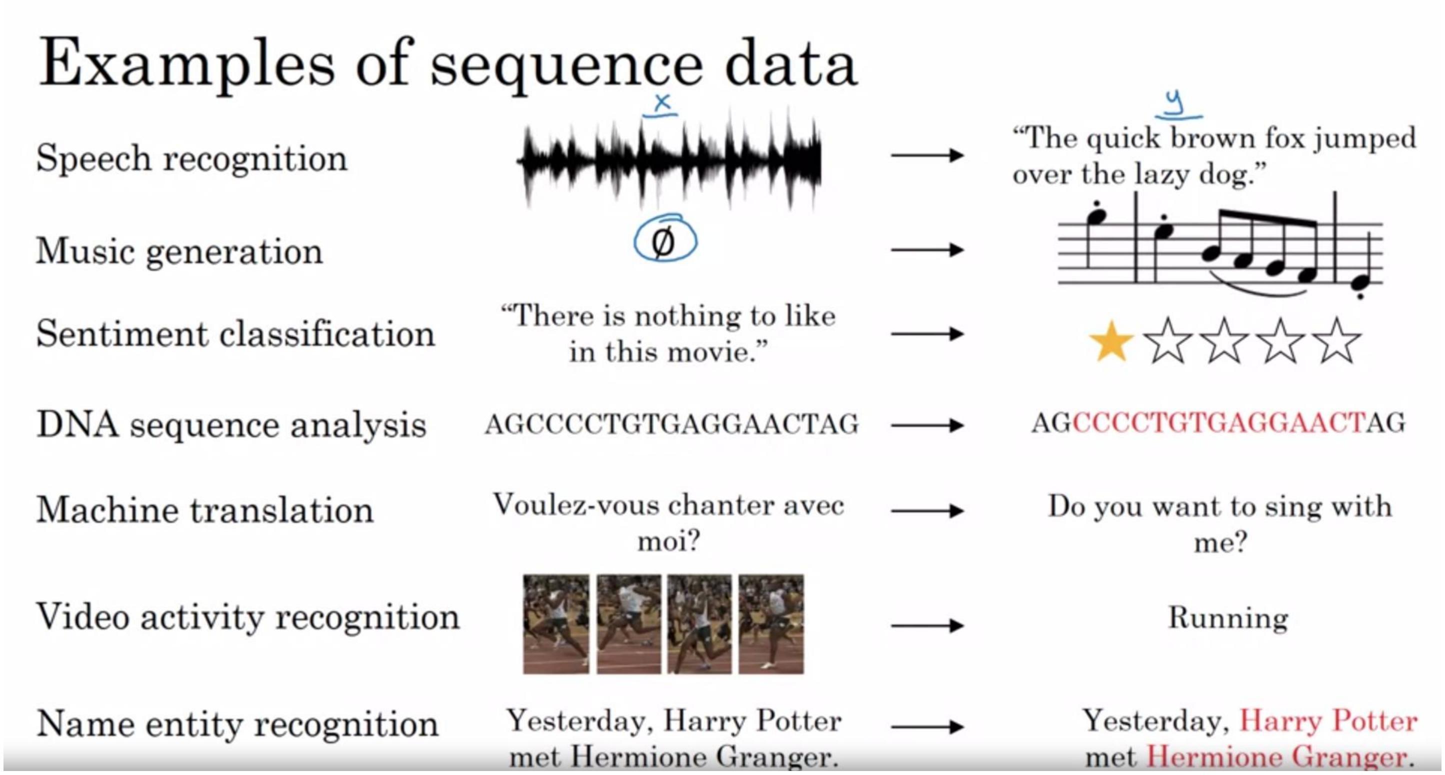
Course 1 Week1

Sequence Models:

Sequence models are one of the most exciting areas in deep learning which have examples like recurrent neural networks that has made fields like speech recognition, natural language processing (NLP) and other areas develop in the recent time.

Examples:



Examples of sequence data

Speech recognition		"The quick brown fox jumped over the lazy dog."
Music generation	∅	
Sentiment classification	"There is nothing to like in this movie."	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi?	Do you want to sing with me?
Video activity recognition		Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	Yesterday, Harry Potter met Hermione Granger.

Notes:

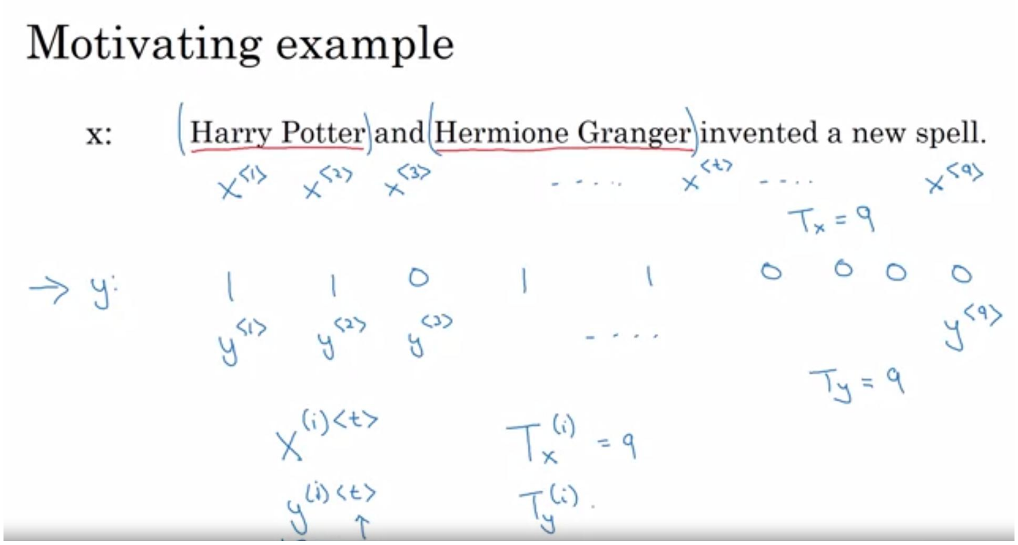
1- Input x and output y can be both sequence data or just one of them is sequence data.

2- All of these problems mentioned above can be addressed as supervised learning as having labeled data set X and Y as a training set.

3- Sometimes X and Y are having different length like Speech recognition and some have the same length like DNA and Name entity recognition problems.

Notations:

Assuming having input sentence of 9 words and output sequence of same length indicating either names or not (1 or 0).



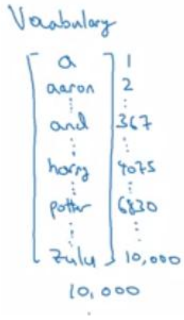
Motivating example

x: Harry Potter and Hermione Granger invented a new spell.

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ .... $x^{<t>}$ .... $x^{<9>}$
$T_x = 9$
→ y: 1 1 0 1 1 0 0 0 0
$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ .... $y^{<9>}$
$T_y = 9$
$X^{(i)<t>}$
$T_x^{(i)} = 9$
$y^{(i)<t>}$
$T_y^{(i)}$

$X^{<i>}$ is the ith element in the input sequence.
$Y^{<i>}$ is the ith element in the output sequence.
T_x is the length of the input sequence.
T_y is the length of the output sequence.
If you use (i) instead of <i> then this will represent the ith example in the training set.
Hence, $X^{(i)<t>}$ is the tth element in the ith example in the training set.

Representation of words in the NLP field is represented by a one hot encoded vector having one in the place that corresponds this word to a predefined dictionary and with the same length of this dictionary.



Vocabulary

a	1
aaron	2
and	367
harry	4075
potter	6830
zulu	10,000

10,000

If an unknown word is encountered in the sentence we use a new token called UNK for example to represent an unknown word in a sentence.

Why not use standard neural networks?
1- Inputs and outputs can be of different length for each example.
2- It doesn’t shared features learned at different positions of text.
3- By using the idea of vocabulary, the weights of each layer will be enormous in length.

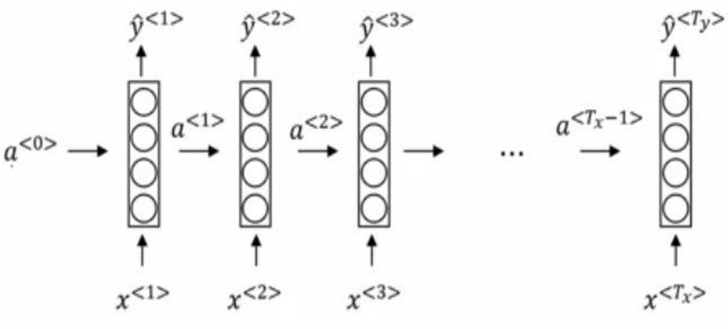
Recurrent Neural Network (RNN):

The activations from the previous layers are fed to the next layers in the recurrent neural networks to have that sequence of feeding throughout the network as shown in the figure.

Note: The first input is fed up by vector of zeros as an initial feeding activation $a^{<0>}$.

The weights for X are annotated w_{ax} and the weights for Y are annotated w_{ya} while the one for the activations is called w_{aa}.

Note: Each layer only uses the previous layers information in producing the output but doesn't take into consideration the next one in this shown model that is known as unidirectional RNN.



Forward propagation:

$a^{<t>} = g\,(w_{aa}\, a^{<t-1>} + w_{ax}\, x^{<t-1>} + b_a)$ ==> activation function is mostly tanh (sometimes ReLu).

$y^{<t>} = g\,(w_{ya}\, a^{<t>} + b_y)$ ==> activation function here is mainly sigmoid.

Simplifying the equations above would make them as follows:

$a^{<t>} = g\,(w_a\, [a^{<t-1>}, x^{<t-1>]}] + b_a)$ where:

$W_a = [w_{aa} \mid w_{ax}]$ and $[a^{<t-1>}, x^{<t-1>]}] = [\begin{matrix} a^{<t-1>} \\ x^{<t>} \end{matrix}]$

Backpropagation through time:

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) =- y^{(t)} log(\hat{y}^{<t>} - (1 - y^{<t>}) log(1 - \hat{y}^{<t>})$$

$$L(\hat{y}, y) = \sum_{t=1}^{t=T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Architectures:

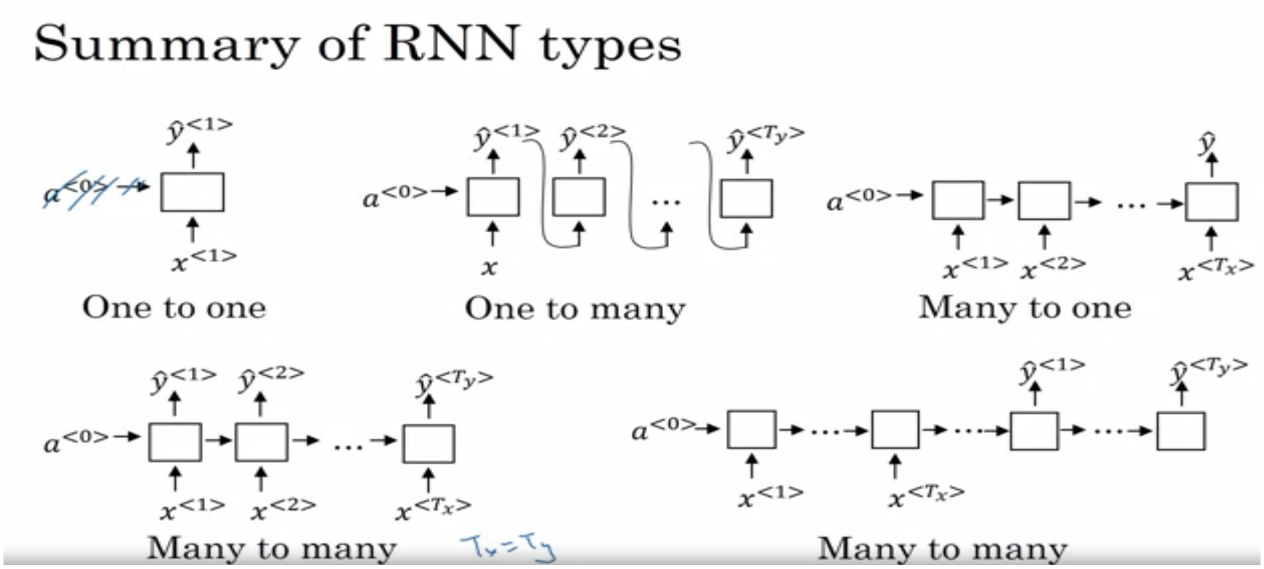
1- Many-to-Many: having many inputs and many outputs as name entity problem.

2- Many-to-one: having many inputs and outputs only one output as sentiment classification.

3- One-to-Many: having only one input and generate sequence of output as music generation.

Note:

Many-to-Many architecture could have the same or different lengths of inputs and outputs.


Summary of RNN types
$\hat{y}^{<1>}$
$a^{<0>}$
$x^{<1>}$
One to one
$a^{<0>}$
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$
x
One to many
$a^{<0>}$
$\hat{y}$
$x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Many to one
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$
$a^{<0>}$
$x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Many to many
$T_x = T_y$
$a^{<0>}$
$\hat{y}^{<1>}$ $\hat{y}^{<T_y>}$
$x^{<1>}$ $x^{<T_x>}$
Many to many

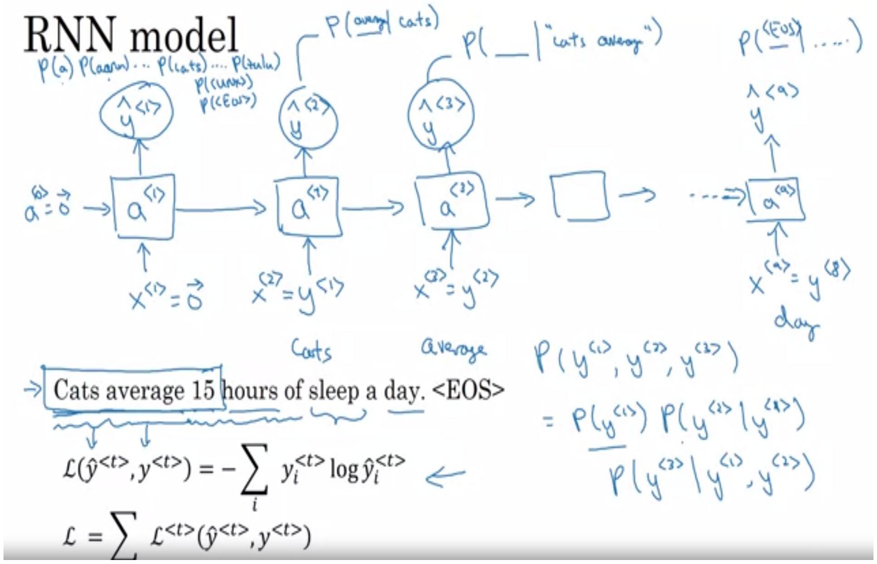
Language Modeling:

By saying we are building a speech recognition system, every valid sentence that can be constructed is given a probability and the highest probability is the one that takes the lead in this model. So language model just gives the probability of any sentence in a particular sentence. Language models is also used in machine translation systems.

How to build an language model?

Using a training set that has a large corpus of english text for example and then tokenize it so that you can form your vocabulary and then map each word to its corresponding place in the vocabulary vector. Don't forget also to add the End Of Sentence (EOS) token to this vocabulary.

For example: ==>


RNN model

Cats average 15 hours of sleep a day. <EOS>

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

$$\mathcal{L} = \sum \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Note: You can build a character-level language model which by default makes the input sequence as characters instead of words while having the vocabulary as alphabets.

Vanishing Gradients of RNNs:

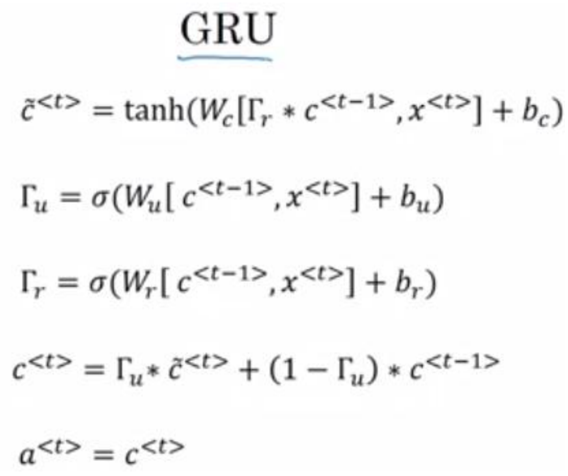
By having examples of these 2 sentences:
- The cat, which already ate ...., was full.
- The cats, which already ate ...., were full.

We can notice here that words was/were are dependant on the words cat/cats which is a very long-term dependencies as the words cat/cats affect later on another word at the end of the sentence. What will happen here is that during backpropagation it will be so hard that the final layers in such deep network will struggle to be affected from the words in the early stages of this neural network. This is called the vanishing gradient of the network.

Note: Another problem is the exploding gradient but it is less likely to happen and the solution to it is what is called by gradient clipping.

Gated Recurrent Unit (GRU):

GRU solves the problem of vanishing gradient descent by using the idea of the gates (update gate and reset gate) and the memory cell.



GRU

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

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

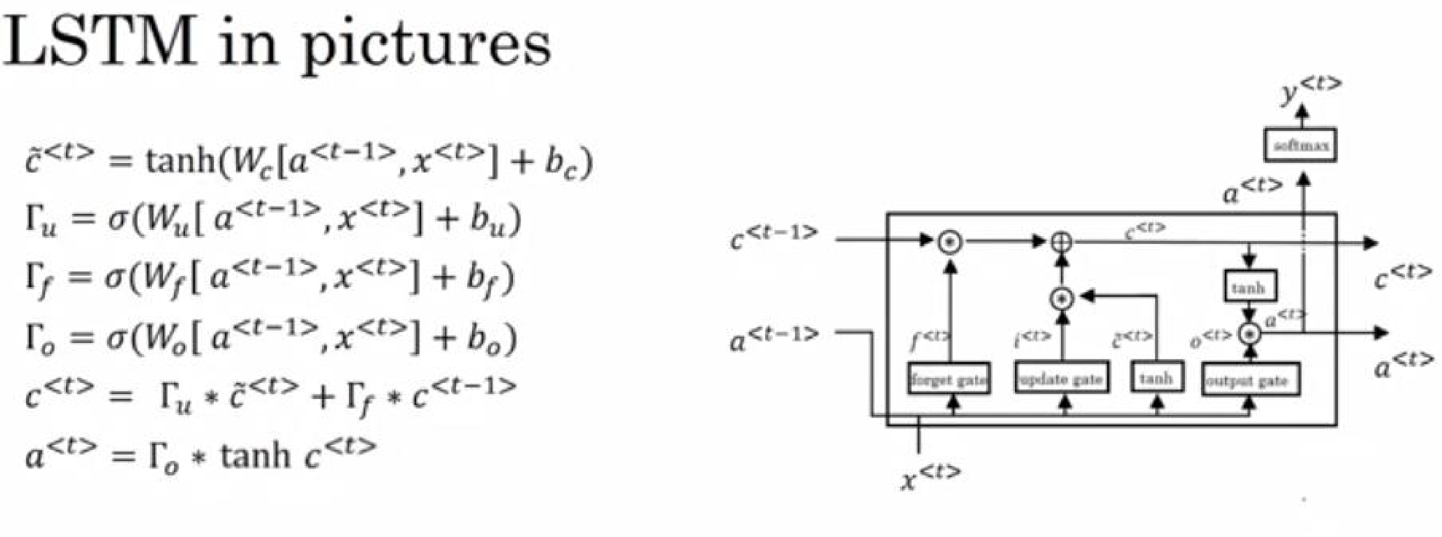
$$\Gamma_r = \sigma(W_r[\, c^{<t-1>}, x^{<t>}] + b_r)$$

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

$$a^{<t>} = c^{<t>}$$

The idea here relies on the concept of passing the important information to the output while blocking the unneeded one. This also is provided with memory to be able to keep this information for long time which is called memory.

Long-Short Term Memory (LSTM):



LSTM in pictures

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

We notice here that it has the same idea of GRU with adding gate for forget and a gate for the output.

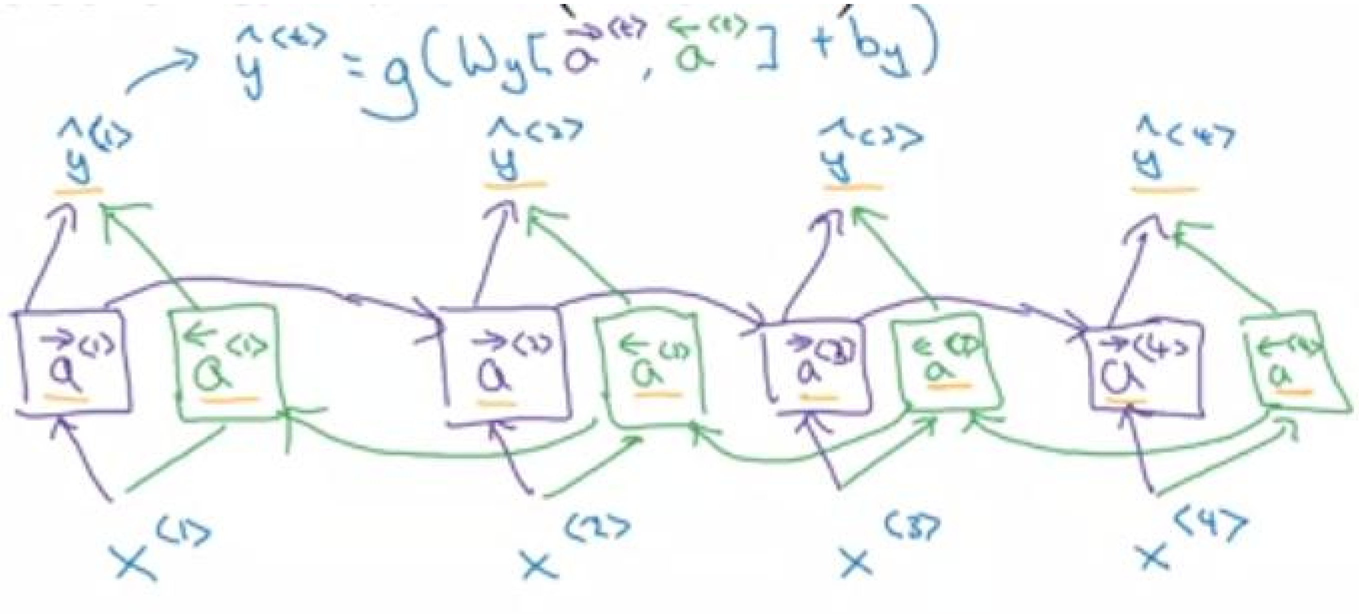
Notes:

1- LSTM was introduced first then GRU was introduced after that as a recent invention.

2- LSTM is more commonly to be chosen as it has proven its success throughout the history.

3- GRU is much simpler as it has only 2 gates so it runs faster than LSTM.

Bidirectional RNN (BRNN):



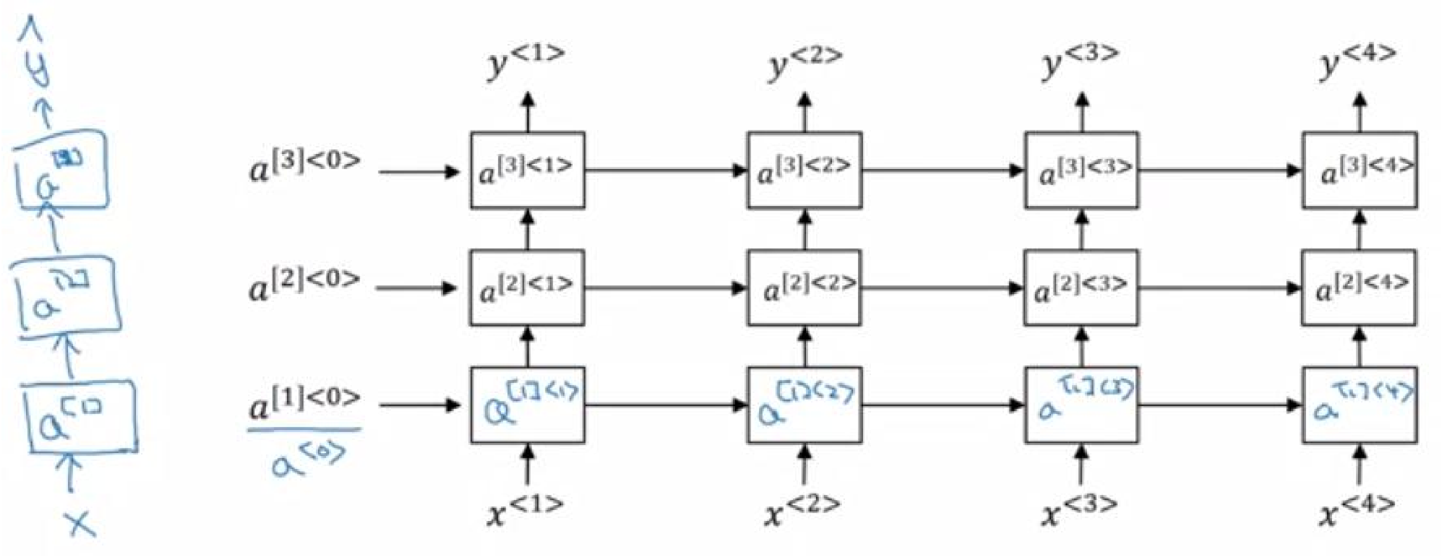
$\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$
$\hat{y}^{<1>}$
$\hat{y}^{<2>}$
$\hat{y}^{<3>}$
$\hat{y}^{<4>}$
$\overrightarrow{a}^{<1>}$
$\overleftarrow{a}^{<1>}$
$\overrightarrow{a}^{<2>}$
$\overleftarrow{a}^{<2>}$
$\overrightarrow{a}^{<3>}$
$\overleftarrow{a}^{<3>}$
$\overrightarrow{a}^{<4>}$
$\overleftarrow{a}^{<4>}$
$x^{<1>}$
$x^{<2>}$
$x^{<3>}$
$x^{<4>}$

As we notice here in this architecture that for each output we have two activations where one gets all the preceding activation while the other gets all the following ones and both gets weighted by w_y to result in the output of the current one in the output sequence.

The disadvantage of such architecture is that you need the entire sequence of data before starting generating the output.

Deep RNNs:

Deep RNNs are actually a sequence of those building block we have talked about in time but stacked across layers to form our deep recurrent neural network as follows:



$\hat{y}$
$a^{[3]}$
$a^{[2]}$
$a^{[1]}$
x
$y^{<1>}$
$y^{<2>}$
$y^{<3>}$
$y^{<4>}$
$a^{[3]<0>}$
$a^{[3]<1>}$
$a^{[3]<2>}$
$a^{[3]<3>}$
$a^{[3]<4>}$
$a^{[2]<0>}$
$a^{[2]<1>}$
$a^{[2]<2>}$
$a^{[2]<3>}$
$a^{[2]<4>}$
$a^{[1]<0>}$
$a^{<0>}$
$a^{[1]<1>}$
$a^{[1]<2>}$
$a^{[1]<3>}$
$a^{[1]<4>}$
$x^{<1>}$
$x^{<2>}$
$x^{<3>}$
$x^{<4>}$