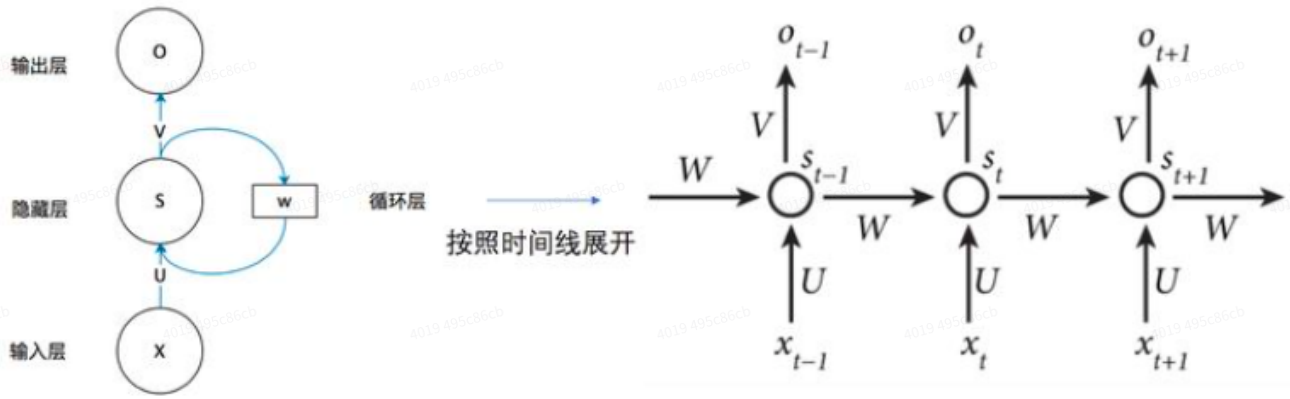
# 3.RNN& LSTM& GRU

## 1.RNN

循环神经网络(Recurrent Neural network, RNN) 是一种用于**处理序列数据**的神经网络。相比与一般的神经网络，其很适合用于处理序列变化的数据。

### 1.1RNN的基本结构

由于本质是处理序列数据（一般按时间顺序，也有可能按照文本顺序）。其基本结构如下：




RNN时间线展开图

现在看上去就比较清楚了，这个网络在t时刻接收到输入xt之后，**隐藏层的值（这个时刻下对应的状态值）是St , 输出值是Ot**。

$$S_t = f(U \cdot X_t + W \cdot S_{t-1})$$

$$O_t = g(V \cdot S_t)$$

从公式中，可以看出，St的值不仅仅取决于xt,也取决于St-1。

同时，每个神经元都会接收上一个神经元的输出（其实这些神经元**都是相同的，只有三个参数矩阵（U，W, V），每一层的参数相同，只是使用的状态不一样**）。神经元的输出重新作为输入，因此将其称为循环神经网络。

### 1.2RNN的问题

1. **只考虑了短期因素，没有考虑长期因素，因此不适合长序列**。

一般的RNN，由于梯度弥散（消失），导致在序列很长的时候，无法在较后的时间步中，按照梯度更新较前时间步的W，导致无法根据后项序列来修改前向序列的参数，使得前向序列无法很好的做特征提取，使得在长时间过后，模型将无法再次获取有效的前向序列的记忆信息。

这一特点：导致了RNN不具备长期记忆的特点，只拥有短期记忆。

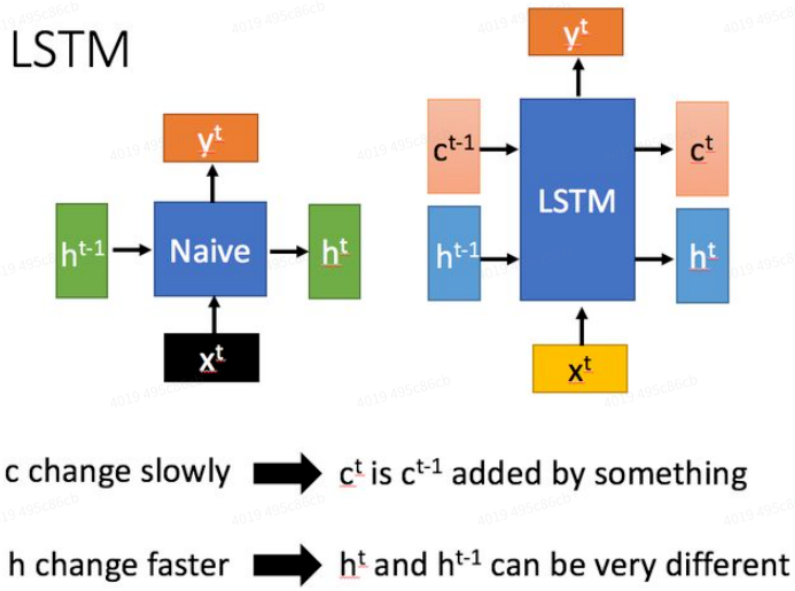
2. **长序列训练过程中存在梯度消失和梯度爆炸问题。**

梯度消失：为此便提出了LSTM，GRU等结构变种，来解决RNN短期记忆的问题。

梯度爆炸：加入梯度裁剪即可有一定缓解。

# 2.LSTM的结构详解

## 2.1粗对比



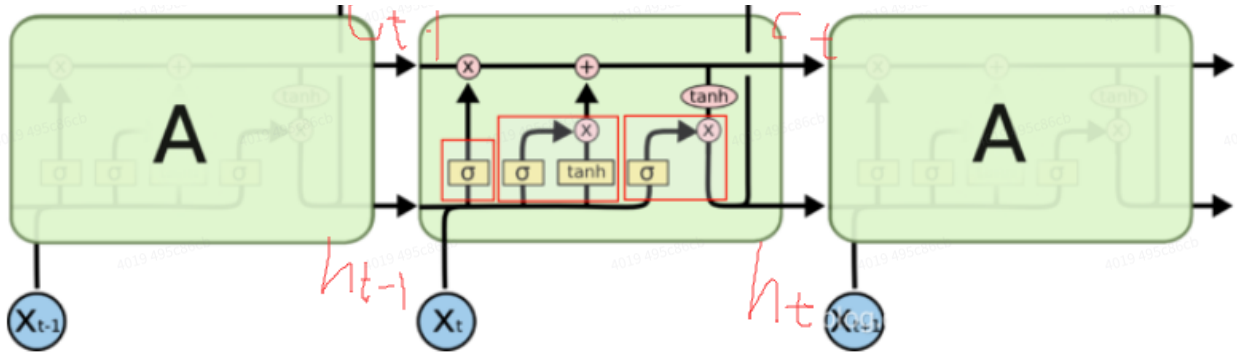

相比RNN只有一个传递状态ht,LSTM有两个传输状态，一个Ct(cell state)和一个ht(hidden state)。Tips,RNN中的ht相对于LSTM中的Ct。

其中对于传递下去的ct改变的很慢，通常输出的Ct是上一个状态传过来的C(t-1)加上一些数值。

ht在不同节点下往往会有很大的区别。

## 2.2LSTM详解

为了解决梯度消失和爆炸以及更好的预测和分类序列数据等问题，rnn逐渐转变为lstm。




$$
\begin{aligned}
i^{(t)} &= \sigma(W^{(i)}x^{(t)} + U^{(i)}h^{(t-1)}) && \text{(Input gate)} \\
f^{(t)} &= \sigma(W^{(f)}x^{(t)} + U^{(f)}h^{(t-1)}) && \text{(Forget gate)} \\
o^{(t)} &= \sigma(W^{(o)}x^{(t)} + U^{(o)}h^{(t-1)}) && \text{(Output/Exposure gate)} \\
\tilde{c}^{(t)} &= \tanh(W^{(c)}x^{(t)} + U^{(c)}h^{(t-1)}) && \text{(New memory cell)} \\
c^{(t)} &= f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)} && \text{(Final memory cell)} \\
h^{(t)} &= o^{(t)} \circ \tanh(c^{(t)})
\end{aligned}
$$



看的不是特别懂，下面就来逐一分析。

## 2.2.1遗忘门（第一个框）

**这个阶段主要是对上一个节点传进来的输入进行选择性忘记**。简单来说就是会 “忘记不重要的，记住重要的”。具体来说是通过计算得到的f(t)表示forget来作为忘记门控，来控制上一个状态的 C(t－1) 哪些需要留, 哪些需要忘。(第二个公式)

## 2.2.2输入门（第二个框）

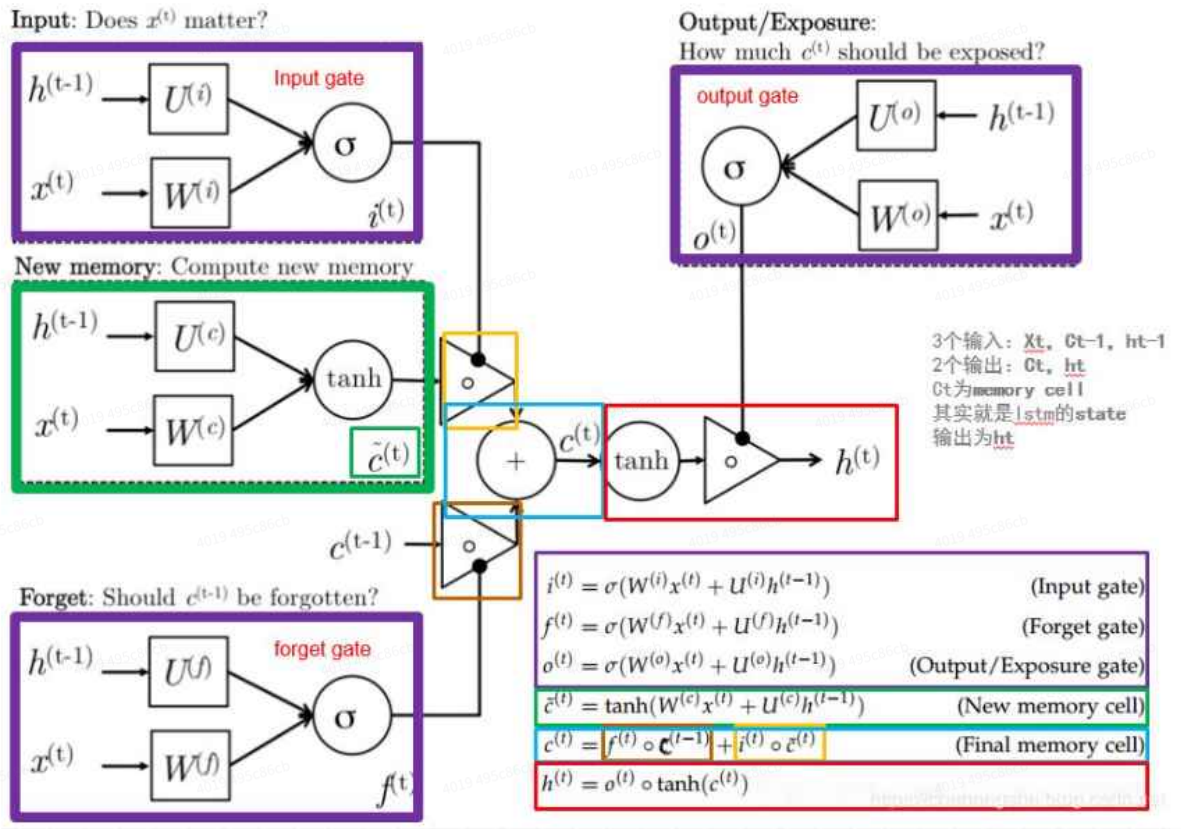
**这个阶段将这个阶段的输入有选择性地进行“记忆”**。主要是会对输入x(t)进行选择记忆。哪些重要则着重记录下来，哪些不重要，则少记一些。而选择的门控信号则是由 i代表(information)来进行控制。(第一个公式)

## 2.2.3输出门（第三个框）

**这个阶段将决定哪些将会被当成当前状态的输出。**

主要是通过o(t)来进行控制的(第三个公式)。并且还对上一阶段得到c的进行了放缩（通过一个tanh激活函数进行变化）。（第四个公式）

## 2.3LSTM各部分展开图




1. 首先输入为三个值，一个是此刻的输入值xt，另一个是上一时刻的状态值c(t-1)，最后一个是上一个单元的输出h(t-1)。
2. 最终输出为两个值，一个是此刻产生的状态值ct和输出ht.

3. 首先是输入值x和上一个单元的输出h，分别两个输入都有对应的权重，在经过sigmoid激活作用下得到0-1的值，也就是三个门值。**（得到紫色框中的三个门值）**

$$
\begin{aligned}
i^{(t)} &= \sigma(W^{(i)}x^{(t)} + U^{(i)}h^{(t-1)}) && \text{(Input gate)} \\
f^{(t)} &= \sigma(W^{(f)}x^{(t)} + U^{(f)}h^{(t-1)}) && \text{(Forget gate)} \\
o^{(t)} &= \sigma(W^{(o)}x^{(t)} + U^{(o)}h^{(t-1)}) && \text{(Output/Exposure gate)}
\end{aligned}
$$

4. 和3差不多，依然还是 输入值x和上一个单元的输出h，两个值有对应的权重和3中的描述一模一样，**唯一的区别在于有一个tanh激活函数**，最后相当于得到此时输入得到的当前state，也就是new memory。

这里可以理解为输入其实是近似的x和h的concatenate操作，经过正常的神经网络的权重，最后经过tanh激活函数得到此时输入的当前的state，**x相当于此刻的输入，h为前面历史的输入，合在一起就是整个序列的信息，也就是此时的new memory**。

$$\tilde{c}^{(t)} = \tanh(W^{(c)}x^{(t)} + U^{(c)}h^{(t-1)}) \qquad \text{(New memory cell)}$$

5. 最后输出的state，也就是final memory的计算利用了input gate和forget gate，output gate只与输出有关。

   final memory的计算自然而然和**上一步算得此时的记忆state相关**并且和**上一个输出的final memory相关**，故为忘记门和Ct-1的乘积加上上一步算出来的此时单元的C和输入门的乘积为最终的state。

   **忘记门和Ct-1的乘积：选择遗忘哪一些之前的信息**

   **单元的C和输入门的乘积：选择保留当前状态的那些信息**

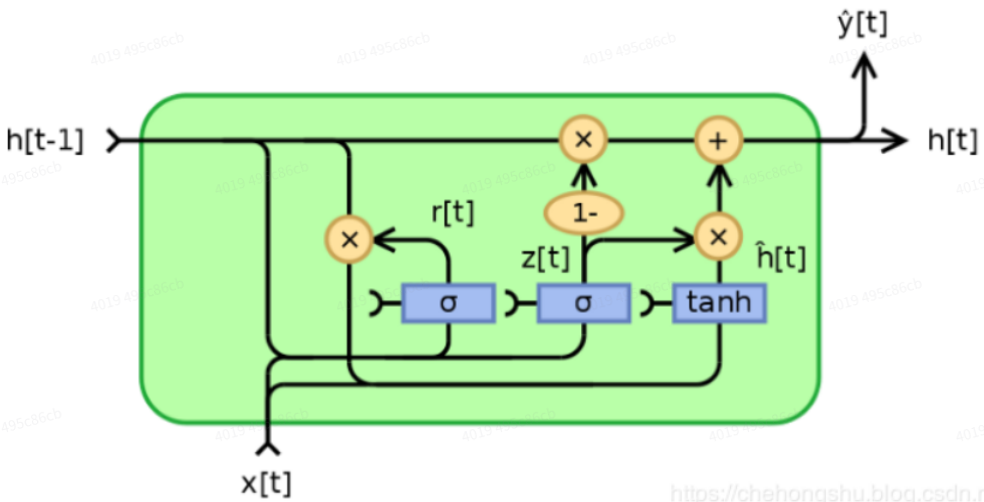
$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)} \qquad \text{(Final memory cell)}$$

6. 输出门只与输出相关，最终的输出h为输出门乘以tanh（c）

$$h^{(t)} = o^{(t)} \circ \tanh(c^{(t)})$$

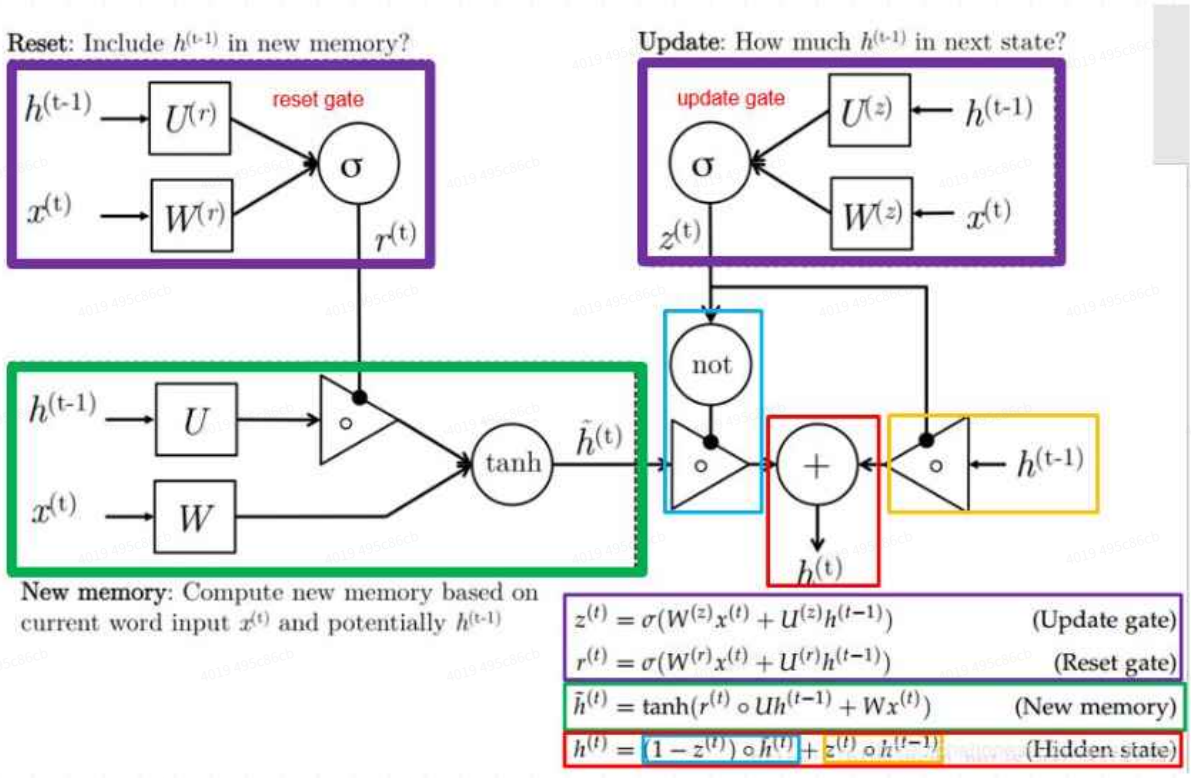
# 3.GRU

因为LSTM的训练比较慢，而GRU在其上稍微修改，速度可以快很多，而精度基本不变，所以GRU也十分流行



$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \\
\tilde{h}_t &= \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
\end{aligned}
$$

换个图看看：




1. 这里GRU只有两个gate，一个是reset gate， 一个是update gate，

**update gate的作用类似于input gate和forget gate，**

**(1-z)相当于input gate， z相当于forget gate。**

2. 输入为两个值，输出也为一个值，输入为输入此时时刻值x和上一个时刻的输出ht-1， 输出这个时刻的输出值ht。
3. 首先依然是利用xt和ht-1经过权重相乘通过sigmoid，得到两个0-1的值，即两个门值。

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

4. 接下来这里有一些不同，并且经常容易搞混淆。对于LSTM来说依然还是xt与ht-1分别权重相乘相加，之后经过tanh函数为此时的new memory。

   而GRU为在这个计算过程中，在ht-1与权重乘积之后和reset gate相乘，之后最终得到new memory，**这里的reset gate的作用为让这个new memory包括之前的ht-1的信息的多少。**

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

5. 接下来和lstm得到final memory其实一样，只是GRU只有两个输入，一个输出，其实这里h即输出也是state，就是说GRU的输出和state是一个值，所以4步骤得到的是new_h，这步骤得到的是final_h，通过update gate得到。

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

# 4.GRU与LSTM之间的比较

## 4.1 结构上

1. lstm为三个输入xt，ht-1， ct-1，两个输出。gru为两个输入xt， ht-1，一个输出ht，输出即state。
2. lstm有三个门，输入输出忘记门。gru有两个门，reset，update 门。
3. update 类似于 input gate和forget gate.

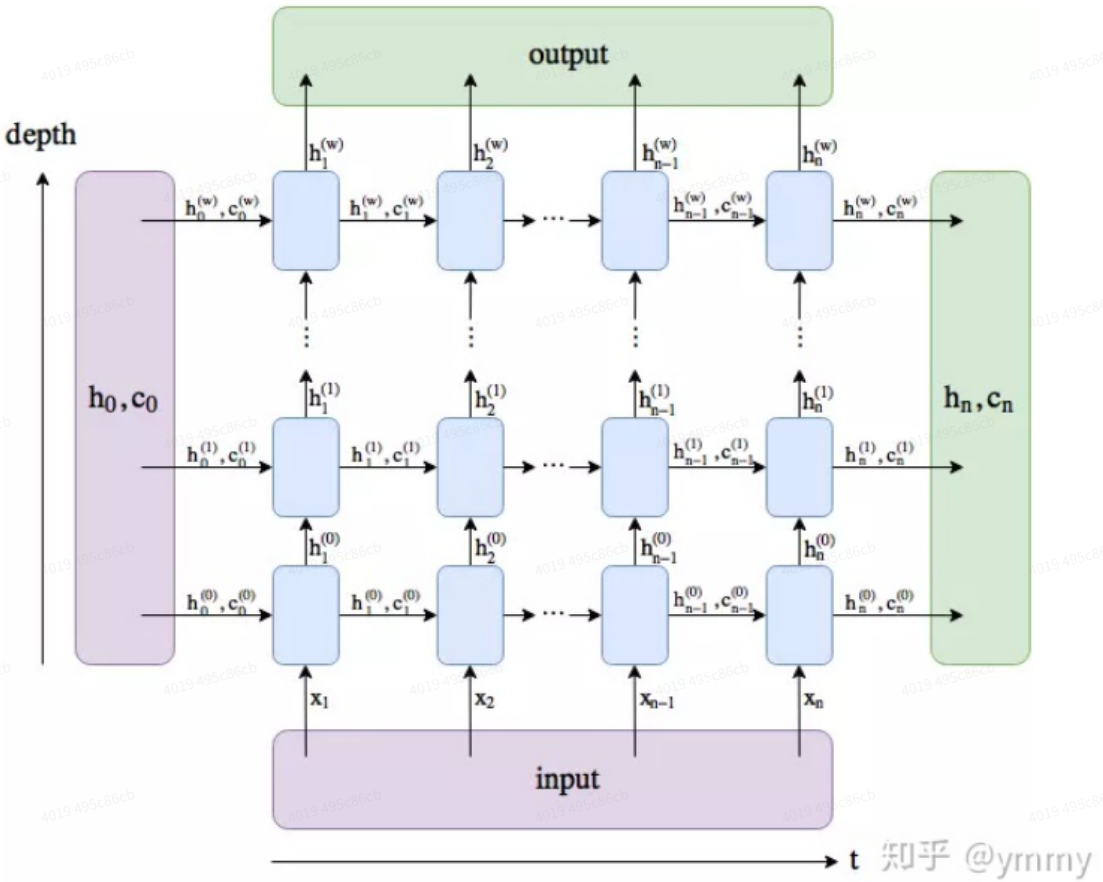
## 4.2 功能上

4. GRU参数更少，训练速度更快，**相比之下需要的数据量更少。**
5. **如果有足够的数据，LSTM的效果可能好于GRU。**

# 4.LSTM实操

## 1.LSTM网络结构与pytorch

LSTM中将整个网络看成这样：




蓝色的模块是指前面的单位LSTM结构。横向连接的各个LSTM结构就是序列化。不同状态（时间）下的LSTM结构。纵向连接的是同一状态下LSTM结构。或者我们可以简单看作是多个LSTM细胞结构串联，其个数为num_layers, 将其视为一个新的LSTM结构，并且按照时序连接起来。图中每个参数都是自带维度的。

Pytorch中的库：

**nn.LSTM(input_size, hidden_size, num_layers)**

input_size：输入的X的维度。

hidden_size: 输出和输入的hi 的维度。

num_layers: depth，即串联的LSTM个数。

# 2.时间序列实战

## 2.1. 数据

数据集，是从1949-1960,12年，每个月的乘客数量。即一共有144条数据，表示了144个月的乘客数量。

| | year | month | passengers |
|---|---|---|---|
| 0 | 1949 | January | 112 |
| 1 | 1949 | February | 118 |
| 2 | 1949 | March | 132 |
| 3 | 1949 | April | 129 |
| 4 | 1949 | May | 121 |
| 139 | 1960 | August | 606 |
| 140 | 1960 | September | 508 |
| 141 | 1960 | October | 461 |
| 142 | 1960 | November | 390 |
| 143 | 1960 | December | 432 |

## 2.2 思路

采用的肯定是利用前几期的数据来预测当前期的数据。(具体的方法是：利用前N（N= 3）期的数据为输入，当前期的数据为标签计算误差)。个人觉得可以N是一个超参，可以慢慢调.

由于时间序列数据每个时期的数据样本只有一个，那么X为(time_step, 1, input_size), Y为(time_step, 1, output_size)。

构建一个LSTM网络，输入的数据就是X，hidden_size可任意取，num_layers也可视情况取，即用多少层LSTM串联(同一时期内）。

最后用一层线性层Linear(hidden_size, output_size)进行输出。比较输出与Y的误差。不断迭代对参数进行优化。

## 2.3代码

Python

```python
#step 1. 加载飞行数据
flight_data = pd.read_csv('flights.csv')
# 数据归一化
maxPassenger = flight_data['passengers'].max()
minPassenger = flight_data['passengers'].min()
flight_data['passengers'] = (flight_data['passengers'] - minPassenger) \
     / (maxPassenger - minPassenger)

dataset = flight_data['passengers'].values.tolist()

#step 2. 划分数据集
# 数据集合目标函数值赋值，其中dateset为数据，look_back为以几行数据为特征数目
# look_back表示3期回头，即使用前三期的数据预测下一期
# 用前3期数据预测下1期
def createDataset(dataset, look_back):
    dataX = []
    dataY = []
    for i in range(len(dataset)-look_back):
        dataX.append(dataset[i:i+look_back])
        dataY.append(dataset[i+look_back])

    dataX = torch.tensor(dataX)
    dataX = dataX.reshape(-1, 1, look_back)
    dataY = torch.tensor(dataY)
    dataY = dataY.reshape(-1, 1, 1)
    return dataX, dataY

data = createDataset(dataset=dataset, look_back=3)    # 划分数据集,3个月为一组

# step3. 划分训练集和测试集
# 由于是时间序列数据，不适合这样随机打乱
def splitData(data,rate=0.7): #默认是0.7的训练集，0.2的测试集
    # 默认训练集比例为0.7
    dataX, dataY = data
    nSample = dataX.shape[0]
    nTrain = int(nSample*rate)
    trainData = (dataX[:nTrain], dataY[:nTrain])
    testData = (dataX[nTrain:], dataY[nTrain:])
    return trainData, testData


# 获取训练集和测试集，用80%的数据来训练拟合，20%的数据来预测
rate = 0.8
trainData, testData = splitData(data, rate=rate)
```

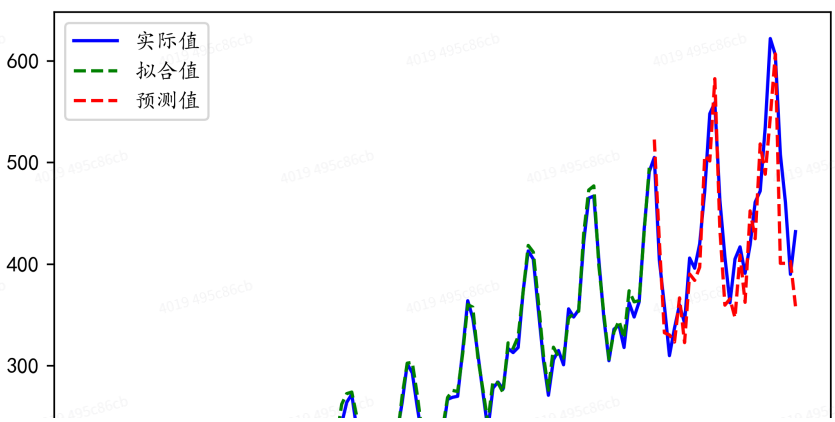
Python

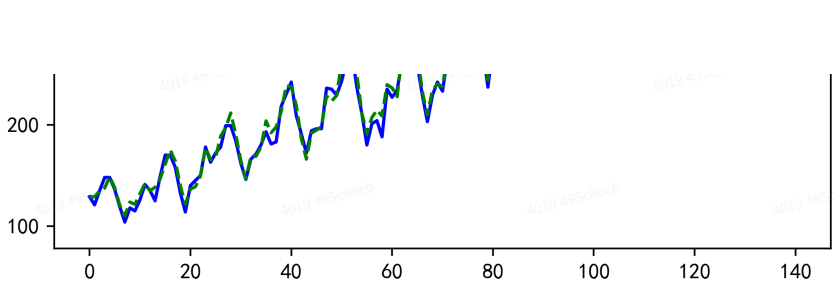
```python
# step4: 定义模型
class LstmModel(nn.Module):
    def __init__(self, inputSize=5, hiddenSize=6):
        super().__init__()
        # LSTM层-> 两个LSTM单元叠加
        self.lstm = nn.LSTM(input_size = inputSize,
                            hidden_size = hiddenSize,
                            num_layers = 2)
        self.output = nn.Linear(6,1)  # 线性输出


    def forward(self,x):
        # x: input->(time_step, batch, input_size)
        # x的维度是【数量量：整批数量量：输入特征维度】
        # X是【112 ,1, 3】
        # lstm两层，目标是从 3->6

        x1, (h ,c)= self.lstm(x)
        # x1: output->(time_step, batch, output_size)

        a, b, c = x1.shape
        out = self.output(x1.view(-1,c))  # 只有三维数据转化为二维才能作为输入
        # 重新将结果转化为三维
        out = out.view(a,b,-1)
        return out

# 定义模型
lstm = LstmModel(inputSize=3) # inputSize与look_back保持一致

# step5.模型训练
def training_loop(nEpochs, model, optimizer, lossFn, trainData,
testData=None):
    trainX, trainY = trainData
    if testData is not None:
        testX, testY = testData
    for epoch in range(1, nEpochs+1):
        optimizer.zero_grad()  # 梯度清0
        trainP = model(trainX)
        loss = lossFn(trainP, trainY)
        loss.backward()  # 反向传播
        optimizer.step()
        if epoch % 100 == 0:
            print(f"Epoch: {epoch}, Loss: {loss.item()}")
    return model

```

```
45
46  # 使用优化器Adam比SGD更好
47  optimizer = optim.Adam(lstm.parameters(), lr=0.1)
48  loss_func = nn.MSELoss()
49
50  # 训练模型
51  lstm = training_loop(nEpochs=1000, model= lstm,
52                       optimizer=optimizer, lossFn=loss_func,
    trainData=trainData)
```

Dockerfile

```
1   #Step6: 可视化
2   dataX, dataY = data  # 原始数据 -> (time_step, batch, input_size)
3   dataY = dataY.view(-1).data.numpy()  # 展开为1维
4   dataY = dataY * (maxPassenger - minPassenger) + minPassenger
5   dataP = lstm(dataX)  # 进行拟合
6   dataP = dataP.view(-1).data.numpy()  # 展开为1维
7   dataP = dataP * (maxPassenger - minPassenger) + minPassenger
8
9   nTrain = int(dataX.shape[0] * rate)  # 拟合的数量
10  nData = dataX.shape[0]  # 预测的数量
11
12  # 绘制对比图
13  plt.rcParams['font.sans-serif'] = 'KaiTi'  # 正常显示中文
14  fig = plt.figure(dpi=400)
15  ax = fig.add_subplot(111)
16  ax.plot(dataY, color='blue', label="实际值")
17  ax.plot(np.arange(nTrain), dataP[:nTrain], color='green',\
18      linestyle='--', label = '拟合值')
19  ax.plot(np.arange(nTrain, nData), dataP[nTrain:], \
20       linestyle='--', color = 'red', label='预测值')
21  ax.legend()
22  fig.savefig('test.png', dpi=400)
23
```




200
100
0
20
40
60
80
100
120
140

# 5.一维卷积

## 5.1函数

一维卷积不代表卷积核只有一维，也不代表被卷积的feature是一维的。

**一维的意思是，卷积的方向是一维的。**

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size,
                stride=1, padding=0, dilation=1, groups=1, bias=True)

in_channels(int) - 输入信号的通道。在文本分类中，即为词向量的维度
out_channels(int) - 卷积产生的通道。卷积核的个数。

kernel_size(int or tuple) - 卷积核的宽度，长度由in_channels来决定的
                            卷积核的大小【in_channels，kernel_size】

stride(int or tuple, optional) - 卷积步长
padding (int or tuple, optional)- 输入的每一条边补充0的层数

bias(bool, optional) - 如果bias=True，添加偏置
```

## 5.2 实例

```
import torch
import torch.nn as nn

# inchannels = 4, out_channels = 2,kernel_size = 3
# 卷积核的大小 (inchannels * kernel_size)  (4,3)
# 输出的维度是2: 卷积核的个数 是 2.
m = nn.Conv1d(4, 2, 3, stride = 2)

# 第一个参数理解为batch的大小，输入是4 * 9格式
input = torch.randn(1, 4, 9)
output = m(input)

print(output.size())  # (1,2,4)
```

理解输入：**输入是一个三通道的矩阵【N，X，Y】。**

**N： 样本的数量，**那么每一个样本的特征就是一个二维的矩阵。

**X，Y： 就是这个样本的特征矩阵。**

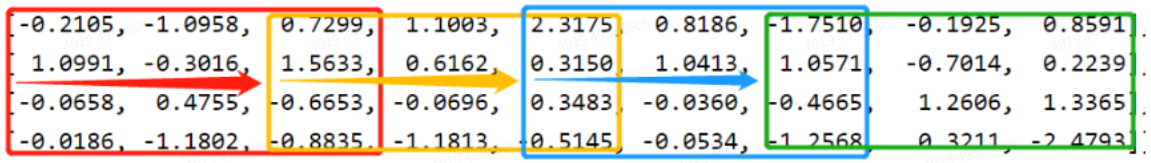
那么卷积核就是直接对【X,Y】矩阵进行**卷积**操作，得到一个结果。

二维卷积的滑动窗口，向右滑动，向下滑动。**一维卷积的滑动窗口就是一个方向滑动，那就是向右/ 向下。**

原始的输入的矩阵大小为： 【4,9】

那么卷积核的大小是： 【4,9】 , 每隔两个步骤卷积一次， 【4*3】卷4 次即可。

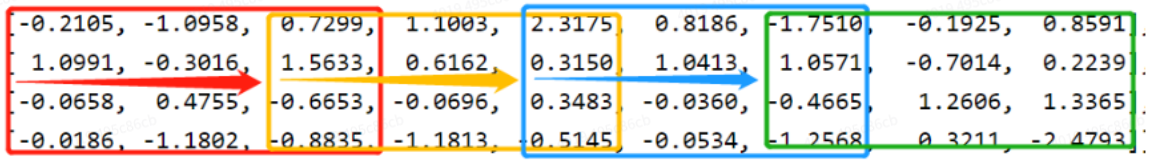
第一个卷积核进行如下操作：




得到输出1*4的输出：

[-0.8012, 0.0589, 0.1576, -0.8222]

第二个卷积核进行类似操作：




得到输出1*4的输出：

[-0.8231, -0.4233, 0.7178, -0.6621]

合并得到最后的2*4的结果：

```
tensor([[[-0.8012,  0.0589,  0.1576, -0.8222],
         [-0.8231, -0.4233,  0.7178, -0.6621]]], grad_fn=<SqueezeBackward1>)
```

输入的input为 4 * 9 ， 输出为 2 * 4

## 5.3完整分类实例

```python
class CNN(nn.Module):
    def __init__(self, B):
        super(CNN, self).__init__()
        self.B = B
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Sequential(
            nn.Conv1d(in_channels=15, out_channels=64, kernel_size=2),  # 24 - 2 + 1 = 23
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=1),  # 23 - 2 + 1 = 22
        )
        self.conv2 = nn.Sequential(
            nn.Conv1d(in_channels=64, out_channels=128, kernel_size=2),  # 22 - 2 + 1 = 21
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=1),  # 21 - 2 + 1 = 20
        )
        self.Linear1 = nn.Linear(self.B * 128 * 20, self.B * 50)
        self.Linear2 = nn.Linear(self.B * 50, self.B)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # print(x.size())  # 15 127 20
        x = x.view(-1)
        # print(x.size())
        x = self.Linear1(x)
        x = self.relu(x)
        x = self.Linear2(x)
        x = x.view(x.shape[0], -1)

        return x

```