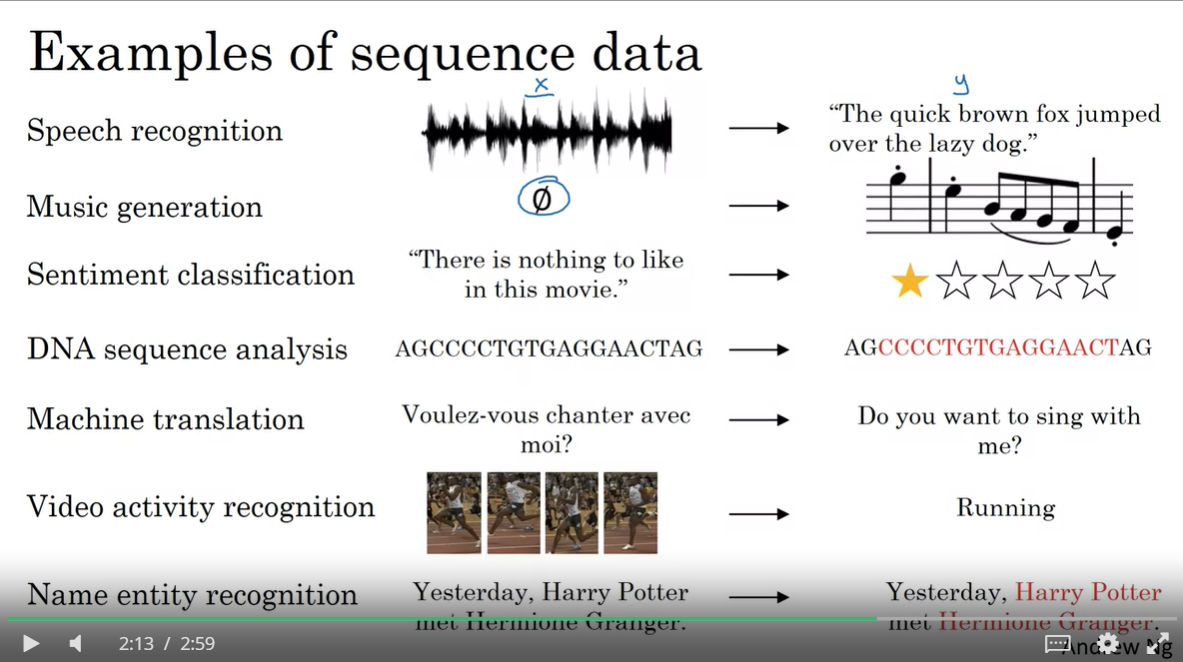
## Some of the usage of Sequence models

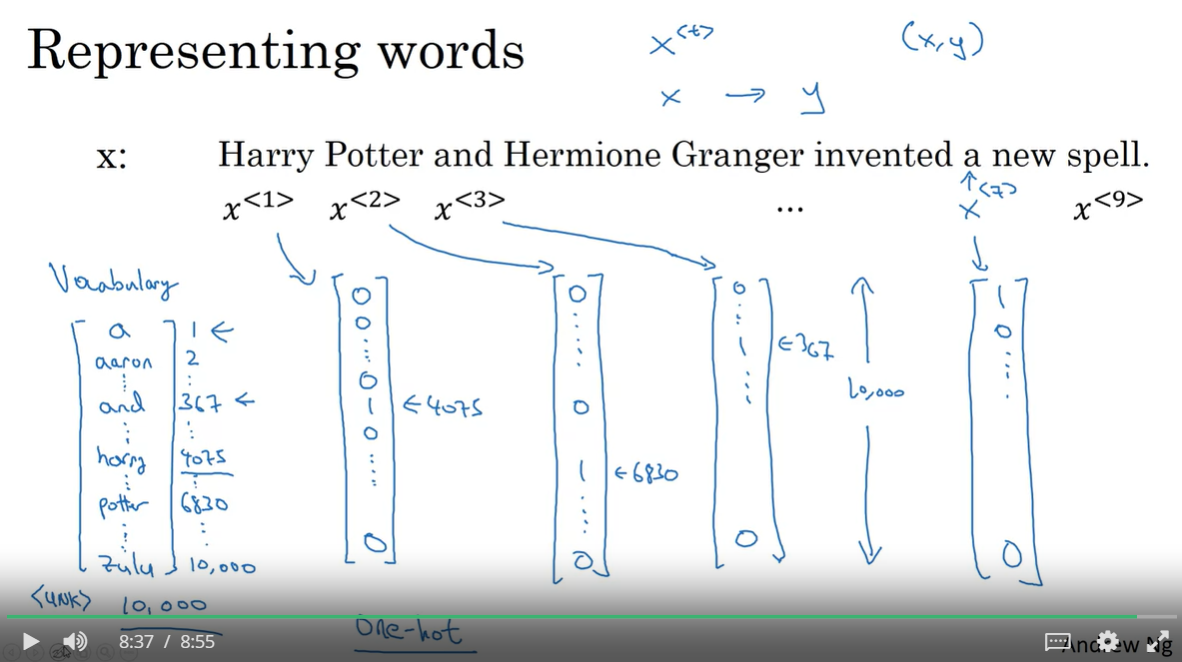


We see that some of the examples has a sequence as an input (X) and a sequence as an output ( Y ) like speech recognition , DNA sequence analysis , Machine translation

And in another cases like Sentiment classification , Name and Video activity recognition we found the sequence data only in the input

And finally , there is examples like Music generation where we found the sequence data in t-he output

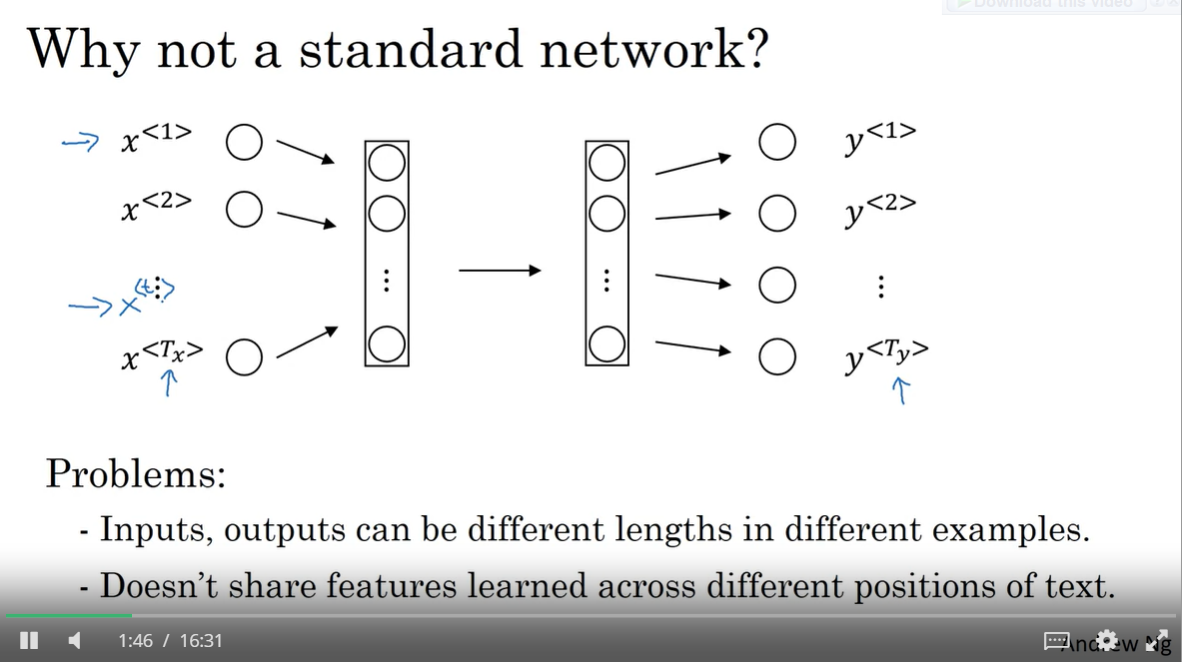
## A Small demo about the sequence models input: name entity recognition



* For name entity recognition we have before everything our Vocabulary vector which contains all the words to recognize ( in this case our vocabulary vector contains 10K words , each word is identified by its position in the vector , the word “and’ for example is identified by 367
  + We add in the last index : a special word <unk> to identify the uknown words by the index 10001
* Our dataset will be surely a sentences , each sentence X contain words labeled by x<t> where t represents its position in the sentence
  + The X<t> will be represents by a **one-hot vector**  with same length as our vocabulary vector ( 10K ) **,** that means it’s in form of vector of zeros and only a one ( 0,0,0,….,1,0,0,….,0) . in our case the ‘1’ will be in the index of the specified word ( so x<3> is ‘and’ so the 1 will be in position [367] )

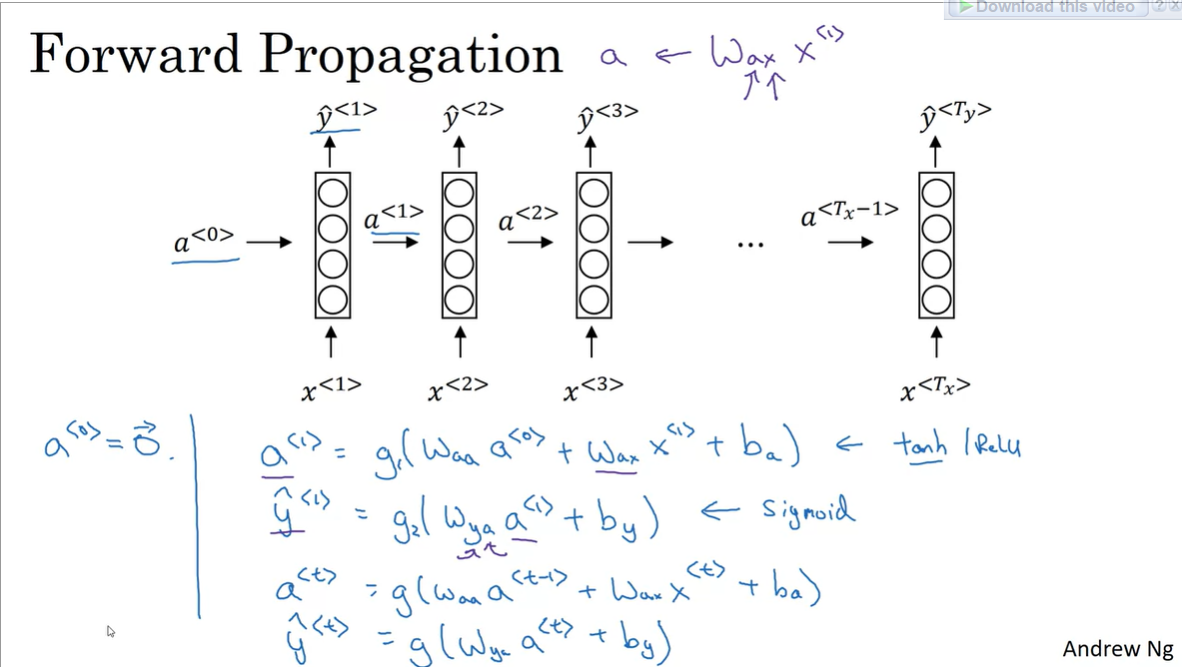
## Recurrent Neural networks RNN model

Instead of the the classical neural networks , we need a new architecture to handle the sequence data due to some of its weaknesses



* There isn’t any specified length of the input layer and for the output layer also ,   
  like we cannot specify the length of the input and/or the output of the Machine translation
* If we regroup all our words in a one vector ( by summing the one-hot vector of each word ) , We lose an important feature in the sequence data which is the order and the position of the text

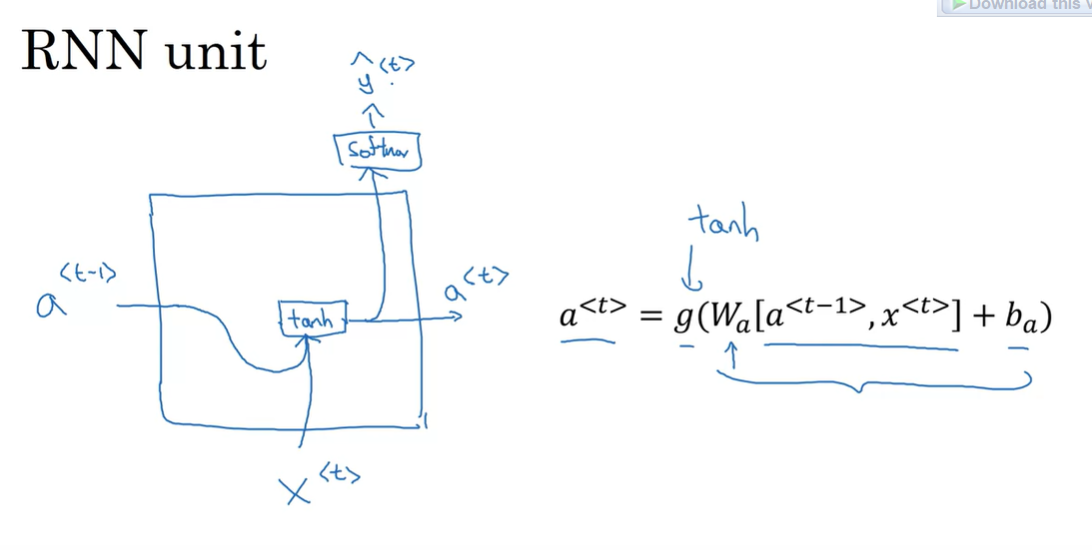
### The New architecture: RNN



* As the architecture demonstrates and its mathematic formula showed : for every prediction y<t> related to x<t> , we are going to use an additional parameter : a<t> , this parameter uses x<t-1> to be calculated , so every x<t> will use the words of the position [1,t-1] to calculate its prediction which is a thing we didn’t see on the classical neural networks , and this is called forward propagation
* The g() function represents the optimizer function
  + We often use ‘tanh’ or sometimes ‘relu’ as the optimizer function to calculate the a<t>
  + We often use “sigmoid’ optimizer to calculate the y<t> ( or “SoftMax” if we found ourselves with categorical-classification problem )
  + In this situation we will geta value between 0 and 1 which represents the probability of a word to be a Named entity , so it’s a binary classification => sigmoid is the idea classifier
* Waa , Wax , Way , by and ba are hyper parameters , the Ws are matrixes and the Bs are biases
* The a<t> are called “activation functions”

### A RNN UNIT :

This is a general representation of each unit inside the RNN



### The RNNs weaknesses:

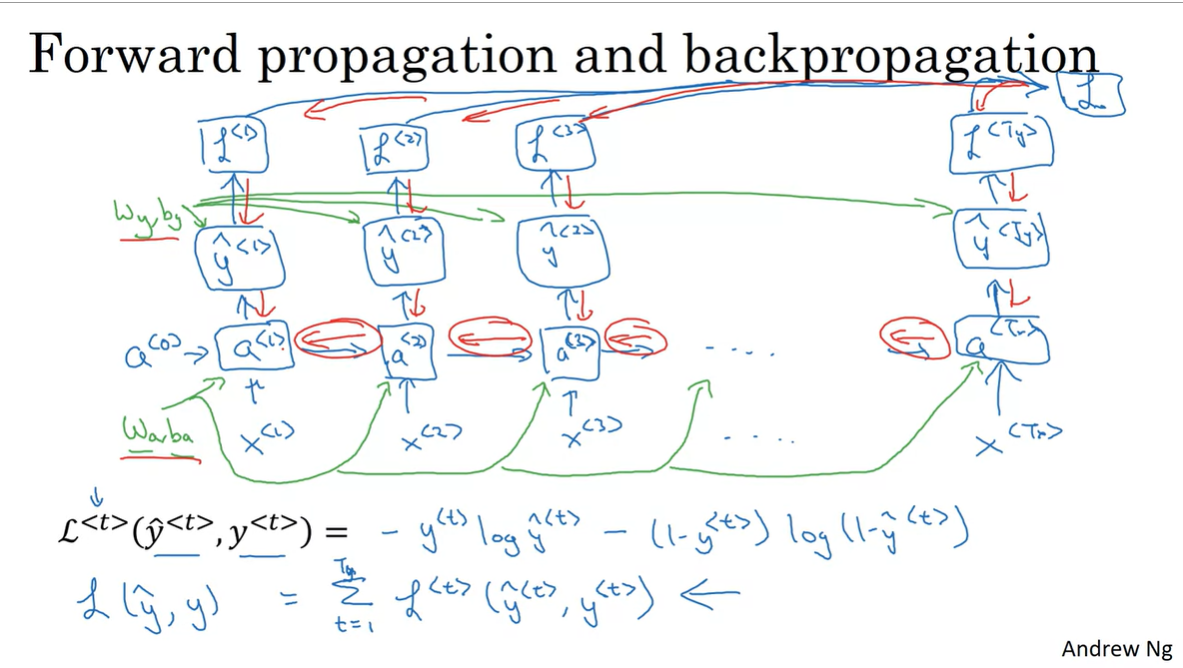
The current model isn’t perfect to do its work with sequence data , and here is a simple example to demonstrate one of its biggest weakness related to Named Entity Recognition:

* Given these two sentences:
  + He said “**Teddy** Roosevelt was a great president”
  + He said “**Teddy** bears are on sale”
    - We cannot decide whether “Teddy” is a named entity or not without inspecting the coming words (x<t+1> , x<t+2> , …. )

So, sometimes looking to the precedent entities isn’t sufficient to do a correct prediction but we might need the next entities

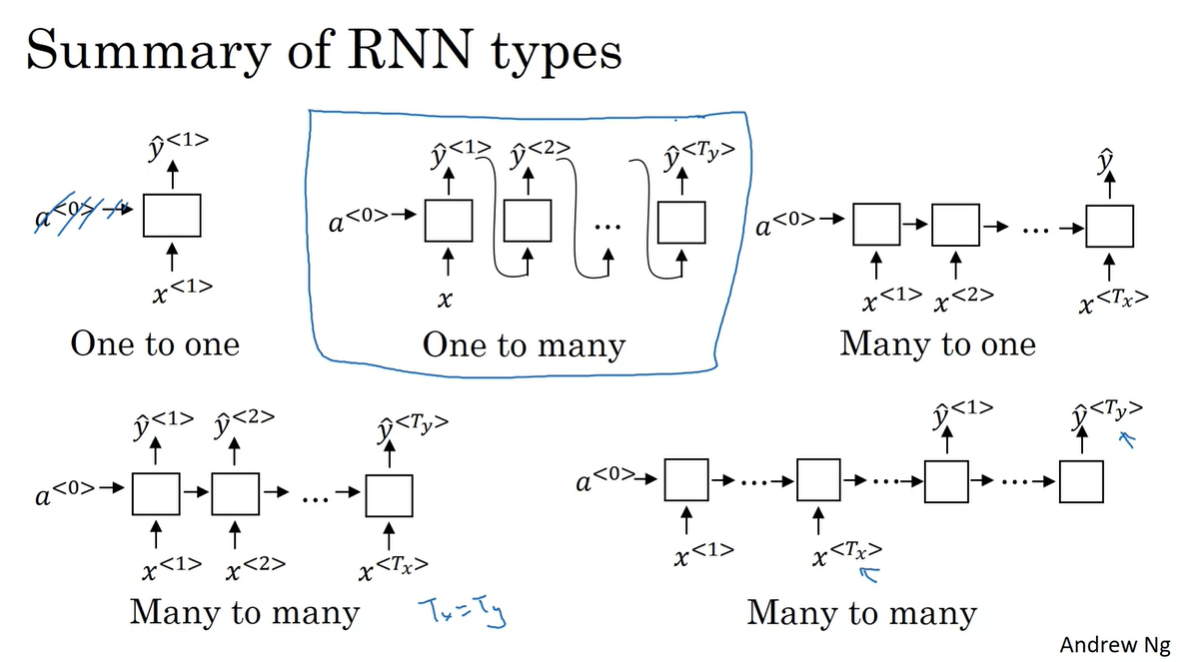
Solution: Backward propagation with Bidirectional RNN (BRNN)

### Forward vs Backward propagation :



As I already wrote, The backward propagation use the future data (x<t+1> , x<t+2> , .. ) , and actually it needs an additional parameter which is the loss function ( the difference between the real value and the predicted one ) , and in this slide it’s the Cross-Entropy one , We will calculate the loss for each y<t> L(y^<t> , y<t>), and then we will sum them to get the global loss L( y^ , y )

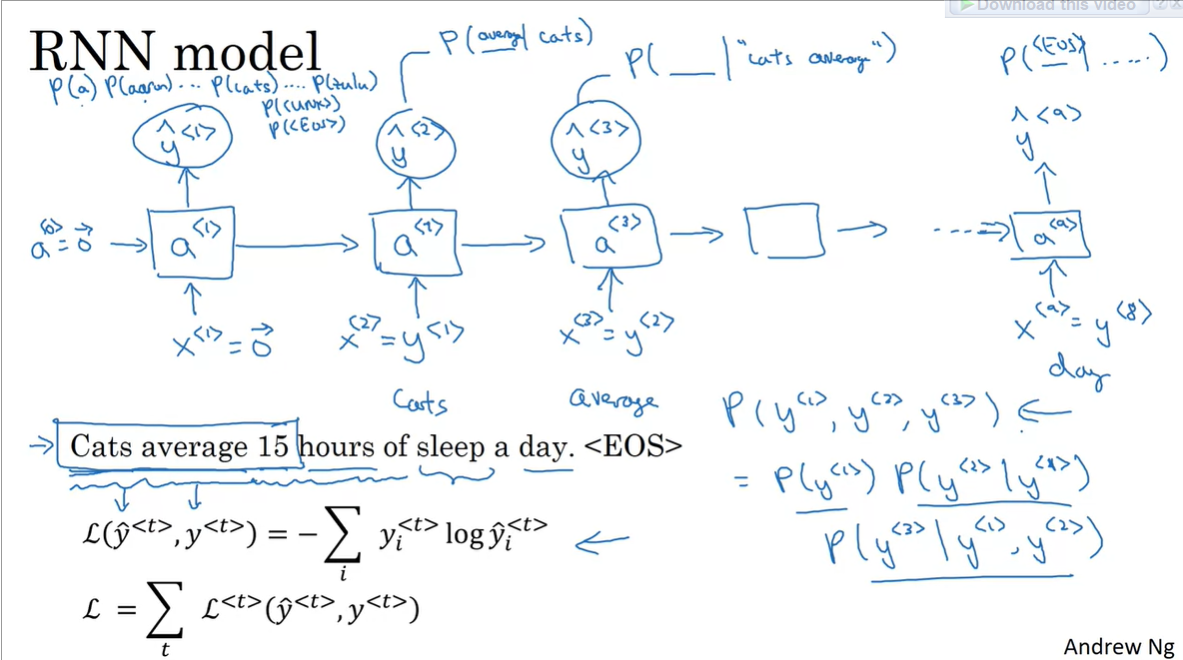
### RNN types :



* **One to one:** this model is the classic Neural network model (for each input we got a single output)
* **Many to Many (Left Side) :** this is the one we represented previously for Named entity recognition problem , for each x<t> we got an output y<t>
* **Many to Many (Right Side ) :** We will get this model if the length of the output is not equal to the input one ( Example : Machine Translation ) , So we read all the inputs x<t> one by one ( Encoding step ) before receiving the outputs y<t> which represents the translated text ( Decoding step )
* **Many to One:** The sentiment analysis takes a text ( sequence of x<t> ) and the output is a single output which represents the corresponding number of stars to the comment ( between 1 and 5 )
* **One to many:** The music generation takes as an input a number representing the Music to generate genre and the output is a set of y<t> which represents the generated music notes

## Language models and Sequence generation

In this section, I’m going to explain how the process of generating a sentence with a sense using the RNN (one to many particularly)



* First of all , We have to feed our model by a training set which is in form of Text from newspaper , articles , books ,…etc
* The sentence generation start by generating the first word , The model is going to look to the training text and using Softmax optimizer ; he is going to gives us the word of the highest probability to be in the start of the sentence , which is “Cats” in this case
* For the second word generation we are going giving our model the first generated word “Cats” as an input ( x<2> = y<1> ) and he is going to gives us a word y<2> with the highest probability to be beside “Cats” : y<2> with Max( P(y<2> / y<1> = “Cats” ) ) = “average”
* For the third word we are going to have the same thing by taking in consideration the first two generated word : y<3> with Max( P( y<3> / y<1> =”Cats” AND y<2>=”average” ) = “15” , and so on !

## Vanishing Gradient problem

### Definitions

- **Vanishing gradient definition:** the gradients of the loss gradient-base function approach zero (like Sigmoid ) , making the network hard to train (learning rate tends to 0)

- **Exploding gradient definition:** it’s the opposite of vanishing gradient : Exploding gradients are a problem when large error gradients accumulate and result in very large updates to neural network model weights during training. Gradients are used during training to update the network weights, but when the typically this process works best when these updates are small and controlled.

### How RNN are risked to have Vanishing gradient problem?

Let’s take as an example of a generated sequence:

* + - The **cat,** which ate fish, cheese and milk **was** full
    - The **cats**, which ate fish, cheese and milk **were** full

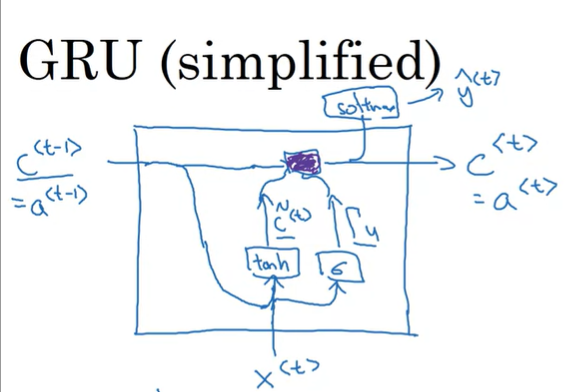
As we see, the y<2> ( “cat”/”cats”) is strongly dependent to y<9> ( “was”/”were” )

And we will get a wrong sentence if we combine cat with were or cats with was .

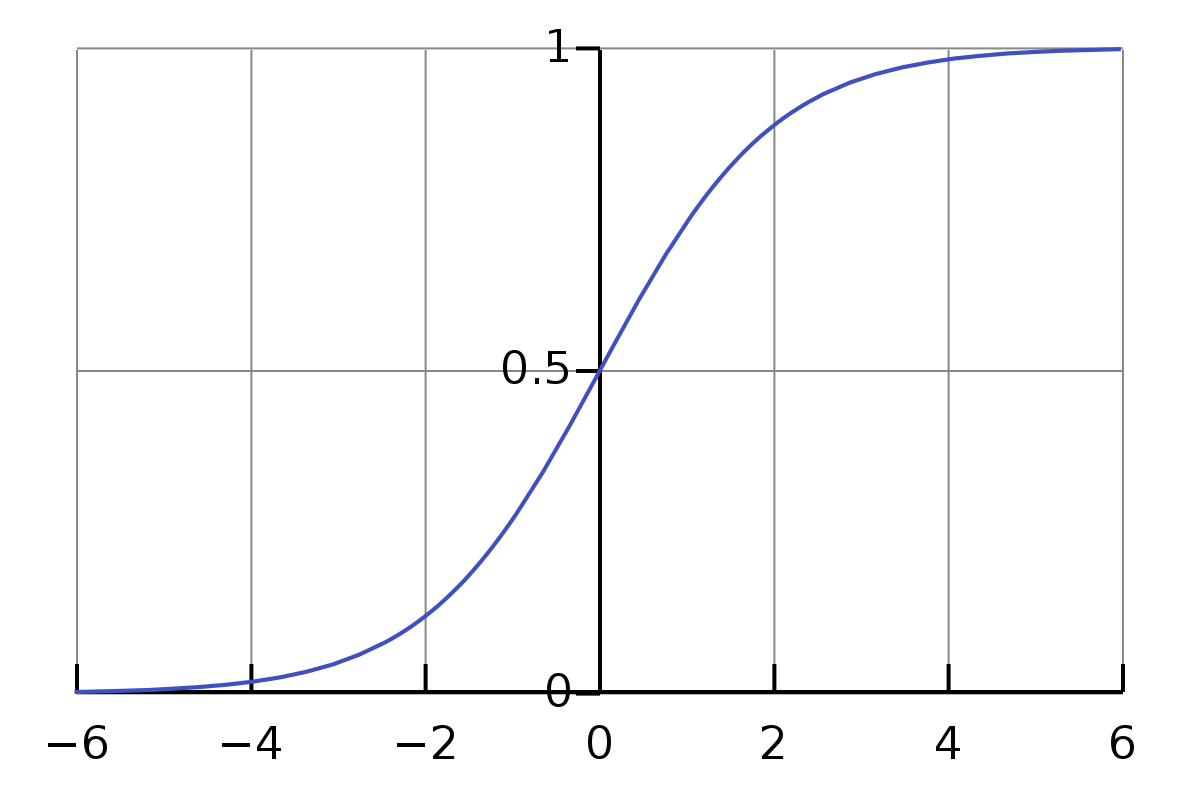
The problem appeared well in this particular sentence because there is a big gap between the dependent words: there are not neighbors, and for the RNN: the y<t> is strongly dependent to its neighbors ( like y<t-1> , y<t-2> , y<t+1> , y<t+2> ..) but in this case there is a dependence between y<2> and y<9> ,

## Global recurrent Unit ( GRU ) is the solution :

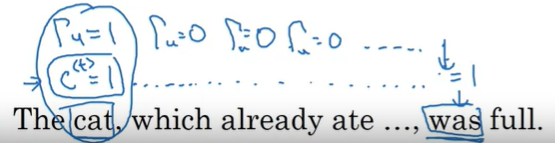
### GRU Architecture ( simplified version ) :



* **C<t-1> , C<t> :**  represents the cell memory , it’s the parameter that we use to memorize that we have as a subject “cat” and not “cats” ( saved information in C : 1 if plural and 0 if it’s singular ) , C could be also a vector instead of a single value in order to store more important information instead of only singular/plural
  + For GRU : the memory cell C<t> is equal to the activation function a<t>
* **c ~<t> :**  It’s the candidature value for c<t> if we gonna erase the memory cell c<t-1>
* **Γu :** called “Update gate” , it’s value is between 0 and 1 ( due to Sigmoid function ) , actually is either too close to 1 or too close to 0 ( thanks to sigmoid function )



The role of the Update gate is to decider whether we update the current value of the memory cell c<t> or change it , here is an example while inspecting our sentence :

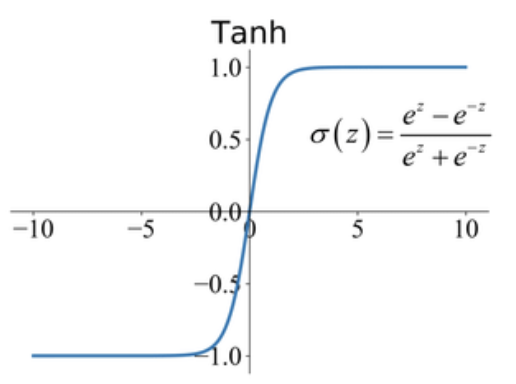


* The **Γu**  was 1 when meeting the word “cat” , and she stored value 1 in c<t> ( 1 for singular ) , and then **Γu**  continues of being 0 in order to keep the stored value of c<2> in c<3> , c<4> …… until we reach the word “was”

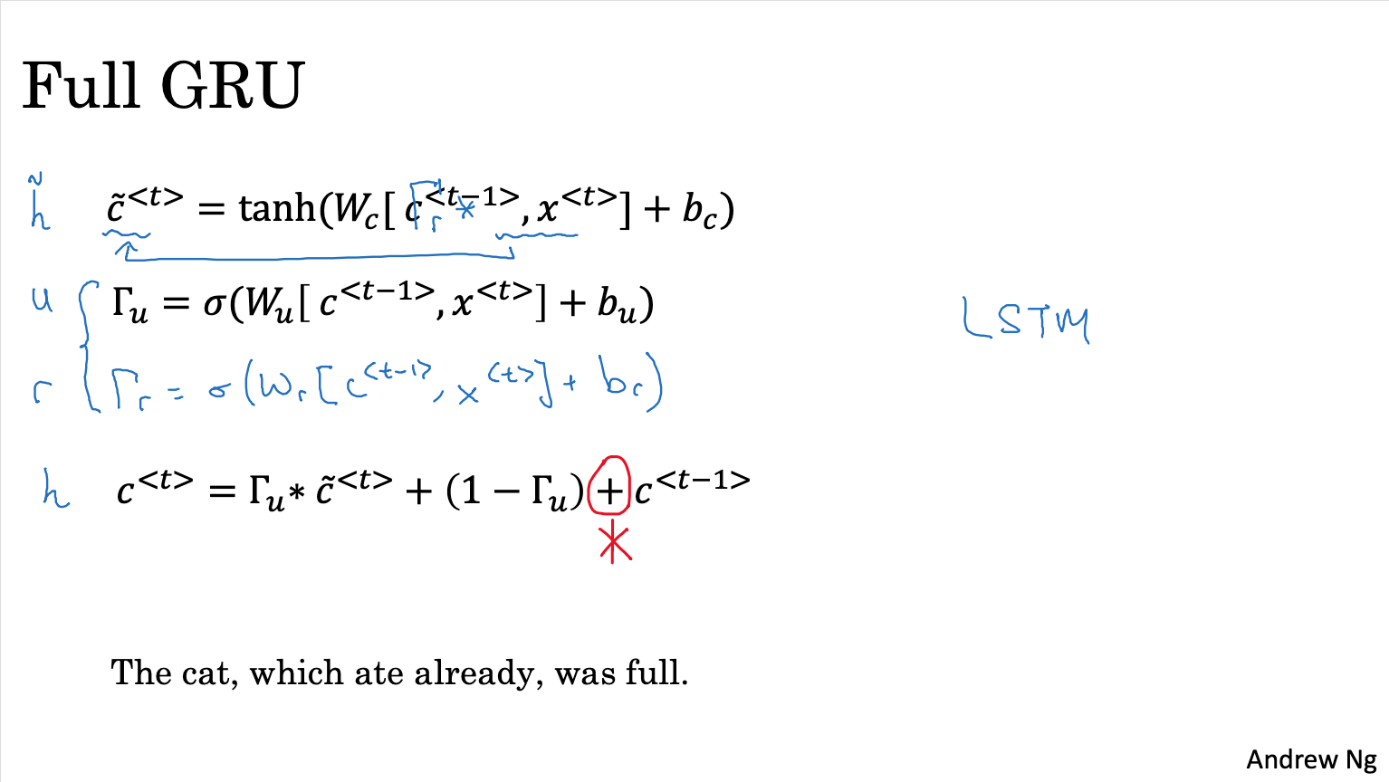
### The mathemathic formulas :

### 

* Quick interpretation about the c<t> formula:
  + **If Γu is close to 0 :** 
    - C<t> = C<t-1> , and this means that we are saving the old information for the coming iterations
  + **If Γu is close to 0 :** 
    - C<t> = c ~<t> , and this represents a new information we are going to store ( it uses tanh function which has a values between -1 and 1 )



### Full GRU formula :



It’s almost identical to the simplified one except of adding a new parameter which is **Γr**

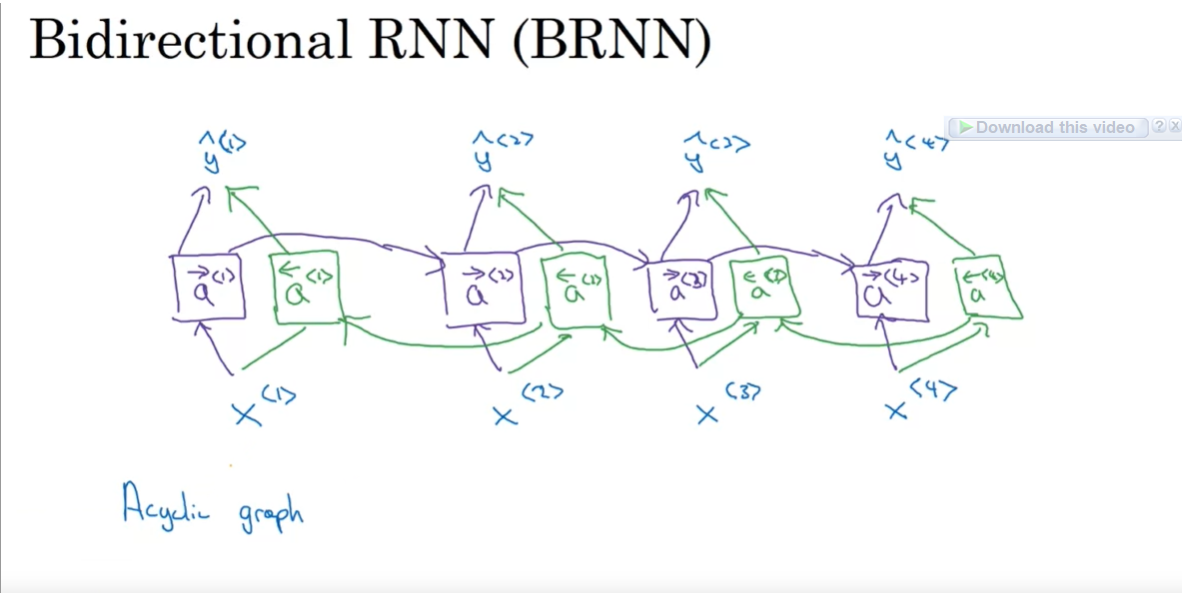
* **Γr :** called “relevance gate” **,** it tells how much c<t-1> is relevant to c<t>, it’s value is between 0 and 1 due to the Sigmoid function
* This new parameter is useful to avoid more and more the vanishing gradient issue

## Long Short Term Memory ( LSTM )



* As we see, LSTM and GRU are many common things except of additional changes in LSTM :
  + a<t> is no longer equal to c<t>
  + We have now three gates **Γ** all of them are sigmoid function ( their values between 0 and 1 )
    - **Γu : “update gate” :** it decides whether we are going to affect our cell memory c<t> by the new condidtaure value c~<t> or not
    - **Γf : “forget gate” :**  it decides whether we are going to keep the stored value in c<t-1> or not
    - **Γf : “output gate” :**  used to calculate a<t> using c<t> , it’s called output because it decides whate we gonna send as a prediction
  + Thanks to the update and the forget gate we can combine in the actual memory cell c<t> : the old memory c<t-1> and an updated info c~<t>
  + GRU is simpler and it takes less time in the computation while the LSTM is more powerful but it takes more time to do its computation

## Bidirectional RNN ( BRNN ) :



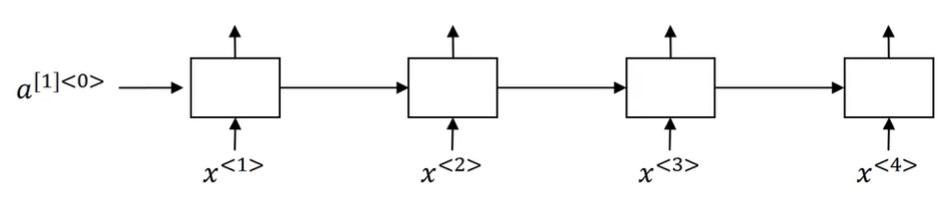
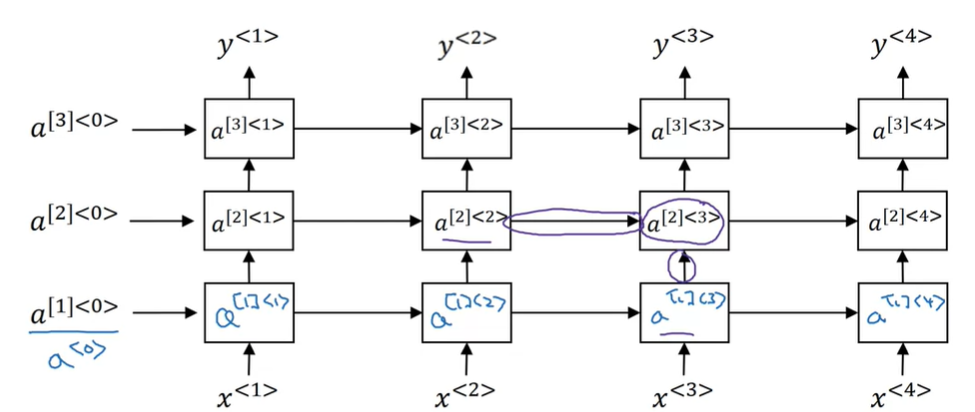
* Given these two sentences:
  + He said “**Teddy** Roosevelt was a great president”
  + He said “**Teddy** bears are on sale”
    - We cannot decide whether “Teddy” is a named entity or not without inspecting the coming words (x<t+1> , x<t+2> , …. )
* To resolve this kind of issues we need to combine the forward propagation ( reading the precedent text : From left to right) with a backward propagation (reading the future text : From right to left )
* Beside our four activation functions a<1> -> , ….. , a<4> -> , we are going to need four more activation functions for the backward propagation , we notate them by a<1> <- , ….. ,a<4> <-
* The chronological order of the computation of the activation functions in BRNN is : a<1> -> , ……, a<4>-> , a<4> <- , ….a<1> <- : we compute the forward activation function from right to left then the backward propagation functions from left to right
* After this computation we can calculate the y^<t> the predicted output using a<t> -> and a<t> <-
* This technique can be applied to the classical RNN, GRU and LSTM
* For NLP problems: the most common solution is LSTN architecture with BRNN technique

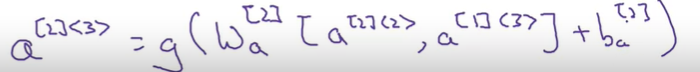
### Disadvantages of RNN:

Since we need the whole sequence to do our prediction (Past and future) : we cannot do this kind of architecture to handle a real-time problems ( because there is no end of the sequence ) , and for speech recognition system : We cannot do an instant translation : But we must wait till the user ends recording its vocal , and then we can start our speech recognition process

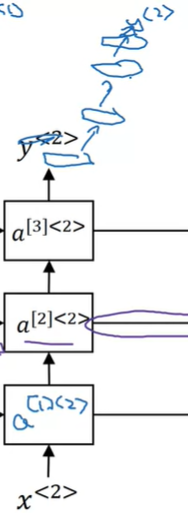
## Deep RNN

It consists of stacking up more than a one layer of RNN ( as we already saw so far )

* + - **RNN/LSTM/GRU with one layer architecture:**
    - **DRNN architecture for classical RNN/LSTM/GRU :**
* The output of a layer will be fed to the next layer
* Each layer has its own activation functions the layer one have a[1]<0> , …, a[1]<4> and so on
* The activation functions of a layer will be calculated using activation function from the bottom layer , for a[2]<3> for example : a[1]<3> is Involved beside a[2]<2> , and here is its formula :



* The predicted output y<t> is the output of the last layer
* We can instead of using directly the output of the last layer : We use it as an input to our Deep neural network ( which is not recurrent ) like it showed in this image for y<2> :



* Generally , a three layers are enough to implement a deep RNN ( and not more ; because RNN takes too much time top do its computation )

## What I’ve learned from the assignments :

* Usually we do the training by batches , and that means while fitting our model we pass more then one training example ( one example = 1 sequence like sentence ) simultaneously to gain time
  + The size of the mini batches are  (𝑛𝑥,𝑚,𝑇𝑥) :
    - **Nx :** is the vocabulary size ( one hot vector )
    - **m :** is the size of the batch ( number of trainings per batch )
    - **Tx** : is the size of the longest sequence
  + x<t> in this case is equal to X[: , : , t]
  + The dimension of the prediction is y^ is ( ny , m , Ty ) because we are going to predict a one-hot vector for each timestep In the sequence
* the activation a<t> is called “hidden state”
* np.dot() or @ are used to do a real mathemathical matrix multiplication while \* is used to do a scalar multiplication ( a[i][j]\*b[i][j] )
* For NLP process , In the character-level sequence generation ( we generate character per character and not world per world ) we should replace each character by a unique index to replace it , because the RNN accepts only numerical values , and for each training example ( word) we end ‘/n’ as a significance about the end of the generated word . the ‘\n’ should be represented too in our dictionary char->index so when our model generates it , we will know it’s the end of the generated sequence
* While Vanishing gradient is a common problem in the RNNs , we must take ion consideration also the Exploding gradient issue by using the gradient clipping technique
  + Exploding gradients make the training process more difficult, because the updates may be so large that they "overshoot" the optimal values during back propagation.
  + The gradient is represented by the parameters : "dWaa", "dWax", "dWya", "db", "dby"
  + We define a max and a min value for the gradient array values [-max , max ] , if a value is inferior than -max we replace it by -max and if a value surpass max we replace it by max , and by this process we assure a controlled values under the gradient
  + We use the function **np.clip()** to achieve that
* For the sampling ( a sequence generation process after the training ) : We said that we will take y<t> with the highest probability , the occurred problem is we will fell always into the same sequence ( if y<0> with Max(P(y<0>) is “cats” , then all our generated sequences will start with “cats” and we will get the same y<1>,y<2> , ….y<ty> always and this is not really interesting )
  + To make the results more interesting, use np.random.choice to select a next letter that is likely, but not always the same.
  + his means that you will pick the index (idx) according to the distribution:  
    if the output of the softmax function is y= [ 0.1 , 0, 0.7 , 0.2] then 𝑃(𝑖𝑛𝑑𝑒𝑥=0)=0.1,𝑃(𝑖𝑛𝑑𝑒𝑥=1)=0.0,𝑃(𝑖𝑛𝑑𝑒𝑥=2)=0.7,𝑃(𝑖𝑛𝑑𝑒𝑥=3)=0.2 .
  + We use the function **np.random.choice()**  to achieve that