

DIFFERENT MODELS UNDER NLP ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

RNN

Bi directional RNN

LSTM

Encoder – Decoder (transfrormers)

BERT

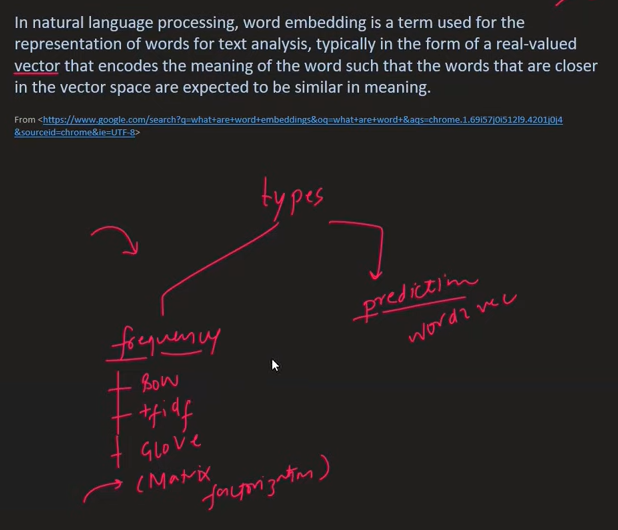
LIBRARIES USED ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

**NLP PIPELINE::::::**

**1. Text Preprocessing**

**2. Text Vectorization /or/ Feature Extraction /or/ Text Representation using Word Embeddings :** Used to convert text/words to vectors or numerical representation.

What are Word Embeddings ??



Many ways to do this are:

1. Bag of Words – uses Count Vectorizer.

Here , i) Order of occurrence of words don’t matter/ or/ are Not taken into account.

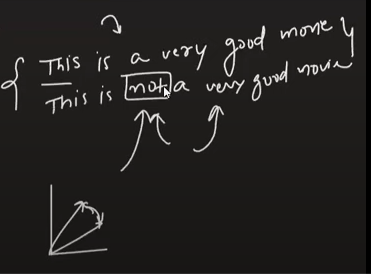
ii) Context of the word/Semantic meaning of word is NOT captured. (however, some researchers have shown that a little bit of semantic meaning is captured by BOW technique)

iii) If there is a new word added in test phase, which was NOT present in the input test Corpus during training, then the count vectorizer ignores that word.

iv) Sparse matrix problem exists . This leads to overfitting.

v) Ordering is Not used: If 2 sentences have similar words, then BOW considers them similar, which is NOT always true.

Example:



The above 2 sentences have completely different sentiments, however the BOW would represent them with vectors which are very close to each other. THE SOLUTION TO THIS IS IN USE OF “ENGRAMS”.

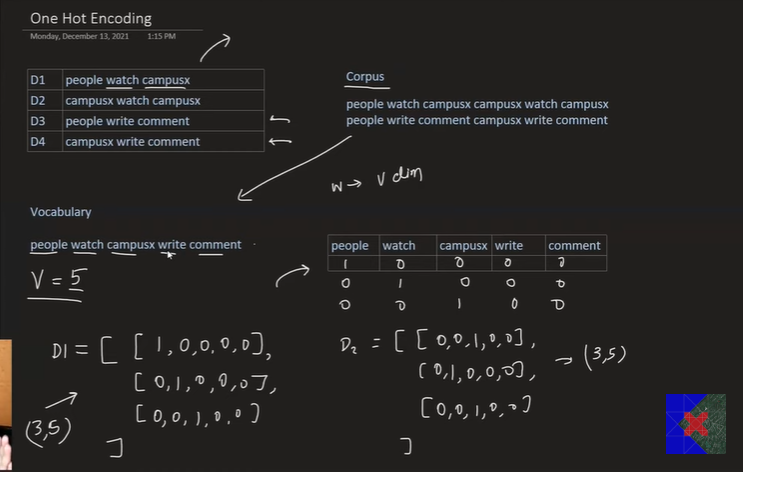
1. Tf-Idf - calculates score (see Semantic Segmentaion Word Doc.).

**Tf** shows how commonly does a word occur in one document. And **Idf** shows how rarely does a that word occur in all of the documents.



Oov = out of vocabulary . i.e. new words introduced during testing are ignored while creating the matrix.

1. One-Hot Encoding (NOT USED B/C OF SPARSE ARRAY): consider 4 documents / 4 sentences with 5 unique words ( hence vocabulary = v =5). Here, we have done OHE on 1st and 2nd sentences only.( output D1 and D2)

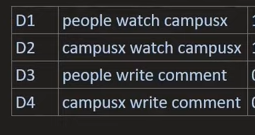


Sparse Array:-----Array with many many 0s as compared to 1s. Thus as size of document grows, the size of sparse arrays also grows. This leads to Over-fitting.

Also, OHE will throw an ERROR if a new word is present in testing Corpus. B/c of size of matrix mismatch in the vocabulary and the OHE is incapable of ignoring the new word (like BOW ignores) .

1. Engrams

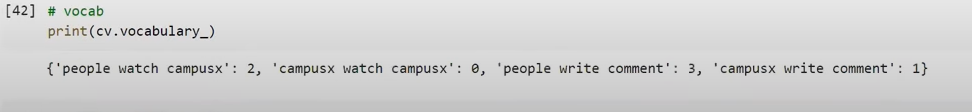
Given, the Corpus of 4 Documents/4sentences .



Case 1: Bag of Bi-grams



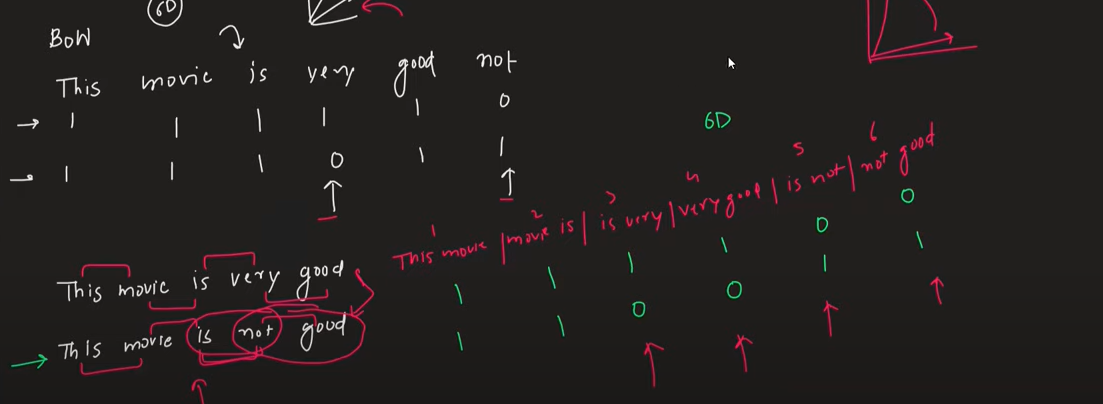
Case 2: Bag of Tri-gram . Vocabulary would contain the set of 3 words as shown below



BOW v/s ngrams(n=2) : Why ngrams is able to better capture semantic meaning/ context of the words

Example shows 2 sentences /or/ 2 Documents which have completely different context/semantic meaning.

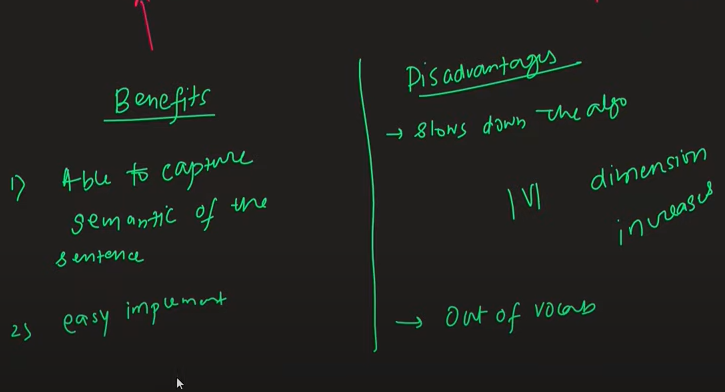
We solve this using BOW and BOngrams/ngrams (n=2) approach as shown below.



Observe: that:

1. using BOW approach, we get 2 vectors in 6D, out of which 4D are same .
2. While using ngrams approach, we get 2 vectors in 6D, out of which only 2D are same and rest 4 D are different. THUS, n=2 grams was better able to differentiate the semantic meaning of the two sentences and hence we say that ngrams is able to capture the ORDERING of the words (which BOW does NOT do).

Case 3: Bag of Quad- gram ----Here NOT POSSIBLE for the given dataset, since each sentence has a maximum of only 3 words .



Exp. for Disadvantages: ----

1. If we talk about computation time and complexity, then, n=3grams > n=2grams > n=1grams . b/c the size of the vocabulary ( which decides shape of vectors) will always be higher for n=3 ,than for n=2 and for n=1. (see example above).
2. It also ignores the new word in the corpus, if it was NOT present during training. This leads to loss of information or loss in correctly understanding the context.
3. Word2vec ---- This is the only one among these that is based on deep learning. Rest of the above do NOT have any learnable parameters in them.

Advantages:

1. It(and GLoVe) are the only ones techniques that fully captures the Semantic meaning of the words.

Example: ‘Cheerful’ and ‘Joy’ have similar semantic meaning, hence they will be represented closely in word embeddings and therefore have Vectors with close/similar values . On the other hand, BOW or TfIdf will have completely different vector notations for these words.

1. It produces a matrix/array of much smaller dimensions than the other techniques like BOW does. This increases the computation speed drastically.
2. It produces a dense matrix/array compared than the other techniques which produce sparse matrix. (Note: Sparse matrix/vectors are ones which have a lot of 0s in them, which increases their dimensions and also leads to overfitting of data)

**Method 1**: **use Word2vec with Pretrained Weights:** -- example: Pretrained Word2vec by Google called “**GoogleNews-vectors-negative300”.**

---It is pretrained on 3 billion words by google engineers on Google News corpus. It has a 1D vector representation of size (300,) for each words.

**Method 2: Self-Trained**

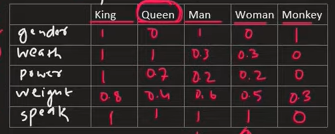
**INTITUITION BEHIND WORD2VEC:**

* Word2vec creates features( or finds/learns hidden features) and assign values to each token in the corpus corresponding to these features .

**Example**: In **GoogleNews-vectors-negative300 pretrained Model,** the 300 weights/values corresponding to each word (in the 3million word corpus), is nothing but the values of 300 hidden features per word, which the NN learnt using Backpropogation.

* These features are nameless i.e. they may represent anything humanly meaningful or something bizarre with no human significance. In any case it DOES NOT matter.
* Explanation: In Below figure, features are the 1st column i.e. Gender, wealth etc. which are the learnable parameters/features and the 1st Row are the Vocabulary of the trainable Corpus.

Thus here, after the word2vec N.N. does the training based on the text corpus (here in below we have only 5 words as vocabulary), the NN ends up learning the values of 5 hidden features (denoted in 1st column, but obviously we decide the no. of hidden features that NN should find b/c no of hidden features = no of neuron of hidden layer in NN, which obviously we give when we design NN’s architecture), but the values of these hidden features corresponding to each word in vocabulary is learnt using Back Propagation. The vectors used to represent each word is denoted in each column



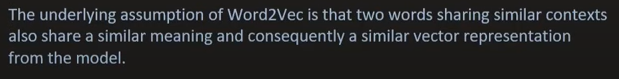
After learning, the vector representation will look something like:

King = [ 1,1,1,0.8,1]

Monkey = [1,0,0,0.3,0] ………&so on . Thus, even though we may have a Corpus with 8 million words, we can **choose** to get a 1D vector representation of just a size say 200 or 210 or 300 or 500 or 1000 or……by choosing the the no. of neurons in Hidden layer.

NOTE: In above example, we have just assigned random names to features like gender, wealth etc. BUT, in reality the word2vec NN does NOT tell us what are the names of the features or do they have any meaning at all. They look more like f1,f2,f3,f4 and f5 .

* Word2vec is very good at capturing the context of the words .



Example: a) A football player took a shot ; b) A Hockey player took a shot.

c) A Cricket player took a catch.

Here, since football and hockey player are used in similar context, they will have similar vector representation(or close vectors) , but Cricket player will Not be similar to them.

2 Types of word2vec: CBOW and Skip-gram

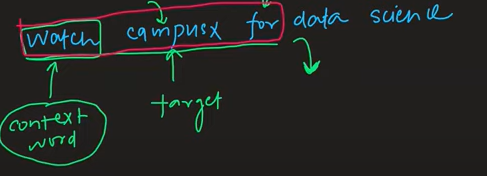
These are very similar to each other b/c they are both Shallow NN architectures. But their NN architectures are kind of reverse to each other.

1. **CBOW :**

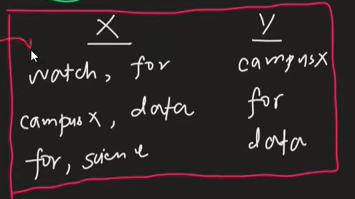
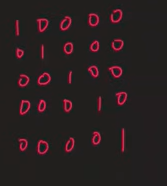
Consider a sentence “watch campusx for Data Science” and we want to convert each word into a vector representation. But, we cant just run a NN on this .

* First, we decide the vector size = no of hidden features for each word , say 3 .
* Then, for training the NN, we pick a windows size of say 3 words(or even 5 words) and assign the center word as the “target word” and the left and right words as “context words” .

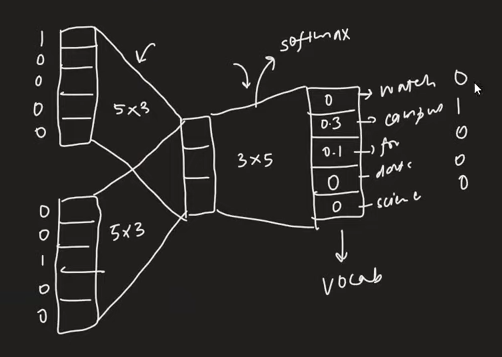
example below: - campus = = target word & watch, for == context word



* Then we slide the window to right and a new set of target and context words. & so on. We get the following

* Now we create the OHE vectors for each word in vocabulary of corpus. (see above)
* Now as discussed, both CBOW and skip-grams are fully connected shallow NN. So we create such an architecture and feed the “input” words = “context words”(in below fig they are ‘watch’ and ‘for’) as OHE vectors and get the prediction vector .
* Then compare this predicted vector(below it is [0,0.3,0.1,0,0] for campusx ) to Actual OHE vector of the “target words” (which is [0,1,0,0,0] for campusx after OHE was done above). campusx is target word for ‘watch and for’).



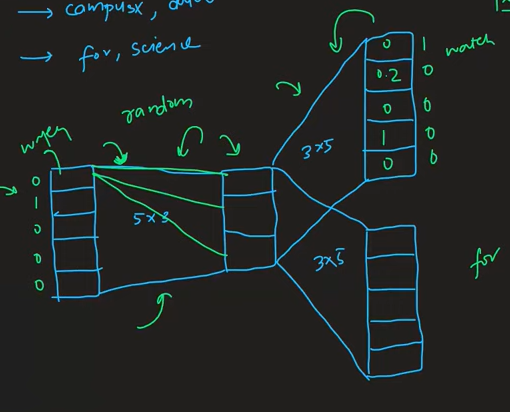
In above fig: the left most 2 columns of input layers with 5 values each is used to pass the 2 context words in OHE format. The center most “hidden layer” layer has 3 values b/c we decided that we want the size of output vector for each word = (3, ). This layer will be the output vector for each word after training is done. The outermost layer is Softmax Layer with 5 output values as we have output vector for each word as a (5,) OHE vector. Thus, in order to calculate loss and carry out Backpropogation, we need to convert the (3,) vector from hidden layer to (5,) size.

NOTE : 5X3,3X5,…etc denote learnable parameters.

* Then we get the “loss” b/w the actual vector and predicted vector and use Backpropogation to arrive at the correct values for the Hidden Layer, which is same as the Output Vector or vectorial representation of each word
* THUS, vectorial representation of each word is entirely dependent on the no. of neurons in hidden layer. In prev. example, we could easily have chosen 5 neurons in hidden layer to get a vector representation of each word of size (5, ) instead of (3, ).

1. **Skip-gram**

* Just note that its Architecture is Opposite to that of CBOW.
* **Difference**: CBOW tries to predict a word on the basis of its neighbors, while Skip Gram tries to predict the neighbors of a word.
* It is also just a Shallow fully connected NN.



**WHEN TO USE CBOW & WHEN TO USE SKIP-GRAM ???**

* For Small Data Set ---- Use CBOW
* For Large Data Set ---- Use Skip-Gram

Based on Research and this is proven.

**HOW TO IMPROVE ACCURACY/PERFORMANCE OF WORD2VEC MODELS ??**

* Increase the training size
* Increase the dimensions of vectors i.e. increase the neurons in hidden layer
* Increase window size i.e in above example, we took a window size of 3 words , but we could also have gone for a window size of 5 words with 1 target word and 4 context words (2 on each side) . However, this leads to increase in Training time and computation time.

**Gensim Library :**

1. **models.keyedvectors:**

Since trained word vectors are independent from the way they were trained ([**Word2Vec**](https://radimrehurek.com/gensim/models/word2vec.html#gensim.models.word2vec.Word2Vec), [**FastText**](https://radimrehurek.com/gensim/models/fasttext.html#gensim.models.fasttext.FastText) etc), they can be represented by a standalone structure, as implemented in this module.

The structure is called “KeyedVectors” and is essentially a mapping between keys and vectors. Each vector is identified by its lookup key, most often a short string token, so this is usually a mapping between {str => 1D numpy array}.

**The key is simply a word (so the mapping maps words to 1D vectors).**

**Why use KeyedVectors instead of a full model?**

The main difference is that KeyedVectors do not support further training. On the other hand, by shedding the internal data structures necessary for training, KeyedVectors offer a smaller RAM footprint and a simpler interface.

CODE::::How to get a KeyedVectors for full model : Train a full model, then access its model.wv property, which holds the standalone keyed vectors. For example, using the Word2Vec algorithm to train the vectors

**>>> from** **gensim.test.utils** **import** lee\_corpus\_list

**>>> from** **gensim.models** **import** Word2Vec

>>>

**>>>** model = Word2Vec(lee\_corpus\_list, vector\_size=24, epochs=100)

**>>>** word\_vectors = model.wv # now, word\_vectors has only keyedVectors

The reason for separating the trained vectors into KeyedVectors is that if you don’t need the full model state any more (don’t need to continue training), its state can be discarded, keeping just the vectors and their keys proper

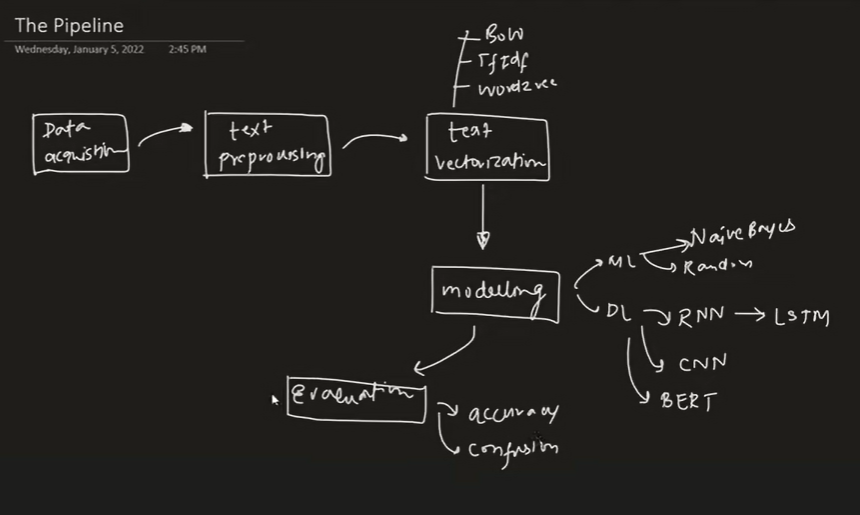
# **models.word2vec :**

# [gensim.models.Word2Vec](https://tedboy.github.io/nlps/generated/generated/gensim.models.Word2Vec.html#gensim.models.Word2Vec)

**classgensim.models.Word2Vec(sentences=None, size=100, alpha=0.025, window=5, min\_count=5, max\_vocab\_size=None, sample=0.001, seed=1, workers=3, min\_alpha=0.0001, sg=0, hs=0, negative=5, cbow\_mean=1, hashfxn=<built-in function, *vector\_size=100,***

**hash>, iter=5, null\_word=0, trim\_rule=None, sorted\_vocab=1, batch\_words=10000)**

* window = 5 --- here in coding means 5 words on each side of the center target word i.e. a/q our Notes(theory) the actual window size = 5+5+1 = 11
* min\_count = 5 -- means we only cosider sentences with at least 5 words in them.
* vector\_size = 100 ---- means our desired "output vector size " = no of neurons in hidden layer = 100



ML Techniques : Popular for Text Classification

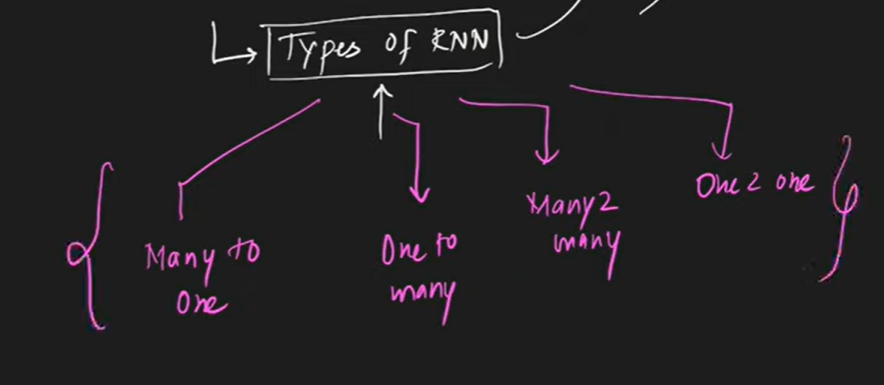
1. Naïve Bayes
2. Random Forest
3. S.V.M.

DL TECHNIQUES : Popular for Text Classification

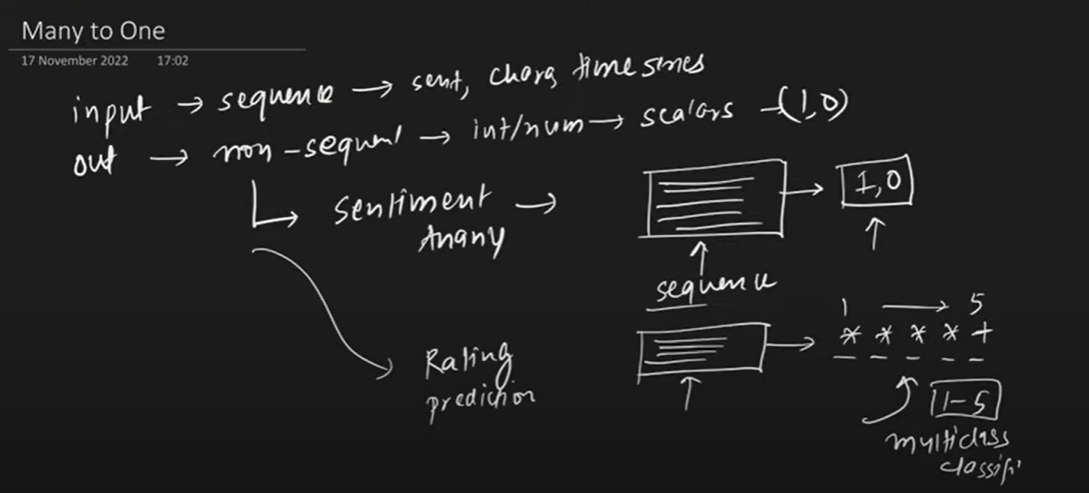
1. RNN --- Example LSTM model
2. Pretrained Models --- like BERT which is a Transformer Based Pretrained Model

RNN

Types:

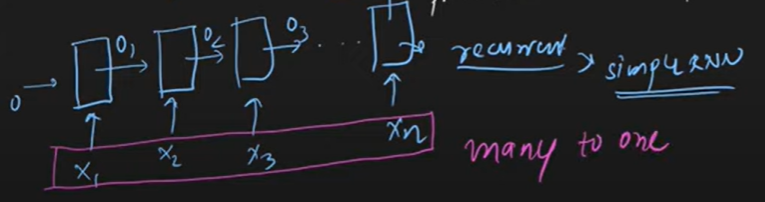


1. Many to One:



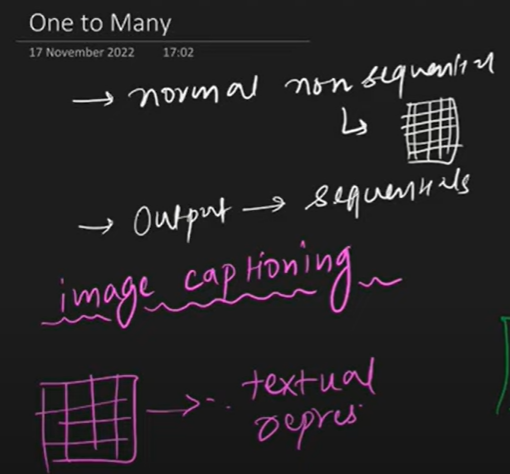
“Sequences” can be Sentences, Characters, Time Series input data.

Architecture:



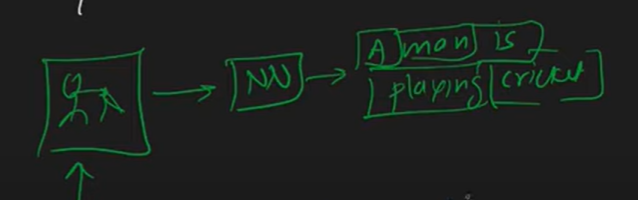
Observe : input is many and output will be Non sequential (but could be more than one value ).The output of previous layer(o1) goes as input to the next layer along with the next value in the input sequence(x1+1).

1. One to Many:

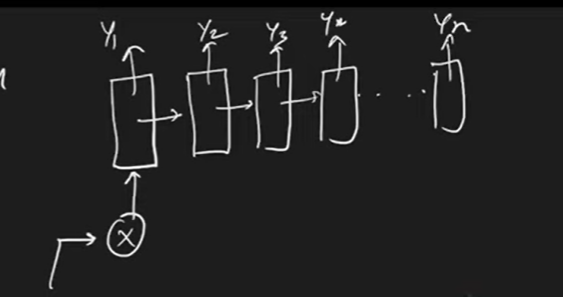


Input : Non sequential

Output: Sequential

Example: Image Captioning ----Here a image is given as an input, and, output is given as as a text which summarizes the object of the image. Since output is a text, hence sequential, while input is a non-sequential b/c it is a image. 

Architecture: It has only one input(x) in beginning and each time step results in a output(yn) which goes an input to the next time step.

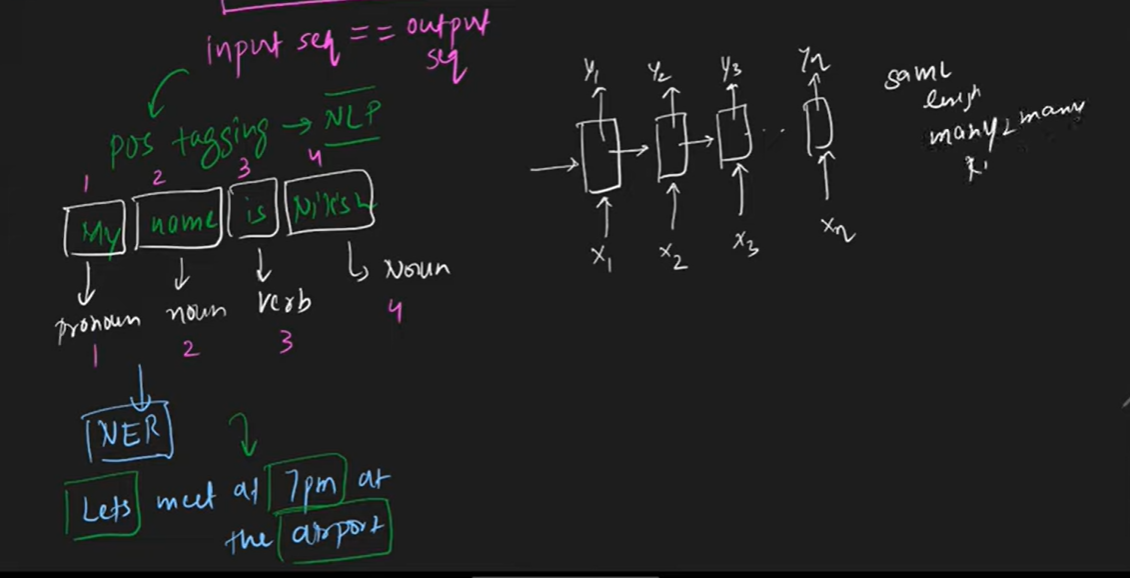


1. Many to Many /or/ Sequence 2 sequence RNN :

Two Types:

1. Same length i.e. Input length == Output Length.

Ex: POS tagging --- b/c every input word gets tagged with some POS tag. Hence same no of output.

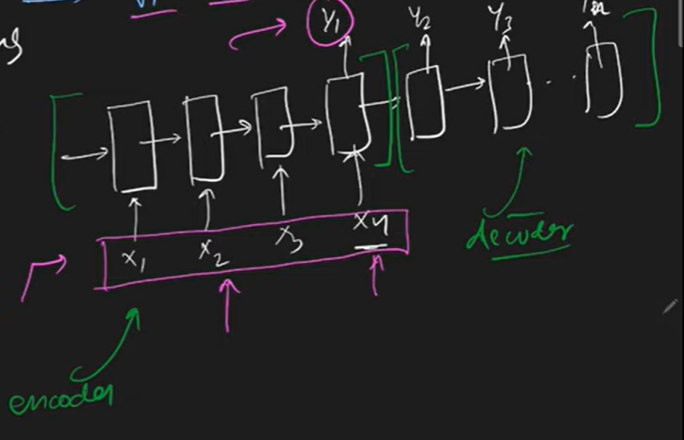


Observe : Every input (xn-1) has a corresponding output (yn-1). BUT, this yn-1 DOES NOT go as input to the next layer along with the next word in sequence(xn).

1. Variable Length i.e. Input length =! Output Length

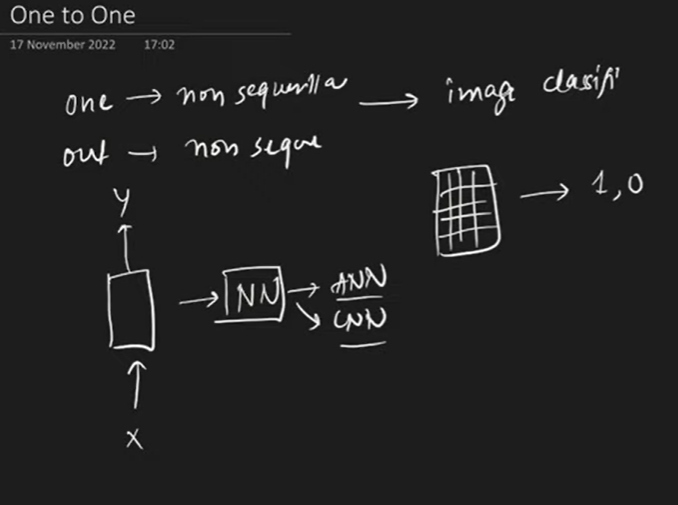
Ex: Language translation(google translate) --- one language to another. These models are also called Encoder-Decoder Model.

Architecture:

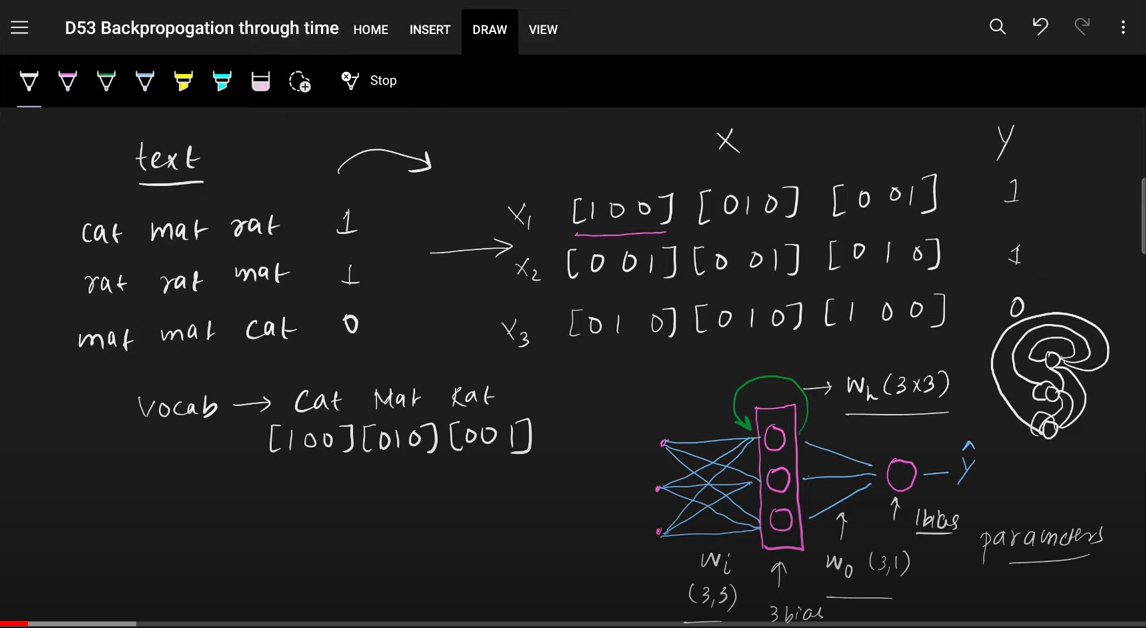


Here, in above picture, say the input sentence in English has 4 words(x1 to x4) . These are sent to the Encoder and unless and until the last word (x4) is sent to Encoder, the Output is NOT generated. Once the last word is sent, Then only Decoding / Translation starts inside the Decoder and we get output (y1 to yn) .

1. One to One RNN: Technically, this is NOT an RNN b/c it does NOT have any Recurrence and No feedback. Hence this is a simple N.N. which can be CNN or ANN.

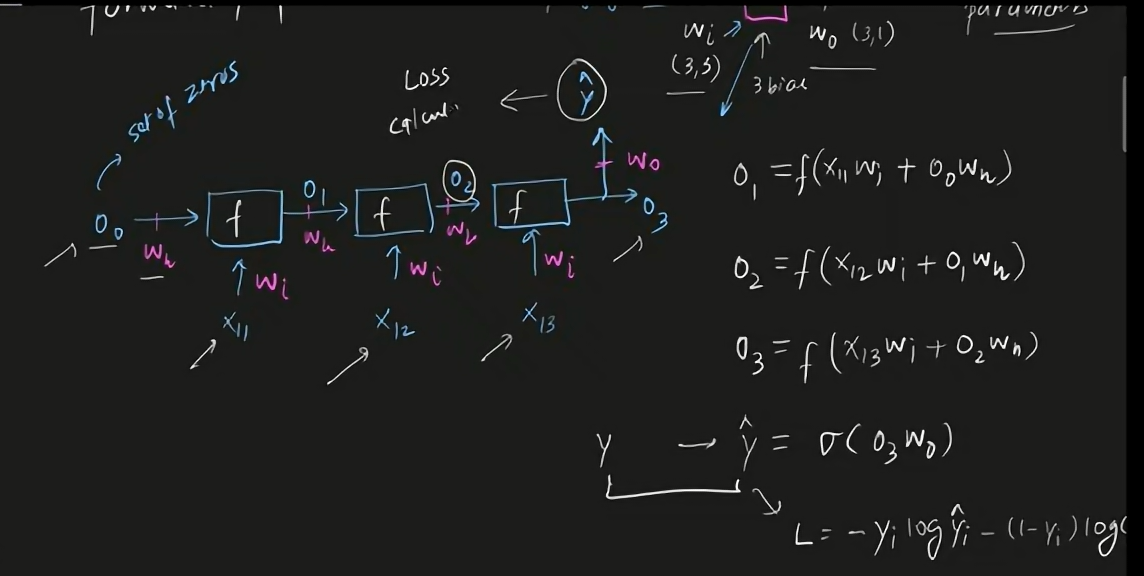


BACKPROPOGATION IN RNN

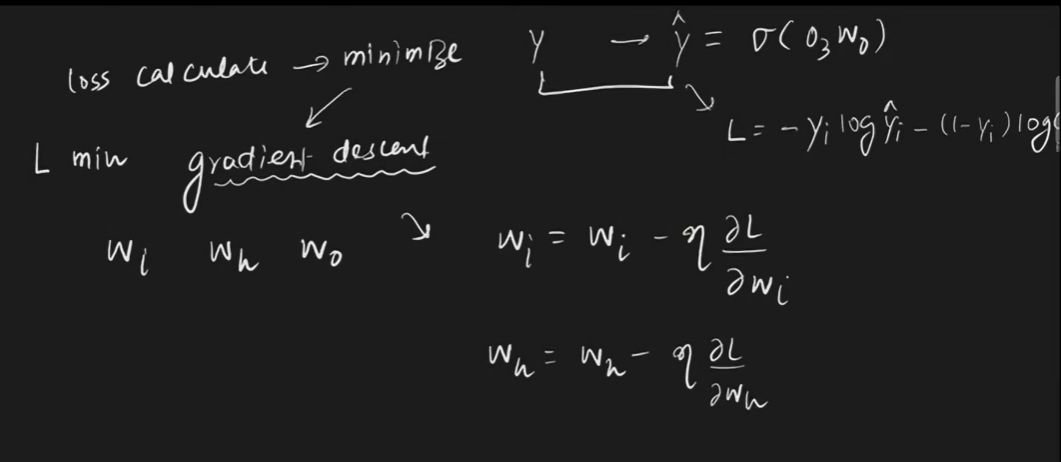


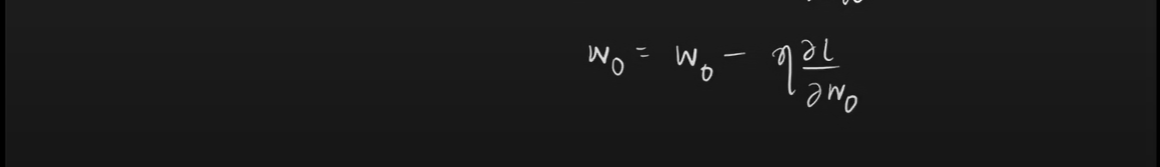
Consider (above image): It has 3 sentences with a total vocab of 3 words/tokens. We send each sentence in to the RNN one by one with each word of that sentence being sent sequentially. Hence the first layer must be of 3 neurons b/c each word is represented with a 3D vector. Next the hidden layer can be of any neurons (here it is 3). And the last layer is made of 1 neuron with sigmoid activationfor prediction.

Forward Propagation in RNN (discussed earlier):

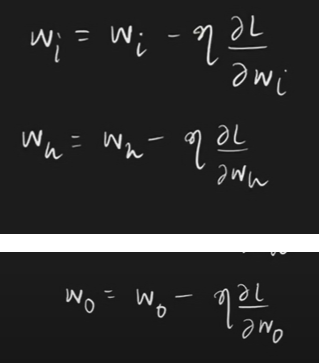


Backpropogation in RNN: In RNN Backpropogation is done through time(explained later). It is obviously done to minimize the Loss in each step. We calculate gradient descent after every step and get new values of weights and biases after every step.

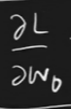


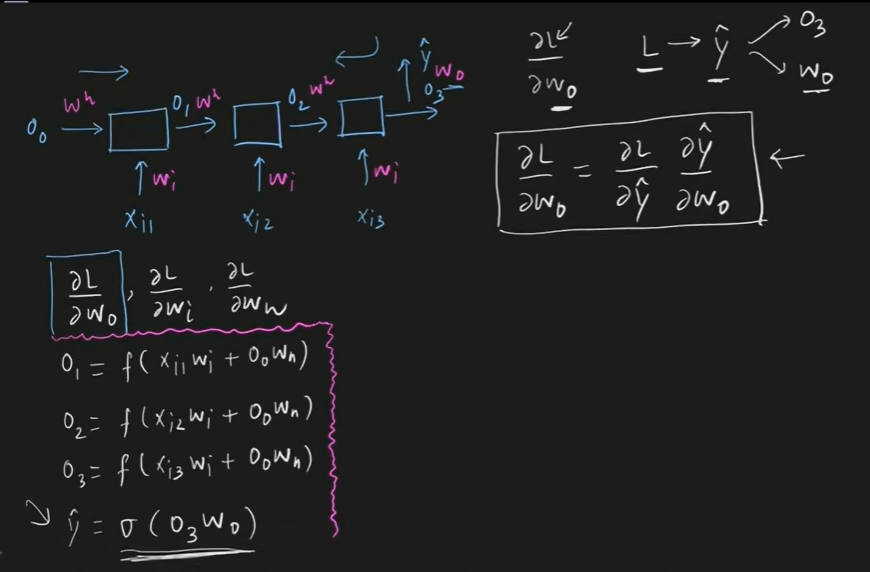


This Backpropogation is similar to what is read in NN or CNN i.e. we first get the derivative terms using Chain Rule.i.e. we get the following terms  first and then calculate updated values of “w” and “b” using the following eqns:

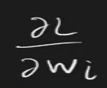


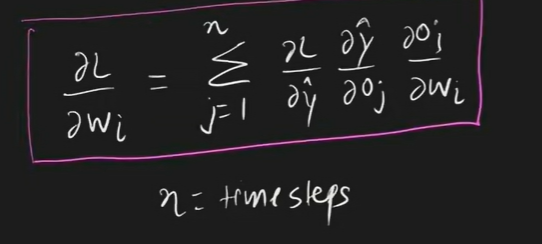
1. TO GET



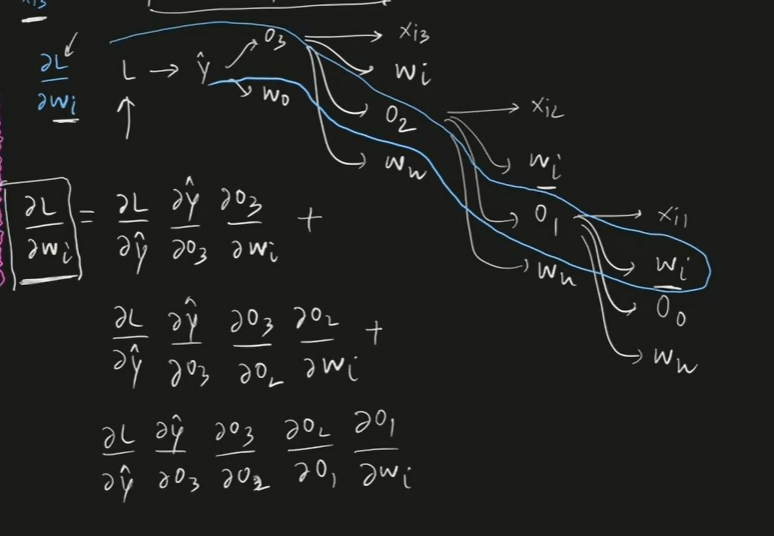


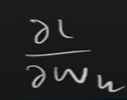
1. TO GET

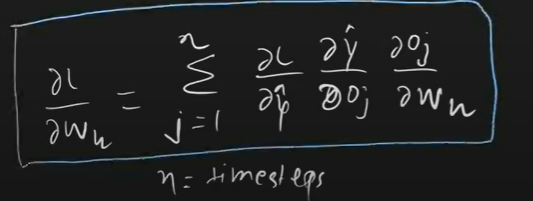




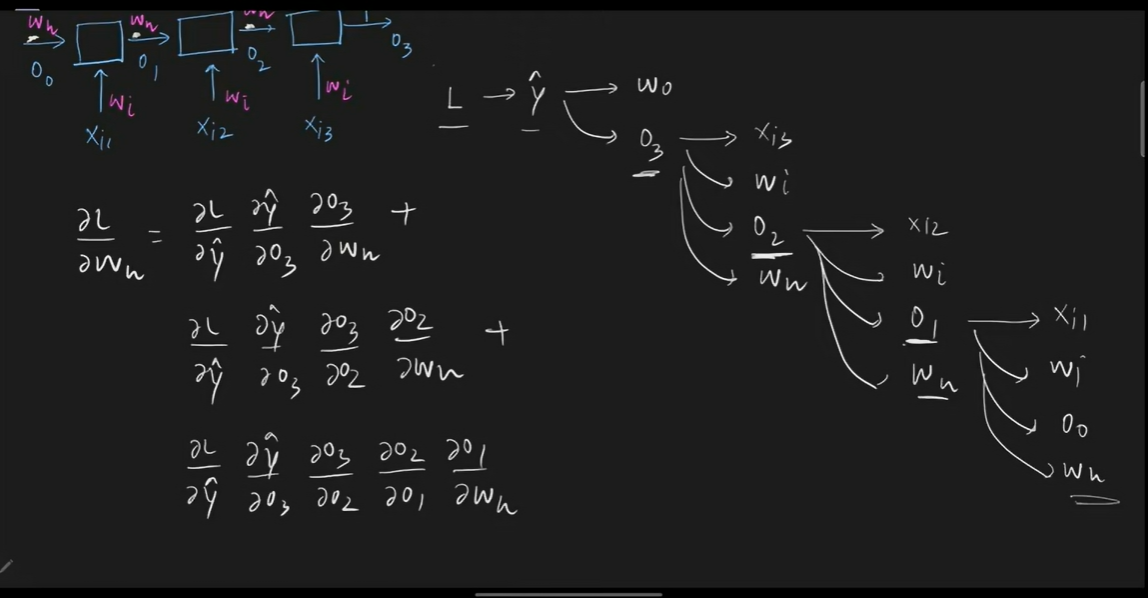
Upon expanding, we see why Backpropogation in RNN is done through time. This is b/c in order to get the value of the derivative , we would the values/outputs from previous time stamps (i.e. O2 and O1) along with the output of the current time step ( i.e O3 here) . As discussed earlier, RNN has the ability to remember the past state/values (i.e.O2 and O1) . To access these past values and get the derivative using the Chain Rule, the RNN must unfold through time.



1. TO GET   
   



Upon expanding,



## **Two Issues of Standard RNNs**

### **1. Vanishing Gradient Problem**

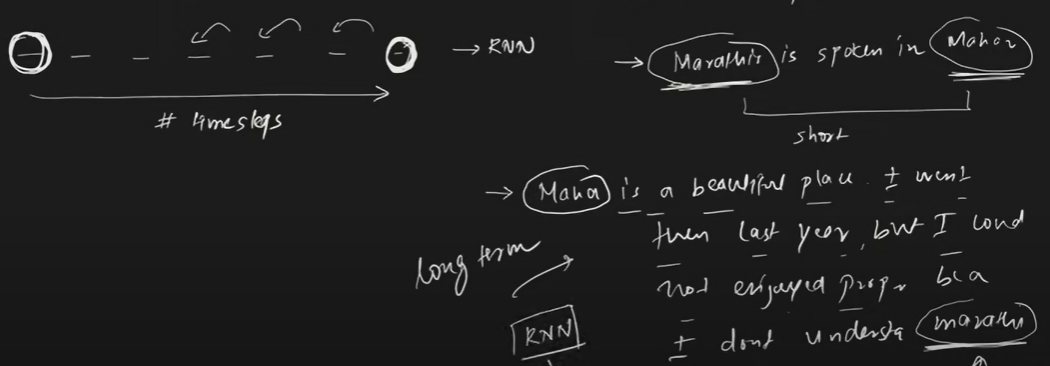
Recurrent Neural Networks enable you to model time-dependent and sequential data problems, such as stock market prediction, machine translation, and text generation. You will find, however, RNN is hard to train because of the gradient problem.

RNNs suffer from the problem of vanishing gradients. The gradients carry information used in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult. This problem is consistent in sentences with Long Term Dependencies in them.

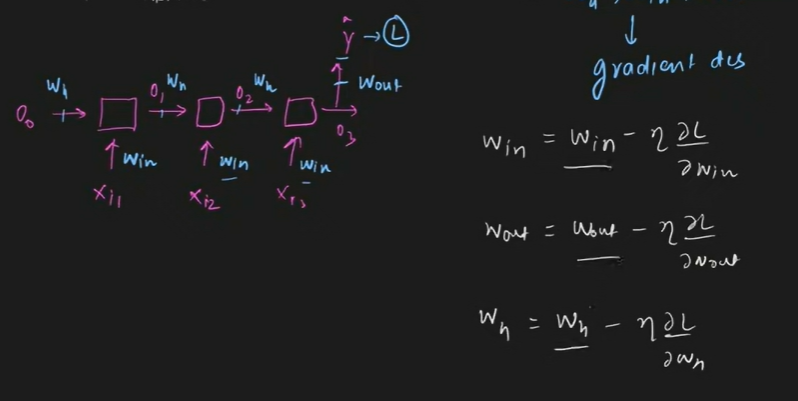
Consider 2 sentences shown in below image:

In sent 1. Output is Maharashtra which depends on the 1st input token “marathi” --- This sent is an example of Short term dependency b/c output depends on 4th to last word in sentence.

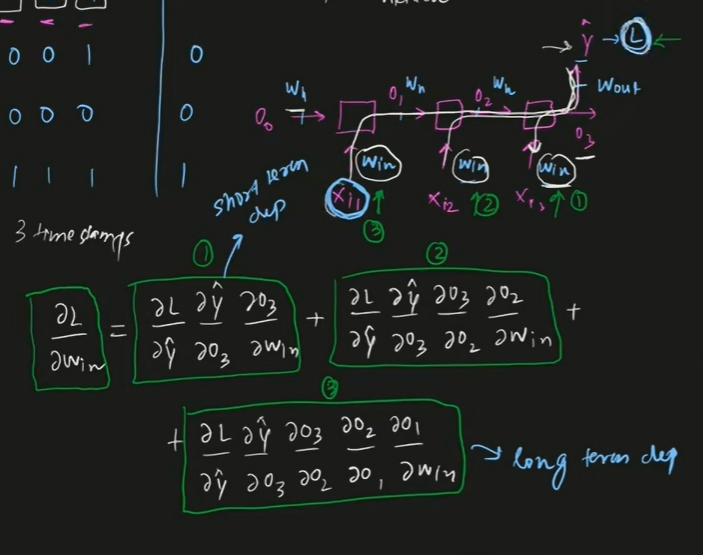
In sent2. Output is marathi which depends on the 1st input token “Maharashtra” ---- This is an example of Long term dependency b/c output depends on 21st to last word in sentence.



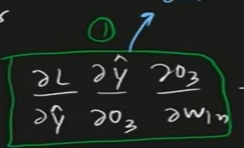
So what causes Vanishing Gradient??

Consider a RNN with 3 time steps/3 words in a sentence max.  


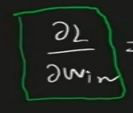
The objective of RNN is to minimize “L” by Optimizing values of Win, Wout and Wh. To do this we calculate gradients which, as discussed earlier, using chain rule (as shown below).



The above picture shows that the 1st  term of the Chain rule i.e.

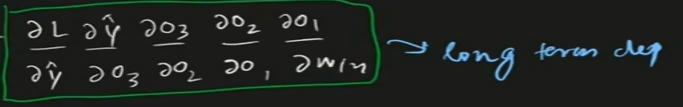


Used to to get

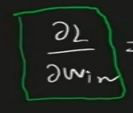


denotes the Short term dependency b/c Win in that term is the last word (or we can say the most immediate word entering the NN at last time step) of the input sequence. Thus this term denotes only short term/immediate term dependency.

Slly, That picture also shows that the 3rd  term/or/ last term of the Chain rule i.e



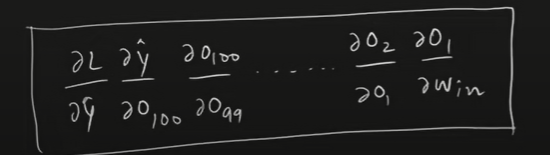
Used to to get



denotes the Long term dependency b/c Win in that term is the first word (or we can say the earliest word entering the NN at first time (t=1) step) of the input sequence. Thus this term denotes only Long term dependency b/c memory would have remembered this term( this Win) the longest period of time b/c it entered the NN at the earliest start of the time step.

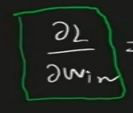
Now this last term when we only had 3 input sequences i.e. a time step of 3. But, what if we had 100 time steps/or/ 100 input words in a sentences/sequene.

Then, the last term which would denote the long term dependency would look like this:====

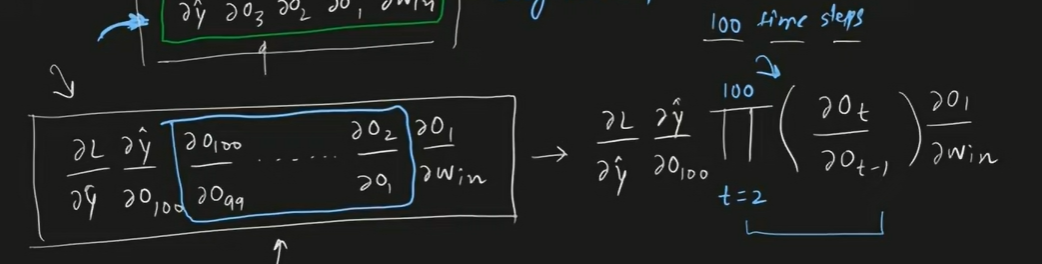


LOGIC OF VANISHING GRADIENT IN LONG TERM DEPENDENCY: We can show that Longer the dependencies get, the smaller the values of the trailing terms(or last terms) become. i.e. the values of the last or second last or 3rd last …terms, that we get during chain rule will be very very small.

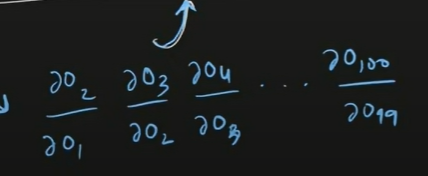
Consider the last term of a sequence which has 100 word/100 time steps in it. And we want to get

 using Chain rule.

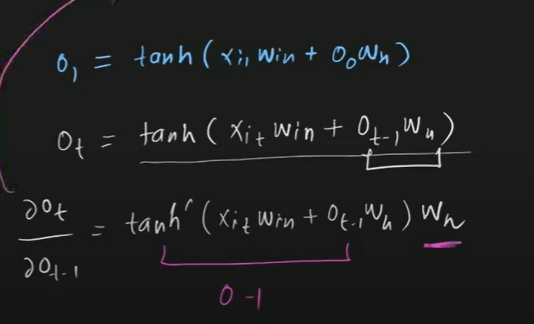
This is how the last term would look :----



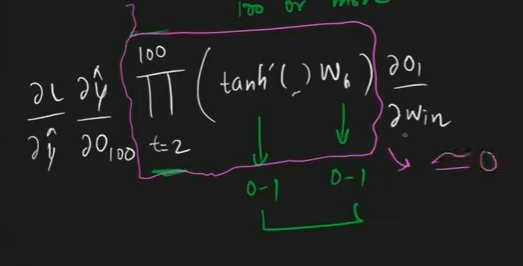
Consider this portion only -----



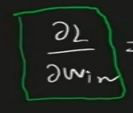
Recall on how to get O1/or/Ot :--here tanh is the activation fn. Note: we can also add bias “b” to the term



Substituting --- Consider that we initialized Wh with values (0-1), then in that case derievative of tanh (also b/w 0-1) multiplied by Wh will always result in a value <1 . Then this values are multiplied to 100 other values, all of which lies b/w 0-1 , would result in a Net value which is close to 0. This is what is called Vanishing Gradient Problem.



Thus we can say that all the Long term dependency terms will result in values which are very very close to 0. Hence , when we calculate

 using Chain rule, the output would only depend on the Immediate/Short dependency terms/most recent terms, as all the Long dependency terms will be 0.

How to solve this problem of Vanishing Gradient???

* Use different activation function than tanh – like use RELU or Leaky RELU
* Better Weight Inintialization – that is we ensure weights have higher values >>1
* Use Skip RNNs
* Forget RNNs , Use a different Architecture like LSTM ….(most preferred).

### **2. Exploding Gradient Problem**

While training a neural network, if the slope tends to grow exponentially instead of decaying, this is called an Exploding Gradient. This problem arises when large error gradients accumulate, resulting in very large updates to the neural network model weights during the training process

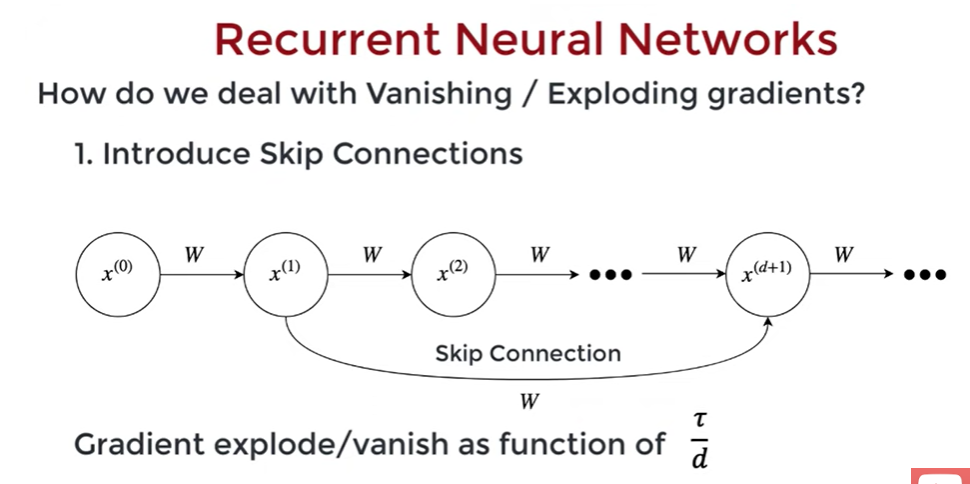
This problem can occur when we have Activation Function as RELU whose derievatives can be any value b/w (0, infinity). B/c of this if we end up with a large value, then the Long Dependency terms would explode(become very large) b/c of which our weights would become very large and training would crash. That is why, we keep tanh as our default Activation for hidden layer in RNN

Another way we may end up in this mess, is if we initialized the learning rate with very high values,

* Graddient Clipping --- Here we define a upper limit on the Gradient using our code.
* Skip Connections
* Forget RNNs , Use a different Architecture like LSTM ….(most preferred).

SKIP CONNECTIONS:

we can add additional edges called skip connections to connect States some D neurons in front of it so the current state is influenced by the previous state and a state that occurred D time steps ago gradients will now explode or vanish as a function of tau over D instead of just a function of tau .



**2. use Gated RNN : example lstm**

