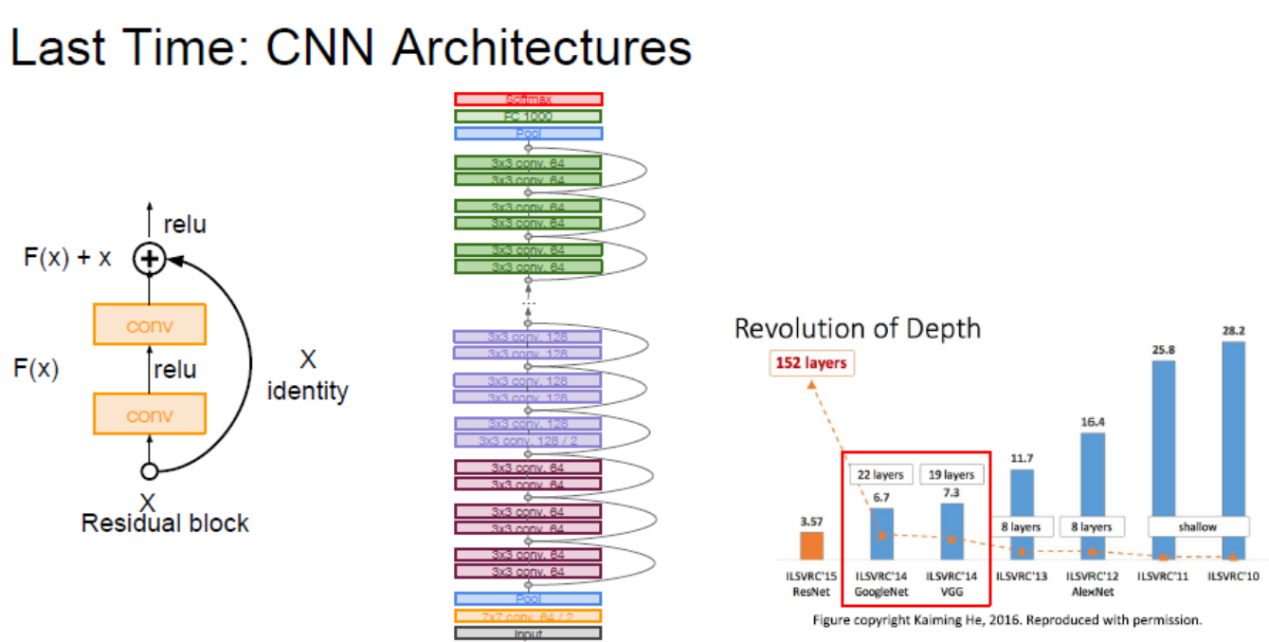


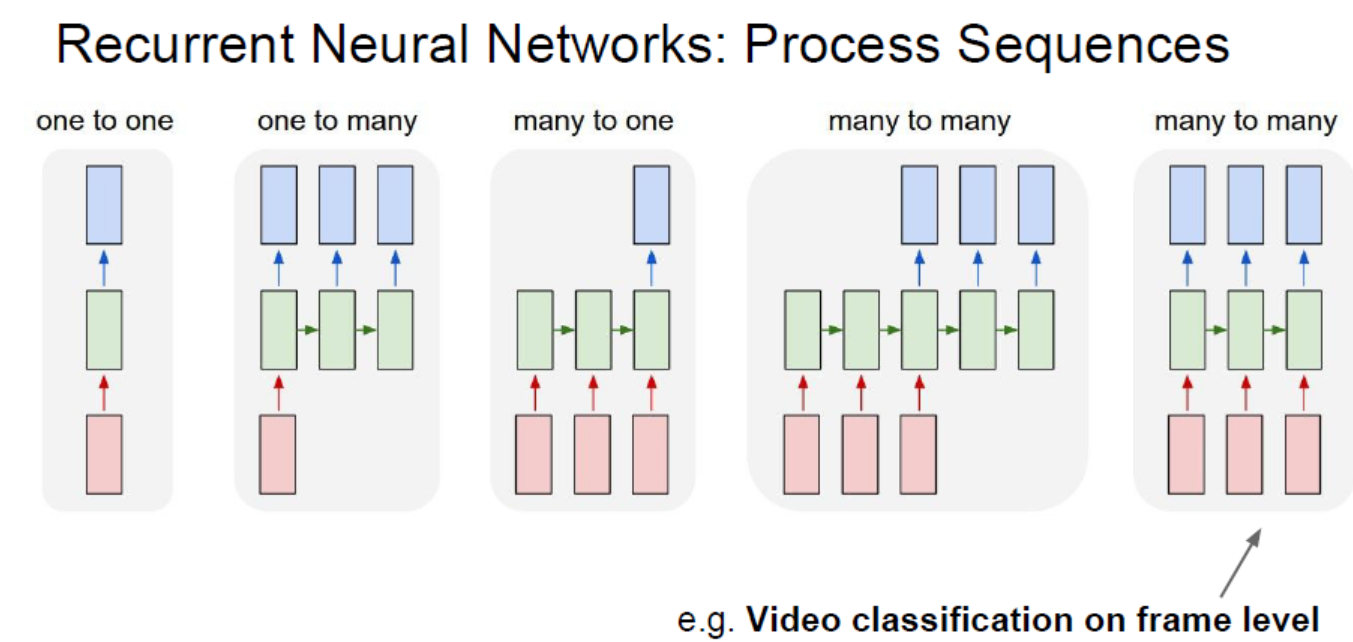


在2014年前，没有batch normalize，训练过程一度比较难控制

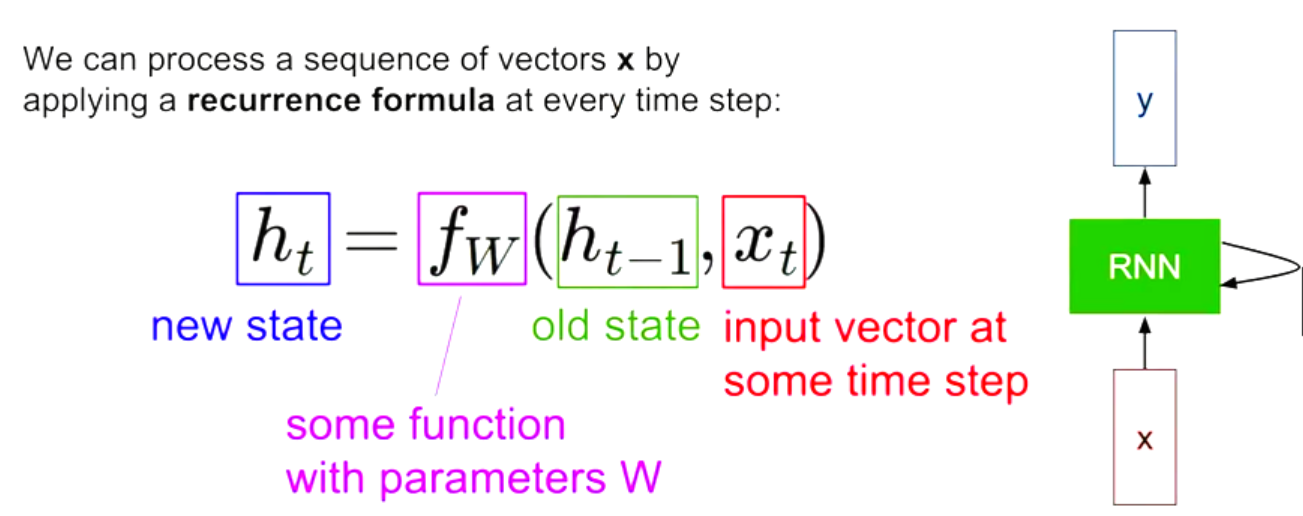


梯度流的概念

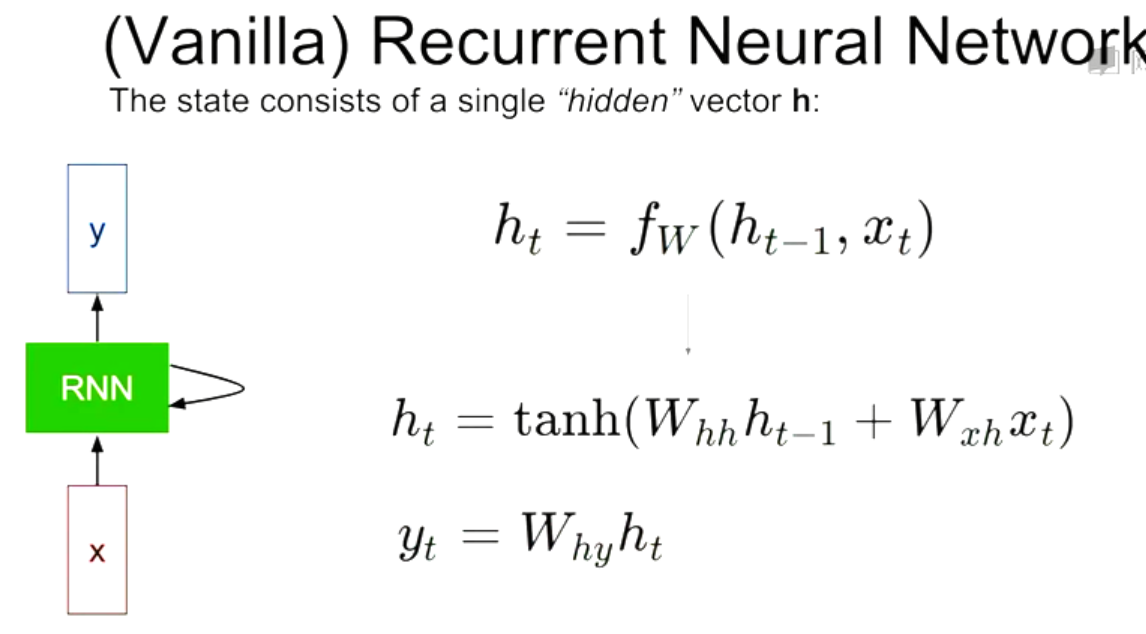
在反向传播的过程中，对于一个加法门，反向 传播时会沿两个加法的路径回流



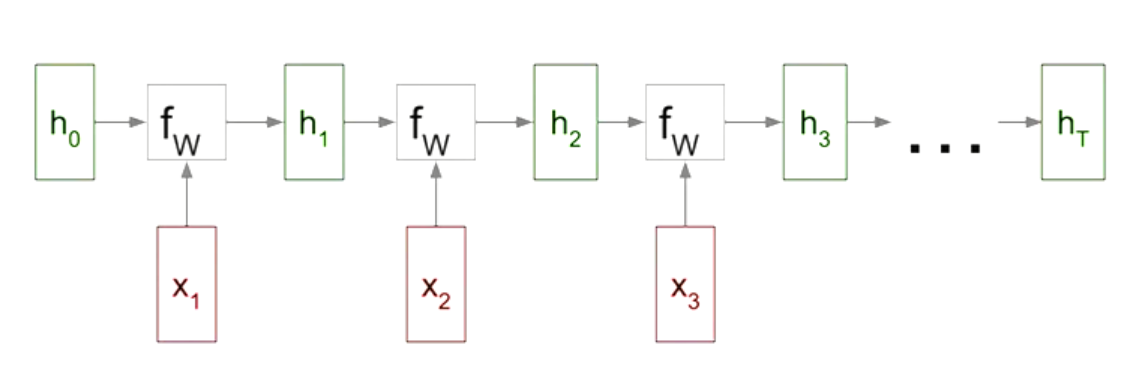
## RNN的核心：输入 -> 更新hidden state -> -> 新的输入…



\*\*\* 注意 fW不变，变的只是输入和目前的hidden state. 而这个function 和 对应的 W就是我们要训练的模型



RNN 状态更新图：



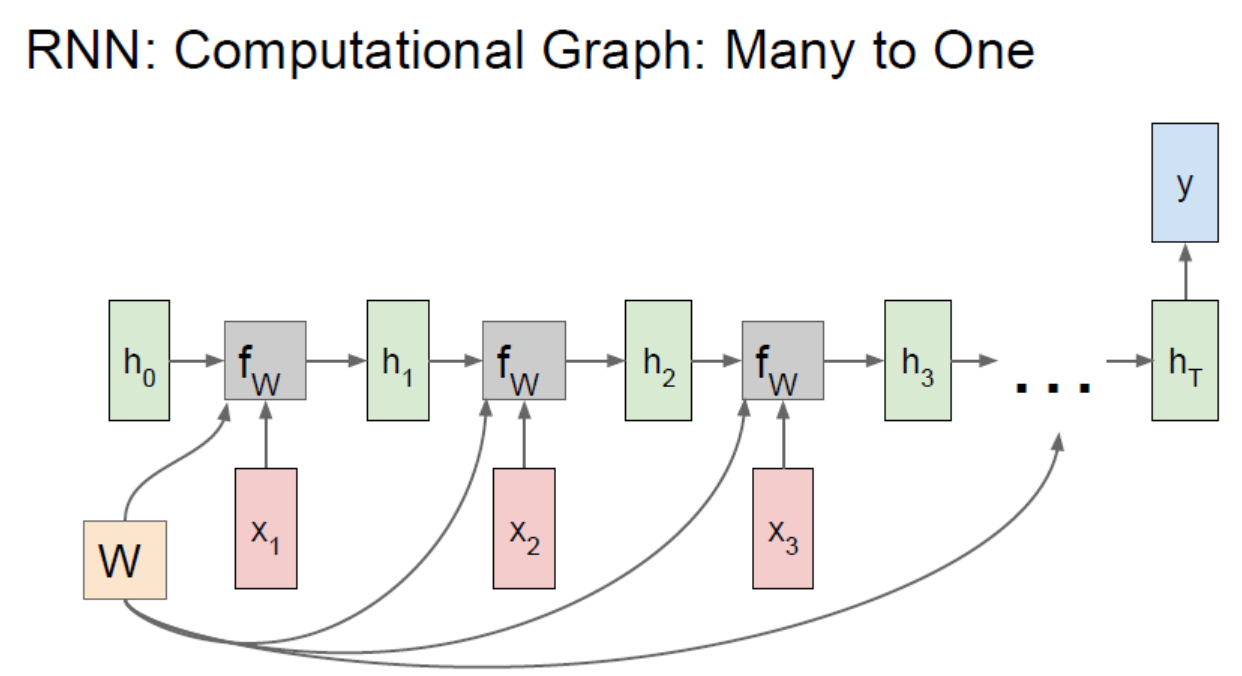
而他们共用了一个W,并且可以计算出每块的Loss，相加后就是总Loss

多对多：



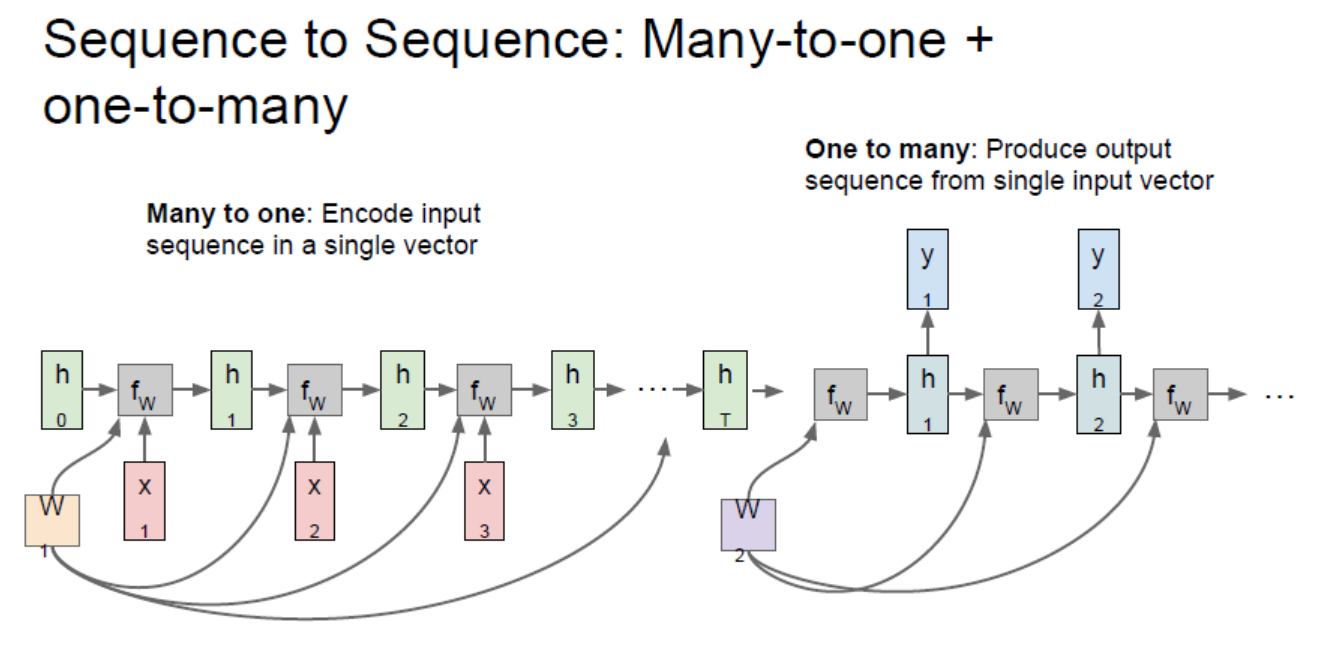
在反向传播时就可以把把对应块的Loss值反向传给W，更新W

一对多：



Sequence to Sequence 序列对序列：

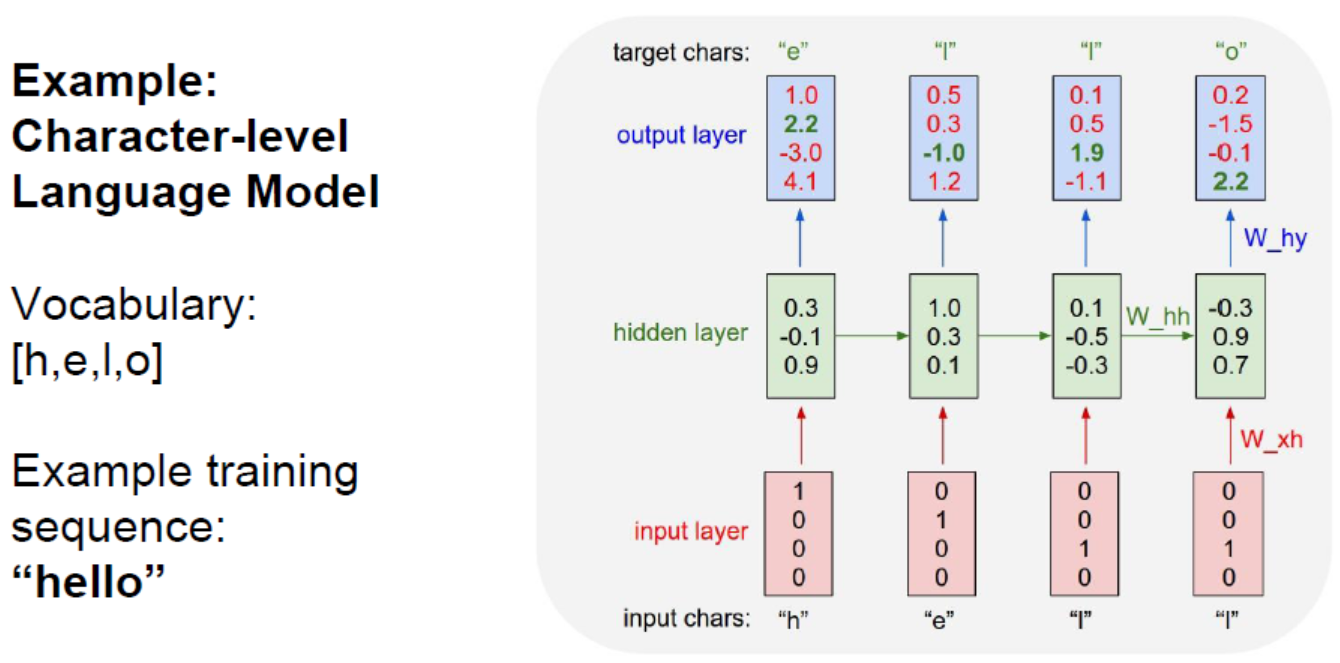
Sutskever et al, “Sequence to Sequence Learning with Neural Networks”, NIPS 2014



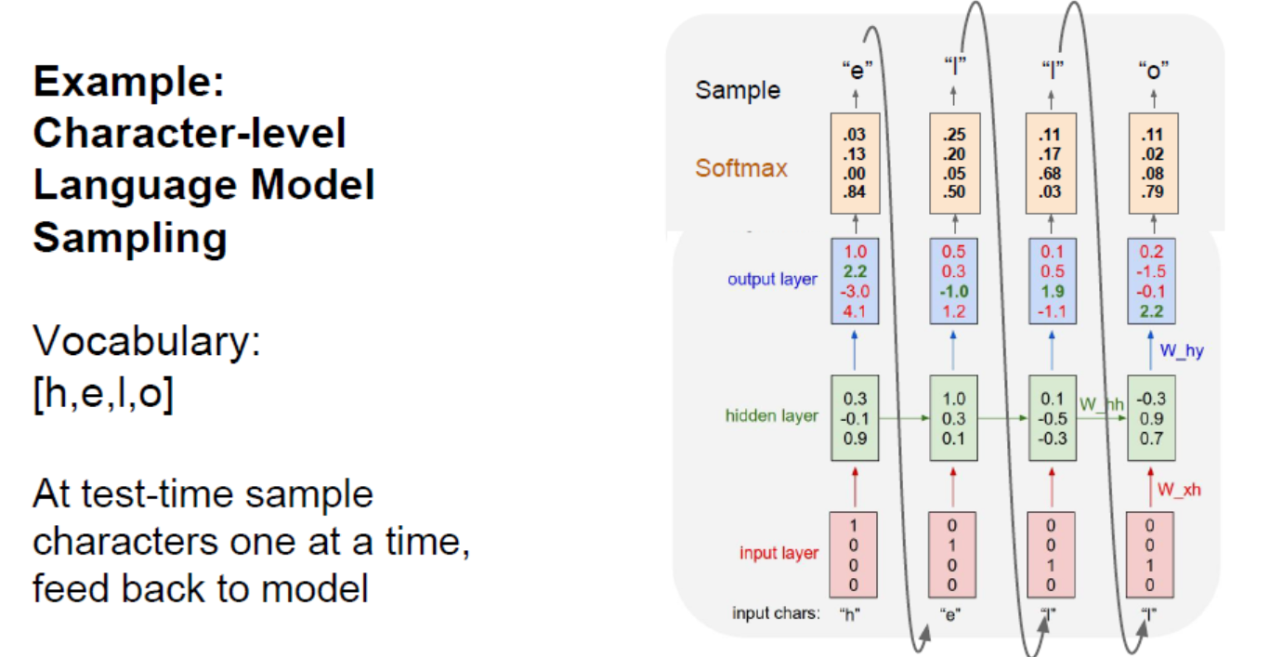
这个过程包括 encode 和 decode

而这就是最近很流行的语言建模

### 例：train-time

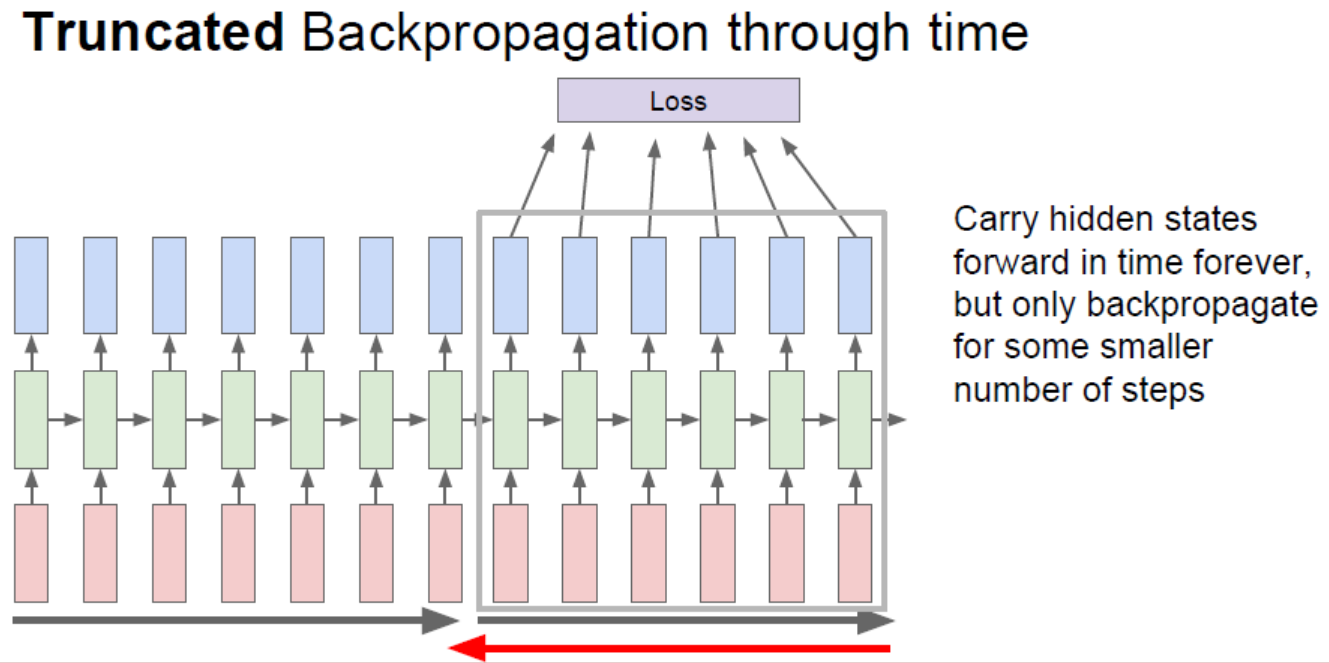


### Test-time



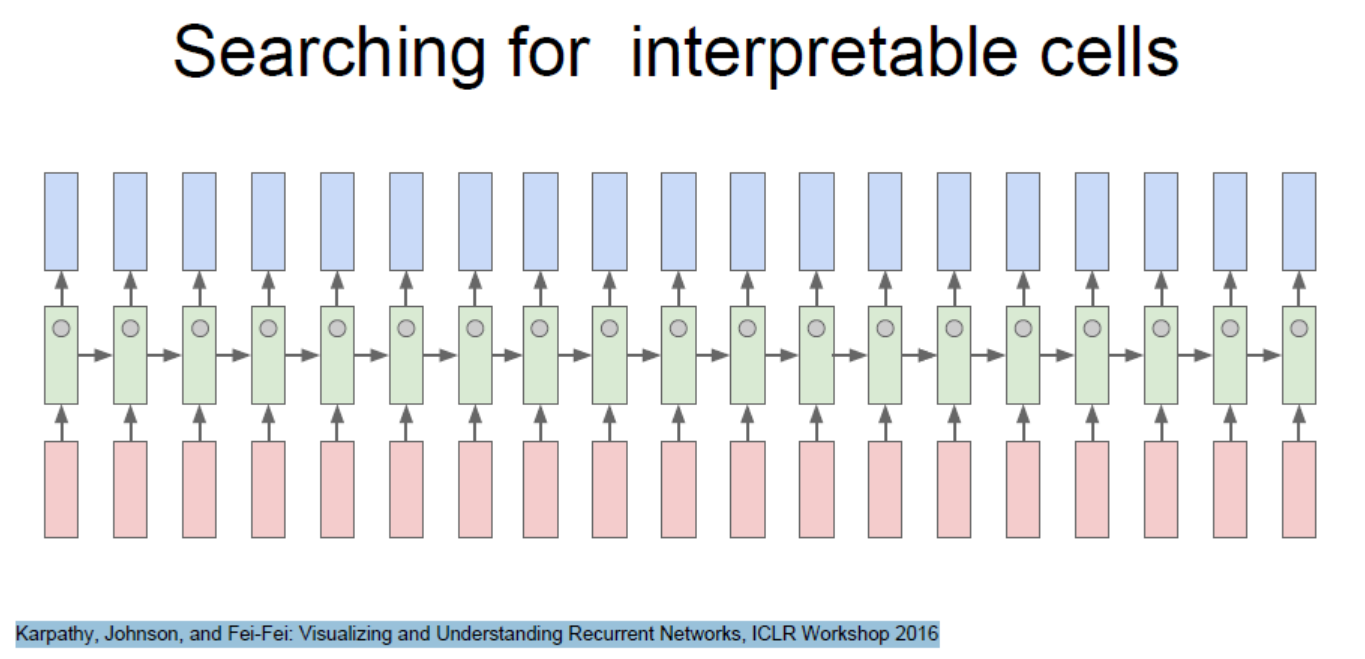
One hot vector: 0,0,0,1 0,0,1,0 …

序列太长，一般前向一步，就再后向一步



Min-char-rnn.py只用了112行代码就实现了这个RNN算法

|  |
| --- |
| """ |
|  | Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy) |
|  | BSD License |
|  | """ |
|  | import numpy as np |
|  |  |
|  | # data I/O |
|  | data = open('input.txt', 'r').read() # should be simple plain text file |
|  | chars = list(set(data)) |
|  | data\_size, vocab\_size = len(data), len(chars) |
|  | print 'data has %d characters, %d unique.' % (data\_size, vocab\_size) |
|  | char\_to\_ix = { ch:i for i,ch in enumerate(chars) } |
|  | ix\_to\_char = { i:ch for i,ch in enumerate(chars) } |
|  |  |
|  | # hyperparameters |
|  | hidden\_size = 100 # size of hidden layer of neurons |
|  | seq\_length = 25 # number of steps to unroll the RNN for |
|  | learning\_rate = 1e-1 |
|  |  |
|  | # model parameters |
|  | Wxh = np.random.randn(hidden\_size, vocab\_size)\*0.01 # input to hidden |
|  | Whh = np.random.randn(hidden\_size, hidden\_size)\*0.01 # hidden to hidden |
|  | Why = np.random.randn(vocab\_size, hidden\_size)\*0.01 # hidden to output |
|  | bh = np.zeros((hidden\_size, 1)) # hidden bias |
|  | by = np.zeros((vocab\_size, 1)) # output bias |
|  |  |
|  | def lossFun(inputs, targets, hprev): |
|  | """ |
|  | inputs,targets are both list of integers. |
|  | hprev is Hx1 array of initial hidden state |
|  | returns the loss, gradients on model parameters, and last hidden state |
|  | """ |
|  | xs, hs, ys, ps = {}, {}, {}, {} |
|  | hs[-1] = np.copy(hprev) |
|  | loss = 0 |
|  | # forward pass |
|  | for t in xrange(len(inputs)): |
|  | xs[t] = np.zeros((vocab\_size,1)) # encode in 1-of-k representation |
|  | xs[t][inputs[t]] = 1 |
|  | hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state |
|  | ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars |
|  | ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars |
|  | loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss) |
|  | # rint inputs  # for t in reversed(xrange(len(inputs)))pass: compute gradients going backwards |
|  | dWxh, dWhh, dWhy = np.zeros\_like(Wxh), np.zeros\_like(Whh), np.zeros\_like(Why) |
|  | dbh, dby = np.zeros\_like(bh), np.zeros\_like(by) |
|  | dhnext = np.zeros\_like(hs[0]) |
|  | for t in reversed(xrange(len(inputs))): |
|  | dy = np.copy(ps[t]) |
|  | dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-study/#grad if confused here |
|  | dWhy += np.dot(dy, hs[t].T) |
|  | dby += dy |
|  | dh = np.dot(Why.T, dy) + dhnext # backprop into h |
|  | dhraw = (1 - hs[t] \* hs[t]) \* dh # backprop through tanh nonlinearity |
|  | dbh += dhraw |
|  | dWxh += np.dot(dhraw, xs[t].T) |
|  | dWhh += np.dot(dhraw, hs[t-1].T) |
|  | dhnext = np.dot(Whh.T, dhraw) |
|  | for dparam in [dWxh, dWhh, dWhy, dbh, dby]: |
|  | np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients |
|  | return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1] |
|  |  |
|  | def sample(h, seed\_ix, n): |
|  | """ |
|  | sample a sequence of integers from the model |
|  | h is memory state, seed\_ix is seed letter for first time step |
|  | """ |
|  | x = np.zeros((vocab\_size, 1)) |
|  | x[seed\_ix] = 1 |
|  | ixes = [] |
|  | for t in xrange(n): |
|  | h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh) |
|  | y = np.dot(Why, h) + by |
|  | p = np.exp(y) / np.sum(np.exp(y)) |
|  | ix = np.random.choice(range(vocab\_size), p=p.ravel()) |
|  | x = np.zeros((vocab\_size, 1)) |
|  | x[ix] = 1 |
|  | ixes.append(ix) |
|  | return ixes |
|  |  |
|  | n, p = 0, 0 |
|  | mWxh, mWhh, mWhy = np.zeros\_like(Wxh), np.zeros\_like(Whh), np.zeros\_like(Why) |
|  | mbh, mby = np.zeros\_like(bh), np.zeros\_like(by) # memory variables for Adagrad |
|  | smooth\_loss = -np.log(1.0/vocab\_size)\*seq\_length # loss at iteration 0 |
|  | while True: |
|  | # prepare inputs (we're sweeping from left to right in steps seq\_length long) |
|  | if p+seq\_length+1 >= len(data) or n == 0: |
|  | hprev = np.zeros((hidden\_size,1)) # reset RNN memory |
|  | p = 0 # go from start of data |
|  | inputs = [char\_to\_ix[ch] for ch in data[p:p+seq\_length]] |
|  | targets = [char\_to\_ix[ch] for ch in data[p+1:p+seq\_length+1]] |
|  |  |
|  | # sample from the model now and then |
|  | if n % 100 == 0: |
|  | sample\_ix = sample(hprev, inputs[0], 200) |
|  | txt = ''.join(ix\_to\_char[ix] for ix in sample\_ix) |
|  | print '----\n %s \n----' % (txt, ) |
|  |  |
|  | # forward seq\_length characters through the net and fetch gradient |
|  | loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev) |
|  | smooth\_loss = smooth\_loss \* 0.999 + loss \* 0.001 |
|  | if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth\_loss) # print progress |
|  |  |
|  | # perform parameter update with Adagrad |
|  | for param, dparam, mem in zip([Wxh, Whh, Why, bh, by], |
|  | [dWxh, dWhh, dWhy, dbh, dby], |
|  | [mWxh, mWhh, mWhy, mbh, mby]): |
|  | mem += dparam \* dparam |
|  | param += -learning\_rate \* dparam / np.sqrt(mem + 1e-8) # adagrad update |
|  |  |
|  | p += seq\_length # move data pointer |
|  | n += 1 # iteration counter |



Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

# Image caption 输入图片 输出文本

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.

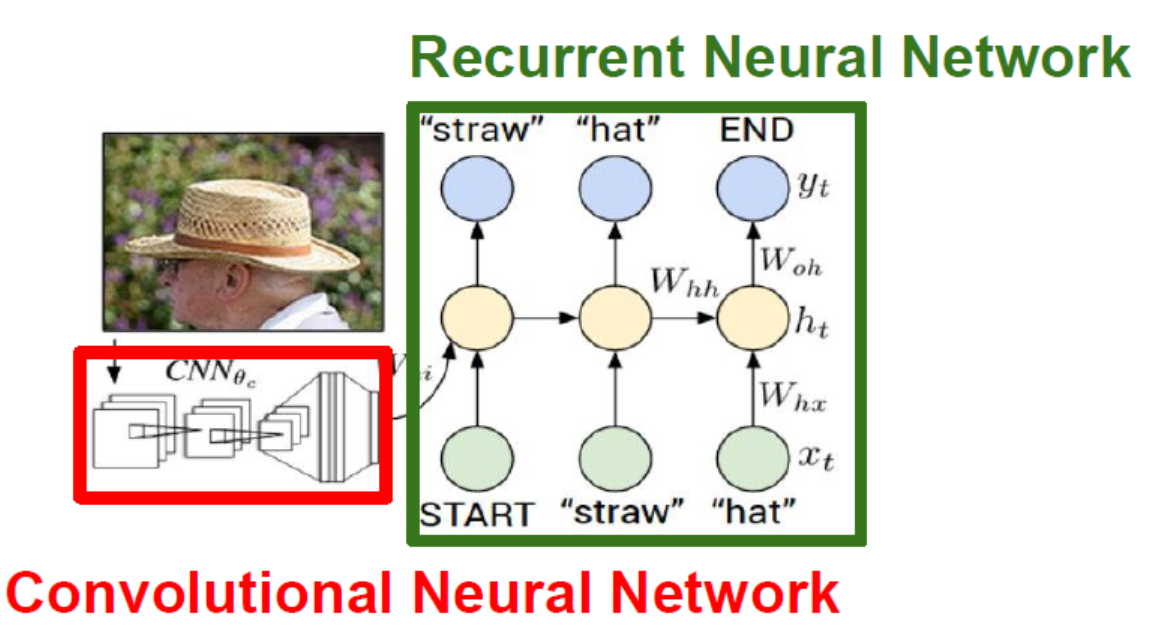
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei

Show and Tell: A Neural Image Caption Generator, Vinyals et al.

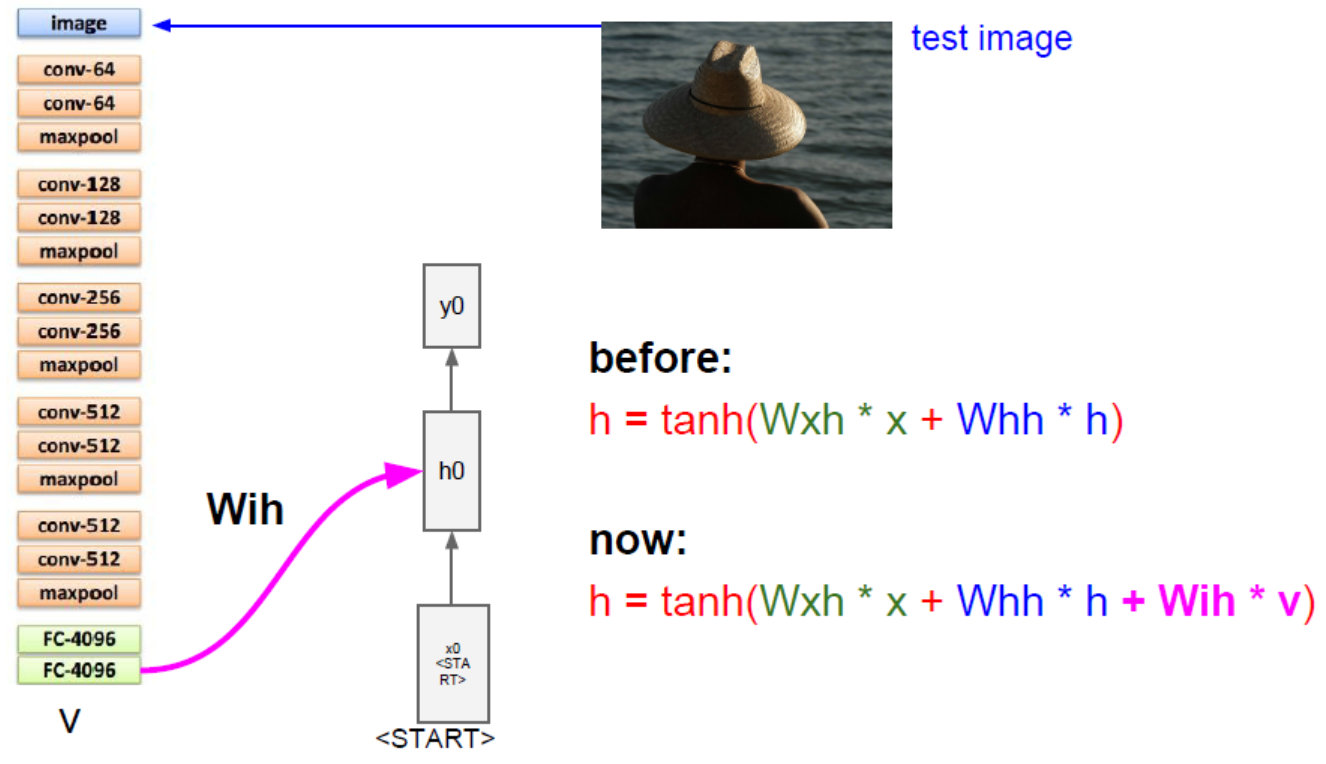
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.

Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnicks

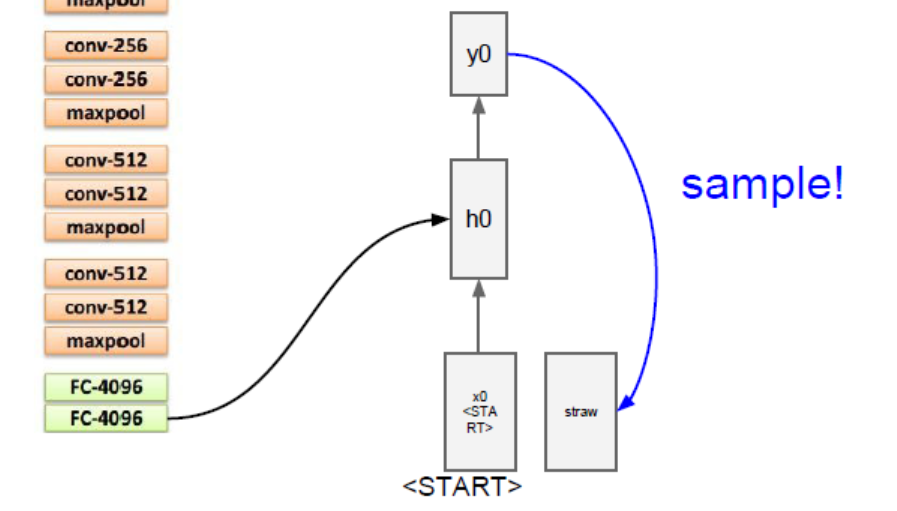
图片是输入，输出Caption是可变长度的词的序列，适合用RNN

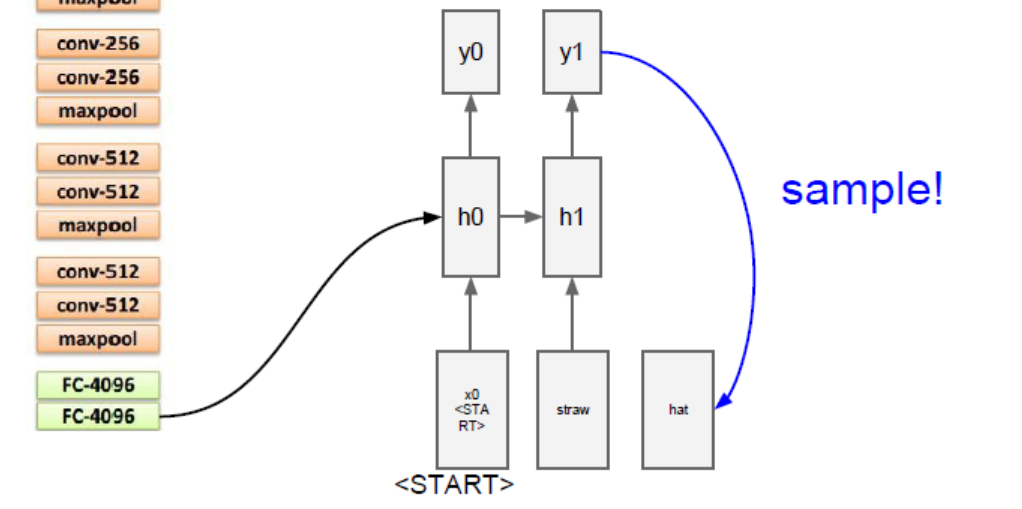


更新方式： 添加一个对向图片的权重矩阵

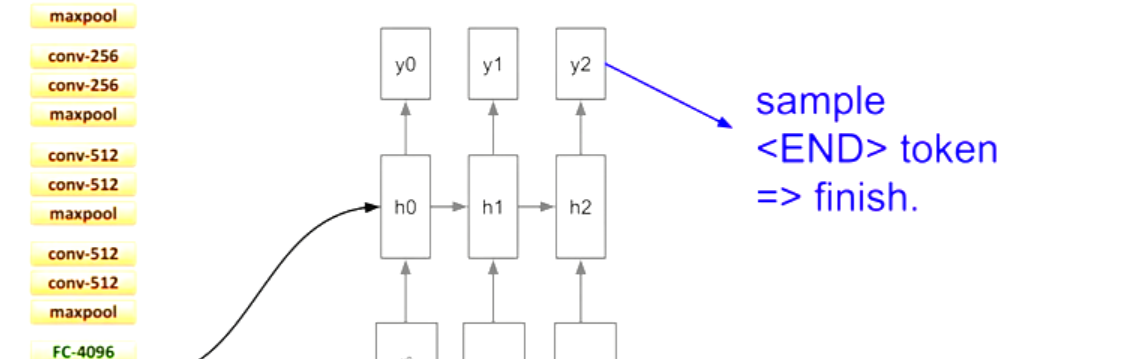


计算后从中采样



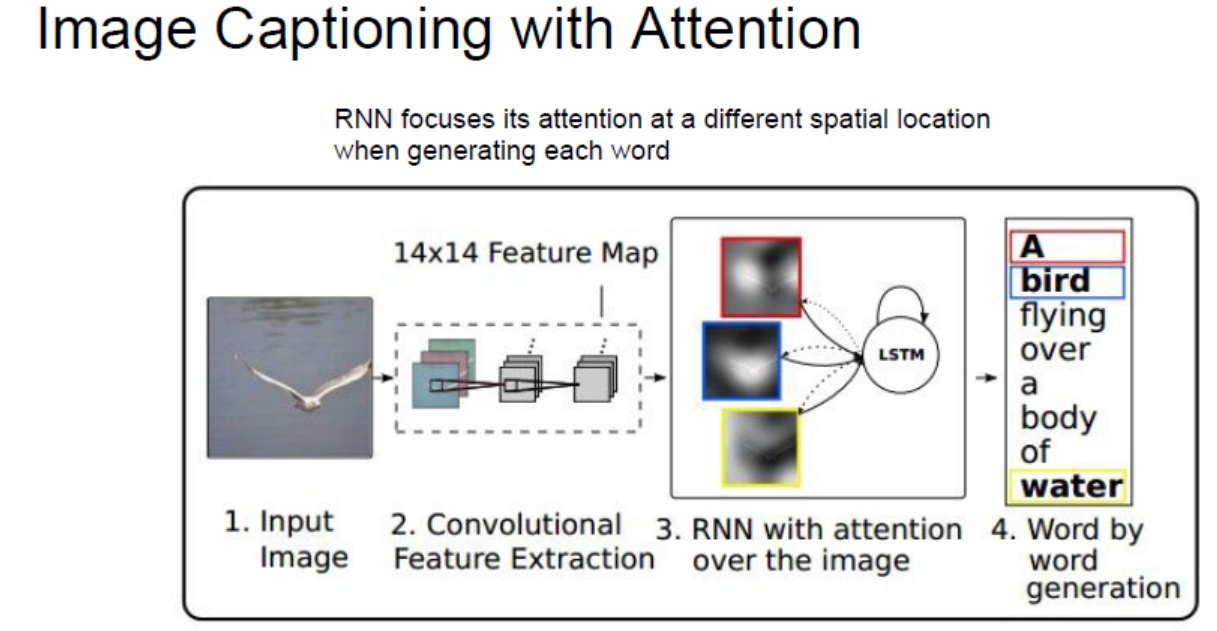


停止：



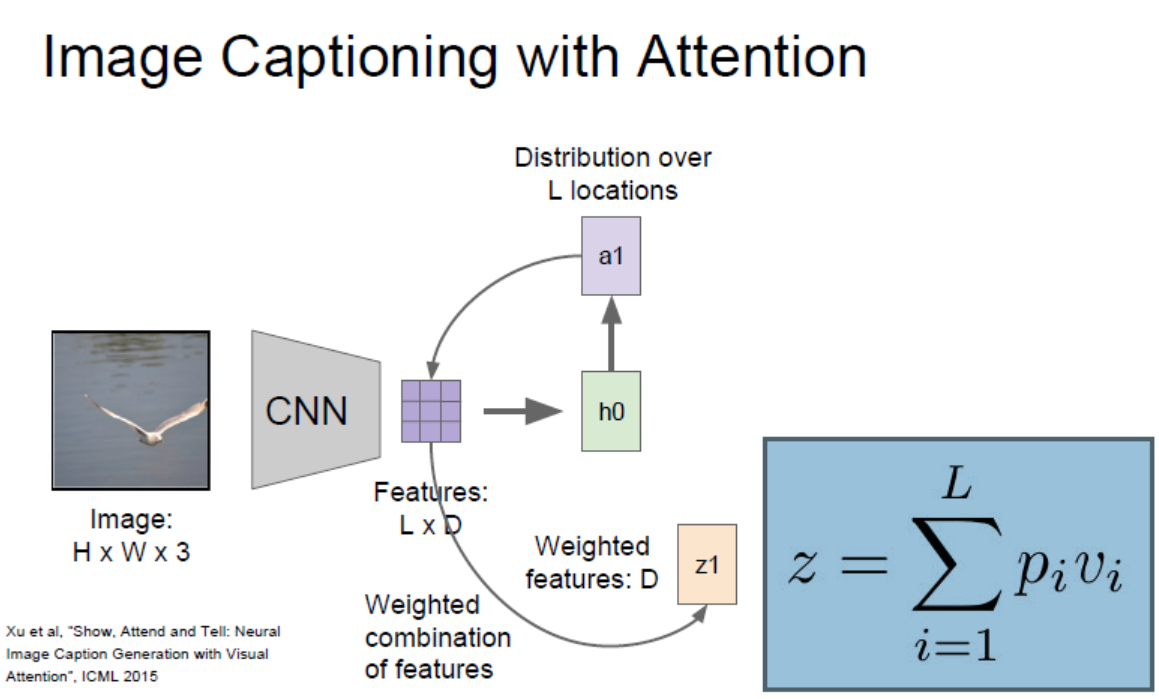
可以找到NLP标记的图片库，如 Microsoft‘s COCO

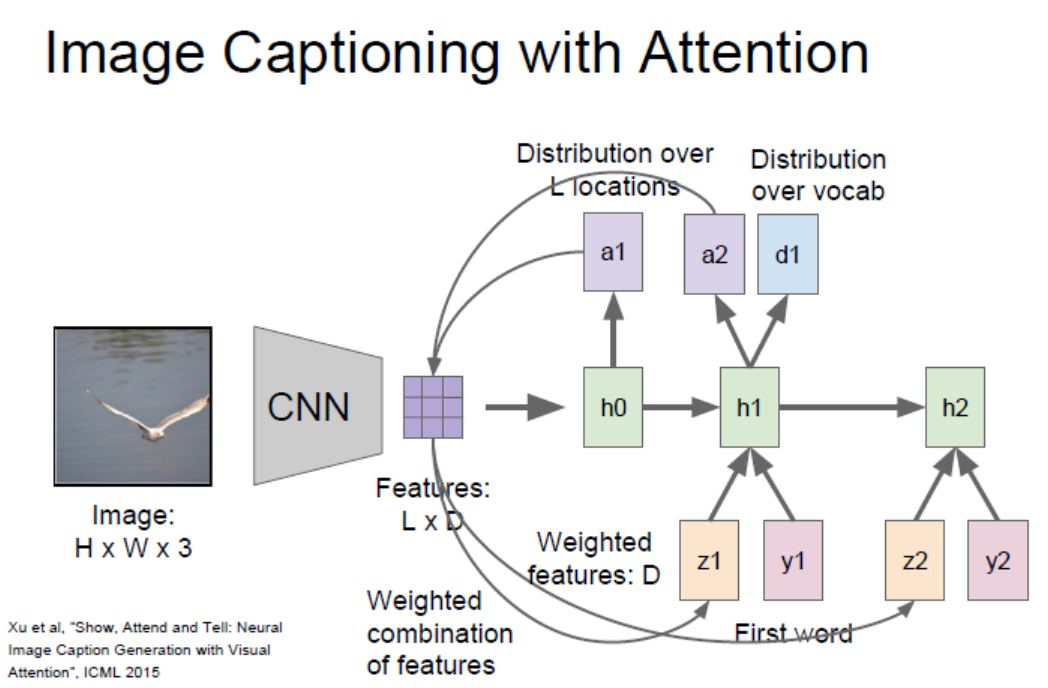
模型：Image Captioning with Attention

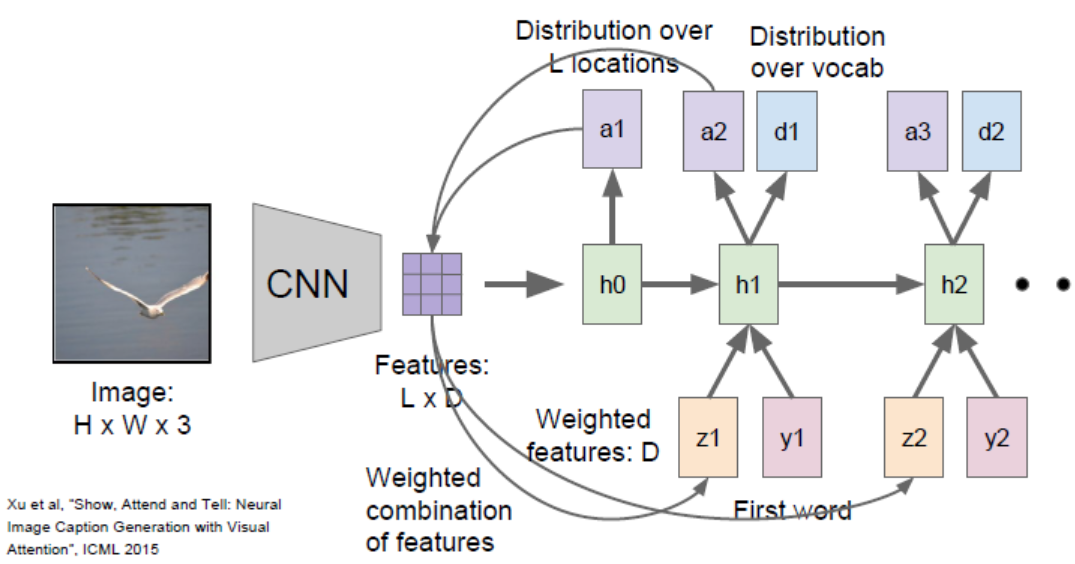


Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

Figure copyright Kelvin Xu, Jimmy Lei Ba, Jamie Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Benchio, 2015. Reproduced with permission.



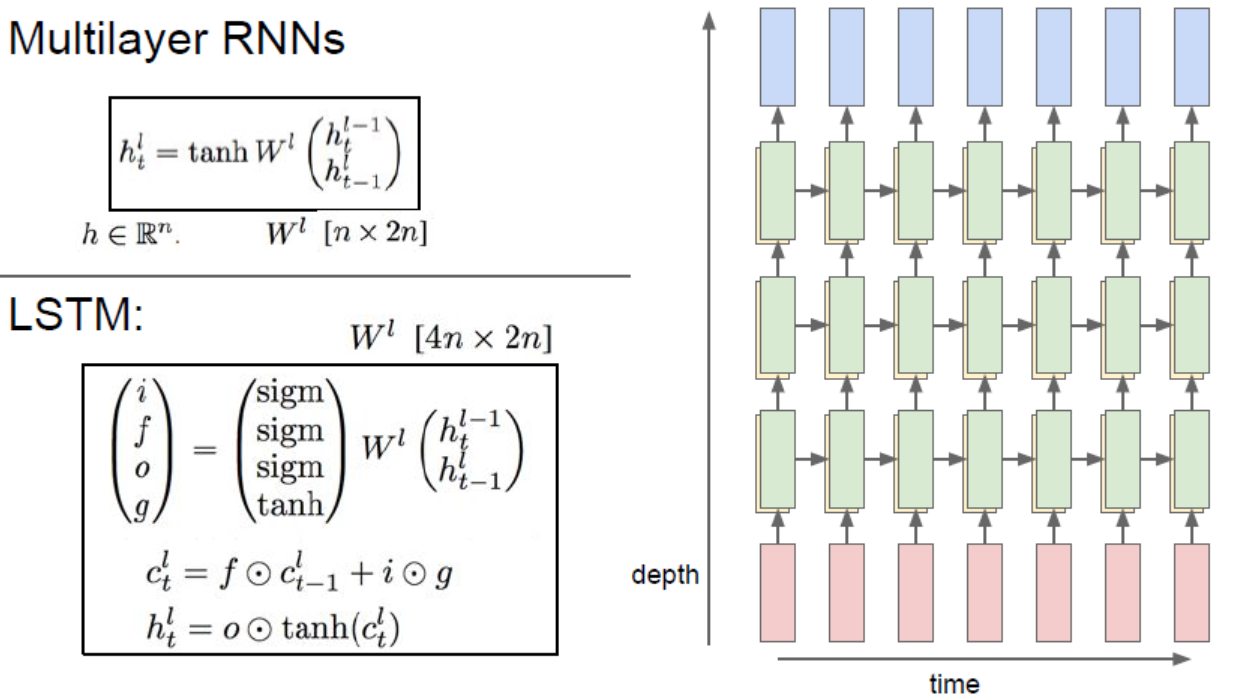




以上都是RNN的一些应用，

目前 的RNN都是一层Hidden layer, 实际上更多的是多层Hidden States Layers

如下 三层的RNN 网络：



一般而言，2-4层RNN已对够了

RNN的梯度流

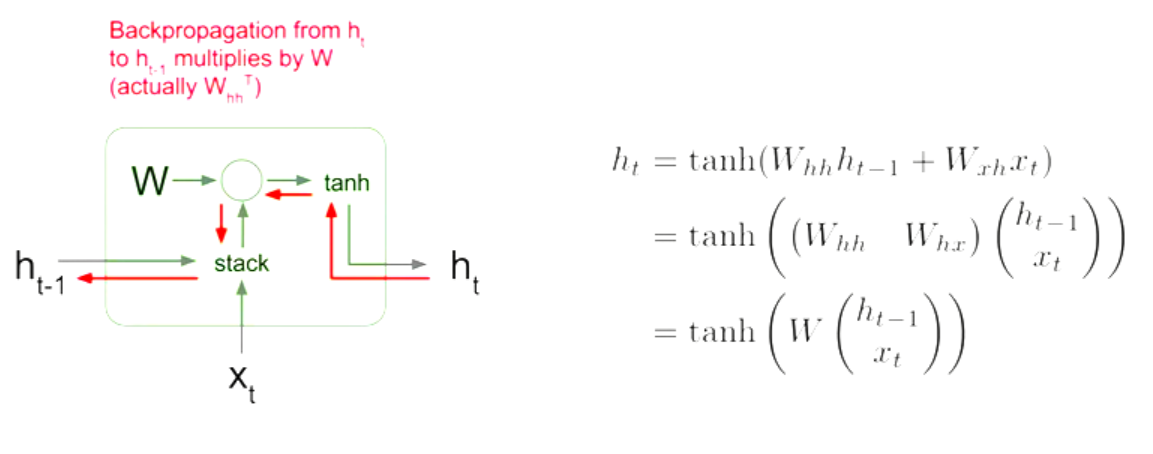
Vanilla RNN Gradient Flow

Bengio et al, “Learning long-term dependencies with gradient descent

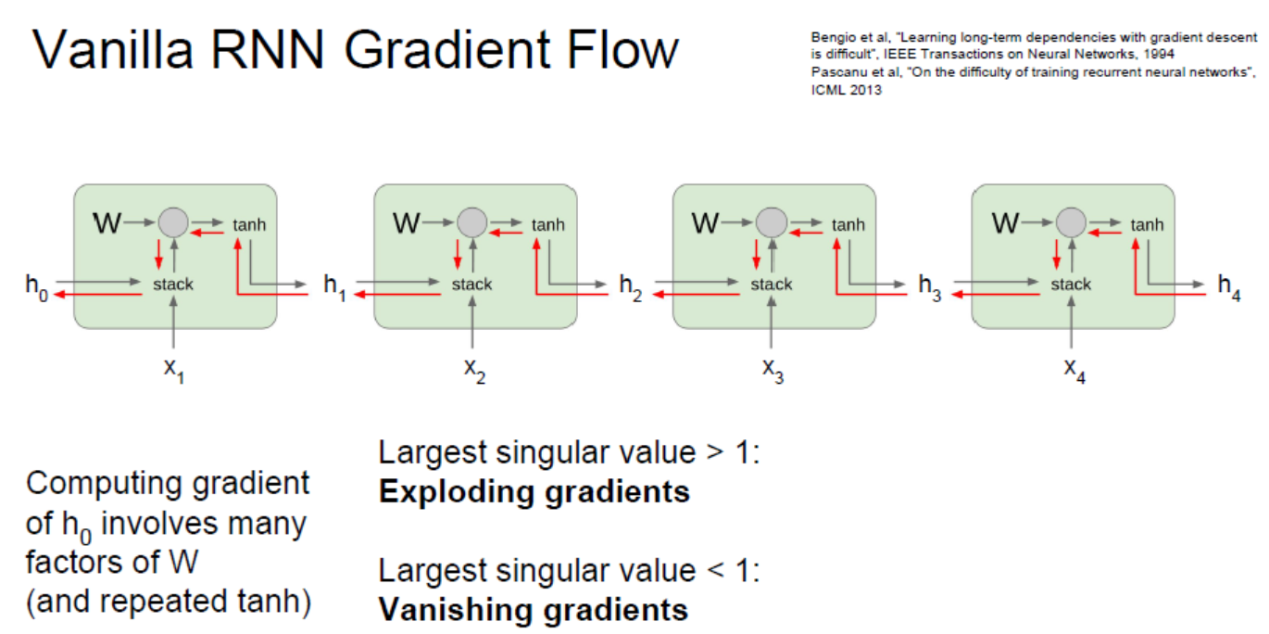
is difficult”, IEEE Transactions on Neural Networks, 1994

Pascanu et al, “On the difficulty of training recurrent neural networks”,

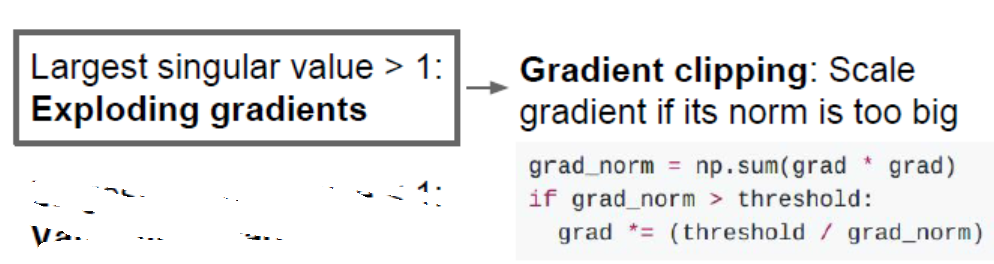
ICML 2013



当反射传播计算梯度时，要不断计算同一个W矩阵，不断相乘后，梯度，要么爆炸 要么消失



### 解决办法：梯度裁剪

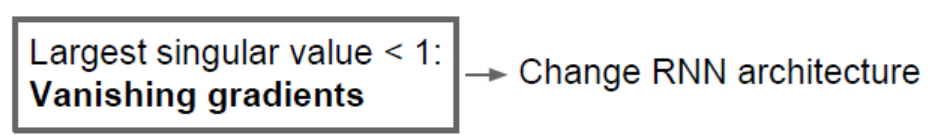


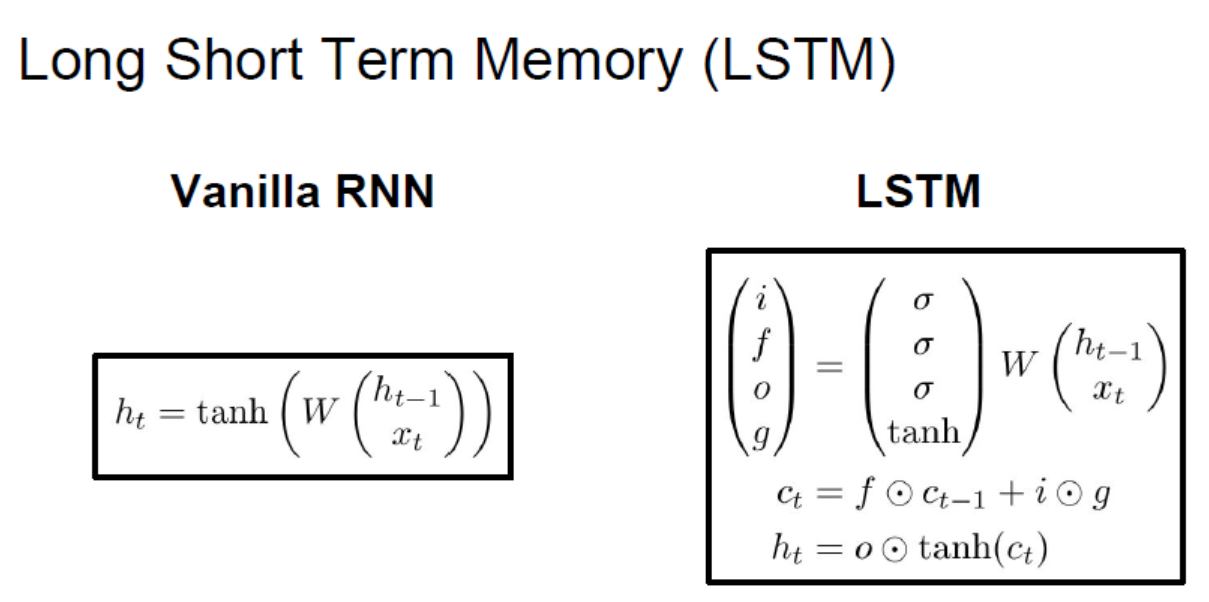
梯度到了一定程度不下降后就不再做计算

### 梯度消失：改变RNN的结构 -> LSTM (Long Short Term Memory)

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation

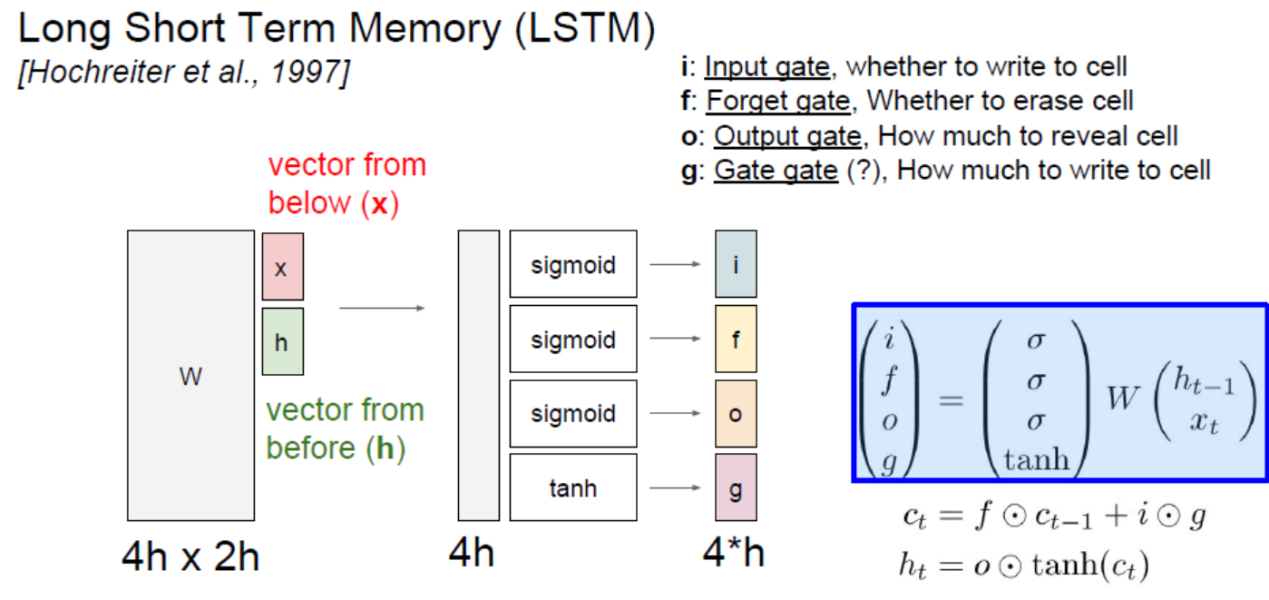
1997



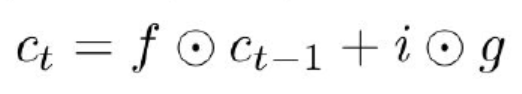


普通的RNN的一步只是保留一个hidden state, 而LSTM在过程中每一步保留两个 hidden state.

如上图一个是ht : hidden state ,一个是ct: cell state



注意这里的乘法是元素想乘，而不是矩阵乘法



首先 ,

f决定 c(t-1)的cell state是否传播到下一个cell state c(t) ,

i 决定 input是否传播给 下一个 cell state

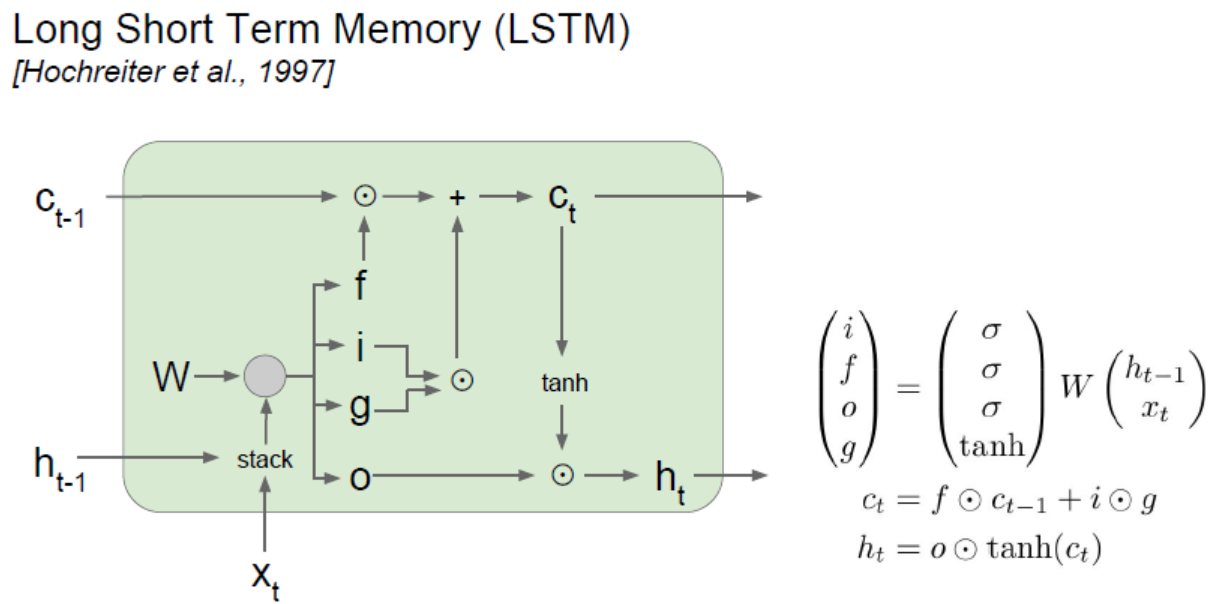
g 表示传输多少

这里 i,f,g 都是sigmoid，所以可以当成元素都是0或1



上一步计算得到的c(t)，带入计算最终的hidden state h(t)

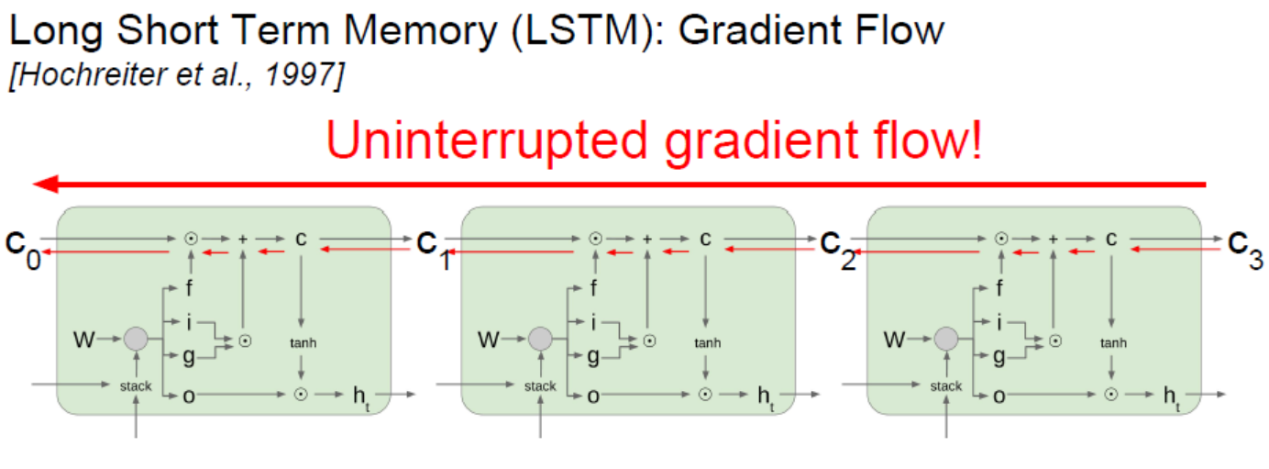
*[Hochreiter et al., 1997]*



这里比普通RNN优化的地方：

1. 这里是矩阵位置相乘，而不是矩阵乘法，计算量更小
2. 因为这里乘的 i,f,g 都是sigmoid， 不会出现不停乘小数或大数的情况，因为都是乘0或 1
3. 求导时，只有一个tanh，其他都是c(t)的式子，

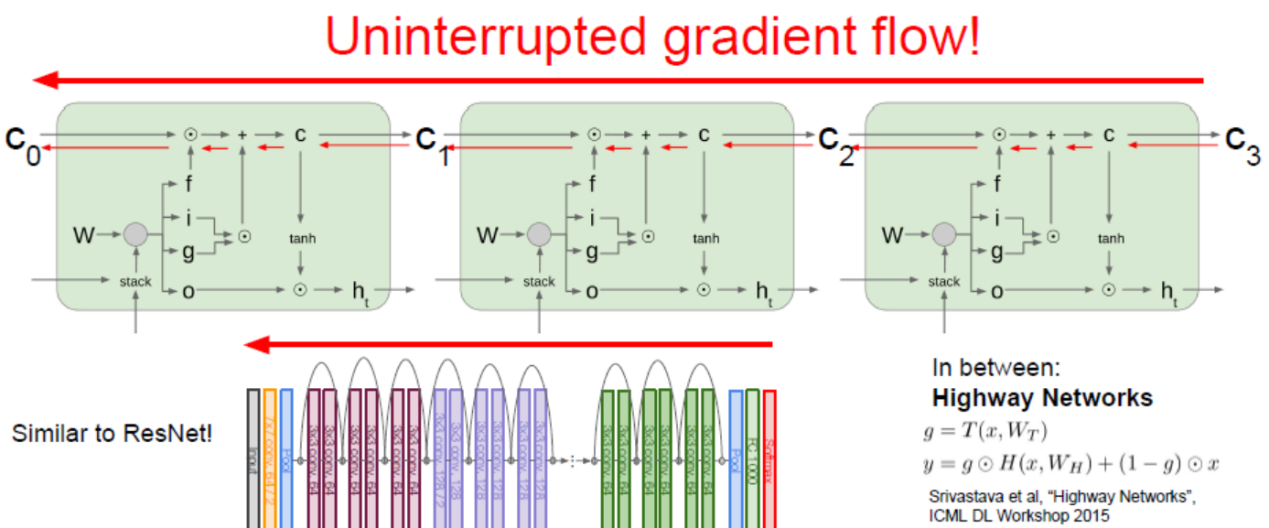
关于c的梯度：



W的梯度：

每一个步长都会保留

有时后所有的f都小于1，那么这样连续相乘也可以出现梯度消失，一般初始化f gate的biases为一些正数

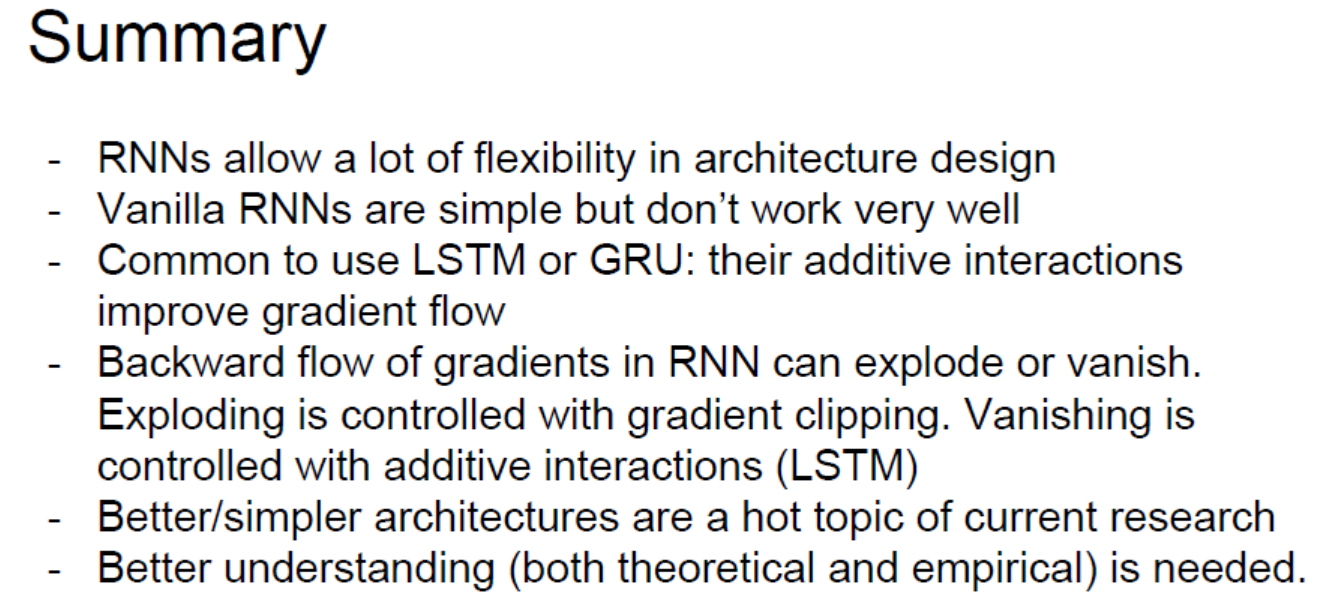


Srivastava et al, “Highway Networks”,

ICML DL Workshop 2015

推荐论文：

[*LSTM: A Search Space Odyssey*, Greff et al., 2015]

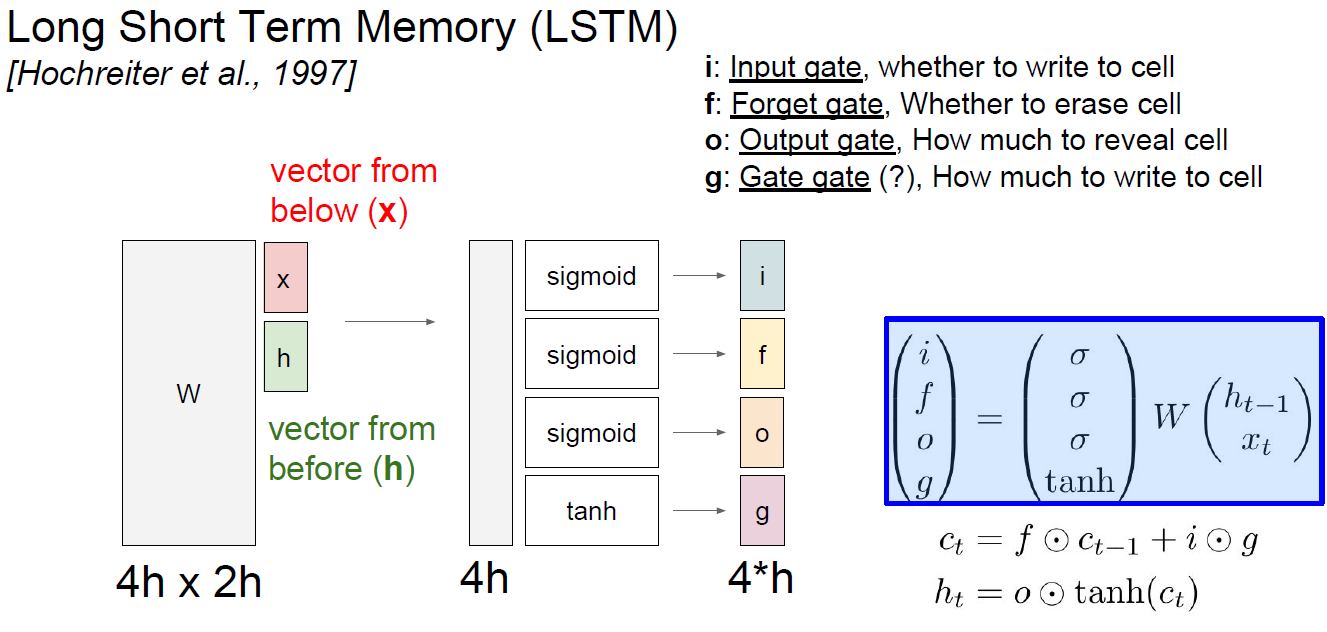


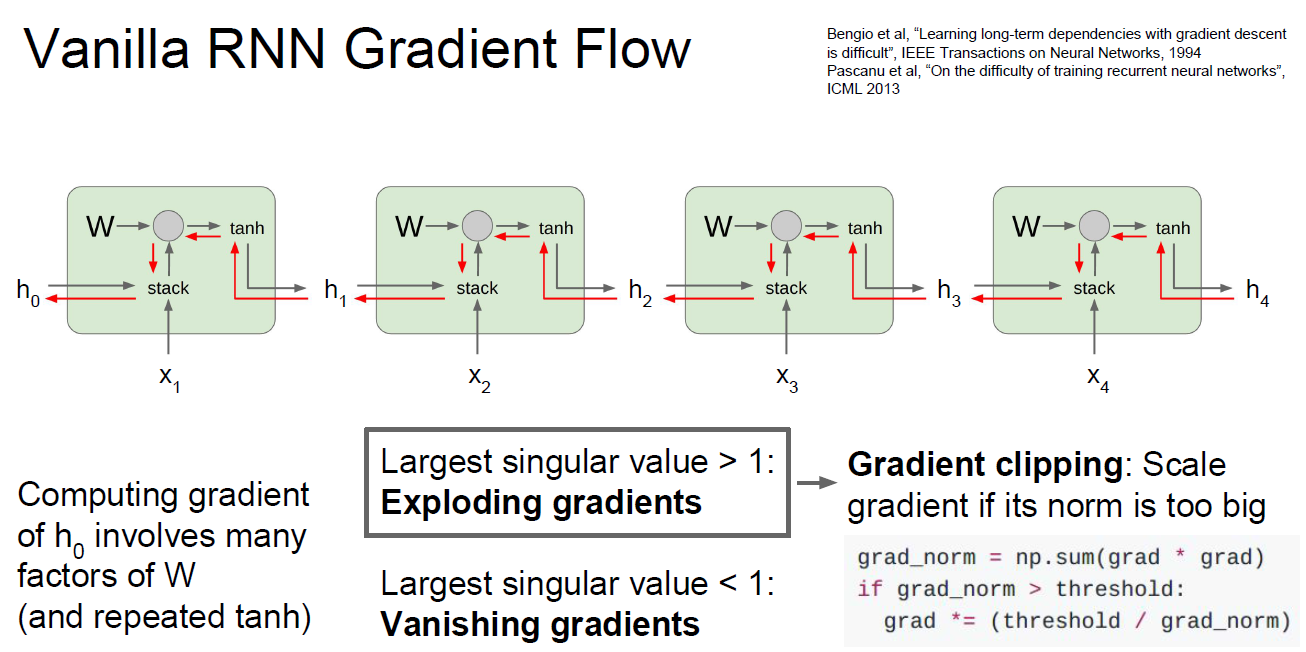
练习：assignment3

RNN Image Caption:

Layers structure:







对h和W的求导每一步都要乘一个W.T

dprev\_h = np.dot(dh\_raw, Wh.T)

dx = np.dot(dh\_raw, Wx.T)



x(input): (N,D) (1,D)

(hidden state) : (N,H) (1,H)

(cell state) : (N,H) (1,H)

 : (4H,D)

 : (4H,H)

计算：

传播：

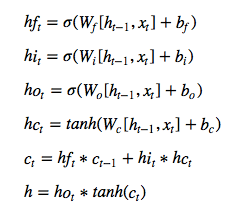
  ,向量a再 分为 ,从上到下都是

计算出不同的gate：

\*\*\*𝜎 is the sigmoid 都是按位相乘，不是矩阵乘法  ⊙表示对应位置相乘

或者也有这样的表示 ：



下面就可以计算cell state: 和 hidden state: 



