## Sequence models with TensorFlow for NLP:

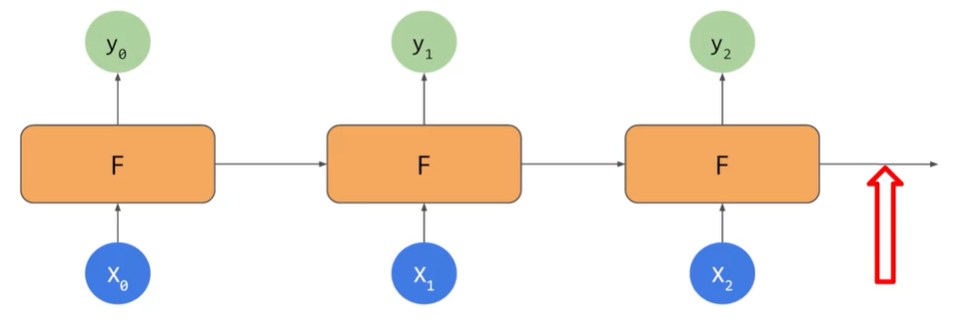
* Our previous process ( tokenizing and padding the sequence ) didn’t take in consideration the relative order between the words while learning the rules or while generating the output of the model for example “the cat sit on the hat” and “the hat sit on the cat” two sentences with the same words but with different orders
* Another issue is sometimes the context of a word is found in a position far of the target word, and without having any mechanism of memorizing the sentence : our model will be accurate in some specific cases like in the sentence “In **Ireland** , We learnt In school how to speak **\_\_\_” .**
  + Thanks to the word “Ireland” we can know that the word after “speak” should be “Irish”
* As another example to show the importance of the relative order of the words and memorizing the context is:



* And because of these two cases : Implementing a RNN mechanism ( or even LSTM mechanism with contains the cell state ) is necessary

## Quick Illustration:

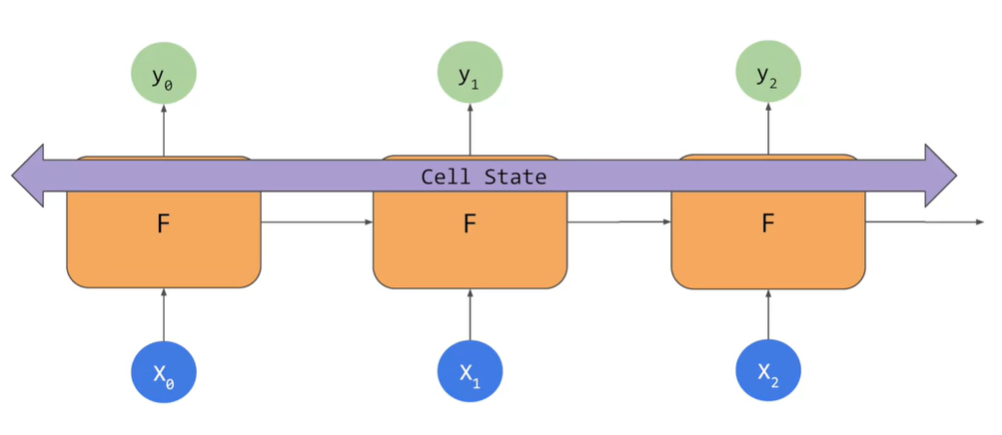
#### RNN :



* For RNN : In the red arrow , the output take in consideration X0 ,X1 and X2 in order to generate the Y3

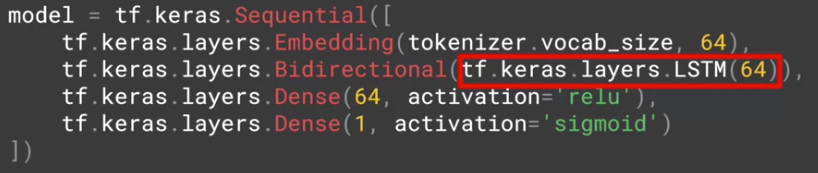
#### LSTM :

* The problem with RNN is that for the long sequences : It’s hard to memorize the whole previous sequence while generating/looking in the target word ( like The Ireland Example ) and we will end up by loosing the context

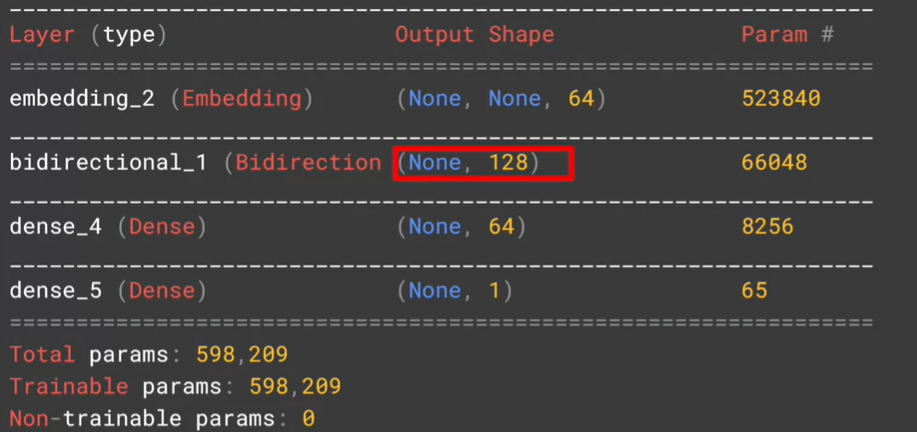


* The LSTMs has in addition to the RNN architecture : a new pipeline which holds the cell state in order to keep the context from the whole sentence
* In this example , the Cell state is bidirectional so it will take in consideration of the previous and next words while learning to generate the output y<t>

## LSTMs in TensorFlow:

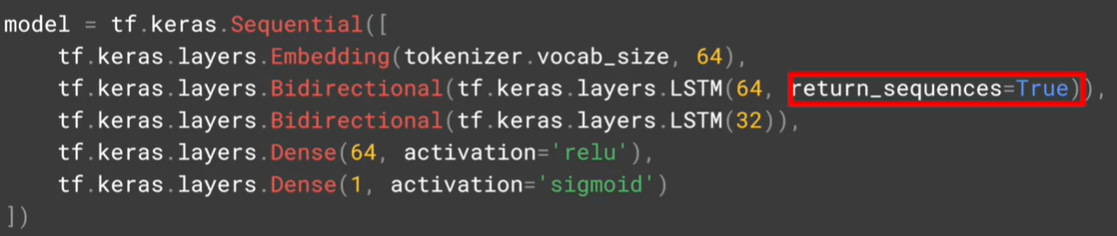


* We are going to keep our previous and we will just add additional layer after the Embedding Layer
* We replaced the GlobalAveragepooling1D/Flatten layer by a LSTM layer with 64 units
* We wrapped that new layer with Bidirectional to make the cell state go in the both direction of the sentence

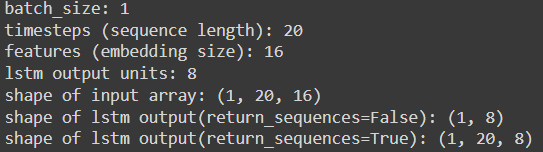


* By looking to the model summary , we notice that the Bidirectional Layer got in its output shape , in the last dimension : 128 which is the double of the output shape of the previous layer : Embedding (64) due to the bidirectional behavior ( going from right to left hand from left to the right )

## Deep LSTMs in TensorFlow:

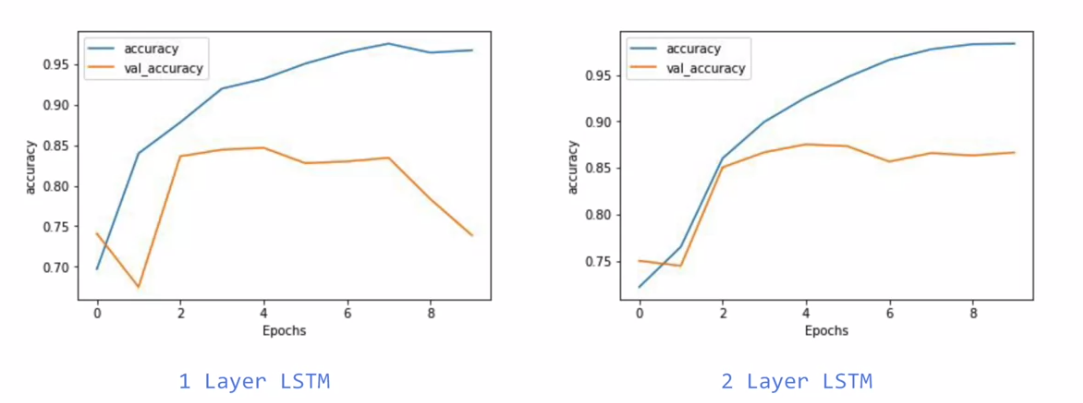


* In order to implement a deep LSTM , the first LSTM layer should have return\_sequence=True because it’s going to be fed to another LSTM layer so thanks to this argument , we are going to assure that the output of the first LSTM layer match the input of the second LSTM layer
* Here is a comparison of the outputs shape with and without return\_sequence=true



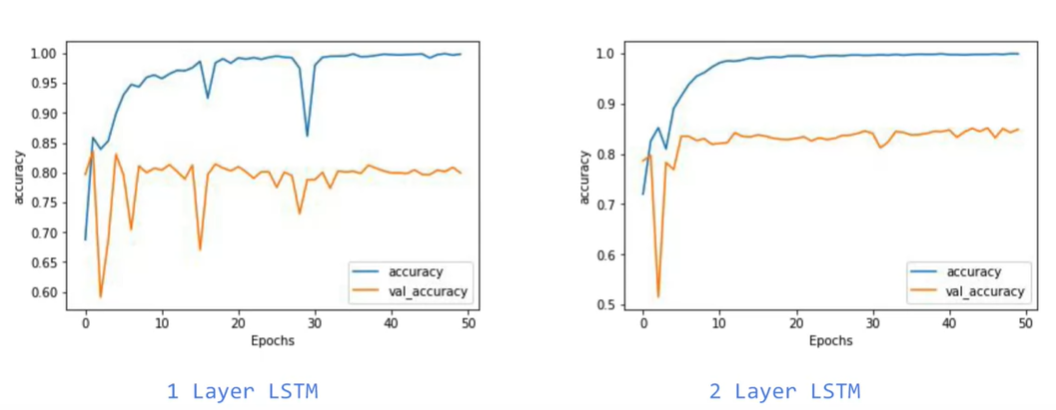
## Comparing between the statistics of 1 Layer LSTM vs 2 Layers LSTM:

#### For 10 epochs :



* For the training set , the 2 layer LSTM has a smoother graph ( less of edges ) which is a good signification that the model is improved
* For the validation set , we notice that there is a drop of val\_accuracy for 1 LSTM Layer after some epochs which is a sign of an overfitting unlike the 2 Layer LSTM’s where we see that the val\_accuracy continues forward for 85%

#### For 50 epochs :

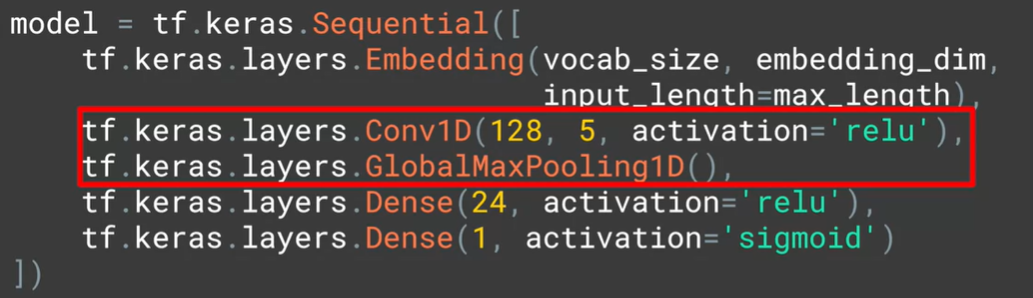


* Globally , we see that the 2 layer LSTM is less jagged for the val\_accuracy which proves that he is more stable generally when it comes to validating the set

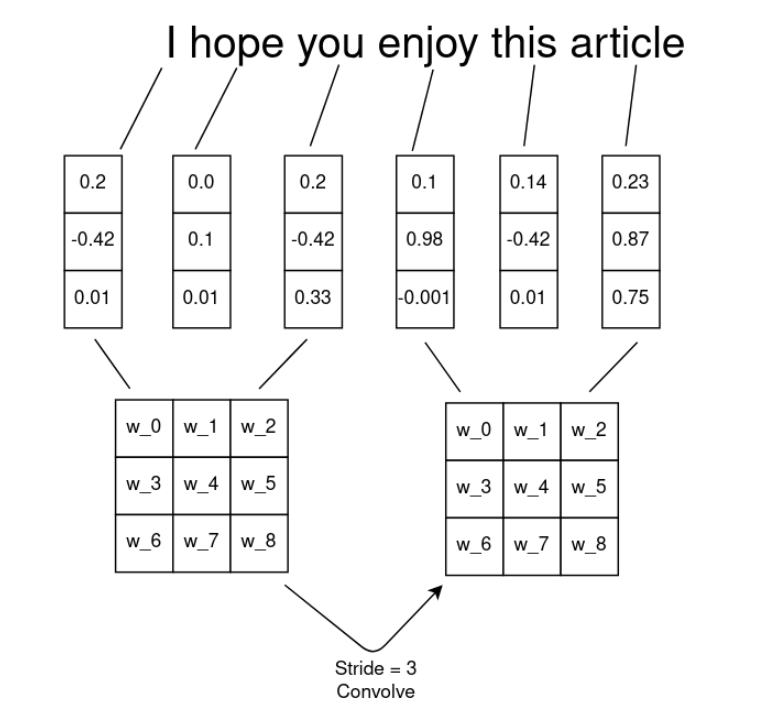
## Comparing between the statistics of using LSTM vs not using it

* We see that the accuracy of training set continues climbing for LSTM version until reaching a value close to 100% while it et stuck in 85% for the Non-LSTM version
* For the validation set , we notice that for the Non-LSTM it reached quickly 80% accuracy and it gets stabilized on this value , but for the LSTM version , we got in the first epochs an accuracy close to 82% but it was continuing dropping to values closed to 78% which is a signification of the overfitting ( because the training accuracy keeps climbing )
* The potential solution of this overfitting is by messing with the hyper parameters and changing them

## Convolutional layers are also a used for NLP sequence data ! :

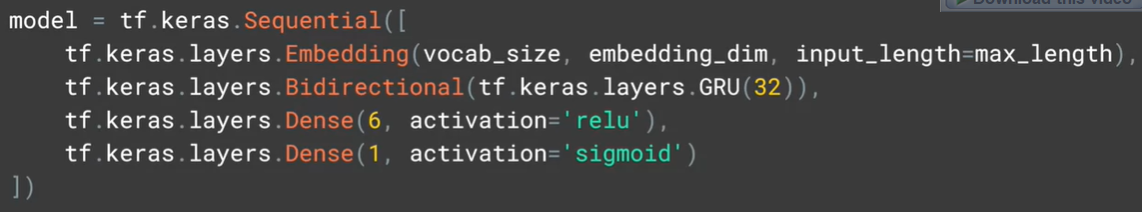


* Using Conv1D followed by GlobalMaxPooiling1D is also a valid architecture to treat the sequence textual data
* The first argument ( 128 ) indicates the number of convolutions ( filters ) to apply , the second argument ( 5) is to indicate the size of the convolution , and as usual we use “relu” as activation function



* The idea is to concatenate the word embedding vectors per group , each group has a dimension equals to the filter size , and then we do the conv manipulations ( wise-multiplication by the filter followed by a MaxPooling )

## Using GRUs in TensorFlow:



* No explanation needed , everything is clear
* If we need to do Deep GRUs , we will do as we used to with LSTMs , we specify return\_sequences=True for the GRU layers that are expected to be fed to another GRU layer

## NLP tasks and the overfitting:

with text, you'll probably get a bit more overfitting than you would have done with images. Not least because you'll almost always have out of vocabulary words in the validation data set. That is words in the validation dataset that weren't present in the training, naturally leading to overfitting, these words can't be classified and, of course, you're going to have these overfitting issues.

* There isn’t any global solution to avoid overfitting otherwise tweaking the hyper parameters of the tokenizer , the pad\_sequence method , and the tensorflow model

## My personal comparison between LSTM , GRU and CONV1D

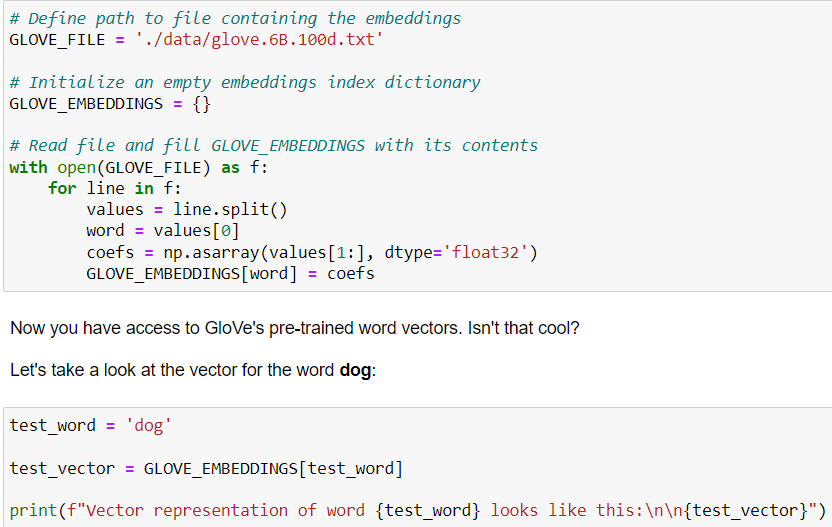
* The LSTM takes too much on the training
* CONV1D surprisingly have an accuracy while training almost equal to the LSTM one

## Using pre-trained embedding matrix and integrating in our model instead of the keras embedding layer

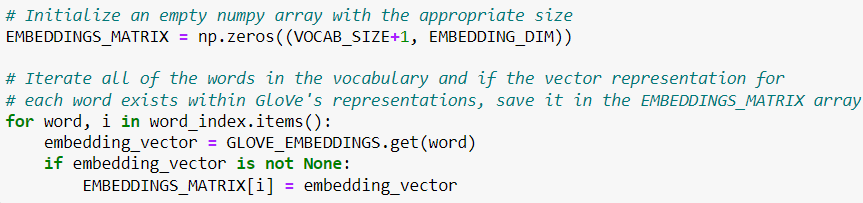
* Using a pre-trained embedding matrix has a benefits that we couldn’t find in the Embedding layer of keras
  + Since it’s pretrained , surely it was trained from a huge corpus so It will cover almost all the vocabulary words

#### How to use pre-trained Embedding in Tensorflow

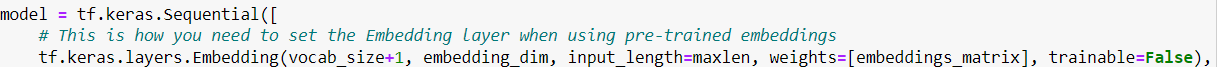
* Importing the embedding matrix in Python:



* Getting the words in my vocabulary using the embeddings:



* + We initialized our zeros-matrix with dimension VOCAB\_SIZE+1 because the word\_index keys start from 1 and not 0
* Calling the pre-defined Embedding in our Sequential model :



* We specified the values of the pre-defined embedding using ‘weights’ arg
* The trainable=False arg is specified to fix the values of the weight without getting changed while fitting