Metidji Sid Ahmed Natural Language Processing in TensorFlow DeepLearning.ai Course summary

I. From words to Numbers (words encoding):

- To pass from words to numbers (the classical way that our Neural Networks do the training and the prediction), we should give to each word a unique Id
- This process is included in the Tokenization process where we switch from String Data to vector of integers (the words index)

A. The code :

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'I love my dog',
    'I love my cat'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
print(word_index)
```

- We are going to use Tokenizer class from Keras
  - **Num_words=100 :**
    - it will take only the top 100 words which appears the most in our training text and encode them , for the other words : he will just ignore them
  - **Fit_on_texts(sentences) :**
    - Using this method , we will feed the Tokenizer by our text in order to encode its words
  - **.word_index** :
    - It has the Python dict which contains our text words as Key and their corresponding number as Value:

```
{'i': 1, 'my': 3, 'dog': 4, 'cat': 5, 'love': 2}
```

    - The words appearing the most will have a lower Values

➢ Interesting characteristics about the Tokenizer:

- In the image above, we see that it contains 'i' instead of 'I' and this because the Tokenizer lower all the word cases in order to normalize the text
- In the image below, we see by adding a new sentence with "dog!" as a word , Tokenizer is clever enough to consider it exactly as "dog" because it gets rid of the punctuation marks while doing the words encoding process

```
sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!'
]
```

```
{'i': 3, 'my': 2, 'you': 6, 'love': 1, 'cat': 5, 'dog': 4}
```

II. Text to sequence + the Padding

B. Convert my training data to tokens

```
from tensorflow.keras.preprocessing.text import Tokenizer

sentences = [
    'I love my dog',
    'I love my cat',
    'You love my dog!',
    'Do you think my dog is amazing?'
]

tokenizer = Tokenizer(num_words = 100)
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index

sequences = tokenizer.texts_to_sequences(sentences)

print(word_index)
print(sequences)
```

- Using texts_to_sequence() method in my training data , this will convert my array of string to array of vectors of the corresponding tokens to each word.

```
{'amazing': 10, 'dog': 3, 'you': 5, 'cat': 6,
 'think': 8, 'i': 4, 'is': 9, 'my': 1, 'do': 7,
 'love': 2}


[[4, 2, 1, 3], [4, 2, 1, 6], [5, 2, 1, 3], [7, 5,
8, 1, 3, 9, 10]]
```

- For the first row of the output : "I love my dog" was converted to [4,2,1,3] (4 = I , 2 = love , ...etc)
- The tokens will be associated according to the last called method to *.fit_on_texts()*

```
test_data = [
    'i really love my dog',
    'my dog loves my manatee'
]

test_seq = tokenizer.texts_to_sequences(test_data)
print(test_seq)


[[4, 2, 1, 3], [1, 3, 1]]

{'think': 8, 'amazing': 10, 'my': 1, 'love': 2, 'dog': 3, 'is': 9, 'you': 5, 'do': 7,
'cat': 6, 'i': 4}
```

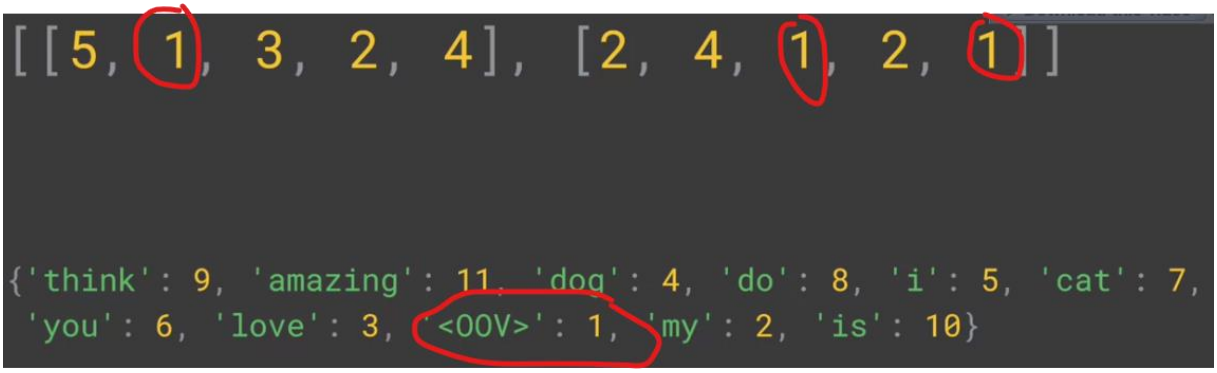
- In the image below , we tried to call *texts_to_sequence()* method in a new corpus which contains a new words like "really" and "manatee" comparing to the last time we called *fit_on_texts()* and "I really love my dog" gets the same sequence as "I love my dog"
- The conversion to sequence will simply ignore these unknown words to him

➢ Making the tokenizer taking in consideration the unkown words :

```
tokenizer = Tokenizer(num_words = 100, oov_token="<OOV>")
```

- Thanks to the argument *oov_token* , we are assigning the token '<OOV>' to the unknown words compared to the fitting phase (oov = out of vocabulary)

- The image below shows the final result after taking in consideration the unknown words:



And now , the words “really” and “manatee” has 1 as a token

➢ Padding the sequences:

```
padded = pad_sequences(sequences)
```

- you will usually need to pad the sequences into a uniform length because that is what your model expects : a constant length of sequence , and we will do that thanks to the method *pad_sequences()* , it will add 0s in the left (or the right) of the sequences in order to make them in the same length (usually equals to the longest sequence) , and this is the final output :

- Interesting args to pad_seqeunces :
  - *padding=”post” / padding=”pre” :*
    - It tells the method where it should put the 0s (post = in the right and pre= in the left)
  - *Maxlen= Number & truncating=’post’ / truncating=’pre’:*

- It tells the max size of the sequence after the padding process , it will cut words from the sentences who had longer length
- The truncating argument tells the method where it should do the cut : from the right or the left

C. Resming the pre-processing step :

1. Initialize the Tokenizer class
2. Calling fit_on_text() on the training corpus text :
   a. That corpus should be a huge text in order to cover the max possible words in the language in order to avoid the <OOV> token in our training set
   b. Its role is to generate a unqiue token to each word
   c. It returns Key-Value dict , the key field contains the words while the value contains their corresponding tokens
   d. We can check this dict by accessing to .word_index attribute
3. Calling text_to_sequences() on our training set :
   a. Its goal is converting our array of sentences to array of sequences (vector of tokens)
4. Calling pad_sequences() on the sequences array :
   a. Its goal is to unify the sentences length (to be generally equal to the max length sentence by adding 0s in the post/pre sequence depending to truncating arg)

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Initialize the Tokenizer class
tokenizer = Tokenizer(oov_token="<OOV>")

# Generate the word index dictionary
tokenizer.fit_on_texts(sentences)

# Print the length of the word index
word_index = tokenizer.word_index
print(f'number of words in word_index: {len(word_index)}')

# Print the word index
print(f'word_index: {word_index}')
print()

# Generate and pad the sequences
sequences = tokenizer.texts_to_sequences(sentences)
padded = pad_sequences(sequences, padding='post')

# Print a sample headline
index = 2
print(f'sample headline: {sentences[index]}')
print(f'padded sequence: {padded[index]}')
print()

# Print dimensions of padded sequences
print(f'shape of padded sequences: {padded.shape}')
```

III. Important Remark about max_words argument in Tokenizer instantiation

- We know that the max_words arg tells the Tokenizer class how many top words to take in consideration while tokenizing our sentences

```
number of words in word_index: 29657
word_index: {'<OOV>': 1, 'to': 2, 'of': 3, 'the': 4, 'in': 5, 'for': 6, 'a': 7, 'on': 8, 'and': 9, 'with': 10, 'is': 11, 'new'
```

- We see that the word_index size is 29657 even though we specified max_words=100
    - This because that the max_words argument has nothing to do with the length of word_index , in the all cases it will contain all the words seeing in the fitting
    - The difference is shown when we call texts_to_sequences() method :
        - If the word token in inferior than 100 : we will associate to him its token
        - If it's superior we will associate to him the '<OOV>' token
- The challenge with max_words arg even it's useful to eliminate the rare words : we have to make sure that the chosen tokens (whose index inferior than max_words) must be semantic and useful words , and not just 'to' , 'of' , 'the' ..etc like we are seeing in the image above
    - One possible solution is to eliminate and delete these meaningless pronouns from the training set before calling fit_on_texts()

IV. Usage of Embeddings Layer in Keras :

➢ Quick reminder about word embeddings:

- Instead of the classic tokenization of the words by a random meaningless number , each word will be encoded by a vector of N-dimension , This vector will present the semantic of the word in a way of the similar words will have a close projection in the plan to that word (for example , words like "Sad" , "horrible" and "awful" will have similar vectors)

➢ Keras Embedding Layer :

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

- The role of the Embedding layer is to to give for the words appearing in the comments of the same labels: a close representations , so for words like " joyful" , "amazing" , "wonderful" : they all appeared in positiove comments labeled by 1, the Embedding layer detects that and do the necessary work to know that they have a similar semantics
- The first argument is the vocabulary_size : how much distinct words are tokenized
- The second argument is the dimension of the embedding vector of each word
- The input_length arguments specify the number of words in a sequence (which is the length of the maxsimum sequence after padding the shorten ones)
- The role of the Flatten Array is to pass from 2D array (embedding dim * input_length) to 1D flat vector with embedding dim*input_length elements , the image below shows the impact of Flatten layer

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, 120, 16)           160000
_________________________________________________________________
flatten_3 (Flatten)          (None, 1920)              0
_________________________________________________________________
dense_14 (Dense)             (None, 6)                 11526
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 7
=================================================================
Total params: 171,533
Trainable params: 171,533
Non-trainable params: 0
```

- The last layer has only one output with sigmoid as an activation function because we are dealing with a Binary classification problem (given a IMDB comment , guess if it's a negative or a positive comment)

➢ GlobalAveragePooling1D as an alternative to Flatten Layer:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_11 (Embedding)     (None, 120, 16)           160000
_________________________________________________________________
global_average_pooling1d_3 ( (None, 16)                0
_________________________________________________________________
dense_16 (Dense)             (None, 6)                 102
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 7
=================================================================
Total params: 160,109
Trainable params: 160,109
Non-trainable params: 0
```

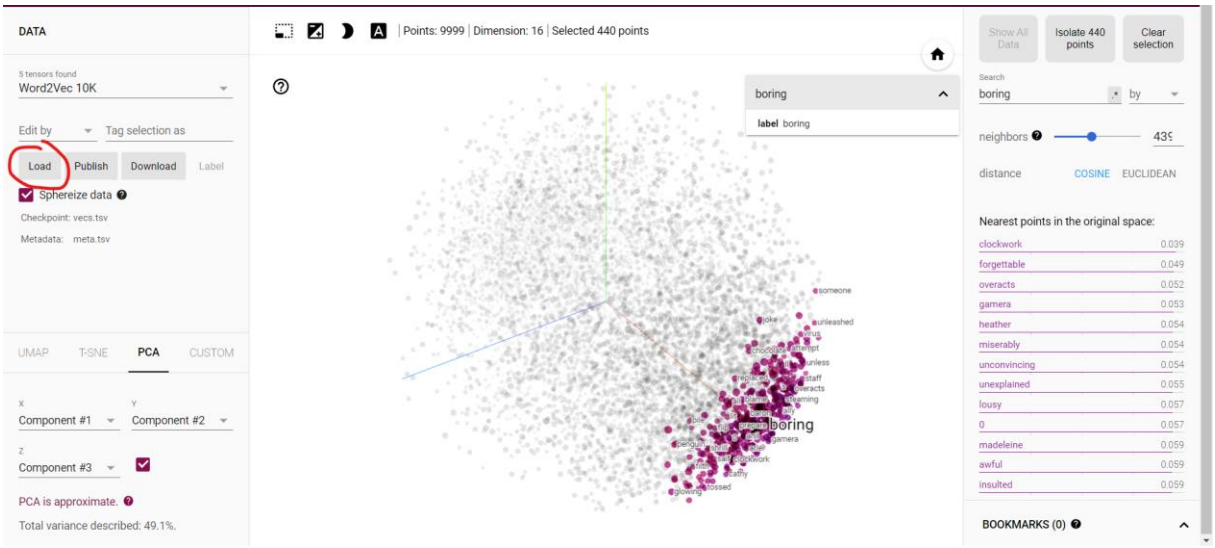
- Instead of just flattening the 2D array , GlobalAveragePooling1D() will return a 1D vector with a dimensions equals to the last dimension of the input (which is input_length=16)
- For each word , he will take the average of its embedding vector (or generally , doing the average based on the last axis : 16)
- The number of parameters has a significant decrease compared to the first model (which had the Flatten layer instead)

➢ To check the shape of the first Layer : Embedding Layer :

```
e = model.layers[0]
weights = e.get_weights()[0]
print(weights.shape) # shape: (vocab_size, embedding_dim)


(10000, 16)
```

V. Tensorflow Embedding projector

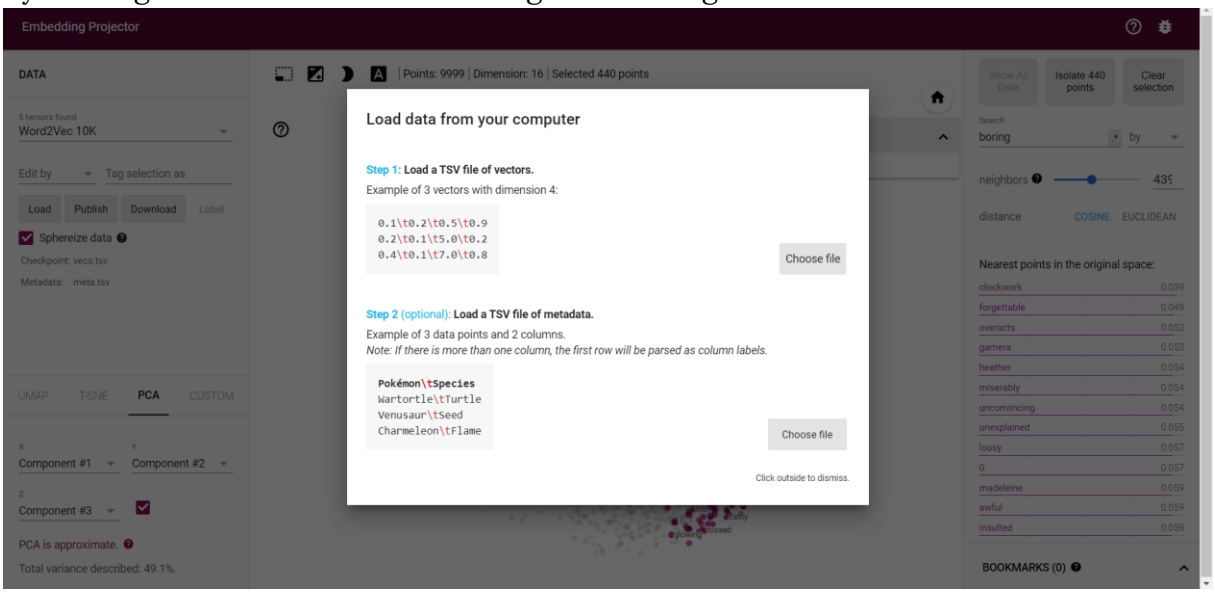
- TensorFlow had an interesting website : https://projector.tensorflow.org/ which provides you a 2D visualization of the generated word embeddings vector by the Embedding layer by loading meta and vector files I should generate from the Embedding Layer



- We can see our embedding layer give a close presentation to words “boring” , “forgettable” , “miserably” and “awful” which is pretty good !

➢ Loading vector.tsv and meta.tsv files in the website :

- By clicking in the load button I should get this dialog :



- From the dialog we see that the TSV file of vectors is a text file , each row represents a word embedding vector , its components are separated by /t
- TSV file of meta data contains is a text file , each row contains the vector name

➢ How to generate those two files in Python :

```
import io

# Open writeable files
out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
out_m = io.open('meta.tsv', 'w', encoding='utf-8')

# Initialize the loop. Start counting at `1` because `0` is just for the padding
for word_num in range(1, vocab_size):

  # Get the word associated at the current index
  word_name = reverse_word_index[word_num]

  # Get the embedding weights associated with the current index
  word_embedding = embedding_weights[word_num]

  # Write the word name
  out_m.write(word_name + "\n")

  # Write the word embedding
  out_v.write('\t'.join([str(x) for x in word_embedding]) + "\n")

# Close the files
out_v.close()
out_m.close()
```

- In Collab I should call the cell below to download the files locally

```
# Import files utilities in Colab
try:
  from google.colab import files
except ImportError:
  pass

# Download the files
else:
  files.download('vecs.tsv')
  files.download('meta.tsv')
```

VI. Prepackaged Datasets and the pre-tokenization

- Tensorflow has a prepackaged Datasets called “Tensorflow DataSets” : tfds

- This kind of prepackaged datasets usually the word are pre-tokenized , that means we will have already the tokenization of the sentences in the dataset in a particular variable :

```
import tensorflow_datasets as tfds

# Download the plain text default config
imdb_plaintext, info_plaintext = tfds.load("imdb_reviews", with_info=True, as_supervised=True)

# Download the subword encoded pretokenized dataset
imdb_subwords, info_subwords = tfds.load("imdb_reviews/subwords8k", with_info=True, as_supervised=True)
```

- In the code above we see that the path “imdb_reviews/subwords8k” contains the data pretokenized

- Subwords8K means that the technique used is Sub-Word Tokenization with vocab_size=8K=8000

VII. Padded_batch() an interesting method :

In tfds library, we have an interesting method called padded_batch() :

- Its role is to Batch and pad the datasets to the maximum length of the sequences

```
BUFFER_SIZE = 10000
BATCH_SIZE = 256

# Get the train and test splits
train_data, test_data = dataset['train'], dataset['test'],

# Shuffle the training data
train_dataset = train_data.shuffle(BUFFER_SIZE)

# Batch and pad the datasets to the maximum length of the sequences
train_dataset = train_dataset.padded_batch(BATCH_SIZE)
test_dataset = test_data.padded_batch(BATCH_SIZE)
```

  The more the BATCH_SIZE Is big , the more the training will be quicker (per epoch)

VIII. Sub-Word Tokenization:

- Subword-based tokenization is a solution between word and character-based tokenization.

- It's born to solve the character-based tokenization issues (very long sequences and meaningless tokens) and the word-based tokenization (a high risk to get <OOV> token due to the difficulty to cover all the vocabulary words)
- For binary classifiers, this might not have a big impact but you may have other applications that will benefit from avoiding OOV tokens when training the model (e.g. text generation). If you want the tokenizer above to not have OOVs, then the vocab_size will increase to more than 88k. This can slow down training and bloat the model size. The encoder also won't be robust when used on other datasets which may contain new words, thus resulting in OOVs again.

➢ Description of the technique used:

- It keeps the most-frequently used words, but it splits the rare words and the composed ones into smaller meaningful words.
    - As Example: we keep "boy" but we split "boys" into "boy" and "s" , This helps the model learn that the word "boys" is formed using the word "boy" with slightly different meanings but the same root word.
    - The word "tokenization" for example will be splitted into "token" and "ization"
- As we may guess , the length of the tokenized sentence will be longer than the original size of the sentence (because word = token or more than one token)
    - A concrete example :
        - Comparison about word and sub-word tokenization of the sentence **"TensorFlow, from basics to mastery**" :
            - Word-tokenization:

```
Tokenized string is [[1, 37, 1, 6, 1]]
The original string: ['<OOV> from <OOV> to <OOV>']
```

- We got <OOV> in the place of Tensorflow , basics and mastery because our Tokenizer didn't meet those words in the training set
            - Sub-Word Tokenziation :

```
Tokenized string is [6307, 2327, 4043, 2120, 2, 48, 4249, 4429, 7, 2652, 8050]
```

```
# Show token to subword mapping:
for ts in tokenized_string:
  print ('{} ----> {}'.format(ts, tokenizer_subwords.decode([ts])))
```

```
6307 ----> Ten
2327 ----> sor
4043 ----> Fl
2120 ----> ow
2 ----> ,
48 ----> from
4249 ----> basi
4429 ----> cs
7 ----> to
2652 ----> master
8050 ----> y
```

- Thanks to splitting the rare words : we can tokenize and decode the whole sentence without getting <OOV> tokens

➢ Conclusion about Sub-Word tokenization:

- We saw how subworld text encoding can be a robust technique to avoid out-of-vocabulary tokens. It can decode uncommon words it hasn't seen before even with a relatively small vocab size. Consequently, it results in longer token sequences when compared to full word tokenization

IX. Sequence models with TensorFlow for NLP:

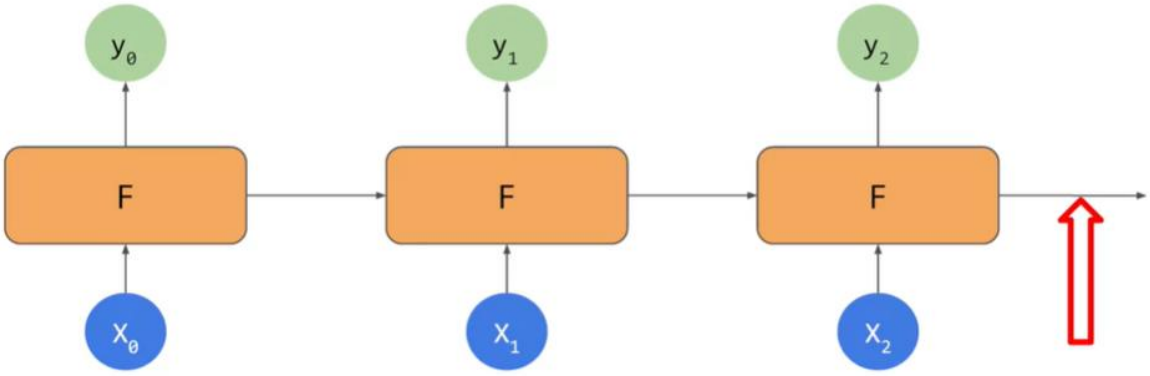
- Our previous process (tokenizing and padding the sequence) didn't take in consideration the relative order between the words while learning the rules or while generating the output of the model for example "the cat sit on the hat" and "the hat sit on the cat" two sentences with the same words but with 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:

```
1: My friends do like the movie but I don't. --> negative review
2: My friends don't like the movie but I do. --> positive review
```

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

X. Quick Illustration:

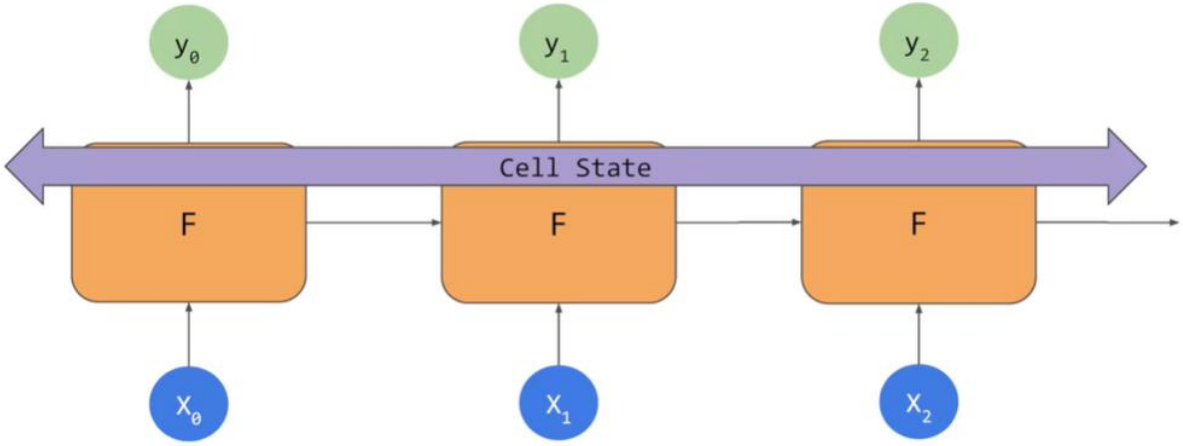
➢ RNN :



- 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>

XI. LSTMs in TensorFlow:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

- 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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 64)          523840
_________________________________________________________________
bidirectional_1 (Bidirection (None, 128)               66048
_________________________________________________________________
dense_4 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65
=================================================================
Total params: 598,209
Trainable params: 598,209
Non-trainable params: 0
```

- 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)

XII. Deep LSTMs in TensorFlow:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

- 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*

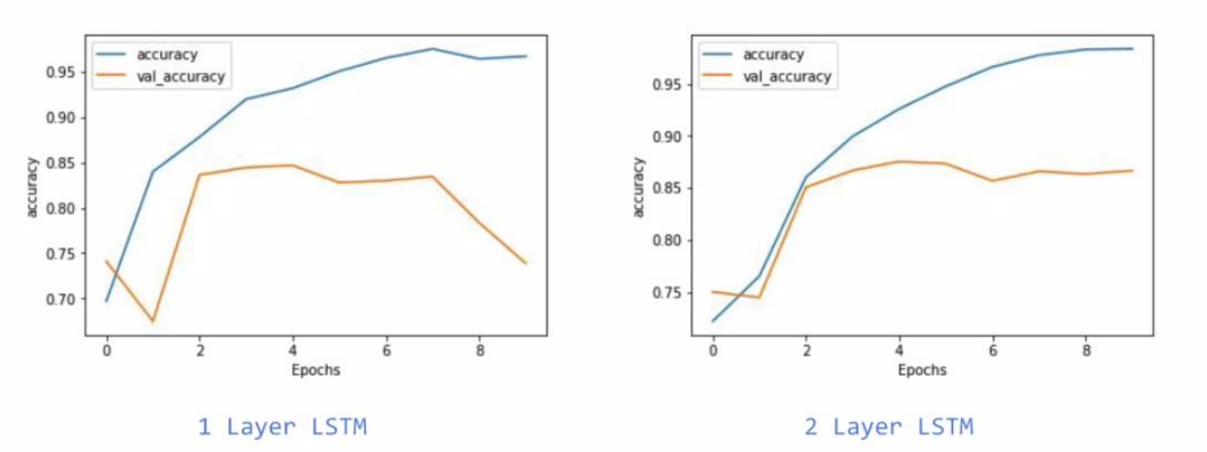
to this argument , we are going to assure that the output of the first LSTM layer match the input of the second LSTM layer

- Here is a comparison of the outputs shape with and without return_sequence=true

```
batch_size: 1
timesteps (sequence length): 20
features (embedding size): 16
lstm output units: 8
shape of input array: (1, 20, 16)
shape of lstm output(return_sequences=False): (1, 8)
shape of lstm output(return_sequences=True): (1, 20, 8)
```

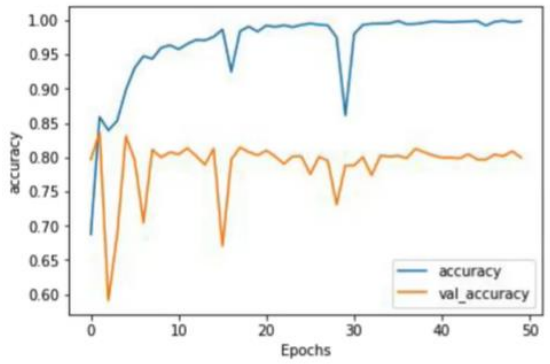
XIII. Comparing between the statistics of 1 Layer LSTM vs 2 Layers LSTM:

➢ <u>For 10 epochs :</u>

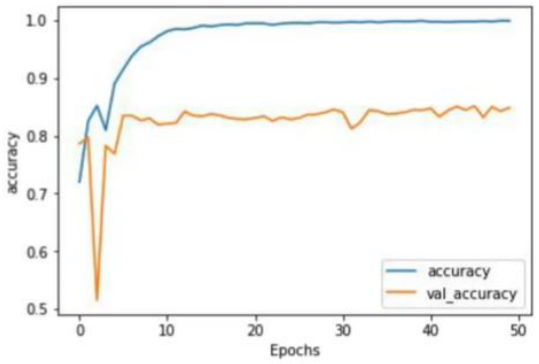


- For the training set , the 2 layer LSTM has a smoother graph (less of edges) which is a good signification that the model is improved
- For the validation set , we notice that there is a drop of val_accuracy for 1 LSTM Layer after some epochs which is a sign of an overfitting unlike the 2 Layer LSTM's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

➢ For 50 epochs :



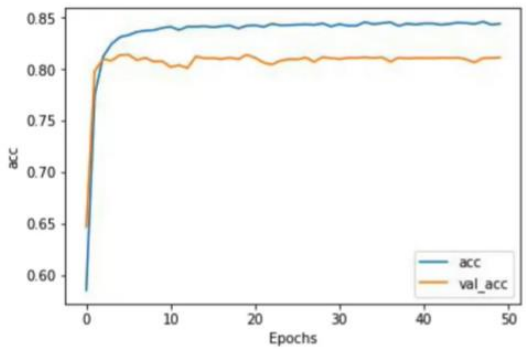
1 Layer LSTM



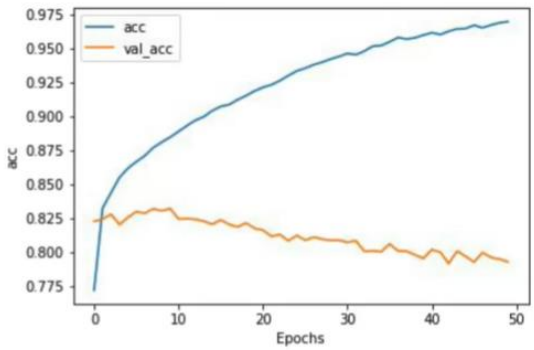
2 Layer LSTM

- 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

XIV. Comparing between the statistics of using LSTM vs not using it



Without LSTM



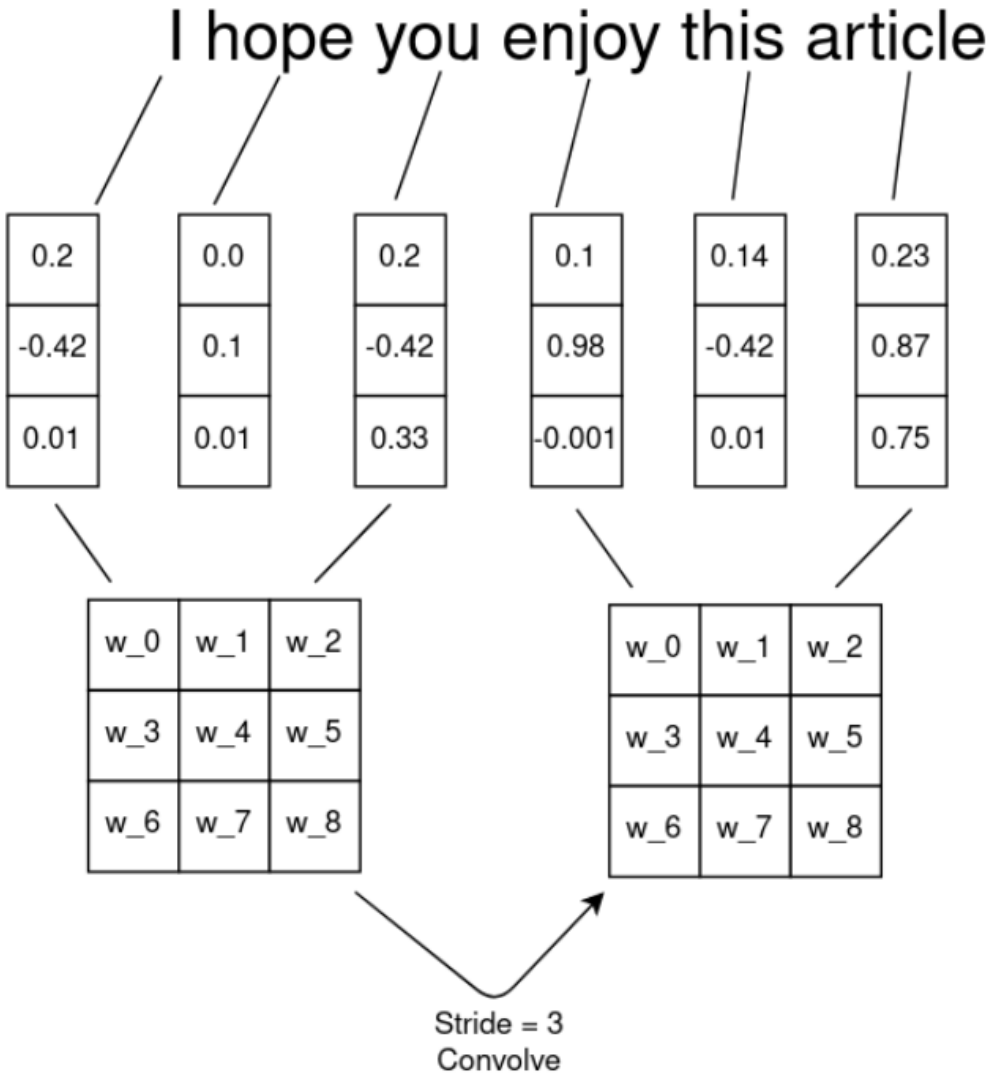
With LSTM

- 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

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

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              input_length=max_length),
    tf.keras.layers.Conv1D(128, 5, activation='relu'),
    tf.keras.layers.GlobalMaxPooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

- 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)

XVI. Using GRUs in TensorFlow:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Bidirectional(tf.keras.layers.GRU(32)),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

- 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

XVII. 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

XVIII. 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

XIX. 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:

```
# Define path to file containing the embeddings
GLOVE_FILE = './data/glove.6B.100d.txt'

# Initialize an empty embeddings index dictionary
GLOVE_EMBEDDINGS = {}

# Read file and fill GLOVE_EMBEDDINGS with its contents
with open(GLOVE_FILE) as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        GLOVE_EMBEDDINGS[word] = coefs
```

Now you have access to GloVe's pre-trained word vectors. Isn't that cool?

Let's take a look at the vector for the word **dog**:

```
test_word = 'dog'

test_vector = GLOVE_EMBEDDINGS[test_word]

print(f"Vector representation of word {test_word} looks like this:\n\n{test_vector}")
```

- Getting the words in my vocabulary using the embeddings:

```
# Initialize an empty numpy array with the appropriate size
EMBEDDINGS_MATRIX = np.zeros((VOCAB_SIZE+1, EMBEDDING_DIM))

# Iterate all of the words in the vocabulary and if the vector representation for
# each word exists within GloVe's representations, save it in the EMBEDDINGS_MATRIX array
for word, i in word_index.items():
    embedding_vector = GLOVE_EMBEDDINGS.get(word)
    if embedding_vector is not None:
        EMBEDDINGS_MATRIX[i] = embedding_vector
```

    - 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 :

```
model = tf.keras.Sequential([
   # This is how you need to set the Embedding layer when using pre-trained embeddings
   tf.keras.layers.Embedding(vocab_size+1, embedding_dim, input_length=maxlen, weights=[embeddings_matrix], trainable=False),
```

- 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

- The week is booked for Text generation and how we can build tensorflow models to generate texts

XX. Text generation principle

- Text generation isn't finally a very complex problem like it appeared at the first time if we look at it as a prediction task , where given as in input X : a sentence (a sequence of words) : try to predict the next word Y . It's a multi classification problem where we are gonna pick the word with the biggest softmax value.
    - So as example , if we already know that "Twinkle , Twinkle little Star" is a valid sentence , wee can consider "Twinkle , Twinkle little" as X and its next word "Star' as the expected output . In the same context we can consider "Twinkle , twinkle" as X and "Little" is its Y .

XXI. Preparing the training Dataset

D.Choosing the corpus of text to train on:

```
In the town of Athy one Jeremy Lanigan
Battered away til he hadnt a pound.
His father died and made him a man again
Left him a farm and ten acres of ground.

He gave a grand party for friends and relations
Who didnt forget him when come to the wall,
And if youll but listen Ill make your eyes glisten
Of the rows and the ructions of Lanigan's Ball.

Myself to be sure got free invitation,
For all the nice girls and boys I might ask,
And just in a minute both friends and relations
Were dancing round merry as bees round a cask.

Judy ODaly, that nice little milliner,
She tipped me a wink for to give her a call,
And I soon arrived with Peggy McGilligan
Just in time for Lanigans Ball.
```

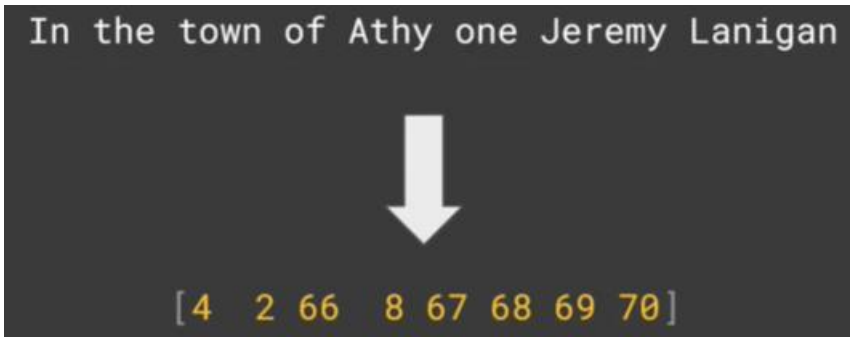
In the course example , it was an Irish song

E. Importing it and splitting to sentences to fit the tokenizer :

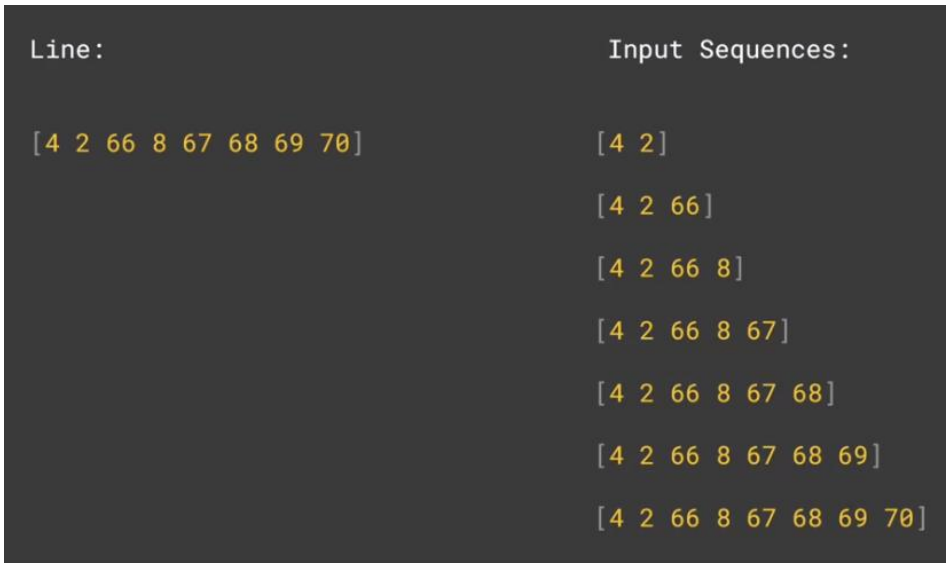
```
tokenizer = Tokenizer()

data="In the town of Athy one Jeremy Lanigan \n Battered away ... ..."
corpus = data.lower().split("\n")

tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1
```

- For the third line , we called lower method to make our training non—case sensitive and the we called split('/n') to have array each element is a sentence from this song
- In the last line , we added 1 to count the <OOV> token

F. Generating from each sentence many training data :



- Let's say for example this is the array of index words corresponding to the first sentence



- Our goal is to make "the" is the generated word from "In" and "town" is the generated word from "In the" , ….. , and "Lanigan" is the generated word from the input "In the town of Athy one Jeremy"
- First of all , we are going to create number of n-grams sequences, namely the first two words in the sentence or one sequence, then the first three are another sequence etc , and to do so : the code below makes this process possible :

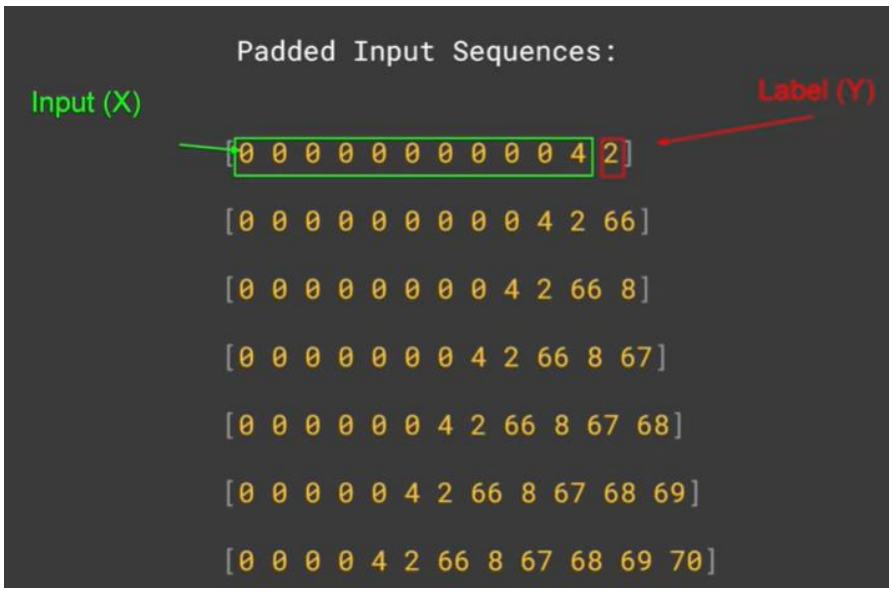
```
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
```

- For each sentence :
    - We are going to append to the training data : a sentence containing only the first two words , then the sentence containing only the first three words , …… , then finally the sentence containing the whole words

➢ <u>Padding of course is the next step , but it must be PRE</u>

```
Line:                                Padded Input Sequences:

[4 2 66 8 67 68 69 70]               [0 0 0 0 0 0 0 0 0 0 4 2]
                                     [0 0 0 0 0 0 0 0 0 4 2 66]
                                     [0 0 0 0 0 0 0 0 4 2 66 8]
                                     [0 0 0 0 0 0 0 4 2 66 8 67]
                                     [0 0 0 0 0 0 4 2 66 8 67 68]
                                     [0 0 0 0 0 4 2 66 8 67 68 69]
                                     [0 0 0 0 4 2 66 8 67 68 69 70]
```

- Of course , we need to do padding process to unify the input shape of our model , the max_length must be equal to the longest sentence length .
- the padding must be **<u>pre</u>** , it's important so it's easy to extract the output Y from each sentence which is the last item In each vector :



- <u>Extracting the Labels and the inputs from the padded sequences:</u>

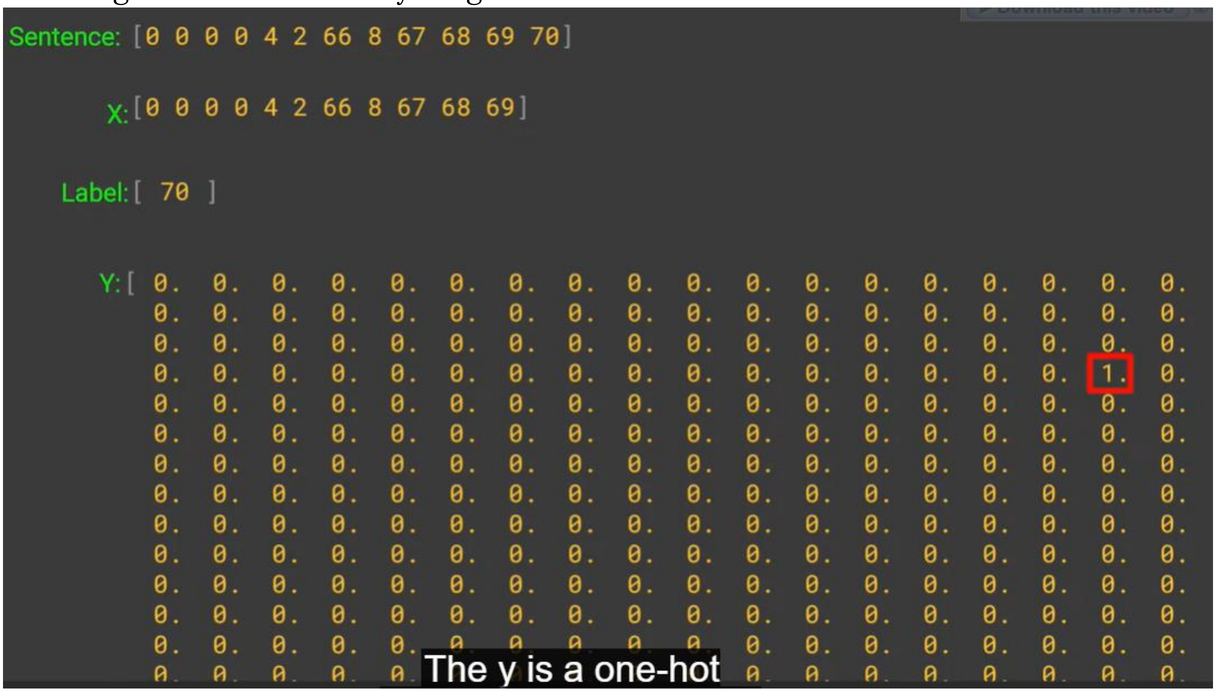
```
xs = input_sequences[:,:-1]
labels = input_sequences[:,-1]
```

- The code showed how can we slice the last column to get the labels

- <u>It's a classification problem , we should call to categorical on the labels !</u>

```
ys = tf.keras.utils.to_categorical(labels, num_classes=total_words)
```

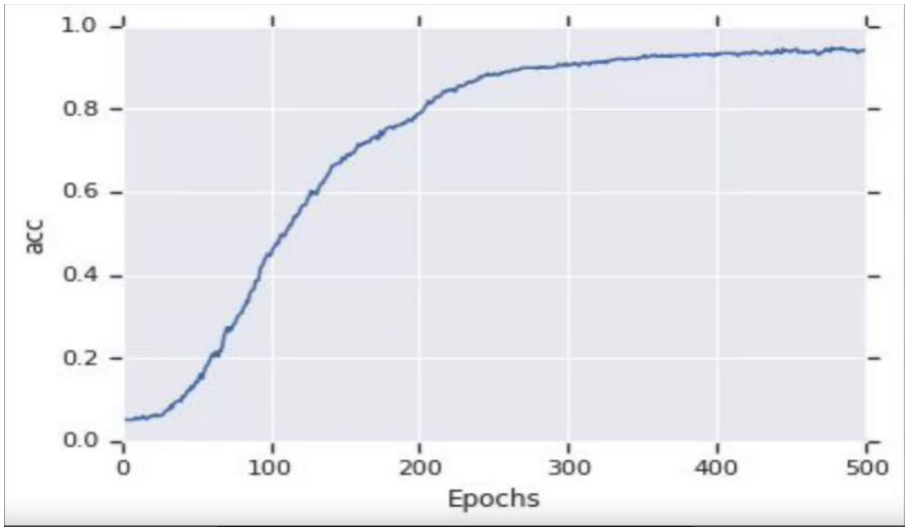
- The role of to_categorical function is to transform the index of the label (of the generated word) into one-hot vector its diamneison equals to the vocabulary size
- This is important , because in the last layer of our model we are gonna have a dense layer , each word must have its own unit which is gonna triggered for the most suitable word to generate : so the number of units in the last layer is equal to vocabulary size , with softmax function as activation function of course
- The image below makes everything clear :



➢ Finally : the model

```
model = Sequential()
model.add(Embedding(total_words, 64, input_length=max_sequence_len - 1))
model.add(Bidirectional(LSTM(20)))
model.add(Dense(total_words, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['
model.fit(xs, ys, epochs=500, verbose=1)
```

- Nothing surprising in the final model now
  - The embedding layer will have as a dimension vocab_size*64*padded_sequence_length
    - 64 is the embedding vector dimension , it's the only hyper parameter we can tweak here
  - The Bidirectional LSTM layer is the next layer
    - It's logic to use LSTM in these kind of problems to memorize the whole sentence context while generating the next word
    - It's important to be bidirectional to get the context from the previous and the next of the words to generate while training the model
  - The output layer is a dense layer with softmax as activation function since it's a multi classification problem
    - The number of units must be equal to the vocabulary sizes because it's the total number of our classes which defines which word to generate
  - The loss is of course 'categorical_crossentropy' dur to the nature of the problem : multi classification
  - The Adam optimizer seems the best optimizer of generating sequences tasks due to mathematical reasons
- An interesting thing to note is the huge number of epochs we specified : 500 and this is because this kind of models takes a lot of epochs to converge , the graph below show the progression of the training accuracy during the 500 epochs :



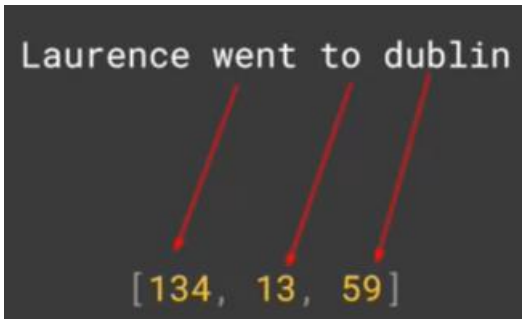
G. Some examples of the generating sequences :

- We specified here “Laurence went to Dublin” as the start of our sequences (we called it seed_text) and we told the model to generate the next 10 words

```
Laurence went to dublin round the plenty as red wall me for wall wall
Laurence went to dublin odaly of the nice of lanigans ball ball ball hall
Laurence went to dublin he hadnt a minute both relations hall new relations youd
```

H. How is that possible ? generating word by word

➢ Tokenize the seed text :



- Laurence hasn’t a corresponding index because it’s unknown word

➢ The pre-padding :

```
token_list = pad_sequences([token_list], maxlen=max_sequence_len - 1, padding='pre')
```

➢ Predicting the next word:

```
predicted = model.predict(token_list)
predicted = np.argmax(probabilities, axis= - 1)[0]
```

- We call the predict method in the seed_tokenized_pre-papadded sequence
- We took the word with the biggest softmax probability : it’s the next generated word !

➢ Adding the generated word to the seed text :

```
output_word = tokenizer.index_word[predicted]
seed_text += " " + output_word
```

- Repeating the whole process in a loop with number of iterations equals to the desired final length :

```
seed_text = "Laurence went to dublin"
next_words = 10

for _ in range(next_words):
  token_list = tokenizer.texts_to_sequences([seed_text])[0]
  token_list = pad_sequences([token_list], maxlen=max_sequence_len - 1, padding='pre')
  predicted = model.predict_classes(token_list, verbose=0)
  output_word = output_word = tokenizer.index_word[predicted]
  seed_text += " " + output_word
print(seed_text)
```

XXII. The generating sequence tasks aren't really so accurate:

```
Laurence went to dublin round a cask cask cask cask cask
squeezed forget tea twas make eyes glisten mchugh mchugh
lanigan lanigan glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
glisten glisten glisten glisten glisten glisten glisten
```

- Here is an example of a generated sequence by the model
- The reason of that is because the more we generated words after the base sentence : the more we would have less certainty about the correctness of the generated word