Chapter 1: Part 3

**Seq2Seq Model**

**1.3.1 Limitations of Bag of words**

We talked about *Non-DL-Bag of words* using only logistic regression and *DL-Bag of words* using NN. *DL-Bag of words* is better than *Non-DL-Bag of words*. However our output was only Yes/No. But we have following issues that make it undesirable for a ChatBot Project.

1. Fixed sized input: We have to use same level of input for the training. For both NN and Logistic type model.
2. No word order: We're just counting the occurrence of each word rather than looking at their order in the sentence.

* Eg: For the Meaningless sentence made up with bunch of random words, we can get the same representation on the Bag-of-word.
* ***Is the cat shorter than the dog***. The answer might be Yes.
* If you just change the cat and the dog has two words around and becomes: ***Is the dog shorter than the cat***. The model might return Yes, but the real answer is No.

Exactly same words with opposite meaning. But the *model* is going to get you the *same answer*.

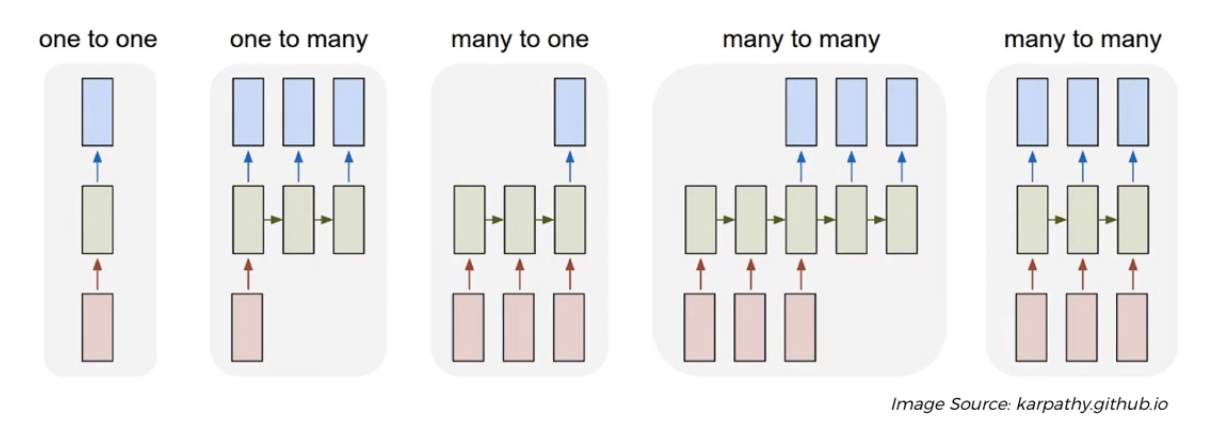
* Basically we're *losing information* through the *bag of words model* just the way it's constructed because we're not taking into account the position of the words.

1. Fixed size output: we already talked about fixed size output in this case is just one word Yes/No.

* However, we may have multiple words but it won't be variable.
* We might have five options or something like that, but it will be fixed.

**1.3.2 Using RNNs**

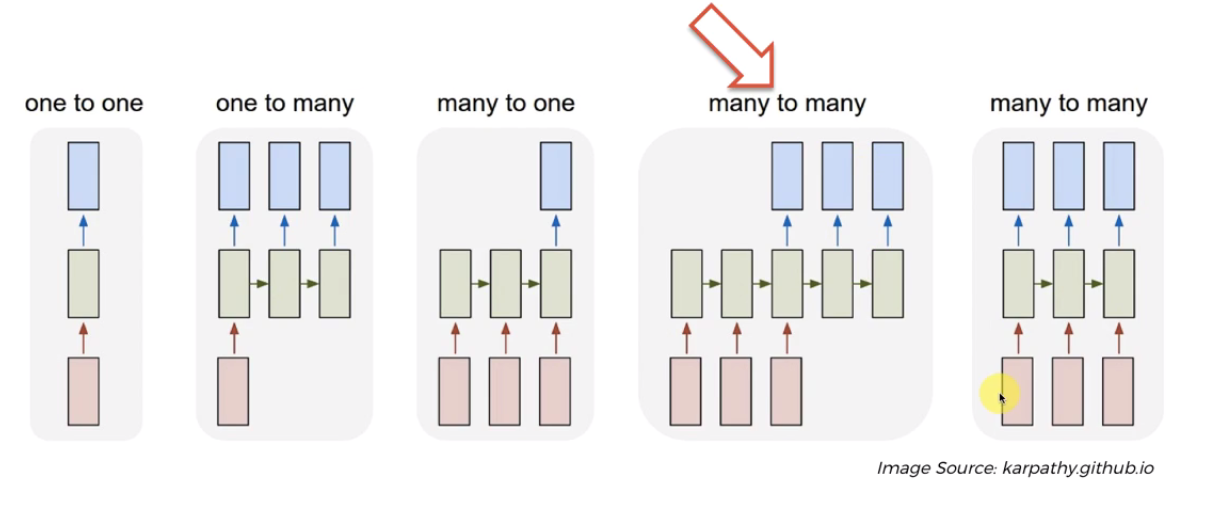
* Using RNNs: To overcome all these limitations, we use RNNs.



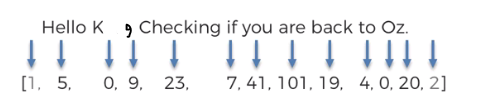
* Above image is from Andrey Karpovsky's github page. These are the different types of RNN architectures.
* Every single box is a whole *layer* of *neurons*.
* And we're looking at *unravel time a loop* of the *RNN*.
* The *green layer* is the NN *feeding* into *itself* through *time*,

1. One to one: One *fixed* input through NN gets one *fixed* output.
2. One to many: One *fixed* input through NN gets many *fixed* output.
3. Many to one: Many *fixed* input through NN gets one *fixed* output.
4. Many to Many (variable): Many *variable* input through NN gets many *variable* output.
5. Many to Many (fixed): Many *fixed* input through NN gets many *fixed* output.

* We'll focus on Many to Many (variable) type RNN because our input will vary and also output varies. We want a variable sized input and a variable sized output.



* So if somebody asks us a question or five words then we will have 3,4 or 6 words output.
* We don't need ***20000*** element *long vector* which we use for the *bag of words* (we now use it differently). We can now work with *any* *size sentences*.
* We're going to take the *each single word* including the *comma* and *full stop* and we're going to code them.



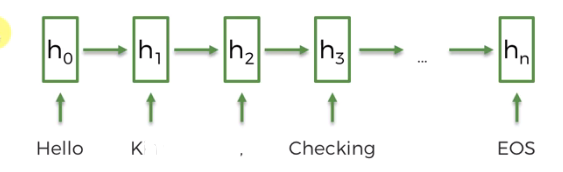
* We take those position from our ***vector*** of ***20000*** ***words***, but our input vector is just those words in our sentence, with order maintained. So we don't use ***20000 long input vector***.
* We are creating new *input vector* (and its length varies) using the *words position* from our *20,000 words vector*.
* ***0*** is ***non-English*** words. ***1*** is ***SOS*** and ***2*** is ***EOS***.
* If we have *multiple* *words/symbols*, we use the corresponding *codes/position\_no* *multiple* *times*.

And so as you can see the *size/length* of this *input-vector* will *depend* on how many words we have in our *email/text*.

**1.3.3 Seq2Seq Architecture**

Now it is possible to encode your text in the version of like a vector. But we're not going to work with the vector.

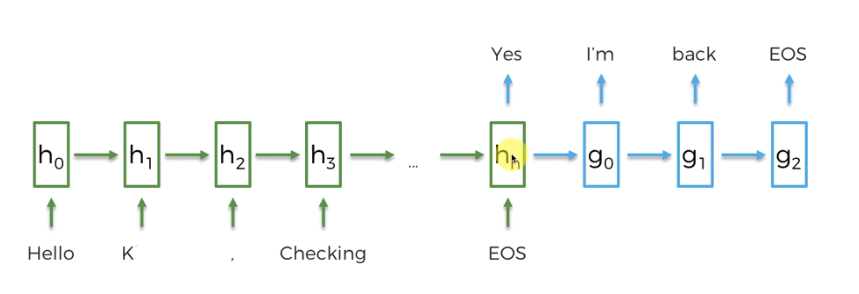
* We're going to work with the text. So we're going to create our NN working with the text. Just keep in mind that in a machine sense we do need to work numbers and it is possible to construct that vector of numbers from our email. The vector of numbers will be about the same size as the email itself.



* You can see we droped the ***SOS***. Because it's not that important. Every sentence will always start with ***SOS*** so it won't make any difference. ***EOS*** is important because it will dictate when the output will terminate.
* Basically what we're doing is we're feeding these words one by one into our RNN. This classic RNN feeding into itself through *time*.
* i.e. at *first time step* we feed the layer *new information* then we get some *output*.
* In the *second step* we feed the layer *new information* + *previous\_step\_output*, then we get 2nd steps output.
* In the *third step* we feed the layer *new information* + *previous\_step\_output*, then we get 3rd steps output.
* This continues until EOS occurs.
* Each word goes one by one into our first layer of RNN.
* Eg: "Hello" goes to the RNN first, then RNN gives an output and then it feeds itself the second word "K" and again gives some output and this "feeding & output" process continues until "EOS" occurs.
* It is just a classic RNN but will use LSTM.
* So it's kind of is feeding into itself the previous steps output + we are putting in new information into a memory cell.
* Output: When EOS occurs, after that we are starting the outputs. As we can see from "many-many" architecture, the output begins when the input stops.

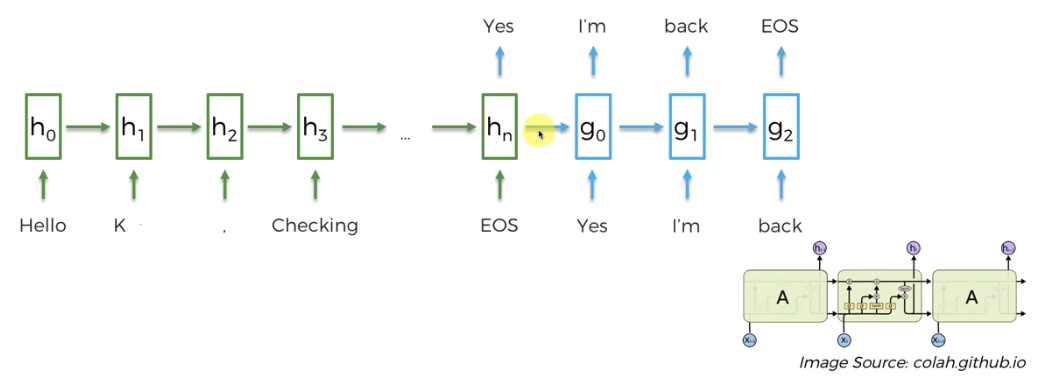
|  |  |
| --- | --- |
|  |  |

* Then the RNN gives the output one by one. EG: "i", "am", "back". The process is similar:
* Layer gives the first output, then it feeds itself the output in next step,
* Then it gives the 2nd output and so on.
* So basically it gives the output and simultaneously it feeds itself.



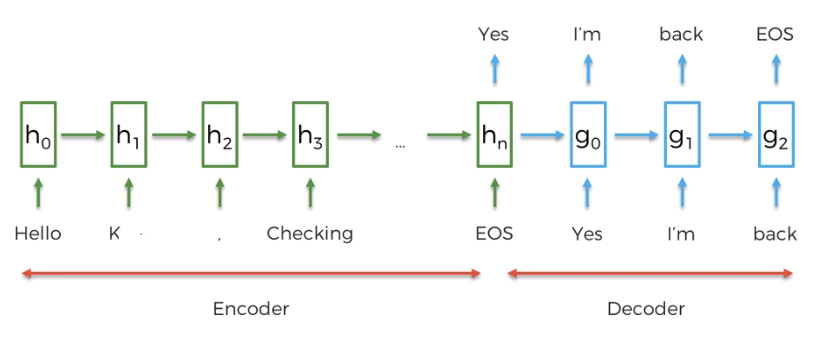
* How to determine the EOS for output: The idea is simple:
* Here the encoding is like a parameter to the model and it kind of determine the length of the output.
* In the *output*, *predictions* happen. For 20k bag-of-word, there are *20k possible output*.
* First the model assigns a probability for *each* of the *20k words*. Then it *picks* the *words* having the *high probability*.
* It picks the *first word* having *highest* *probability*. Then *feed* itself the *output*
* Next it picks the *2nd word* having *highest probability* (taking account of *first output word*), *feed* itself the *output*
* If it find *EOS* the *highest probability* in account of *all output words*, it ends the sentence.

So its all up to the NN to choose the amount of words and EOS. It learns from its training. (in exceptional case we can limit the output).

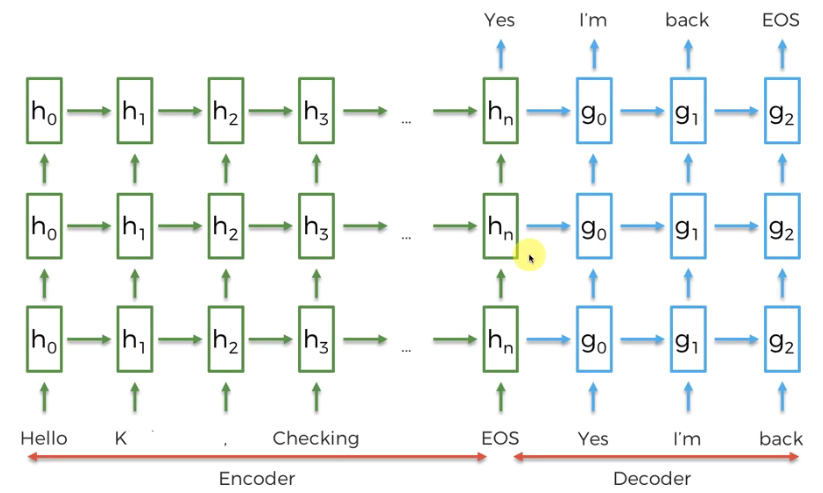


|  |
| --- |
| * The *horizontal arrow* indicates the *memory cell/pipeline*. And we are actually using an *LSTM* over here. * LSTM make sure that we're taking into account the output that we generate before and that will help us help the network preserve understand the meaning. * It can create better sentences because it remembers not only just the "overall memory that's been passed through the network that's updated through the cells" but also remembers the "immediately preceding word" and helps it formulate better sentences. |

* Encoder & Decoder:
* EOS cell: In encoding part we've got our cells feeding information to create "meaningful vector" for us.
* When we reach at , where the EOS occurs, the meaning of the sentences basically contained in this cell. And from this cell we're starting to extract the answer.

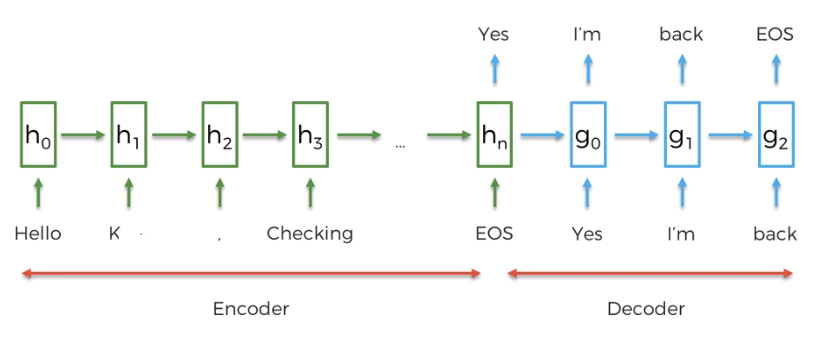


* That's why the *first part* before EOS (including EOS cell) called "*Encoder*" and *rest* of the *next-cells from EOS* called "*Deocoder*".
* All this information that we have from the start we're *encoding* *it* into this one cell and then we are *decoding* *it* to providing a response.
* It's got a first part that *encodes* the *meaning*. Then it's got a second part that *decodes* the *answer* from the *encoded meaning*.
* In reality you can actually make this NN very deep using multiple layer of NN-layers.



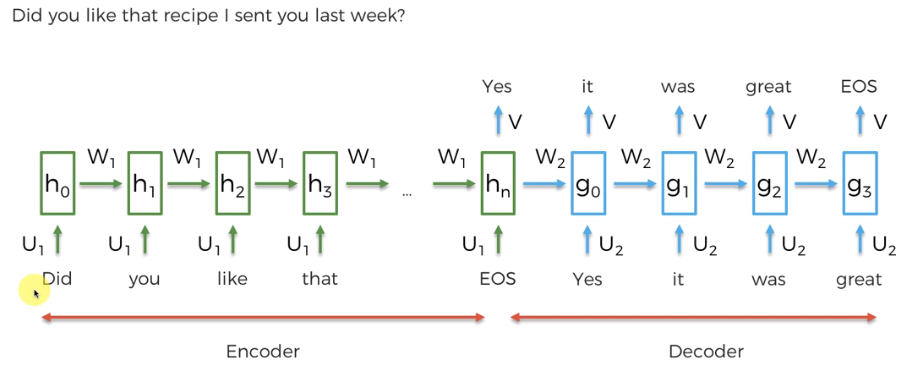
**1.3.4 Seq2Seq training**

Now we're going to look at how the *Seq2Seq* models are *trained*. Following is our architecture, with *encoder* & *decoder* which resembled to *RNN*. In *encoding* part our inputs are fed in.



And then once we have our *inputs* then the *decoder* starts *decoding* the values (output) and we *fed* those output values back into the *decoder*.

* Training: As we discussed, the above section is actually a *trained* Deep-NLP model. Where the *output/response* are *unknown*. We get the *response* applying the *trained model*.
* Email & Reply dataset: For the *training*, the things are little bit different. For training we're going to look at a different set of emails, those provides the *email-itself* and the *corresponding response*. That is our train data will be the data-set of Email & Reply.
* Here we work with *real reply* of the *email*. When we got an *output* from the *model* for an *email*, we are going to *compare* it with our *real reply*.



* How it trains: For example let the email was ***"Did you like the recipe I sent you last week?"*** (12 inputs including EOS) and the actual response for this case is ***"Yes it was great"***.
* So in the *training part* we actually know the *end result* and now that will allow us to *train* the *model*.
* Here we've got 's , 's for the "Encoder" and 's , 's and 's for the "Decoder". (In different literature you might find different notations).
* Those represent the parameters of our network such as ***weights*** of the ***NN*** that we need to update through training.

***There are two main points that we're going to look at.***

* The main goal for the neural network is to make sure that *it* can *model* the *logic* and *behavior* that was *observed* in the *emails* and corresponding email-responses.
* The model looks at an email, it understands the meaning behind it and then it creates a response and that should be in line with all the training it has.
* What is the network is going to learn: When the input is done, the NN is going through an iterative process to look at the different options that it can spit out.
* It knows that the correct first word is "Yes" form the given real-email-response. The NN has 20000 different options to pick from 20000 different words.
* So through the process of back propagation this Seq2Seq model is going to *update* its *weight* in such a way to make sure that the *probability* of getting the word "Yes" is the highest (from 20000 words).
* Next it proceed to the next word "it" from the given real-email-response. Then it going to *update* its *weight* (through the process of *Stochastic Gradient Decent* and *Back Propagation*) in such a way that the *probability* of getting the word "it" is the highest (from 20000 words) given that the first output ward is "Yes" (conditional).
* The *error* is going to be *back-propagate* and *weight* is going to *update itself* in order to *maximize* the *score* that is assigned to the word it in this case out of 20000 words.
* Once that is done it move on to the *next word* of *real-reply* and will do the same thing for the *third* place.
* Given that we have *first* word is "yes", the *second* one is "it". It will maximize the score that is assigned to the word "was". So again backwardation and so on.
* In practice, these kind of *email data* for *training* is lot more. And we train the model for 1000 of epochs.
* In reality these are all *layers* of *neurons* and it can actually have even more *levels*. It could be a *deeper network* as we saw before.
* How it works with variable I/O: that we need to observe here is that the *weight*. So a natural question is, in this example, we've *trained* up this network for all *12 inputs*, the output is 5. Total 17 steps.
* What if we have *20 steps* and then were and the *response* that the net generates is going to be like *3 words*? So how is it?
* How can it explore options like a different emails that have different lengths that it haven't anything seen before?
* The answer is YES.
* And the reason for that is as you can see the weights 's , 's for the "Encoder" and 's , 's and 's for the "Decoder", they are actually the *same* for the *layers*.
* So as you *add/remove* more *words* or more *sentence*, to get *variable output*, the *number* of *weights* are *same* to be trained.
* Those are going to be the foundations of this Seq2Seq model. These *weights* are going to be *trained* through the normal process of Back-propagation, SGD
* So however long the sentences are *we're training* the *same* thing, same *weights*.