Neural Machine Translation

Welcome to your first programming assignment for this week!

You will build a Neural Machine Translation (NMT) model to translate human readable dates ("25th of June, 2009") into machine readable dates ("2009-06-25"). You will do this using an attention model, one of the most sophisticated sequence to sequence models.

This notebook was produced together with NVIDIA's Deep Learning Institute.

Let's load all the packages you will need for this assignment.

In [14]:

```
from keras.layers import Bidirectional, Concatenate, Permute, Dot, Input, LSTM, M
from keras.layers import RepeatVector, Dense, Activation, Lambda
from keras.optimizers import Adam
from keras.utils import to_categorical
from keras.models import load_model, Model
import keras.backend as K
import numpy as np

from faker import Faker
import random
from tqdm import tqdm
from babel.dates import format_date
from nmt_utils import *
import matplotlib.pyplot as plt
%matplotlib inline
print("end3...")
```

```
end3...
```

1 - Translating human readable dates into machine readable dates

The model you will build here could be used to translate from one language to another, such as translating from English to Hindi. However, language translation requires massive datasets and usually takes days of training on GPUs. To give you a place to experiment with these models even without using massive datasets, we will instead use a simpler "date translation" task.

The network will input a date written in a variety of possible formats (*e.g.* "*the 29th of August 1958*", "*03/30/1968*", "*24 JUNE 1987*") and translate them into standardized, machine readable dates (*e.g.* "*1958-08-29*", "*1968-03-30*", "*1987-06-24*"). We will have the network learn to output dates in the common machine-readable format YYYY-MM-DD.

1.1 - Dataset

We will train the model on a dataset of 10000 human readable dates and their equivalent, standardized, machine readable dates. Let's run the following cells to load the dataset and print some examples.

In [15]:

```
m = 10000
dataset, human_vocab, machine_vocab, inv_machine_vocab = load_dataset(m)
print("len(dataset):",len(dataset));
print("human_vocab:",human_vocab);
print("machine_vocab:",machine_vocab);
```

```
len(dataset): 10000
human_vocab: {' ': 0, '.': 1, '/': 2, '0': 3, '1': 4, '2': 5, '3': 6
, '4': 7, '5': 8, '6': 9, '7': 10, '8': 11, '9': 12, 'a': 13, 'b': 1
4, 'c': 15, 'd': 16, 'e': 17, 'f': 18, 'g': 19, 'h': 20, 'i': 21, 'j
': 22, 'l': 23, 'm': 24, 'n': 25, 'o': 26, 'p': 27, 'r': 28, 's': 29
, 't': 30, 'u': 31, 'v': 32, 'w': 33, 'y': 34, '<unk>': 35, '<pad>':
36}
machine_vocab: {'-': 0, '0': 1, '1': 2, '2': 3, '3': 4, '4': 5, '5':
6, '6': 7, '7': 8, '8': 9, '9': 10}
```

In [4]:

```
dataset[:10]
```

Out[4]:

```
[('9 may 1998', '1998-05-09'),
 ('10.09.70', '1970-09-10'),
 ('4/28/90', '1990-04-28'),
 ('thursday january 26 1995', '1995-01-26'),
 ('monday march 7 1983', '1983-03-07'),
 ('sunday may 22 1988', '1988-05-22'),
 ('tuesday july 8 2008', '2008-07-08'),
 ('08 sep 1999', '1999-09-08'),
 ('1 jan 1981', '1981-01-01'),
 ('monday may 22 1995', '1995-05-22')]
```

You've loaded:

- `dataset`: a list of tuples of (human readable date, machine readable date)
- `human_vocab`: a python dictionary mapping all characters used in the human readable dates to an integer-valued index
- `machine_vocab`: a python dictionary mapping all characters used in machine readable dates to an integer-valued index. These indices are not necessarily consistent with `human_vocab`.
- `inv_machine_vocab`: the inverse dictionary of `machine_vocab`, mapping from indices back to characters.

Let's preprocess the data and map the raw text data into the index values. We will also use Tx=30 (which we assume is the maximum length of the human readable date; if we get a longer input, we would have to truncate it) and Ty=10 (since "YYYY-MM-DD" is 10 characters long).

In [16]:

```
Tx = 30
Ty = 10
X, Y, Xoh, Yoh = preprocess_data(dataset, human_vocab, machine_vocab, Tx, Ty)

print("X.shape:", X.shape)
print("Y.shape:", Y.shape)
print("Xoh.shape:", Xoh.shape)
print("Yoh.shape:", Yoh.shape)
```

```
X.shape: (10000, 30)
Y.shape: (10000, 10)
Xoh.shape: (10000, 30, 37)
Yoh.shape: (10000, 10, 11)
```

You now have:

- `X`: a processed version of the human readable dates in the training set, where each character is replaced by an index mapped to the character via `human_vocab`. Each date is further padded to T_x T_x values with a special character (< pad >). `X.shape = (m, Tx)`
- `Y`: a processed version of the machine readable dates in the training set, where each character is replaced by the index it is mapped to in `machine_vocab`. You should have `Y.shape = (m, Ty)`.
- `Xoh`: one-hot version of `X`, the "1" entry's index is mapped to the character thanks to `human_vocab`. `Xoh.shape = (m, Tx, len(human_vocab))`
- `Yoh`: one-hot version of `Y`, the "1" entry's index is mapped to the character thanks to `machine_vocab`. `Yoh.shape = (m, Tx, len(machine_vocab))`. Here, `len(machine_vocab) = 11` since there are 11 characters ('-' as well as 0-9).

Lets also look at some examples of preprocessed training examples. Feel free to play with `index` in the cell below to navigate the dataset and see how source/target dates are preprocessed.

In [9]:

```
index = 0
print("Source date:", dataset[index][0])
print("Target date:", dataset[index][1])
print()
print("Source after preprocessing (indices):", X[index])
print("Target after preprocessing (indices):", Y[index])
print()
print("Source after preprocessing (one-hot):", Xoh[index])
print("Target after preprocessing (one-hot):", Yoh[index])
```

```
Source date: 9 may 1998
Target date: 1998-05-09

Source after preprocessing (indices): [12  0 24 13 34  0  4 12 12 11
36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36
 36 36 36 36 36]
Target after preprocessing (indices): [ 2 10 10  9  0  1  6  0  1 10
]

Source after preprocessing (one-hot): [[ 0.  0.  0. ...,  0.  0.  0.
]
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 0.  0.  0. ...,  0.  0.  1.]
 [ 0.  0.  0. ...,  0.  0.  1.]
 [ 0.  0.  0. ...,  0.  0.  1.]]
Target after preprocessing (one-hot): [[ 0.  0.  1.  0.  0.  0.  0.
0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
```

2 - Neural machine translation with attention

If you had to translate a book's paragraph from French to English, you would not read the whole paragraph, then close the book and translate. Even during the translation process, you would read/re-read and focus on the parts of the French paragraph corresponding to the parts of the English you are writing down.

The attention mechanism tells a Neural Machine Translation model where it should pay attention to at any step.

2.1 - Attention mechanism

In this part, you will implement the attention mechanism presented in the lecture videos. Here is a figure to remind you how the model works. The diagram on the left shows the attention model. The diagram on the right shows what one "Attention" step does to calculate the attention variables $\alpha^{\langle t,t' \rangle}$ $\alpha^{\langle t,t' \rangle}$, which are used to compute the context variable $context^{\langle t \rangle}$ $context^{\langle t \rangle}$ for each timestep in the output ($t = 1, \ldots, T_y$ $t = 1, \ldots, T_y$).

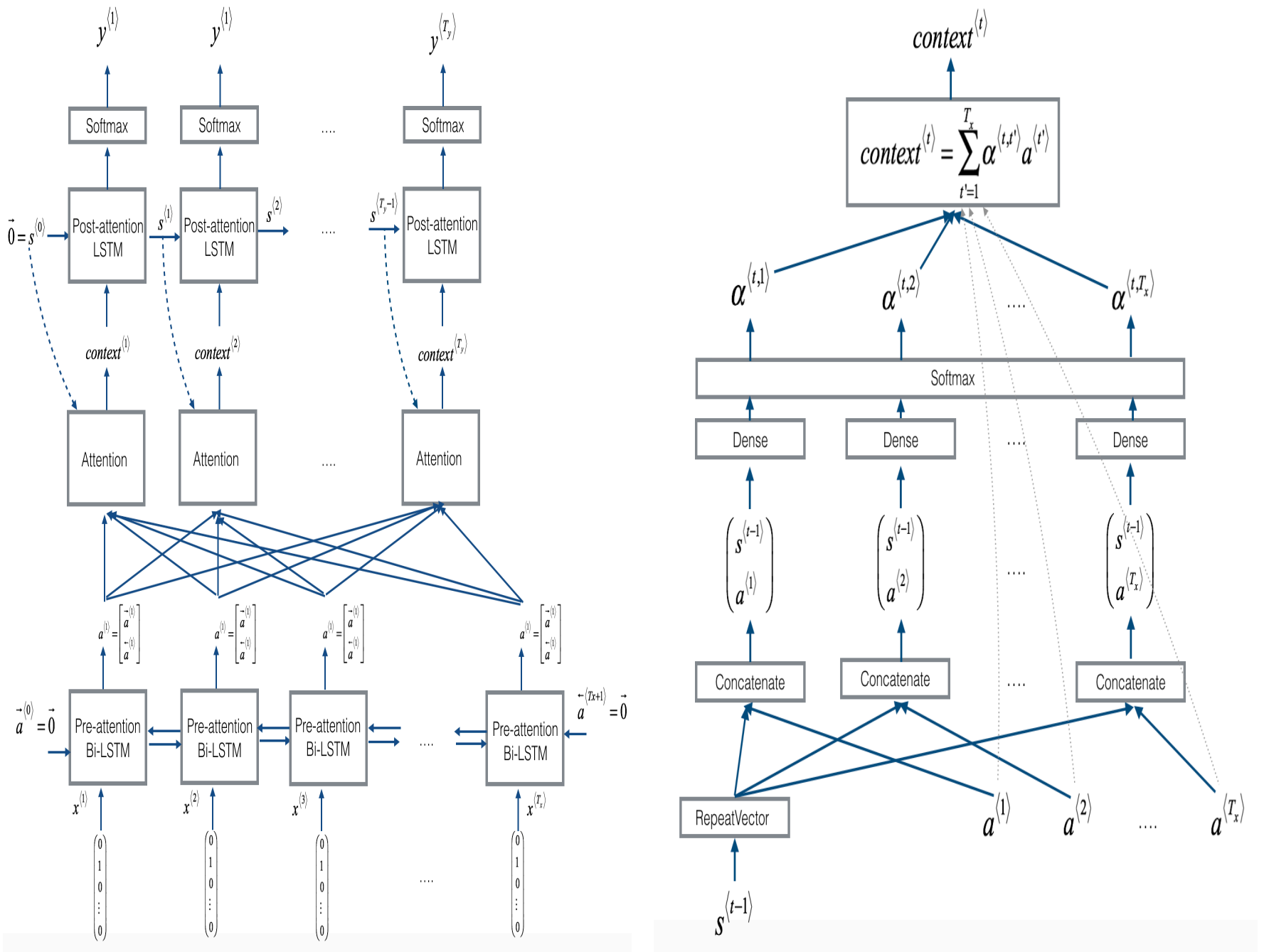

Figure 1: Neural machine translation with attention

Here are some properties of the model that you may notice:

- There are two separate LSTMs in this model (see diagram on the left). Because the one at the bottom of the picture is a Bi-directional LSTM and comes *before* the attention mechanism, we will call it *pre-attention* Bi-LSTM. The LSTM at the top of the diagram comes *after* the attention mechanism, so we will call it the *post-attention* LSTM. The pre-attention Bi-LSTM goes through T_x T_x time steps; the post-attention LSTM goes through T_y T_y time steps.
- The post-attention LSTM passes $s^{\langle t \rangle}$, $c^{\langle t \rangle}$ $s^{\langle t \rangle}$, $c^{\langle t \rangle}$ from one time step to the next. In the lecture videos, we were using only a basic RNN for the post-activation sequence model, so the state captured by the RNN output activations $s^{\langle t \rangle}$ $s^{\langle t \rangle}$. But since we are using an LSTM here, the LSTM has both the output activation $s^{\langle t \rangle}$ $s^{\langle t \rangle}$ and the hidden cell state $c^{\langle t \rangle}$ $c^{\langle t \rangle}$. However, unlike previous text generation examples (such as Dinosaurus in week 1), in this model the post-activation LSTM at time tt does will not take the specific generated $y^{\langle t-1 \rangle}$ $y^{\langle t-1 \rangle}$ as input; it only takes $s^{\langle t \rangle}$ $s^{\langle t \rangle}$ and $c^{\langle t \rangle}$ $c^{\langle t \rangle}$ as input. We have designed the model this way, because (unlike language generation where adjacent characters are highly correlated) there isn't as strong a dependency between the previous character and the next character in a YYYY-MM-DD date.
- We use $a^{\langle t \rangle} = [\overrightarrow{a}^{\langle t \rangle}; \overleftarrow{a}^{\langle t \rangle}]$ $a^{\langle t \rangle} = [\vec{a}^{\langle t \rangle}; \overleftarrow{a}^{\langle t \rangle}]$ to represent the concatenation of the activations of both the forward-direction and backward-directions of the pre-attention Bi-LSTM.
- The diagram on the right uses a `RepeatVector` node to copy $s^{\langle t-1 \rangle}$ $s^{\langle t-1 \rangle}$'s value T_x T_x times, and then `Concatenation` to concatenate $s^{\langle t-1 \rangle}$ $s^{\langle t-1 \rangle}$ and $a^{\langle t \rangle}$ $a^{\langle t \rangle}$ to compute $e^{\langle t,t' \rangle}$ $e^{\langle t,t' \rangle}$, which is then passed through a softmax to compute $\alpha^{\langle t,t' \rangle}$ $\alpha^{\langle t,t' \rangle}$. We'll explain how to use `RepeatVector` and `Concatenation` in Keras below.

Lets implement this model. You will start by implementing two functions: `one_step_attention()` and `model()`.

1) `one_step_attention()`: At step tt, given all the hidden states of the Bi-LSTM ($[a^{<1>}, a^{<2>}, \ldots, a^{<T_x>}]$ $[a^{<1>}, a^{<2>}, \ldots, a^{<T_x>}]$) and the previous hidden state of the second LSTM ($s^{<t-1>}$ $s^{<t-1>}$), `one_step_attention()` will compute the attention weights ($[\alpha^{<t,1>}, \alpha^{<t,2>}, \ldots, \alpha^{<t,T_x>}]$ $[\alpha^{<t,1>}, \alpha^{<t,2>}, \ldots, \alpha^{<t,T_x>}]$) and output the context vector (see Figure 1 (right) for details):

$$context^{<t>} = \sum_{t'=0}^{T_x} \alpha^{<t,t'>} a^{<t'>} \tag{1}$$

$$context^{<t>} = \sum_{t'=0}^{T_x} \alpha^{<t,t'>} a^{<t'>}$$

Note that we are denoting the attention in this notebook $context^{\langle t \rangle}$ $context^{\langle t \rangle}$. In the lecture videos, the context was denoted $c^{\langle t \rangle}$ $c^{\langle t \rangle}$, but here we are calling it $context^{\langle t \rangle}$ $context^{\langle t \rangle}$ to avoid confusion with the (post-attention) LSTM's internal memory cell variable, which is sometimes also denoted $c^{\langle t \rangle}$ $c^{\langle t \rangle}$.

2) `model()`: Implements the entire model. It first runs the input through a Bi-LSTM to get back $[a^{<1>}, a^{<2>}, \ldots, a^{<T_x>}]$ $[a^{<1>}, a^{<2>}, \ldots, a^{<T_x>}]$. Then, it calls `one_step_attention()` T_y T_y times (`for` loop). At each iteration of this loop, it gives the computed context vector $c^{<t>}$ $c^{<t>}$ to the second LSTM, and runs the output of the LSTM through a dense layer with softmax activation to generate a prediction $y^{<t>}$ $\hat{y}^{<t>}$.

Exercise: Implement `one_step_attention()`. The function `model()` will call the layers in `one_step_attention()` T_y T_y using a for-loop, and it is important that all T_y T_y copies have the same weights. I.e., it should not re-initiaiize the weights every time. In other words, all T_y T_y steps should have

shared weights. Here's how you can implement layers with shareable weights in Keras:

1. Define the layer objects (as global variables for examples).
2. Call these objects when propagating the input.

We have defined the layers you need as global variables. Please run the following cells to create them. Please check the Keras documentation to make sure you understand what these layers are: RepeatVector() (https://keras.io/layers/core/#repeatvector), Concatenate() (https://keras.io/layers/merge/#concatenate), Dense() (https://keras.io/layers/core/#dense), Activation() (https://keras.io/layers/core/#activation), Dot() (https://keras.io/layers/merge/#dot).

In [17]:

```
# Defined shared layers as global variables
repeator = RepeatVector(Tx)
concatenator = Concatenate(axis=-1)
densor1 = Dense(10, activation = "tanh")
densor2 = Dense(1, activation = "relu")
activator = Activation(softmax, name='attention_weights') # We are using a custom
dotor = Dot(axes = 1)
print("end2...")
```

```
end2...
```

Now you can use these layers to implement `one_step_attention()`. In order to propagate a Keras tensor object X through one of these layers, use `layer(X)` (or `layer([X,Y])` if it requires multiple inputs.), e.g. `densor(X)` will propagate X through the `Dense(1)` layer defined above.

In [18]:

```
# GRADED FUNCTION: one_step_attention

def one_step_attention(a, s_prev):
    """
    Performs one step of attention: Outputs a context vector computed as a dot pr
    "alphas" and the hidden states "a" of the Bi-LSTM.

    Arguments:
    a -- hidden state output of the Bi-LSTM, numpy-array of shape (m, Tx, 2*n_a)
    s_prev -- previous hidden state of the (post-attention) LSTM, numpy-array of

    Returns:
    context -- context vector, input of the next (post-attetion) LSTM cell
    """

    ### START CODE HERE ###
    # Use repeator to repeat s_prev to be of shape (m, Tx, n_s) so that you can c
    s_prev = repeator(s_prev) #[m, Tx, n_s]
    # Use concatenator to concatenate a and s_prev on the last axis (≈ 1 line)
    concat = concatenator([a,s_prev]) #[m, Tx, n_s+2*n_a]
    # Use densor1 to propagate concat through a small fully-connected neural netw
    e = densor1(concat) #[m, Tx, 10]
    # Use densor2 to propagate e through a small fully-connected neural network t
    energies = densor2(e) #[m, Tx, 1]
    # Use "activator" on "energies" to compute the attention weights "alphas" (≈
    alphas = activator(energies) #[m, Tx, 1]
    # Use dotor together with "alphas" and "a" to compute the context vector to b
    context = dotor([alphas,a])
    ### END CODE HERE ###

    return context
```

You will be able to check the expected output of `one_step_attention()` after you've coded the `model()` function.

Exercise: Implement `model()` as explained in figure 2 and the text above. Again, we have defined global layers that will share weights to be used in `model()`.

In [19]:

```
n_a = 32
n_s = 64
post_activation_LSTM_cell = LSTM(n_s, return_state = True)
output_layer = Dense(len(machine_vocab), activation=softmax)
print("end3")
```

```
end3
```

Now you can use these layers T_y T_y times in a `for` loop to generate the outputs, and their parameters will not be reinitialized. You will have to carry out the following steps:

1. Propagate the input into a Bidirectional (https://keras.io/layers/wrappers/#bidirectional) LSTM (https://keras.io/layers/recurrent/#lstm)
2. Iterate for $t = 0, \ldots, T_y - 1$$t = 0, \ldots, T_y - 1$:
    A. Call `one_step_attention()` on $[\alpha^{<t,1>}, \alpha^{<t,2>}, \ldots, \alpha^{<t,T_x>}]$$[\alpha^{<t,1>}, \alpha^{<t,2>}, \ldots, \alpha^{<t,T_x>}]$ and $s^{<t-1>}$ $s^{<t-1>}$ to get the context vector $context^{<t>}$ $context^{<t>}$.
    B. Give $context^{<t>}$ $context^{<t>}$ to the post-attention LSTM cell. Remember pass in the previous hidden-state $s^{\langle t-1\rangle}$ $s^{\langle t-1\rangle}$ and cell-states $c^{\langle t-1\rangle}$ $c^{\langle t-1\rangle}$ of this LSTM using `initial_state= [previous hidden state, previous cell state]`. Get back the new hidden state $s^{<t>}$ $s^{<t>}$ and the new cell state $c^{<t>}$ $c^{<t>}$.
    C. Apply a softmax layer to $s^{<t>}$ $s^{<t>}$, get the output.
    D. Save the output by adding it to the list of outputs.
3. Create your Keras model instance, it should have three inputs ("inputs", $s^{<0>}$ $s^{<0>}$ and $c^{<0>}$ $c^{<0>}$) and output the list of "outputs".

In [28]:

```
# GRADED FUNCTION: model

def model(Tx, Ty, n_a, n_s, human_vocab_size, machine_vocab_size):
    """
    Arguments:
    Tx -- length of the input sequence
    Ty -- length of the output sequence
    n_a -- hidden state size of the Bi-LSTM
    n_s -- hidden state size of the post-attention LSTM
    human_vocab_size -- size of the python dictionary "human_vocab"
    machine_vocab_size -- size of the python dictionary "machine_vocab"

    Returns:
    model -- Keras model instance
    """

    # Define the inputs of your model with a shape (Tx,)
    # Define s0 and c0, initial hidden state for the decoder LSTM of shape (n_s,)
    X = Input(shape=(Tx, human_vocab_size)) #[Tx, human_vocab_size]
    s0 = Input(shape=(n_s,), name='s0')
    c0 = Input(shape=(n_s,), name='c0')
    s = s0
    c = c0

    # Initialize empty list of outputs
    outputs = []

    ### START CODE HERE ###

    # Step 1: Define your pre-attention Bi-LSTM. Remember to use return_sequences
    a = Bidirectional(LSTM(n_s,return_sequences=True))(X)

    # Step 2: Iterate for Ty steps
```

```
    for t in range(Ty):

        # Step 2.A: Perform one step of the attention mechanism to get back the c
        context = one_step_attention(a, s)

        # Step 2.B: Apply the post-attention LSTM cell to the "context" vector.
        # Don't forget to pass: initial_state = [hidden state, cell state] (≈ 1 l
        s, _, c = post_activation_LSTM_cell(context,initial_state=[s,c])

        # Step 2.C: Apply Dense layer to the hidden state output of the post-atte
        out = output_layer(s)

        # Step 2.D: Append "out" to the "outputs" list (≈ 1 line)
        outputs.append(out)

    # Step 3: Create model instance taking three inputs and returning the list of
    model = Model(inputs=[X,s0,c0], outputs=outputs,name='attention_model')

    ### END CODE HERE ###

    return model
print("end...")
```

```
end...
```

Run the following cell to create your model.

In [29]:

```
model = model(Tx, Ty, n_a, n_s, len(human_vocab), len(machine_vocab))
```

Let's get a summary of the model to check if it matches the expected output.

In [30]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Connected to
=================================================================
input_7 (InputLayer)         (None, 30, 37)            0
_________________________________________________________________
s0 (InputLayer)              (None, 64)                0
_________________________________________________________________
bidirectional_7 (Bidirectional) (None, 30, 128)        52224
input_7[0][0]
_________________________________________________________________
repeat_vector_2 (RepeatVector) (None, 30, 64)          0
s0[0][0]
```

Expected Output:

Here is the summary you should see

Total params:	52,960
Trainable params:	52,960
Non-trainable params:	0
bidirectional_1's output shape	(None, 30, 64)
repeat_vector_1's output shape	(None, 30, 64)
concatenate_1's output shape	(None, 30, 128)
attention_weights's output shape	(None, 30, 1)
dot_1's output shape	(None, 1, 64)
dense_3's output shape	(None, 11)

As usual, after creating your model in Keras, you need to compile it and define what loss, optimizer and metrics your are want to use. Compile your model using `categorical_crossentropy` loss, a custom Adam (https://keras.io/optimizers/#adam) optimizer (https://keras.io/optimizers/#usage-of-optimizers) (`learning rate = 0.005`, $\beta_1 = 0.9$, $\beta_2 = 0.999$, `decay = 0.01`) and `['accuracy']` metrics:

In [44]:

```
### START CODE HERE ### (≈2 lines)
opt = Adam(lr=0.005,beta_1=0.9,beta_2=0.999,decay=0.01)
model.compile(loss='categorical_crossentropy',optimizer=opt,metrics=['accuracy'])
### END CODE HERE ###
```

The last step is to define all your inputs and outputs to fit the model:

- You already have X of shape $(m = 10000, T_x = 30)$ containing the training examples.
- You need to create `s0` and `c0` to initialize your `post_activation_LSTM_cell` with 0s.
- Given the `model()` you coded, you need the "outputs" to be a list of 11 elements of shape (m, T_y). So that: `outputs[i][0], ..., outputs[i][Ty]` represent the true labels (characters) corresponding to the i^{th} training example (`X[i]`). More generally, `outputs[i][j]` is the true label of the j^{th} character in the i^{th} training example.

In [40]:

```
s0 = np.zeros((m, n_s))
c0 = np.zeros((m, n_s))
outputs = list(Yoh.swapaxes(0,1))
```

Let's now fit the model and run it for one epoch.

In [43]:

```
model.fit([Xoh, s0, c0], outputs, epochs=10, batch_size=100)
```

```
Epoch 1/10
10000/10000 [==============================] - 46s - loss: 8.1446 -
dense_6_loss_1: 0.1405 - dense_6_loss_2: 0.1162 - dense_6_loss_3: 0.
8303 - dense_6_loss_4: 1.9593 - dense_6_loss_5: 0.0382 - dense_6_los
s_6: 0.2686 - dense_6_loss_7: 1.7330 - dense_6_loss_8: 0.0298 - dens
e_6_loss_9: 0.9677 - dense_6_loss_10: 2.0610 - dense_6_acc_1: 0.9719
- dense_6_acc_2: 0.9723 - dense_6_acc_3: 0.6902 - dense_6_acc_4: 0.3
128 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9352 - dense_6_acc_7:
0.3972 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.5992 - dense_6_acc
_10: 0.2484
Epoch 2/10
10000/10000 [==============================] - 44s - loss: 7.2231 -
dense_6_loss_1: 0.1071 - dense_6_loss_2: 0.0917 - dense_6_loss_3: 0.
6896 - dense_6_loss_4: 1.7429 - dense_6_loss_5: 0.0247 - dense_6_los
s_6: 0.1993 - dense_6_loss_7: 1.5275 - dense_6_loss_8: 0.0208 - dens
e_6_loss_9: 0.8773 - dense_6_loss_10: 1.9421 - dense_6_acc_1: 0.9762
- dense_6_acc_2: 0.9760 - dense_6_acc_3: 0.7442 - dense_6_acc_4: 0.4
111 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9529 - dense_6_acc_7:
0.4637 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.6305 - dense_6_acc
_10: 0.2898
Epoch 3/10
10000/10000 [==============================] - 44s - loss: 6.4517 -
dense_6_loss_1: 0.0921 - dense_6_loss_2: 0.0789 - dense_6_loss_3: 0.
6132 - dense_6_loss_4: 1.4960 - dense_6_loss_5: 0.0192 - dense_6_los
```

```
s_6: 0.1635 - dense_6_loss_7: 1.3383 - dense_6_loss_8: 0.0162 - dens
e_6_loss_9: 0.8113 - dense_6_loss_10: 1.8231 - dense_6_acc_1: 0.9780
- dense_6_acc_2: 0.9784 - dense_6_acc_3: 0.7713 - dense_6_acc_4: 0.5
026 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9607 - dense_6_acc_7:
0.5315 - dense_6_acc_8: 0.9998 - dense_6_acc_9: 0.6557 - dense_6_acc
_10: 0.3282
Epoch 4/10
10000/10000 [==============================] - 46s - loss: 5.6963 -
dense_6_loss_1: 0.0796 - dense_6_loss_2: 0.0708 - dense_6_loss_3: 0.
5509 - dense_6_loss_4: 1.2647 - dense_6_loss_5: 0.0151 - dense_6_los
s_6: 0.1343 - dense_6_loss_7: 1.1308 - dense_6_loss_8: 0.0141 - dens
e_6_loss_9: 0.7524 - dense_6_loss_10: 1.6837 - dense_6_acc_1: 0.9802
- dense_6_acc_2: 0.9806 - dense_6_acc_3: 0.7985 - dense_6_acc_4: 0.5
749 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9664 - dense_6_acc_7:
0.6355 - dense_6_acc_8: 0.9999 - dense_6_acc_9: 0.6828 - dense_6_acc
_10: 0.3734
Epoch 5/10
10000/10000 [==============================] - 46s - loss: 4.9740 -
dense_6_loss_1: 0.0722 - dense_6_loss_2: 0.0658 - dense_6_loss_3: 0.
4824 - dense_6_loss_4: 1.0400 - dense_6_loss_5: 0.0126 - dense_6_los
s_6: 0.1130 - dense_6_loss_7: 0.9703 - dense_6_loss_8: 0.0111 - dens
e_6_loss_9: 0.6983 - dense_6_loss_10: 1.5083 - dense_6_acc_1: 0.9810
- dense_6_acc_2: 0.9803 - dense_6_acc_3: 0.8215 - dense_6_acc_4: 0.6
613 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9726 - dense_6_acc_7:
0.7039 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.7195 - dense_6_acc
_10: 0.4295
Epoch 6/10
10000/10000 [==============================] - 44s - loss: 4.2964 -
dense_6_loss_1: 0.0676 - dense_6_loss_2: 0.0613 - dense_6_loss_3: 0.
4433 - dense_6_loss_4: 0.8371 - dense_6_loss_5: 0.0111 - dense_6_los
s_6: 0.1043 - dense_6_loss_7: 0.8466 - dense_6_loss_8: 0.0090 - dens
e_6_loss_9: 0.6484 - dense_6_loss_10: 1.2675 - dense_6_acc_1: 0.9804
- dense_6_acc_2: 0.9807 - dense_6_acc_3: 0.8341 - dense_6_acc_4: 0.7
365 - dense_6_acc_5: 0.9998 - dense_6_acc_6: 0.9716 - dense_6_acc_7:
0.7460 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.7482 - dense_6_acc
_10: 0.5272
Epoch 7/10
10000/10000 [==============================] - 43s - loss: 3.4275 -
dense_6_loss_1: 0.0618 - dense_6_loss_2: 0.0548 - dense_6_loss_3: 0.
3777 - dense_6_loss_4: 0.5751 - dense_6_loss_5: 0.0099 - dense_6_los
s_6: 0.0918 - dense_6_loss_7: 0.7057 - dense_6_loss_8: 0.0083 - dens
e_6_loss_9: 0.5910 - dense_6_loss_10: 0.9514 - dense_6_acc_1: 0.9820
- dense_6_acc_2: 0.9821 - dense_6_acc_3: 0.8535 - dense_6_acc_4: 0.8
297 - dense_6_acc_5: 0.9999 - dense_6_acc_6: 0.9756 - dense_6_acc_7:
0.8001 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.7827 - dense_6_acc
_10: 0.6603
Epoch 8/10
10000/10000 [==============================] - 42s - loss: 2.7760 -
dense_6_loss_1: 0.0564 - dense_6_loss_2: 0.0500 - dense_6_loss_3: 0.
3401 - dense_6_loss_4: 0.4142 - dense_6_loss_5: 0.0076 - dense_6_los
s_6: 0.0814 - dense_6_loss_7: 0.5860 - dense_6_loss_8: 0.0078 - dens
e_6_loss_9: 0.5328 - dense_6_loss_10: 0.6996 - dense_6_acc_1: 0.9835
- dense_6_acc_2: 0.9836 - dense_6_acc_3: 0.8651 - dense_6_acc_4: 0.8
853 - dense_6_acc_5: 0.9999 - dense_6_acc_6: 0.9787 - dense_6_acc_7:
0.8395 - dense_6_acc_8: 0.9997 - dense_6_acc_9: 0.8054 - dense_6_acc
_10: 0.7712
Epoch 9/10
```

```
10000/10000 [==============================] - 42s - loss: 2.3059 -
dense_6_loss_1: 0.0524 - dense_6_loss_2: 0.0456 - dense_6_loss_3: 0.
3035 - dense_6_loss_4: 0.3235 - dense_6_loss_5: 0.0056 - dense_6_los
s_6: 0.0694 - dense_6_loss_7: 0.4822 - dense_6_loss_8: 0.0063 - dens
e_6_loss_9: 0.4627 - dense_6_loss_10: 0.5546 - dense_6_acc_1: 0.9846
- dense_6_acc_2: 0.9841 - dense_6_acc_3: 0.8731 - dense_6_acc_4: 0.9
107 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9827 - dense_6_acc_7:
0.8744 - dense_6_acc_8: 0.9999 - dense_6_acc_9: 0.8371 - dense_6_acc
_10: 0.8205
Epoch 10/10
10000/10000 [==============================] - 42s - loss: 1.9289 -
dense_6_loss_1: 0.0474 - dense_6_loss_2: 0.0406 - dense_6_loss_3: 0.
2721 - dense_6_loss_4: 0.2513 - dense_6_loss_5: 0.0045 - dense_6_los
s_6: 0.0645 - dense_6_loss_7: 0.3896 - dense_6_loss_8: 0.0056 - dens
e_6_loss_9: 0.4032 - dense_6_loss_10: 0.4501 - dense_6_acc_1: 0.9864
- dense_6_acc_2: 0.9865 - dense_6_acc_3: 0.8801 - dense_6_acc_4: 0.9
360 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9822 - dense_6_acc_7:
0.9053 - dense_6_acc_8: 0.9997 - dense_6_acc_9: 0.8502 - dense_6_acc
_10: 0.8551

Out[43]:

<keras.callbacks.History at 0x7f8540679fd0>
```

While training you can see the loss as well as the accuracy on each of the 10 positions of the output. The table below gives you an example of what the accuracies could be if the batch had 2 examples:

True labels 1	1	9	9	5	-	1	2	-	0	4
Predictions 1	1	9	9	5	-	1	0	-	0	5
True labels 1	1	9	6	8	-	0	1	-	0	4
Predictions 2	1	9	7	8	-	0	3	-	0	4
Index	1	2	3	4	5	6	7	8	9	10
Accuracy	1.0	1.0	0.5	1.0	1.0	1.0	0.0	1.0	1.0	0.5

Thus, `dense_2_acc_8: 0.89` means that you are predicting the 7th character of the output correctly 89% of the time in the current batch of data.

We have run this model for longer, and saved the weights. Run the next cell to load our weights. (By training a model for several minutes, you should be able to obtain a model of similar accuracy, but loading our model will save you time.)

In [38]:

```
model.load_weights('models/model.h5')
```

```
---------------------------------------------------------------------
-------
InvalidArgumentError                      Traceback (most recent cal
l last)
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/c
ommon_shapes.py in _call_cpp_shape_fn_impl(op, input_tensors_needed,
input_tensors_as_shapes_needed, debug_python_shape_fn, require_shape
_fn)
    670           graph_def_version, node_def_str, input_shapes,
input_tensors,
--> 671           input_tensors_as_shapes, status)
    672   except errors.InvalidArgumentError as err:

/opt/conda/lib/python3.6/contextlib.py in __exit__(self, type, value
, traceback)
     88             try:
---> 89                 next(self.gen)
     90             except StopIteration:
```

You can now see the results on new examples.

In [45]:

```
EXAMPLES = ['3 May 1979', '5 April 09', '21th of August 2016', 'Tue 10 Jul 2007',
for example in EXAMPLES:

    source = string_to_int(example, Tx, human_vocab)
    source = np.array(list(map(lambda x: to_categorical(x, num_classes=len(human_
    prediction = model.predict([source, s0, c0])
    prediction = np.argmax(prediction, axis = -1)
    output = [inv_machine_vocab[int(i)] for i in prediction]

    print("source:", example)
    print("output:", ''.join(output))
```

```
source: 3 May 1979
output: 1999-05-03
source: 5 April 09
output: 2009-04-05
source: 21th of August 2016
output: 2016-08-22
source: Tue 10 Jul 2007
output: 2007-07-00
source: Saturday May 9 2018
output: 2018-05-09
source: March 3 2001
output: 2011-03-03
source: March 3rd 2001
output: 2011-02-03
source: 1 March 2001
output: 2011-03-11
```

You can also change these examples to test with your own examples. The next part will give you a better sense on what the attention mechanism is doing--i.e., what part of the input the network is paying attention to when generating a particular output character.

3 - Visualizing Attention (Optional / Ungraded)

Since the problem has a fixed output length of 10, it is also possible to carry out this task using 10 different softmax units to generate the 10 characters of the output. But one advantage of the attention model is that each part of the output (say the month) knows it needs to depend only on a small part of the input (the characters in the input giving the month). We can visualize what part of the output is looking at what part of the input.

Consider the task of translating "Saturday 9 May 2018" to "2018-05-09". If we visualize the computed $\alpha^{\langle t,t' \rangle}$ $\alpha^{\langle t,t' \rangle}$ we get this:

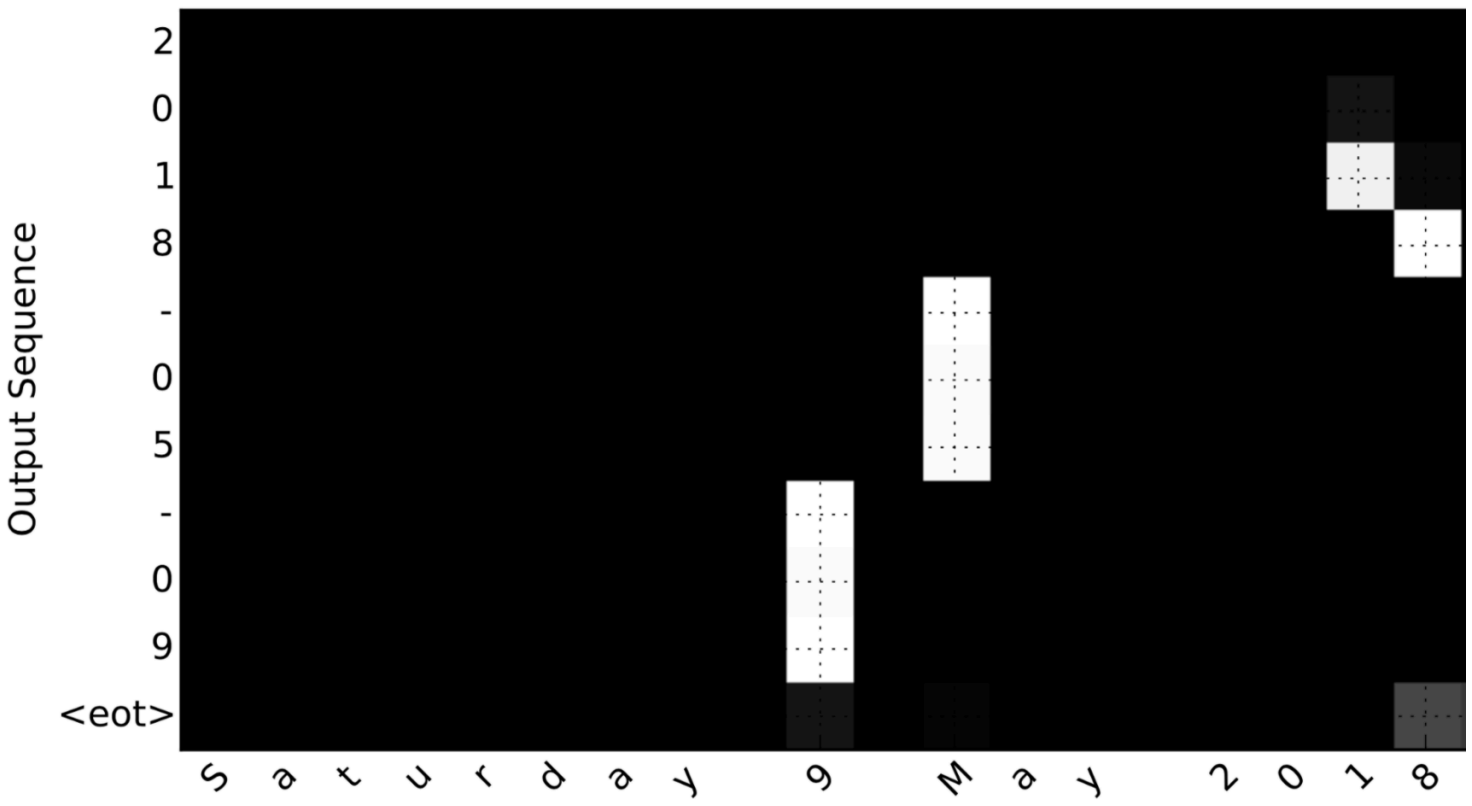

Figure 8: Full Attention Map

Notice how the output ignores the "Saturday" portion of the input. None of the output timesteps are paying much attention to that portion of the input. We see also that 9 has been translated as 09 and May has been correctly translated into 05, with the output paying attention to the parts of the input it needs to to make the translation. The year mostly requires it to pay attention to the input's "18" in order to generate "2018."

3.1 - Getting the activations from the network

Lets now visualize the attention values in your network. We'll propagate an example through the network, then visualize the values of $\alpha^{\langle t,t' \rangle}$ $\alpha^{\langle t,t' \rangle}$.

To figure out where the attention values are located, let's start by printing a summary of the model .

In [46]:

```
model.summary()
```

```
_____________________________________________________________________________
_____________________________
Layer (type)                 Output Shape              Param #     C
onnected to
=============================================================================
==============================
input_7 (InputLayer)         (None, 30, 37)            0
_____________________________________________________________________________
_____________________________
s0 (InputLayer)              (None, 64)                0
_____________________________________________________________________________
_____________________________
bidirectional_7 (Bidirectional)  (None, 30, 128)       52224       i
nput_7[0][0]
_____________________________________________________________________________
_____________________________
repeat_vector_2 (RepeatVector)   (None, 30, 64)        0           s
0[0][0]

lstm_4[21][0]
```

Navigate through the output of `model.summary()` above. You can see that the layer named `attention_weights` outputs the `alphas` of shape (m, 30, 1) before `dot_2` computes the context vector for every time step $t = 0, \ldots, T_y - 1 t = 0, \ldots, T_y - 1$. Lets get the activations from this layer.

The function `attention_map()` pulls out the attention values from your model and plots them.

In [ ]:

```
attention_map = plot_attention_map(model, human_vocab, inv_machine_vocab, "Tuesda
```

On the generated plot you can observe the values of the attention weights for each character of the predicted output. Examine this plot and check that where the network is paying attention makes sense to you.

In the date translation application, you will observe that most of the time attention helps predict the year, and hasn't much impact on predicting the day/month.

Congratulations!

You have come to the end of this assignment

Here's what you should remember from this notebook:

- Machine translation models can be used to map from one sequence to another. They are useful not just for translating human languages (like French->English) but also for tasks like date format translation.
- An attention mechanism allows a network to focus on the most relevant parts of the input when producing a specific part of the output.
- A network using an attention mechanism can translate from inputs of length T_x T_x to outputs of length T_y T_y, where T_x T_x and T_y T_y can be different.
- You can visualize attention weights $\alpha^{\langle t,t' \rangle}$ $\alpha^{\langle t,t' \rangle}$ to see what the network is paying attention to while generating each output.

Congratulations on finishing this assignment! You are now able to implement an attention model and use it to learn complex mappings from one sequence to another.

Neural Machine Translation

Welcome to your first programming assignment for this week!

You will build a Neural Machine Translation (NMT) model to translate human readable dates ("25th of June, 2009") into machine readable dates ("2009-06-25"). You will do this using an attention model, one of the most sophisticated sequence to sequence models.

This notebook was produced together with NVIDIA's Deep Learning Institute.

Let's load all the packages you will need for this assignment.

In [14]:

```
```

end3...

1 - Translating human readable dates into machine readable dates

The model you will build here could be used to translate from one language to another, such as translating from English to Hindi. However, language translation requires massive datasets and usually takes days of training on GPUs. To give you a place to experiment with these models even without using massive datasets, we will instead use a simpler "date translation" task.

The network will input a date written in a variety of possible formats (*e.g. "the 29th of August 1958", "03/30/1968", "24 JUNE 1987"*) and translate them into standardized, machine readable dates (*e.g. "1958-08-29", "1968-03-30", "1987-06-24"*). We will have the network learn to output dates in the common machine-readable format YYYY-MM-DD.

1.1 - Dataset

We will train the model on a dataset of 10000 human readable dates and their equivalent, standardized, machine readable dates. Let's run the following cells to load the dataset and print some examples.

In [15]:

```
len(dataset): 10000
human_vocab: {' ': 0, '.': 1, '/': 2, '0': 3, '1': 4, '2': 5, '3': 6
, '4': 7, '5': 8, '6': 9, '7': 10, '8': 11, '9': 12, 'a': 13, 'b': 1
4, 'c': 15, 'd': 16, 'e': 17, 'f': 18, 'g': 19, 'h': 20, 'i': 21, 'j
': 22, 'l': 23, 'm': 24, 'n': 25, 'o': 26, 'p': 27, 'r': 28, 's': 29
, 't': 30, 'u': 31, 'v': 32, 'w': 33, 'y': 34, '<unk>': 35, '<pad>':
36}
machine_vocab: {'-': 0, '0': 1, '1': 2, '2': 3, '3': 4, '4': 5, '5':
6, '6': 7, '7': 8, '8': 9, '9': 10}
```

In [4]:

Out[4]:

```
[('9 may 1998', '1998-05-09'),
 ('10.09.70', '1970-09-10'),
 ('4/28/90', '1990-04-28'),
 ('thursday january 26 1995', '1995-01-26'),
 ('monday march 7 1983', '1983-03-07'),
 ('sunday may 22 1988', '1988-05-22'),
 ('tuesday july 8 2008', '2008-07-08'),
 ('08 sep 1999', '1999-09-08'),
 ('1 jan 1981', '1981-01-01'),
 ('monday may 22 1995', '1995-05-22')]
```

You've loaded:

- `dataset`: a list of tuples of (human readable date, machine readable date)
- `human_vocab`: a python dictionary mapping all characters used in the human readable dates to an integer-valued index
- `machine_vocab`: a python dictionary mapping all characters used in machine readable dates to an integer-valued index. These indices are not necessarily consistent with `human_vocab`.
- `inv_machine_vocab`: the inverse dictionary of `machine_vocab`, mapping from indices back to characters.

Let's preprocess the data and map the raw text data into the index values. We will also use Tx=30 (which we assume is the maximum length of the human readable date; if we get a longer input, we would have to truncate it) and Ty=10 (since "YYYY-MM-DD" is 10 characters long).

In [16]:

```
X.shape: (10000, 30)
Y.shape: (10000, 10)
Xoh.shape: (10000, 30, 37)
Yoh.shape: (10000, 10, 11)
```

You now have:

- `X`: a processed version of the human readable dates in the training set, where each character is replaced by an index mapped to the character via `human_vocab`. Each date is further padded to T_x values with a special character (< pad >). `X.shape = (m, Tx)`
- `Y`: a processed version of the machine readable dates in the training set, where each character is replaced by the index it is mapped to in `machine_vocab`. You should have `Y.shape = (m, Ty)`.
- `Xoh`: one-hot version of `X`, the "1" entry's index is mapped to the character thanks to `human_vocab`. `Xoh.shape = (m, Tx, len(human_vocab))`
- `Yoh`: one-hot version of `Y`, the "1" entry's index is mapped to the character thanks to `machine_vocab`. `Yoh.shape = (m, Tx, len(machine_vocab))`. Here, `len(machine_vocab) = 11` since there are 11 characters ('-' as well as 0-9).

Lets also look at some examples of preprocessed training examples. Feel free to play with `index` in the cell below to navigate the dataset and see how source/target dates are preprocessed.

In [9]:

```
Source date: 9 may 1998
Target date: 1998-05-09

Source after preprocessing (indices): [12  0 24 13 34  0  4 12 12 11
36 36 36 36 36 36 36 36 36 36 36 36 36 36
 36 36 36 36 36]
Target after preprocessing (indices): [ 2 10 10  9  0  1  6  0  1 10
]

Source after preprocessing (one-hot): [[ 0.  0.  0. ...,  0.  0.  0.
]
 [ 1.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 0.  0.  0. ...,  0.  0.  1.]
 [ 0.  0.  0. ...,  0.  0.  1.]
 [ 0.  0.  0. ...,  0.  0.  1.]]
Target after preprocessing (one-hot): [[ 0.  0.  1.  0.  0.  0.  0.
0.  0.  0.  0.]
```

2 - Neural machine translation with attention

If you had to translate a book's paragraph from French to English, you would not read the whole paragraph, then close the book and translate. Even during the translation process, you would read/re-read and focus on the parts of the French paragraph corresponding to the parts of the English you are writing down.

The attention mechanism tells a Neural Machine Translation model where it should pay attention to at any step.

2.1 - Attention mechanism

In this part, you will implement the attention mechanism presented in the lecture videos. Here is a figure to remind you how the model works. The diagram on the left shows the attention model. The diagram on the right shows what one "Attention" step does to calculate the attention variables $\alpha^{\langle t, t' \rangle}$, which are used to compute the context variable $context^{\langle t \rangle}$ for each timestep in the output ($t=1, \ldots, T_y$).

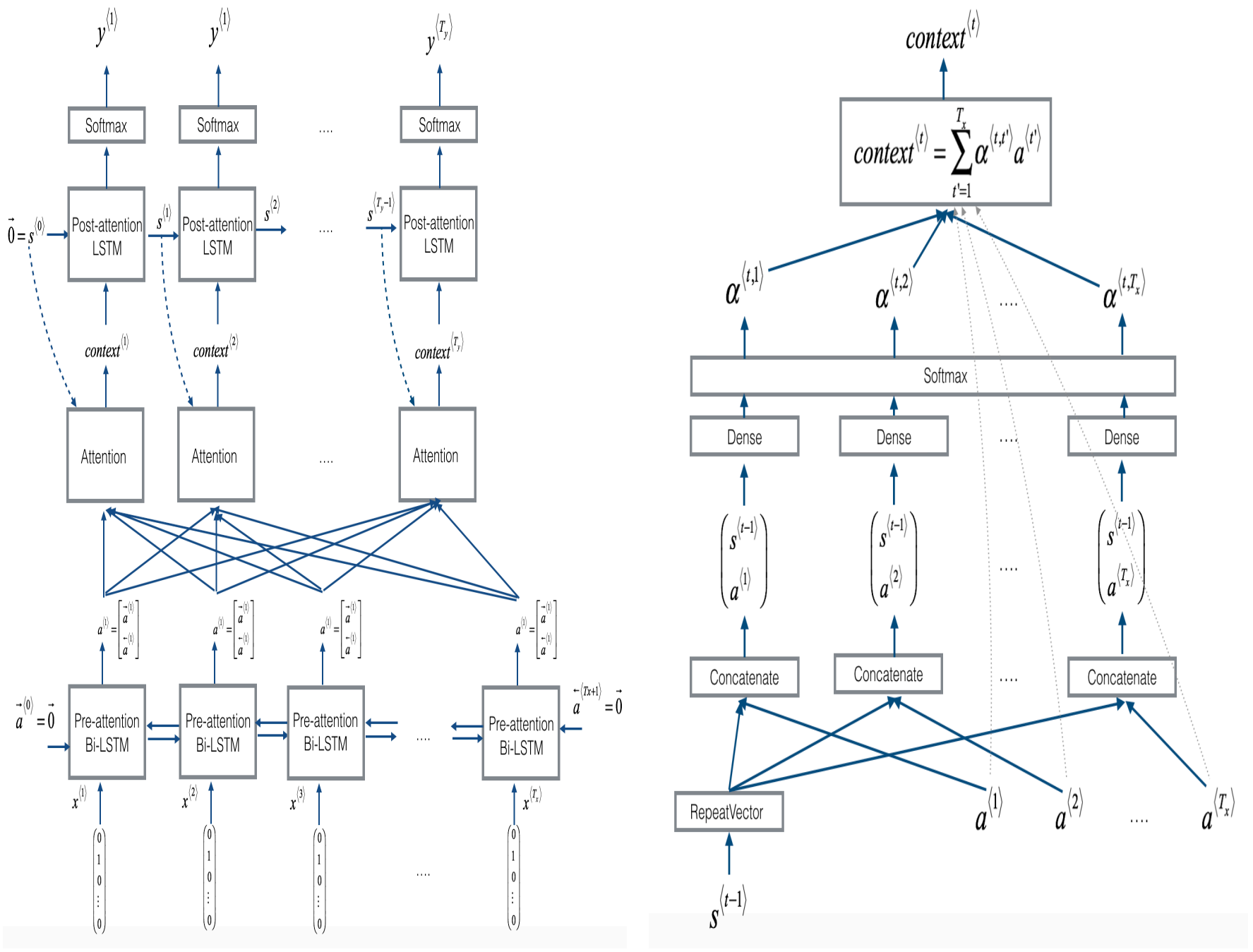

Figure 1: Neural machine translation with attention

Here are some properties of the model that you may notice:

- There are two separate LSTMs in this model (see diagram on the left). Because the one at the bottom of the picture is a Bi-directional LSTM and comes *before* the attention mechanism, we will call it *pre-attention* Bi-LSTM. The LSTM at the top of the diagram comes *after* the attention mechanism, so we will call it the *post-attention* LSTM. The pre-attention Bi-LSTM goes through T_x time steps; the post-attention LSTM goes through T_y time steps.
- The post-attention LSTM passes $s^{\langle t \rangle}, c^{\langle t \rangle}$ from one time step to the next. In the lecture videos, we were using only a basic RNN for the post-activation sequence model, so the state captured by the RNN output activations $s^{\langle t\rangle}$. But since we are using an LSTM here, the LSTM has both the output activation $s^{\langle t\rangle}$ and the hidden cell state $c^{\langle t\rangle}$. However, unlike previous text generation examples (such as Dinosaurus in week 1), in this model the post-activation LSTM at time t does will not take the specific generated $y^{\langle t-1 \rangle}$ as input; it only takes $s^{\langle t\rangle}$ and $c^{\langle t\rangle}$ as input. We have designed the model this way, because (unlike language generation where adjacent characters are highly correlated) there isn't as strong a dependency between the previous character and the next character in a YYYY-MM-DD date.
- We use $a^{\langle t \rangle} = [\overrightarrow{a}^{\langle t \rangle}; \overleftarrow{a}^{\langle t \rangle}]$ to represent the concatenation of the activations of both the forward-direction and backward-directions of the pre-attention Bi-LSTM.
- The diagram on the right uses a `RepeatVector` node to copy $s^{\langle t-1 \rangle}$'s value T_x times, and then `Concatenation` to concatenate $s^{\langle t-1 \rangle}$ and $a^{\langle t \rangle}$ to compute $e^{\langle t, t'}$, which is then passed through a softmax to compute $\alpha^{\langle t, t' \rangle}$. We'll explain how to use `RepeatVector` and `Concatenation` in Keras below.

Lets implement this model. You will start by implementing two functions: `one_step_attention()` and `model()`.

1) `one_step_attention()`: At step t, given all the hidden states of the Bi-LSTM ($[a^{<1>},a^{<2>}, ..., a^{<T_x>}]$) and the previous hidden state of the second LSTM ($s^{<t-1>}$), `one_step_attention()` will compute the attention weights ($[\alpha^{<t,1>},\alpha^{<t,2>}, ..., \alpha^{<t,T_x>}]$) and output the context vector (see Figure 1 (right) for details): $$context^{<t>} = \sum_{t' = 0}^{T_x} \alpha^{<t,t'>}a^{<t'>}\tag{1}$$

Note that we are denoting the attention in this notebook $context^{\langle t \rangle}$. In the lecture videos, the context was denoted $c^{\langle t \rangle}$, but here we are calling it $context^{\langle t \rangle}$ to avoid confusion with the (post-attention) LSTM's internal memory cell variable, which is sometimes also denoted $c^{\langle t \rangle}$.

2) `model()`: Implements the entire model. It first runs the input through a Bi-LSTM to get back $[a^{<1>},a^{<2>}, ..., a^{<T_x>}]$. Then, it calls `one_step_attention()` T_y times (`for` loop). At each iteration of this loop, it gives the computed context vector $c^{<t>}$ to the second LSTM, and runs the output of the LSTM through a dense layer with softmax activation to generate a prediction $\hat{y}^{<t>}$.

Exercise: Implement `one_step_attention()`. The function `model()` will call the layers in `one_step_attention()` T_y using a for-loop, and it is important that all T_y copies have the same weights. I.e., it should not re-initiaiize the weights every time. In other words, all T_y steps should have shared weights. Here's how you can implement layers with shareable weights in Keras:

1. Define the layer objects (as global variables for examples).
2. Call these objects when propagating the input.

We have defined the layers you need as global variables. Please run the following cells to create them. Please check the Keras documentation to make sure you understand what these layers are: RepeatVector() (https://keras.io/layers/core/#repeatvector), Concatenate() (https://keras.io/layers/merge/#concatenate), Dense() (https://keras.io/layers/core/#dense), Activation() (https://keras.io/layers/core/#activation), Dot() (https://keras.io/layers/merge/#dot).

In [17]:

```
end2...
```

Now you can use these layers to implement `one_step_attention()`. In order to propagate a Keras tensor object X through one of these layers, use `layer(X)` (or `layer([X,Y])` if it requires multiple inputs.), e.g. `densor(X)` will propagate X through the `Dense(1)` layer defined above.

In [18]:

You will be able to check the expected output of `one_step_attention()` after you've coded the `model()` function.

Exercise: Implement `model()` as explained in figure 2 and the text above. Again, we have defined global layers that will share weights to be used in `model()`.

In [19]:

```
end3
```

Now you can use these layers T_y times in a `for` loop to generate the outputs, and their parameters will not be reinitialized. You will have to carry out the following steps:

1. Propagate the input into a Bidirectional (https://keras.io/layers/wrappers/#bidirectional) LSTM (https://keras.io/layers/recurrent/#lstm)
2. Iterate for $t = 0, \dots, T_y-1$:
    A. Call `one_step_attention()` on $[\alpha^{<t,1>},\alpha^{<t,2>}, ..., \alpha^{<t,T_x>}]$ and $s^{<t-1>}$ to get the context vector $context^{<t>}$.
    B. Give $context^{<t>}$ to the post-attention LSTM cell. Remember pass in the previous hidden-state $s^{\langle t-1\rangle}$ and cell-states $c^{\langle t-1\rangle}$ of this LSTM using `initial_state= [previous hidden state, previous cell state]`. Get back the new hidden state $s^{<t>}$ and the new cell state $c^{<t>}$.
    C. Apply a softmax layer to $s^{<t>}$, get the output.
    D. Save the output by adding it to the list of outputs.
3. Create your Keras model instance, it should have three inputs ("inputs", $s^{<0>}$ and $c^{<0>}$) and output the list of "outputs".

In [28]:

```
end...
```

Run the following cell to create your model.

In [29]:

Let's get a summary of the model to check if it matches the expected output.

In [30]:

```
__________________________________________________________________________
_____________________________________
Layer (type)                 Output Shape              Param #     C
onnected to
==========================================================================
==============================
input_7 (InputLayer)         (None, 30, 37)            0
__________________________________________________________________________
_____________________________________
s0 (InputLayer)              (None, 64)                0
__________________________________________________________________________
_____________________________________
bidirectional_7 (Bidirectional)  (None, 30, 128)       52224       i
nput_7[0][0]
__________________________________________________________________________
_____________________________________
repeat_vector_2 (RepeatVector)   (None, 30, 64)        0           s
0[0][0]

lstm_4[21][0]
```

Expected Output:

Here is the summary you should see

Total params:	52,960
Trainable params:	52,960
Non-trainable params:	0
bidirectional_1's output shape	(None, 30, 64)
repeat_vector_1's output shape	(None, 30, 64)
concatenate_1's output shape	(None, 30, 128)
attention_weights's output shape	(None, 30, 1)
dot_1's output shape	(None, 1, 64)
dense_3's output shape	(None, 11)

As usual, after creating your model in Keras, you need to compile it and define what loss, optimizer and metrics your are want to use. Compile your model using `categorical_crossentropy` loss, a custom Adam (https://keras.io/optimizers/#adam) optimizer (https://keras.io/optimizers/#usage-of-optimizers) (`learning rate = 0.005`, $\beta_1 = 0.9$, $\beta_2 = 0.999$, `decay = 0.01`) and `['accuracy']` metrics:

In [44]:

```
```

The last step is to define all your inputs and outputs to fit the model:

- You already have X of shape $(m = 10000, T_x = 30)$ containing the training examples.
- You need to create `s0` and `c0` to initialize your `post_activation_LSTM_cell` with 0s.
- Given the `model()` you coded, you need the "outputs" to be a list of 11 elements of shape (m, T_y). So that: `outputs[i][0], ..., outputs[i][Ty]` represent the true labels (characters) corresponding to the i^{th} training example (`X[i]`). More generally, `outputs[i][j]` is the true label of the j^{th} character in the i^{th} training example.

In [40]:

```
```

Let's now fit the model and run it for one epoch.

In [43]:

```
```

```
Epoch 1/10
10000/10000 [==============================] - 46s - loss: 8.1446 -
dense_6_loss_1: 0.1405 - dense_6_loss_2: 0.1162 - dense_6_loss_3: 0.
8303 - dense_6_loss_4: 1.9593 - dense_6_loss_5: 0.0382 - dense_6_los
s_6: 0.2686 - dense_6_loss_7: 1.7330 - dense_6_loss_8: 0.0298 - dens
e_6_loss_9: 0.9677 - dense_6_loss_10: 2.0610 - dense_6_acc_1: 0.9719
- dense_6_acc_2: 0.9723 - dense_6_acc_3: 0.6902 - dense_6_acc_4: 0.3
128 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9352 - dense_6_acc_7:
0.3972 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.5992 - dense_6_acc
_10: 0.2484
Epoch 2/10
10000/10000 [==============================] - 44s - loss: 7.2231 -
dense_6_loss_1: 0.1071 - dense_6_loss_2: 0.0917 - dense_6_loss_3: 0.
6896 - dense_6_loss_4: 1.7429 - dense_6_loss_5: 0.0247 - dense_6_los
s_6: 0.1993 - dense_6_loss_7: 1.5275 - dense_6_loss_8: 0.0208 - dens
e_6_loss_9: 0.8773 - dense_6_loss_10: 1.9421 - dense_6_acc_1: 0.9762
- dense_6_acc_2: 0.9760 - dense_6_acc_3: 0.7442 - dense_6_acc_4: 0.4
111 - dense_6_acc_5: 1.0000 - dense_6_acc_6: 0.9529 - dense_6_acc_7:
0.4637 - dense_6_acc_8: 1.0000 - dense_6_acc_9: 0.6305 - dense_6_acc
```

While training you can see the loss as well as the accuracy on each of the 10 positions of the output. The table below gives you an example of what the accuracies could be if the batch had 2 examples:

True labels 1	1	9	9	5	-	1	2	-	0	4
Predictions 1	1	9	9	5	-	1	0	-	0	5
True labels 1	1	9	6	8	-	0	1	-	0	4
Predictions 2	1	9	7	8	-	0	3	-	0	4
Index	1	2	3	4	5	6	7	8	9	10
Accuracy	1.0	1.0	0.5	1.0	1.0	1.0	0.0	1.0	1.0	0.5

Thus, `dense_2_acc_8: 0.89` means that you are predicting the 7th character of the output correctly 89% of the time in the current batch of data.

We have run this model for longer, and saved the weights. Run the next cell to load our weights. (By training a model for several minutes, you should be able to obtain a model of similar accuracy, but loading our model will save you time.)

In [38]:

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent cal
l last)
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/c
ommon_shapes.py in _call_cpp_shape_fn_impl(op, input_tensors_needed,
input_tensors_as_shapes_needed, debug_python_shape_fn, require_shape
_fn)
    670           graph_def_version, node_def_str, input_shapes,
input_tensors,
--> 671           input_tensors_as_shapes, status)
    672   except errors.InvalidArgumentError as err:

/opt/conda/lib/python3.6/contextlib.py in __exit__(self, type, value
, traceback)
     88             try:
---> 89                 next(self.gen)
     90             except StopIteration:
```

You can now see the results on new examples.

In [45]:

```
source: 3 May 1979
output: 1999-05-03
source: 5 April 09
output: 2009-04-05
source: 21th of August 2016
output: 2016-08-22
source: Tue 10 Jul 2007
output: 2007-07-00
source: Saturday May 9 2018
output: 2018-05-09
source: March 3 2001
output: 2011-03-03
source: March 3rd 2001
output: 2011-02-03
source: 1 March 2001
output: 2011-03-11
```

You can also change these examples to test with your own examples. The next part will give you a better sense on what the attention mechanism is doing--i.e., what part of the input the network is paying attention to when generating a particular output character.

3 - Visualizing Attention (Optional / Ungraded)

Since the problem has a fixed output length of 10, it is also possible to carry out this task using 10 different softmax units to generate the 10 characters of the output. But one advantage of the attention model is that each part of the output (say the month) knows it needs to depend only on a small part of the input (the characters in the input giving the month). We can visualize what part of the output is looking at what part of the input.

Consider the task of translating "Saturday 9 May 2018" to "2018-05-09". If we visualize the computed $\alpha^{\langle t, t' \rangle}$ we get this:

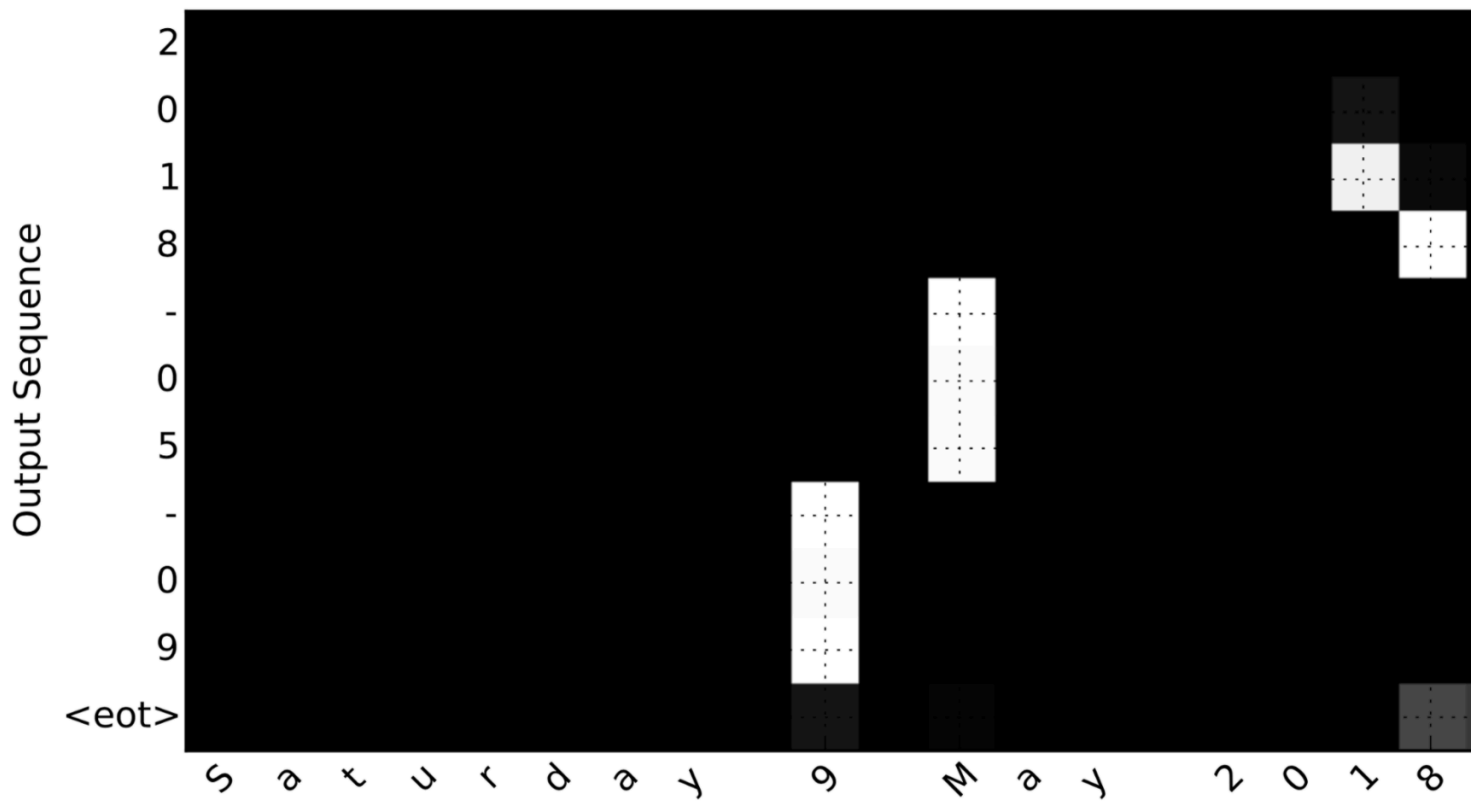

Figure 8: Full Attention Map

Notice how the output ignores the "Saturday" portion of the input. None of the output timesteps are paying much attention to that portion of the input. We see also that 9 has been translated as 09 and May has been correctly translated into 05, with the output paying attention to the parts of the input it needs to to make the translation. The year mostly requires it to pay attention to the input's "18" in order to generate "2018."

3.1 - Getting the activations from the network

Lets now visualize the attention values in your network. We'll propagate an example through the network, then visualize the values of $\alpha^{\langle t, t' \rangle}$.

To figure out where the attention values are located, let's start by printing a summary of the model .

In [46]:

```
_________________________________________________________________
_____________________________________
Layer (type)                 Output Shape              Param #     C
onnected to
==================================================================
==================================
input_7 (InputLayer)         (None, 30, 37)            0
_________________________________________________________________
_____________________________________
s0 (InputLayer)              (None, 64)                0
_________________________________________________________________
_____________________________________
bidirectional_7 (Bidirectional)  (None, 30, 128)       52224       i
nput_7[0][0]
_________________________________________________________________
_____________________________________
repeat_vector_2 (RepeatVector)   (None, 30, 64)        0           s
0[0][0]

lstm_4[21][0]
```

Navigate through the output of `model.summary()` above. You can see that the layer named `attention_weights` outputs the `alphas` of shape (m, 30, 1) before `dot_2` computes the context vector for every time step $t = 0, \ldots, T_y-1$. Lets get the activations from this layer.

The function `attention_map()` pulls out the attention values from your model and plots them.

In [ ]:

On the generated plot you can observe the values of the attention weights for each character of the predicted output. Examine this plot and check that where the network is paying attention makes sense to you.

In the date translation application, you will observe that most of the time attention helps predict the year, and hasn't much impact on predicting the day/month.

Congratulations!

You have come to the end of this assignment

Here's what you should remember from this notebook:

- Machine translation models can be used to map from one sequence to another. They are useful not just for translating human languages (like French->English) but also for tasks like date format translation.
- An attention mechanism allows a network to focus on the most relevant parts of the input when producing a specific part of the output.
- A network using an attention mechanism can translate from inputs of length T_x to outputs of length T_y, where T_x and T_y can be different.
- You can visualize attention weights $\alpha^{\langle t,t' \rangle}$ to see what the network is paying attention to while generating each output.

Congratulations on finishing this assignment! You are now able to implement an attention model and use it to learn complex mappings from one sequence to another.