

时间线：2024年11月7日 start 19.32

- 241109 18.46 done

Docs > torch.nn > RNNCell

▷

RNNCELL ↗

CLASS `torch.nn.RNNCell(input_size, hidden_size, bias=True, nonlinearity='tanh', device=None, dtype=None)` [SOURCE]

An Elman RNN cell with tanh or ReLU non-linearity.

$$h' = \tanh(W_{ih}x + b_{ih} + W_{hh}h + b_{hh})$$

If `nonlinearity` is `'relu'`, then ReLU is used in place of tanh.

Parameters

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **bias** – If `False`, then the layer does not use bias weights b_{ih} and b_{hh} . Default: `True`
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`

首先这个RNNCELL，可以理解为单步的迭代；因为所有的循环神经网络都是有很多步去迭代，最终把每一步的状态取出来作为输出；这里的RNNCELL，也就是说多个，每个时刻的计算就是一个RNNCELL，然后把多个RNNCELL连起来，其实就构成了一个RNN，所以无论是RNN也好，还是GRU也好，还是LSTM也好，它们都有各自的CELL，然后每个CELL，其实就是一个单步的运算，我们可以理解为单个时刻的运算，下面有一个例子

Examples:

```
>>> rnn = nn.RNNCell(10, 20)
>>> input = torch.randn(6, 3, 10)
>>> hx = torch.randn(3, 20)
>>> output = []
>>> for i in range(6):
    hx = rnn(input[i], hx)
    output.append(hx)
```

可以看到，首先实例化了一个 RNNCELL；这个RNNCELL的 input size和hidden size分别为10和20；然后我们定义一个 input 的训练特征，batch size是3，然后时间长度是6，然后特征维度是10，并且定义了一个初始的 hidden state(hx),然后就可以用RNNCELL，来去做每一次迭代，所以我们看到这里有一个 for循环，然后每一步会调用这个RNNCELL的实例化的操作，算出每一时刻的隐含状态 $hx=rnn(input(i),hx)$ 这儿应该是 h_x

所以RNNCELL就是单步的计算，包括 GRUCELL 和LSTMCELL，也是一样的都是单步的，如果是 RNN 是把多个RNNCELL连起来所以是多步的，这个API是比较简单的不再详细介绍

今天主要看 LSTM的API，LSTM

LSTM

CLASS `torch.nn.LSTM(*args, **kwargs)` [\[SOURCE\]](#)

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

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

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

where h_t is the hidden state at time t , c_t is the cell state at time t , x_t is the input at time t , h_{t-1} is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and i_t , f_t , g_t , o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and \odot is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l \geq 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

关于它的原理，可以看这篇博客：

Understanding LSTM Networks

Posted on August 27, 2015

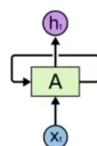
0

Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

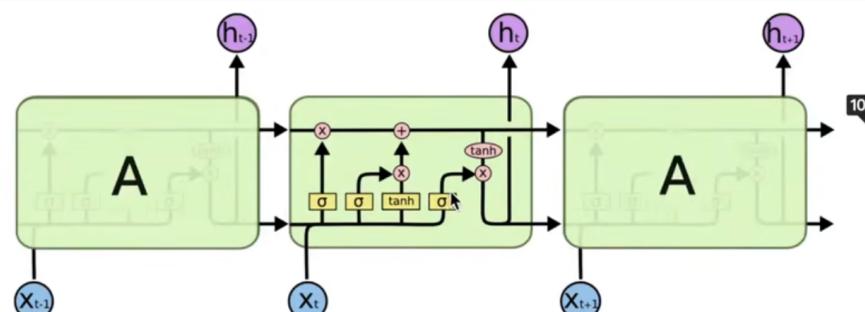
Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



五六年了，up主看LSTM也是看的这篇博客；

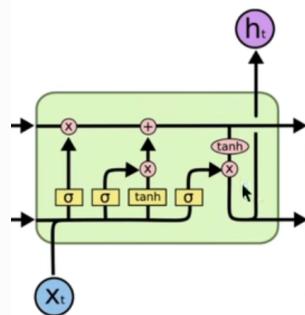
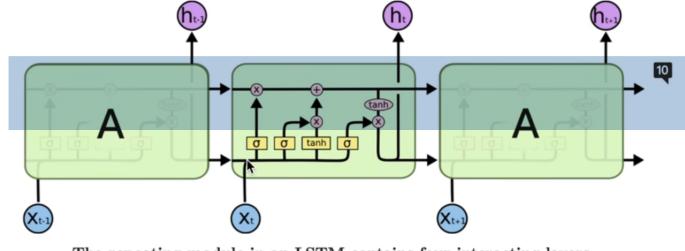
LSTM比RNN实际上是多了几个门；RNN比较简单就是输入和一个隐含状态，只有这两个状态；



The repeating module in an LSTM contains four interacting layers.

在LSTM中，多了一些门，包括输入门、输出门、遗忘门还有一个记忆单元，组成的结构；

我们只要看这幅图就好了



这幅图可以这样理解，最上面的横线，长得像传送带的东西，其实是一个细胞单元，或者说细胞状态，整个LSTM就是靠这个细胞状态，来不断的更新历史信息的，它有哪些们呢？

我们去看官网的API

LSTM

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

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

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

$$\begin{aligned}
 i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
 f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
 g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
 o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
 c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
 h_t &= o_t \odot \tanh(c_t)
 \end{aligned}$$

where h_t is the hidden state at time t , c_t is the cell state at time t , x_t is the input at time t , h_{t-1} is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0 , and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and \odot is the Hadamard product.

i就是输入门

f就是遗忘门

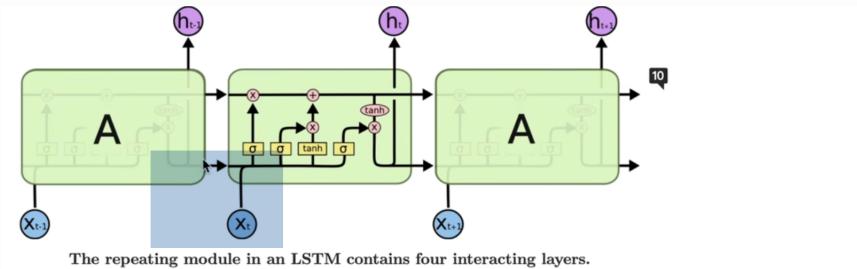
g就是细胞

o就是输出门

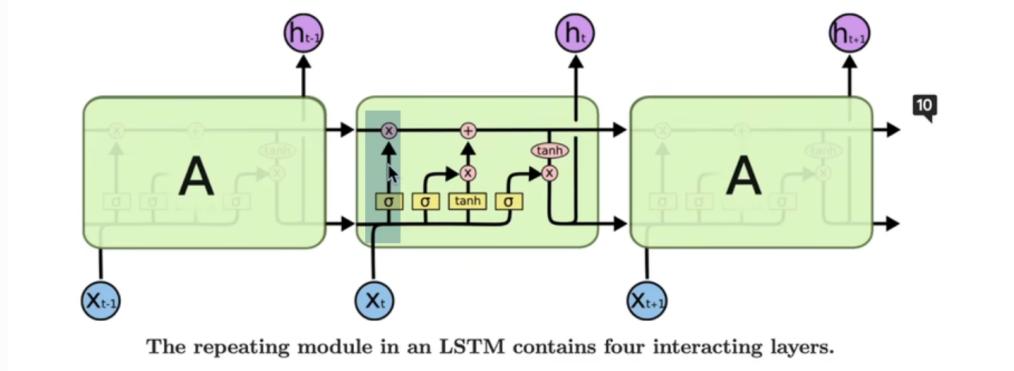
c成为cell，叫细胞单元，或者叫 细胞状态

h就是LSTM的隐含状态；或者说 输出；我们最终输出的是 h_t

api对应到图上，就是



首先 x_t 跟历史的 输出 进行一个交互；然后 经过 σ 这样的一个线性函数，得到遗忘门的输出



遗忘门的输出 跟上一时刻的 c_{t-1} 相乘，其实也就是

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

后面继续看，第二根线是什么呢？同样 x_t ，跟过往的 x_{t-1} ；进行一个交互，然后再经过一个 σ 函数，这得到 输入门 it

(听不懂)

然后 x_t 跟上一时刻的 x_{t-1} ，经过一个 \tanh 激活函数，得到一个什么呢？得到的是 g_t ，我们也称作叫 细胞；

g_t 跟输入门 相乘，相当于 我们对当前的输入信息，进行筛选，然后把这个信息 加到 目前最新的 c_t 上，最新的 c_t 是，上一时刻 c_t ，乘遗忘门，得到了 新的 c_t ，也就是说 我们把 该丢掉的，信息 丢掉了，然后我们再加上 输入门 ，得到的信息，其实就是 这个公式

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

, 最后一根线叫做 输出门 (太复杂了 根本记不住吧) 同样是 x_t 跟 h_{t-1} , 进行一个交互, 然后经过一个sigmoid函数, 得到 o_t , 这就是输出门, 我们最终的输出就是 $o_t \times \tanh(c_t)$, 也就是这里:

$$h_t = o_t \odot \tanh(c_t)$$

h_t 是跟每一时刻 的输入 进行交互的; 或者叫 线性组合; c_t 就是说 不断的 对 历史信息 进行一个 更新; 通过遗忘门、输入门、 不断对 c_t 进行一个 更新, 以上是LSTM的公式 和 结构;

LSTM相比于RNN, 在参数相同的条件下, LSTM的序列建模能力 是要 强于RNN的, 所以现在比较大的 序列建模任务 都是用 LSTM做的, 很少有用简单的 RNN 做序列建模的; 再次回顾 LSTM的公式, 从pytorch api的教程中 来看一下; 这边的api叫做 torch.nn.LSTM

LSTM

CLASS `torch.nn.LSTM(*args, **kwargs)` [\[SOURCE\]](#)



Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

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

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

where h_t is the hidden state at time t , c_t is the cell state at time t , x_t is the input at time t , h_{t-1} is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0 , and i_t , f_t , g_t , o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and \odot is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l \geq 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

If `proj_size > 0` is specified, LSTM with projections will be used. This changes the LSTM cell in the following way. First, the dimension of h_t will be changed from `hidden_size` to `proj_size` (dimensions of W_{hi} will be changed

这是一个class, 也就是一个类, 我们要用的话, 首先要对这个类 进行实例化, 然后得到一个算子, 然后再把 输入喂进去; 能够得到一个序列, 一个序列 经过LSTM网络之后, 得到的一个状态的输出; 最终 我们得到 状态的输出就是 h_t ; 或者说 每一时刻的 h_t 组合起来的一个序列;

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

这里有四个门，分别是i、f、g、o；这里有四个门，其中有三个门非线性激活函数都是sigmoid，那么gt的激活函数是tanh函数；其实这四个门的运算有很大的相似性；可以看到有四个W，并且这四个W，都是跟xt，进行一个矩阵相乘，这个W跟x之间没有任何符号的，就是表示的是矩阵相乘，同样这里的Whi、Whf等等右边这四个W，也是跟ht-1，进行一个矩阵相乘；所以这里虽然看上去有4个Wi，但是我们可以把这个四个Wi叠起来，比方说每个Wi是两行，那么4个Wi，就可以叠成8行；

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

然后再跟xt进行一个相乘；就说把这个四个W乘以x || W \times x组合起来，一起算；

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

同样这里的W乘以h (W \times h) 也是一样的；由于都是乘以同一个h；我们同样可以把四个W堆叠起来，stack堆叠来，算完了再split；

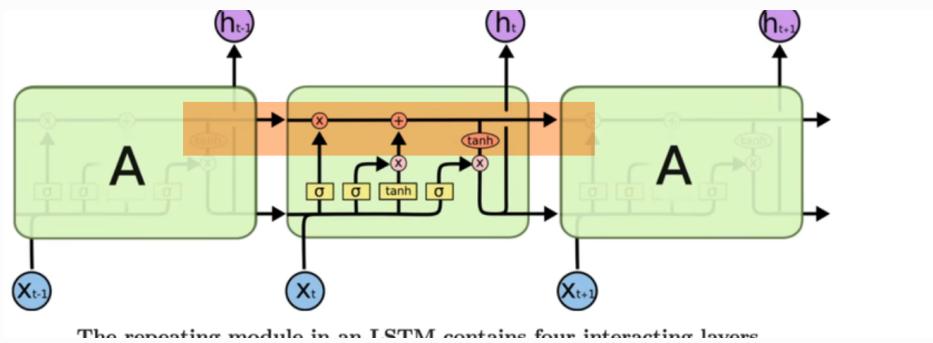
$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

这里还有四个bi和bh，就是对输入linear的偏置和上一时刻隐含状态的线性层的偏置；同样这里四个偏置，4个bi就是直接加了，不需要联合算了，同样这里的bh，也是四个偏置，维度都是跟it ff 维度是一样的，得到的i、f、g、o以后，就可以算出当前时刻细胞的状态ct，

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

c_t 就是 $f_t \times c_{t-1}$, 这里中间的乘, 是逐元素的乘, 它不是矩阵乘法; 我们默认 f_t 跟 c_{t-1} , 维度是一样的, 然后同一位置上的元素 两两相乘, 就可以了。同样 i_t 和 g_t 也是一样, 同一位置的两两元素 相乘; 乘完以后 元素再加起来, 得到 c_t

c_t 就是当前时刻的细胞状态, 其实就是上面的黑线



整个LSTM就是靠这个黑线, 来不断的对 历史信息 进行 筛选 和更新; 得到 c_t 以后, 最终 h_t

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

就是 我们LSTM的输出, 我们LSTM的输出 就是 h_t ; h_t 是由输出门 \times 细胞状态 经过 激活函数 \tanh 函数, 所得到的值, 就是我们的 h_t ; h_t 就是LSTM的输出;

之前RNN的时候, 我们有一个初始状态的概念, 同样在LSTM网络中, 也有一个初始状态, 但是这里的初始状态的概念, 有两个初始状态, 我们要找 初始状态, 其实很好找, 从公式里面找, 哪些符号以 $t-1$ 为下标的, 只要以 $t-1$ 维的就是说需要提供初始状态, 也就是说提供这些量的初始值; 也就是说 从 t 从 1 开始话, 带 $t-1$ 下标的, 就要提供 $t-1$, 所以一定有一个 初始状态; 从公式来看 一共有两个 带 $t-1$ 下标的

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

分别是 h_{t-1}, c_{t-1} , 也就是说在 $t=1$ 时刻的时候, 我们需要提供 h_0 , 和 c_0 , 来算出 $t=1$ 时刻的 h_1 和 c_1 , 就是说LSTM网络, 相比于简单的RNN网络, 初始状态就多了

我们再回顾一下RNN, 看RNN的api

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:

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

where \hat{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`.

首先 RNN的公式很简单, 其实就是 h_t 是 x_t 跟 h_{t-1} 的线性组合, 从RNN的公式中可以看出来, 只有一个符号, 就是 h 下标是 $t-1$, 那也就是说当我们去算RNN的网络的时候, 我们需要提供 h_0 作为初始状态, 因为如果我们要算 h_1 的话, 我们必须要有 h_0 , 所以我们必须提供 h_0 , 当然框架已经默认提供了 h_0 等于一个全0的向量,

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

在LSTM中, 根据公式也能看到必须要提供 h_0 和 c_0 , 这就是LSTM相比于RNN又多了一个初始状态, 不仅有 h_0 , 还有 c_0 , 在框架中, 同样提供了 默认的值; 就是全0

当然我们也可以 不用全0 的默认值, 也可以用 其他的值, 就是说 自己构造一个 h_0 和 c_0 , 就是说 h_0 和 c_0 , 可能是从某一个输入, 映射来的, 这种初始化方法也叫Meta learning; 就是说 我们的初始值 都是靠 学来的,

就是说我们可以让这个状态, 不是完全随机的, 可能与输入有关, 可以这样去构造, 或者说跟 condition有关, 都是可以构造的。

LSTM的所有公式 都在api里了 (看不懂、记不住一点

然后我们来看一下 参数

Parameters
<ul style="list-style-type: none">• 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 <code>num_layers=2</code> would mean stacking two LSTMs together to form a <i>stacked LSTM</i>, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1• bias – If <code>False</code>, then the layer does not use bias weights b_{ih} and b_{hh}. Default: <code>True</code>• batch_first – If <code>True</code>, 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: <code>False</code>• dropout – If non-zero, introduces a <i>Dropout</i> layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to <code>dropout</code>. Default: 0• bidirectional – If <code>True</code>, becomes a bidirectional LSTM. Default: <code>False</code>• proj_size – If > 0, will use LSTM with projections of corresponding size. Default: 0

首先，我们需要实例化LSTM的参数，包括 `input_size`,`hidden_size`, 其实就是 输入序列 特征的大小；

- 还有hidden size，就是LSTM网络 h的大小；当然这个hidden size 也就是c的大小
- 还有层数 也就是 num layers；我们可以构建多层的LSTM，也就是多层堆叠起来，也就是前一层的输出ht，是作为下一层LSTM的输入xt
- 然后还有 bias, bias决定了bi和bh是否可以丢弃
- 另外就是batch first，在pytorch中，默认的是batch是放在中间一维的；如果你觉得别扭的话，你可以把batch first设置成true，这时候 batch 就在 第一维
- 后面 还有Dropout，以及双向 bidirectional，如果要构建双向的话，会像上次讲的一样，有forward layer和backward layer；最后的状态是由forward layer和backward layer拼接起来的状态；
- 最后一个参数 `proj_size`,这个参数相当于LSTM网络的变体，也是今天要讲的主题LSTMP topic： LSTM和LSTMP的原理与源码实现

作用是为了减小LSTM的参数和计算量，因为LSTM的计算量确实是比较大的，我们通过LSTMP对ht进行压缩，ht的维度会变小，整个网络的参数量和运算量 都会变小，也有一些论文表明通过对 ht 进行压缩，性能损失 不是很大，所以在具体地实验中，可以尝试LSTMP这个网络；等下会实现LSTM和LSTMP

以上是所有的LSTM parameters，就是我们实例化所有LSTM class的时候，我们需要传入这么多参数，然后接下来就是说得到这么多实例化后的操作以后，就需要传入一个输入：

Inputs: input, (h_0, c_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 (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num_layers}, N, H_{cell})$ containing the initial cell state for each element in the batch. Defaults to zeros if (h_0, c_0) is not provided.

where:

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

我们要传入的input

- 第一个是 batch size×sequence length×input size。如果是 batch first=true的话
- 第二个是元组的形式，就是为了跟RNN的api保持一致，我们知道RNN的api输入就是两个量，LSTM是RNN一个特殊的变体，所以我们还是说保持，虽然有两个初始状态，用两个，但还是用元组的形式组合起来；这边的两个初始状态分别是 h0和c0；就是我们刚刚找到的，所有带 t-1 下标的；这些符号都需要提供一个初始值

output呢？它的output是有两部分

Outputs: output, (h_n, c_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 LSTM, 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 $(D * \text{num_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.
- **c_n**: tensor of shape $(D * \text{num_layers}, N, H_{cell})$ containing the final cell state for each element in the batch.

- 第一个呢，就是整个模型序列的输出，大小就是 $\text{batch size} \times \text{sequence length} \times \text{hidden size}$ ，这个是 output 也就是反应整个序列的状态输出；
 - 另外还有一个元组形式，就是 h_n ，和 c_n ，这个就是最后一个时刻的隐含状态和细胞状态

那它们的作用呢？都有作用，就是output的话，就是一个many to many的建模，就是说我们输入是一个序列，输出也是一个序列，并且这些序列我们都要，比方说，对一个文本的多音字进行预测，或者说词性进行预测；这些是many to many的任务，我们需要每一时刻的输出；

那 h_n 的话，就是一个many to one的任务，就比方说 我们输入一段话，到LSTM网络中，但是我们最终只取最后一个时刻的状态，为什么呢？因为我们希望最后一个时刻的状态，就能去表征整句话的特征，然后我们再对最后一个状态进行分类，或者说 就当做 sequence embedding也好 都可以，这个是 many to one的任务，就可以用到 h_n

以上是输入和输出；

还有一个点，如果带有projection的话，projection它是对谁进行压缩呢？它是对 h 进行压缩的，如果带有projection的话，我们这里的 h_n 大小就是 projection size，就不是hidden size，我们可以通过代码来演示；在上节课代码的基础上

写了一个RNN forward函数，实现了单向RNN的计算；

写了一个双向RNN的函数，通过这个函数可以知道双向的RNN函数是怎么干的

有一个forward layer还有一个backward layer；在backward layer首先需要对输入进行一个翻转，按照时间维度进行翻转，同样喂入到RNN forward函数中，这边各种翻转，再把forward output和backward output拼起来，在特征维度上拼起来就构成了 h out；这就是双向网络，双向GRU和双向LSTM原理也是一样的，所以不会再讲双向GRU和双向LSTM了

今天：LSTM&LSTMP

照着公式：

LSTM

CLASS `torch.nn.LSTM(*args, **kwargs)` [\[SOURCE\]](#)

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

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

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

where h_t is the hidden state at time t , c_t is the cell state at time t , x_t is the input at time t , h_{t-1} is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0 , and i_t , f_t , g_t , o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and \odot is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l \geq 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

实现LSTM和LSTMP的源码

首先调用官方的API，实现LSTM

首先定义一些常量，batch size、input

batch size；时间T、i_size 输入特征大小、hidden size 网络细胞状态的大小

```
# 定义常量  
bs,T,i_size,h_size = 2,3,4,5
```

还有projection size 也就是投影的大小，可以等下再确定

```
# proj_size
```

首先构建一个输入 input，就是我们要喂入到LSTM网络的一个特征序列，还是在用正态分布初始化 torch.randn 一个输入，

```
input = torch.randn(bs, T, i_size) # 输入序列
```

除了输入序列，还需要初始化两个初始状态，分别是c0和h0

假设只考虑一层，c0的初始状态就是 batch size×hidden size

```
c0 = torch.randn(bs, h_size)
```

因为c，本身就是一个向量，向量长度本身就是 hidden size；这里我们考虑到 batch维度，所以这里我们写 batch size×hidden size；这就是一个初始值，

```
c0 = torch.randn(bs, h_size) # 初始值 不需要参与训练
```

不需要训练，包括h0也是一样的，就是提供了h的初始值，我们写 bs×hidden size；我们首先写 hidden size；等下在考虑projection size

```
h0 = torch.randn(bs, h_size)
```

这样 我们定义好了三个基本的量；分别是输入和初始值

```
# 实现LSTM和LSTMP的源码
# 定义常量
bs, T, i_size, h_size = 2, 3, 4, 5
# proj_size
input = torch.randn(bs, T, i_size) # 输入序列
c0 = torch.randn(bs, h_size) # 初始值, 不需要训练
h0 = torch.randn(bs, h_size)
```

现在可以调用 官方的api

```
# 调用官方LSTM API
```

官方api就是nn.LSTM

nn.LSTM()

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 LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **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 LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If > 0 , will use LSTM with projections of corresponding size. Default: 0

传入的参数顺序分别是 `input size`; `hidden size`; `batch first` 我们先用到这些，`projection size` 暂时不用

`input size` 就是 `i size`; `hidden size` 就是 `h size`; `batch first` 设置成 `true`; 以上实例化了简单的LSTM layer, 定义: lstm layer

```
lstm_layer = nn.LSTM(i_size,h_size,batch_size=True)
```

在定义好LSTM layer以后, 我们就可以把输入和初始状态分别传入到LSTM layer中; 我们来看参数怎么传, 去看api

Inputs: input, (h_0, c_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 (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num_layers}, N, H_{cell})$ containing the initial cell state for each element in the batch. Defaults to zeros if (h_0, c_0) is not provided.

where:

N = batch size

L = sequence length

$D = 2$ if `bidirectional=True` otherwise 1

H_{in} = `input_size`

H_{cell} = `hidden_size`

从这个api可以看到，inputs是 input和一个元组，在元组中，我们需要传入h0， 和c0， 所以我们写input， 然后再写一个元组

```
lstm_layer(input,())
```

元组分别传入 h0 和c0

```
lstm_layer(input,(h0,c0))
```

那维度是多少呢？这里

- **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 (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num_layers}, N, H_{cell})$ containing the initial cell state for each element in the batch. Defaults to zeros if (h_0, c_0) is not provided.

h0的维度是 $D * \text{num_layers} \times N \times H_{out}$;c0也是一样的，首先我们现在初始化的是 $N \times H_{out}$

因为我们这里讲的是 单向的LSTM网络， 同样我们讲的是一层， 所以前面的数字我们省掉了； 那现在我们扩一下， 扩成 三维； 我们需要对 h_0 调用`unsqueeze`函数，在第0维扩一维； c_0 也同样扩维； 这样变成三维的张量； 第0维是1， 就是说 我们现在用的一层LSTM， 然后是单向的； 这样就传入好了， 得到输出；

讲的是LSTM & lstmp源码实现

输出也是 outputs 也是 output hn和cn

Outputs: output, (h_n, c_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 LSTM, 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 $(D * \text{num_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.
 - **c_n**: tensor of shape $(D * \text{num_layers}, N, H_{cell})$ containing the final cell state for each element in the batch.

这里我们写输出

这样调用好了api，我们可以打印output

```
print(output)
```

我们改名字，定义 $h_{\text{finall}}e$ 和 c_{finall} ，表示最后一个时刻的隐含状态和细胞状态

以上 官方api实现LSTM，首先看LSTM layer有哪些参数，可以调用LSTM layer的 named_parameter函数,打印权重和名字

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

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

weight_ih_10 Parameter containing:
tensor([[-0.0751, -0.3417,  0.0048, -0.3152],
       [-0.0956, -0.0862,  0.2326,  0.2558],
       [-0.2981,  0.0209,  0.0918,  0.2844],
       [-0.4270,  0.1765,  0.1386, -0.1095],
       [ 0.3817, -0.1764,  0.0189, -0.1523],
       [ 0.3042, -0.2558, -0.0183,  0.0902],
       [ 0.3179, -0.3432,  0.3432,  0.1860],
       [ 0.3149,  0.3196, -0.0595, -0.1064],
       [ 0.1308, -0.3106, -0.1718, -0.2957],
       [ 0.0144,  0.2237, -0.4342,  0.2291],
       [-0.3550,  0.3979,  0.2086,  0.0497],
       [-0.0039, -0.1336, -0.0202, -0.2414],
       [-0.0268,  0.2100, -0.2255,  0.1231],
       [-0.3835,  0.0839,  0.4422,  0.0739],
       [-0.2856,  0.1064,  0.0349, -0.0077],
       [ 0.3312,  0.2806,  0.1320, -0.0467],
       [-0.2960, -0.1034,  0.3261, -0.2461],
       [-0.0121, -0.3744, -0.3223, -0.0249],
       [ 0.0737, -0.2916,  0.3341, -0.3139],
       [ 0.0015,  0.0627,  0.1504, -0.1644]], requires_grad=True)
weight_hh_10 Parameter containing:
tensor([[ -2.3076e-01,   2.2166e-01,   2.1356e-01,  -3.4520e-01,  -2.9890e-01],
       [ -2.0311e-01,  -3.5511e-01,   2.2567e-01,   2.5014e-01,   2.6045e-01],
       [ -7.7830e-02,   3.8349e-01,   3.2541e-01,   3.9111e-01,  -3.5383e-01],
       [  5.9957e-02,  -1.3982e-01,  -3.7899e-01,   3.5019e-01,  -1.4942e-01],
       [ -3.6386e-01,  -9.7632e-02,  -3.1191e-01,  -4.5985e-02,   3.4561e-02],
       [  3.5979e-01,  -3.3885e-01,   1.5645e-01,   1.5917e-01,  -3.2042e-01],
       [  2.9041e-01,  -2.3166e-01,   1.5727e-04,  -4.0081e-01,   3.6477e-01],
       [  3.0256e-02,   3.3858e-01,   3.9964e-01,  -2.7578e-01,  -3.7292e-01],
       [  2.4728e-01,  -3.0434e-01,  -1.3916e-01,   1.1019e-01,  -3.9678e-01],
```

可以看到LSTM的权重，以及具体的张量；对于这个LSTM有的张量：

- `weight_ih_10` 对应公式里的 W_{ii} W_{if} W_{ig} W_{io} 四个 W_i 放到了一个 weight ih里 面
- `weight_hh_10` 这个参数是公式的四个 W_{hi} W_{hf} W_{hg} W_{ho} W_h 拼起来的
- `bias_ih_10` 然后还有两个偏置项；同样是拼起来的，这样直接看张量不清晰，接下 来 我们直接看shape
- `bias_hh_10`

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

```
5 for k,v in lstm_layer.named_parameters():
6     print(k,v.shape)
[6]    ✓ 0.0s                                         Python
...
weight_ih_l0 torch.Size([20, 4])
weight_hh_l0 torch.Size([20, 5])
bias_ih_l0 torch.Size([20])
bias_hh_l0 torch.Size([20])
```

可以看到在这个LSTM layer中一共有四个参数；

第一个参数是 weight_ih_l0 这个参数是 20×4 ；为什么是 20×4 呢？ 20 是 hidden size，就是 5 这个维度，然后我们把 4 个 W ，拼起来，本来每一个是 5 行，现在拼成了 20 行， 20 就是 5×4 来的；

后面的 4 呢，是 input size，因为这个 W_{ih} 是跟 input 进行一个相乘的；是对 input 进行线性变换的参数；

```
In [6]: # 实现LSTM和LSTMP的源码
# 定义常量
bs, T, i_size, h_size = 2, 3, 4, 5
# proj_size
input = torch.randn(bs, T, i_size) # 输入序列
c0 = torch.randn(bs, h_size) # 初始值, 不需要训练
h0 = torch.randn(bs, h_size)

# 调用官方LSTM API
lstm_layer = nn.LSTM(i_size, h_size, batch_first=True)
output, (h_final, c_final) = lstm_layer(input, (h0.unsqueeze(0), c0.unsqueeze(0)))
for k, v in lstm_layer.named_parameters():
    print(k, v.shape)

weight_ih_l0 torch.Size([20, 4])
weight_hh_l0 torch.Size([20, 5])
bias_ih_l0 torch.Size([20])
bias_hh_l0 torch.Size([20])
```

然后第二个参数 W_{hh} 参数是 20×5 的，同样这个 20 也是 4×5 来的，然后后面这个 5 就是我们 W_{hh} 是跟上一时刻的 隐含状态 进行一个线性变换的，所以它的维度是 5 ；

其实它就是一个 linear 层；就是 $y = wx + b$ ； w 的维度就是 $hidden\ size \times input\ size$ ；这里的 $input\ size$ 就是 5 ，上一个的 $input\ size$ 就是 4 ；

然后另外两个 $bias$ 就比较好理解了；

$bias_{ih}$ 和 $bias_{hh}$ 都是 20 ；这个 20 呢 就是 4×5 来的；就是有 4 个 $bias$ ； 4 个 bi 和 4 个 bh

所以我们要把这四个拼起来了，就是两个，每一个长度都是20的；以上是LSTM不带projection的；我们可以根据这些参数，来自己写一个LSTM模型

自己写一个LSTM模型

根据上面的参数、以及 I_0 、以及 c_0 、以及 input ，就能自己写一个LSTM

```
def lstm_forward():
    pass
```

首先思考这个LSTM模型，需要哪些输入呢？

第一个肯定是 input ，就是我们需要传入 input

第二个是 initial states，这个是元组的形式

然后第三个就是一些权重，包括 W_{ih} ; W_{hh} ; b_{ih} ; b_{hh} ，分别是权重和 bias；有了这些就可以了；这就是LSTM forward的一个签名

```
def lstm_forward(input,initial_states,w_ih,w_hh,b_ih,b_hh):
```

当然如果带 projection的话，后面还需要再加一些参数

首先把initial states拆解出来，首先是 h_0 和 c_0 ；

```
h0,c0 = initial_states # 初始状态
```

然后对 input.shape 也拆解一下，通过 input shape 得到batch size、得到时间T，得到 input size ，得到时间的话，就可以进行for 循环，不断的迭代，不断的运算

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

以上是 input size 还是有 h size ， h size 怎么得到呢？ h size 可以根据 W 随便的 W 的维度来确定就好了；比如我们用 w_{ih} ；然后除以4就好了，因为它 是 四个 W 拼起来的

```
h_size = w_ih.shape[0]//4
```

我们把它的第0维除以4，就是每一维的hidden size

然后我们把 h_0 和 c_0 换一下名字， $prev_h$ 和 $prev_c$ ；因为我们在for循环中，不断的更新 $prev\ h$ 和 $prev\ c$

```
prev_h = h0  
prev_c = c0
```

就是把每一时刻的 h 和 c 当做下一时刻的 $prev\ h$ 和 $prev\ c$

另外还有一个size 叫做 output size；也就是 我们输出的状态大小 就是 hidden size

```
output_size = h_size
```

这样 我们就能初始化一个 output了;在写神经网络 或者 循环神经网络，我们都在初始化一个矩阵，然后对矩阵进行填充；这个矩阵的大小，跟输入特征大小是一样的，跟输入的序列大小是一样的batch size和 time这个维度不变，然后特征维度 改成 output size就好了，这样就是 输出序列 初始化好了

```
output = torch.zeros(bs, T, output_size) # 输出序列
```

当然这个 只是初始化的，接下来，我们就可以对时间 进行 一个遍历就好了

LSTM就是每一时刻 都在对上一时刻的 c 和 h 进行更新，所以我们要写一个 for循环就好了，对每一时刻 进行一个运算；这个循环的步数 就是 大T步

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

在每一个循环的开始，我们需要拿到当前这一时刻的 x ，那么我们就可以通过input拿；因为 input这个维度就是 batch size×T×input size，所以这里 我们拿到 t 这一维度就好了，就是 当前时刻的输入向量

```
x = input[:, t, :] # 当前时刻的输入向量
```

接下来 我们就要根据公式 来进行计算

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

首先计算 $W \times x$; 再计算 $W \times h$; 就是说 我们把 大的一块 先算出来; 就是拼起来的 W 分别与 x 和 h 进行一个相乘, 那么这里相乘, 我们需要运用的是 带batch的矩阵相乘;

我们首先需要弄清楚, 这里的 W_{ih} 和 W_{hh} , 它们的维度是什么;

```
w_ih #  
w_hh
```

现在 w_{ih} 就是 4倍的 $hidden size \times input size$

那下面的呢? 也是4倍的 $hidden size \times hidden size$

这是这两个权重 目前的维度

```
w_ih # [4*h_size, i_size]  
w_hh # [4*h_size, h_size]
```

接下来 我们要将 $w_{ih} \times x$

那现在 x 的维度是多少呢? 就是 $batch size \times input size$;

```
x = input[:, t, :] # 当前时刻的输入向量, [bs, i_size]
```

我们可以看到 现在的 x 是带 batch 的, 但是我们的 w 是不带batch的; 所以我们首先要对 w , 进行 复制一下; 在复制之前, 首先扩个维度; 就是说把batch 维度 扩出来; batch维放在开始, 所以扩 0维;

```
w_ih.unsqueeze(0) # [4*h_size, i_size]  
w_hh # [4*h_size, h_size]
```

我们真正要的batch size是bs，所以我们复制一下就好了，用.tile函数，复制就是对第0维，复制bs倍，后面两个维度不变，我们把这个变量叫做 batch w ih

```
batch_w_ih = w_ih.unsqueeze(0).tile(bs,1,1) #  
[bs,4*h_size,i_size]
```

这时候就多了一维，权重就变成了三维；同样w hh也是一样的

首先扩一个batch维度，然后tile复制一下，定义为batch w hh

```
batch_w_hh = w_hh.unsqueeze(0).tile(bs,1,1)
```

维度就变成了 $bs \times 4$ 倍的hidden size \times hidden size

```
batch_w_ih = w_ih.unsqueeze(0).tile(bs,1,1) #  
[bs,4*h_size,i_size]  
batch_w_hh = w_hh.unsqueeze(0).tile(bs,1,1) #  
[bs,4*h_size,h_size]
```

以上是对权重进行扩维；扩维以后：

- 当前的 batch w ih 形状是 $bs \times 4$ 倍的hidden size \times input size
- 当前的输入向量 x 的形状是：batch size \times input size

我们要让这两个矩阵进行 bmm 的相乘，也就是batch matrix multiplication；那我们就要保持 batch 这个维度是相同的；后面的两个维度要满足矩阵乘法的基本规则：也就是第一个矩阵的列数 = 第二个矩阵的行数

所以我们要对 x 进行扩维；同样对x的第三维增加一个维度；然后就可以去算了；首先计算 `w_times_x`；调用一下 `torch.bmm` 函数；首先传入 batch w ih；

```
w_times_x = torch.bmm(batch_w_ih, )
```

然后传入 x，并对 x 进行一个扩维；在-1维扩一维，变成 batch size \times input size \times 1，然后就可以进行相乘了

```
w_times_x = torch.bmm(batch_w_ih, x.unsqueeze(-1)) #  
[bs, 4*h_size, 1]
```

相乘以后的维度是多少呢？相乘以后的维度就应该是 batch size × 4倍的hidden size × 1；但是这个1这个维度我们就不要了；我们后面再把1这个维度去掉

```
w_times_x = w_times_x.squeeze(-1) # [bs, 4*h_size]
```

这样我们就把1这个维度去掉了；形状是 batch size × 4倍的hidden size

以上是 w times x；我们已经算出来了

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

现在实现完了 W times x；具体来说就是 $W_{ii}x_t$ 、 $W_{if}x_t$ 、 $W_{ig}x_t$ 、 $W_{io}x_t$

还有 w times h就是，LSTM网络中后四项

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

同样也是一样的写法；复制下来改成 w_hh;后面改成 prev h

因为是跟 h_{t-1} 进行线性组合；所以也要写成 h prev；同样也要把维度变成二维的

```
w_times_h = torch.bmm(batch_w_hh, h_prev.unsqueeze(-1))  
# [bs, 4*h_size, 1]  
w_times_h = w_times_h.squeeze(-1) # [bs, 4*h_size]
```

以上算出 wx和wh；最后我们把名称改成 h_prev更好一点，因为是跟上一时刻的 hidden state 进行线性组合

```
w_times_h_prev = torch.bmm(batch_w_hh, h_prev.unsqueeze(-1))  
# [bs, 4*h_size, 1]  
w_times_h_prev = w_times_h.squeeze(-1) # [bs, 4*h_size]
```

有了这些以后，分别算出 输入门、遗忘门、cell和输出门，也就是 if g和o

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

```
# 分别计算输入门(i)、遗忘门(f)、cell门(g)、输出门(o)
```

首先计算 i_t ; 首先根据公式

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

i_t 是 Wx 的第一部分结果+b+ Wh 的第一部分结果+b；这就很简单了；首先把 $w \times x$ 的第一部分结果 取出来；现在 $w \times x$ 的结果是 batch size $\times 4$ 倍的hidden size；现在我们把 batch size这一维度 全部拿出来；hidden size这一维 只拿前 第一部分 hidden size这一维；

因为 我们现在 $w \times x$ 是一个 大的拼起来的结果，我们目前只需要 取前 $\frac{1}{4}$

```
i_t = w_times_x[:, :h_size] +
```

后面也一样 $w \times h$ _prev也是要取 前 $\frac{1}{4}$

```
i_t = w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] +
```

然后还有两个偏置，偏置就是 bih 和 bhh ，也是一样的，只取 前 $\frac{1}{4}$

```
i_t =
w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_ih[:h_size] + b_
hh[:h_size]
```

最后还有非线性激活函数 σ

所以我们在前面加上 `torch.sigmoid`

```
# 自己写一个LSTM模型
def lstm_forward(input, initial_states, w_ih, w_hh, b_ih, b_hh):
    h0, c0 = initial_states # 初始状态
    bs, T, i_size = input.shape
    h_size = w_ih.shape[0] // 4

    prev_h = h0
    prev_c = c0
    batch_w_ih = w_ih.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, i_size]
    batch_w_hh = w_hh.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, h_size]

    output_size = h_size
    output = torch.zeros(bs, T, output_size) # 输出序列

    for t in range(T):
        x = input[:, t, :] # 当前时刻的输入向量, [bs, i_size]
        w_times_x = torch.bmm(batch_w_ih, x.unsqueeze(-1)) # [bs, 4*h_size, 1]
        w_times_x = w_times_x.squeeze(-1) # [bs, 4*h_size]

        w_times_h_prev = torch.bmm(batch_w_hh, prev_h.unsqueeze(-1)) # [bs, 4*h_size, 1]
        w_times_h_prev = w_times_h_prev.squeeze(-1) # [bs, 4*h_size]

        # 分别计算输入门(i)、遗忘门(f)、cell门(g)、输出门(o)
        i_t = torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_ih[:h_size] + b_hh[:h_size])
```

这就是 i_t ; i_t 就是这么来的; 以上是输入门的计算就是 w 乘以 x 的前面四分之一部分, 包括 w 乘以 h_{prev} 也是前四分之一部分; 然后两个 bias 加起来, 经过一个非线性激活函数 `sigmoid` 就得到输入门;

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

接下来遗忘门, 遗忘门也是类似的, 同样也是 `sigmoid`, 所以这里复制一下; 但是遗忘门就不是前四分之一

```
i_t =
torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_
ih[:h_size] + b_hh[:h_size])

i_t =
torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_
ih[:h_size] + b_hh[:h_size])
```

而是前四分之一 到二分之一的部分；后面也是一样的

```
w_times_x(:,h_size:2*h_size)
```

```
f_t =
torch.sigmoid(w_times_x[:,h_size:2*h_size]+w_times_h_prev[:,h_size:2*h_size]+b_ih[h_size:2*h_size]+b_hh[h_size:2*h_size])
```

hidden size 到 2倍的hidden size；以上是 遗忘门；

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

那么细胞门gt呢；gt也是类似的；只不过 当前 sigmoid 替换成了tanh函数

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \end{aligned}$$

所以我们把 gt；继续复制下来，再改 再写一遍

```
g_t =
torch.tanh(w_times_x[:,2*h_size:3*h_size]+w_times_h_prev[:,2*h_size:3*h_size]+b_ih[2*h_size:3*h_size]+b_hh[2*h_size:3*h_size])
```

gt是二倍的hidden size 到 三倍的 hidden size这个区间，同样后面的偏置也是一样的；换一个区间，然后 非线性激活函数 改成 tanh函数

以上是gt，最后是ot，也就是 输出门；tanh函数 换成 sigmoid函数；最后区间 是最后的 四分之一；可以写成 3倍的hidden size到4倍的hidden size；更清晰一点

```
o_t =
torch.sigmoid(w_times_x[:,3*h_size:4*h_size]+w_times_h_prev[:,3*h_size:4*h_size]+b_ih[3*h_size:4*h_size]+b_hh[3*h_size:4*h_size])
```

或者直接 `w_times_x[:,3*h_size:]` 3倍的 hidden size；4可以省略

以上写完了 所有的 ifgo

```

# 分别计算输入门(i)、遗忘门(f)、cell门(g)、输出门(o)
i_t = torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_ih[:h_size] + b_hh[:h_size])
f_t = torch.sigmoid(w_times_x[:, h_size:2*h_size] + w_times_h_prev[:, h_size:2*h_size] \
                     + b_ih[h_size:2*h_size] + b_hh[h_size:2*h_size])
g_t = torch.tanh(w_times_x[:, 2*h_size:3*h_size] + w_times_h_prev[:, 2*h_size:3*h_size] \
                  + b_ih[2*h_size:3*h_size] + b_hh[2*h_size:3*h_size])
o_t = torch.sigmoid(w_times_x[:, 3*h_size:4*h_size] + w_times_h_prev[:, 3*h_size:4*h_size] \
                     + b_ih[3*h_size:4*h_size] + b_hh[3*h_size:4*h_size])
```

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

✓

写完了 ifgo，接下来写细胞状态 ct ht

ct怎么写呢？ct直接是元素相乘，我们不用ct，我们用prev c；因为我们现在用for循环迭代；那么我们要保证下一时刻prev c的量是存在的；我们现在用prev c表示ct；那prev c就等于 ft×ct-1加上 it×gt；其实就是 ft×prev c；加上 it× gt；这个就是对 prev c进行一个更新

```
prev_c = f_t * prev_c + i_t * g_t
```

有了 prev c以后就能计算 prev h；

$$h_t = o_t \odot \tanh(c_t)$$

prev h就是当前时刻 LSTM的一个输出；按照公式就是 输出门 × tanh 细胞状态

```
prev_h = o_t * torch.tanh(prev_c)
```

现在就是对 c和 h进行一个更新

有了 h以后，就可以对输出的矩阵也更新一下

```
output[:, t, :] = prev_h
```

这样就写完了；然后我们返回一下就好了，返回第一个就是输出序列；后面是两个状态构成的元组；这两个状态分别是最后一个时刻的输出和最后一个时刻的细胞状态

```
return output, (prev_h, prev_c)
```

以上实现了整个LSTM：完整的代码如下；

```
# 自己写一个LSTM模型
def lstm_forward(input, initial_states, w_ih, w_hh, b_ih, b_hh):
    h0, c0 = initial_states # 初始状态
    bs, T, i_size = input.shape
    h_size = w_ih.shape[0] // 4

    prev_h = h0
    prev_c = c0
    batch_w_ih = w_ih.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, i_size]
    batch_w_hh = w_hh.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, h_size]

    output_size = h_size
    output = torch.zeros(bs, T, output_size) # 输出序列

    for t in range(T):
        x = input[:, t, :] # 当前时刻的输入向量, [bs, i_size]
        w_times_x = torch.bmm(batch_w_ih, x.unsqueeze(-1)) # [bs, 4*h_size, 1]
        w_times_x = w_times_x.squeeze(-1) # [bs, 4*h_size]

        w_times_h_prev = torch.bmm(batch_w_hh, prev_h.unsqueeze(-1)) # [bs, 4*h_size, 1]
        w_times_h_prev = w_times_h_prev.squeeze(-1) # [bs, 4*h_size]

        # 分别计算输入门(i)、遗忘门(f)、cell门(g)、输出门(o)
        i_t = torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_ih[:h_size] + b_hh[:h_size])
        f_t = torch.sigmoid(w_times_x[:, h_size:2*h_size] + w_times_h_prev[:, h_size:2*h_size] \
                            + b_ih[h_size:2*h_size] + b_hh[h_size:2*h_size])
        g_t = torch.tanh(w_times_x[:, 2*h_size:3*h_size] + w_times_h_prev[:, 2*h_size:3*h_size] \
                          + b_ih[2*h_size:3*h_size] + b_hh[2*h_size:3*h_size])
        o_t = torch.sigmoid(w_times_x[:, 3*h_size:4*h_size] + w_times_h_prev[:, 3*h_size:4*h_size] \
                            + b_ih[3*h_size:4*h_size] + b_hh[3*h_size:4*h_size])

        prev_c = f_t * prev_c + i_t * g_t
        prev_h = o_t * torch.tanh(prev_c)

        output[:, t, :] = prev_h

    return output, (prev_h, prev_c)
```

以上不带 projection 的 LSTM 就写完了；接下来 测试；测试就是把 LSTM layer 的 4 个参数拿出来，然后喂入到自己写的函数中，然后对比结果

首先把函数签名复制下来：

```
lstm_forward(input, initial_states, w_ih, w_hh, b_ih, b_hh)

weight_ih_10 torch.Size([20, 4])
weight_hh_10 torch.Size([20, 5])
bias_ih_10 torch.Size([20])
bias_hh_10 torch.Size([20])
```

input还是input, initial state就是h0和c0；就是我们之前初始化的h0和c0

```
lstm_forward(input,(h0,c0),w_ih,w_hh,b_ih,b_hh)
```

这里的w_ih就可以用我们之前pytorch中 实例化的lstm layer的参数拿出来就好了

```
# 调用官方LSTM API
lstm_layer = nn.LSTM(i_size, h_size, batch_first=True)
output, (h_final, c_final) = lstm_layer(input, (h0.unsqueeze(0), c0.unsqueeze(0)))
for k, v in lstm_layer.named_parameters():
    print(k, v.shape)

# 自己写一个LSTM模型
def lstm_forward(input, initial states, w_ih, w_hh, b_ih, b_hh):
```

就是lstm_layer这个参数

```
    return output, (prev_h, prev_c)

lstm_forward(input, (h0, c0), lstm_layer.weight_ih_10, w_hh, b_ih, b_hh)

weight_ih_10 torch.Size([20, 4])
weight_hh_10 torch.Size([20, 5])
bias_ih_10 torch.Size([20])
bias_hh_10 torch.Size([20])
```

```
lstm_forward(input,
(h0,c0),lstm_layer.weight_ih_10,w_hh,b_ih,b_hh)
```

w hh也是一样的;然后后面还有两个偏置

```
lstm_forward(input,
(h0,c0),
lstm_layer.weight_ih_10,
lstm_layer.weight_hh_10,
lstm_layer.bias_ih_10,
lstm_layer.bias_hh_10)
```

这样我们就实例化好了 我们自己写的lstm forward函数; 前面叫 output, 我们把名字也拷贝下来

```
output, (h_final, c_final) = lstm_forward(input, (h0, c0), lstm_layer.weight_ih_10, lstm_layer.weight_hh_10, \
lstm_layer.bias_ih_10, lstm_layer.bias_hh_10)
```

我们加后缀 custom, 表示自定义的,运行没有问题

```
output_custom, (h_finall_custom,c_finall_custom) = lstm_forward(  
    input,  
    (h0,c0),  
    lstm_layer.weight_ih_10,  
    lstm_layer.weight_hh_10,  
    lstm_layer.bias_ih_10,  
    lstm_layer.bias_hh_10)
```

接下来 我们对比 前面的 output 和我们自定义实现的 output custom, 查看是不是一致 用 torch.allclose()

```
print(torch.allclose(output,output_custom))  
print(torch.allclose(h_finall,h_finall_custom))  
print(torch.allclose(c_finall,c_finall_custom))
```

输出三个True,

后面要讲projection, 首先什么是 projection呢? 如果要写 projection呢, 需要怎么改造。需要在函数里面改造一下就好了; 但是在改造之前, 首先还是对 上面lstm layer; 调用一下官方的projection的api, 看一下 参数有什么变化;

```
1 import torch  
2 import torch.nn as nn  
3 import torch.nn.functional as F  
1] ✓ 0.0s Python
```

实现LSTM & LSTMMP源码

```
1 # 定义常量  
2 bs,T,i_size,h_size = 2,3,4,5  
3 # proj_size  
4 input = torch.randn(bs,T,i_size) # 输入序列  
5 c0 = torch.randn(bs,h_size) # 初始值不需要训练  
6 h0 = torch.randn(bs,h_size)  
2] ✓ 0.0s Python
```

```
[6] 1 # 调用官方LSTM API  
2 lstm_layer = nn.LSTM(i_size,h_size,batch_first=True)  
3 output,(h_final,c_final) = lstm_layer(input,(h0.unsqueeze(0),c0.unsqueeze(0)))  
4  
5 for k,v in lstm_layer.named_parameters():  
6 |   print(k,v.shape)  
7  
[6]  ✓ 0.0s
```

```
1 # 自己写一个LSTM
2 def lstm_forward(input, initial_states, w_ih, w_hh, b_ih, b_hh):
3     # 以上写好了 函数签名
4     h0, c0 = initial_states #初始状态
5     bs, T, i_size = input.shape
6     h_size = w_ih.shape[0] // 4
7
8     prev_h = h0
9     prev_c = c0
10    batch_w_ih = w_ih.unsqueeze(0).tile(bs, 1, 1)
11    batch_w_hh = w_hh.unsqueeze(0).tile(bs, 1, 1)
12
13    output_size = h_size
14    output = torch.zeros(bs, T, output_size) # 输出序列
15
16    for t in range(T):
17        x = input[:, t, :] # 当前时刻的输入向量, [bs, i_size]
18
19        w_times_x = torch.bmm(batch_w_ih, x.unsqueeze(-1)) #[bs, 4*h_size, 1]
20        w_times_x = w_times_x.squeeze(-1) # [bs, 4*h_size]
21
22        w_times_h_prev = torch.bmm(batch_w_hh, prev_h.unsqueeze(-1)) #[bs, 4*h_size, 1]
23        w_times_h_prev = w_times_h_prev.squeeze(-1) # [bs, 4*h_size]
24
25        # 分别计算 输入门(i), 遗忘门(f), cell门(g), 输出门(o)
26        i_t = torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_ih[:h_size] + b_hh[:h_size])
27        f_t = torch.sigmoid(w_times_x[:, h_size:2*h_size] + w_times_h_prev[:, h_size:2*h_size] +
28                             b_ih[h_size:2*h_size] + b_hh[h_size:2*h_size])
29        g_t = torch.tanh(w_times_x[:, 2*h_size:3*h_size] + w_times_h_prev[:, 2*h_size:3*h_size] +
30                           b_ih[2*h_size:3*h_size] + b_hh[2*h_size:3*h_size])
31        o_t = torch.sigmoid(w_times_x[:, 3*h_size:4*h_size] (parameter) b_hh: Any [:, 3*h_size:4*h_size] +
32                           b_ih[3*h_size:4*h_size] + b_hh[3*h_size:4*h_size])
33
34
35        prev_c = f_t * prev_c + i_t * g_t
36        prev_h = o_t * torch.tanh(prev_c)
37
38        output[:, t, :] = prev_h
39
40    return output, (prev_h, prev_c)
41
42 output_custom, (h_final_custom, c_final_custom) = lstm_forward(input, (h0, c0), lstm_layer.weight_ih_l0,
43                                                               lstm_layer.weight_hh_l0,
44                                                               lstm_layer.bias_ih_l0, lstm_layer.bias_hh_l0)
```

```

1 print(torch.allclose(output, output_custom))
2 print(torch.allclose(h_final, h_final_custom))
3 print(torch.allclose(c_final, c_final_custom))

[10]  ✓ 0.0s Python
... True
True
True

```

如果要调用 官方的话 需要加一个量 proj_size， 等于多少呢？我们把上面的proj size反注释

```

In [9]: # 实现LSTM和LSTMP的源码
# 定义常量
bs, T, i_size, h_size = 2, 3, 4, 5
proj_size = 1
input = torch.randn(bs, T, i_size) # 输入序列
c0 = torch.randn(bs, h_size) #初始值, 不需要训练
h0 = torch.randn(bs, h_size)

# 调用官方LSTM API
lstm_layer = nn.LSTM(i_size, h_size, batch_first=True)
output, (h_final, c_final) = lstm_layer(input, (h0.unsqueeze(0), c0.unsqueeze(0)), proj_size=)
print(output)
#for k, v in lstm_layer.named_parameters():
#    print(k, v.shape)

# 自己写一个LSTM模型
def lstm_forward(input, initial_states, w_ih, w_hh, b_ih, b_hh):
    h0, c0 = initial_states # 初始化状态

```

一般 projection size是比hidden size是要小的；就是说 要对hidden state进行一个压缩；就是说 压缩 肯定是要往小的维度压缩了；那我们hidden size等于5的话，那projection size就设置成3， 比hidden size小一点就好了， 这样就实现了一个 projection layer

```

In [9]: # 实现LSTM和LSTMP的源码
# 定义常量
bs, T, i_size, h_size = 2, 3, 4, 5
proj_size = 3
input = torch.randn(bs, T, i_size) # 输入序列
c0 = torch.randn(bs, h_size) #初始值, 不需要训练
h0 = torch.randn(bs, h_size)

# 调用官方LSTM API
lstm_layer = nn.LSTM(i_size, h_size, batch_first=True)
output, (h_final, c_final) = lstm_layer(input, (h0.unsqueeze(0), c0.unsqueeze(0)), proj_size=proj_size)
print(output)
#for k, v in lstm_layer.named_parameters():
#    print(k, v.shape)

```

现在 我们再来看 lstm layer的参数输出;传入proj size以后，还要改变h0

因为如果lstm带了projection的话， 难么它的h呢， 实际上是要压缩的， 维度不再是h size；而是projection size， 所以 h0我们也要 改一下

projection的作用实际上就是对 h0进行一个压缩， 接下来查看模型参数

projection

```
1 # 定义常量
2 bs,T,i_size,h_size = 2,3,4,5
3 proj_size = 3
4 input = torch.randn(bs,T,i_size) # 输入序列
5 c0 = torch.randn(bs,h_size) # 初始值不需要训练
6 h0 = torch.randn(bs,proj_size)
```

```
1 # 调用官方LSTM API
2 lstm_layer = nn.LSTM(i_size,h_size,batch_first=True,proj_size = proj_size)
3 output,(h_finall,c_finall) = lstm_layer(input,(h0.unsqueeze(0),c0.unsqueeze(0)))
4
5 for k,v in lstm_layer.named_parameters():
6     print(k,v.shape)
```

```
weight_ih_l0 torch.Size([20, 4])
weight_hh_l0 torch.Size([20, 3])
bias_ih_l0 torch.Size([20])
bias_hh_l0 torch.Size([20])
weight_hr_l0 torch.Size([3, 5])
```

出现lstmP 网络参数的结果；相比于lstm；lstmP多了一个结果：weight_hr_l0

这个参数就是对 hidden state 进行一个压缩的

hidden state的大小 实际变成了3， 不再是5了， 我们打印output.shape和 h_final.shape、 c_final.shape

```
1 # 调用官方LSTM API
2 lstm_layer = nn.LSTM(i_size,h_size,batch_first=True,proj_size = proj_size)
3 output,(h_final,c_final) = lstm_layer(input,(h0.unsqueeze(0),c0.unsqueeze(0)))
4
5 print(output.shape,h_final.shape,c_final.shape)
6
7 for k,v in lstm_layer.named_parameters():
8     print(k,v.shape)
0.0s

rch.Size([2, 3, 3]) torch.Size([1, 2, 3]) torch.Size([1, 2, 5])
ight_ih_l0 torch.Size([20, 4])
ight_hh_l0 torch.Size([20, 3])
as_ih_l0 torch.Size([20])
as_hh_l0 torch.Size([20])
ight_hr_l0 torch.Size([3, 5])
```

可以看到`lstm`的`output shape`是 $2 \times 2 \times 3$ 的；不是 $2 \times 3 \times 5$ 的，就是因为我们现在对输出进行了压缩；然后`h final`和`c final`分别是 $1 \times 2 \times 3$ 和 $1 \times 2 \times 5$ 的；可以看到 `h final`的大小 也变成了 3，但是`c`的大小 仍然是5；就是我们只对输出 进行了压缩，不会对细胞状态 进行 压缩；

以上是projection的原理，接下来 我们改一下 自定义的函数；

我们多了一个projection参数，所以签名中加一个，w_hr 并且设置默认为None

```
def lstm_forward(input, initial_states, w_ih, w_hh, b_ih, b_hh, w_hr=None):
```

如果是None的话，我们认为就是一个普通的lstm，如果不是None就是带有projection的，这个带进来以后我们需要做哪些修改呢？首先output size做一个判断，如果有projection size，我们的output size就不是 h size了

```
if w_hr is not None:
```

我们就要判断，首先需要找到projection size，我们简写为p size，

```
p_size, _ = w_hr.shape[0]
```

它就是w_hr的第0维

```
weight_hr_l0 torch.Size([3, 5])
```

这个就是projection size；然后 output size，就是等于 p_size

```
output_size = p_size
```

如果 else的话，output size就等于 h size

```
else:  
    output_size = h_size
```

全部的代码：

```
if w_hr is not None:  
    p_size, _ = w_hr.shape[0]  
    output_size = p_size  
else:  
    [ output_size = h_size
```

以上是我们引入projection以后做的改变；另外 w_hr；同样要引入一个batch的维度类似上面的

```

prev_c = c0
batch_w_ih = w_ih.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, i_size]
batch_w_hh = w_hh.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, h_size]

if w_hr is not None:
    p_size, _ = w_hr.shape[0]
    output_size = p_size
    batch_w_hr =

```

同样引入batch的维度，但是这时候，形状就是 $bs \times p\ size \times h\ size$

```

prev_c = c0
batch_w_ih = w_ih.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, i_size]
batch_w_hh = w_hh.unsqueeze(0).tile(bs, 1, 1) # [bs, 4*h_size, h_size]

if w_hr is not None:
    p_size, _ = w_hr.shape[0]
    output_size = p_size
    batch_w_hr = w_hh.unsqueeze(0).tile(bs, 1, 1) # [bs, p_size, h_size]
else:
    output_size = h_size

```

这是对output size做的变更；就是因为引入了projection，所以output大小是变小了；那么接下来要变更哪里呢？

现在引入了projection之后，这里的hidden size是变小的，就是说已经变成了projection size

也就是说我们的 $w \times h$ prev大小仍然是： $bs \times 4$ 倍的hidden size，但它是怎么过来的呢？它是 $bs \times 4$ 倍的hidden size再乘以p size；再跟p size进行相乘；然后p size这个维度就消掉了

```
w_times_h_prev = torch.bmm(batch_w_hh, prev_h.unsqueeze(-1)) # [bs, 4*h_size, 1]
```

那如果引入了projection，我们需要，在这里得到的prev h这里，进行一个压缩

```

prev_c = i_t * prev_c + i_t * g_t
prev_h = o_t * torch.tanh(prev_c) #

```

现在 prev h这里，大小其实也就是 $bs \times h\ size$

但我们输出的h要是p size的，所以这里我们要进行一个压缩

```
prev_h = o_t * torch.tanh(prev_c) # [bs, h_size]
```

那同样，如果 $w\ hr$ 不是None的话，我们就要做projection，我们就要对 prev h进行一个压缩；压缩原理仍然是用，矩阵相乘的算法；就是说我们用压缩矩阵 $w\ hr$ ，跟prev h进行一个相乘就好了，但是我们需要对 prev h进行扩一维，最后一个维度扩一维

```
if w_hr is not None:# 做projection  
    prev_h = torch.bmm(batch_w_hr,prev_h.unsqueeze(-1))
```

这样 prev h的维度 就变成了 $bs \times p\ size \times 1$

我们把这个1，最后再去掉

```
if w_hr is not None:# 做projection  
    prev_h = torch.bmm(batch_w_hr,prev_h.unsqueeze(-1))  
    # [bs,p_size,1]  
    prev_h = prev_h.squeeze(-1) # bs x p_size
```

这样做好了projection；这个就是 lstm projection的原理，就是会对输出的状态 进行一个压缩，然后整个 output的维度就变小了；另外引入projection，整个计算量都是变小的，看这里：

```
w_times_h_prev = torch.bmm(batch_w_hh, prev_h.unsqueeze(-1)) # [bs, 4*h_size, 1]  
w times h prev = w times h prev.squeeze(-1) # [bs, 4*h_size]
```

这里以前是 batch size \times 4倍的hidden size，再乘以 hidden size，这个参数，现在 就变成了 4倍的hidden size \times p size；所以它的参数数目 是有降低的，运算量 是有降低的；

另外 这个 prev c的维度是没有变得，仍然是 hidden size维，但是prev h的维度 是降低的；

以上是 lstm forward 引入了 projection；接下来 继续测试，并且把 weight hr 传入进来，得到 带有projection的自定义函数

接下来 进行测试，结果是一致的；

lstmP其实比较简单 就是对 输出的状态，也就是prev h进行压缩 使得 整个LSTM网络，运算量 和 参数量 都有减小；主要是w hh 权重的维度是有降低的，想知道 减少了多少，自己查，运算量也是有减少的。ok done

LSTMP的全部代码：

projection

```
▷ 
1 # 定义常量
2 bs,T,i_size,h_size = 2,3,4,5
3 proj_size = 3
4 input = torch.randn(bs,T,i_size) # 输入序列
5 c0 = torch.randn(bs,h_size) # 初始值不需要训练
6 h0 = torch.randn(bs,proj_size)
[23] ✓ 0.0s
```

Python

◇ 生成 + 代码 + Markdown

```
1 # 调用官方LSTM API
2 lstm_layer = nn.LSTM(i_size,h_size,batch_first=True,proj_size = proj_size)
3 output,(h_finall,c_finall) = lstm_layer(input,(h0.unsqueeze(0),c0.unsqueeze(0)))
4
5 print(output.shape,h_finall.shape,c_finall.shape)
6
7 for k,v in lstm_layer.named_parameters():
8     print(k,v.shape)
[24] ✓ 0.0s
```

Python

```
... torch.Size([2, 3, 3]) torch.Size([1, 2, 3]) torch.Size([1, 2, 5])
weight_ih_l0 torch.Size([20, 4])
weight_hh_l0 torch.Size([20, 3])
bias_ih_l0 torch.Size([20])
bias_hh_l0 torch.Size([20])
weight_hr_l0 torch.Size([3, 5])
```

```
▷ 
1 # 自己写一个LSTM
2 def lstm_forward(input,initial_states,w_ih,w_hh,b_ih,b_hh,w_hr=None):
3     # 以上写好了 函数签名
4     h0,c0 = initial_states #初始状态
5     bs,T,i_size = input.shape
6     h_size = w_ih.shape[0] // 4
7
8     prev_h = h0
9     prev_c = c0
10    (variable) batch_w_hh: Any 0).tile(bs,1,1)
11    batch_w_nn = w_nn.unsqueeze(0).tile(bs,1,1)
12
13    if w_hr is not None:
14        p_size = w_hr.shape[0]
15        output_size = p_size
16        batch_w_hr = w_hr.unsqueeze(0).tile(bs,1,1) # [bs,p_size,h_size]
17    else:
18        output_size = h_size
19
20    output = torch.zeros(bs,T,output_size) # 输出序列
21
```

```

22     for t in range(T):
23         x = input[:,t,:]
24
25         w_times_x = torch.bmm(batch_w_ih,x.unsqueeze(-1)) #[bs,4*h_size,1]
26         w_times_x = w_times_x.squeeze(-1) #[bs,4*h_size]
27
28         w_times_h_prev = torch.bmm(batch_w_hh,prev_h.unsqueeze(-1)) #[bs,4*h_size,1]
29         w_times_h_prev = w_times_h_prev.squeeze(-1) #[bs,4*h_size]
30
31         # 分别计算 输入门(i), 遗忘门(f), cell门(g), 输出门(o)
32         i_t = torch.sigmoid(w_times_x[:, :h_size] + w_times_h_prev[:, :h_size] + b_ih[:h_size] + b_hh[:h_size] +
33         f_t = torch.sigmoid(w_times_x[:, h_size:2*h_size] + w_times_h_prev[:, h_size:2*h_size] +
34         |   |   |   |   b_ih[h_size:2*h_size] + b_hh[h_size:2*h_size])
35         g_t = torch.tanh(w_times_x[:, 2*h_size:3*h_size] + w_times_h_prev[:, 2*h_size:3*h_size] +
36         |   |   |   |   b_ih[2*h_size:3*h_size] + b_hh[2*h_size:3*h_size])
37         o_t = torch.sigmoid(w_times_x[:, 3*h_size:4*h_size] + w_times_h_prev[:, 3*h_size:4*h_size] +
38         |   |   |   |   b_ih[3*h_size:4*h_size] + b_hh[3*h_size:4*h_size])
39
40
41         prev_c = f_t * prev_c + i_t * g_t
42         prev_h = o_t * torch.tanh(prev_c)
43

```

```

43
44     if w_hr is not None: # 做projection
45         prev_h = torch.bmm(batch_w_hr,prev_h.unsqueeze(-1)) # [bs,p_size,1]
46         prev_h = prev_h.squeeze(-1) # bsx p_size
47
48
49     output[:,t,:] = prev_h
50
51     return output,(prev_h,prev_c)
52
53 output_custom,(h_finall_custom,c_finall_custom) = lstm_forward(input,(h0,c0),lstm_layer.weight_ih_l0,
54 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
55 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
56 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
57 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

```

✓ 0.0s

Python

```

1 print(torch.allclose(output,output_custom))
2 print(torch.allclose(h_finall,h_finall_custom))
3 print(torch.allclose(c_finall,c_finall_custom))

```

✓ 0.0s

Python