Sequence Model

Lecturer: Andrew Ng, Author: Zehua Cheng

Notation:

- Superscript $[l]$ denotes an object associated with the l^{th} layer.
  - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example input.
- Superscript $\langle t \rangle$ denotes an object at the t^{th} time-step.
  - Example: $x^{\langle t \rangle}$ is the input x at the t^{th} time-step. $x^{(i)\langle t \rangle}$ is the input at the t^{th} timestep of example i.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the activations in layer l.

Representing Word

1. 用一个真·字典来存储所有的单词（或者最经常出现的），其存储并编码排序所有的单词。
2. 用one hot方法来表示单词，结果是一个N维的vector（N为字典储存的所有的词数），其vector除第X项以外都是0。（X表示该单词在字典的位置）

如：Abandon是单词表第一个词，单词表有10000个词。那么代表着Abandon的向量用One hot来做的结果就是[1,0,0,0,0,0...,0]，除了第一个是1，其余都是0。

这么做的目的是通过以上过程得到的输入X，导入Sequence Model后得到一个目标输出y。

P.S: **如果遇到不在您的词汇表中的单词，该怎么办？** 那么答案就是，你创建了一个新的标记或者一个叫做Unknown Word的新的假字，注意如下，并且返回UNK表示不在你的词汇表中的单词。

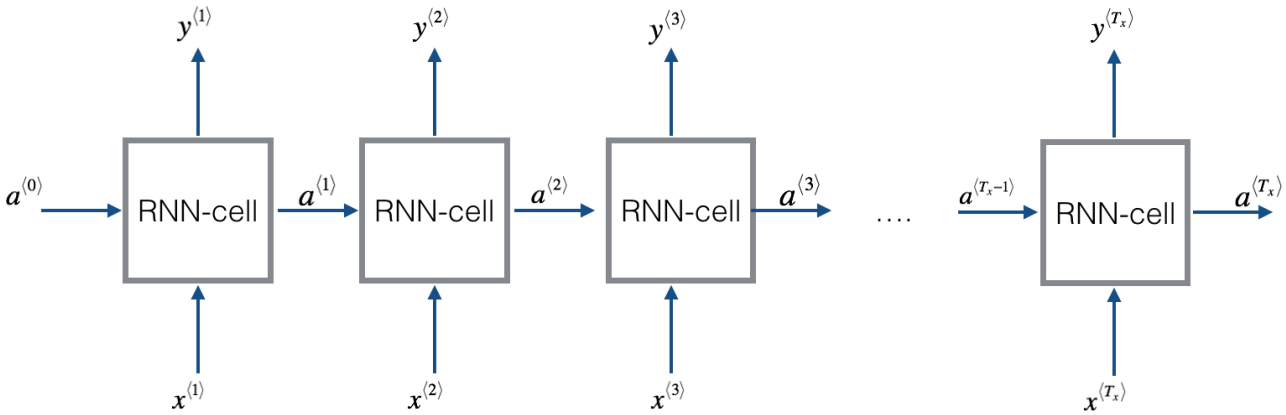
RNN

为什么不用传统神经网络

因为臣妾做不到啊！！！

1. 输入和输出在不同的样例中的长度可能不一样。
2. **不会共享在不同位置上学到的features。**

RNN



开始的时候，第一个词$X^{<1>}$输入，得到一个输出 $\hat{y}^{<1>}$以及一个$a^{<1>}$,$\hat{y}^{<1>}$ 是正经的输出而$a^{<1>}$ 传导到第二个RNN Cell。因此自第二个起，RNN Cell的输出 $\hat{y}^{<T_y>}$ 取决于中间的Hidden Layer以及传入RNN Cell的 $a^{<T_{y-1}>}$ 两个部分。

a^{T_x} 的传递使得features可以在NN Cell之间的Features的传递，也就是可以共享在不同位置上学到的features了。而至于输出不一样这个问题也可以轻松的解决。

延伸

RNN的结构与传统的深度神经网络不同在于他有很多的Cell，每一个Cell的输出可以直接就是所需要的结果。而一个RNN Cell可以有Hidden Layer——也就是说可以作为一个传统的深度神经网络来看。

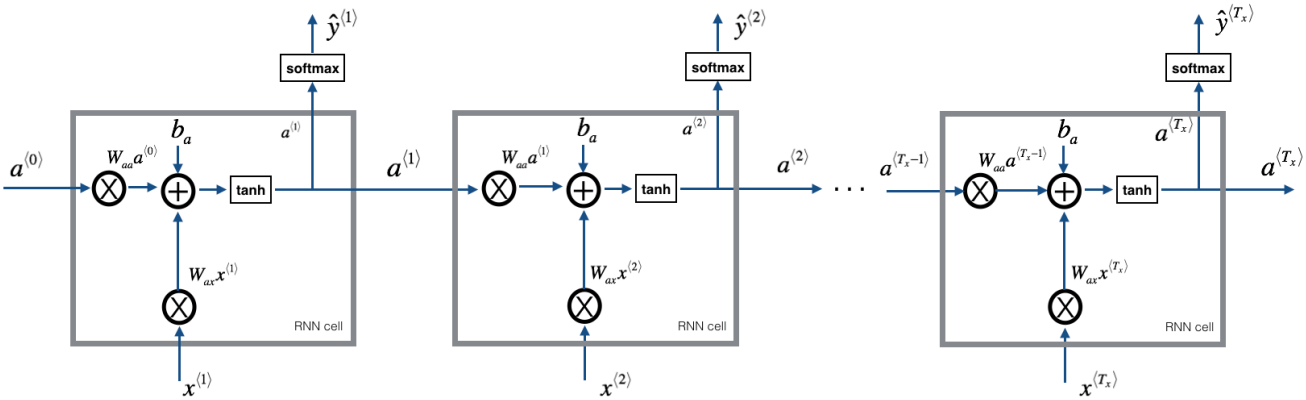
Sequence Model通过调整输出再将其传递以此解决了传统的深度神经网络所解决不了的东西，如果非要评价Sequence Model，那就是让以前的只求纵向【深度】有了另一个方向的突破，从此不只是研究如何【深下去】，更会研究如何优雅的将$a^{<T_x>}$ 传出去，也就是说，考虑到$y^{<T_x>}$ 之间的关系，这个方向走下去，就是LSTM了。

CNN在Sequence Model上的用法

Reference

1. arxiv: 1705.03122v3 Convolutional Sequence to Sequence Learning, FAIR

向前传播的基本递归神经网络

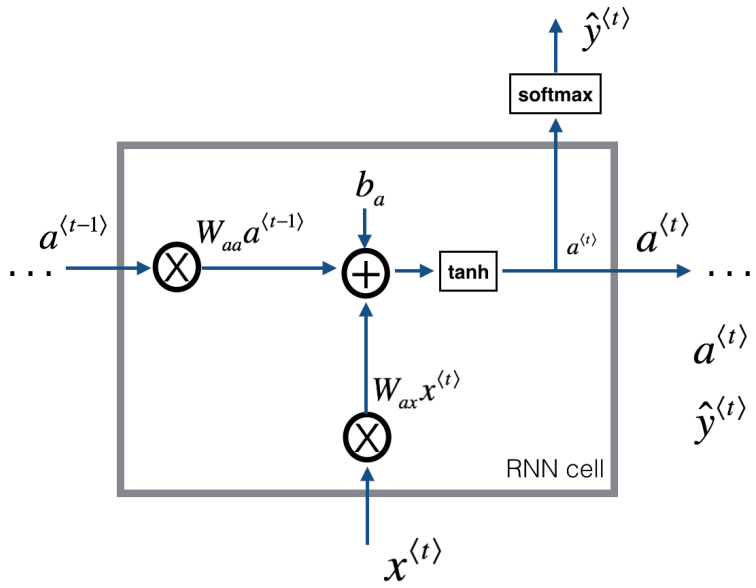


如何实现一个RNN：

1. 实现一个time-step的RNN所需要的计算。
2. 在T_x time-steps上实现一个循环，以便一次性处理所有的输入。

1.1 - RNN cell

递归神经网络可以被看作是单个细胞的重复。你首先要实现一个time-step的计算。



$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$
$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

Instructions:

1. Compute the hidden state with tanh activation: $\boldsymbol{a^{\langle t \rangle} = \tanh(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a)}$.
2. Using your new hidden state $\boldsymbol{a^{\langle t \rangle}}$, compute the prediction $\boldsymbol{\hat{y}^{\langle t \rangle} = softmax(W_{ya}a^{\langle t \rangle} + b_y)}$. We provided you a function: `softmax`.
3. Store $\boldsymbol{(a^{\langle t \rangle}, a^{\langle t-1 \rangle}, x^{\langle t \rangle}, parameters)}$ in cache
4. Return $\boldsymbol{a^{\langle t \rangle}}$, $\boldsymbol{y^{\langle t \rangle}}$ and cache

We will vectorize over $\boldsymbol{m}$ examples. Thus, $\boldsymbol{x^{\langle t \rangle}}$ will have dimension $\boldsymbol{(n_x, m)}$, and $\boldsymbol{a^{\langle t \rangle}}$ will have dimension $\boldsymbol{(n_a, m)}$.

```
# 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正向传递

其实可以将RNN视为刚刚构建的单元格的重复。如果您输入的数据序列带10个time-steps，则您将复制RNN信元10次。每个单元将前一个单元的hidden state $(a^{<t-1>})$ 和当前time-step的输入数据$(x^{<t>})$作为输入。它为这个time-step输出一个hidden-state$(a^{<t>})$和一个预测$(y^{<t>})$。

Instructions:

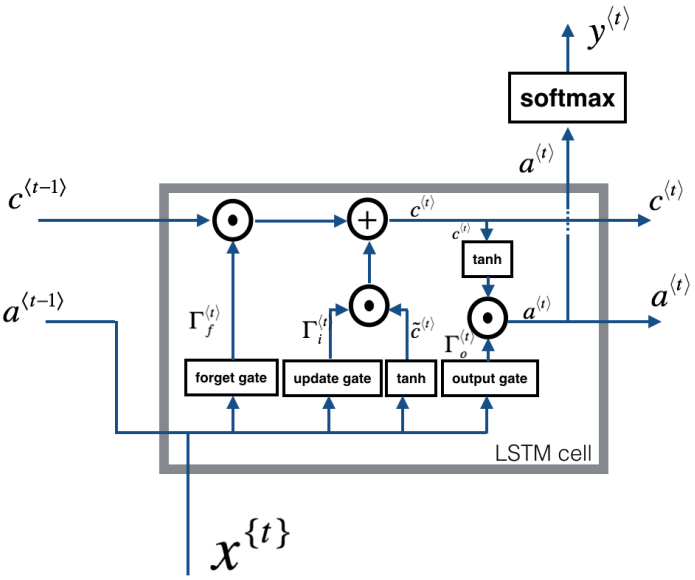
1. Create a vector of zeros (aa) that will store all the hidden states computed by the RNN.
2. Initialize the "next" hidden state as a_0(initial hidden state).
3. Start looping over each time step, your incremental index is t:
    - Update the "next" hidden state and the cache by running `rnn_cell_forward`
    - Store the "next" hidden state in a (t^{th} position)
    - Store the prediction in y
    - Add the cache to the list of caches
4. Return a, y and caches

```
# GRADED FUNCTION: rnn_forward

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((n_a, m, T_x))
    y_pred = np.zeros((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
```

LSTM

下图显示了LSTM单元的运作机理。



$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\langle t \rangle} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C)$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

关于Gate

1. Forget Gate

为了说明这一点，我们假设我们正在阅读一段文字中的单词，并且希望使用LSTM来跟踪语法结构，例如主语是单数还是复数。如果主语从单个单词变成复数单词，我们需要找到一种方法来摆脱先前存储的单数/复数状态的记忆值。

$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f) \tag{1}$$

2. Update Gate

一旦我们忘记所讨论的主题是单数的，我们需要找到一种方法来更新它，以反映新主题现在是复数。这是update gate的公式：

$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\{t\}}] + b_u) \tag{2}$$

GRADED FUNCTION: lstm_cell_forward

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.empty((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 = (ft * c_prev) + (it * cct)

ot = sigmoid(np.dot(Wo, concat) + bo)
```

```
    a_next = 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的正向传递

现在您已经实现了LSTM的一个步骤，现在可以使用for循环对此进行迭代，以处理一系列T_x 的输入。