

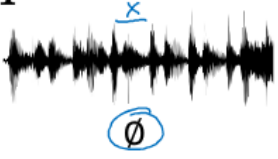

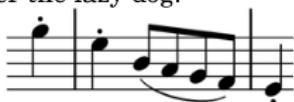


# 循环神经网络

本文尝试对循环神经网络 ( Recurrent Neural Network(RNN) ) 做一个总结，主要包括以下几部分内容：

- 1、循环神经网络的动机
- 2、基本循环神经网络原理及python实现
- 3、梯度消失与梯度爆炸
- 4、LSTM，GRU基本原理及python实现
- 5、训练循环神经网络的一些技巧
- 6、RNN 扩展
- 7、RNN的应用模式
- 8、RNN应用实例

## 循环神经网络的动机

### Examples of sequence data

Speech recognition	→ 	→ "The quick brown fox jumped over the lazy dog."
Music generation	→ 	→ 
Sentiment classification	"There is nothing to like in this movie."	→ 
DNA sequence analysis	→ AGCCCCTGTGAGGAACTAG	→ AG <b>CCCCTGTGAGGAACTAG</b>
Machine translation	Voulez-vous chanter avec moi?	→ Do you want to sing with me?
Video activity recognition		→ Running
Name entity recognition	→ Yesterday, Harry Potter met Hermione Granger.	→ Yesterday, <b>Harry Potter</b> met <b>Hermione Granger</b> . Andrew Ng

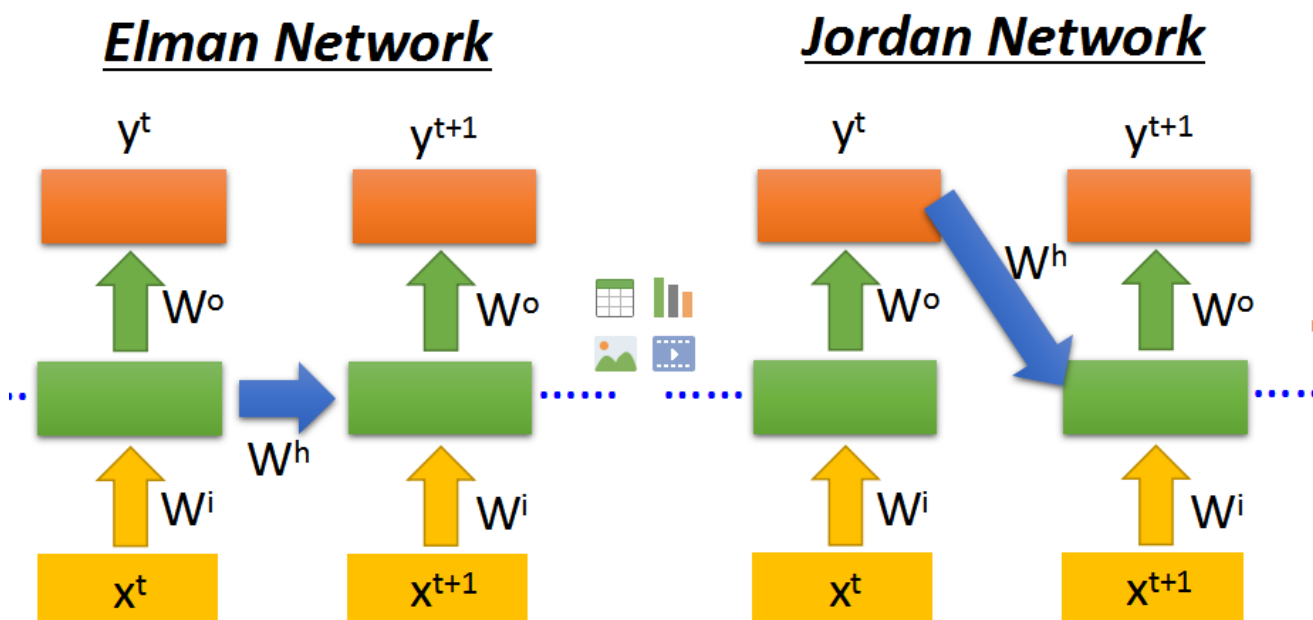
1、NLP的许多任务需要处理序列数据（不同的任务可以有不同的序列模式，sequence to category,synchronous sequence to sequence,Asynchronous sequence to sequence等），事实上，大部分时候即使同一个任务中，不同的样本，输入和输出的序列长度可能不相同，因此需要能够处理这种可变长度序列的模型。

2、与CNN动机相同，语言任务中，相同的特征，可能出现在序列的不同位置，比如，考虑这样两句话，“I went to Nepal in 2009”和“In 2009,I went to Nepal”，如果我们的任务是识别时间，那么无论2009无论出现在哪个位置，都应该能够识别出，因此为了捕获这些信息，需要进行权值共享，使得无论某些特征出现在哪个位置，我们都能够捕获。传统的全连接前馈神经网络会给每个输入特征分配一个单独的参数，需要分别学习每个位置的所有语言规则，事实上，语言本身也不需要每个位置都有不同的规则，因此，我们需要在每个时间点都有相同的转移函数 $f$ 。而且通过权值共享机制，可以有效降低VC维，提升泛化能力

3、词只有通过上下文才有完整的意义，许多时候词只有出现在上下文中才能正确表达出意义，因此需要模型在不同词之间发生互动，建立联系，相互传递信息，事实上许多时候需要建立长距离的联系（这也是许多NLP任务的主要困难）。

因此，基于上述动机，模型具有以下性质，每个时间点，都采用相同的模型，每个时间点的输入处理包含该时间点的输入，也需要包含其他时间点传递来的信息，为了简单起见，我们假设每个时间点只从其上一个时间点获得信息，然后再将该模型进行扩展。

循环神经网络主要包括以下两种：



如上图所示：

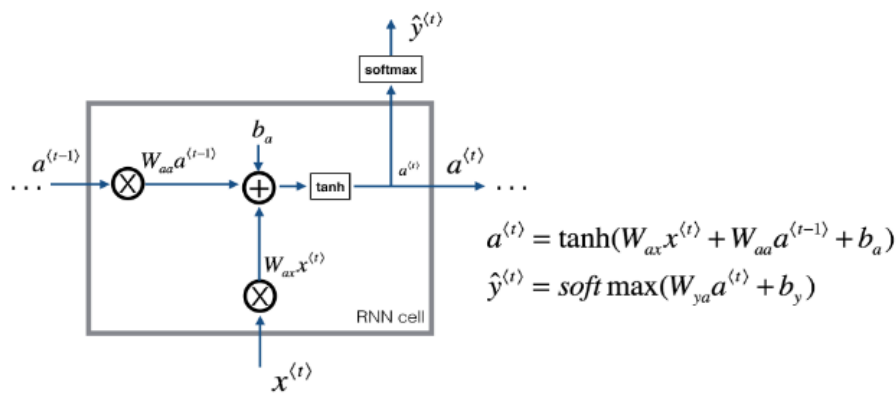
- 1) **Elman Network**：将上一时间点隐藏单元中信息作为输入传递给下一个时间点
- 2) **Jordan Network**：将上一时间点输出单元的信息作为输入传递到下一个时间点

Elman Network 相比 Jordan Network 要更强大，因为输入单元 $y^t$ 需要明确地训练成匹配训练集的目标，因此它不太可能捕捉到对于未来有用的过去的所有信息，会使得长期依赖问题更严重，因此循环神经网络一般将隐藏单元信息传递到下一个时间点作为输入。

## 循环神经网络的原理及python实现

### RNN正向传播

#### RNN-cell 正向传播



**Figure 2:** Basic RNN cell. Takes as input  $x^{(t)}$  (current input) and  $a^{(t-1)}$  (previous hidden state containing information from the past), and outputs  $a^{(t)}$  which is given to the next RNN cell and also used to predict  $y^{(t)}$

上图描述了Basic RNN单元内部细节，定义每个输入的维度为 $n_x$ ，隐藏单元 $a^t$ 的维度为 $n_a$ ，minibatch 大小为 $m$ ，因此每次输入的 $x^t$ 的维度为 $(n_x, m)$ ， $a^t$ 的维度为 $(n_a, m)$ ，因此 $W_{ax}$ 的维度应该为 $(n_a, n_x)$ ， $W_{aa}$ 的维度为 $(n_a, n_a)$ ， $b_a$ 的维度为 $(n_a, 1)$ ，输入 $\hat{y}^t$ 的维度为 $n_y$ ，那么 $W_{ya}$ 的维度 $(n_y, n_a)$ ， $b_y$ 的维度为 $(n_y, 1)$ 。每一次输入为前一时间点隐藏层输出 $a^{t-1}$ 和单前时间点的输入 $x^t$ 。输出为新的隐藏层状态 $a^t$ 和输出 $y^t$ 。

下面使用python 实现Basic RNN cell的前向传播：

```
GRADED FUNCTION: rnn_cell_forward
def rnn_cell_forward(xt, a_prev, parameters):
    """
    Implements a single forward step of the RNN-cell as described in Figure (2)

    Arguments:
    xt -- your input data at timestep "t", numpy array of shape (n_x, m).
    a_prev -- Hidden state at timestep "t-1", numpy array of shape (n_a, m)
    parameters -- python dictionary containing:
                    Wax -- Weight matrix multiplying the input, numpy array of shape (n_a, n_x)
                    Waa -- Weight matrix multiplying the hidden state, numpy array of shape
                        (n_a, n_a)
                    Wya -- Weight matrix relating the hidden-state to the output, numpy array of
                        shape (n_y, n_a)
                    ba -- Bias, numpy array of shape (n_a, 1)
                    by -- Bias relating the hidden-state to the output, numpy array of shape
                        (n_y, 1)

    Returns:
    a_next -- next hidden state, of shape (n_a, m)
    yt_pred -- prediction at timestep "t", numpy array of shape (n_y, m)
    cache -- tuple of values needed for the backward pass, contains (a_next, a_prev, xt, parameters)
    """

    # Retrieve parameters from "parameters"
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    ### START CODE HERE ### (~2 lines)
```

```

# compute next activation state using the formula given above
a_next=np.tanh(np.dot(Wax,xt)+np.dot(Waa,a_prev)+ba)
# compute output of the current cell using the formula given above
yt_pred = softmax(np.dot(Wya,a_next)+by)
### END CODE HERE ###

# store values you need for backward propagation in cache
cache = (a_next, a_prev, xt, parameters)

return a_next, yt_pred, cache

```

## RNN-model正向传播

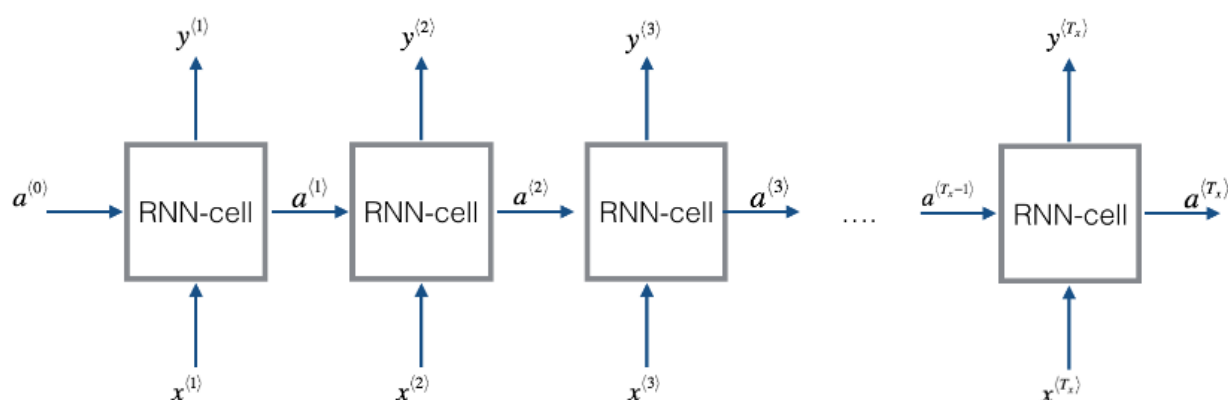


Figure 1: Basic RNN model

RNN 可以看作是RNN-cell在时间上的重复，如果序列长度为10，那么就将RNN-cell拷贝10次（可以使用任意输入长度序列），每一次输入为前一时间点隐藏层输出  $a^{t-1}$  和单前时间点的输入  $x^t$ 。输出为新的隐藏层状态  $a^t$  和输出  $y^t$ 。

```

def rnn_forward(x, a0, parameters):
    """
    Implement the forward propagation of the recurrent neural network described in Figure (3).

    Arguments:
    x -- Input data for every time-step, of shape (n_x, m, T_x).
    a0 -- Initial hidden state, of shape (n_a, m)
    parameters -- python dictionary containing:
        Waa -- Weight matrix multiplying the hidden state, numpy array of shape
        (n_a, n_a)
        Wax -- Weight matrix multiplying the input, numpy array of shape (n_a, n_x)
        Wya -- Weight matrix relating the hidden-state to the output, numpy array of
        shape (n_y, n_a)
        ba -- Bias numpy array of shape (n_a, 1)
        by -- Bias relating the hidden-state to the output, numpy array of shape
        (n_y, 1)

    Returns:
    a -- Hidden states for every time-step, numpy array of shape (n_a, m, T_x)
    y_pred -- Predictions for every time-step, numpy array of shape (n_y, m, T_x)
    caches -- tuple of values needed for the backward pass, contains (list of caches, x)
    """

```

```

"""

# Initialize "caches" which will contain the list of all caches
caches = []

# Retrieve dimensions from shapes of x and parameters["Wya"]
n_x, m, T_x = x.shape
n_y, n_a = parameters["Wya"].shape

### START CODE HERE ###

# initialize "a" and "y" with zeros (≈2 lines)
a = np.zeros(shape=[n_a,m,T_x])
y_pred = np.zeros(shape=[n_y,m,T_x])

# Initialize a_next (≈1 line)
a_next = a0

# loop over all time-steps
for t in range(T_x):
    # Update next hidden state, compute the prediction, get the cache (≈1 line)
    a_next, yt_pred, cache = rnn_cell_forward(x[:, :, t], a_next, parameters)
    # Save the value of the new "next" hidden state in a (≈1 line)
    a[:, :, t] = a_next
    # Save the value of the prediction in y (≈1 line)
    y_pred[:, :, t] = yt_pred
    # Append "cache" to "caches" (≈1 line)
    caches.append(cache)

### END CODE HERE ###

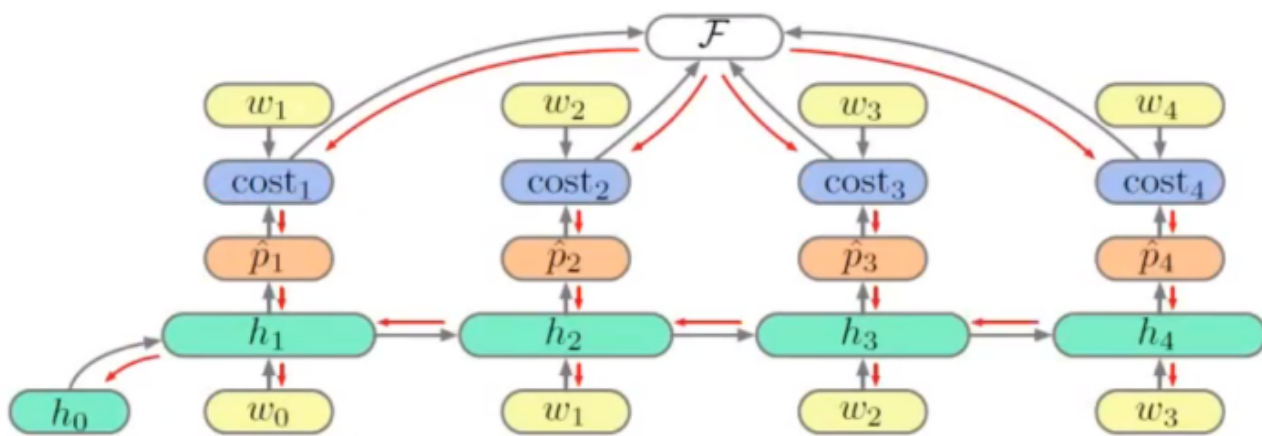
# store values needed for backward propagation in cache
caches = (caches, x)

return a, y_pred, caches

```

## RNN的反向传播

RNN的反向传播也采用BP算法，但是由于每个时间点都接受前一个时间点隐藏层的输出作为输入，并且在该时间点也包含输出，因此，每一个时间点的梯度除了来自该时间点输出产生的梯度，还包含下一时间点传递回来的梯度。



为了方便推导，做一些标记：

$$\text{定义 } \theta^t = W_{ya} a^t + b_y$$

$$\text{则 } y^t = \text{softmax}(\theta^t)$$

假设进行分类任务，损失函数为交叉熵：

$$t\text{时刻损失函数：交叉熵 } J^t = - \sum_{j=1}^{|V|} y_j^t \log \hat{y}_j^t$$

$$\text{总的损失函数为 } J = \frac{1}{T} \sum_{t=1}^T J^t$$

$$J^t = -\log \hat{y}_k^t = -\log\left(\frac{\exp(\theta_k^t)}{\sum_{c=1}^C \exp(\theta_c^t)}\right) = \log(\sum_{c=1}^C \exp(\theta_c^t)) - \theta_k^t$$

$$\frac{\alpha \log(\sum_{c=1}^C \exp(\theta_c^t))}{\theta_j^t} = \frac{\exp(\theta_j^t)}{\sum_{c=1}^C \exp(\theta_c^t)} = \hat{y}_j^t$$

$$\text{从而有 } \frac{\alpha J^t}{\alpha \theta_j^t} = \begin{cases} \hat{y}_j^t - 1, j = k \\ \hat{y}_j^t, j \neq k \end{cases} = \begin{cases} \hat{y}_j^t - y_j^t, j = k \\ \hat{y}_j^t - y_j^t, j \neq k \end{cases}$$

$$\frac{\alpha J^t}{\alpha \theta^t} = \hat{y}^t - y^t = \delta_1$$

$$\frac{\alpha J^t}{\alpha b_y} = \frac{\alpha J^t}{\alpha \theta^t} = \delta_1$$

$$\theta_j^t = \sum_{k=1}^{n_a} W_{ya_{jk}} a_k^t$$

$$\frac{\alpha \theta_j^t}{\alpha W_{ya_{jk}}} = a_k^t$$

$W_{ya_{jk}}$  只与  $\theta^t$  中的  $\theta_j^t$  有关

$$\frac{\alpha \theta^t}{\alpha W_{ya_{jk}}} = \frac{\alpha \theta_j^t}{\alpha W_{ya_{jk}}} = a_k^t$$

$$\text{从而有 } \frac{\alpha J^t}{\alpha W_{ya_{jk}}} = \frac{\alpha J^t}{\alpha \theta_j^t} \frac{\alpha \theta_j^t}{\alpha W_{ya_{jk}}} = (\hat{y}_j^t - y_j^t) a_k^t$$

如果只考虑一个样本， $y^t$  与  $a^t$  均为列向量

$$\text{从而有 } \frac{\alpha J^t}{\alpha W_{ya}} = \delta_1 (a^t)^T$$

$$\frac{\alpha J^t}{\alpha a_k^t} = \sum_{j=1}^{n_y} \frac{\alpha J^t}{\alpha \theta_j^t} \frac{\alpha \theta_j^t}{\alpha a_k^t} = \sum_{j=1}^{n_y} W_{ya_{jk}} (\hat{y}_j^t - y_j^t)$$

从而有  $\frac{\alpha J^t}{\alpha a^t} = W_{ya}^T \delta_1$

通过上边的推导，可以得出，当矩阵与列向量相乘时，假设  $U=Mv+b$  (M为矩阵，b,v为列向量)

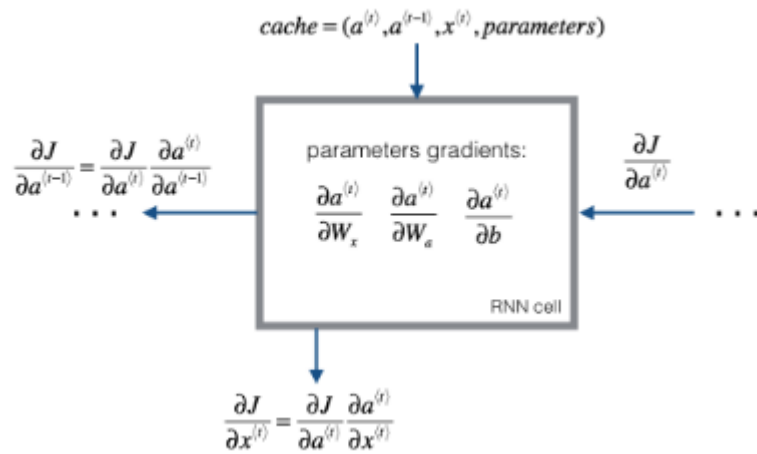
则有： $\frac{\alpha L}{\alpha v} = M^T \frac{\alpha L}{\alpha U}$ ， $\frac{\alpha L}{\alpha M} = \frac{\alpha L}{\alpha U} v^T$

(需要说明的是在深度学习框架中，一般是行向量与矩阵相乘，此时  $U=VM+b$  (U,V为行向量，M为矩阵，b为行向量)

那么： $\frac{\alpha L}{\alpha V} = \frac{\alpha L}{\alpha U} * M^T$ ， $\frac{\alpha L}{\alpha M} = V^T \frac{\alpha L}{\alpha U}$ ，推导与上述相同，这里就不赘述)

## RNN-cell的反向传播

通过上边的推导得出： $\frac{\alpha J^t}{\alpha a^t} = W_{ya}^T (\hat{y}^t - y^t)$



为了实现BPTT算法，我们需要计算  $J^t$  关于  $W_{ax}, W_{aa}, b_a, a^{t-1}, x^t$  等

为了方便，定义  $h^t = W_{ax}x^t + W_{aa}a^{t-1} + b_a$

那么  $a^t = \tanh(h^t)$

$$\frac{\alpha a^t}{\alpha h^t} = 1 - (a^t)^2$$

从而有  $\frac{\alpha J^t}{\alpha W_{ax}} = \frac{\alpha J^t}{\alpha a^t} \frac{\alpha a^t}{\alpha h^t} \frac{\alpha h^t}{\alpha W_{ax}} = \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2) (x^t)^T$

$$\frac{\alpha J^t}{\alpha W_{aa}} = \frac{\alpha J^t}{\alpha a^t} \frac{\alpha a^t}{\alpha h^t} \frac{\alpha h^t}{\alpha W_{aa}} = \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2) (a^{t-1})^T$$

$$\frac{\alpha J^t}{\alpha x^t} = \frac{\alpha J^t}{\alpha a^t} \frac{\alpha a^t}{\alpha h^t} \frac{\alpha h^t}{\alpha x^t} = W_{ax}^T \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2)$$

$$\frac{\alpha J^t}{\alpha a^{t-1}} = \frac{\alpha J^t}{\alpha a^t} \frac{\alpha a^t}{\alpha h^t} \frac{\alpha h^t}{\alpha a^{t-1}} = W_{aa}^T \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2)$$

$$\frac{\alpha J^t}{\alpha b_a} = \frac{\alpha J^t}{\alpha a^t} \frac{\alpha a^t}{\alpha h^t} \frac{\alpha h^t}{\alpha b_a} = \sum_{batch} \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2)$$

根据上述公式，使用python实现

```
def rnn_cell_backward(da_next, cache):
    """
    Implements the backward pass for the RNN-cell (single time-step).

    Arguments:
    da_next -- Gradient of loss with respect to next hidden state
    cache -- python dictionary containing useful values (output of rnn_cell_forward())
```

```

Returns:
gradients -- python dictionary containing:
    dx -- Gradients of input data, of shape (n_x, m)
    da_prev -- Gradients of previous hidden state, of shape (n_a, m)
    dWax -- Gradients of input-to-hidden weights, of shape (n_a, n_x)
    dWaa -- Gradients of hidden-to-hidden weights, of shape (n_a, n_a)
    dba -- Gradients of bias vector, of shape (n_a, 1)
"""

# Retrieve values from cache
(a_next, a_prev, xt, parameters) = cache

# Retrieve values from parameters
Wax = parameters["Wax"]
Waa = parameters["Waa"]
Wya = parameters["Wya"]
ba = parameters["ba"]
by = parameters["by"]

### START CODE HERE ###
# compute the gradient of tanh with respect to a_next (~1 line)
dtanh = np.multiply(da_next, (1 - np.power(a_next, 2)))

# compute the gradient of the loss with respect to Wax (~2 lines)
dxt = np.dot(Wax.T, dtanh)
dWax = np.dot(dtanh, xt.T)

# compute the gradient with respect to Waa (~2 lines)
da_prev = np.dot(Waa.T, dtanh)
dWaa = np.dot(dtanh, a_prev.T)

# compute the gradient with respect to b (~1 line)
dba = np.sum(dtanh, axis=1)

### END CODE HERE ###

# Store the gradients in a python dictionary
gradients = {"dxt": dxt, "da_prev": da_prev, "dWax": dWax, "dWaa": dWaa, "dba": dba}

return gradients

```

获得  $\frac{\alpha J^t}{\alpha a^{t-1}} = W_{aa}^T \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2)$ , 按照相同的方式向前传播即可

```

def rnn_backward(da, caches):
    """
    Implement the backward pass for a RNN over an entire sequence of input data.

    Arguments:
    da -- Upstream gradients of all hidden states, of shape (n_a, m, T_x)
    caches -- tuple containing information from the forward pass (rnn_forward)

    Returns:
    gradients -- python dictionary containing:

```



```

dx -- Gradient w.r.t. the input data, numpy-array of shape (n_x, m, T_x)
da0 -- Gradient w.r.t the initial hidden state, numpy-array of shape (n_a,
m)

dWax -- Gradient w.r.t the input's weight matrix, numpy-array of shape (n_a,
n_x)

dWaa -- Gradient w.r.t the hidden state's weight matrix, numpy-array of shape
(n_a, n_a)

dba -- Gradient w.r.t the bias, of shape (n_a, 1)
"""

### START CODE HERE ###

# Retrieve values from the first cache (t=1) of caches (≈2 lines)
(caches, x) = caches
#获取参数
(a1, a0, x1, parameters) = caches[0]

# Retrieve dimensions from da's and x1's shapes (≈2 lines)
n_a, m, T_x = da.shape
n_x, m = x1.shape

# initialize the gradients with the right sizes (≈6 lines)
dx = np.zeros(shape=[n_x,m,T_x])
dWax = np.zeros(shape=[n_a,n_x])
dWaa = np.zeros(shape=[n_a,n_a])
dba = np.zeros(shape=[n_a,1])
da0 = np.zeros(shape=[n_a,m])
da_prevt = np.zeros(shape=[n_a,m])

# Loop through all the time steps
for t in reversed(range(T_x)):
    # Compute gradients at time step t. Choose wisely the "da_next" and the "cache" to use in
    the backward propagation step. (≈1 line)
    gradients = rnn_cell_backward(da[:, :, t] + da_prevt, caches[t])
    # Retrieve derivatives from gradients (≈ 1 line)
    dxt, da_prevt, dWaxt, dWaat, dbat = gradients["dxt"], gradients["da_prev"],
    gradients["dWax"], gradients["dWaa"], gradients["dba"]
    # Increment global derivatives w.r.t parameters by adding their derivative at time-step t
    (≈4 lines)
    dx[:, :, t] = dxt
    dWax += dWaxt
    dWaa += dWaat
    dba += dbat[:, np.newaxis]

# Set da0 to the gradient of a which has been backpropagated through all time-steps (≈1 line)
da0 = da_prevt
### END CODE HERE ###

# Store the gradients in a python dictionary
gradients = {"dx": dx, "da0": da0, "dWax": dWax, "dWaa": dWaa, "dba": dba}

return gradients

```

## 梯度消失与梯度爆炸

通过前一节推导可知：

$$\frac{\alpha J^t}{\alpha a^{t-1}} = W_{aa}^T \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2)$$

$$\text{同样的：} \frac{\alpha J^t}{\alpha a^{t-2}} = W_{aa}^T \frac{\alpha J^t}{\alpha a^{t-1}} (1 - (a^{t-1})^2) = W_{aa}^T (W_{aa}^T \frac{\alpha J^t}{\alpha a^t} (1 - (a^t)^2)) (1 - (a^{t-1})^2)$$

通过上述推导可以看到，通过隐藏层梯度向前传播时，每向前传递一个时间点，需要在原来的梯度上，前边乘上  $W_{aa}$ ，后边乘上  $1 - (a^{t-i})^2$ ，如果计算  $a^t$  时，使用sigmoid函数，那么需要乘上  $a^{t-i}(1 - a^{t-i})$

根据线性代数相关知识，可以将W进行分解  $W = S^{-1} \Lambda S$  (S为W的特征向量组成的矩阵)

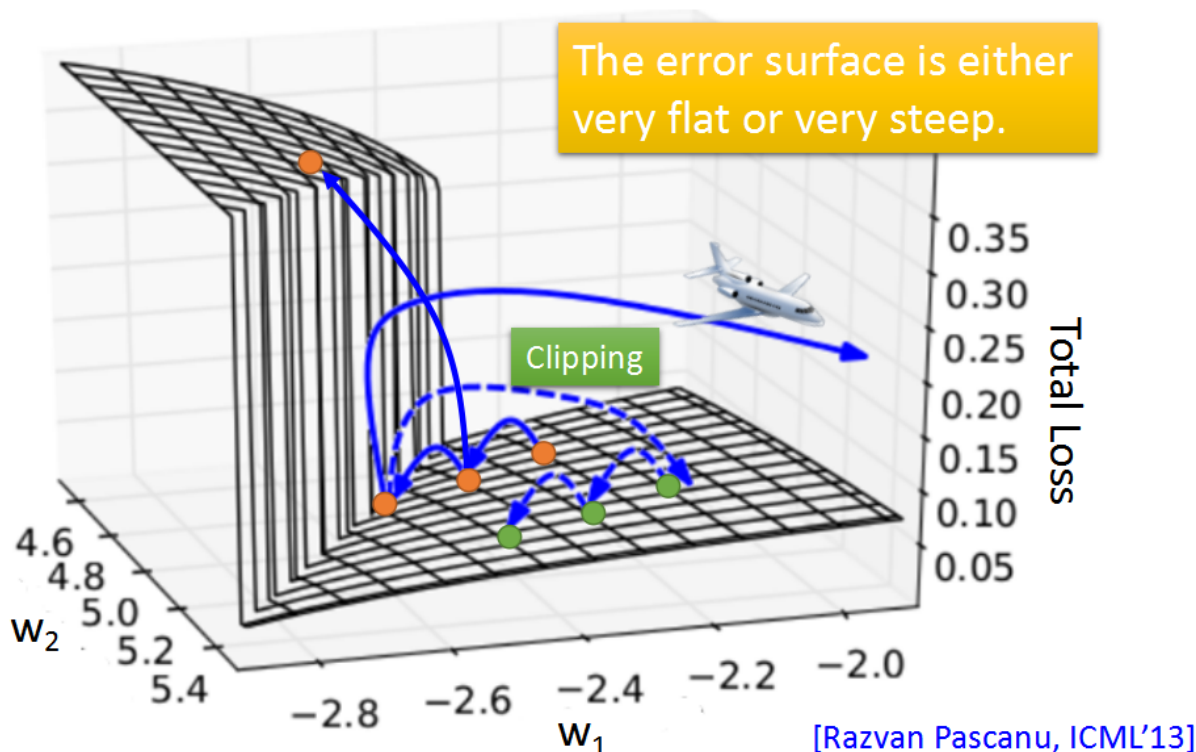
$$\text{那么 } W^k = S^{-1} \Lambda^k S$$

如果当W的特征值小于1,当k很大时， $W^k$ 趋向于0，即传递的梯度值会趋向于0，发生梯度消失问题，难以知道参数朝那个方向移动从而改进损失函数。

如果当W的特征值大于1,那么k很大时， $W^k$ 会迅速膨胀，传递的梯度值会损失增大，发生梯度爆炸问题，会使得学习变得不稳定。

如下图所示，损失函数关于参数W要么非常平坦，要么非常非常陡峭。

# The error surface is rough.



梯度爆炸问题，容易发现，也比较容易处理，进行简单的梯度裁剪即可，即如果某个梯度值大于某个阈值，那么将梯度修改为该阈值。

python中具体实现如下：

```
GRADED FUNCTION: clip
```

```
def clip(gradients, maxValue):
    '''
    Clips the gradients' values between minimum and maximum.

    Arguments:
    gradients -- a dictionary containing the gradients "dWaa", "dWax", "dWya", "db", "dby"
    maxValue -- everything above this number is set to this number, and everything less than -
    maxValue is set to -maxValue

    Returns:
    gradients -- a dictionary with the clipped gradients.
    '''

    dWaa, dWax, dWya, db, dby = gradients['dWaa'], gradients['dWax'], gradients['dWya'],
    gradients['db'], gradients['dby']

    ### START CODE HERE ###
    # clip to mitigate exploding gradients, loop over [dWax, dWaa, dWya, db, dby]. (≈2 lines)
    for gradient in [dWax, dWaa, dWya, db, dby]:
        np.clip(gradient, -maxValue, maxValue, out=gradient)
    ### END CODE HERE ###

    gradients = {"dWaa": dWaa, "dWax": dWax, "dWya": dWya, "db": db, "dby": dby}

    return gradients
```

关于梯度消失还有一个需要注意的，根据上边的推导，还需要乘上关于激活函数的导数，如果采用sigmoid或者tanh函数会使梯度消失问题变得更严重。

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \in [0, 0.25]$$

$$\tanh'(x) = 1 - (\tanh(x))^2 \in [0, 1]$$

梯度消失问题会使得正向传播时，多次矩阵相乘和激活函数的影响，使得前边时间点的信息无法向后很好的传播，梯度也很难向前传播，即前边时间点的信息与后边时间点的信息无法发生互动，即会出现长期依赖问题。但是NLP中许多任务，比如机器翻译，指代消解等任务，都需要比较长的信息。

为了解决梯度消失问题，可以采用更好的激活函数，更好的初始化方法，更好的网络架构（LSTM，GRU）等。因此，接下来说明LSTM，GPU等门控RNN。

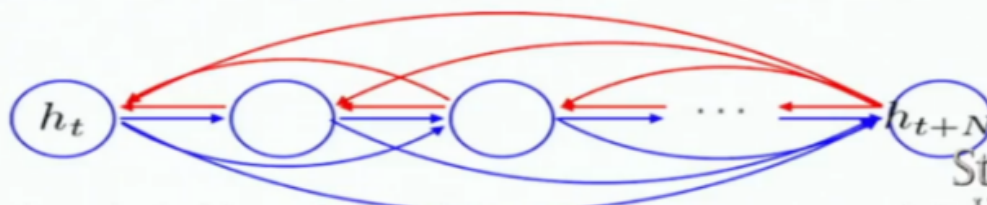
## LSTM与GRU基本精神，基本原理与python实现

LSTM与GRU等门控单元与卷积神经网络中ResNet精神基本一致，即采用skip-connection思想，直接获得更早时间的输入对现在和将来的影响，而不用经过很长的矩阵相乘，从而使信息可以传出去，梯度可以传回来。

It implies that the error must backpropagate through all the intermediate nodes:



Perhaps we can create shortcut connections.

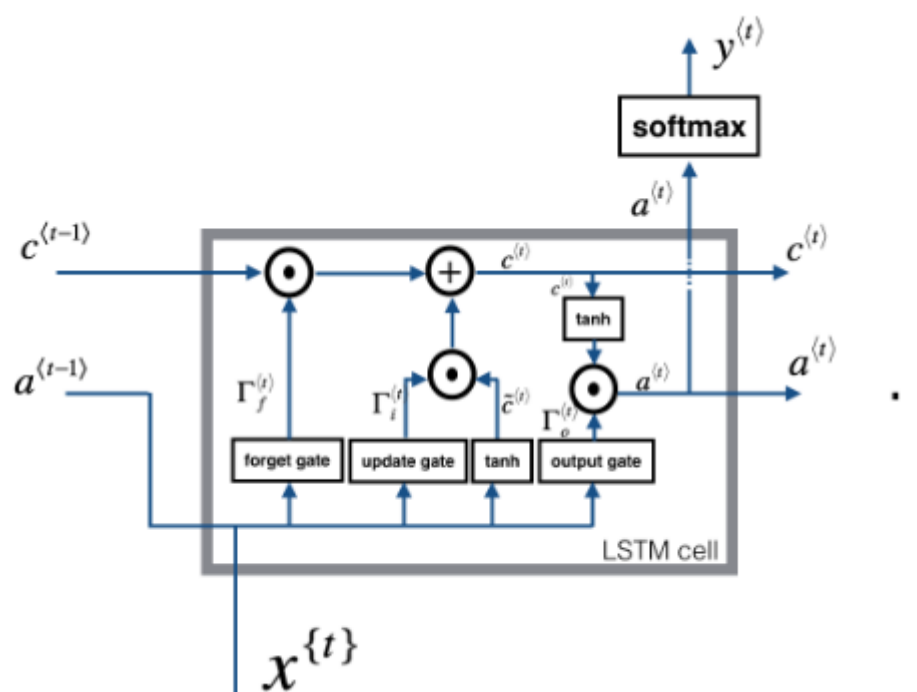


## LSTM(long short term memory)长短期记忆单元

LSTM 由Hochreiter & Schmidhuber (1997)提出,并在近期被Alex Graves进行了改良和推广。在很多问题，LSTM都取得相当巨大的成功，极大的推动了深度学习在NLP中的应用，事实上可以说 Bi-direction LSTM加Attention统治了NLP。

CS224n课程中Chris Manning在课程中演示了，普通的RNN由于矩阵相乘，梯度会迅速消失（5-6步），而LSTM则大概可以传递大约100步左右梯度才会消失。

下面开始介绍LSTM的基本原理：



如上图所示，LSTM单元，接收3个输入，当前时间点的输入 $x^t$ ，上一层的隐藏层输出 $a^{t-1}$ ，上一层memory输出 $c^{t-1}$ ，输出也包含三个部分，当前时间节点的输出 $y^t$ （可能不输出），当前时间节点隐藏层： $a^t$ ，当前时间节点memory： $c^t$ 。 $c^t$ 与 $a^t$ 扮演不同的角色， $c^t$ 用来存储历史信息，只是将当前时间点信息加到memory中，（事实上正是这个加法，使得LSTM可以有效记忆long short memory）变化相对不大， $a^t$ 是继承自基本的RNN，与当前时间点的输出有关，变化相对比较剧烈。

LSTM具体实现如下：

**potential hidden** :  $\tilde{a}^t = \tanh(W_a[a^{t-1}, x^t] + b_a)$  继承自基本的RNN

**forget gate** :  $f^t = \sigma(W_f[a^{t-1}, x^t] + b_f)$ ，与上一时间点memory中信息相乘，实现了对过去信息遗忘的能力（忘记过去不重要信息）。如果forget\_gate 某一维度为0，则代表遗忘，为1，则表示全部继承。

**update gate** :  $u^t = \sigma(W_u[a^{t-1}, x^t] + b_u)$

**output gate** :  $o^t = \sigma(W_o[a^{t-1}, x^t] + b_o)$

**memory updata** :  $c^t = f^t \cdot c^{t-1} + u^t \cdot \tilde{a}^t$

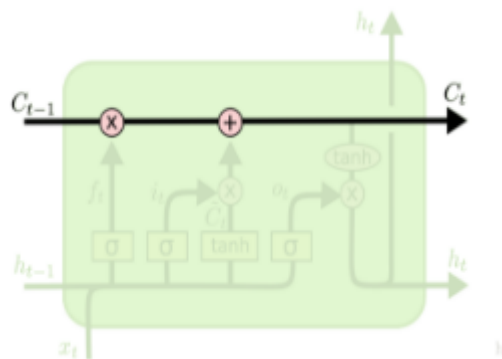
**hidden updata** :  $a^t = o^t \cdot \tanh(c^t)$

forget gate,update gate,output gate 均使用sigmoid作为激活函数，将输出限制在0-1之间，实现了某种称作"门"的结构，实现信息选择通过，从而实现去除或者增加信息到细胞状态的能力。

forget gate 与上一时间点memory中信息 $c^{t-1}$ 实现元素相乘，实现了对过去信息遗忘的能力（忘记过去不重要信息）。如果forget\_gate 某一维度为0，则代表遗忘，为1，则表示全部继承。

update gate 与potential hidden元素相乘，决定了如何将当前时间点信息写入到memory  $c^t$ 中，实现了对当前重要信息进行筛选的机制。

LSTM 的关键就是memory updata，水平线在图上方贯穿运行。类似于传送带。直接在整个链上运行，只有一些少量的线性交互。信息在上面流传保持不变会很容易，从而记住long short memory称为可能。



output gate 决定了当前节点的输出

## LSTM python 实现

LSTM cell 正向传播：

```
def lstm_cell_forward(xt, a_prev, c_prev, parameters):
    """
    Implement a single forward step of the LSTM-cell as described in Figure (4)

    Arguments:
    xt -- your input data at timestep "t", numpy array of shape (n_x, m)
    a_prev -- Hidden state at timestep "t-1", numpy array of shape (n_a, m)
    c_prev -- Memory state at timestep "t-1", numpy array of shape (n_a, m)
    parameters -- python dictionary containing:

                Wf -- Weight matrix of the forget gate, numpy array of shape (n_a, n_a +
```

```

n_x)
    bf -- Bias of the forget gate, numpy array of shape (n_a, 1)
    Wi -- Weight matrix of the update gate, numpy array of shape (n_a, n_a +
n_x)
    bi -- Bias of the update gate, numpy array of shape (n_a, 1)
    Wc -- Weight matrix of the first "tanh", numpy array of shape (n_a, n_a +
n_x)
    bc -- Bias of the first "tanh", numpy array of shape (n_a, 1)
    Wo -- Weight matrix of the output gate, numpy array of shape (n_a, n_a +
n_x)
    bo -- Bias of the output gate, numpy array of shape (n_a, 1)
    Wy -- Weight matrix relating the hidden-state to the output, numpy array of
shape (n_y, n_a)
    by -- Bias relating the hidden-state to the output, numpy array of shape
(n_y, 1)

Returns:
a_next -- next hidden state, of shape (n_a, m)
c_next -- next memory state, of shape (n_a, m)
yt_pred -- prediction at timestep "t", numpy array of shape (n_y, m)
cache -- tuple of values needed for the backward pass, contains (a_next, c_next, a_prev, c_prev,
xt, parameters)

Note: ft/it/ot stand for the forget/update/output gates, cct stands for the candidate value (c
tilde),
      c stands for the memory value
"""

# Retrieve parameters from "parameters"
Wf = parameters["Wf"]
bf = parameters["bf"]
Wi = parameters["Wi"]
bi = parameters["bi"]
Wc = parameters["Wc"]
bc = parameters["bc"]
Wo = parameters["Wo"]
bo = parameters["bo"]
Wy = parameters["Wy"]
by = parameters["by"]

# Retrieve dimensions from shapes of xt and Wy
n_x, m = xt.shape
n_y, n_a = Wy.shape

### START CODE HERE ###
# Concatenate a_prev and xt (~3 lines)
concat = np.zeros(shape=[n_a+n_x,m])
concat[: n_a, :] = a_prev
concat[n_a :, :] = xt

# Compute values for ft, it, cct, c_next, ot, a_next using the formulas given figure (4) (~6
lines)

ft = sigmoid(np.dot(Wf,concat)+bf)

```

```

it = sigmoid(np.dot(Wi,concat)+bi)
cct = np.tanh(np.dot(Wc,concat)+bc)
c_next = np.multiply(ft,c_prev)+np.multiply(it,cct)
ot = sigmoid(np.dot(Wo,concat)+bo)
a_next = np.multiply(ot,np.tanh(c_next))

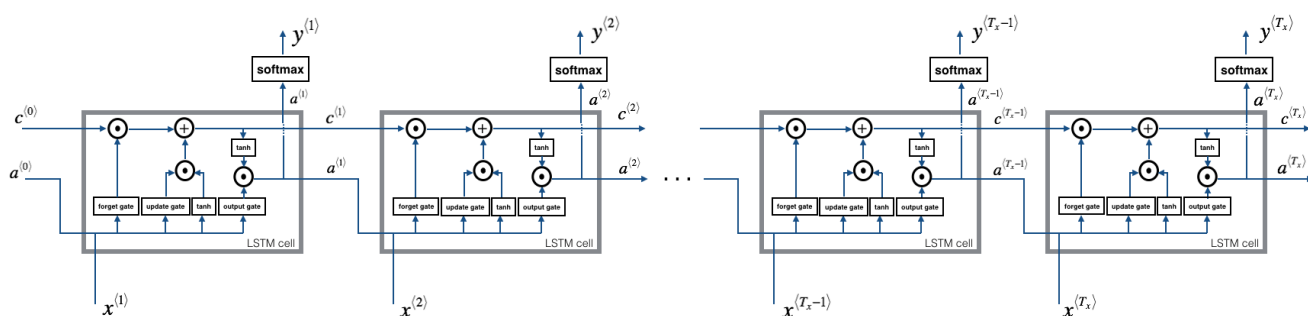
# Compute prediction of the LSTM cell (≈1 line)
yt_pred = softmax(np.dot(Wy,a_next)+by)
### END CODE HERE ###

# store values needed for backward propagation in cache
cache = (a_next, c_next, a_prev, c_prev, ft, it, cct, ot, xt, parameters)

return a_next, c_next, yt_pred, cache

```

## LSTM 正向传播



上图展示了LSTM的前向传递过程，初始的memory  $c^0$  和hidden state  $a^0$  均为0向量。

```
def lstm_forward(x, a0, parameters):
```

```

"""
Implement the forward propagation of the recurrent neural network using an LSTM-cell described
in Figure (3).

```

Arguments:

x -- Input data for every time-step, of shape (n\_x, m, T\_x).

a0 -- Initial hidden state, of shape (n\_a, m)

parameters -- python dictionary containing:

Wf -- Weight matrix of the forget gate, numpy array of shape (n\_a, n\_a + n\_x)

bf -- Bias of the forget gate, numpy array of shape (n\_a, 1)

Wi -- Weight matrix of the update gate, numpy array of shape (n\_a, n\_a + n\_x)

bi -- Bias of the update gate, numpy array of shape (n\_a, 1)

Wc -- Weight matrix of the first "tanh", numpy array of shape (n\_a, n\_a + n\_x)

bc -- Bias of the first "tanh", numpy array of shape (n\_a, 1)

Wo -- Weight matrix of the output gate, numpy array of shape (n\_a, n\_a + n\_x)

bo -- Bias of the output gate, numpy array of shape (n\_a, 1)

Wy -- Weight matrix relating the hidden-state to the output, numpy array of

```

shape (n_y, n_a)
        by -- Bias relating the hidden-state to the output, numpy array of shape
(n_y, 1)

Returns:
a -- Hidden states for every time-step, numpy array of shape (n_a, m, T_x)
y -- Predictions for every time-step, numpy array of shape (n_y, m, T_x)
caches -- tuple of values needed for the backward pass, contains (list of all the caches, x)
"""

# Initialize "caches", which will track the list of all the caches
caches = []

#### START CODE HERE ####
# Retrieve dimensions from shapes of x and parameters['Wy'] (~2 lines)
n_x, m, T_x = x.shape
n_y, n_a = Wy.shape

# initialize "a", "c" and "y" with zeros (~3 lines)
a = np.zeros(shape=[n_a,m,T_x])
c = np.zeros(shape=[n_a,m,T_x])
y = np.zeros(shape=[n_y,m,T_x])

# Initialize a_next and c_next (~2 lines)
a_next = a0
c_next = np.zeros(shape=[n_a,m])

# loop over all time-steps
for t in range(T_x):
    # Update next hidden state, next memory state, compute the prediction, get the cache (~1
line)
    a_next, c_next, yt, cache = lstm_cell_forward(x[:, :, t], a_next, c_next, parameters)
    # Save the value of the new "next" hidden state in a (~1 line)
    a[:, :, t] = a_next
    # Save the value of the prediction in y (~1 line)
    y[:, :, t] = yt
    # Save the value of the next cell state (~1 line)
    c[:, :, t] = c_next
    # Append the cache into caches (~1 line)
    caches.append(cache)

#### END CODE HERE ####

# store values needed for backward propagation in cache
caches = (caches, x)

return a, y, c, caches

```

## LSTM反向传播

### LSTM cell 反向传播

LSTM cell的反向传播比较复杂，我们先将正向传播添加一些标记，重新书写正向传播，然后在推导反向传播。



$$I\tilde{a}^t = W_a[a^{t-1}, x^t] + b_a$$

$$\tilde{a}^t = \tanh(I\tilde{a}^t)$$

$$If^t = W_f[a^{t-1}, x^t] + b_f$$

$$f^t = \sigma(If^t)$$

$$Iu^t = W_u[a^{t-1}, x^t] + b_u$$

$$u^t = \sigma(Iu^t)$$

$$Io^t = W_o[a^{t-1}, x^t] + b_o$$

$$o^t = \sigma(Io^t)$$

$$c^t = f^t \cdot c^{t-1} + u^t \cdot \tilde{a}^t$$

$$a^t = o^t \cdot \tanh(c^t)$$

LSTM-cell反向传播时，接受来自下一层梯度 $d_{a^t}, d_{c^t}$ 和正向传播时输出的相关参数。根据链式法则进行推导：

根据： $a^t = o^t \cdot \tanh(c^t)$ ，得到 $d_{o^t} = d_{a^t} * \tanh(c^t)$ ,  $d_{c^t} = d_{c^t} + d_{a^t} \cdot o^t \cdot (1 - (\tanh(c^t))^2)$

根据： $c^t = f^t \cdot c^{t-1} + u^t \cdot \tilde{a}^t$ 得到 $d_{\tilde{a}^t} = d_{c^t} \cdot u^t$ ,  $d_{u^t} = d_{c^t} \cdot \tilde{a}^t$ ,  $d_{f^t} = d_{c^t} \cdot c^{t-1}$ ,  $d_{c^{t-1}} = d_{c^t} \cdot f^t$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

根据： $o^t = \sigma(Io^t)$ 得到： $d_{Io^t} = d_{o^t} (1 - d_{o^t})$

根据： $f^t = \sigma(If^t)$ 得到： $d_{If^t} = d_{f^t} (1 - d_{f^t})$

根据： $u^t = \sigma(Iu^t)$ 得到： $d_{Iu^t} = d_{u^t} (1 - d_{u^t})$

$$\tanh'(x) = 1 - \tanh(x)^2$$

根据： $\tilde{a}^t = \tanh(I\tilde{a}^t)$ 得到： $d_{I\tilde{a}^t} = 1 - (\tanh(\tilde{a}^t))^2$

将 $a^{t-1}$ 与 $x^t$ 合并得到conax。

根据RNN反向传播时得到的结论：当矩阵与列向量相乘时，假设 $U=Mv+b$ (M为矩阵，b,v为列向量)

$$\text{则有：} \frac{\partial L}{\partial v} = M^T \frac{\partial L}{\partial U}, \quad \frac{\partial L}{\partial M} = \frac{\partial L}{\partial U} v^T$$

根据 $I\tilde{a}^t = W_a[a^{t-1}, x^t] + b_a$ ，可以得到 $d_{W_a} = d_{I\tilde{a}^t} \text{conax}.T$ ,  $d_{b_a} = \sum_{batch} d_{I\tilde{a}^t}$

根据 $If^t = W_f[a^{t-1}, x^t] + b_f$ ，可以得到 $d_{W_f} = d_{If^t} \text{conax}.T$ ,  $d_{b_f} = \sum_{batch} d_{If^t}$

根据 $Iu^t = W_u[a^{t-1}, x^t] + b_u$ ，可以得到 $d_{W_u} = d_{Iu^t} \text{conax}.T$ ,  $d_{b_u} = \sum_{batch} d_{Iu^t}$

根据 $Io^t = W_o[a^{t-1}, x^t] + b_o$ ，可以得到 $d_{W_o} = d_{Io^t} \text{conax}.T$ ,  $d_{b_o} = \sum_{batch} d_{Io^t}$

$x^t$ 与 $a^{t-1}$ 与 $I\tilde{a}^t, If^t, Iu^t, Io^t$ 都有关：

从而有

$$d_{a^{t-1}} = W_f[:, n_a]^T \cdot d_{f^t} + W_o[:, n_a]^T \cdot d_{f_o^t} + W_u[:, n_a]^T \cdot d_{u^t} + W_a[:, n_a]^T \cdot d_{\tilde{a}^t}$$

$$d_{x^t} = W_f[:, n_a:]^T \cdot d_{f^t} + W_o[:, n_a:]^T \cdot d_{f_o^t} + W_u[:, n_a:]^T \cdot d_{u^t} + W_a[:, n_a:]^T \cdot d_{\tilde{a}^t}$$

LSTM cell 反向传播python实现：

```

def lstm_cell_backward(da_next, dc_next, cache):
    """
    Implement the backward pass for the LSTM-cell (single time-step).

    Arguments:
    da_next -- Gradients of next hidden state, of shape (n_a, m)
    dc_next -- Gradients of next cell state, of shape (n_a, m)
    cache -- cache storing information from the forward pass

    Returns:
    gradients -- python dictionary containing:
        dxt -- Gradient of input data at time-step t, of shape (n_x, m)
        da_prev -- Gradient w.r.t. the previous hidden state, numpy array of shape
        (n_a, m)
        dc_prev -- Gradient w.r.t. the previous memory state, of shape (n_a, m, T_x)
        dWf -- Gradient w.r.t. the weight matrix of the forget gate, numpy array of
        shape (n_a, n_a + n_x)
        dWi -- Gradient w.r.t. the weight matrix of the update gate, numpy array of
        shape (n_a, n_a + n_x)
        dWc -- Gradient w.r.t. the weight matrix of the memory gate, numpy array of
        shape (n_a, n_a + n_x)
        dWo -- Gradient w.r.t. the weight matrix of the output gate, numpy array of
        shape (n_a, n_a + n_x)
        dbf -- Gradient w.r.t. biases of the forget gate, of shape (n_a, 1)
        dbi -- Gradient w.r.t. biases of the update gate, of shape (n_a, 1)
        dbc -- Gradient w.r.t. biases of the memory gate, of shape (n_a, 1)
        dbo -- Gradient w.r.t. biases of the output gate, of shape (n_a, 1)
    """

    # Retrieve information from "cache"
    (a_next, c_next, a_prev, c_prev, ft, it, cct, ot, xt, parameters) = cache

    ### START CODE HERE ###
    # Retrieve dimensions from xt's and a_next's shape (~2 lines)
    n_x, m = xt.shape
    n_a, m = a_next.shape

    # Compute gates related derivatives, you can find their values can be found by looking carefully
    # at equations (7) to (10) (~4 lines)
    dot = da_next*np.tanh(c_next)
    dct = dc_next + da_next*ot*(1-np.power(np.tanh(c_next),2))
    dcct = dct*it
    dit = dct*cct
    dft = dct*c_prev

    # Code equations (7) to (10) (~4 lines)
    dit = dit*it*(1-it)
    dft = dft*ft*(1-ft)
    dot = dot*ot*(1-ot)
    dcct = dcct*(1-np.power(cct,2))

    # Compute parameters related derivatives. Use equations (11)-(14) (~8 lines)
    concat = np.concatenate([a_prev,xt])

```

```

dWf = np.dot(dft,concat.T)
dWi = np.dot(dit,concat.T)
dWc = np.dot(dcct,concat.T)
dWo = np.dot(dot,concat.T)
dbf = np.sum(dft,axis=1,keepdims=True)
dbi = np.sum(dit,axis=1,keepdims=True)
dbc = np.sum(dcct,axis=1,keepdims=True)
dbo = np.sum(dot,axis=1,keepdims=True)

# Compute derivatives w.r.t previous hidden state, previous memory state and input. Use
equations (15)-(17). (≈3 lines)
da_prev = np.dot(parameters['Wf'][:, :n_a].T,dft) + np.dot(parameters['Wi']
[:, :n_a].T,dit)+np.dot(parameters['Wo'][:, :n_a].T,dot)+np.dot(parameters['Wc'][:, :n_a].T,dcct)
dc_prev = dct*ft
dxt = np.dot(parameters['Wf'][:, n_a:].T,dft) + np.dot(parameters['Wi']
[:, n_a:].T,dit)+np.dot(parameters['Wo'][:, n_a:].T,dot)+np.dot(parameters['Wc'][:, n_a:].T,dcct)
### END CODE HERE ###

# Save gradients in dictionary
gradients = {"dxt": dxt, "da_prev": da_prev, "dc_prev": dc_prev, "dWf": dWf,"dbf": dbf, "dWi":
dWi,"dbi": dbi,
            "dWc": dWc,"dbc": dbc, "dWo": dWo,"dbo": dbo}

return gradients

```

## LSTM model 的反向传播

```

def lstm_backward(da, dc, caches):
    """
    Implement the backward pass for the RNN with LSTM-cell (over a whole sequence).

    Arguments:
    da -- Gradients w.r.t the hidden states, numpy-array of shape (n_a, m, T_x)
    dc -- Gradients w.r.t the memory states, numpy-array of shape (n_a, m, T_x)
    caches -- cache storing information from the forward pass (lstm_forward)

    Returns:
    gradients -- python dictionary containing:
        dx -- Gradient of inputs, of shape (n_x, m, T_x)
        da0 -- Gradient w.r.t. the previous hidden state, numpy array of shape (n_a,
m)
        dWf -- Gradient w.r.t. the weight matrix of the forget gate, numpy array of
shape (n_a, n_a + n_x)
        dWi -- Gradient w.r.t. the weight matrix of the update gate, numpy array of
shape (n_a, n_a + n_x)
        dWc -- Gradient w.r.t. the weight matrix of the memory gate, numpy array of
shape (n_a, n_a + n_x)
        dWo -- Gradient w.r.t. the weight matrix of the save gate, numpy array of
shape (n_a, n_a + n_x)
        dbf -- Gradient w.r.t. biases of the forget gate, of shape (n_a, 1)
        dbi -- Gradient w.r.t. biases of the update gate, of shape (n_a, 1)
        dbc -- Gradient w.r.t. biases of the memory gate, of shape (n_a, 1)
        dbo -- Gradient w.r.t. biases of the save gate, of shape (n_a, 1)
    """

```

```

"""

# Retrieve values from the first cache (t=1) of caches.
(caches, x) = caches
(a1, c1, a0, c0, f1, i1, cc1, o1, x1, parameters) = caches[0]

### START CODE HERE ###
# Retrieve dimensions from da's and x1's shapes (≈2 lines)
n_a, m, T_x = da.shape
n_x, m = x1.shape

# initialize the gradients with the right sizes (≈12 lines)
dx = np.zeros(shape=[n_x,m,T_x])
da0 = np.zeros(shape=[n_a,m])
da_prevt = np.zeros(shape=[n_a,m])
dc_prevt = np.zeros(shape=[n_a,m])
dWf = np.zeros(shape=[n_a,n_a+n_x])
dWi = np.zeros(shape=[n_a,n_a+n_x])
dWc = np.zeros(shape=[n_a,n_a+n_x])
dWo = np.zeros(shape=[n_a,n_a+n_x])
dbf = np.zeros(shape=[n_a,1])
dbi = np.zeros(shape=[n_a,1])
dbc = np.zeros(shape=[n_a,1])
dbo = np.zeros(shape=[n_a,1])

# loop back over the whole sequence
for t in reversed(range(T_x)):
    # Compute all gradients using lstm_cell_backward
    gradients = lstm_cell_backward(da[:, :, t] + da_prevt, dc[:, :, t] + dc_prevt, caches[t])
    # Store or add the gradient to the parameters' previous step's gradient
    dx[:, :, t] = gradients['dxt']
    dWf += gradients['dWf']
    dWi += gradients['dWi']
    dWc += gradients['dWc']
    dWo += gradients['dWo']
    dbf += gradients['dbf']
    dbi += gradients['dbi']
    dbc += gradients['dbc']
    dbo += gradients['dbo']
    da_prevt = gradients['da_prev']
    dc_prevt = gradients['dc_prev']
# Set the first activation's gradient to the backpropagated gradient da_prev.
da0 = gradients['da_prev']

### END CODE HERE ###

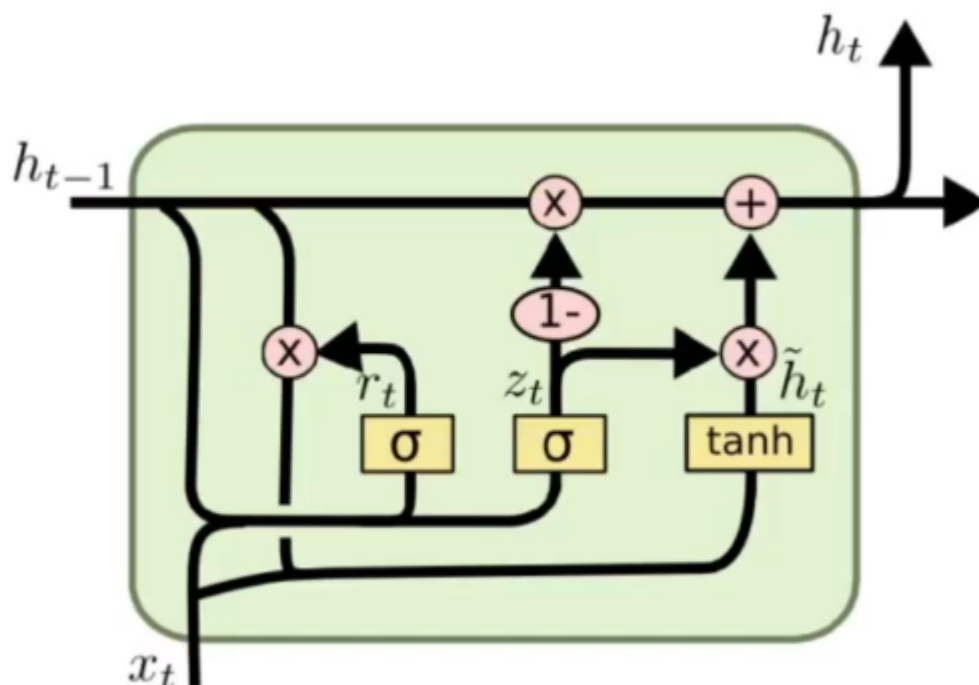
# Store the gradients in a python dictionary
gradients = {"dx": dx, "da0": da0, "dWf": dWf, "dbf": dbf, "dWi": dWi, "dbi": dbi,
            "dWc": dWc, "dbc": dbc, "dWo": dWo, "dbo": dbo}

return gradients

```

## GRU基本原理

GRU与LSTM基本精神相同，舍弃了LSTM的一些特征，从某种意义上说是LSTM的一个简化，只有两个门reset gate和update gate，使新信息的写入和旧memory的去除联动起来，只有在新信息写入时才取出旧信息，另外输出只有hidden，没有memory，因此，更加简单，参数量更少，运行的更快，虽然LSTM依然是许多NLP应用的首选，但GRU获得越来越多的关注度。



如上图所示，GRU需要两个输入，上一个时间点的hidden state  $h^{t-1}$  和当前时间点的输入  $x^t$ ，输出当前时间点的隐藏state  $h^t$ 和当前时刻输出（如果需要）。

具体实现如下：

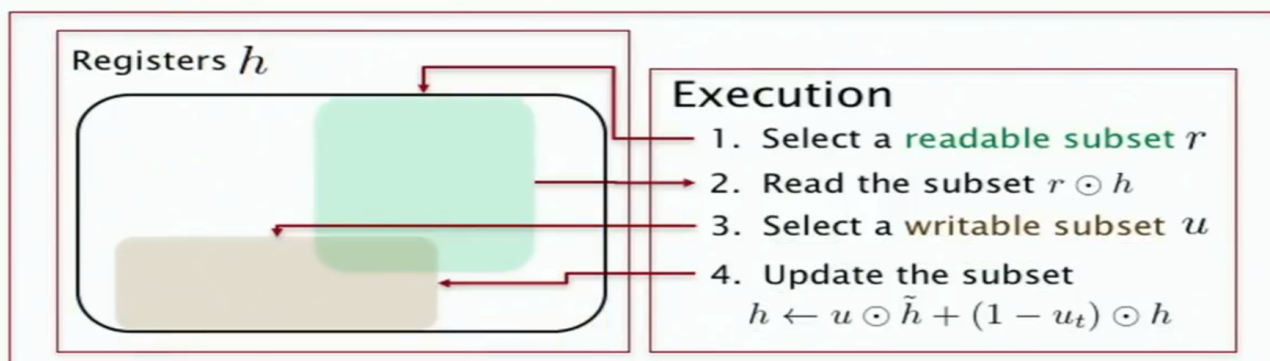
**update gate** :  $u^t = \sigma(W_u[a^{t-1}, x^t] + b_u)$

**reset gate** :  $r^t = \sigma(W_r[a^{t-1}, x^t] + b_r)$

**candidate update** :  $\tilde{h}^t = \tanh(W_{hx}x^t + W_{hh}(r^t \cdot h^{t-1}) + b_h)$

**output hidde** :  $h^t = u^t \cdot \tilde{h}^t + (1 - u^t) \cdot h^{t-1}$

*GRU ...*



如果将memory看作寄存器，可以将GRU的操作看作如上图所示情况：

reset gate( $r^t$ )决定使用那一部分信息去更新候选的candidate update

update gate( $u^t$ )决定覆写隐藏层中的那些信息，只有在写入新信息时才会覆写旧的memory，而LSTM没有这种机制。

GRU相比于LSTM的实现相对更简单，就不用python实现，只附上tensorflow中的实现：

```
class GRUCell(tf.nn.rnn_cell.RNNCell):
    """Wrapper around our GRU cell implementation that allows us to play
    nicely with TensorFlow.
    """
    def __init__(self, input_size, state_size):
        self.input_size = input_size
        self._state_size = state_size

    @property
    def state_size(self):
        return self._state_size

    @property
    def output_size(self):
        return self._state_size

    def __call__(self, inputs, state, scope=None):
        """Updates the state using the previous @state and @inputs.
        Remember the GRU equations are:

        z_t = sigmoid(x_t U_z + h_{t-1} W_z + b_z)
        r_t = sigmoid(x_t U_r + h_{t-1} W_r + b_r)
        o_t = tanh(x_t U_o + r_t * h_{t-1} W_o + b_o)
        h_t = z_t * h_{t-1} + (1 - z_t) * o_t

        TODO: In the code below, implement an GRU cell using @inputs
        (x_t above) and the state (h_{t-1} above).
        - Define W_r, U_r, b_r, W_z, U_z, b_z and W_o, U_o, b_o to
          be variables of the appropriate shape using the
          `tf.get_variable` functions.
        - Compute z, r, o and @new_state (h_t) defined above
        Tips:
        - Remember to initialize your matrices using the xavier
          initialization as before.
        Args:
        inputs: is the input vector of size [None, self.input_size]
        state: is the previous state vector of size [None, self.state_size]
        scope: is the name of the scope to be used when defining the variables inside.
        Returns:
        a pair of the output vector and the new state vector.
        """
        scope = scope or type(self).__name__

        # It's always a good idea to scope variables in functions lest they
        # be defined elsewhere!
```

```

with tf.variable_scope(scope):
    ### YOUR CODE HERE (~20-30 lines)
    U_z = tf.get_variable(name='U_z', shape=
[self.input_size,self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    U_r = tf.get_variable(name='U_r', shape=
[self.input_size,self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    U_o = tf.get_variable(name='U_o', shape=
[self.input_size,self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    W_z = tf.get_variable(name='W_z', shape=
[self._state_size,self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    W_r = tf.get_variable(name='W_r', shape=
[self._state_size,self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    W_o = tf.get_variable(name='W_o', shape=
[self._state_size,self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    b_z = tf.get_variable(name='b_z', shape=
[self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    b_r = tf.get_variable(name='b_r', shape=
[self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    b_o = tf.get_variable(name='b_o', shape=
[self._state_size],dtype=tf.float32,initializer=tf.contrib.layers.xavier_initializer())
    z_t = tf.sigmoid(tf.matmul(inputs,U_z)+tf.matmul(state,W_z)+b_z,name='z_t')
    r_t = tf.sigmoid(tf.matmul(inputs,U_r)+tf.matmul(state,W_r)+b_r,name='r_t')
    o_t =
tf.tanh(tf.matmul(inputs,U_o)+tf.matmul(tf.multiply(r_t,state),W_o)+b_o,name='o_t')
    new_state = tf.multiply(z_t,state) + tf.multiply((1-z_t),o_t)
    ### END YOUR CODE ###

    # For a GRU, the output and state are the same (N.B. this isn't true
    # for an LSTM, though we aren't using one of those in our
    # assignment)
    output = new_state
    return output, new_state

```

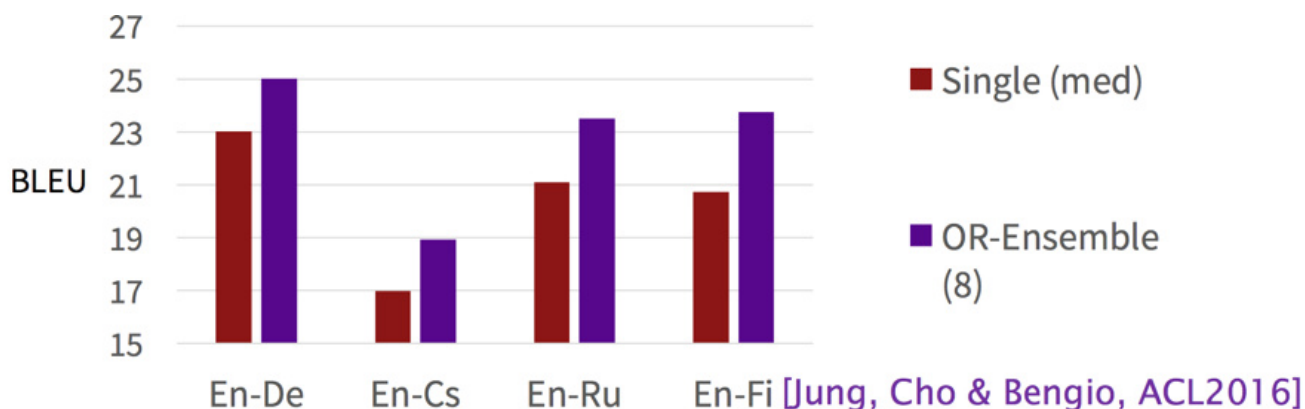
需要说明的是 LSTM和GRU只能捕捉long short memory，并不能捕捉非常长的long memory，非常长时间的依赖问题依然是尚未克服的难点。

## 训练循环神经网络的一些技巧

CS224n 课程中给出了一些训练RNN的技巧

- 将递归权值矩阵初始化为正交
- 将其他矩阵初始化为较小的值
- 将forget gate偏置设为1：默认为不遗忘
- 使用自适应的学习率算法：Adam、AdaDelta
- 裁剪梯度的长度为小于1-5
- 在Cell中垂直应用Dropout而不是水平Dropout
- 保持耐心，通常需要训练很长时间

ensembles(集成)会显著提升泛化能力，可能获得2个百分点的效果提升，但需要更多的计算成本。



## RNN扩展

### 1、Bidirectional RNN

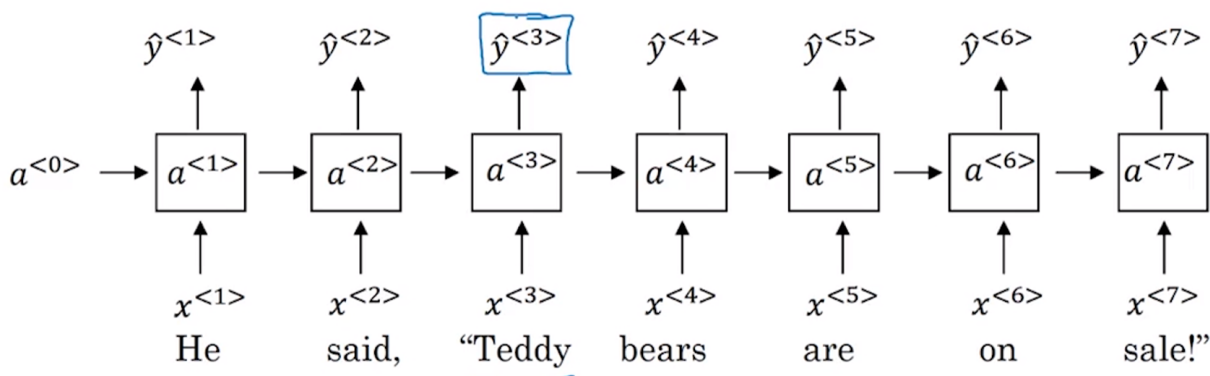
NLP中的许多任务，在某个时间点做出输出时，不仅需要过去的信息，还需要来自未来的信息。如下图所示，我们希望判断 Teddy 是否是一个姓名时，根据前三个单词我们无法判断，需要来自未来的信息。

## Getting information from the future

网易云课堂

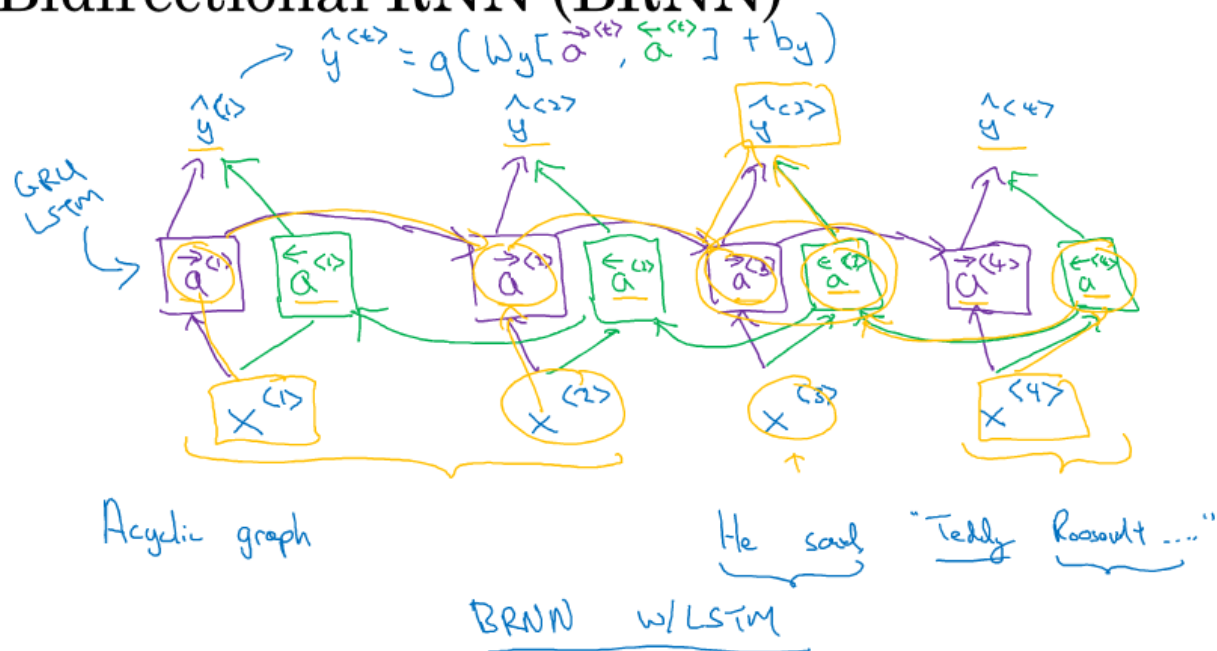
He said, "Teddy bears are on sale!"

He said, "Teddy Roosevelt was a great President!"





# Bidirectional RNN (BRNN)



$$\vec{a}_t = f(\vec{W}_l x_t + \vec{V}_l \vec{a}_{t-1} + \vec{b}_l)$$

$$\overleftarrow{a}_t = f(\overleftarrow{W}_r x_t + \overleftarrow{V}_r \overleftarrow{a}_{t+1} + \overleftarrow{b}_r)$$

$$\hat{y}_t = g(U[\vec{a}_t; \overleftarrow{a}_t] + c)$$

Bidirectional RNN 的结构如上图所示，输入一个序列，每个时间点，都会接收来自两个方向传递来的隐藏信息，当进行输出时，将两个方向的隐藏state拼接起来，进行输出，从而使得每个时间点做出预测时，不仅有来自过去的信息，也获得来自未来的信息。

## 2、Deep RNN

Bidirectional LSTM 或者GRU已经可以获得很好的表现，但是如果我们需要实现更复杂的函数，我们可以使用 Deep RNN,事实上实际应用中RNN都是Deep RNN ( 增大hidden size, 也可以实现更复杂的模型,但是如果增大hidden size, 需要的参数量将会以二次的形式增加, 计算和内存压力会非常大, 通过过多的参数也降低泛化能力, 与在MLP中的讨论一样, 相比于使用矮胖的模型, 瘦高的模型参数具有更高的表达能力, 也能显著提升泛化能力 )

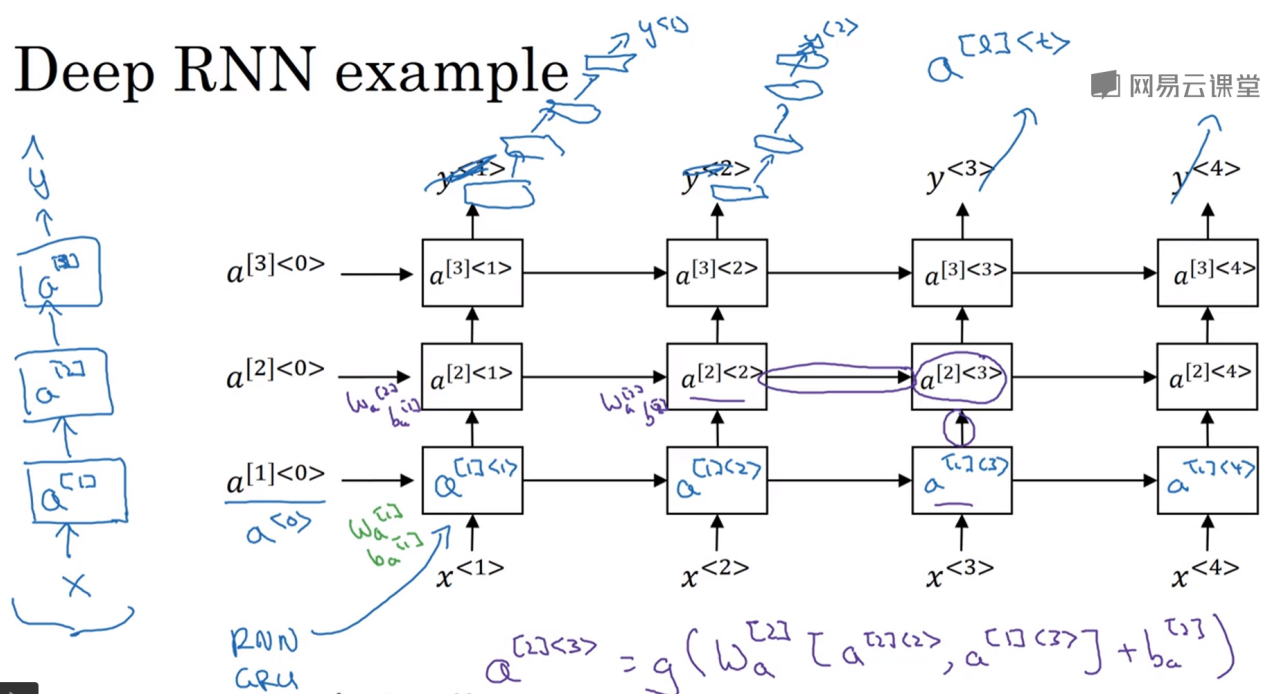
$$\vec{a}_t^i = f(\vec{W}_l^i \vec{a}_t^{i-1} + \vec{V}_l^i \vec{a}_{t-1}^i + \vec{b}_l^i)$$

$$\overleftarrow{a}_t^i = f(\overleftarrow{W}_r^i \overleftarrow{a}_t^{i-1} + \overleftarrow{V}_r^i \overleftarrow{a}_{t+1}^i + \overleftarrow{b}_r^i)$$

$$\hat{y}_t = g(U[\vec{a}_t^L; \overleftarrow{a}_t^L] + c)$$

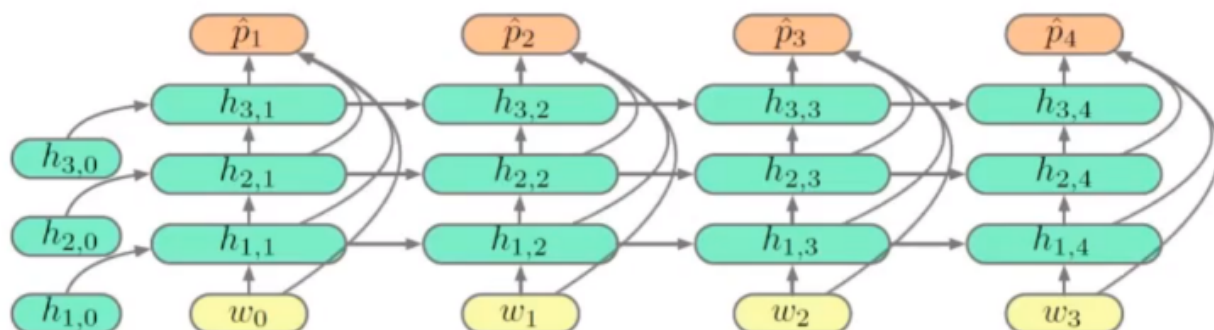
在做许多任务时，在输出层可以紧接一个全连接网络作为输出。

# Deep RNN example



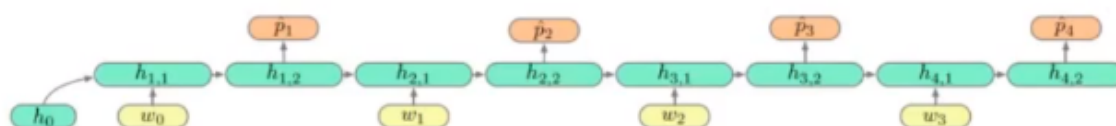
## 3、Deep RNN中引入skip-connection

与ResNet一样，在深度方向上建立skip-connection（并不需要像下图那样在每层都连接到输出层），从而更好的解决梯度消失问题，使学习更复杂的网络更加简单，训练更加快速。



## 4、在时间方向上 进行deep

可以考虑在真实的时间步上增加额外的时间步，实现更多的非线性表达，提升模型表达能力（这个思路在最近提出的循环高速网络中表现非常好）



## 5、将上述技巧融合起来

需要说明的是即使看起来并不深的RNN，尤其Deep LSTM或者Deep GRU，训练时间可能非常长，需要大量的计算能力。

## RNN 应用模式

NLP许多任务可以归结为以下三类，

**1、Sequence to Category** 即分类问题，输入为文本序列，输出只有一个y值。

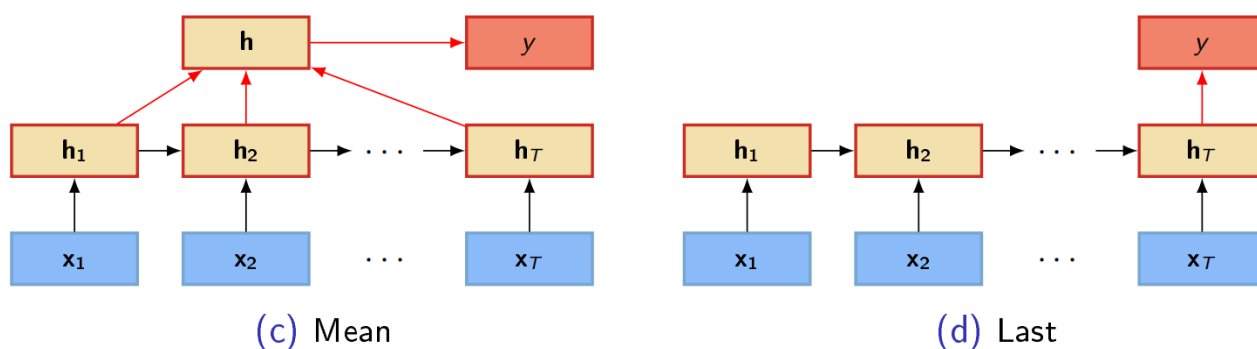
比如文本分类，情感分析，垃圾邮件识别，文本主题分析等。

**2、Synchronous Sequence to Sequence** 即序列标记问题，输入与输出都为序列，且一个输入对应一个输出，比如词性标记，中文分词，命名实体识别，有时为了更好的利用未来的信息，采用Bidirection LSTM或GRU效果会更好。

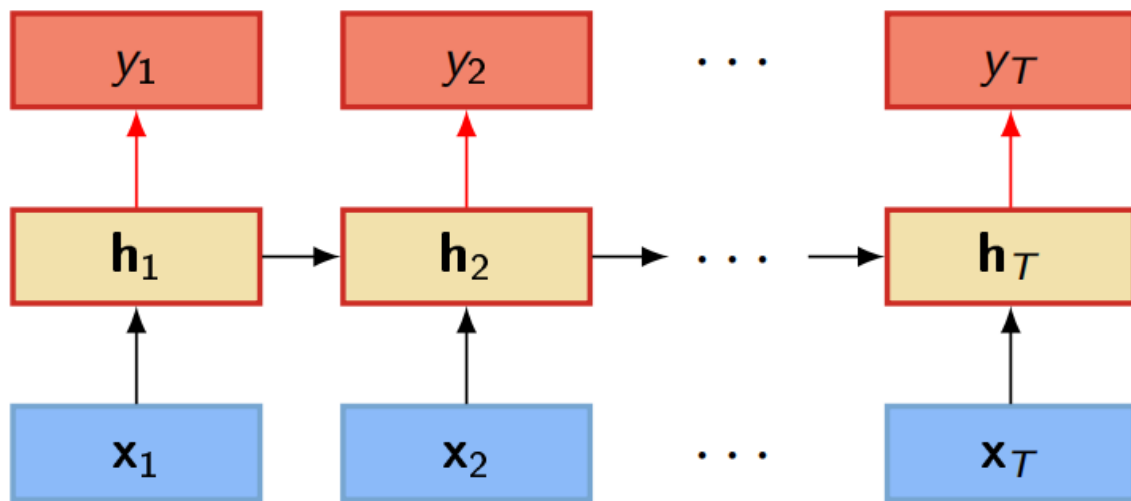
**3、Asynchronous Sequence to Sequence** 即条件语言模型，对应encoder-decoder 模型，输入与输出都为序列，但输入与输出个数不相同。读入整个序列，然后使用条件语言模型生成另外的序列，比如机器翻译，语音识别，image caption，聊天机器人，音乐合成等。

对应的网络架构如下图所示，

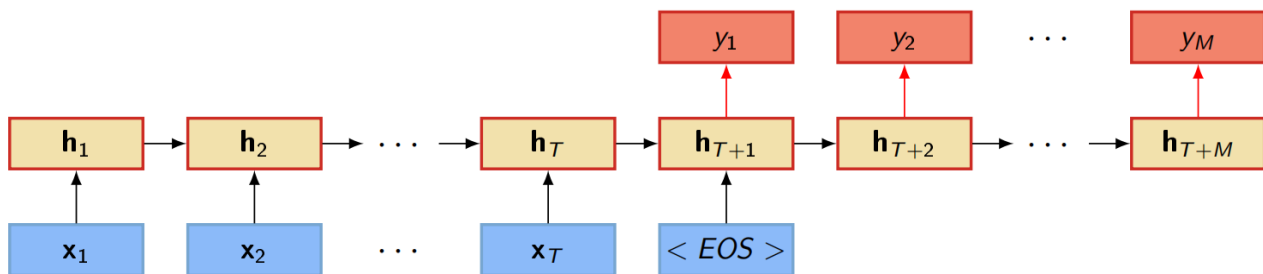
### 1、Sequence to Category



### 2、Synchronous Sequence to Sequence



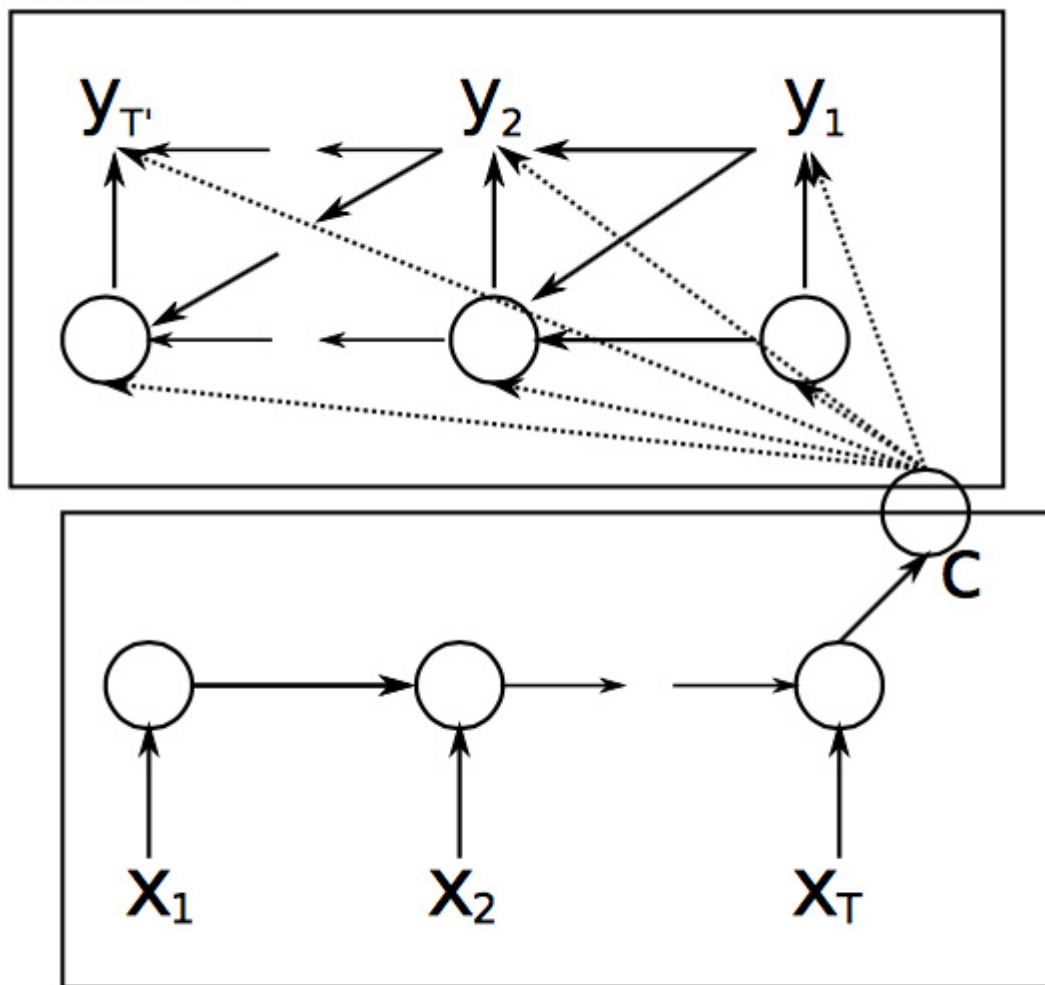
### 3、Asynchronous(异步) Sequence to Sequence



decoder中的隐藏层的输入来自3个方面：

- 前一个时刻的隐藏层
- encoder的最后一个隐藏层( $c = h_T$ )
- 前一个预测结果  $y^{t-1}$

## Decoder



## Encoder

需要说明的是，encoder输入部分也可以不是文本数据，也可以是图像（image caption），语音信息（speech recognition）等，因此，对应的网络也不仅仅局限于RNN，也可以是CNN，或者CNN与RNN的结合，或者更复杂的模式，但decoder部分需要为RNN。

encoder-decoder 模型的一个最主要问题是,encoder 将信息传递给decoder时，如果只传递一个向量，会有两个问题：

- 1、**单一向量容量太小**，不太可能表示之前所有的信息，
- 2、要面对**长期依赖**问题。

因此，需要更好的机制，即注意力模型。将用专门的一篇进行说明。

## RNN应用实例

为了文章的完整性，增加一个RNN的应用实例，实例来自吴恩达深度学习工程师序列模型作业：Improvise a Jazz Solo with an LSTM Network，为了简单，只说明重点部分。

### 训练集

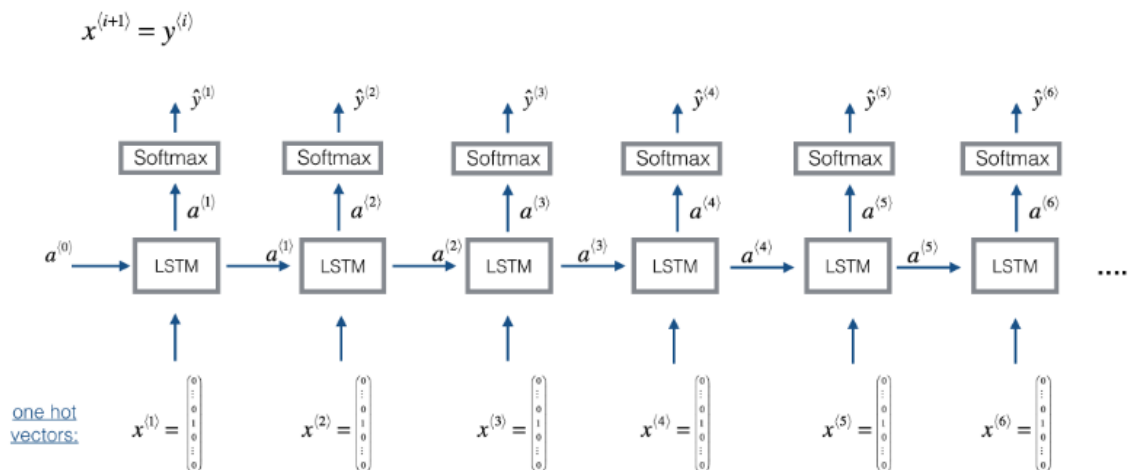
训练集的shape如下：总共有60个训练集，每个样本有30个时间点，每个时间点输入都是一个78维的one-hot vector，代表音乐生成可能的输出，输出与输入一直， $x^t = y^{t-1}$ ，只是为了后续方便调换样本数和时间维度。

```

shape of X: (60, 30, 78)
number of training examples: 60
Tx (length of sequence): 30
total # of unique values: 78
Shape of Y: (30, 60, 78)

```

## 模型架构



## 构建模型

定义hidden size :

```
n_a = 64
```

keras中为了在每个时间点share weight，需要先定义layer object，然后再构建整体模型时调用。

本实例中定义了三个layer object：重采样层：reshape，LSTM层：LSTM\_cell和全连接层dense

```

reshapor = Reshape((1, 78))          # Used in Step 2.B of djmodel(), below
LSTM_cell = LSTM(n_a, return_state = True) # Used in Step 2.C
dense = Dense(n_values, activation='softmax') # Used in Step 2.D

```

## 定义模型：

用零向量 初始化memory state和 hidden state

每个时间点，通过Lambda层获取该时间点的输入，然后对其重采样，输入到LSTM\_cell中，并将前一步的hidden state 和memory state也输入到LSTM中。每次输入通过一个全连接层进行输出，并将输出存在outputs中。

```
def djmodel(Tx, n_a, n_values):
```

```

"""
Implement the model

Arguments:
Tx -- length of the sequence in a corpus
n_a -- the number of activations used in our model
n_values -- number of unique values in the music data

```

```

Returns:
model -- a keras model with the
"""

# Define the input of your model with a shape
X = Input(shape=(Tx, n_values))

# Define s0, initial hidden state for the decoder LSTM
a0 = Input(shape=(n_a,), name='a0')
c0 = Input(shape=(n_a,), name='c0')
a = a0
c = c0

### START CODE HERE ###
# Step 1: Create empty list to append the outputs while you iterate (≈1 line)
outputs = []

# Step 2: Loop
for t in range(Tx):

    # Step 2.A: select the "t"th time step vector from X.
    x = Lambda(lambda x: X[:,t,:])(X)
    # Step 2.B: Use reshapor to reshape x to be (1, n_values) (≈1 line)
    x = reshapor(x)
    # Step 2.C: Perform one step of the LSTM_cell
    a, _, c = LSTM_cell(inputs=x, initial_state=[a, c])
    # Step 2.D: Apply densor to the hidden state output of LSTM_Cell
    out = densor(a)
    # Step 2.E: add the output to "outputs"
    outputs.append(out)

# Step 3: Create model instance
model = Model(inputs=[X,a0,c0],outputs=outputs)

### END CODE HERE ###

return model

```

## 生成模型实例

```
model = djmodel(Tx = 30 , n_a = 64, n_values = 78)
```

## 生成优化器对象，并编译

```

opt = Adam(lr=0.01, beta_1=0.9, beta_2=0.999, decay=0.01)

model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

```

## 初始化memory state 和 hidden state

```
m = 60
a0 = np.zeros((m, n_a))
c0 = np.zeros((m, n_a))
```

## 训练模型

```
model.fit([X, a0, c0], list(Y), epochs=100)
```

经过100次训练loss从130降到5

## 预测和采样

模型基本与训练时一致，但是有一个重要差别，即训练时 $x^t$ 是reference，生成时 $x^t$ 是通过采样得到的。

```
GRADED FUNCTION: music_inference_model
```

```
def music_inference_model(LSTM_cell, densor, n_values = 78, n_a = 64, Ty = 100):
```

```
"""
Uses the trained "LSTM_cell" and "densor" from model() to generate a sequence of values.

Arguments:
LSTM_cell -- the trained "LSTM_cell" from model(), Keras layer object
densor -- the trained "densor" from model(), Keras layer object
n_values -- integer, number of unique values
n_a -- number of units in the LSTM_cell
Ty -- integer, number of time steps to generate

Returns:
inference_model -- Keras model instance
"""

# Define the input of your model with a shape
x0 = Input(shape=(1, n_values))

# Define s0, initial hidden state for the decoder LSTM
a0 = Input(shape=(n_a,), name='a0')
c0 = Input(shape=(n_a,), name='c0')
a = a0
c = c0
x = x0

### START CODE HERE ###
# Step 1: Create an empty list of "outputs" to later store your predicted values (~1 line)
outputs = []

# Step 2: Loop over Ty and generate a value at every time step
for t in range(Ty):

    # Step 2.A: Perform one step of LSTM_cell (~1 line)
    a, _, c = LSTM_cell(inputs=x, initial_state=[a, c])
```



```

# Step 2.B: Apply Dense layer to the hidden state output of the LSTM_cell (~1 line)
out = densor(a)

# Step 2.C: Append the prediction "out" to "outputs". out.shape = (None, 78) (~1 line)
outputs.append(out)

# Step 2.D: Select the next value according to "out", and set "x" to be the one-hot
representation of the
#           selected value, which will be passed as the input to LSTM_cell on the next step.
We have provided
#           the line of code you need to do this.
x = Lambda(one_hot)(out)

# Step 3: Create model instance with the correct "inputs" and "outputs" (~1 line)
inference_model = Model(inputs=[x0,a0,c0],outputs=outputs)

### END CODE HERE ###

return inference_model

```

## 生成采样模型实例并初始化

对于条件生成模型，memory和didden state 可以采用encoder时的输出（代表条件），这里为了简单采用零向量

```

inference_model = music_inference_model(LSTM_cell, densor, n_values = 78, n_a = 64, Ty = 50)
x_initializer = np.zeros((1, 1, 78))
a_initializer = np.zeros((1, n_a))
c_initializer = np.zeros((1, n_a))

```

## 预测和采样

```

def music_inference_model(LSTM_cell, densor, n_values = 78, n_a = 64, Ty = 100):
    """
    Uses the trained "LSTM_cell" and "densor" from model() to generate a sequence of values.

    Arguments:
    LSTM_cell -- the trained "LSTM_cell" from model(), Keras layer object
    densor -- the trained "densor" from model(), Keras layer object
    n_values -- integer, umber of unique values
    n_a -- number of units in the LSTM_cell
    Ty -- integer, number of time steps to generate

    Returns:
    inference_model -- Keras model instance
    """

    # Define the input of your model with a shape
    x0 = Input(shape=(1, n_values))

    # Define s0, initial hidden state for the decoder LSTM
    a0 = Input(shape=(n_a,), name='a0')
    c0 = Input(shape=(n_a,), name='c0')

```

```

a = a0
c = c0
x = x0

### START CODE HERE ###
# Step 1: Create an empty list of "outputs" to later store your predicted values (≈1 line)
outputs = []

# Step 2: Loop over Ty and generate a value at every time step
for t in range(Ty):

    # Step 2.A: Perform one step of LSTM_cell (≈1 line)
    a, _, c = LSTM_cell(inputs=x,initial_state=[a,c])

    # Step 2.B: Apply Dense layer to the hidden state output of the LSTM_cell (≈1 line)
    out = densor(a)

    # Step 2.C: Append the prediction "out" to "outputs". out.shape = (None, 78) (≈1 line)
    outputs.append(out)

    # Step 2.D: Select the next value according to "out", and set "x" to be the one-hot
    #           representation of the
    #           selected value, which will be passed as the input to LSTM_cell on the next step.
    #           We have provided
    #           the line of code you need to do this.
    x = Lambda(one_hot)(out)

# Step 3: Create model instance with the correct "inputs" and "outputs" (≈1 line)
inference_model = Model(inputs=[x0,a0,c0],outputs=outputs)

### END CODE HERE ###

return inference_model

```

## 输出

```

results, indices = predict_and_sample(inference_model, x_initializer, a_initializer,
c_initializer)

```

## 相关课程

吴恩达 深度学习工程师 序列模型

台湾大学 李宏毅 深度学习 循环神经网络 , tips for generation

斯坦福 CS224n:Natural Language Processing with Deep Learning Recurrent Neural Networks and Language Models,Machine translation and advanced recurrent LSTMs and GRUs.

牛津大学&DeepMind Deep Learning for Natural Language Processing,Language Modeling

## 相关书籍

Ian Goodfellow ,Yoshua Bengio ,Aaron Courville 深度学习 , 序列模型 : 循环和递归神经网络

## 相关PPT

复旦大学 Xipeng Qiu Deep Learning for Natural Language Processing