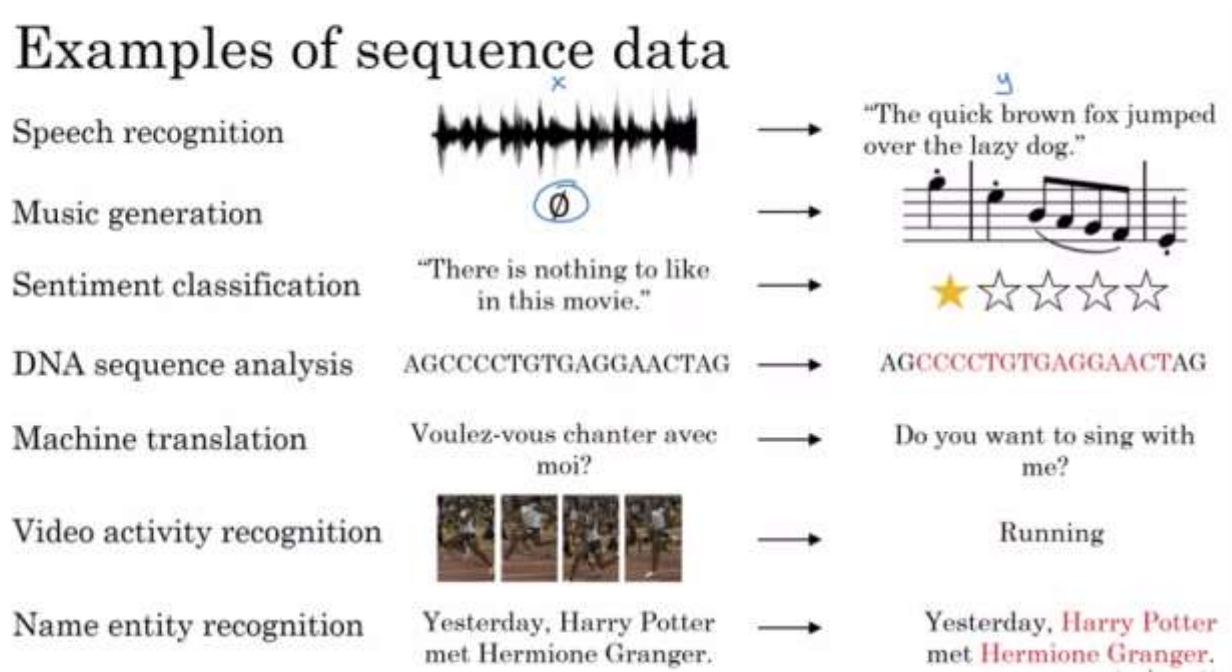
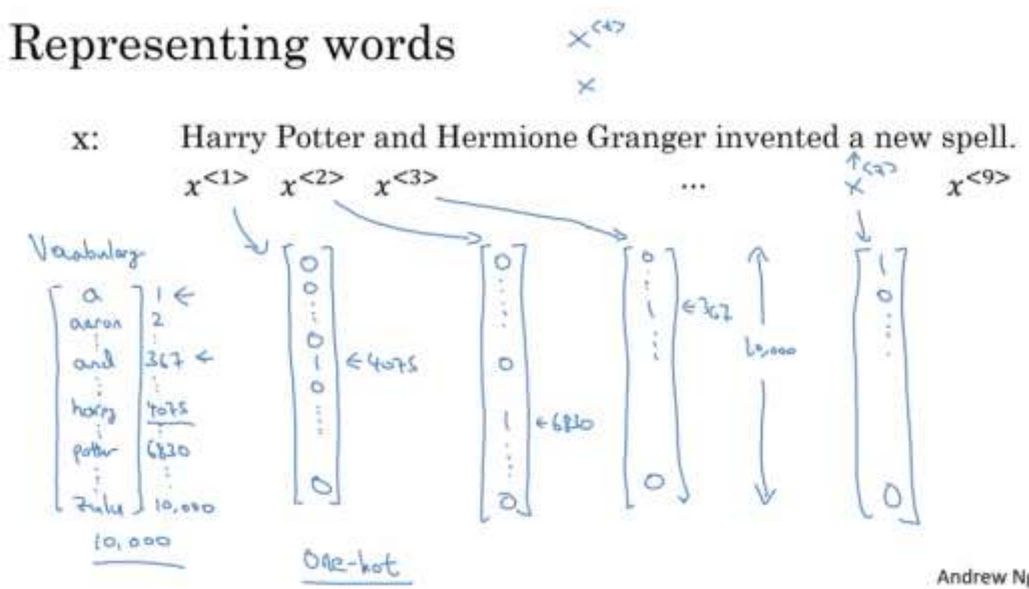
Sequence Models

Week 1



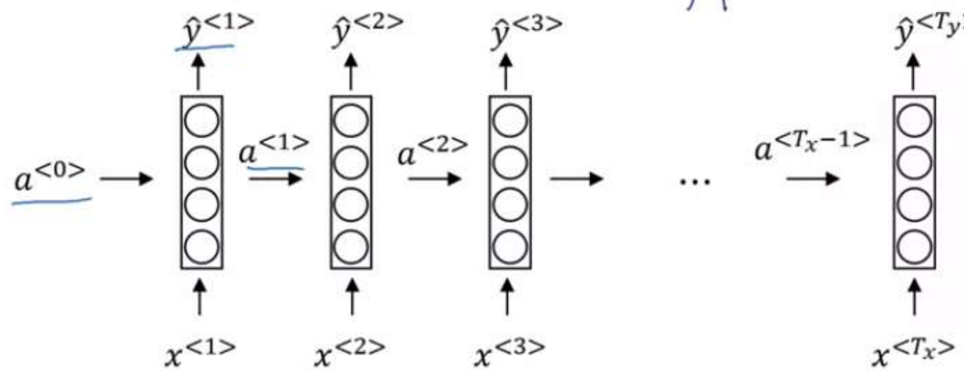


Problems with regular neural networks for sequence models:

- Inputs, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of text.

Forward Propagation

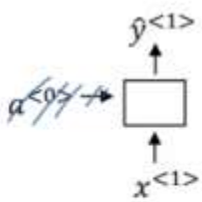
$a \leftarrow W_{ax} x^{<1>}$



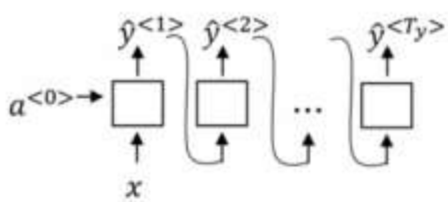
$a^{<0>} = \vec{0}$.

$a^{<1>} = g(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a) \leftarrow$ tanh / ReLU

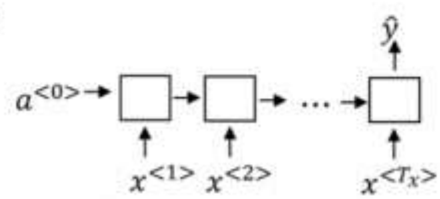
$\hat{y}^{<1>} = g(W_{ya} a^{<1>} + b_y) \leftarrow$ sigmoid



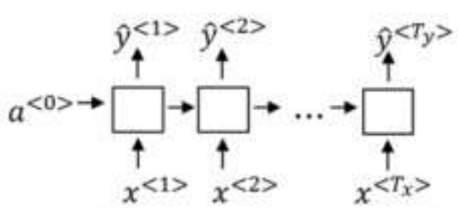
One to one



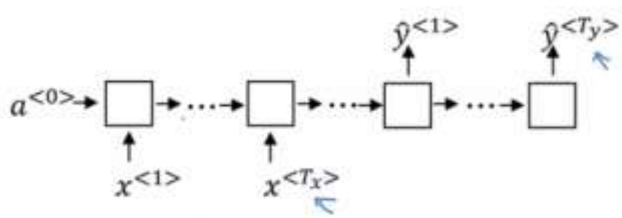
One to many



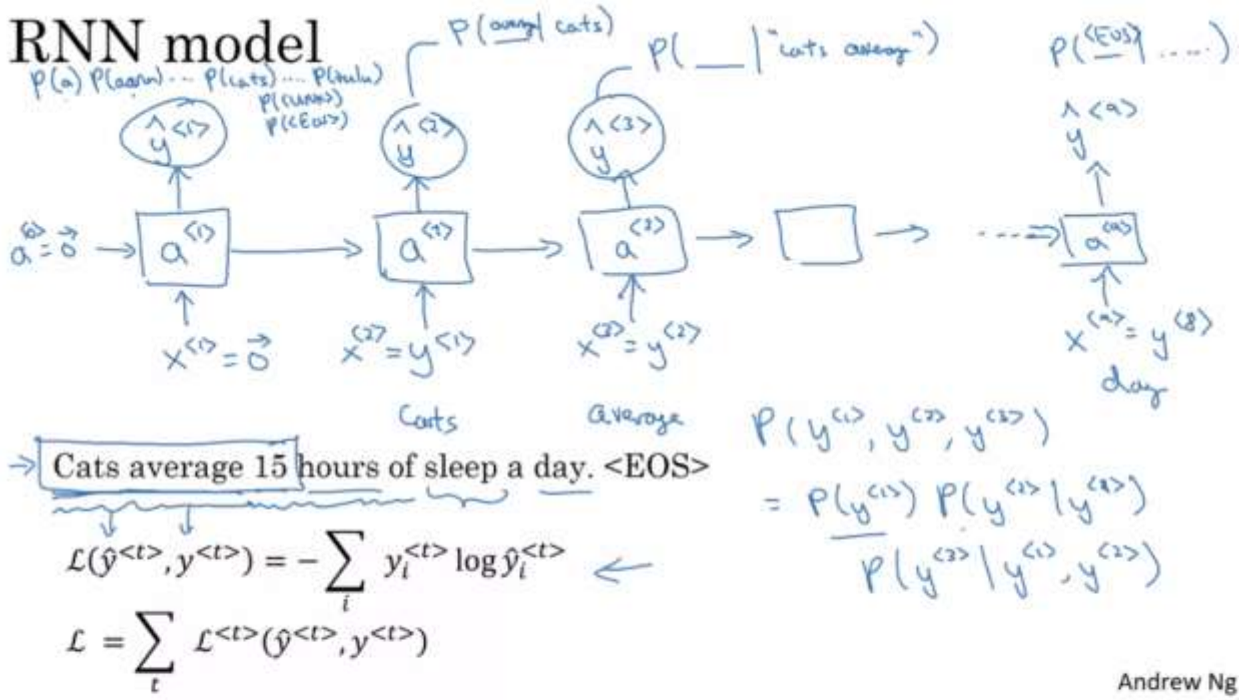
Many to one

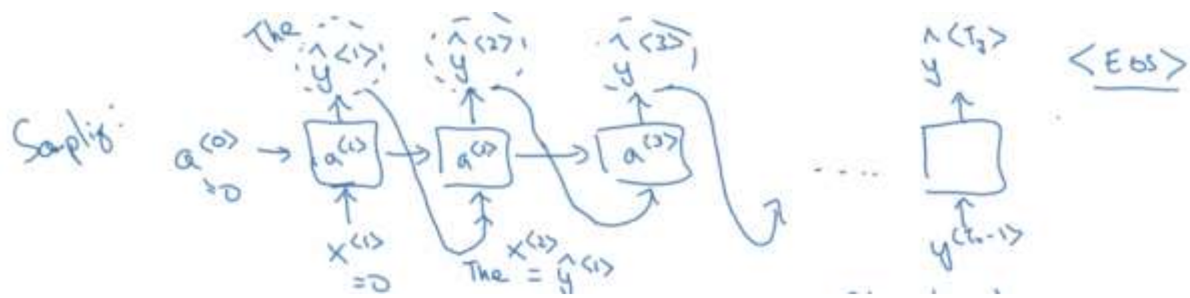


Many to many $T_x = T_y$

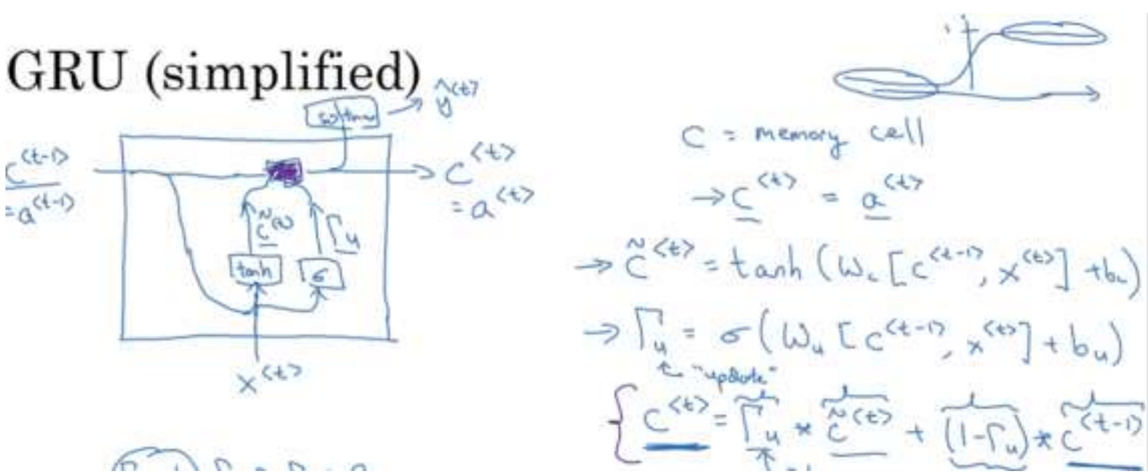


Many to many





For sampling, we may use np.random.choice.



Full GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\,\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[\,c^{<t-1>}, x^{<t>}] + b_u)$$

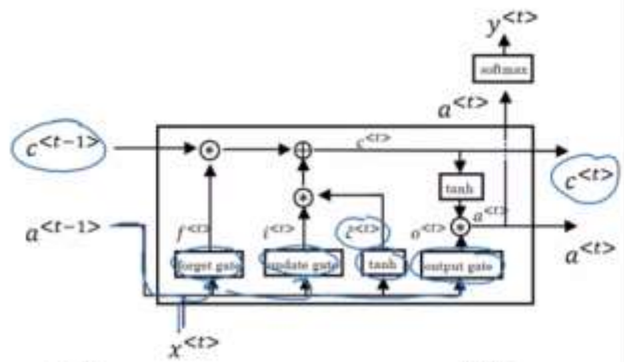
$$\Gamma_r = \sigma(W_r[\,c^{<t-1>}, x^{<t>}] + b_r)$$

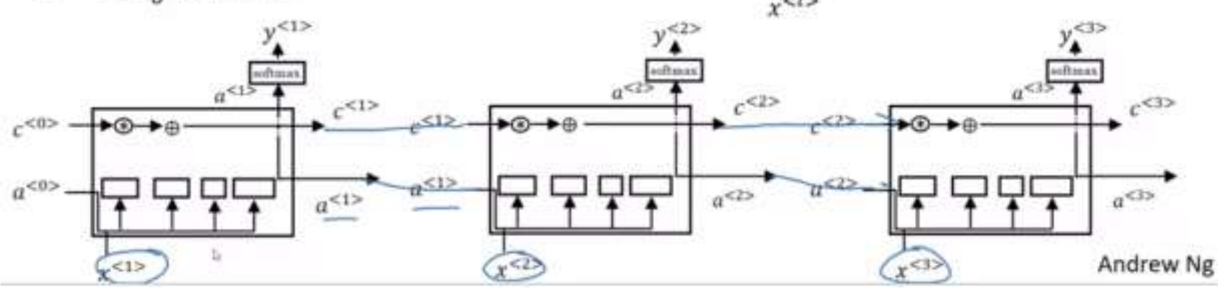
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) + c^{<t-1>}$$

The last line should use an element-wise multiplication "*" instead of a plus sign "+".

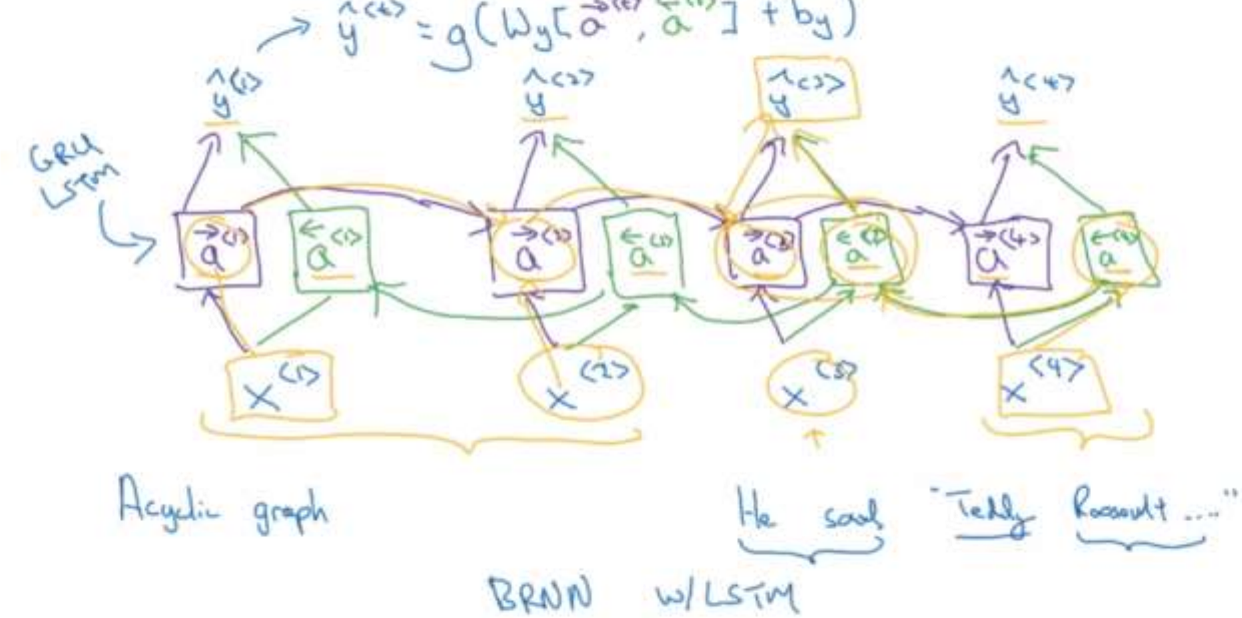
LSTM in pictures

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[\,a^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_f = \sigma(W_f[\,a^{<t-1>}, x^{<t>}] + b_f)$$
$$\Gamma_o = \sigma(W_o[\,a^{<t-1>}, x^{<t>}] + b_o)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$

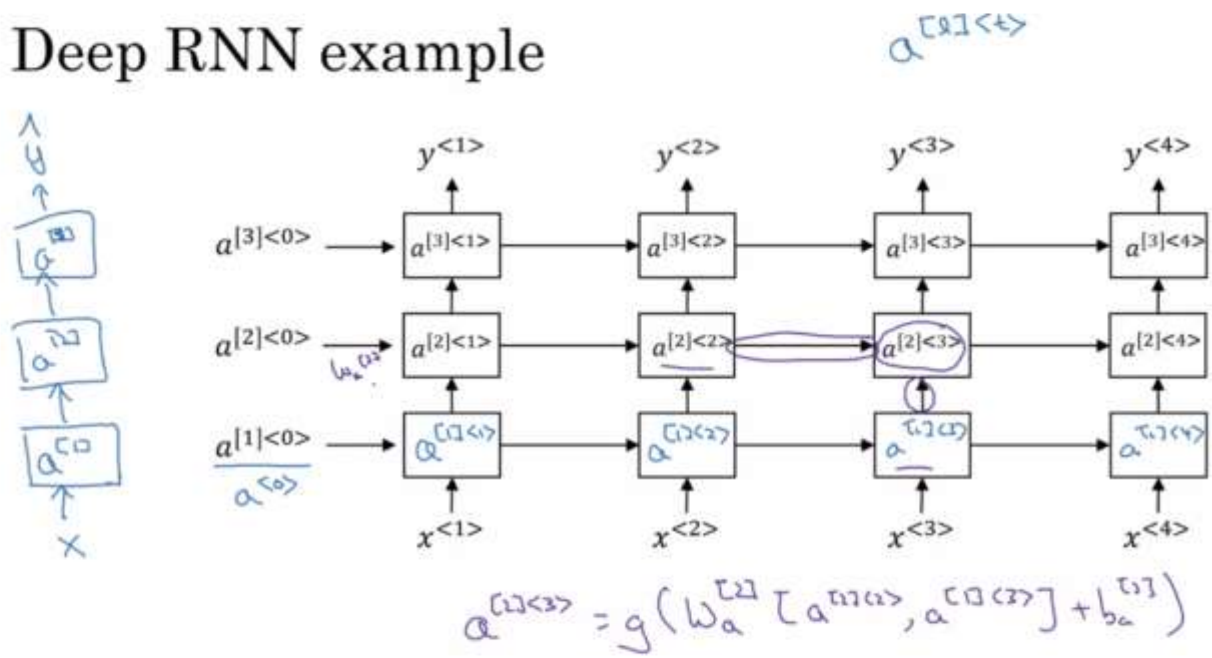




Bidirectional RNN (BRNN)



Deep RNN example



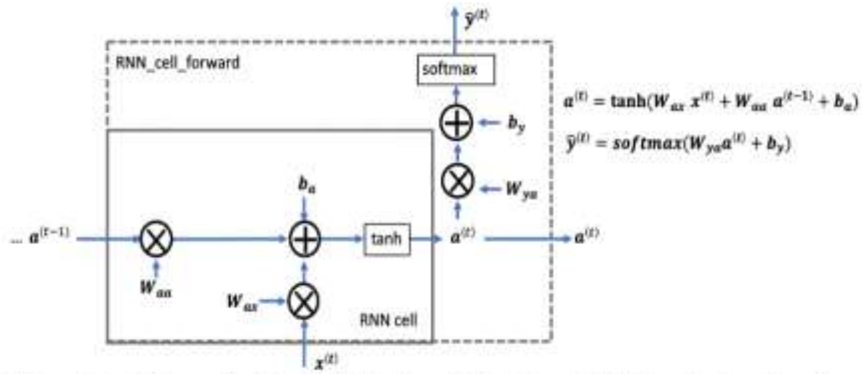


Figure 2: Basic RNN cell. Takes as input $x^{\langle t \rangle}$ (current input) and $a^{\langle t-1 \rangle}$ (previous hidden state containing information from the past), and outputs $a^{\langle t \rangle}$ which is given to the next RNN cell and also used to predict $\hat{y}^{\langle t \rangle}$

Before updating the parameters, you will perform gradient clipping to make sure that your gradients are not "exploding."

```
np.random.seed(0)
probs = np.array([0.1, 0.0, 0.7, 0.2])
idx = np.random.choice([0, 1, 2, 3] p = probs)
```

X = Input(shape=(Tx, n_values)) # X has 3 dimensions and not 2: (m, Tx, n_values)

Most computational music algorithms use some post-processing because it is difficult to generate music that sounds good without such post-processing. The post-processing does things such as clean up the generated audio by making sure the same sound is not repeated too many times, that two successive notes are not too far from each other in pitch, and so on. One could argue that a lot of these post-processing steps are hacks; also, a lot of the music generation literature has also focused on hand-crafting post-processors, and a lot of the output quality depends on the quality of the post-processing and not just the quality of the RNN. But this post-processing does make a huge difference, so let's use it in our implementation as well.