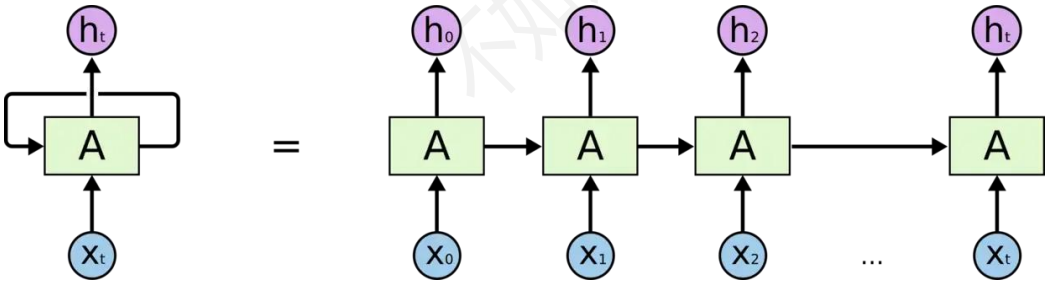
1 RNN 的缺陷——长期依赖的问题 （The Problem of Long-Term Dependencies）

前面一节我们学习了 RNN 神经网络，它可以用来处理序列型的数据，比如一段文字，视频等等。RNN 网络的基本单元如下图所示，可以将前面的状态作为当前状态的输入。



但也有一些情况，我们需要更“长期”的上下文信息。比如预测最后一个单词“我在中国长大……我说一口流利的**。”“短期”的信息显示，下一个单词很可能是一种语言的名字，但如果我们想缩小范围，我们需要更长期语境——“我在中国长大”，但这个相关信息与需要它的点之间的距离完全有可能变得非常大。

不幸的是，随着这种距离的扩大，RNN 无法学会连接这些信息。

从理论上讲，RNN 绝对有能力处理这种“长期依赖性”。人们可以为他们精心选择参数，以解决这种形式的问题。遗憾的是，在实践中，RNN 似乎无法学习它们。

幸运的是，LSTM 没有这个问题！

2、LSTM 神经网络

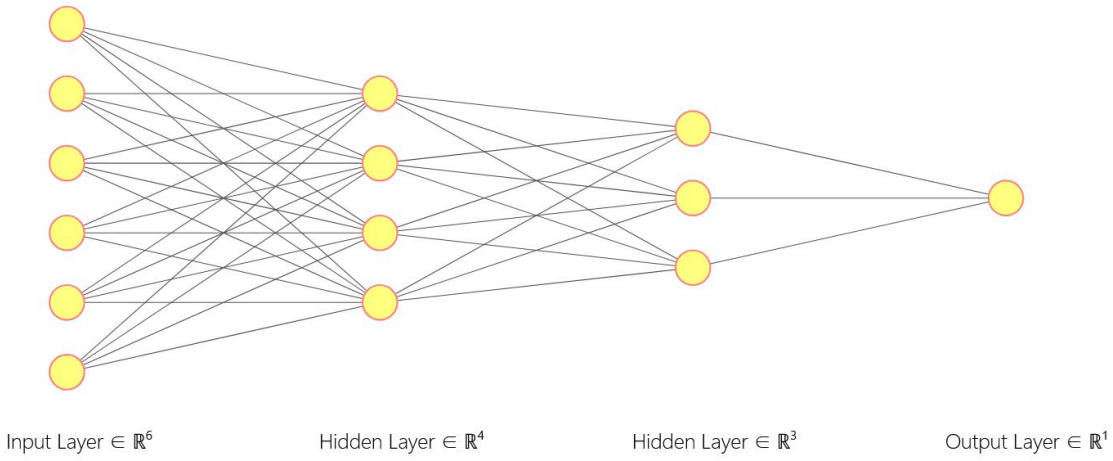
长期短期记忆网络通常被称为“LSTM”，是一种特殊的 RNN，能够学习长期依赖关系。它们由 Hochreiter & Schmidhuber(1997)引入，并在随后的工作中被许多人提炼和推广。

LSTM 的设计就是为了避免长期依赖问题，在各种各样的问题上都做得非常好，现在被广泛使用，并取得了令人难以置信的成功:语音识别、语言建模、翻译、图像字幕等等。

2.1 总体结构框架

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

多层感知机（线性连接层）



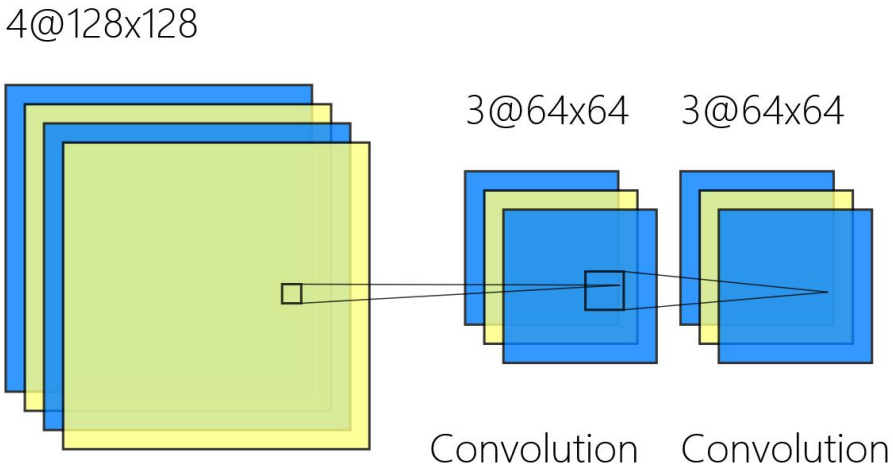
从特征角度考虑：

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

隐藏层：然后根据隐藏层神经元的数量 m 将前层输入的特征用 m*1 的单维向量进行表示（对特征进行了提取变换，隐藏层的数据特征），单个隐藏层的神经元数量就代表网络参数，可以设置多个隐藏层；

输出特征：最终根据输出层的神经元数量 y 输出 y*1 的单维向量。

卷积神经网络结构



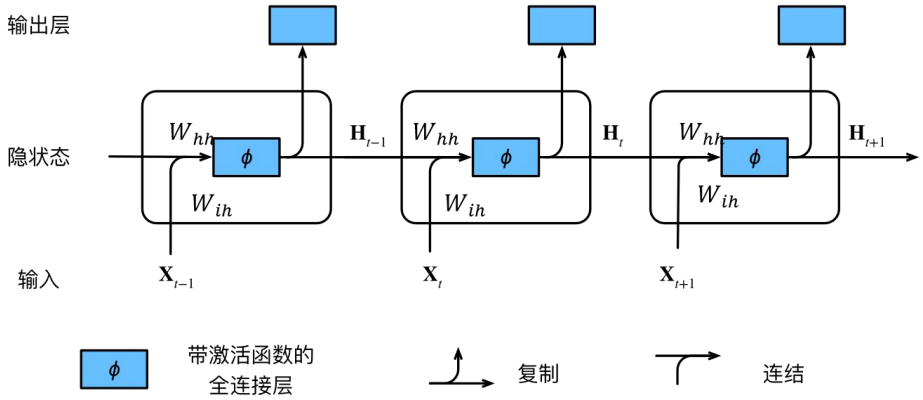
从特征角度考虑：

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

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

输出特征：根据场景的需要设置后面的输出，可以是多分类的单维向量等等。

循环神经网络 RNN 结构



从特征角度考虑：

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

我们来详细对比一下卷积神经网络的输入特征，

(batch)*T_seq*feature_size

(batch)*channel*width*height，

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

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

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

隐藏层：明确了输入特征之后，我们再来看看隐藏层代表着什么。隐藏层有 T_seq 个隐状态 H_t(和输入序列长度相同)，每个隐状态 H_t 类似于一个 channel，对应着 T_seq 中的 t 时刻的输入特征；而每个隐状态 H_t 是用 hidden_size*1 的单维向量表示的，所以一个隐含层是 T_seq*hidden_size 的张量；对应时刻 t 的输入特征由 feature_size*1 变为 hidden_size*1 的向量。如图中所示，同一个隐含层不同时刻的参数 W_ih 和 W_hh 是共享的；隐藏层可以有 num_layers 个（图中只有 1 个）

以 t 时刻具体阐述一下：

X_t 是 t 时刻的输入，是一个 feature_size*1 的向量

W_ih 是输入层到隐藏层的权重矩阵

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

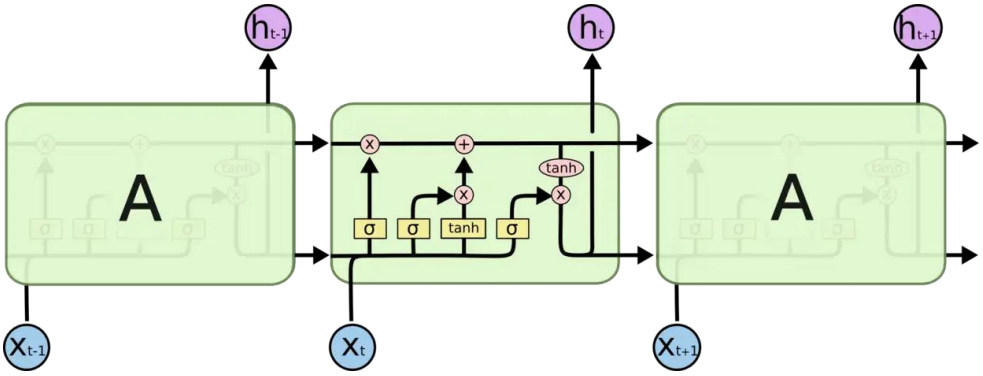
W_hh 是上一时刻的隐藏层的值传入到下一时刻的隐藏层时的权重矩阵

Ot 是 t 时刻 RNN 网络的输出

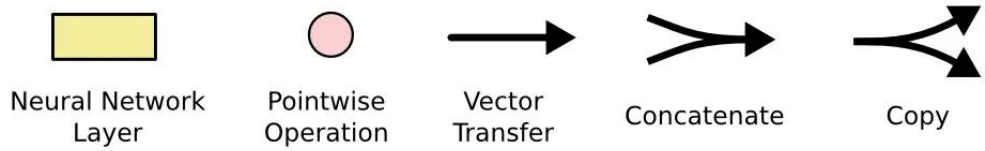
从上右图中可以看出这个 RNN 网络在 t 时刻接受了输入 Xt 之后，隐藏层的值是 St，输出的值是 Ot。但是从结构图中我们可以发现 St 并不单单只是由 Xt 决定，还与 t-1 时刻的隐藏层的值 St-1 有关。

LSTM 与 RNN 的区别与联系

LSTM 在总的框架上基本同 RNN 一致，也是链状的，不同之处即在于从输入特征到隐状态的计算方式是不同的，不是只有一个神经网络层，而是有四个以非常特殊的方式进行交互的层，下面将详细介绍如何计算的。



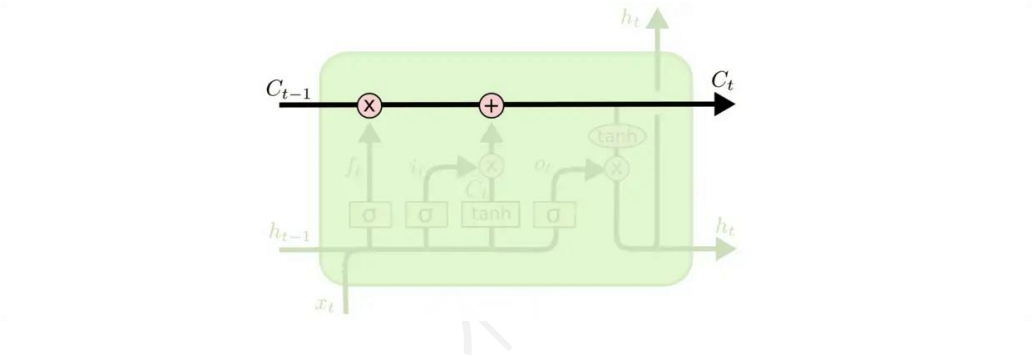
对于图中的符号：



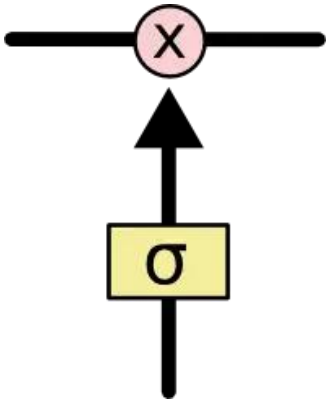
在上图中，每条线都承载着整个矢量，从一个节点的输出到另一节点的输入。粉色圆圈表示逐点操作，例如矢量加法，而黄色框表示学习的神经网络层。 合并的行表示串联，而分叉的行表示要复制的内容，并且副本将到达不同的位置。

2.2 LSTM 背后的核心思想 （The Core Idea Behind LSTMs）

LSTM 的关键是单元状态（cell state），水平线贯穿图的顶部。通过利用三个门来选择性地删除或向单元状态添加信息。单元状态有点像传送带，它沿整个链条一直沿直线延伸，只有一些较小的线性相互作用，信息不加改变地流动非常容易。



门是一种选择性地让信息通过的方式。 它们由一个含有 sigmoid 激活函数的网络层和点乘操作组成。



sigmoid layer 输出介于 0 和 1 之间的数字，描述每个组件应允许通过多少。

值为 0 表示“不让任何内容通过”，

值为 1 表示“让所有内容通过！”

LSTM 具有三个这样的门，以保护和控制单元状态。

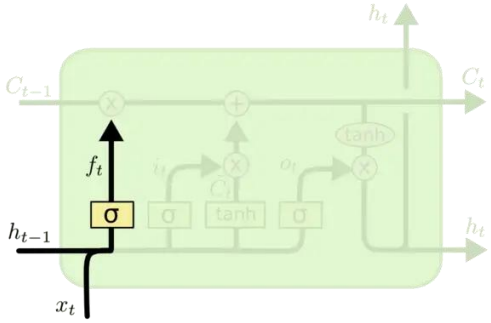
2.3 LSTM 详细步骤分析

（1）明确输入

首先明确 LSTM 的输入包括 xt 和 ht-1，xt 是当前时刻的输入，是(batch)*1*feature_size 的张量；ht-1 是上一时刻的隐状态，(batch)*1*hidden_size 的张量。

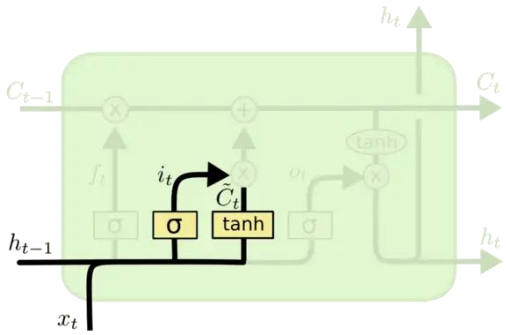
（2）分别计算 3 个门，进一步控制往单元状态删除或添加信息。

第一个是“遗忘门”（forget gate layer）：决定要从上一单元状态中丢弃哪些信息。其根据当前时刻的输入 xt 和上一时刻的隐状态 ht-1 输出介于 0 和 1 之间的数字，并决定上一单元状态 Ct-1 传到当前单元状态 Ct 的比例，1 代表“完全保留 Ct-1”，0 代表“完全删除 Ct-1”。如下：



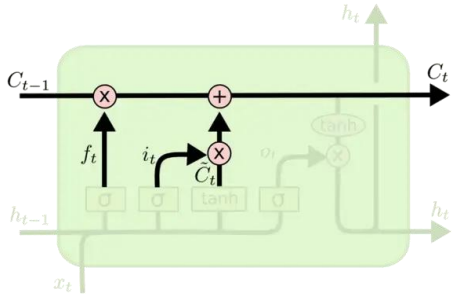
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

第二个是“输入门”：决定要在当前单元状态存储哪些新信息。这包括两个部分。首先，称为“输入门”的 sigmoid 层决定了我们将更新的比例。这一步的输入仍是当前时刻的输入 xt 和上一时刻的隐状态 ht-1，输出介于 0 和 1 之间的数字；接下来是 tanh 层创建一个新候选值 $\tilde{C}_t$，的向量，这个是实际要添加到当前单元状态的，将两者结合起来即是当前单元状态的最终更新方法。



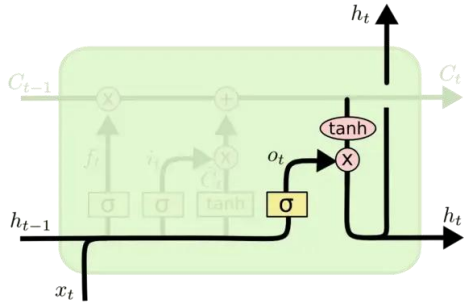
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] \; + \; b_C\right)$$

第一步是前一单元状态遗忘多少（换句话说还留到当前单元状态多少），第二步是决定当前单元状态新增多少信息，接下来就是融合更新最终的当前单元状态了。即将旧状态 Ct-1 乘以 ft，然后加上 $i_t * \tilde{C}_t$。



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

第三个是“输出门”，有了当前单元状态，我们还需要知道怎么由当前单元状态输出得到当前隐状态，同样地，使用一个 sigmoid 层，输入仍是当前时刻的输入 xt 和上一时刻的隐状态 ht-1，输出介于 0 和 1 之间的数字；然后通过 tanh 放置单元状态（将值推到-1 和 1 之间），然后将其乘以输出门的数值，得到隐状态。



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

3 代码

model.py

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

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

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

    # 这里比 RNN 多了一个参数，c_prev，这也是 LSTM 的核心单元状态，具体参照原理讲解
    def forward(self,x,hidden_prev,c_prev):
        # 每一次调用 rnn 层返回的就是输出层和隐状态值和单元状态，隐状态和单元状态又是下一循环的上一状态值，所以用 hidden_prev&c_prev 表示
        out,(hidden_prev,c_prev)=self.lstm(x,(hidden_prev,c_prev))
        print("out1&hidden_prev.shape",out.shape,hidden_prev.shape)
        #view()相当于 reshape、resize，重新调整 PyTorch 中的 Tensor 形状，若非 Tensor 类型，可使用 data = torch.tensor(data)来进行转换。
        #out=out.view(-1,self.hidden_size)
        print("out2.shape", out.shape)
        out=self.linear(out)
        print("out3.shape", out.shape)
       # out=out.unsqueeze(0)
        print("out4.shape", out.shape)
        #输出的维度是 batch_size*T_seq*hidden_size
        return out,(hidden_prev,c_prev)


#自己实现一个 LSTMN 函数
#这里的函数参数需要手动给定网络结构参数，
def My_LSTM(input,initial_states,w_ih,w_hh,b_ih,b_hh):
    #比 RNN 多了一个初始状态 c0
```

```
    h0,c0=initial_states
    batch_size,T_seq,feature_size=input.shape
    hidden_size=w_ih.shape[0]//4

    prev_h=h0
    prev_c=c0
    batch_w_ih=w_ih.unsqueeze(0).tile(batch_size,1,1)
    batch_w_hh=w_hh.unsqueeze(0).tile(batch_size,1,1)

    output_feature_size=hidden_size
    output=torch.zeros(batch_size,T_seq,output_feature_size)

    for t in range(T_seq):
        #当前时刻的输入向量，(batch_size*feature_size)
        x=input[:,t,:]
        #计算两个 tensor 的矩阵乘法，torch.bmm(a,b),tensor a  的 size 为(b,h,w),tensor b 的 size 为(b,w,m)
        # 也就是说两个 tensor 的第一维是相等的，然后第一个数组的第三维和第二个数组的第二维度要求一样，其实就是第一维不变，后面二维张量相乘，h*w*w*m=h*m
        # 对于剩下的则不做要求，输出维度 （b,h,m）
        # batch_w_ih=batch_size*(4*hidden_size)*feature_size
        #x=batch_size*feature_size*1
        #w_times_x=batch_size*(4*hidden_size)*1
        ##squeeze，在给定维度（维度值必须为 1）上压缩维度，负数代表从后开始数
        w_times_x=torch.bmm(batch_w_ih,x.unsqueeze(-1))
        w_times_x=w_times_x.squeeze(-1)
       # print(batch_w_ih.shape, x.shape)
        # batch_w_hh=batch_size*(4*hidden_size)*hidden_size
        # prev_h=batch_size*hidden_size*1
        # w_times_h_prev=batch_size*(4*hidden_size)*1
       # print(batch_w_hh.shape,prev_h.shape)
        w_times_h_prev=torch.bmm(batch_w_hh,prev_h.unsqueeze(-1))
        w_times_h_prev=w_times_h_prev.squeeze(-1)

        #分别计算输入门(i),遗忘门(f)，cell 门(g)，输出门(o)，这里可以看到参数是共享的
        i_t=torch.sigmoid(w_times_x[:,:hidden_size]+b_ih[:hidden_size]+
                          w_times_h_prev[:,:hidden_size]+b_hh[:hidden_size])
        f_t = torch.sigmoid(w_times_x[:, hidden_size:2 * hidden_size] +
b_ih[hidden_size:2 * hidden_size] +
                            w_times_h_prev[:, hidden_size:2 * hidden_size] +
b_hh[hidden_size:2 * hidden_size])
```

```
            g_t = torch.sigmoid(w_times_x[:, 2*hidden_size:3 * hidden_size] +
b_ih[2*hidden_size:3 * hidden_size] +
                                w_times_h_prev[:, 2*hidden_size:3 * hidden_size]
+ b_hh[2*hidden_size:3 * hidden_size])
            o_t = torch.sigmoid(w_times_x[:, 3 * hidden_size:4 * hidden_size] + b_ih[3
* hidden_size:4 * hidden_size] +
                                w_times_h_prev[:, 3 * hidden_size:4 * hidden_size]
+ b_hh[3 * hidden_size:4 * hidden_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)



# 测试代码
            # 每个 python 模块（python 文件）都包含内置的变量 __name__，当该
模块被直接执行的时候，__name__ 等于文件名（包含后缀 .py ）
            # 如果该模块 import 到其他模块中，则该模块的 __name__ 等于模块
名称（不包含后缀.py）
            #“__main__”始终指当前执行模块的名称（包含后缀.py）
            # if 确保只有单独运行该模块时，此表达式才成立，才可以进入此判断
语法，执行其中的测试代码，反之不行

if __name__=="__main__":
        batch_size=2
        T_seq=5
        feature_size=8
        hidden_size=6

        input=torch.randn(batch_size,T_seq,feature_size)
        c0=torch.randn(batch_size,hidden_size)
        h0=torch.randn(batch_size,hidden_size)

        lstm_layer=nn.LSTM(feature_size,hidden_size,batch_first=True)
        output,(h_final,c_final)=lstm_layer(input,(h0.unsqueeze(0),c0.unsqueeze(0)))
        print(output,(h_final,c_final))
        #.named_parameters()遍历得到网络参数
        for k,v in lstm_layer.named_parameters():
            print(k,v.shape)
```

```
my_output,(my_h_final,my_c_final)=My_LSTM(input,(h0,c0),lstm_layer.weight_ih_
l0,lstm_layer.weight_hh_l0,

lstm_layer.bias_ih_l0,lstm_layer.bias_hh_l0)

    print(my_output,(my_h_final,my_c_final))
```

train.py

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

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

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

hidden_size=3#隐含层维度
output_size=2#输出层维度

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

def train(input):
    model=LSTM(feature_size,hidden_size,num_layers,output_size)
    print("model:\n",model)
```

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

        output,(hidden_prev,c_prev)=model(x,hidden_prev,c_prev)
        print("output_size:",output.shape)
        y = torch.randn(batch_size,T_seq,output_size)
        print("y:", y.shape)
        #返回一个新的 tensor，从当前计算图中分离下来的，但是仍指向原变量的存放位置,
        # 不同之处只是 requires_grad 为 false，得到的这个 tensor 永远不需要计算其梯度，不具有 grad。
        hidden_prev=hidden_prev.detach()
        c_prev = c_prev.detach()

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

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

    return hidden_prev,c_prev, model
```

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

参考资料

https://colah.github.io/posts/2015-08-Understanding-LSTMs/
https://zh-v2.d2l.ai/chapter_recurrent-neural-networks/rnn.html
https://www.bilibili.com/video/BV1zq4y1m7aH/?spm_id_from=333.788&vd_source=cf7630d31a6ad93edecfb6c5d361c659