第三次作业问题1和问题2

编写：张礼俊/SlyneD

校对：毛丽

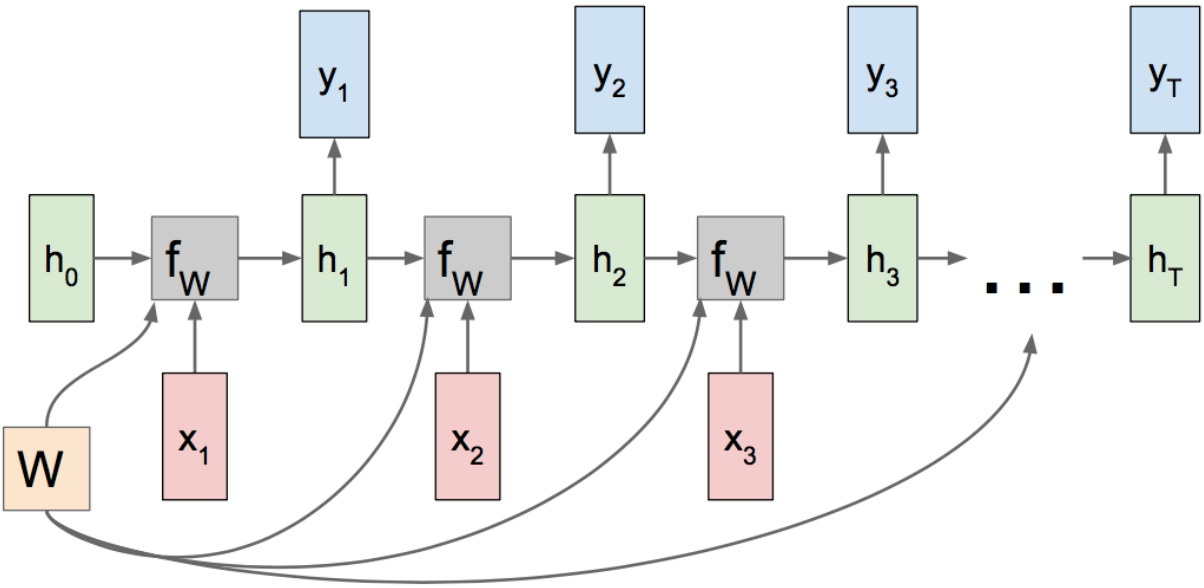
总校对与审核：寒小阳

1. 问题背景

在问题1里，我们要训练一个递归神经网络（Recurrent neural networks）来生成一个图片的文字注释(captions)。问题2中，用以长短时记忆单元（Long-short term memory，LSTM）为基础的递归神经网络来完成同样的任务。 我们将用到的数据集是微软的COCO数据集，该数据集是测试为图片加文字注释算法的标准数据集。在该数据集中有大概80000张训练图片和40000张测试图片，每张图片在Amazon Mechanical Turk上征集了五个志愿者来手动写文字说明。 我们可以运行问题1的前几个单元来对我们将要应用的数据有个直观的概念，要注意一点的是之后的处理中我们不会再碰到原始图像，该问题已经为我们提取了图像特征。

2. 递归神经网络

问题1要求我们实现一个递归神经网络。这一部分讲述了问题1的前半部分代码。 递归神经网络是一类处理序列数据的神经网络。在本题用到的递归神经网络的架构如下图所示：



1)单步前向传播（RNN step forward）

我们要实现的第一段代码是单步前向传播。对于每一层神经网络，或每一个时刻，我们输入一个隐藏状态h_{t-1}(上一层神经网络的输出)，一个外部输入x_t;之后得到下一个隐藏状态h_t，以及该时刻的输出y_t。对应的数学表达式为

$$\begin{aligned} h &= f_w(h_{t-1}, x_t) \\ &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b) \end{aligned}$$

b为偏差值。 作业中，我们要在rnn_layers.py文件下实现如下函数：

```
def rnn_step_forward(x, prev_h, Wx, Wh, b):
```

```
    """
    输入:
    - x: 外部输入数据，维度 (N, D).
    - prev_h: 上一个时刻的隐藏状态，维度 (N, H)
    - Wx: x对应的权值矩阵，维度 (D, H)
    - Wh: 隐藏状态h对应的权值矩阵，维度 (H, H)
    - b: 偏差值 shape (H,)

    输出:
    - next_h: 下一个时态的隐藏状态，维度 (N, H)
    - cache: 计算梯度反向传播时需要用到的变量.
    """
    temp1 = np.dot(x,Wx)
    temp2 = np.dot(prev_h,Wh)
    cache=(x,prev_h,Wx,Wh,temp1+temp2+b)
    next_h = np.tanh(temp1+temp2+b)
    return next_h, cache
```

单层的前向传播并不难，实现的时候只要熟练使用numpy底下的矩阵运算即可。一个要注意的点是这里的激活函数是作用于向量中的每一个元素（element wise）。

2）梯度单步反向传播（RNN step backward）

可以说几乎所有训练神经网络的算法都是基于梯度下降（gradient descent）的，所以如何获得每个节点，以及相应的参数对应的梯度至关重要。正如之前的作业中训练神经网络时做的，求梯度反向传播其实只是在不断的应用求导的链式法则。假设最终的损失函数为$\boldsymbol{E}$，在进行反向传播时，我们得到了上一层传来的梯度$\frac{dE}{dh_t}$，我们需要计算$\frac{dE}{dh_{t-1}}$,$\frac{dE}{dx_t}$,$\frac{dE}{dW_{xh}}$,$\frac{dE}{dW_{hh}}$和$\frac{dE}{db}$。还记得前向传播的公式为

$$\begin{aligned} h &= f_w(h_{t-1}, x_t) \\ &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b) \end{aligned}$$

假设$a = W_{hh}h_{t-1} + W_{xh}x_t + b$，那么利用求导的链式法则，我们会有：

$$\begin{aligned} \frac{dE}{dh_{t-1}} &= \frac{dE}{dh_t}\frac{dh_t}{dh_{t-1}} \\ &= \frac{dE}{dh_t}\frac{dh_t}{da}\frac{da}{dh_{t-1}} \end{aligned}$$

而$\frac{dh_t}{da}$就是在对激活函数求导。$\frac{da}{dh_{t-1}}$就是权值矩阵W_{hh}。 要注意一点的是，为了更易于阐明概念，上式做了一些简化，$\frac{dE}{dh_t}$和$\frac{dE}{dh_t}$都是向量，他们的相乘是逐项（element wise）相乘。之后与$\frac{da}{dh_{t-1}}$相乘，则是一般的向量乘矩阵，编程时注意对应的维度就好。 我们可以用相似的思路获得$\frac{dE}{dx_t}$,$\frac{dE}{dW_{xh}}$,$\frac{dE}{dW_{hh}}$和$\frac{dE}{db}$的计算方式。 作业中，我们要实现如下函数：

```
def rnn_step_backward(dnext_h, cache):
    """
    输入:
    - dnext_h: 下一层传来的梯度
    - cache: 前向传播存下来的值

    输出

    - dx: 输入x的梯度，维度(N, D)
```

```
    - dprev_h: 上一层隐藏状态的梯度，维度(N, H)
    - dWx: 权值矩阵Wxh的梯度，维度(D, H)
    - dWh: 权值矩阵Whh的梯度，维度(H, H)
    - db: 偏差值b的梯度，维度（H,）
    """
    x=cache[0]
    h=cache[1]
    Wx=cache[2]
    Wh=cache[3]
    # cache[4]对应着公式中的a
    cacheD=cache[4]
    N,H=h.shape
    # 计算激活函数的导数
    temp = np.ones((N,H))-np.square(np.tanh(cacheD))
    delta = np.multiply(temp,dnext_h)
    # 计算x的梯度
    tempx = np.dot(Wx,delta.T)
    dx=tempx.T
    # h的提督
    temph = np.dot(Wh,delta.T)
    dprev_h=temph.T
    # Wxh的梯度
    dWx = np.dot(x.T,delta)
    # Whh
    dWh = np.dot(h.T,delta)
    # b的梯度
    tempb = np.sum(delta,axis=0)
    db=tempb.T
    return dx, dprev_h, dWx, dWh, db
```

3）前向传播（RNN forward）

我们完成了单步前向传播之后，对于整个递归神经网络的前向传播过程就是单步前向传播的循环，而且在本题中，每一层神经网络都共享了参数W_{xh}，W_{hh}，b。对应的代码如下：

```
def rnn_forward(x, h0, Wx, Wh, b):
    """

    输入:
    - x: 整个序列的输入数据，维度 (N, T, D).
    - h0: 初始隐藏层，维度 (N, H)
    - Wx: 权值矩阵Wxh，维度 (D, H)
    - Wh: 权值矩阵Whh，维度 (H, H)
    - b: 偏差值，维度 (H,)

    输出:
    - h: 整个序列的隐藏状态，维度 (N, T, H).
    - cache: 反向传播时需要的变量
    """
    N,T,D=x.shape
    N,H=h0.shape
    prev_h=h0
    # 之前公式中对应的a
```

```
    h1=np.empty([N,T,H])
    # 隐藏状态h的序列
    h2=np.empty([N,T,H])
    # 滞后h一个时间点
    h3=np.empty([N,T,H])
    for i in range(0, T):
        #单步前向传播
        temp_h, cache_temp = rnn_step_forward(x[:,i,:], prev_h, Wx, Wh, b)
        #记录下需要的变量
        h3[:,i,:]=prev_h
        prev_h=temp_h
        h2[:,i,:]=temp_h
        h1[:,i,:]=cache_temp[4]
    cache=(x,h3,Wx,Wh,h1)
    return h2, cache
```

4）梯度反向传播（RNN backward）

和前向传播一样，我们已经有了单步反向传播，编程时反向传播就是单步反向传播的循环。另外因为每一层神经网络都共享了参数W_{xh}，W_{hh}，b，最终的$\frac{dE}{dW_{xh}}$是每一层计算得到的$\frac{dE}{dW_{xh}}$的和。另外两个参数也一样。

```
def rnn_backward(dh, cache):
    """
    输入:
    - dh: 损失函数关于每一个隐藏层的梯度，维度 (N, T, H)
    - cache: 前向传播时存的变量
    输出
    - dx: 每一层输入x的梯度，维度(N, T, D)
    - dh0: 初始隐藏状态的梯度，维度(N, H)
    - dWx: 权值矩阵Wxh的梯度，维度(D, H)
    - dWh: 权值矩阵Whh的梯度，维度(H, H)
    - db: 偏差值b的梯度，维度（H,）
    """
    x=cache[0]
    N,T,D=x.shape
    N,T,H=dh.shape
    #初始化
    dWx=np.zeros((D,H))
    dWh=np.zeros((H,H))
    db=np.zeros(H)
    dout=dh
    dx=np.empty([N,T,D])
    dh=np.empty([N,T,H])
    #当前时刻隐藏状态对应的梯度
    hnow=np.zeros([N,H])

    for k in range(0, T):
        i=T-1-k
        #我们要注意，除了上一层传来的梯度，我们每一层都有输出，对应的误差函数也会传入梯度
        hnow=hnow+dout[:,i,:]
        cacheT=(cache[0][:,i,:],cache[1][:,i,:],cache[2],cache[3],cache[4][:,i,:])
        #单步反向传播
```

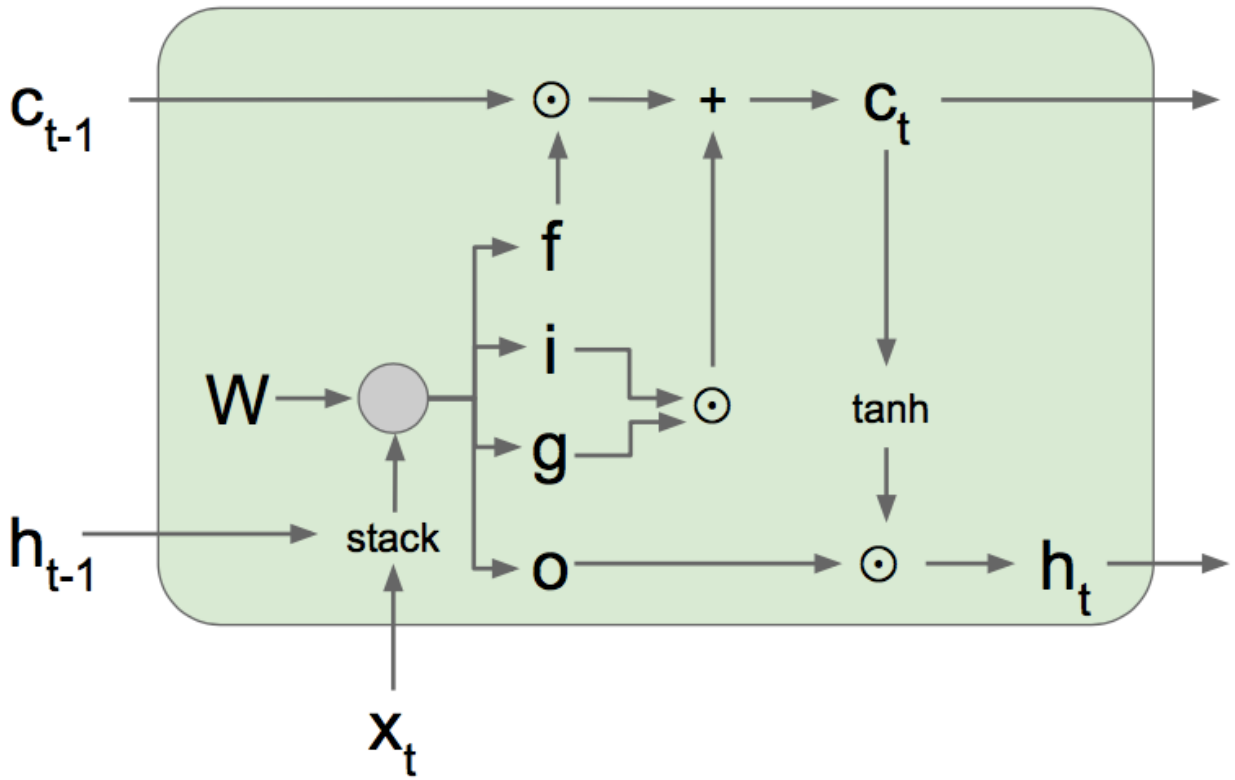
```
        dx_temp, dprev_h, dWx_temp, dWh_temp, db_temp = rnn_step_backward(hnow, cacheT)
        hnow=dprev_h
        dx[:,i,:]=dx_temp
        #将每一层共享的参数对应的梯度相加
        dWx=dWx+dWx_temp
        dWh=dWh+dWh_temp
        db=db+db_temp

    dh0=hnow
    return dx, dh0, dWx, dWh, db
```

至此我们完成了图一所示的递归神经网络的训练。该训练算法也被称为Back-Propagation Through Time(BPTT)，是最典型的训练递归神经网络的算法之一。 我们可以试着跑一下上面的代码，然后可以通过问题中每一小段之后的error测试。

3. 长短时记忆单元（Long-Short Term Memory, LSTM）

这一部分讲述问题2中关于长短时记忆单元的代码。 在训练上一部分里实现的递归神经网络时，我们往往会碰到随着反向传播层数的增加，梯度越来越小，或者梯度越来越大的问题。为了解决这个问题，就有了以LSTM为基础单元的神经。一个LSTM单元如下图所示：



1）单步前向传播（LSTM step forward）

和一般的递归神经网络相似，在LSTM中，每一步我们会得到一个输入$x_t \in \mathbb{R}^D$，之前的隐藏状态$h_{t-1} \in \mathbb{R}^H$。除此之外，LSTM还保存每一个H维的单元状态，所以我们还会收到之前单元的状态$c_{t-1} \in \mathbb{R}^H$。LSTM的参数则是输入到隐藏层权值矩阵$W_x \in \mathbb{R}^{4H \times D}$，一个隐藏层到隐藏层的权值矩阵$W_h \in \mathbb{R}^{4H \times H}$和偏差向量$b \in \mathbb{R}^{4H}$。

每一步，我们先计算一个激活向量$a \in \mathbb{R}^{4H}$，$a = W_x x_t + W_h h_{t-1} + b$。然后我们该向量分成$a_i,a_f,a_o,a_g \in \mathbb{R}^H$四个子向量。其中$a_i$是向量$a$的前$H$个元素，$a_f$是$a$之后的$H$个元素，以此类推。然后我们如下计算计算输入门$i \in \mathbb{R}^H$，遗忘门$f \in \mathbb{R}^H$，输出门$o \in \mathbb{R}^H$和块输入$g \in \mathbb{R}^H$：

$$i = \sigma(a_i) \quad f = \sigma(a_f) \quad o = \sigma(a_o) \quad g = \tanh(a_g)$$

其中σ时Sigmoid激活函数，$\tanh$是双曲正切函数。 最终，我们如下计算下一个单元的状态c_t以及下一个隐藏状态h_t:

$$c_t = f \odot c_{t-1} + i \odot g \qquad h_t = o \odot \tanh c_t$$

其中$\odot$表示逐项相乘。 至此，我们得到了LSTM每一步前向传播的规则，在代码中只要按照公式编写即可。

```
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):
    """
    输入:
    - x: 输入数据，维度 (N, D)
    - prev_h: 上一层隐藏状态，维度 (N, H)
    - prev_c: 上一层单元状态，维度 (N, H)
    - Wx: 输入到隐藏层的权值矩阵，维度 (D, 4H)
    - Wh: 隐藏层到隐藏层的权值矩阵，维度 (H, 4H)
    - b: 偏差，维度 (4H,)

    输出:
    - next_h: 下一个隐藏状态，维度 (N, H)
    - next_c: 下一个单元状态，维度 (N, H)
    - cache: 反向传播时需要的变量.
    """

    N,H=prev_h.shape
    A= x.dot(Wx)+prev_h.dot(Wh)+b
    ai=A[:,0:H]
    af=A[:,H:2*H]
    ao=A[:,2*H:3*H]
    ag=A[:,3*H:4*H]

    i=sigmoid(ai)
    f=sigmoid(af)
    o=sigmoid(ao)
    g=np.tanh(ag)

    next_c=np.multiply(f,prev_c)+np.multiply(i,g)
    next_h=np.multiply(o,np.tanh(next_c))

    cache=(x,prev_h,prev_c,i,f,o,g,Wx,Wh,next_c,A)

    return next_h, next_c, cache
```

2）梯度单步反向传播（LSTM step backward）

虽然同样是利用了求导的链式法则，但是由于多了一条关于c_t传播的路径，LSTM的反向传播会比一般的递归神经网络复杂一些。假设传到当前单元的导数为$\frac{dE}{dh_t}$，$\frac{dE}{dc_t}$，我们先讨论如何计算$\frac{dE}{dc_{t-1}}$。$\frac{dE}{dc_{t-1}}$可以由两条路径产生，一条是通过$\frac{dE}{dc_t}$,一条是通过$\frac{dE}{dh_t}$.

$$\begin{aligned}\frac{dE}{dc_{t-1}} &= \frac{dE}{dh_t}\frac{dh_t}{dc_{t-1}} + \frac{dE}{dc_t}\frac{dc_t}{dc_{t-1}} \\ &= \frac{dE}{dh_t}\frac{dh_t}{dc_t}\frac{dc_t}{dc_{t-1}} + \frac{dE}{dc_t}\frac{dc_t}{dc_{t-1}}\end{aligned}$$

这里$\frac{dh_t}{dc_t} = o\frac{d\tanh c_t}{dc_t}$，$\frac{dc_t}{dc_{t-1}} = f$。这里的所有乘法都是逐项（element wise）相乘。为了计算关于$\frac{dE}{dh_{t-1}}$等梯度，我们先要得到到i, f, g, o这些节点的梯度。到o的梯度易得:

$$\frac{dE}{do} = \frac{dE}{dh_t}\frac{dh}{do}$$

其中$\frac{dh}{do} = \tanh c_t$。 到$f$的梯度需要注意的是依然有两条路径,

$$\begin{aligned}\frac{dE}{df} &= \frac{dE}{dh_t}\frac{dh_t}{df} + \frac{dc_t}{df} \\ &= \frac{dE}{dh_t}\frac{dh_t}{dc_t}\frac{dc_t}{df} + \frac{dc_t}{df}\end{aligned}$$

其中$\frac{dc_t}{df} = c_{t-1}$。 到$i$和$g$的梯度计算过程相似，下面以计算$\frac{dE}{di}$为例:

$$\begin{aligned}\frac{dE}{di} &= \frac{dE}{dh_t}\frac{dh_t}{di} + \frac{dc_t}{di} \\ &= \frac{dE}{dh_t}\frac{dh_t}{dc_t}\frac{dc_t}{di} + \frac{dc_t}{di}\end{aligned}$$

其中$\frac{dc_t}{di} = g$。 而o, g, i, f是a_o, a_g, a_i, a_g通告激活函数而得，所以我们有:

$$\frac{dE}{da_o} = \frac{dE}{do}\frac{do}{da_o} \qquad \frac{dE}{da_g} = \frac{dE}{dg}\frac{dg}{da_g}$$

$$\frac{dE}{da_i} = \frac{dE}{di}\frac{di}{da_i} \qquad \frac{dE}{da_f} = \frac{dE}{df}\frac{df}{da_f}$$

将$\frac{dE}{da_i}, \frac{dE}{da_i}, \frac{dE}{da_i}, \frac{dE}{da_i}$相连，我们得到$\frac{dE}{da}$。至此，之后求$\frac{dE}{dh_{t-1}}$，$\frac{dE}{dW_{hh}}$等梯度的步骤和之前求一般递归神经网络反向传播公式时相同，这里略去。注意以上的乘法都为逐项相乘。 最后，逐一将上面求梯度的步骤用代码实现:

```
def lstm_step_backward(dnext_h, dnext_c, cache):
    """
    输入:
    - dnext_h: 下一层传来关于h的梯度，维度 (N, H)
    - dnext_c: 下一层传来关于c的梯度，维度 (N, H)
    - cache: 前向传播存的变量

    输出
    - dx: 输入x的梯度，维度(N, D)
    - dprev_h: 上一层隐藏状态的梯度，维度(N, H)

    - dprev_c: 上一层单元状态的梯度，维度(N, H)
```

```
    - dWx: 权值矩阵Wxh的梯度，维度(D，4H)
    - dWh: 权值矩阵Whh的梯度，维度(H，4H)
    - db: 偏差值b的梯度，维度（4H，）
    """
    #提取cache中的变量
    N,H=dnext_h.shape
    f=cache[4]
    o=cache[5]
    i=cache[3]
    g=cache[6]
    nc=cache[9]
    prev_c=cache[2]
    prev_x=cache[0]
    prev_h=cache[1]
    A=cache[10]
    ai=A[:,0:H]
    af=A[:,H:2*H]
    ao=A[:,2*H:3*H]
    ag=A[:,3*H:4*H]
    Wx=cache[7]
    Wh=cache[8]
    #计算到c_t-1的梯度
    dc_c=np.multiply(dnext_c,f)
    dc_h_temp=np.multiply(dnext_h,o)
    temp = np.ones_like(nc)-np.square(np.tanh(nc))
    temp2=np.multiply(temp,f)
    dprev_c=np.multiply(temp2,dc_h_temp)+dc_c

    #计算(dE/dh)(dh/dc)
    dc_from_h=np.multiply(dc_h_temp,temp)

    dtotal_c=dc_from_h+dnext_c

    #计算到o,f,i,g的梯度
    tempo=np.multiply(np.tanh(nc),dnext_h)
    tempf=np.multiply(dtotal_c,prev_c)
    tempi=np.multiply(dtotal_c,g)
    tempg=np.multiply(dtotal_c,i)

    #计算到ao,ai,af,ag的梯度
    tempao=np.multiply(tempo,np.multiply(o,np.ones_like(o)-o))
    tempai=np.multiply(tempi,np.multiply(i,np.ones_like(o)-i))
    tempaf=np.multiply(tempf,np.multiply(f,np.ones_like(o)-f))
    dtanhg = np.ones_like(ag)-np.square(np.tanh(ag))
    tempag=np.multiply(tempg,dtanhg)

    #计算各参数的梯度
    TEMP=np.concatenate((tempai,tempaf,tempao,tempag),axis=1)
    dx=TEMP.dot(Wx.T)
    dprev_h=TEMP.dot(Wh.T)
    xt=prev_x.T
    dWx=xt.dot(TEMP)

    ht=prev_h.T
```

```
    dWh=ht.dot(TEMP)
    db=np.sum(TEMP,axis=0).T

    return dx, dprev_h, dprev_c, dWx, dWh, db
```

3）前向传播（LSTM forward）

我们已经得到了LSTM单步前向传播，和一般递归神经网络一样，这里只要写个循环。

```
def lstm_forward(x, h0, Wx, Wh, b):
    """

    输入:
    - x: 整个序列的输入数据，维度 (N, T, D).
    - h0: 初始隐藏层，维度 (N, H)
    - Wx: 权值矩阵Wxh，维度 (D, H)
    - Wh: 权值矩阵Whh，维度 (H, H)
    - b: 偏差值，维度 (H,)

    输出:
    - h: 整个序列的隐藏状态，维度 (N, T, H).
    - cache: 反向传播时需要的变量

    """
    N,T,D=x.shape
    N,H=h0.shape
    prev_h=h0
    #以下的变量为反向传播时所需
    h3=np.empty([N,T,H])
    h4=np.empty([N,T,H])
    I=np.empty([N,T,H])
    F=np.empty([N,T,H])
    O=np.empty([N,T,H])
    G=np.empty([N,T,H])
    NC=np.empty([N,T,H])
    AT=np.empty([N,T,4*H])
    h2=np.empty([N,T,H])
    prev_c=np.zeros_like(prev_h)
    for i in range(0, T):
        h3[:,i,:]=prev_h
        h4[:,i,:]=prev_c
        #单步前向传播
        next_h, next_c, cache_temp = lstm_step_forward(x[:,i,:], prev_h, prev_c, Wx, Wh, b)
        prev_h=next_h
        prev_c=next_c
        h2[:,i,:]=prev_h
        I[:,i,:]=cache_temp[3]
        F[:,i,:]=cache_temp[4]
        O[:,i,:]=cache_temp[5]
        G[:,i,:]=cache_temp[6]
        NC[:,i,:]=cache_temp[9]
        AT[:,i,:]=cache_temp[10]
```

```
    cache=(x,h3,h4,I,F,O,G,Wx,Wh,NC,AT)



    return h2, cache
```

4）梯度反向传播（LSTM backward）

和之前一样，关于单步反向传播的循环

```
def lstm_backward(dh, cache):
    """
    输入:
    - dh: 损失函数关于每一个隐藏层的梯度，维度 (N, T, H)
    - cache: 前向传播时存的变量
    输出
    - dx: 每一层输入x的梯度，维度(N, T, D)
    - dh0: 初始隐藏状态的梯度，维度(N, H)
    - dWx: 权值矩阵Wxh的梯度，维度(D, 4H)
    - dWh: 权值矩阵Whh的梯度，维度(H, 4H)
    - db: 偏差值b的梯度，维度（4H,）
    """
    x=cache[0]
    N,T,D=x.shape
    N,T,H=dh.shape

    dWx=np.zeros((D,4*H))
    dWh=np.zeros((H,4*H))
    db=np.zeros(4*H)
    dout=dh
    dx=np.empty([N,T,D])
    hnow=np.zeros([N,H])
    cnow=np.zeros([N,H])
    for k in range(0, T):
        i=T-1-k
        hnow=hnow+dout[:,i,:]
        cacheT=(cache[0][:,i,:],cache[1][:,i,:],cache[2][:,i,:],cache[3][:,i,:],cache[4]
[:,i,:],cache[5][:,i,:],cache[6][:,i,:],cache[7],cache[8],cache[9][:,i,:],cache[10][:,i,:])

        dx_temp, dprev_h, dprev_c, dWx_temp, dWh_temp, db_temp = lstm_step_backward(hnow, cnow,
cacheT)
        hnow=dprev_h
        cnow=dprev_c
        dx[:,i,:]=dx_temp
        dWx=dWx+dWx_temp
        dWh=dWh+dWh_temp
        db=db+db_temp

    dh0=hnow


    return dx, dh0, dWx, dWh, db
```

4. 图片注释(captioning)生成

我们已经完成了一般递归神经网络和以LSTM为基础的递归神经网络的编写。下一步，我们将整合所写的内容，运用到图片注释生成之中。

1）词嵌入（Word embedding）

在深度学习系统中，我们通常将词用向量表示。词表中的每一个词都将和一个向量关联，这些向量则会和系统的其余部分一样进行训练。

```
def word_embedding_forward(x, W):
    """
    Inputs:
    输入:
    - x: 维度为(N,T)的整数列，每一项是相应词汇对应的索引。
    - W: 维度为(V,D)权值矩阵，V是词表的大小，每一列对应着一个词的向量表示
    Returns a tuple of:
    输出
    - out: 维度为(N, T, D)，由所有输入词的词向量所组成
    - cache: 反向传播时需要的变量
    """
    #这里只要把x里的整数对应到词向量表中即可
    out = W[x, :]
    cache = x, W
    return out, cache
```

2）词嵌入梯度反向传播

```
def word_embedding_backward(dout, cache):
    """
    输入
    - dout: 梯度， 维度(N, T, D)
    - cache: 前向传播存的变量

    Returns:
    输出:
    - dW: 词嵌入矩阵的梯度，维度 (V, D).
    """
    # 提示: 使用np.add.at函数
    x, W = cache
    dW=np.zeros_like(W)
    # 在x指定的位置将dout加到dW上
    np.add.at(dW, x, dout)
    return dW
```

3）图片注释生成系统

除了上面我们写的函数，本题已经为我们提供了线性运算层和softmax损失函数的计算代码，我们可以直接使用。至此，我们有了写一个图片注释生成系统所需的所有模块，下面这段代码我们来将这些小模块整合到一块。 整体架构是，图像特征对应于初始隐藏状态，在训练时真实的图片注释为每一时刻的输入，输出序列为RNN/LSTM对图片注释序列的预测。

```
    def loss(self, features, captions):
        """
        计算训练时RNN/LSTM的损失函数。我们输入图像特征和正确的图片注释，使用RNN/LSTM计算损失函数和所有参数的梯度
        输入:
        - features: 输入图像特征，维度 (N, D)
        - captions: 正确的图像注释; 维度为(N, T)的整数列

        输出一个tuple：
        - loss: 标量损失函数值
        - grads: 所有参数的梯度
        """

        """
        这里将captions分成了两个部分，captions_in是除了最后一个词外的所有词，是输入到RNN/LSTM的输入；captions_out是除了第一个词外的所有词，是RNN/LSTM期望得到的输出。
        """
        captions_in = captions[:, :-1]
        captions_out = captions[:, 1:]

        # 之后会用到
        mask = (captions_out != self._null)

        # 从图像特征到初始隐藏状态的权值矩阵和偏差值
        W_proj, b_proj = self.params['W_proj'], self.params['b_proj']

        # 词嵌入矩阵
        W_embed = self.params['W_embed']

        # RNN/LSTM参数
        Wx, Wh, b = self.params['Wx'], self.params['Wh'], self.params['b']

        # 每一隐藏层到输出的权值矩阵和偏差
        W_vocab, b_vocab = self.params['W_vocab'], self.params['b_vocab']

        N,D=features.shape

        # 用线性变换从图像特征值得到初始隐藏状态，将产生维度为(N,H)的数列
        out, cache_affine = temporal_affine_forward(features.reshape(N,1,D), W_proj, b_proj)
        N,t,H=out.shape
        h0=out.reshape(N,H)

        # 用词嵌入层将captions_in中词的索引转换成词响亮，得到一个维度为(N, T, W)的数列
        word_out,cache_word=word_embedding_forward(captions_in, W_embed)

        # 用RNN/LSTM处理输入的词向量，产生每一层的隐藏状态,维度为(N,T,H),这里演示的是LSTM的
        # RNN forward
        # hidden, cache_hidden = rnn_forward(word_out, h0, Wx, Wh, b)
```

```
        # LSTM forward
        hidden, cache_hidden = lstm_forward(word_out, h0, Wx, Wh, b)


        # 用线性变换计算每一步隐藏层对应的输出(得分)，维度(N, T, V)
        out_vo, cache_vo = temporal_affine_forward(hidden, W_vocab, b_vocab)

        # 用softmax函数计算损失，真实值为captions_out，用mask忽视所有向量中<NULL>词汇
        loss, dx = temporal_softmax_loss(out_vo[:,:,:],captions_out, mask, verbose=False)

        #之后再逐步计算反向传播，得到对应的参数
        dx_affine,dW_vocab,db_vocab=temporal_affine_backward(dx, cache_vo)
        grads['W_vocab']=dW_vocab
        grads['b_vocab']=db_vocab

        # RNN backward
        # dx_hidden, dh0, dWx, dWh, db = rnn_backward(dx_affine, cache_hidden)
        # LSTM bascward
        dx_hidden, dh0, dWx, dWh, db = lstm_backward(dx_affine, cache_hidden)

        grads['Wx']=dWx
        grads['Wh']=dWh
        grads['b']=db

        dW_embed = word_embedding_backward(dx_hidden, cache_word)
        grads['W_embed']=dW_embed

        dx_initial,dW_proj,db_proj=temporal_affine_backward(dh0.reshape(N,t,H), cache_affine)
        grads['W_proj']=dW_proj
        grads['b_proj']=db_proj

        return loss, grads
```

至此，我们可以通过图片和真实的图片注释训练我们的RNN/LSTM系统。在下一步中我们将介绍如何采样一个图片注释。

4）图片注释采样

该问题的最后一步，我们要对得到的图片，用训练好的RNN/LSTM来生成一个图片注释。

```
    def sample(self, features, max_length=30):
        """
        和上一段代码不同的地方是，这里我们没有了真实的图像注释。所以每一时刻的输入这样获得：计算隐藏层对应的输出，这些输出表示所有词汇表中词汇的得分，取得分最高的词汇，作为下一时刻的输入。其他与上一节里的代码相同。因为不能同时获得所有输入，我们必须循环应用RNN/LSTM step forward。
        对于LSTM，还需记录单元c的状态，初始值为0

        Inputs:
        - features: Array of input image features of shape (N, D).
        - max_length: Maximum length T of generated captions.
        输入：
        - captions: 输入图像特征，维度 (N, D)
        - max_length: 生成的注释的最长长度
```

```
        输出:
        - captions: 采样得到的注释，维度(N, max_length), 每个元素是词汇的索引
        """
        N = features.shape[0]
        captions = self._null * np.ones((N, max_length), dtype=np.int32)

        # 参数
        W_proj, b_proj = self.params['W_proj'], self.params['b_proj']
        W_embed = self.params['W_embed']
        Wx, Wh, b = self.params['Wx'], self.params['Wh'], self.params['b']
        W_vocab, b_vocab = self.params['W_vocab'], self.params['b_vocab']

        N,D=features.shape

        # 用线性变换从图像特征值得到初始隐藏状态，将产生维度为(N,H)的数列
        out, cache_affine = temporal_affine_forward(features.reshape(N,1,D), W_proj, b_proj)
        N,t,H=out.shape
        h0=out.reshape(N,H)
        h=h0
        # 初始输入
        x0=W_embed[[1,1],:]
        x_input=x0
        captions[:,0]=[1,1]
        # prev_c only for lstm
        prev_c=np.zeros_like(h)
        for i in range(0,max_length-1):
            # RNN step forward
            # next_h, _ = rnn_step_forward(x_input, h, Wx, Wh, b)

            # LSTM step forward
            next_h, next_c, cache = lstm_step_forward(x_input, h, prev_c, Wx, Wh, b)
            # only for lstm
            prev_c=next_c
            #计算每一层输出
            out_vo, cache_vo = temporal_affine_forward(next_h.reshape(N,1,H), W_vocab, b_vocab)
            #找到输出最大值的项作为下一时刻的输入
            index=np.argmax(out_vo,axis=2)
            x_input=np.squeeze(W_embed[index,:])
            h=next_h
            #记录其索引
            captions[:,i+1]=np.squeeze(index)

        return captions
```