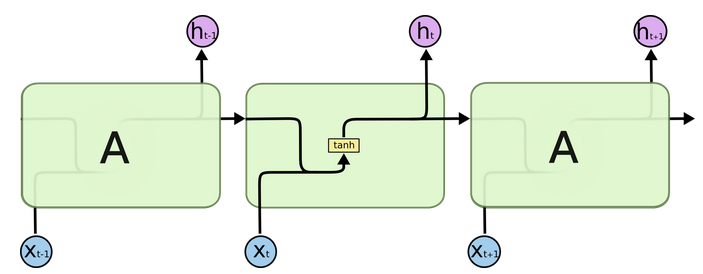
### 实验四 LSTM的实现

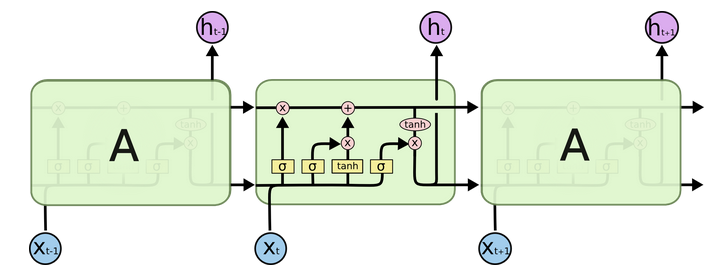
LSTM（Long Short-Term Memory）是长短期记忆网络，是一种时间递归神经网络，适合于处理和预测时间序列中间隔和延迟相对较长的重要事件。LSTM区别于RNN的地方，主要就在于它在算法中加入了一个判断信息有用与否的“处理器”，这个处理器作用的结构被称为cell。一个cell当中被放置了三扇门，分别叫做输入门、遗忘门和输出门。一个信息进入LSTM的网络当中，可以根据规则来判断是否有用。只有符合算法认证的信息才会留下，不符的信息则通过遗忘门被遗忘。说起来无非就是一进二出的工作原理，却可以在反复运算下解决神经网络中长期存在的大问题。目前已经证明，LSTM是解决长序依赖问题的有效技术，并且这种技术的普适性非常高，导致带来的可能性变化非常多。各研究者根据LSTM纷纷提出了自己的变量版本，这就让LSTM可以处理千变万化的垂直问题。

**实验目的：**加深对LSTM模型的理解，能够使用LSTM模型解决简单问题

**实验原理：**LSTM是一类可以处理长期依赖问题的特殊的RNN，LSTM主要用来处理长期依赖问题，与传统RNN相比，长时间的信息记忆能力是与生俱来的。所有的RNN链式结构中都有不断重复的模块，用来随时间传递信息。传统的RNN使用十分简单的结构，如下图所示。



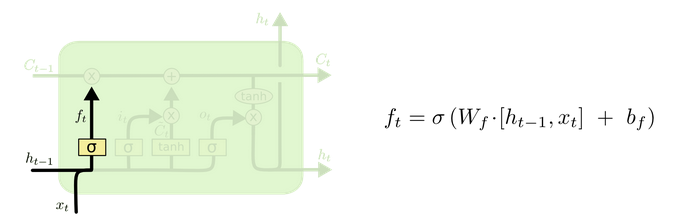
LSTM链式结构中重复模块的结构更加复杂，有四个互相交互的层 (如下图所示)。



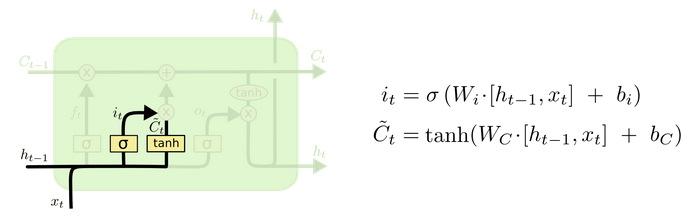
与传统RNN相比，除了拥有隐藏状态外，LSTM还增加了一个细胞状态，记录随时间传递的信息。在传递过程中，通过当前输入、上一时刻隐藏层状态、上一时刻细胞状态以及门结构来增加或删除细胞状态中的信息。门结构用来控制增加或删除信息的程度，一般由sigmoid函数和向量点乘来实现。

LSTM共包含3个门结构，来控制细胞状态和隐藏状态，下边分别进行介绍。

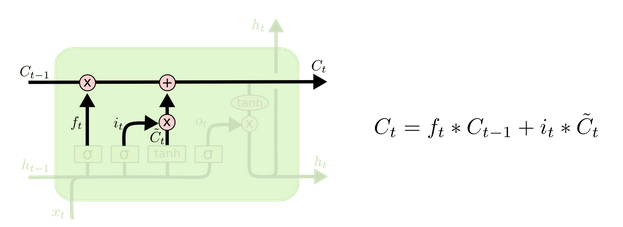
遗忘门。遗忘门决定上一时刻细胞状态中的多少信息可以传递到当前时刻中。



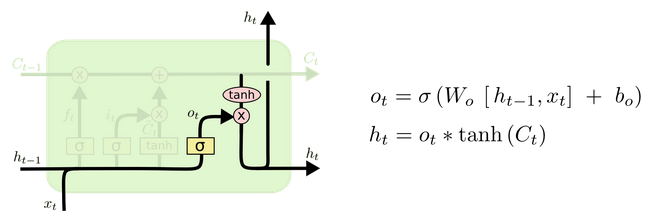
输入门。输入门用来控制当前输入新生成的信息中有多少信息可以加入到细胞状态中。Ct层用来产生当前时刻新的信息，it层用来控制有多少新信息可以传递给细胞状态。



更新细胞状态。基于遗忘门和输入门的输出，来更新细胞状态。更新后的细胞状态有两部分构成:一，来自上一时刻旧的细胞状态信息；二，当前输入新生成的信息。



输出门，基于更新的细胞状态，输出隐藏状态。



**实验内容：**根据LSTM模型的相关知识，使用Python语言实现一个简单LSTM模型。简单程序示例如下：

import numpy as np

# 输出单元激活函数

def softmax(x):

x = np.array(x)

max\_x = np.max(x)

return np.exp(x-max\_x) / np.sum(np.exp(x-max\_x))

def sigmoid(x):

return 1.0/(1.0 + np.exp(-x))

def tanh(x):

return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))

class myLSTM:

def \_\_init\_\_(self, data\_dim, hidden\_dim=100):

# data\_dim: 词向量维度，即词典长度; hidden\_dim: 隐单元维度

self.data\_dim = data\_dim

self.hidden\_dim = hidden\_dim

# 初始化权重向量

self.whi, self.wxi, self.bi = self.\_init\_wh\_wx()

self.whf, self.wxf, self.bf = self.\_init\_wh\_wx()

self.who, self.wxo, self.bo = self.\_init\_wh\_wx()

self.wha, self.wxa, self.ba = self.\_init\_wh\_wx()

self.wy, self.by = np.random.uniform(-np.sqrt(1.0/self.hidden\_dim), np.sqrt(1.0/self.hidden\_dim),

(self.data\_dim, self.hidden\_dim)), \

np.random.uniform(-np.sqrt(1.0/self.hidden\_dim), np.sqrt(1.0/self.hidden\_dim),

(self.data\_dim, 1))

# 初始化 wh, wx, b

def \_init\_wh\_wx(self):

wh = np.random.uniform(-np.sqrt(1.0/self.hidden\_dim), np.sqrt(1.0/self.hidden\_dim),

(self.hidden\_dim, self.hidden\_dim))

wx = np.random.uniform(-np.sqrt(1.0/self.data\_dim), np.sqrt(1.0/self.data\_dim),

(self.hidden\_dim, self.data\_dim))

b = np.random.uniform(-np.sqrt(1.0/self.data\_dim), np.sqrt(1.0/self.data\_dim),

(self.hidden\_dim, 1))

return wh, wx, b

# 初始化各个状态向量

def \_init\_s(self, T):

iss = np.array([np.zeros((self.hidden\_dim, 1))] \* (T + 1)) # input gate

fss = np.array([np.zeros((self.hidden\_dim, 1))] \* (T + 1)) # forget gate

oss = np.array([np.zeros((self.hidden\_dim, 1))] \* (T + 1)) # output gate

ass = np.array([np.zeros((self.hidden\_dim, 1))] \* (T + 1)) # current inputstate

hss = np.array([np.zeros((self.hidden\_dim, 1))] \* (T + 1)) # hidden state

css = np.array([np.zeros((self.hidden\_dim, 1))] \* (T + 1)) # cell state

ys = np.array([np.zeros((self.data\_dim, 1))] \* T) # output value

return {'iss': iss, 'fss': fss, 'oss': oss,

'ass': ass, 'hss': hss, 'css': css,

'ys': ys}

# 前向传播，单个x

def forward(self, x):

# 向量时间长度

T = len(x)

# 初始化各个状态向量

stats = self.\_init\_s(T)

for t in range(T):

# 前一时刻隐藏状态

ht\_pre = np.array(stats['hss'][t-1]).reshape(-1, 1)

# input gate

stats['iss'][t] = self.\_cal\_gate(self.whi, self.wxi, self.bi, ht\_pre, x[t], sigmoid)

# forget gate

stats['fss'][t] = self.\_cal\_gate(self.whf, self.wxf, self.bf, ht\_pre, x[t], sigmoid)

# output gate

stats['oss'][t] = self.\_cal\_gate(self.who, self.wxo, self.bo, ht\_pre, x[t], sigmoid)

# current inputstate

stats['ass'][t] = self.\_cal\_gate(self.wha, self.wxa, self.ba, ht\_pre, x[t], tanh)

# cell state, ct = ft \* ct\_pre + it \* at

stats['css'][t] = stats['fss'][t] \* stats['css'][t-1] + stats['iss'][t] \* stats['ass'][t]

# hidden state, ht = ot \* tanh(ct)

stats['hss'][t] = stats['oss'][t] \* tanh(stats['css'][t])

# output value, yt = softmax(self.wy.dot(ht) + self.by)

stats['ys'][t] = softmax(self.wy.dot(stats['hss'][t]) + self.by)

return stats

# 计算各个门的输出

def \_cal\_gate(self, wh, wx, b, ht\_pre, x, activation):

return activation(wh.dot(ht\_pre) + wx[:, x].reshape(-1,1) + b)

# 预测输出，单个x

def predict(self, x):

stats = self.forward(x)

pre\_y = np.argmax(stats['ys'].reshape(len(x), -1), axis=1)

return pre\_y

# 计算损失， softmax交叉熵损失函数， (x,y)为多个样本

def loss(self, x, y):

cost = 0

for i in xrange(len(y)):

stats = self.forward(x[i])

# 取出 y[i] 中每一时刻对应的预测值

pre\_yi = stats['ys'][xrange(len(y[i])), y[i]]

cost -= np.sum(np.log(pre\_yi))

# 统计所有y中词的个数, 计算平均损失

N = np.sum([len(yi) for yi in y])

ave\_loss = cost / N

return ave\_loss

# 初始化偏导数 dwh, dwx, db

def \_init\_wh\_wx\_grad(self):

dwh = np.zeros(self.whi.shape)

dwx = np.zeros(self.wxi.shape)

db = np.zeros(self.bi.shape)

return dwh, dwx, db

# 求梯度, (x,y)为一个样本

def bptt(self, x, y):

dwhi, dwxi, dbi = self.\_init\_wh\_wx\_grad()

dwhf, dwxf, dbf = self.\_init\_wh\_wx\_grad()

dwho, dwxo, dbo = self.\_init\_wh\_wx\_grad()

dwha, dwxa, dba = self.\_init\_wh\_wx\_grad()

dwy, dby = np.zeros(self.wy.shape), np.zeros(self.by.shape)

# 初始化 delta\_ct，因为后向传播过程中，此值需要累加

delta\_ct = np.zeros((self.hidden\_dim, 1))

# 前向计算

stats = self.forward(x)

# 目标函数对输出 y 的偏导数

delta\_o = stats['ys']

delta\_o[np.arange(len(y)), y] -= 1

for t in np.arange(len(y))[::-1]:

# 输出层wy, by的偏导数，由于所有时刻的输出共享输出权值矩阵，故所有时刻累加

dwy += delta\_o[t].dot(stats['hss'][t].reshape(1, -1))

dby += delta\_o[t]

# 目标函数对隐藏状态的偏导数

delta\_ht = self.wy.T.dot(delta\_o[t])

# 各个门及状态单元的偏导数

delta\_ot = delta\_ht \* tanh(stats['css'][t])

delta\_ct += delta\_ht \* stats['oss'][t] \* (1-tanh(stats['css'][t])\*\*2)

delta\_it = delta\_ct \* stats['ass'][t]

delta\_ft = delta\_ct \* stats['css'][t-1]

delta\_at = delta\_ct \* stats['iss'][t]

delta\_at\_net = delta\_at \* (1-stats['ass'][t]\*\*2)

delta\_it\_net = delta\_it \* stats['iss'][t] \* (1-stats['iss'][t])

delta\_ft\_net = delta\_ft \* stats['fss'][t] \* (1-stats['fss'][t])

delta\_ot\_net = delta\_ot \* stats['oss'][t] \* (1-stats['oss'][t])

# 更新各权重矩阵的偏导数，由于所有时刻共享权值，故所有时刻累加

dwhf, dwxf, dbf = self.\_cal\_grad\_delta(dwhf, dwxf, dbf, delta\_ft\_net, stats['hss'][t-1], x[t])

dwhi, dwxi, dbi = self.\_cal\_grad\_delta(dwhi, dwxi, dbi, delta\_it\_net, stats['hss'][t-1], x[t])

dwha, dwxa, dba = self.\_cal\_grad\_delta(dwha, dwxa, dba, delta\_at\_net, stats['hss'][t-1], x[t])

dwho, dwxo, dbo = self.\_cal\_grad\_delta(dwho, dwxo, dbo, delta\_ot\_net, stats['hss'][t-1], x[t])

return [dwhf, dwxf, dbf,

dwhi, dwxi, dbi,

dwha, dwxa, dba,

dwho, dwxo, dbo,

dwy, dby]

# 更新各权重矩阵的偏导数

def \_cal\_grad\_delta(self, dwh, dwx, db, delta\_net, ht\_pre, x):

dwh += delta\_net \* ht\_pre

dwx += delta\_net \* x

db += delta\_net

return dwh, dwx, db

# 计算梯度, (x,y)为一个样本

def sgd\_step(self, x, y, learning\_rate):

dwhf, dwxf, dbf, \

dwhi, dwxi, dbi, \

dwha, dwxa, dba, \

dwho, dwxo, dbo, \

dwy, dby = self.bptt(x, y)

# 更新权重矩阵

self.whf, self.wxf, self.bf = self.\_update\_wh\_wx(learning\_rate, self.whf, self.wxf, self.bf, dwhf, dwxf, dbf)

self.whi, self.wxi, self.bi = self.\_update\_wh\_wx(learning\_rate, self.whi, self.wxi, self.bi, dwhi, dwxi, dbi)

self.wha, self.wxa, self.ba = self.\_update\_wh\_wx(learning\_rate, self.wha, self.wxa, self.ba, dwha, dwxa, dba)

self.who, self.wxo, self.bo = self.\_update\_wh\_wx(learning\_rate, self.who, self.wxo, self.bo, dwho, dwxo, dbo)

self.wy, self.by = self.wy - learning\_rate \* dwy, self.by - learning\_rate \* dby

# 更新权重矩阵

def \_update\_wh\_wx(self, learning\_rate, wh, wx, b, dwh, dwx, db):

wh -= learning\_rate \* dwh

wx -= learning\_rate \* dwx

b -= learning\_rate \* db

return wh, wx, b

# 训练 LSTM

def train(self, X\_train, y\_train, learning\_rate=0.005, n\_epoch=5):

losses = []

num\_examples = 0

for epoch in xrange(n\_epoch):

for i in xrange(len(y\_train)):

self.sgd\_step(X\_train[i], y\_train[i], learning\_rate)

num\_examples += 1

loss = self.loss(X\_train, y\_train)

losses.append(loss)

print 'epoch {0}: loss = {1}'.format(epoch+1, loss)

if len(losses) > 1 and losses[-1] > losses[-2]:

learning\_rate \*= 0.5

print 'decrease learning\_rate to', learning\_rate

**实验要求：**

1. 随机产生0-127之间的两个八位的二进制整数，作为一组输入数据，将这两个数的和作为一个标签，这三个数据组成一组训练数据，训练数据的组数应尽可能多。
2. 创建LSTM网络。
3. 实现两个八位的二进制整数的加法运算，网络能够输出正确的加法运算结果。