需要传入第一个就好了`h prev[0]`

那么反向的也是类似的

bi_rnn.weight_ih_l0 reverse

后面也是一样 bi_rnn.weight hh lo reverse

bi_rnn.bias ih lo reverse

bi_rnn.bias hh lo reverse

`h prev reverse`, 我们用`h prev [1]`就好了

并定义 `custom_bi_rnn_output`, `custom_bi_state_fn`接收输出

接下来分别打印api的结果 和自己写的函数的结果

```
PyTorch API output:  
tensor([[[ 0.3433,  0.2587, -0.3621,  0.7698, -0.7718,  0.0057],  
       [ 0.3661,  0.3816, -0.9138,  0.4536, -0.1733, -0.2512],  
       [ 0.2900,  0.5627, -0.9739,  0.0438, -0.1577, -0.4674]],  
       [[ 0.3041,  0.3743, -0.1807,  0.4116, -0.9148, -0.3805],  
       [ 0.1930,  0.1964, -0.6753,  0.8398, -0.3411,  0.4770],  
       [ 0.1126,  0.7779, -0.5593,  0.2074, -0.8978, -0.5869]]],  
grad_fn=<TransposeBackward1>)  
tensor([[[ 0.2900,  0.5627, -0.9739],  
       [ 0.1126,  0.7779, -0.5593]],  
       [[ 0.7698, -0.7718,  0.0057],  
       [ 0.4116, -0.9148, -0.3805]]], grad_fn=<StackBackward>)  
  
custom bidirectional_rnn_forward function output:  
tensor([[[ 0.3433,  0.2587, -0.3621,  0.0438, -0.1577, -0.4674],  
       [ 0.3661,  0.3816, -0.9138,  0.4536, -0.1733, -0.2512],  
       [ 0.2900,  0.5627, -0.9739,  0.7698, -0.7718,  0.0057]],  
       [[ 0.3041,  0.3743, -0.1807,  0.2074, -0.8978, -0.5869],  
       [ 0.1930,  0.1964, -0.6753,  0.8398, -0.3411,  0.4770],  
       [ 0.1126,  0.7779, -0.5593,  0.4116, -0.9148, -0.3805]]],  
grad_fn=<CopySlices>)  
tensor([[[ 0.2900,  0.5627, -0.9739],  
       [ 0.1126,  0.7779, -0.5593]],  
       [[ 0.7698, -0.7718,  0.0057],  
       [ 0.4116, -0.9148, -0.3805]]], grad_fn=<TransposeBackward0>)
```

这个结果有问题；up主也没发现；（照着评论区改了就是各种翻转）

由于是双向的 hidden size=3，但是输出状态长度是6，这是因为双向的有拼接