## Recurrent Neural Network

- In this assignment, you will implement your first Recurrent Neural Network in numpy.
- Recurrent Neural Networks (RNN) are very effective for Natural Language Processing and other sequence tasks because they have "memory". They can read inputs $x\langle t \rangle$ (such as words) one at a time, and remember some information/context through the hidden layer activations that get passed from one time-step to the next.

1

## 1 - Packages

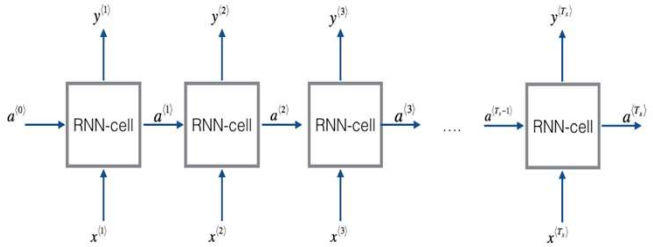
Let's first import all the packages that you will need during this assignment.

```
import numpy as np
from rnn_utils import *
```

2

## 1 - Forward propagation

The basic RNN that you will implement has the structure below.



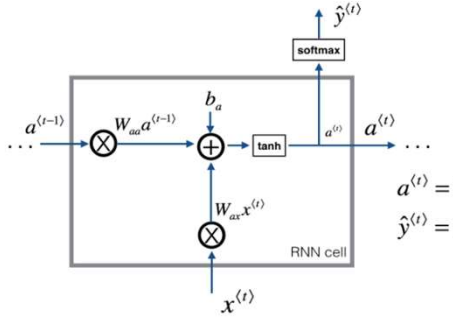

**Steps**:

- Implement the calculations needed for one time-step of the RNN.
- Implement a loop over $Tx$Tx time-steps in order to process all the inputs, one at a time.

3

## 1.1. RNN-Cell

A Recurrent neural network can be seen as the repetition of a single cell. You are first going to implement the computations for a single time-step. The following figure describes the operations for a single time-step of an RNN cell.




$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

4

## FUNCTION: rnn_cell_forward

Implements a single forward step of the RNN-cell

**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)

5

```
def rnn_cell_forward(xt, a_prev, parameters):


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

    # 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)

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

    return a_next, yt_pred, cache
```

6

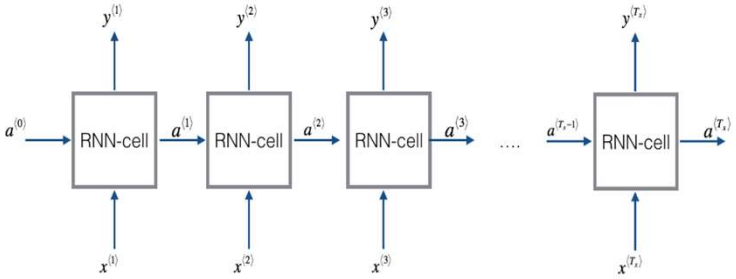
```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa": Waa, "Wax": Wax, "Wya": Wya, "ba": ba, "by": by}

a_next, yt_pred, cache = rnn_cell_forward(xt, a_prev, parameters)
print("a_next[4] = ", a_next[4])
print("a_next.shape = ", a_next.shape)
print("yt_pred[1] =", yt_pred[1])
print("yt_pred.shape = ", yt_pred.shape)
```

```
a_next[4] =  [ 0.59584544  0.18141802  0.61311866  0.99808218  0.85016201  0.99980978
 -0.18887155  0.99815551  0.6531151   0.82872037]
a_next.shape =  (5, 10)
yt_pred[1] = [0.9888161  0.01682021 0.21140899 0.36817467 0.98988387 0.88945212
 0.36920224 0.9966312  0.9982559  0.17746526]
yt_pred.shape =  (2, 10)
```

7

## 1.2 - RNN forward pass

Each cell takes as input the hidden state from the previous cell ($a\langle t-1\rangle$) and the current time-step's input data ($x\langle t\rangle$). It outputs a hidden state ($a\langle t\rangle$) and a prediction ($y\langle t\rangle$) for this time-step.




8

## FUNCTION: rnn_forward

Implement the forward propagation of the recurrent neural network

**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 -- 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)

9

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

    # 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


    # 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(0,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)

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

    return a, y_pred, caches
```

10

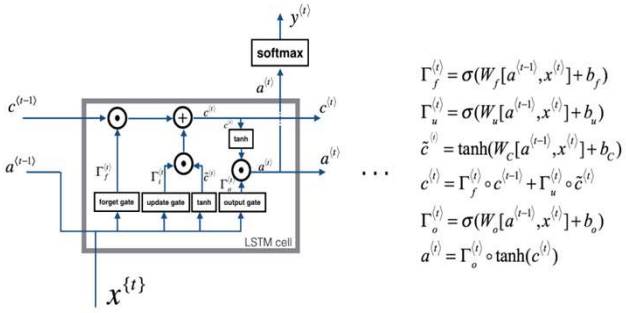
```
np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa": Waa, "Wax": Wax, "Wya": Wya, "ba": ba, "by": by}

a, y_pred, caches = rnn_forward(x, a0, parameters)
print("a[4][1] = ", a[4][1])
print("a.shape = ", a.shape)
print("y_pred[1][3] =", y_pred[1][3])
print("y_pred.shape = ", y_pred.shape)
print("caches[1][1][3] =", caches[1][1][3])
print("len(caches) = ", len(caches))
```

```
a[4][1] =  [-0.99999375  0.77911235 -0.99861469 -0.99833267]
a.shape =  (5, 10, 4)
y_pred[1][3] = [0.79560373 0.86224861 0.11118257 0.81515947]
y_pred.shape =  (2, 10, 4)
caches[1][1][3] = [-1.1425182  -0.34934272 -0.20889423  0.58662319]
len(caches) =  2
```

11

## 2 - Long Short-Term Memory (LSTM) network

LSTM-cell. This tracks and updates a "cell state" or memory variable $c\langle t \rangle$ at every time-step, which can be different from $a\langle t \rangle$




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

12

## FUNCTION: lstm_cell_forward

Implement a single forward step of the LSTM-cell

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)

13

## FUNCTION: lstm_cell_forward (2)

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)

14

```
def lstm_cell_forward(xt, a_prev, c_prev, parameters):

    # 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

    # Concatenate a_prev and xt (≈3 lines)
    concat = np.zeros((n_x + n_a, m))
    concat[: n_a, :] = a_prev
    concat[n_a :, :] = xt
```

15

```
    # 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 = c_prev*ft +  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)

    # 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
```

16

```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
c_prev = np.random.randn(5,10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5,1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5,1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5,1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5,1)
Wy = np.random.randn(2,5)
by = np.random.randn(2,1)

parameters = {"Wf": Wf, "Wi": Wi, "Wo": Wo, "Wc": Wc, "Wy": Wy, "bf":
              bf, "bi": bi, "bo": bo, "bc": bc, "by": by}

a_next, c_next, yt, cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)
print("a_next[4] = ", a_next[4])
print("a_next.shape = ", c_next.shape)
print("c_next[2] = ", c_next[2])
print("c_next.shape = ", c_next.shape)
print("yt[1] =", yt[1])
print("yt.shape = ", yt.shape)
print("cache[1][3] =", cache[1][3])
print("len(cache) = ", len(cache))
```

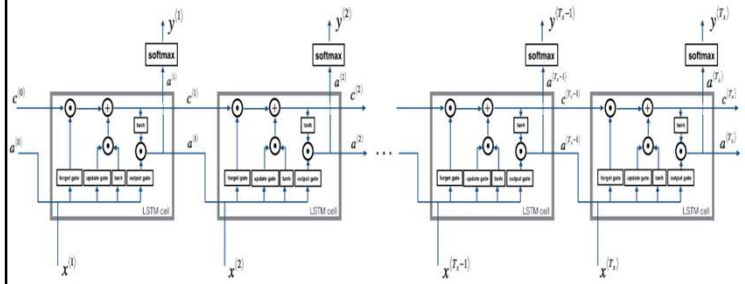
17

```
a_next[4] =  [-0.66408471  0.0036921   0.02088357  0.22834167 -0.85575339  0.00138482
  0.76566531  0.34631421 -0.00215674  0.43827275]
a_next.shape =  (5, 10)
c_next[2] =  [ 0.63267805  1.00570849  0.35504474  0.20690913 -1.64566718  0.11832942
  0.76449811 -0.0981561  -0.74348425 -0.26810932]
c_next.shape =  (5, 10)
yt[1] = [0.79913913 0.15986619 0.22412122 0.15606108 0.97057211 0.31146381
 0.00943007 0.12666353 0.39380172 0.07828381]
yt.shape =  (2, 10)
cache[1][3] = [-0.16263996  1.03729328  0.72938082 -0.54101719  0.02752074 -0.30821874
  0.07651101 -1.03752894  1.41219977 -0.37647422]
len(cache) =  10
```

18

## 2.2 - Forward pass for LSTM

Now that you have implemented one step of an LSTM, you can now iterate this over this using a for-loop to process a sequence of $Tx$Tx inputs.




19

## FUNCTION: lstm_forward

Implement the forward propagation of the recurrent neural network using an LSTM-cell

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)

20

## GRADED FUNCTION: lstm_forward

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)

21

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

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

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

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

    # Initialize a_next and c_next (≈2 lines)
    a_next = a0
    c_next = np.zeros((n_a, m))
```

22

```
    # loop over all time-steps
    for t in range(0, 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)

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

    return a, y, c, caches
```

23

```
np.random.seed(1)
x = np.random.randn(3,10,7)
a0 = np.random.randn(5,10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5,1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5,1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5,1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5,1)
Wy = np.random.randn(2,5)
by = np.random.randn(2,1)

parameters = {"Wf": Wf, "Wi": Wi, "Wo": Wo, "Wc": Wc, "Wy": Wy,
              "bf": bf, "bi": bi, "bo": bo, "bc": bc, "by": by}

a, y, c, caches = lstm_forward(x, a0, parameters)
print("a[4][3][6] = ", a[4][3][6])
print("a.shape = ", a.shape)
print("y[1][4][3] =", y[1][4][3])
print("y.shape = ", y.shape)
print("caches[1][1[1]] =", caches[1][1][1])
print("c[1][2][1]", c[1][2][1])
print("len(caches) = ", len(caches))
```

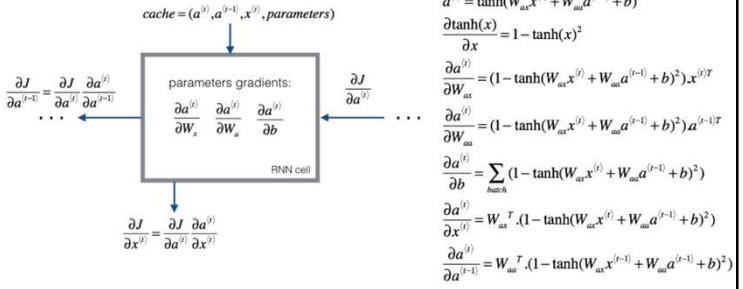
24

```
a[4][3][6] =  0.17211776753291672
a.shape =  (5, 10, 7)
y[1][4][3] = 0.9508734618501101
y.shape =  (2, 10, 7)
caches[1][1[1]] = [ 0.82797464  0.23009474  0.76201118 -0.22232814 -0.20075807  0.18656139
  0.41005165]
c[1][2][1] -0.8555449167181982
len(caches) =  2
```

25

## 3 - Backpropagation in recurrent neural networks

### 3.1 Basic RNN backward pass




26

## FUNCTION: rnn_cell_backward

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)

27

```python
def rnn_cell_backward(da_next, cache):

    # 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"]

    # compute the gradient of tanh with respect to a_next (≈1 line)
    dtanh = 1 - np.power(a_next,2)

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

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

    # compute the gradient with respect to b (≈1 line)
    dba = np.sum(da_next*dtanh, axis=1, keepdims=True)

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

    return gradients
```

28

```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Wax = np.random.randn(5,3)
Waa = np.random.randn(5,5)
Wya = np.random.randn(2,5)
b = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "ba": ba, "by": by}

a_next, yt, cache = rnn_cell_forward(xt, a_prev, parameters)

da_next = np.random.randn(5,10)
gradients = rnn_cell_backward(da_next, cache)
print("gradients[\"dxt\"][1][2] =", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape =", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3] =", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape =", gradients["da_prev"].shape)
print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape =", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape =", gradients["dWaa"].shape)
print("gradients[\"dba\"][4] =", gradients["dba"][4])
print("gradients[\"dba\"].shape =", gradients["dba"].shape)
```

29

```
gradients["dxt"][1][2] = -0.4605641030588796
gradients["dxt"].shape = (3, 10)
gradients["da_prev"][2][3] = 0.08429686538067718
gradients["da_prev"].shape = (5, 10)
gradients["dWax"][3][1] = 0.3930818739219303
gradients["dWax"].shape = (5, 3)
gradients["dWaa"][1][2] = -0.2848395578696067
gradients["dWaa"].shape = (5, 5)
gradients["dba"][4] = [0.80517166]
gradients["dba"].shape = (5, 1)
```

30

## Backward pass through the RNN

- Computing the gradients of the cost with respect to $a\langle t\rangle$a⟨t⟩ at every time-step $t$t is useful because it is what helps the gradient backpropagate to the previous RNN-cell.
- To do so, you need to iterate through all the time steps starting at the end, and at each step, you increment th overall $dba, dWaa, dWax$ and you store $dx$d.

31

## FUNCTION: rnn_backward

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, the input's weight matrix, numpy-array of shape (n_a, n_x)
- dWaa -- Gradient, the hidden state's weight matrix, numpy-array of (n_a, n_a)
- dba -- Gradient w.r.t the bias, of shape (n_a, 1)

32

```
def rnn_backward(da, caches):

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

    # 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((n_x, m, T_x))
    dWax = np.zeros((n_a, n_x))
    dWaa = np.zeros((n_a, n_a))
    dba = np.zeros((n_a, 1))
    da0 = np.zeros((n_a, m))
    da_prevt = np.zeros((n_a, m))
```

33

```
    # Loop through all the time steps
    for t in reversed(range(0, 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

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

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

    return gradients
```

34

```
np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Wax = np.random.randn(5,3)
Waa = np.random.randn(5,5)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "ba": ba, "by": by}
a, y, caches = rnn_forward(x, a0, parameters)
da = np.random.randn(5, 10, 4)
gradients = rnn_backward(da, caches)

print("gradients[\"dx\"][1][2] =", gradients["dx"][1][2])
print("gradients[\"dx\"].shape =", gradients["dx"].shape)
print("gradients[\"da0\"][2][3] =", gradients["da0"][2][3])
print("gradients[\"da0\"].shape =", gradients["da0"].shape)
print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape =", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape =", gradients["dWaa"].shape)
print("gradients[\"dba\"][4] =", gradients["dba"][4])
print("gradients[\"dba\"].shape =", gradients["dba"].shape)
```

35

```
gradients["dx"][1][2] = [-2.07101689 -0.59255627  0.02466855  0.01483317]
gradients["dx"].shape = (3, 10, 4)
gradients["da0"][2][3] = -0.31494237512664996
gradients["da0"].shape = (5, 10)
gradients["dWax"][3][1] = 11.264104496527777
gradients["dWax"].shape = (5, 3)
gradients["dWaa"][1][2] = 2.303333126579893
gradients["dWaa"].shape = (5, 5)
gradients["dba"][4] = [-0.74747722]
gradients["dba"].shape = (5, 1)
```

36

## 3.2 - LSTM backward pass

### 3.2.1 One Step backward

The LSTM backward pass is slighltly more complicated than the forward one. We have provided you with all the equations for the LSTM backward pass below.

**gate derivatives**

$$d\Gamma_o^{\langle t \rangle} = da_{next} * \tanh(c_{next}) * \Gamma_o^{\langle t \rangle} * (1 - \Gamma_o^{\langle t \rangle})$$

$$d\tilde{c}^{\langle t \rangle} = dc_{next} * \Gamma_u^{\langle t \rangle} + \Gamma_o^{\langle t \rangle}(1 - \tanh(c_{next})^2) * i_t * da_{next} * \tilde{c}^{\langle t \rangle} * (1 - \tanh(\tilde{c})^2)$$

$$d\Gamma_u^{\langle t \rangle} = dc_{next} * \tilde{c}^{\langle t \rangle} + \Gamma_o^{\langle t \rangle}(1 - \tanh(c_{next})^2) * \tilde{c}^{\langle t \rangle} * da_{next} * \Gamma_u^{\langle t \rangle} * (1 - \Gamma_u^{\langle t \rangle})$$

$$d\Gamma_f^{\langle t \rangle} = dc_{next} * \tilde{c}_{prev} + \Gamma_o^{\langle t \rangle}(1 - \tanh(c_{next})^2) * c_{prev} * da_{next} * \Gamma_f^{\langle t \rangle} * (1 - \Gamma_f^{\langle t \rangle})$$

37

**parameter derivatives**

$$dW_f = d\Gamma_f^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T$$

$$dW_u = d\Gamma_u^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T$$

$$dW_c = d\tilde{c}^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T$$

$$dW_o = d\Gamma_o^{\langle t \rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T$$

To calculate $db_f, db_u, db_c, db_o$ you just need to sum across the horizontal (axis= 1) axis on $d\Gamma_f^{\langle t \rangle}, d\Gamma_u^{\langle t \rangle}, d\tilde{c}^{\langle t \rangle}, d\Gamma_o^{\langle t \rangle}$ respectively. Note that you should have the `keep_dims = True` option.

Finally, you will compute the derivative with respect to the previous hidden state, previous memory state, and input.

$$da_{prev} = W_f^T * d\Gamma_f^{\langle t \rangle} + W_u^T * d\Gamma_u^{\langle t \rangle} + W_c^T * d\tilde{c}^{\langle t \rangle} + W_o^T * d\Gamma_o^{\langle t \rangle}$$

Here, the weights for equations 13 are the first n_a, (i.e. $W_f = W_f[:n_a, :]$ etc...)

$$dc_{prev} = dc_{next}\Gamma_f^{\langle t \rangle} + \Gamma_o^{\langle t \rangle} * (1 - \tanh(c_{next})^2) * \Gamma_f^{\langle t \rangle} * da_{next}$$

$$dx^{\langle t \rangle} = W_f^T * d\Gamma_f^{\langle t \rangle} + W_u^T * d\Gamma_u^{\langle t \rangle} + W_c^T * d\tilde{c}_t + W_o^T * d\Gamma_o^{\langle t \rangle}$$

where the weights for equation 15 are from n_a to the end, (i.e. $W_f = W_f[n_a :, :]$ etc...)

38

## FUNCTION: lstm_cell_backward

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
- dbo -- Gradient w.r.t. biases of the output gate, of shape (n_a, 1)

39

## FUNCTION: lstm_cell_backward

**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 – the weight matrix of the forget gate, numpy array of shape (n_a, n_a + n_x)
- dWi -- the weight matrix of the update gate, numpy array of shape (n_a, n_a + n_x)
- dWc -- the weight matrix of the memory gate, numpy array of (n_a, n_a + n_x)
- dWo -- 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)

40

```python
def lstm_cell_backward(da_next, dc_next, cache):

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

    # 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)
    dcct = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*it
    dit = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*cct
    dft = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*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))
```

41

```python
    # Compute parameters related derivatives. Use equations (11)-(14) (≈8 lines)
    concat = np.zeros((n_x + n_a, m))
    concat[: n_a, :] = a_prev
    concat[n_a :, :] = 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_prevx = np.dot(parameters['Wf'].T, dft) + np.dot(parameters['Wo'].T, dot) +
               np.dot(parameters['Wi'].T, dit) + np.dot(parameters['Wc'].T, dcct)
    da_prev = da_prevx[: n_a, :]
    dc_prev = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*ft
    dxt = da_prevx[n_a :, :]

    # 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
```

42

```python
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
c_prev = np.random.randn(5,10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5,1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5,1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5,1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5,1)
Wy = np.random.randn(2,5)
by = np.random.randn(2,1)

parameters = {"Wf": Wf, "Wi": Wi, "Wo": Wo, "Wc": Wc, "Wy": Wy, "bf": bf,
              "bi": bi, "bo": bo, "bc": bc, "by": by}

a_next, c_next, yt, cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)

da_next = np.random.randn(5,10)
dc_next = np.random.randn(5,10)
gradients = lstm_cell_backward(da_next, dc_next, cache)
```

43

```python
print("gradients[\"dxt\"][1][2] =", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape =", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3] =", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape =", gradients["da_prev"].shape)
print("gradients[\"dc_prev\"][2][3] =", gradients["dc_prev"][2][3])
print("gradients[\"dc_prev\"].shape =", gradients["dc_prev"].shape)
print("gradients[\"dWf\"][3][1] =", gradients["dWf"][3][1])
print("gradients[\"dWf\"].shape =", gradients["dWf"].shape)
print("gradients[\"dWi\"][1][2] =", gradients["dWi"][1][2])
print("gradients[\"dWi\"].shape =", gradients["dWi"].shape)
print("gradients[\"dWc\"][3][1] =", gradients["dWc"][3][1])
print("gradients[\"dWc\"].shape =", gradients["dWc"].shape)
print("gradients[\"dWo\"][1][2] =", gradients["dWo"][1][2])
print("gradients[\"dWo\"].shape =", gradients["dWo"].shape)
print("gradients[\"dbf\"][4] =", gradients["dbf"][4])
print("gradients[\"dbf\"].shape =", gradients["dbf"].shape)
print("gradients[\"dbi\"][4] =", gradients["dbi"][4])
print("gradients[\"dbi\"].shape =", gradients["dbi"].shape)
print("gradients[\"dbc\"][4] =", gradients["dbc"][4])
print("gradients[\"dbc\"].shape =", gradients["dbc"].shape)
print("gradients[\"dbo\"][4] =", gradients["dbo"][4])
print("gradients[\"dbo\"].shape =", gradients["dbo"].shape)
```

44

```
gradients["dxt"][1][2] = 3.2305591151091884
gradients["dxt"].shape = (3, 10)
gradients["da_prev"][2][3] = -0.0639621419710924
gradients["da_prev"].shape = (5, 10)
gradients["dc_prev"][2][3] = 0.7975220387970015
gradients["dc_prev"].shape = (5, 10)
gradients["dWf"][3][1] = -0.1479548381644968
gradients["dWf"].shape = (5, 8)
gradients["dWi"][1][2] = 1.0574980552259903
gradients["dWi"].shape = (5, 8)
gradients["dWc"][3][1] = 2.304562163687667
gradients["dWc"].shape = (5, 8)
gradients["dWo"][1][2] = 0.3313115952892109
gradients["dWo"].shape = (5, 8)
gradients["dbf"][4] = [0.18864637]
gradients["dbf"].shape = (5, 1)
gradients["dbi"][4] = [-0.40142491]
gradients["dbi"].shape = (5, 1)
gradients["dbc"][4] = [0.25587763]
gradients["dbc"].shape = (5, 1)
gradients["dbo"][4] = [0.13893342]
gradients["dbo"].shape = (5, 1)
```

45

## References

- https://github.com/Kulbear/deep-learning-coursera/blob/master/Sequence%20Models/Building%20a%20Recurrent%20Neural%20Network%20-%20Step%20by%20Step%20-%20v2.ipynb

46