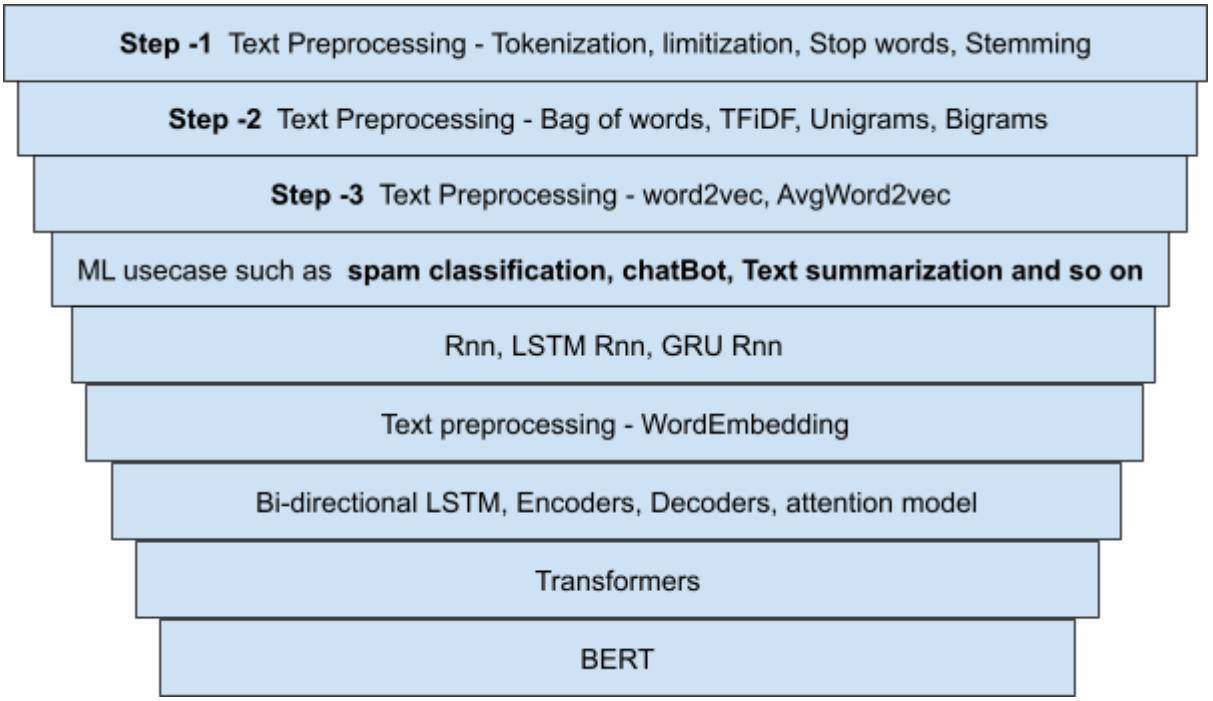
Natural language Processing

What is text preprocessing?

It means how we clean our data or how efficiently our words or vectors are.

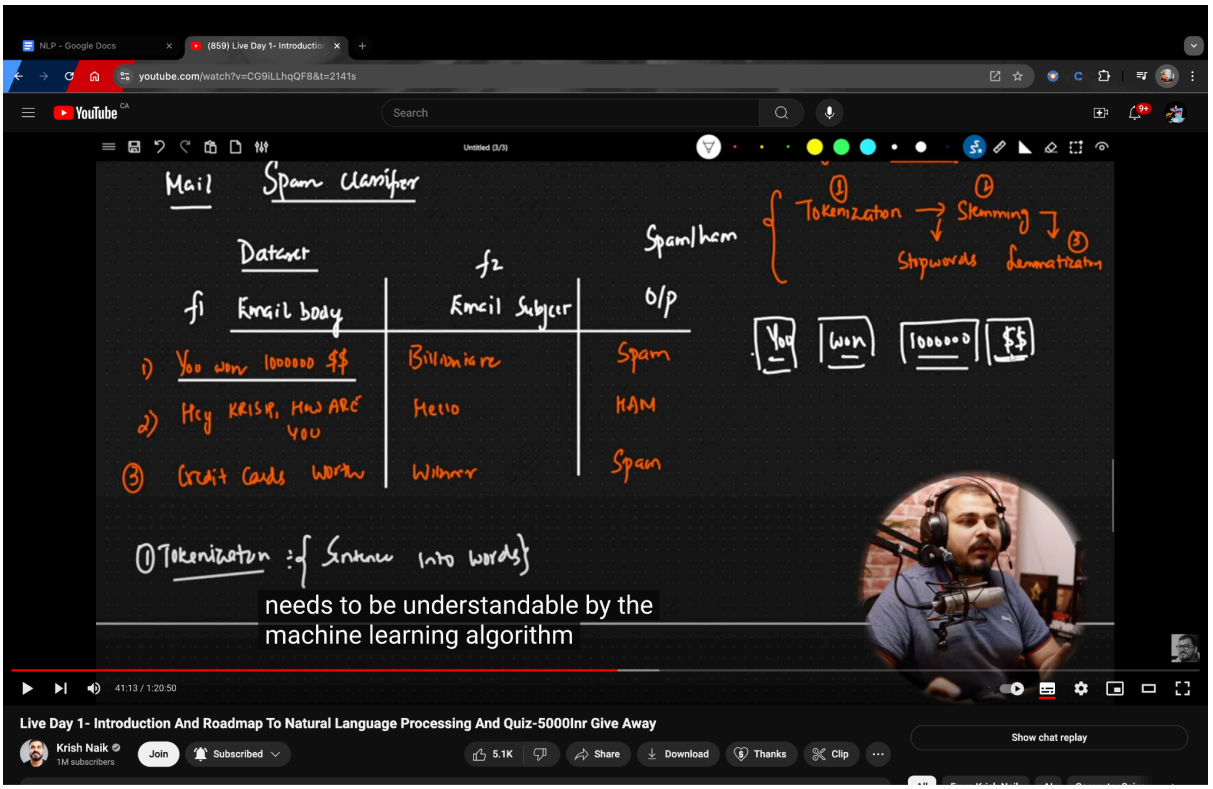


As per the above figure step 1 is for **cleaning data**, then step 2 techniques are used for **converting data into vectors**. Moreover, step 2 techniques **are not sufficient** to convert into vectors so we used others **for the same reason**. Until here we have a **good amount of vectors**. Then we use the **MI use case** basically these are the problem statements after that we apply **deep learning techniques.** Now we use advanced text preprocessing which is **WordEmbedding which internally** uses a technique of word2vec but is far more advanced compared to other techniques such as **word2vec**, **AvgWrod2Vec.**

What is the workflow of NLP?

Let's say we have use case of **spam classification.** In the below image, we can see the dataset.

The workflow of NLP is that first, we do tokenization, stemming, stopwords, limitization.



Firstly, Tokenization , it will just convert **sentence into words, then** use stop words. For example, **“Hey buddy i want to go to your house”.**

Here, we remove not useful words such as “to” , “of” and etc., so we remove them , But words like ‘**not’** is an important.**However,** in spam classification we don’t need these words but in **text summarization** we need these words.

Moreover, we can also create our own Stop words list too

Next is stemming, we focus on is that we try to find out the base of this words or base stem of a specific words. For instance, we have two words “**historical**” , “**history**” so here what is our base? Base is **histor ,** this words doesn’t have any meaning. So, **Stemming** is the process of reducing words to their base word stem and has only one **disadvantage** is that after stemming that word doesn’t have any meaning.Also, its helpful when we have words like going. Goes, gone then these time our base is **go** which is meaningful word.

Moving forward with lemmatization, In orer to overcome the stemming problem, we use lemmatization. It helps to get meaningful words from base word. **For example,** history, historical we get **histor** as a base word and if we use lemmatization then we get **history** as a output.

Let's take another example, Changing and changed so the final output is change.The advantage of lemmatization is that we get meaningful words, while the disadvantage is that its slow as it has a whole library where it checks the words and then it gaves us output.

We use stemming in some usecase such as **spam classification, Review classification,** whereas lemmatization can be used in **text summarization, language Transaltion, chatbot.**

In the steps of text-preprocessing, 1. Tokenization 2. Stop words 3. Stemming 4. Lemmatization. However, sometimes we don't need stemming and it depends on the usecases which mention above.

Now, our second step is to convert words into vectors and there are different techniques such as bag of words, TFiDF, Word2Vec.

Basic Terminologies used in NLP. 1. Corpus **2.** Documents **3.** Vocabulary **4.** Words. For example, we have dataset for santiment analysis

TEXT	**OUTPUT**
The food is good	1
The food is bad	0
Pizza is amazing	1
Burger is bad	0

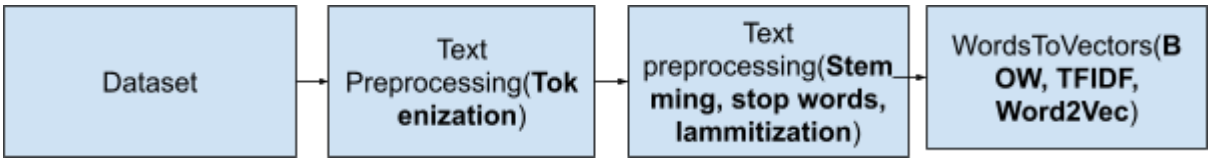
What is Corpus?
Corpus means that we combine all the text and make **paragraph.**

What is Document?
Document means that each data point or sentence is a document such as "The food is good". We say its sentence but in **NLP** it considers as a document

What is Vocabulary?
It means that How many uniques words have in dataset. **Suppose,** we have 10k uniques words in our dataset, so these are considered as a vocabulary.

What is words? -
No need explain as know what it is.

Steps that we perform in NLP task



First let's understand One hot encoding.

Let's say we have **corpus,**

[A man eat food
Cat eat food
People watch Krish YT**]**

Here, we have 9 vocabulary as follows

[A man eat food cat people watch krish yt]

Let's convert it into vectors
D1 [[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]] => This is One hot encoders.

D2 [[1 0 0]
[0 1 0]
[0 0 1]]

Here, Prons and cons are as below

Advantages	**Disadvantages**
Simple to implement	Spare Matrix (**unnecessary save of memory**)
Intutive	Out of Vocabulary
	Semantic meaning between word isn't captured (no relation between any words in document)

Extra vocab in test data then not able to handle it right

What is Bag of words (BOW) ?

Let's say our dataset have three documents as below

D1 -> He is a good boy
D2 -> She is a good girl
D3 -> boy and girl are good

In above three sentence, there are some generic word which we will remove such as He, is ,a , and, she, are with the help of Stop words as these words helpful for Chatbots and not for the sentiment analysis , So our new and cleaned dataset looks like.

Also during Stop words, ensure that all the text must be in lower case

D1 -> good boy
D2 -> good girl
D3 -> boy girl good

Let's count unique words for vocabulary

There are three Uniques words Good, boy, girl then we check the frequency of each words like good 3 , boy 2, girl 2 .

Now these vocabulary become our feature 1, 2 and 3. And order of the feature will be based on the frequency such as feature 1 will be good, feature 2 will be boy and feature 3 will be girl.

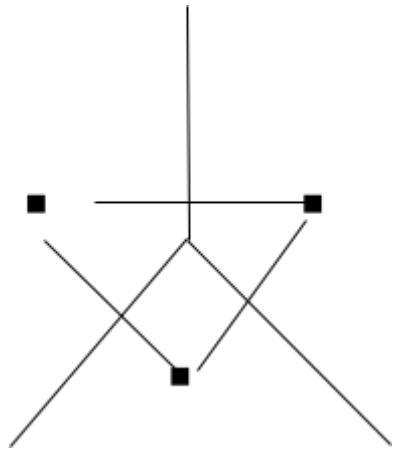
Remember we cleaned our data that we use in below table.AS we know Our document 1 has Good boy so we increase our count 1 in good and boy, and in girl 0. Likewise we do for the Document 2 and 3.

	F1	**F2**	**F3**	**O/p**
	Good	**Boy**	**Girl**	
D1	1	1	0	0
D2	1	0	1	1
D3	1	1	1	1

Now in BOW,, we increase the count but here there is an option which is **binary BOW** it means it even tough there are multiple same words, count will be always 1

Advantages	**Disadvantages**
Simple to implement	Sparsity is still there (**unnecessary save of memory**)
Intutive	Out of Vocabulary
	Ordering of the words has changed
	Semantic meaning not able to capture. (**Cosine similarity**)

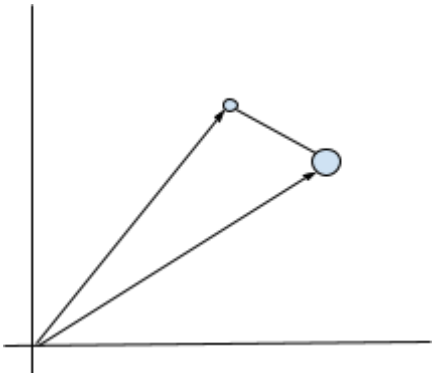
Now, if we plot above points in 3D way,



There are some techniques to get the distance between two points, such as **Euclidian method ,** and **Cosine Similarity.**

Cosine similarity is really helpful in **recommendation system.**

What is cosine similarity ?



Suppose we have two points, and angle between points is Cos 45 and what it is around 0.53 , now to find out the , cosine similarity = 1 - 0.53 = **0.47**
Now, if we have one point on Y-axis and one on X-axis then Angle will be 90 degree, so cos 90 is 0, then Cosine similarity is 1 and distance is huge.

Let's understand through example, we have one points **Avengers** on X-axis (1,0), and another point on same axis (0.5,0) which is **iron name. Here,** angle is 0 that means cos 0 is 1, then Cosine similarity is 0 and very close to each other.

Now, in order to **Capture the semantic info**, we use **Ngrams.**

There are types - Bigrams, trigrams, and so on.

	Good	**Boy**	**Girl**
D1	1	1	0
D2	1	0	1
D3	1	1	1

What bigrams means that apart from these single features it makes combination of features such as good boy , good girl and add these into as features

Good boy	**Good girl**
1	0
0	1
0	0

In Ngrams, we provide value such as (1,3) and it means we apply from unigrams to trigrams. For example, "Krish is not feeling well", then first we make single features such as krish, is, not, feeling,well then we use bigrams and make feature, then trigrams will be applied like krish is not, is not feeling, not felling well.

TF-IDF

There are two sentences, D1 -> The food is good, D2 -> The food is not good

	Food	**Good**	**is**	**not**	**The**
D1	1	1	1	0	1
D2	1	1	1	1	1

Now, above both sentences are totally opposite, however if we use cosine similarity, We get this similar and they aren't. To resolve this issue, we use TF-IDF.
Also, in vectors either we get 1 or 0, so there is one more issue that there is not any semantic meaning as there are too many 1, Given more important to them.

Now what is TF-IDF?

It means Term frequency and inverse document frequency, here term frequency means give more weightage to the rare words that are present in sentences.**Now,** except the rare words, other words are captured by inverse document frequency.
When we combine both of them, we are able to find the rare words in documents.

How to calculate Term Frequency ?

Term frequency = No. of repetition of words in sentence / no. of words in sentence

How to calculate Inverse Document Frequency?

Inverse Document Frequency = $log_e(No.\ of\ sentences\ /\ No.\ of\ sentences\ containing\ words)$

After multiply both, we get TF-IDF. Let's take above documents to calculate TF and IDF.

D1 -> good boy
D2 -> good girl
D3 -> boy girl good

Term Frequency			
	Sentence 1	Sentence 2	Sentence 3
Good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

Inverse Document Frequency	
Words	**Sentence 1**
Good	$log_e(3/3) = 0$
boy	$log_e(3/2)$
girl	$log_e(3/2)$

Now, we multiply both tables,

	Feature 1	Feature 2	Feature 3
	Good	boy	girl
Sentence 1	½ * 0 = 0	½ * $log_e(3/2)$	0 * $log_e(3/2)$
Sentence 2	½ * 0 = 0	0 * $log_e(3/2)$	½ * $log_e(3/2)$
Sentence 3	⅓ * 0 = 0	⅓ * $log_e(3/2)$	⅓ * $log_e(3/2)$

Now, as per the our sentences, good is present everywhere and in above table, good has all zero value which shows that its not playing big role , while boy and girl

present in only two sentences and both has some values. So thats how we get vectros with the help of TF-IDF.

Advantages

1. Intuitive
2. Getting word importance

Disadvantages

1. Sparsity is still there, however, it's less than a **Bag of words**
2. Out of vocabulary

Word Embeddings

It’s a technique that converts words into vectors. Whatever the techniques we learn, it's divided into two ways 1) Count or frequency based on frequency of words, 2) Deep learning trained model.

Firstly, count or frequency, there are three techniques such as Bag of words, TF-IDF, One hot encoding. Secondly, in deep learning trained model, we use Word2Vec and it has two more sub techniques such as CBOW, skip grams.

Word2Vec

One important thing in word2vec is Feature representation. Let’s say we have some features vocab, let’s consider we have 6 features, boy,girl,queen, king, mango,apple.

Before we start word2vec, the problem of BOW and Tf-IDF was semantic meaning, there weren’t any semantic meaning for example, honest and good can be had semantic meaning. Also, thet have sparse matrix, too many 0 and 1 in matrix and somewhere due to BOW and TD-IDF, there some value in decimal values(0.2341) because of this, huge dimensions were crated. It takes lot of time for processing, training and it leads to overfitting since we have lots of 0 in matrix. To solve all these problems we use **Word2Vec.**

In word2vec, we use limited dimension that helps to represent entire vectors such as we created 100 dimensions so within we represent entire vectors. Also, sparsity will be reduced, less number of 0s will be there. Apart from this, semantic meaning will be maintained.

Now, In Word2Vec, we represent our feature words with some number of features, for example, gender, royal, age, food, like this we have n numbers of dimensions features.

Let's say vectors of king, man, women, and queen are [0.96, 0.95] , [0.95, 0.98], [-0.94, -0.96], [-0.96, 0.95] respectively.

If we perform king - men +women we get Queen and how will it happen ?

It happens by cosine similarity, how to find distance between two points ?

Formula is **distance = 1- cosine similarity**, lets say we have two points and angle between them is 45 degree. Then Cosine similarity = cos theta = Cos 45 = 0.7071

Now the distance is 1- 0.7071 which is nearly equal to 0.3

In **cosine similarity** if the value is **towards 1** then it would be considered as very **similar**. If the value is t**owards 0** then there is **no or less similarity** between the two points or vectors, while in **cosine distance** if the value is **more towards 0** then there is a **similarity** between the points or vectors and vice versa.

How word2vec is trained and created ?

1) CBOW(Continuous Bag of words):

Corpus: Krish channel is related to data science

In order to use CBOW, we need to decide window size, lets say window size is 5 and it helps us to create training data.Training data has two important thing first Independent features and second is Output.

Let's consider window size is 5 and **it always good to have size in odd number** to get the same number of independent features, so we take first 5 words of corpus ,so " Krish channel is related to " are the first 5 words and take the center word which is " is ". So, " is " the Output or target, while other words are context words and it really helpful to bring semantic meaning.

Then we move our channel right side so next words will be
"Channel is to data", and output is "related".

Likewise, move to right and get "is related data science" and target is "to".Now, we can't move right as we don't get 5 words so we stop here.

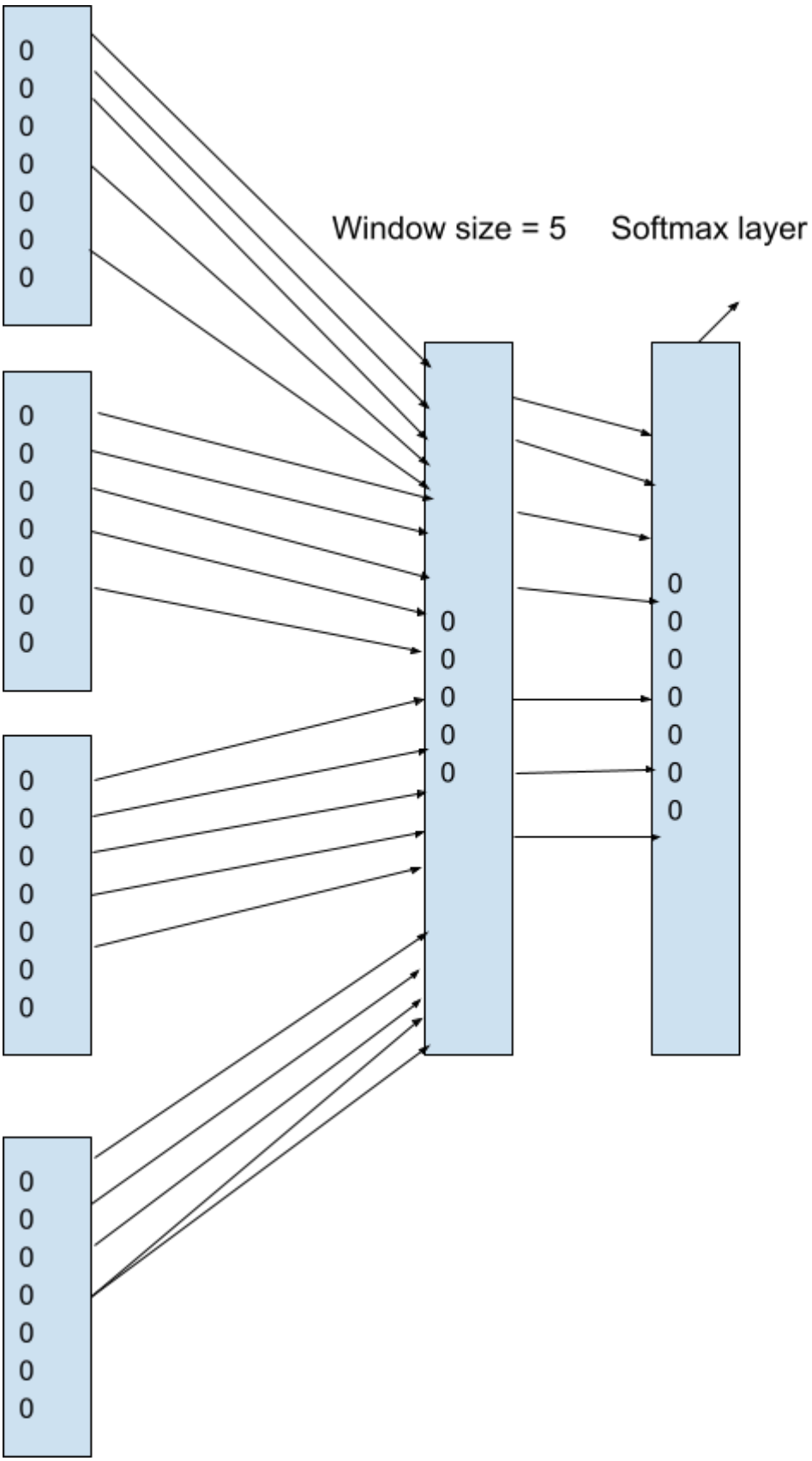
Independent feature	Output
Krish channel related to	is
Channel is to data	related
is related data science	to

The main aim of **word2vec is** to take independent features and based on the output create vectors. Vectors should be in such a way that has a semantic meaning like output feature.

We simply use BOW for the vectors and we know how to make vectors see below example. Take single words and as per the corpus we set vectors.

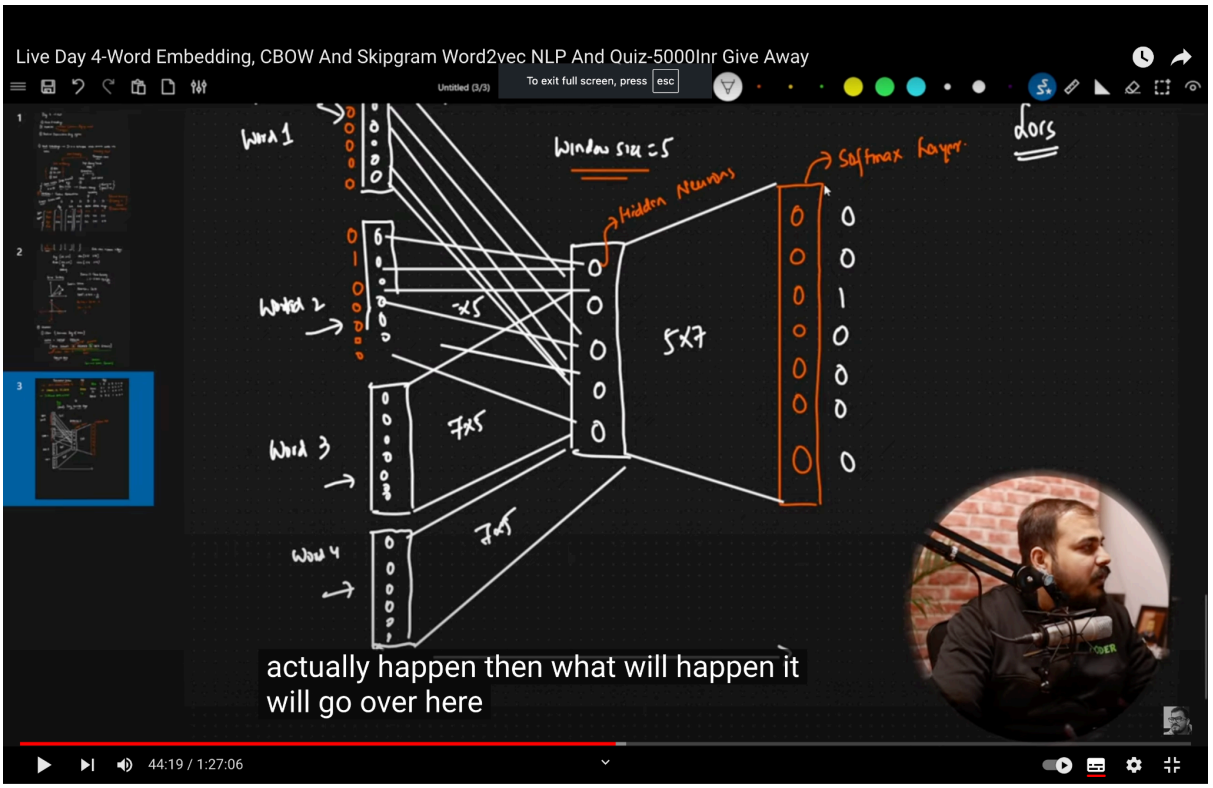
Krish 1 0 0 0 0 0 0
Channel 0 1 0 0 0 0 0 , likewise till the end of each words.

Now, we gather all the independent features, output along with our vectors and make fully connected layer (**ANN**). If our input is 4 then we take the 4 matrix as per below



Here, each input has 7X5 matrix. And hidden layer will be 5x7 matrix.

Here 0s are not the values those are the neurons,there are 7 neurons and their values are as per the vectors like krish [1 0 0 0 0 0 0 0]. Also, in hidden layer, we use 5 neurons as windows size was 5 then we use softmax layer with 7 neurons.



Moving forward with forward propagation then we calculate the loss = (y - y^). We subtract the value we get from neuron and actual value.

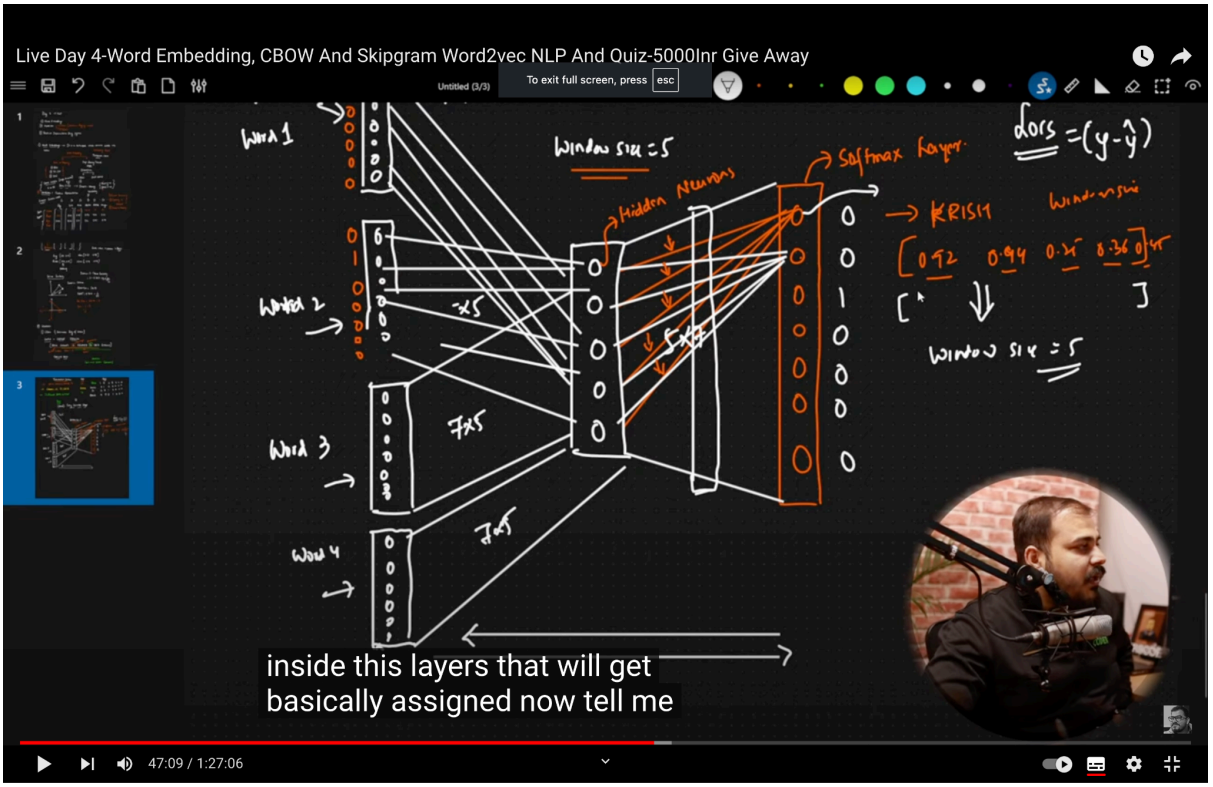
For example, independent feature is “Krish channel related to “ and to calculate the loss, neuron values would be anything and actual value is “is” so value will be [0 0 1 0 0 0 0] then we calculate the loss and then we do **backward propagation** , and keep doing forward and backward until we don’t get a loss value very low.

In the end how do we represent each nodes of softmax layer , we know first node represent krish, second node represent channel, third represent is and so on,

what would be the vector after training ?
Its simple that we know its 5x7 matrix so each nodes of hidden layer is connected to each those 7 vectors.

So the node that represent krish given by five vectors(hidden layer), likewise for each nodes this will happen. Hence, each node will have 5 vectors because of

window size. **For hyperparameter tuning ,** we just extend windows size.



So now the embedding layers will come between hidden and softmax layer as per the image above. After trained and all we get all the semantic meaning.**So this the CBOW.**

2) Skip Grams

Here, in skip grams, inputs are the outputs of CBOW and outputs are the inputs of CBOW. For example, in CBOW “is”, “to”, “related” are the output while these are the inputs in Skip grams.

Output	Independent features
Krish channel related to	is
Channel is to data	related
is related data science	to

That’s the difference between skip grams and CBOW and rest of the thing will be same like neuron layer.

There are two types of Word2Vec which are CBOW and Skip Grams.Also, we know the library which is gensim. Through it we can train model and there are two types of word2vec first one is pretrained(**Google news, 300 size**) and another one is
 Train model from scratch.

Should we create model from scratch or use pretrained model?

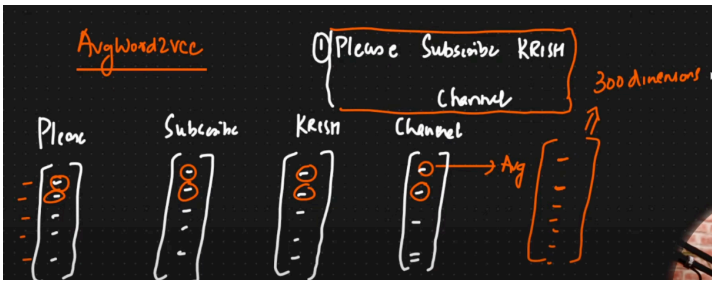
If a pretrained model captured 75% of our dataset, we will use it, otherwise we use a model from scratch.

Now, when we use word2vec what happened there ?

So word2vec converts each word into vectors in specific size. For example, “I want to eat pizza”, and **use google pretrained model or Model from scratch** so that each word has 300 dimensions, but we want that the whole input has only 300 dimensions instead of each word.

To fix this issue what we use?

There is a concept called **AvgWord2Vec,**



Let’s take an example, “**Please subscribe Krish channel**”

Each word has 300 dimensions if we use a **pretrained model**, but we want 300 dimensions of the whole sentence.So, as per the above image, we take **the first dimension from 300 dimensions of each word** and do **addition to get an average**. Likewise, we do for all dimensions and make new dimensions ,so that's how we make 300 dimensions for the whole sentence as per the above image.

Once it’s done we have input of 300 dimensions and o/p will be spam and ham.

Recurrent Neural Network (RNN)

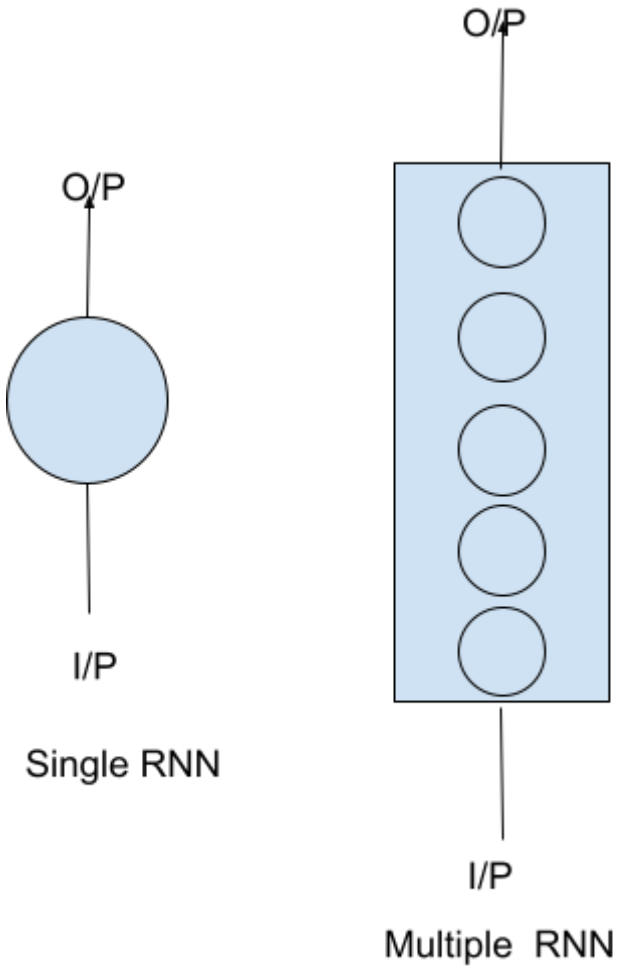
Let's say we build a chatbot application and we need some specific answer of specific questions and here **Sequence of words** very important.

Another application is **Translation** where sequence of words also crucial and next application is Text generation when we write any email or sending any message we get suggestion of words.

For all these tasks, we can't depend on **Machine learning task,** we need **deep learning models** such as RNN, LSTM-RNN, Transformer, BERT.

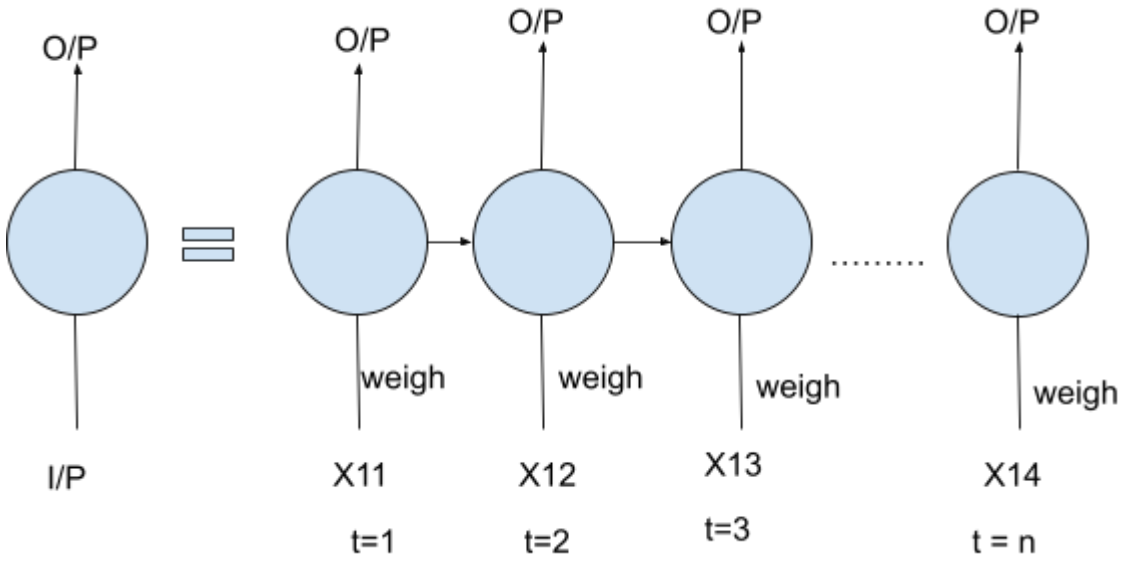
Then what is word2vec?

Word2vec is a technique in deep learning to train model or pretrain model to get vectors from words which helps to capture semantic meaning.



Whether its single or multiple RNN, output will be sent back to the neuron everytime.

Let's Expand it by considering the sentiment analysis.



Every output is given to the next neuron and this will work with respect to the timestamp with t = 1,2,3,..,n. So whatever the output we are getting we are giving it to next timestamp.

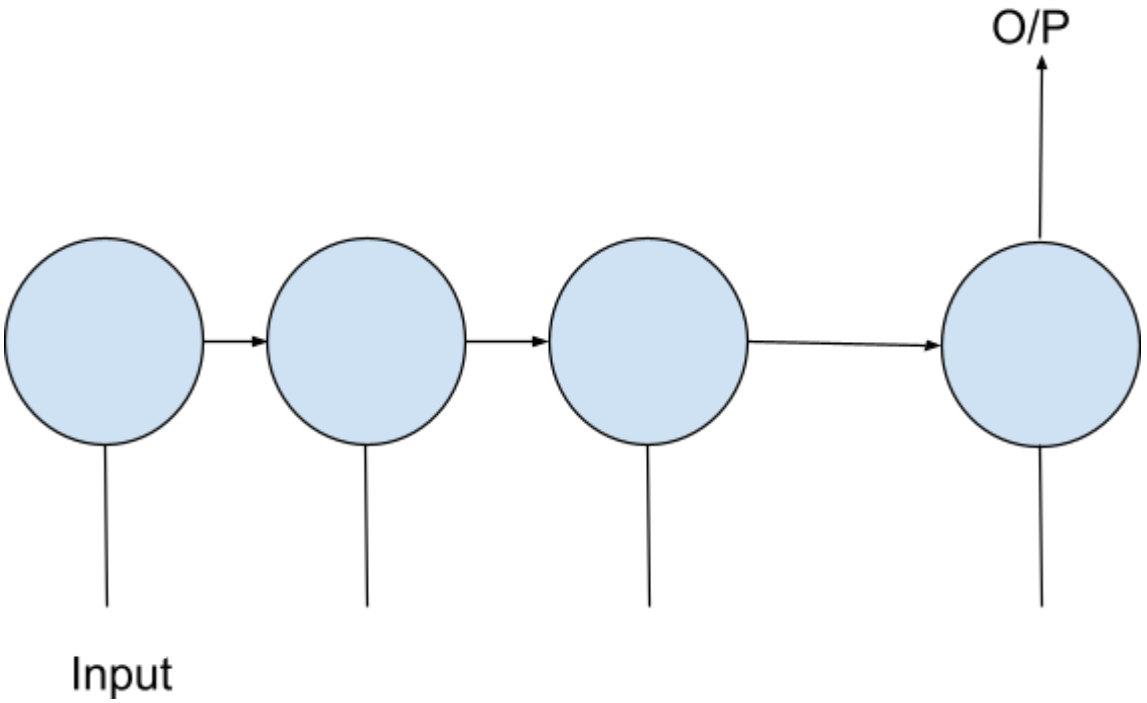
Let's understand by Sentiment analysis, " **The food is good**" and output is "**Positive**"

Now, how will information be sent to the RNN ?

The words of all sentences are represented by X11, X12, X13, X14 respectively. We give X11 input to the first neuron and then apply weight and then get output, after that output will be sent to the next neuron and in next neuron X12 input will be applied and then weight. Likewise it goes ahead.

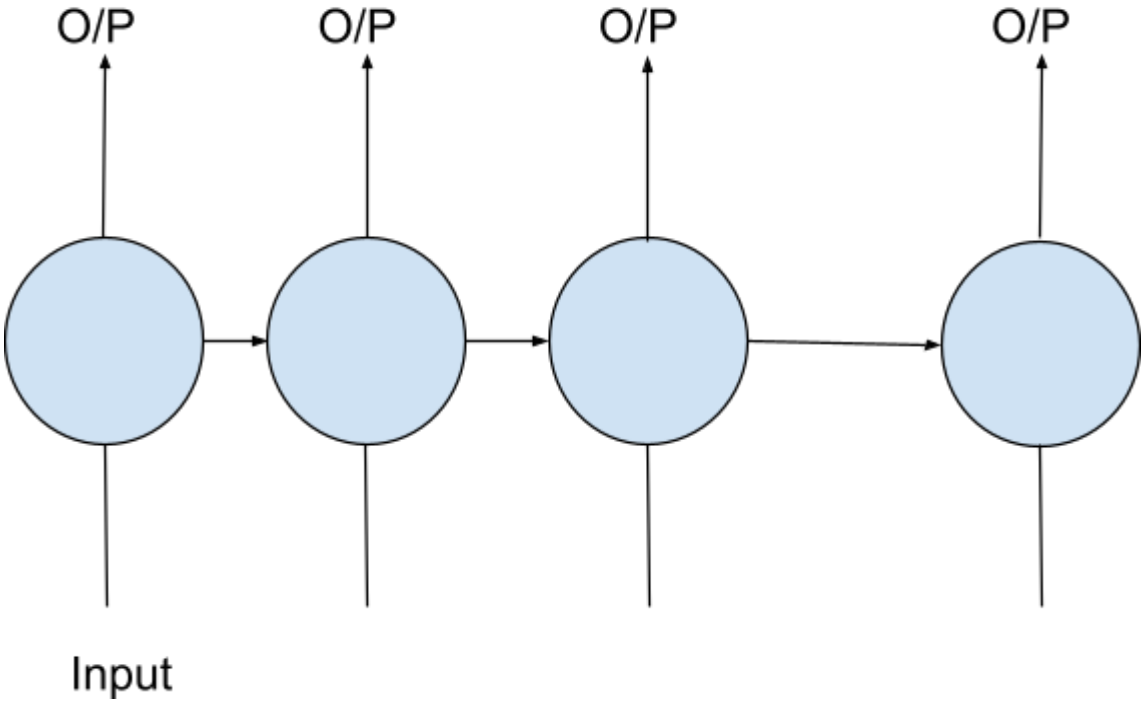
Of course, input will be converted into vectors of 300 dimensions(**Features**). Also weights will not be updated as it's just a forward propagation.

There are multiple types of RNN. 1) One to one RNN, 2) One to many RNN, 3) Many to one RNN and 4) Many to Many RNN.



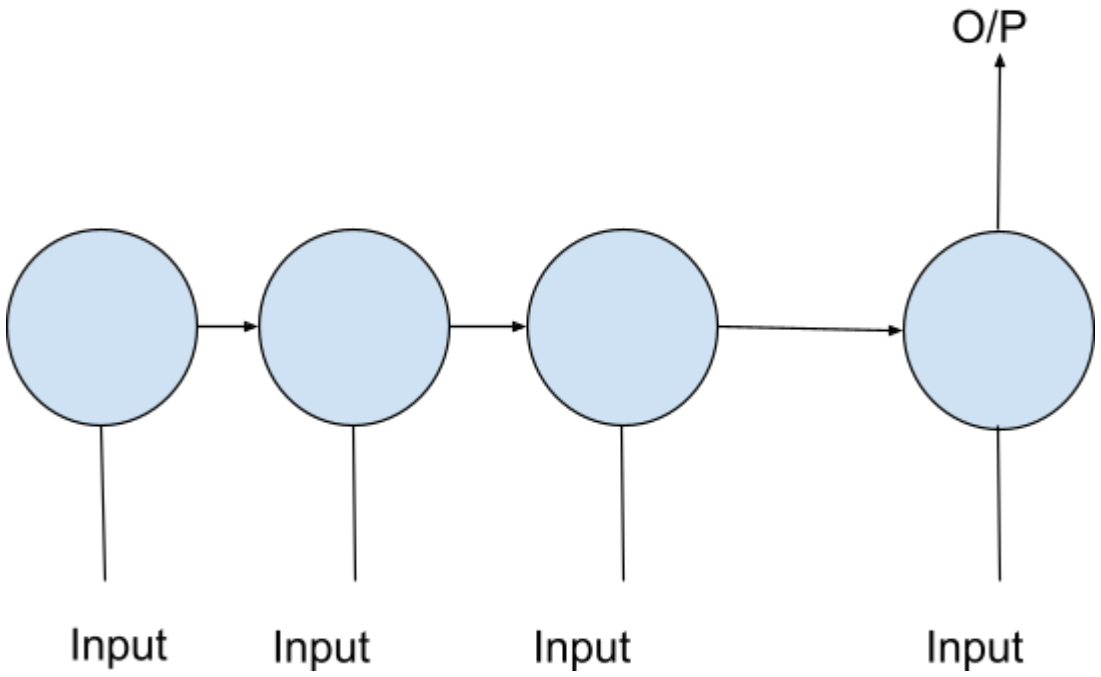
One to One RNN

In this type of RNN, Once input is given, we don't get output from each neuron, instead we pass it into another neuron. For example, Image classification



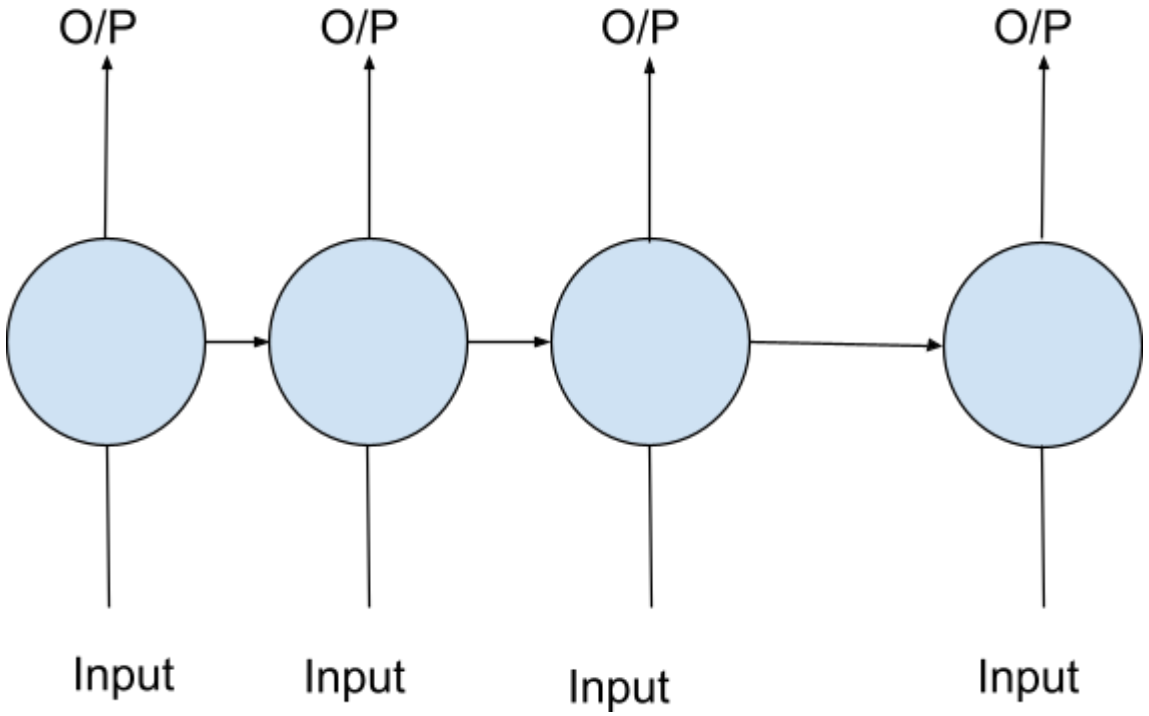
One to many RNN

For Example, Text generation, music generation, google search suggestion, movie recommendation system.



Many to one RNN

For example, Sentiment analysis



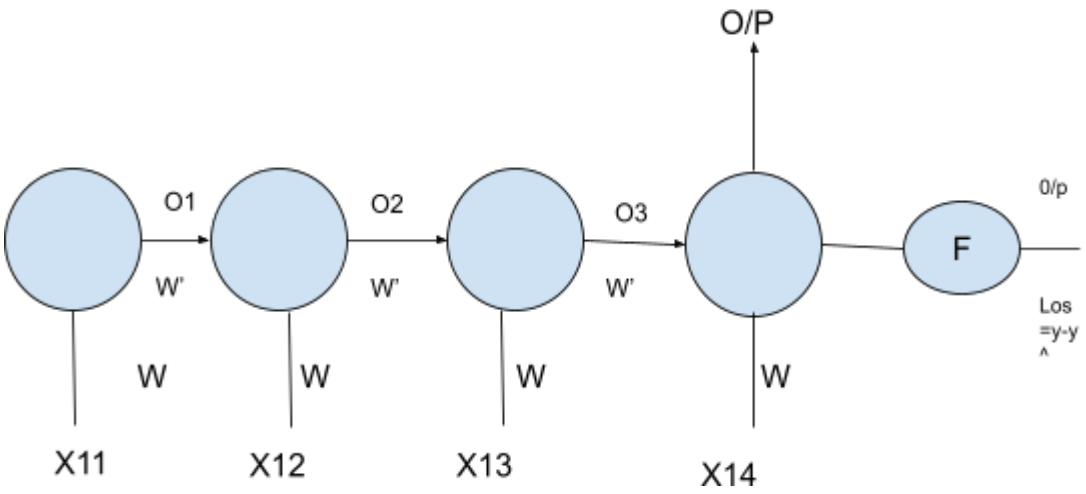
Many to Many RNN

For Example, Language translation, question answer, Chatbot

We use RNN with time series data

1) Forward Propagation in RNN

Let’s take many to one Rnn(sentiment analysis) example over here.



Here in above image final output is given to the activation function which is denoted by F.

So how we get **O1 ?**

Here consider neuron as a function so **O1 =** f (X11 * W) + bias, then we find **O2 And for it** O2 = f(X12*W + **O1 * W’**) + bias.

Since this is Many to one means there will be one output in the end so we apply **Sigmoid** activation function, while if there are multiple output,we will use **Softmax** activation function. After then, we find the loss function, **loss = (y - y^)**.

2) Back propagation in RNN

We know the formula of updating the back propagation which is,
W’new = W’old - n dL/ dWold , **Where** dL / dW’old = dL/dy^ * dy^/dW’old
Wnew = Wold - n dL/dWold , **where** dL/dy^ * dy^/dO4 * dO4/dWold

Thats how back propagation works, and perform some epochs with early stopping if want. Also when loss reaches to stagnant.

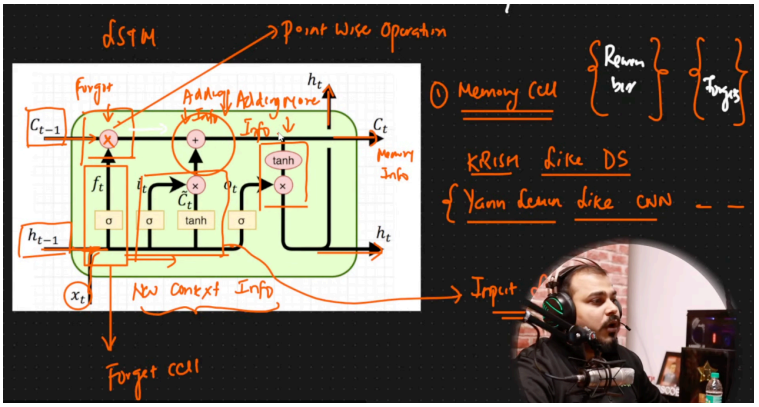
In Rnn, we might face issue when we have deep RNN which is vanishing gradient problem.

Let's say we use sigmoid activation function and the output is between 0 and 1, while derivative of sigmoid is between 0 and 0.25, So at some point , weight will not be updation, however it is but value will be minimal.Also, relu can actually dead neuron during back propagation.∂

So to solve above problem we use **LSTM RNN.**

LSTM RNN (long short-term memory RNN**)**

There are some important parts in LSTM Rnn which are Memory cell, Forget cell, Input cell. Also, sometimes some words are dependent on other words such as **"my name is Anuj and I like cars"**, so here "cars" is dependent on "like". Whenever we have deep neural RNN, It will not be able to capture the context properly, So that is the reason why we use LSTM RNN.



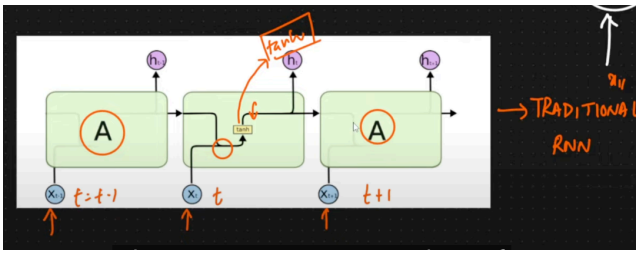
In above figure, **X means** point wise cross operation, while **+ means** point wise Plus operation. This X cell is a Memory **cell, it** indicates that you need to remember some things and forget things, and between **X cell and plus cell** ,it says how many things you need to remember and how many things you need to forget. **Plus operation** will add more information. New context info(**X**t) will be combined at **ht-1.**

Let's say in the previous cell the sentence is "Krish like cs", but the sentence is "Anuj like cnn" in the next cell is changed, now if i want to predict new words, it will be related to "Anuj like cnn" in this sentence.

Why **LSTM RNN** Instead of RNN ?

1) Vanishing Gradient or Dead Neuron
2) Context info , Info gap will be huge in LSTM RNN.

LSTM RNN - In Depth Architecture

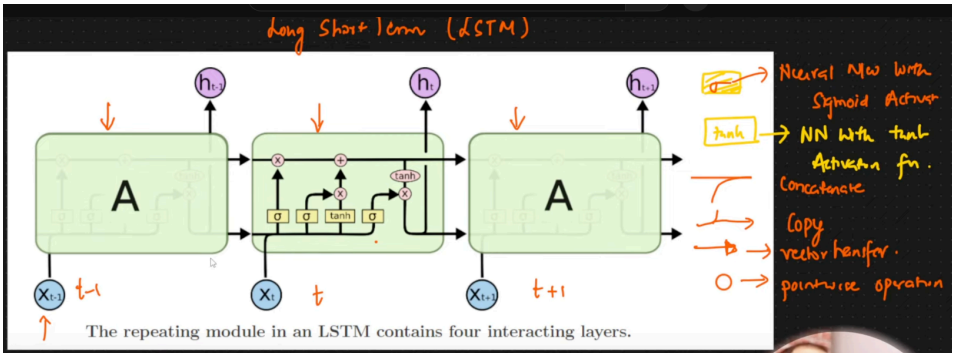


In the above image, X is the input and **tanh** is the activation function that will apply after concatenation. The same things will happen in other blocks,So this is the **Traditional RNN.**

There are two important reasons why we use LSTM-RNN ?

1) Long sentence - If there is a dependency of the output with the first word and we really want to predict the first word. in these sentences, Context regularly will be switching.

   Understand all the notations from the below image for LSTM-RNN.



   Now, How LSTM will ensure that the longer context information is also kept in mind and it also have focus point wherein it should be able to context switching.

   Let's take an example, "Krish likes to eat pizza but my friend likes to eat burger" , we need to predict burger here.

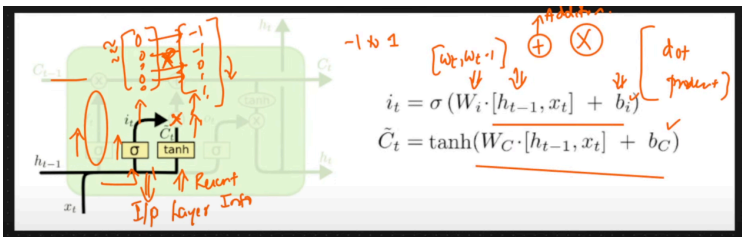
So what will happen is that each will go one by one,and after the ‘but’ goes, context will be switched to friend then **should LSTM-RNN need to remember the past words which are ” krish likes to eat pizza but his”?** LSTM-RNN will forget these words.

How do we forget info?

In the memory cell, we can add information and remove information.

In forget layer cell, our input X1 is friend and another input coming from past neuron will be passed through sigmoid layer and sigmoid activation layer will be trained such a way that for mos**t of the inf**o, **we’ll get zero, If** previous info and current info **are similar** or having the **same context** most values are **near to 1.**

If we get zeros, making the memory cell **forget** info as **forward** we have **point wise operation where** we do multiply with values like zeros, eventually our output will be zero so that info will be forgotten**.**

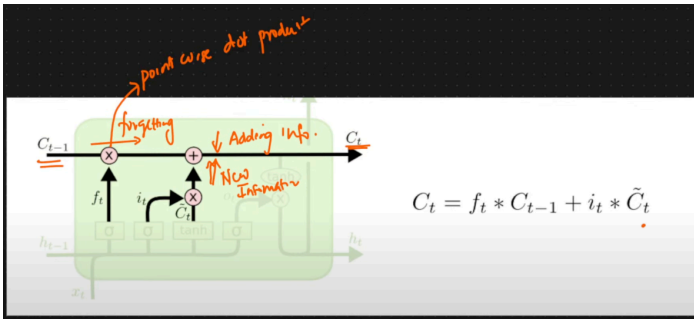


Let’s take a look in above image, there are Two sentences, in the first one, there is **no context change, so input that coming from previous neuron** and current info will be quite similar so output **will be nearer to 1,** while in the second sentence, There is **a context switching**, **so outputs will be nearer to 0.**

The formula we have in the above image, **ht-1** is the previous info while **Xt** is new info. **W**f is the combined weight of **ht-1** and **Xt.**

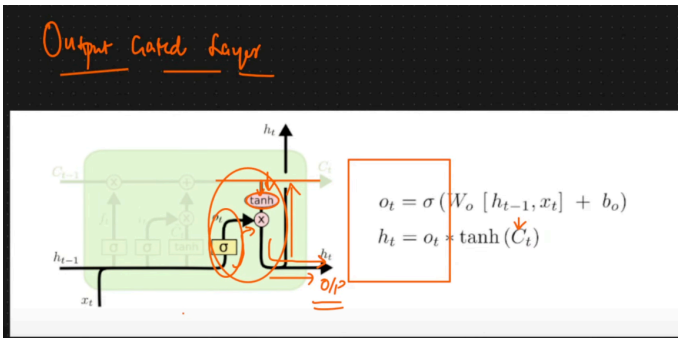
Now, let’s understand **Input gate layer**, we take a previous example,
“**Krish likes to eat pizza but his friend likes to eat burger**”

Here in above sentence, previous info will be removed in forgot layer and as we go ahead we really need to store friend’s information in memory cell which is symboled with ⊕**,** in order to add in memory cell, we pass it through sigmoid activation function and **Tanh** function and we know the value range between -1 to 1 in **tanh. Whe**n we combined both then that new context information will be passed.



This is a point wise operation, where first we forget some info then add new info and we get **Ct** from **Ct-1.**

Now, let's see **output gate layer.**

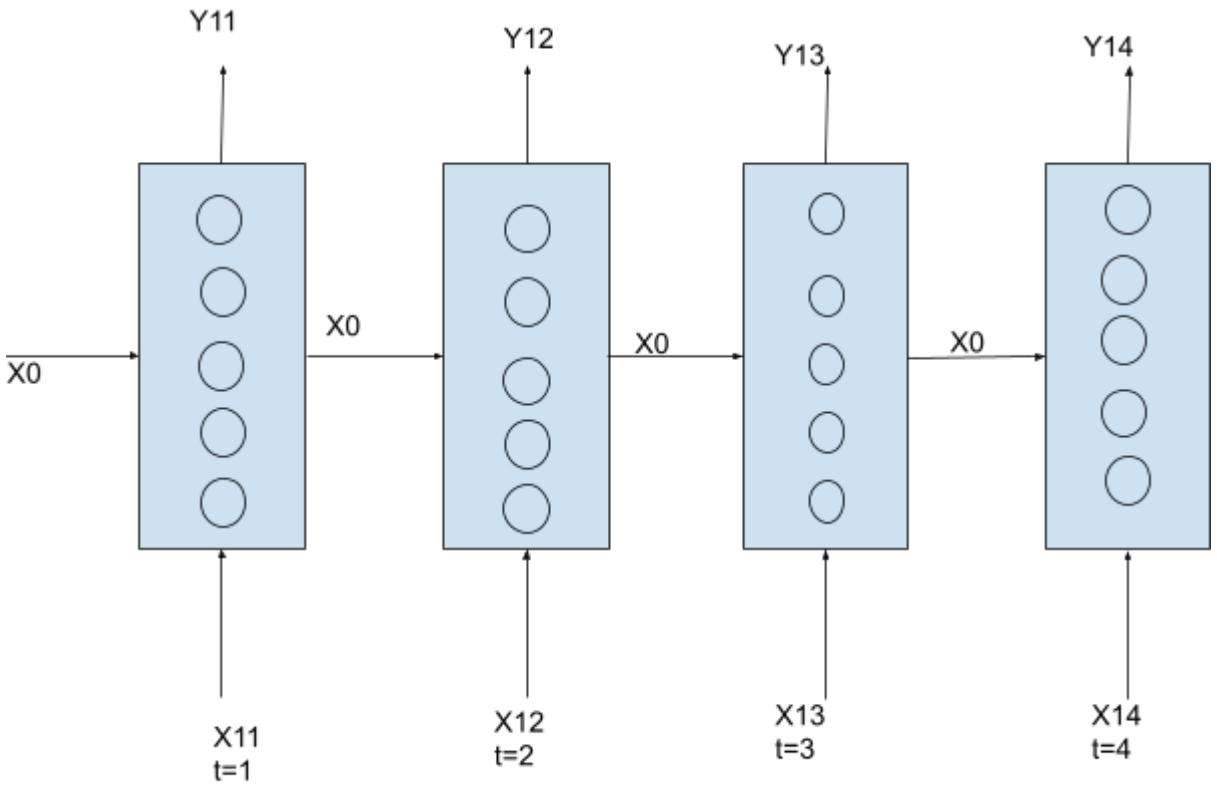


As we can see in above image, we apply **tanh on the output of memory cell where we forgot and add info, then we combine both previous info and recent info with dot product and get output which pass to the next cell and so on.**

Bi-Directional LSTM-RNN

Let's say we have a task that predicts the next word, **f.g, Krish likes to eat ____ in Bangalore**

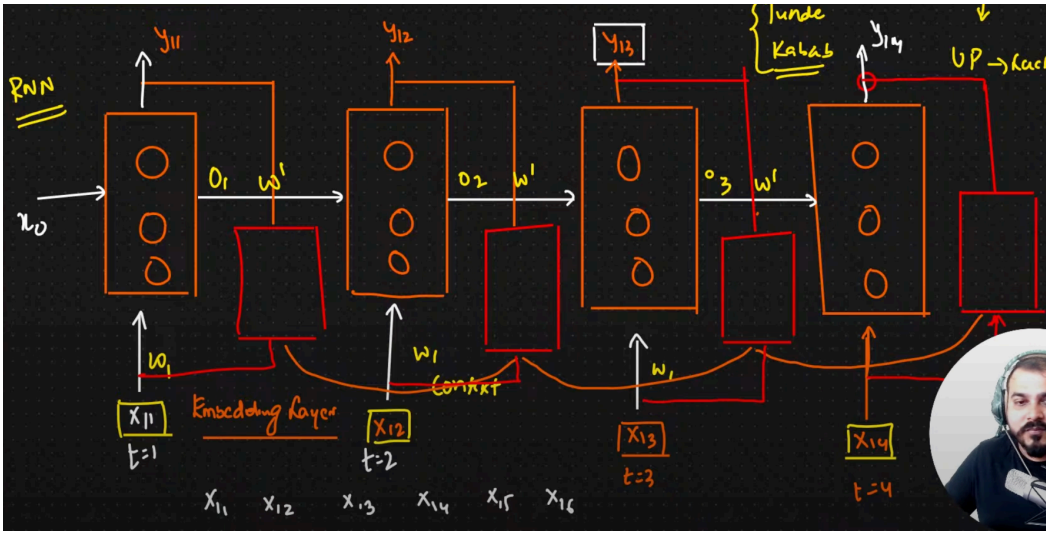
Let's say we have some words such as X11,X12,X13,X14,X15,X16



Consider Y13 is a word that we need to predict so Y13 will have context of X13, X12,X11. Y13 won't have any idea that what is X14. How can we provide forward words to y13. **That's why we use Bi-directional LSTM RNN.**

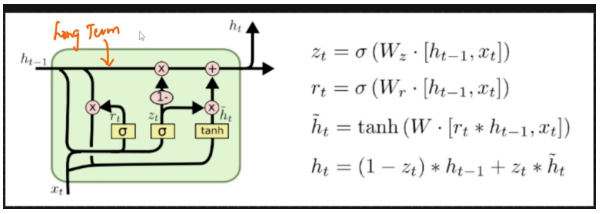
We use reverse RNN, we are sending text from reverse order also, where all small RNN will be connected with each other ,and inputs and outputs of big RNN as per the below picture.

Now. as we go with Y13, not only it has context of backward words but also forward words too.


RNN
y_{11}
y_{12}
y_{13}
y_{14}
Kabab
x_0
o_1
o_2
o_3
w'
w_1
context
Embedding layer
x_{11}
x_{12}
x_{13}
x_{14}
t=1
t=2
t=3
t=4
x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}

GRU RNN - Gated Recurrent Unit **RNN**

The LSTM RNN has a complex architecture as it has three Gates such as **Forget, input + candidate memory, and Output Gates.** As we know these gates will have their weights such as [Wf, Wi, Wo] along with bias , so there are the trainable parameters. Because of the **Complex Architeture,** trainable parameters will be increased and when it happens **training time will also increase.**



So what does GRU do? Is that instead of using long-term Memory and short term memory, it has combined in GRU as seen in above image that Ht-1 is combination of short and long term memory.

Let's jump on the formulas, **Zt** is known as **Updated Gate** and it takes Xt and Ht-1 along with Its weight such as Wz, **Rt** is known as **Reset Gate** and it takes same as Zt but weight will be Wr. **h`t** is known as **Temporary hidden state.**

After calculating Rt we do point-wise operation with Ht-1, once we do this we pass it to Tanh along with Xt , this is where we get **Temporary hidden state,** then we do again point-wise operation with **Zt.**

How we get **Ht?** First we subtract 1 from Zt then point wise operation with Ht-1 after this, again point wise operation with Zt and h`t , this we do plus wise operation between these two final values then We get **Ht.**

What does Rt do?

It helps to reset Xt values. For example, ht-1 = [0.6 0.5 0.3 0.9] , and rt = [0.2 0.4 0.8 0.2] then we do Point wise operation in order to reset Xt. So new Resetd Xt looks like [0.12 0.2 0.24 0.18] , we reset only 0.2 (20%) from 0.6 so thats 0.12.

What Zt says that **what context needs to be added ,** but this dependent on the candidate hidden state and candidate hidden state talks about **Current Context**
And if current info is important then we add it with less information from **Zt.**

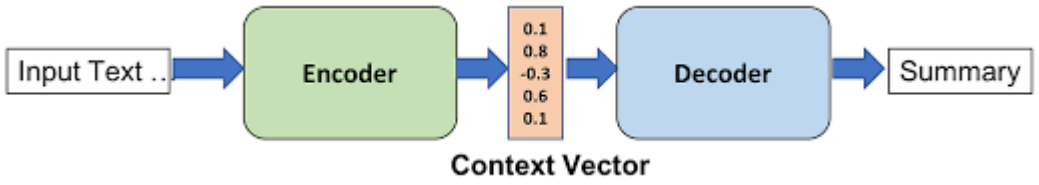
Encoders & Decoders

So far, we have discussed Simple, Lstm, GRu, and Bi-directional RNN.
What was the problem in simple Rnn that **Vanishing gradient problem**, we saw different RNNs to solve such as LSTM, and GRU RNN and by using these two techniques we are able to solve problem such as **many to one** like **sentiment**, **predicting next word**. **The problem with LSTM and GRU** that if there is a dependency of further words then we use **bi-directional RNN.**

Let's talk about **Many to Many RNN,** it means we have many inputs and many outputs for example **language Translation. Also** this type of RNN called **Sequence to sequence neural network** which means we give inputs in sequence and get outputs in sequence.

For such cases, we can't use other RNNs, we can use only **Encoders and Decoders.**

When we give input sentences to encoders through **embedding layers**, the encoder converts sentences into arrays of numbers, then we **create hidden states** its also known as **context vectors** then it's forwarded to Decoders and the main aim of decoder is to generate output words by word and keep feeding the previous word into the decoder again.



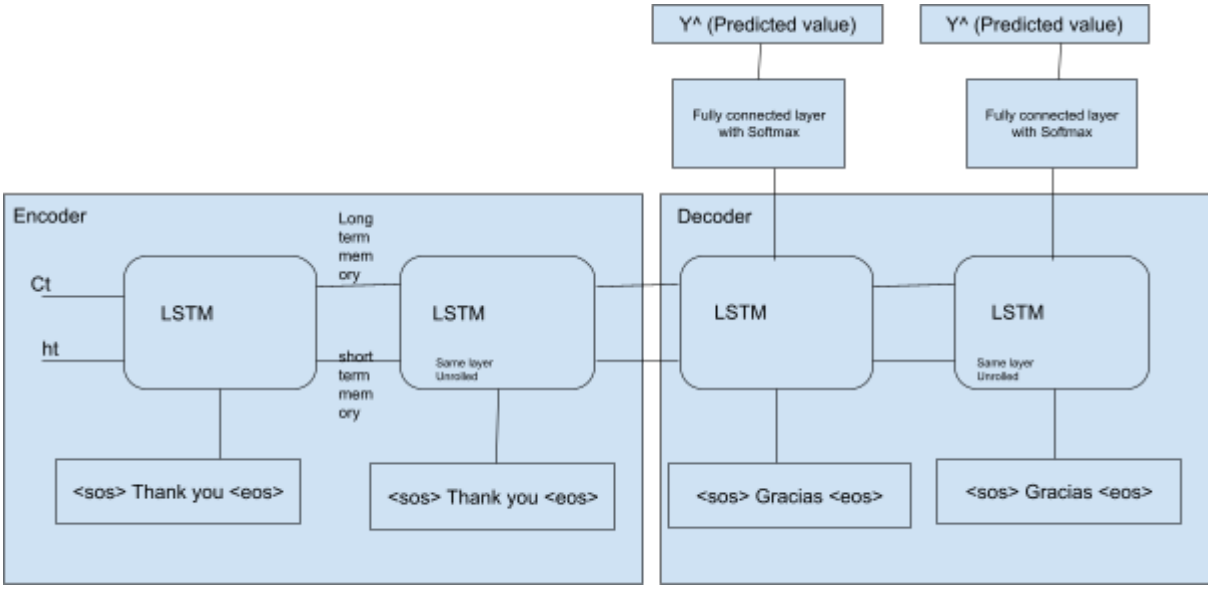
1) Encoders => I/P => Context vector
2) Decoders => Context vectores => O/P

So, what exactly happens?

The encoder takes **Inputs,** converts them into **Context vectors,**
The decoder takes Context vectors and gives **output into input form.**

There are some examples, such as **translation, text generation, and text suggestions.**

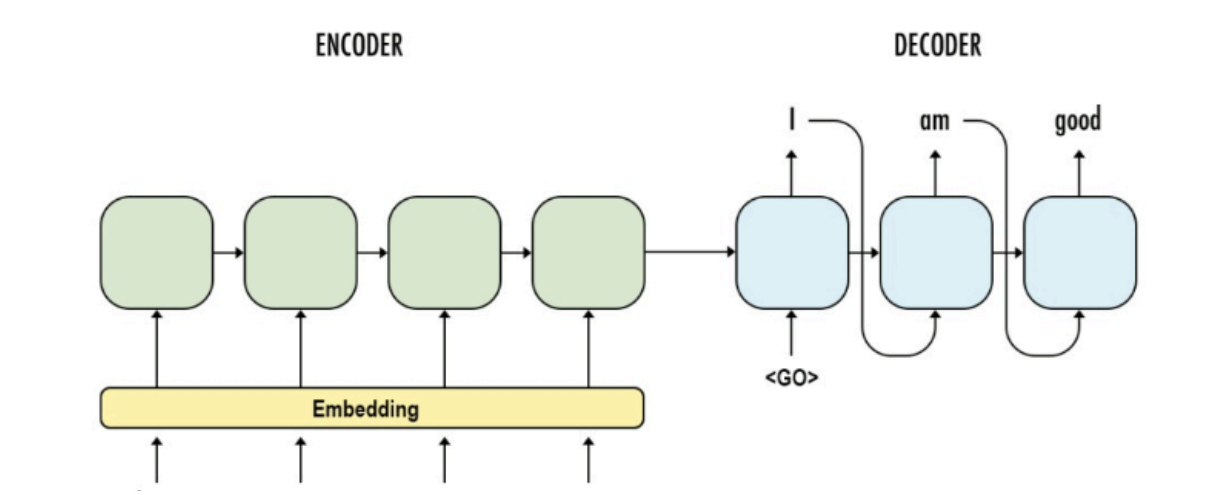
In the encoder and decoder, we use **LSTM RNN** and **GRU RNN,** The **reason** we don't use **Simple RNN is** that it has **Gradient vanishing Problem.**



From **C0** , we pass that what context needs to be added into longer memory cell and what context needs to be removed from the cell. From **ht** what needs to be retained in short term memory which called **Hidden states.**

Let's say in our dataset there are two columns English and French(**Output**) , " Thank you" "Gracias" are values for columns respectively **,** So in encoder, i will pass " Thank you" along with **special characters**
Like **<sos> (**start of sentence**)** and **<eos> (**end of sentence**).**

Likewise with the french word which is "Gracias", we add sos and eos, it derives for neural networks that this is start of a sentences and end of a sentence.



As seen in above image, we pass each words one by one in **LSTM RNN** after using **ONE-HOT encoding technique**, in our case there are **four** words so we pass first **SOS , Thank , you, EOS** , after that when we **combine the long term memory and short term memory outputs** we get c**ontext vectors** in the end of encoder.

Now, in our training data, There are three words for output which are <sos>, Gracias, <eos>. Once we get the **output** we pass it to the **fully connected layer with softmax activation function** as seen in the **diagram of previous page.** With respect to softmax activation function i will be getting three outputs which is **denoted** by **Y^.**

In our training dataset there **three** words in output column we pass each one by one firstly <sos> ,we get three probabilites(**Outputs**) from **softmax** activation function which would be liked **[0.1, 0.6, 0.3],** and we know that softmax works by doing probability now **0.6** is highest so as per the **data** which were **[<sos> , Gracias , <eos>]** , we get **Gracias.** In different scenario**,** what if those **three probability** would be **[0.1 0.3 0.6]** then output would be **<eos>** ,it means we are **not moving further** as it's the **end of sentence** but **backward-propagation** will start and **updates weights** and again do **forward-propagation** , ensure this time we get **lower** difference of **[y-y^].**Also, first we pass **all inputs** in **encoder, then** the output from the last layer or output from the last LSTM with respect to timestamp that context vector gets passed to the decoder.

Now, the output was "**Gracias**", we pass this output to next **LSTM** along with the **context vectors** and after all process we might get **output <EOS>** as we don't have any words after **Gracias. Once** we get EOS , we stop training further.

We will have **Y^** and this will be with respect to timestamp like t=1,2,3 and for each timestamp we get y and y^, we find loss function **(y-y^)^2** after

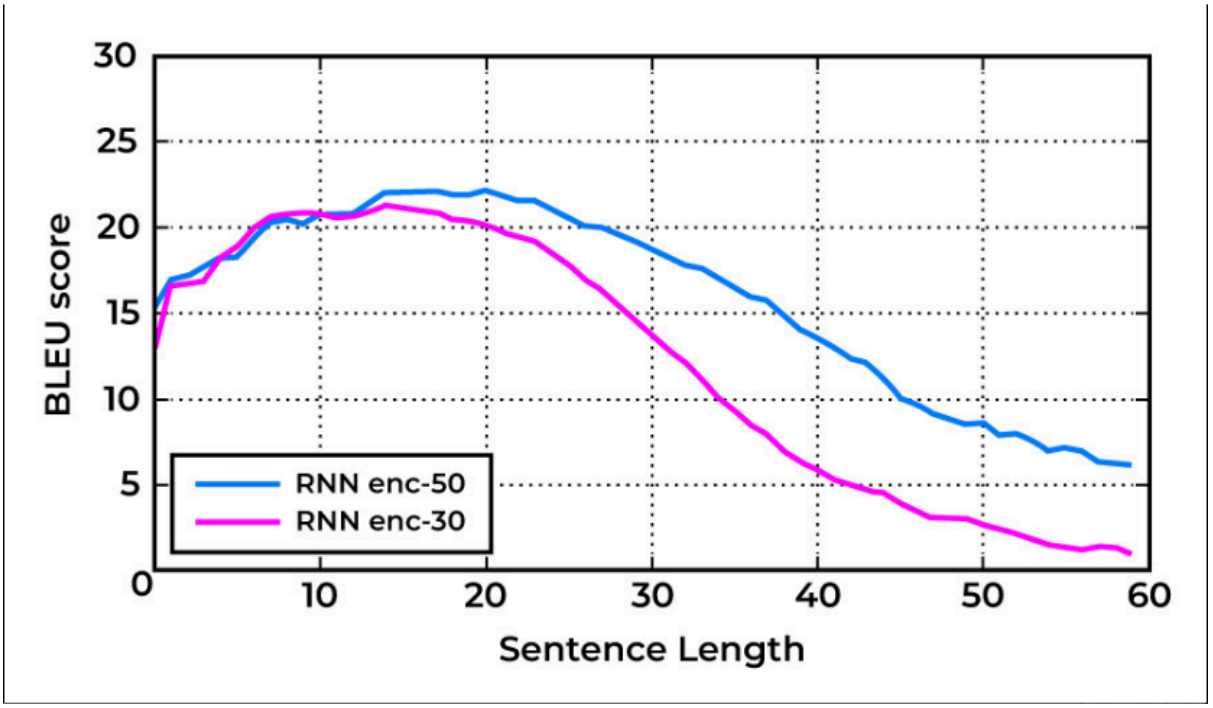
forward-Propagation and try to get lower this loss function. In order to reduce it we use **Optimizer.** It will update all weights ,backward propagation will do then again forward propagation.

So this way the entire training of a sentence happens.

So The important is **how we convert inputs into vectors?** Before sending it to LSTM, we send it into embedding layer for both encoder and decoder.

Problems with Encoder and Decoder seq2seq Architecture ?

During the research of encdoer and decoder, they used performance metrics that research paper was related to bleu score and sentences length. They got below graph



So, as per the graph, as sentence length increasing after a certain point **, Accuracy will be decreasing , bleu score started decreasing.**

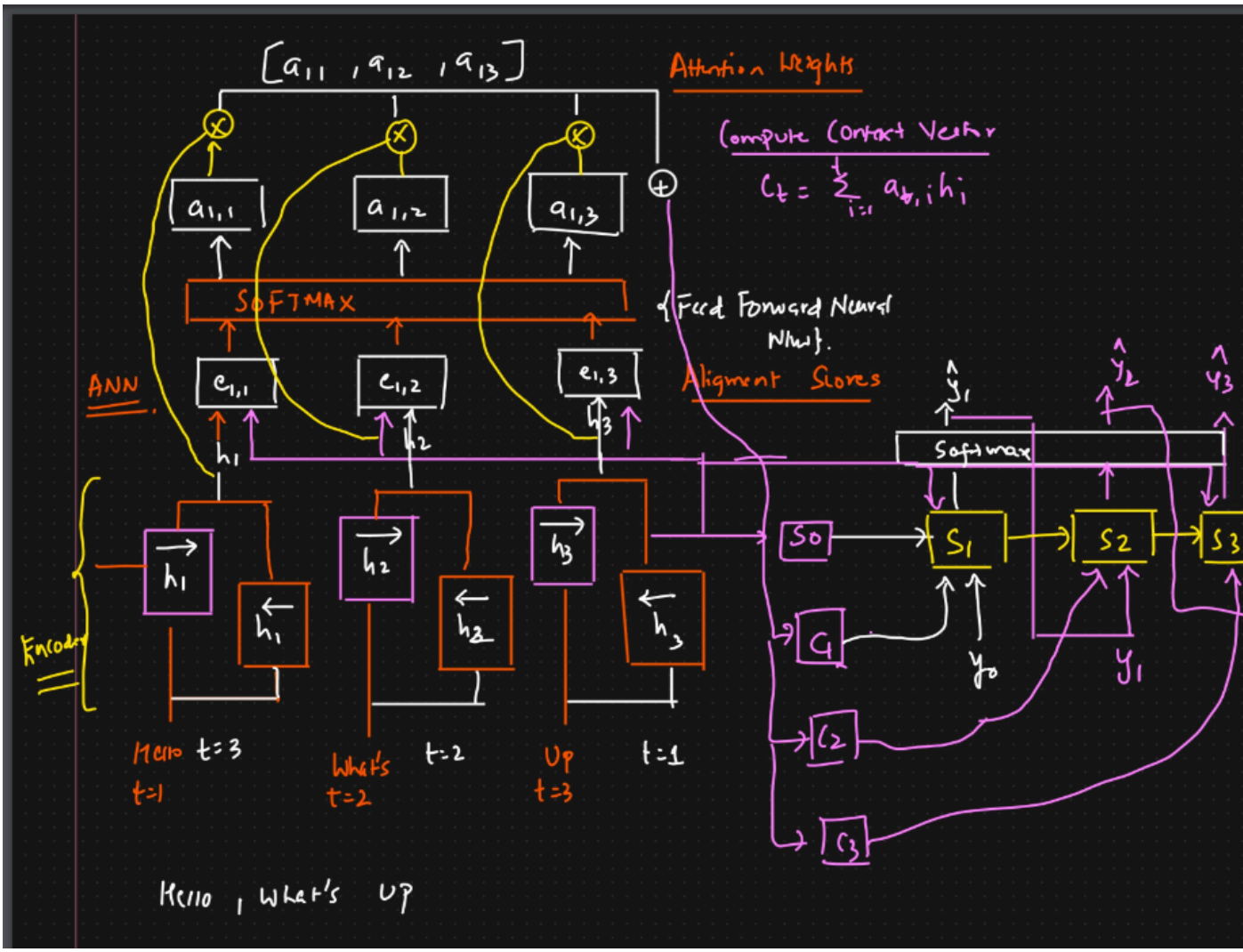
The reason is that from encoder we are not taking all outputs from LSTM, instead we take the last LSTM output from encoder, whatever context vector it is basically creating. This context vector has more info with the nearest word that just passed in a particular timestamp. Let say there are 100 timestamp, t=100 so context vector have more info that passed in t=100 and **less info** from t=1,2,3 and so on.

In order to solve this problem, **We use Attention Mechanism.**

Attention Mechanism📎

For longer or shorter sentences, along with context vector and we will go ahead and provide more context. This is the main idea behind it.

The main idea is to provide more context to Decoder from not only taking last lstm output but also consider all lstm outptuts and make context vector that will pass to dcoder for preediction. Instead of simple LSTM, we use Bi-directional LSTM in encoder so we can provide further words context.



Now, we will create some notations and in this notations we will pass our **h** along with s0(hidden state) as shown in above image.

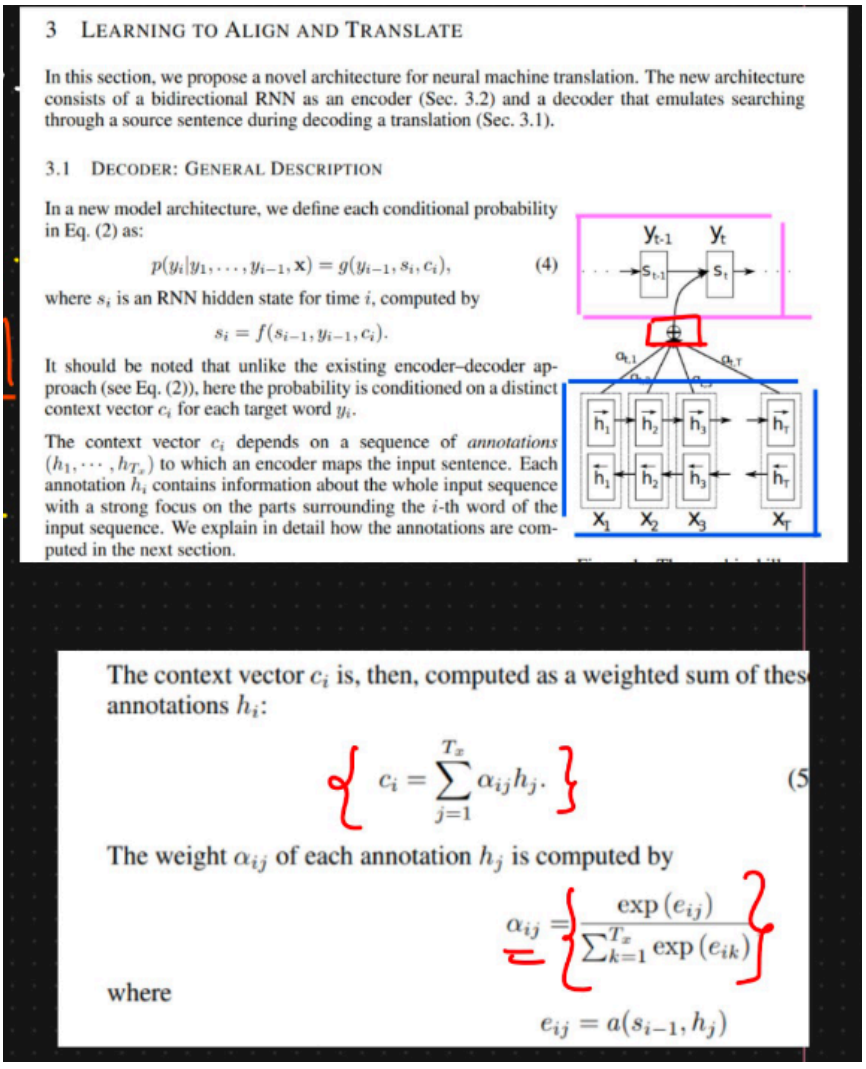
With respect to words this is e11 , e12 , e13 explain it in later. See it in below image we get it with the help by sending hidden state and h values in activation function.

Then we are going to pass it to **Softmax** that means we are going to train it with ANN. So, here we create a feed forward neural network, Inside this neural network,we will give combination of **h** and **s0** then we train it with ANN, these are called **alignment scores**, and pass that output to softmax which becomes **Multi-classification problem statement.** After that, we get some nodes which are A11 , A12, A13 as per the above image. Then we create **attention weights** is all about how much context we really need to give it to the **decoder** in order to do specific **prediction.**

What we do now that whatever **output** we got from **h1**, we **combine** it with the output of **A11** then we do **pointwise operation** as per the above image.

The importance of **A11, A12, A13** is that it basically used in **deciding How much context of h1,h2,h3 i should consider**. We do **plusewise operation** with the **outputs of all pointwise operation**. And here we will **compute our Context vector.**

The formula of Context vector is C_t = summation of t from i =1 to t (a_{ti} . h_i)



3 LEARNING TO ALIGN AND TRANSLATE

In this section, we propose a novel architecture for neural machine translation. The new architecture consists of a bidirectional RNN as an encoder (Sec. 3.2) and a decoder that emulates searching through a source sentence during decoding a translation (Sec. 3.1).

3.1 DECODER: GENERAL DESCRIPTION

In a new model architecture, we define each conditional probability in Eq. (2) as:

$$p(y_i|y_1,\dots,y_{i-1},\mathbf{x}) = g(y_{i-1},s_i,c_i), \quad (4)$$

where s_i is an RNN hidden state for time i, computed by

$$s_i = f(s_{i-1},y_{i-1},c_i).$$

It should be noted that unlike the existing encoder–decoder approach (see Eq. (2)), here the probability is conditioned on a distinct context vector c_i for each target word y_i.

The context vector c_i depends on a sequence of *annotations* $(h_1,\cdots,h_{T_x})$ to which an encoder maps the input sentence. Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence. We explain in detail how the annotations are computed in the next section.

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \quad (5$$

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$$

where

$$e_{ij} = a(s_{i-1},h_j)$$

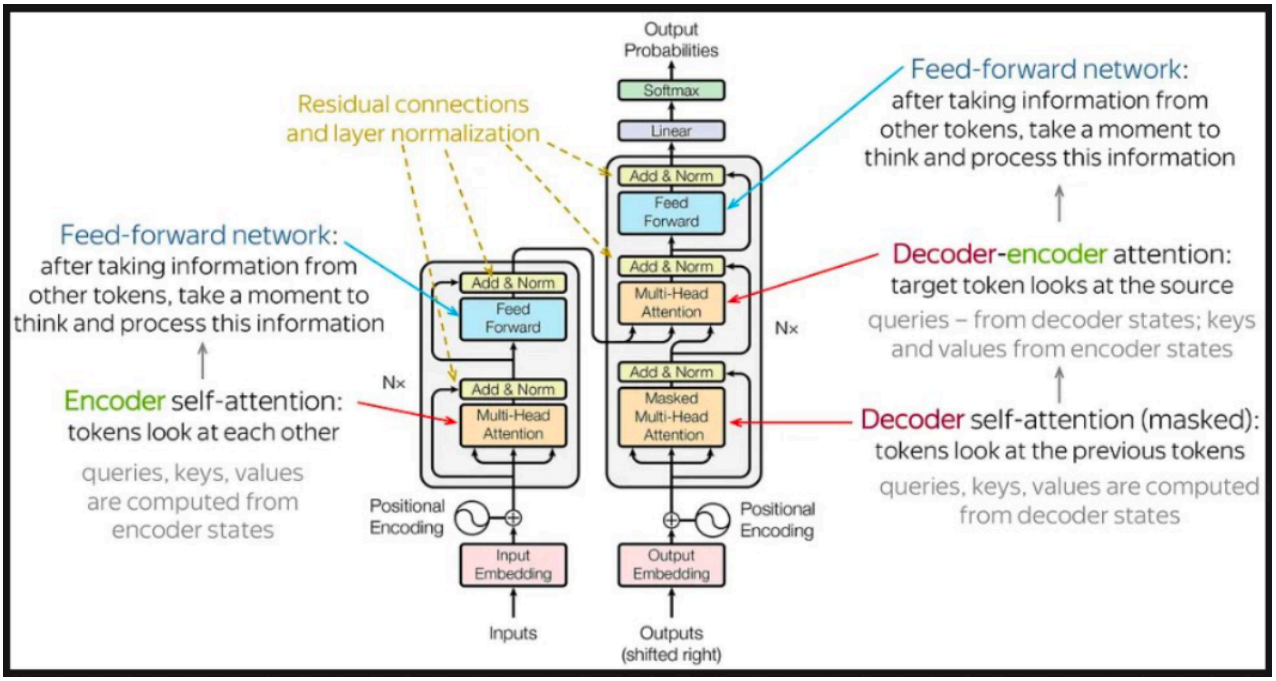
Now we pass **context vector** that we got after **Pluse wise operation** from Encoder to **decoder** along with that **S0 (Hidden state)** and **y0 (Truth value) then** pass it to softmax and we get Y1^ .

What happened in next step ? i am getting information of S0 to S1 , now for S2
Again we have to generate Context vector. For that instead of passing just from S0 we will remove line from S0 that we send it to get eij for attention score and and pass it to S1 for generating context vector and it will pass to S2 along with the output of S1 which is Y1^ and then we calculate Y2^ this happening for every timestamp.

Transformers

What is transformers? - Transformers in Natural language processing (NLP) are a type of deep learning model that uses **Self-attention mechanisms** to analyze and process natural language data. They are encoder-decoder models that can be used for many applications, including **the machine translation task,** which means sequence-2-sequence task. For example, language translation.

One problem we face in the Attention mechanism is that parallelly we can't send all words in sentences that's why it isn't scalable. When a dataset is huge, the encoder-decoder attention mechanism isn't scalable with respect to **training.**
When it comes to transformers, they **never use LSTM RNN** in the encoder and decoder. Instead of LSTM, they use **a self-attention module; because of this,** all words are sent parallel to the encoder.



Why transformers performed very well is that as we keep an increased dataset we get amazing state of the Art (SOTA) models. **Even Transformers are used** in multi-model task which means task that has both NLP + image.

One more problem we face in transformers that is respect to **contextual embedding.**
E.g: My name is Krish and i Play cricket.

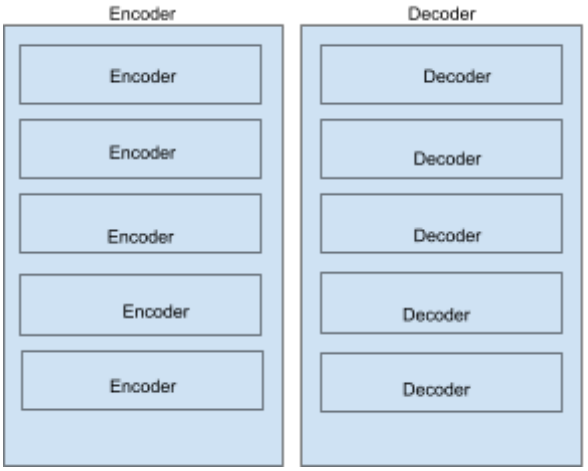
Now we pass above sentence to embedding layer from there we get **Vectors for each words.** Let say we use Word2Vec embedding technique.

Now what is the contextual vectors? It means we should get vectors whenever we have longer sentence, based on the relationship with other words. For example, in above sentence, **I** is obviously related with **Krish** , So we give same words to **Contextual vector embedding** then there will be some relation in vectors with respect to other words such as between I and Krish.

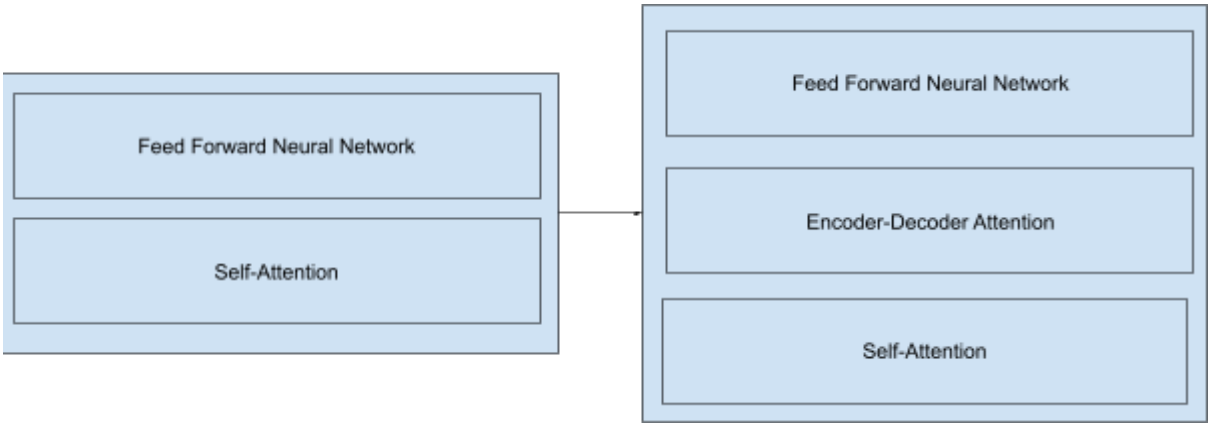
This entire Problem is solved by **Self Attention** because of this, transformers are so **accurate.**

Let's understand Transformers

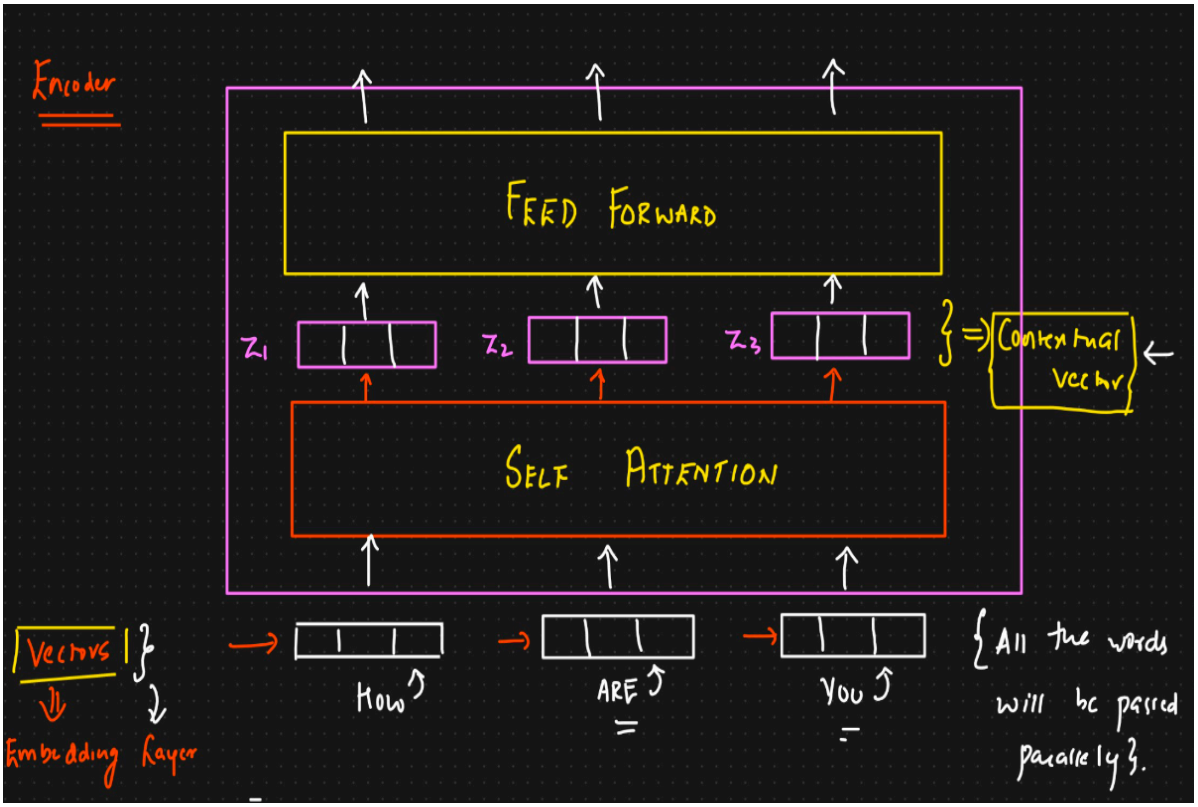
Transformers follows Encoder - decoder Architecture and in encoder there are multiple encoders. One input pass to one encoder to another encoder. Likewise for decoder too. Refer to below diagram.



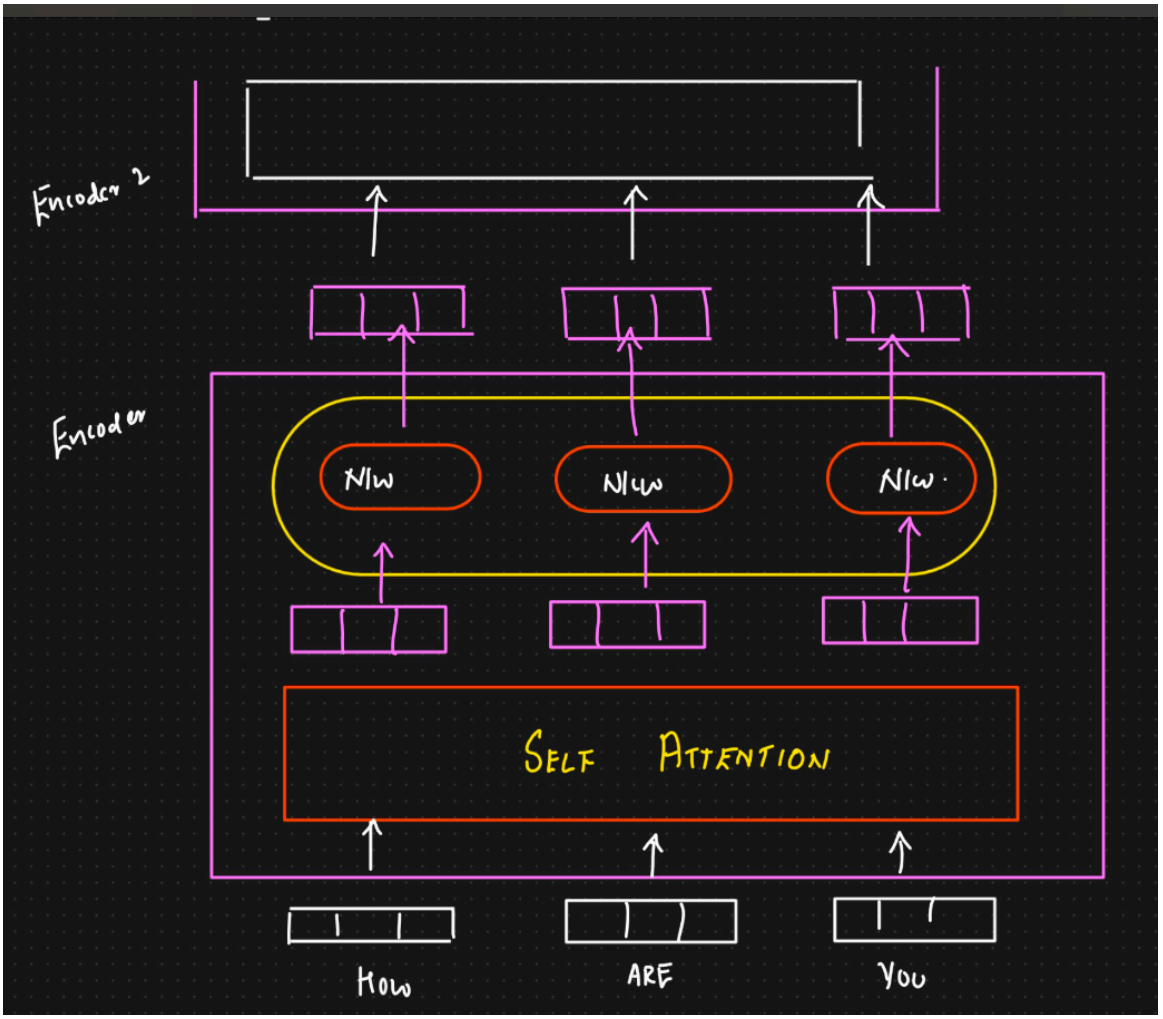
Let see what is inside a encoder,



There are two things inside a encoder firstly, **self-attention** and second **a feed forward neural network**, while in decoder there are these **both things along with encoder- decoder attention.**



As per the above image, **words** will be converted into **vectors** through **embedding layers and pass** it to **self attention** from where we will get **another vectors** which are known as **Contextual vectors. These** contextual vectors will be passed to Feed Forward and we will get vectors as an outputs and these output vectors are sent to **another encoder** as **input vectors.** Take a look in below image.



Self-Attention At higher level and Detailed Level

Self Attention, also known as **Scaled dot-product attention**, is a crucial mechanism in the transformer architecture that allows the model to **weigh the importance of different tokens** in the **input sequence** to **relative to each other.**

For example, We have sentence “The Cat sat “ , now we pass it to self-attention through Embedding layer. Now when we get contextual embedding vectors as outputs from self-attention and these vectors are not fixed like the first embedding vectors that were passed into self-attention, instead contextual embedding vectors will also have an importance based on other words. Moreover, getting **Contextual vectors** from **fixed vectros** is **dependent** on the **sentence or dataset.**

The reason we do this as with respect to different sentences and paragraphs will have different context, one word will dependent on other words and we are able to build application such as **Language translation, Text summarization.**

Let's see how Self-attention is going to convert vectors into Contextual Vectors.

1) **Inputs**: FIrst we need to create three vectors such as **Q**(Queries), **K**(Keys), **V**(Values)

   For every words we create a model to compute Queries,keys, and values for converting into Contextual vectors.

   Q - Query vector represent the token for which we are calculating attention. For example, we are **calculating attention** which is **contextual vector** for the passing word "**The**". So "The" is **Query Vector. Also,** it helps to decide the importance of other tokens **(Words)** in the context of **current token (The).**

   There are **two important** things in **Query.**

   1) **Focus Determination: Queries** help the **model decide** which **part of a sequence** to focus on for each **specific token(Word)**. By Calculating the dot product between **a query vector** and **all key vectors**, the **model decide** how much attention to give to each token relative to current token. For example, there are three words the, cat, sat, so "**The**" will be Query vector and we do dot-product with all key vectors and then we get final vector for "**The**".

   2) **Contextual Understanding:** Queries Contribute to understanding the relationship between the current token and the rest of the sequence.

2) **Key Vectors:** it represents all the tokens in the sequence and are used to compare with the query vectors to get **attention score.**

   There are two major things in **Key vectors.**

   a) **Relevance Measurement:** As we know keys are compared with query vectors. **This comparison** helps to decide **how much attention** each token(word) should **receive**.

   b) **Information Retrieval:** Keys play a critical role in retrieving the most relevant information from the sequence by providing a basis of attention mechanism to compute similarity scores.

3) **Value Vectors:** Value vectors hold the actual information that will be aggregated to form the output of the attention mechanism.

   There are two crucial things in Value vectors.

   a) Information aggregation: Values contain the data that will be weighted by the attention score. This weighted sum of values forms the output of

the self-attention layer which will be passed to the next layers in the network.

b) Context preservation: By weighting the values according to the attention score, the model preserves and aggregates relevant context from the entire sequence, which is crucial for tasks like language translation, text summarization and more.

Let's take an example and see how we create **query, key, and value vectors.**

E.g = ["'The", "cat", "Sat"] and embedding size is 4 that means every words will convert into 4 vectors, also for **Q,K,V** we use dimension size 4.

First step is Token Embedding and it means whatever sentence we have, will convert it into some vectors. For example, E_{the} = [1 0 1 0] , E_{cat} = [0 1 0 1] , E_{sat} = [1 1 1 1] , this is how fixed tokens will represent for entire sentence.

Second step is Linear transformation, We create Q,K,V by **multiplying embeddings** by **learned weights** matrics **Wq , Wk ,and Wv.**

Let's say I have word cat and it represents in **vectors [1 0 0]** and then we do dot-product with **Wq** to get **Query** vectors, likewise we do dot-product with **Wk** and **Wv** to get **Key** and **value** vectors respectively. **Initially, we** assign some weights and later on with the help of **Backpropagation, weights needs to be learned to get key, query, value vectors.**

Let's do one example,we have fixed vectors E_{the} = [1 0 1 0] , E_{cat} = [0 1 0 1] , E_{sat} = [1 1 1 1] , now initialize weights

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

2 x 2

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3 x 3

Wq = Wk = Wv = Identity matrix

Now how we get Qthe , Kthe, and Vthe, for that we do dot-product with Embedding vector Ethe with this Identity matrix ,For example, Qthe = [1 0 1 0] . I , Likewise with Kthe, Vthe

Now for Qcat = Kcat = Vcat = [0 1 0 1] , Qsat = Ksat = Vsat = [1 1 1 1]

Now Third step is to Compute attention scores. Let's calculate score for "The" word ,So we the Query vector of The word and Key vector of that word also but we take transform of Key vector to do dot-product with QThe. Likewise we do with all key vectors such as Kcat, Ksat.

Score(Qthe, Kthe) = [1 0 1 0] . [1 0 1 0]T = 2,
Score(Qthe , Kcat) = [1 0 1 0] . [0 1 0 1]T = 0,
Score(Qthe , Ksat) = [1 0 1 0] . [1 1 1 1]T = 2,

For Cat

Score(Qcat, Kthe) = [0 1 0 1] . [1 0 1 0]T = 0,
Score(Qcat , Kcat) = [0 1 0 1] . [0 1 0 1]T = 2,
Score(Qcat , Ksat) = [0 1 0 1] . [1 1 1 1]T = 2,

For Sat

Score(Qsat, Kthe) = [1 1 1 1] . [1 0 1 0]T = 2,
Score(Qsat , Kcat) = [1 1 1 1] . [0 1 0 1]T = 2,
Score(Qsat , Ksat) = [1 1 1 1] . [1 1 1 1]T = 4

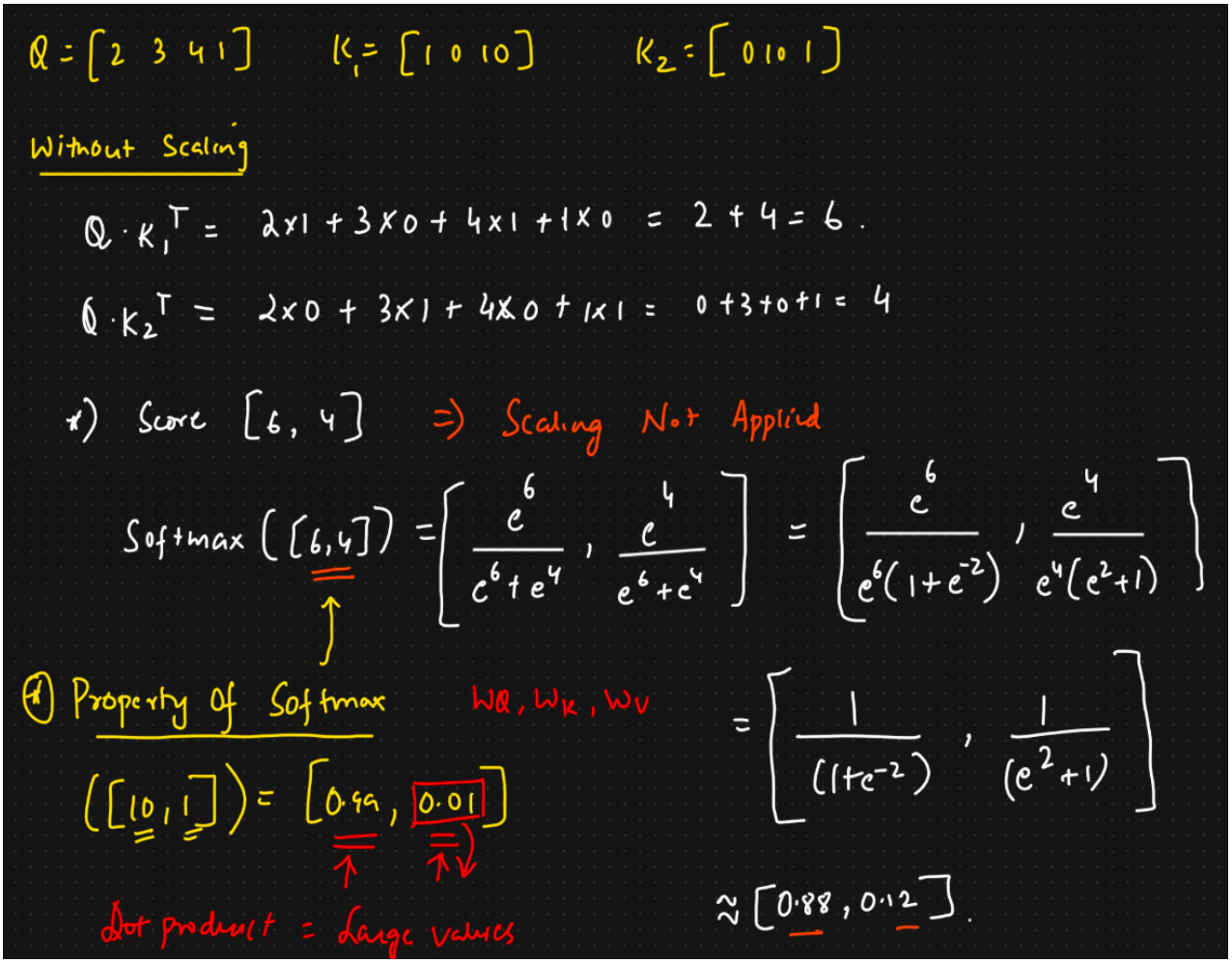
Forth step is Scaling , we take the scores and scale down by dividing the scores by the squared-root of dimension of key vectors. We took our $d_k = 4$ so squared-root of dimension of key is 2. **Scaling** in the attention mechanism is crucial to prevent the dot product from growing too large. Why we need to do this to ensure stable gradients during Training.

If dk is getting large, we face Two types of problems, 1) **Gradient Exploding -** during backpropagation it becomes large and because of this training will not be stable. 2) **Softmax saturated -** it means where most of attention weights assigned to single token and other tokens weights assign near to Zero.

Let see one Example, Q = [2 3 4 1] , K1 = [1 0 1 0] , K2 = [0 1 0 1]

Dot product **without Scaling,**
Q. K1^T = 2 .1 + 3.0 + 4.1 + 1.0 = 2 + 4 = 6
Q.K2^T = 2.0 + 3.1 + 4.0 + 1.1 = 3 +1 = 4

Score is [6, 4] now we apply softmax([6,4]) and we get nearly equal to [0.88 , 0.12] we can see calculation in below image.



What does the [0.88 , 0.12] values mean that most of attention weight is assigned to the first key vector , and very little to the second vector. So when we do backpropagation and weights will be updated then this small value don't have major impact to change weights value then we face problem like **Softmax saturation** where gradient will not updated more also said vanishing gradient problem. Also, this will be continue if dot product have very large values.

Dot Product **With scaling.**

First of all, we compute scaled dot product now attention score was [6 , 4] and if we want to scale it then we divide it with squared root of dk which is 2 , So [6/2 , 4/2] = [3 , 2]

Now , we apply **softmax to [3 , 2]** = [0.73 , 0.27] , there is less different than we got in **without scaling** , here we won't face **vanishing gradient problem** during **backpropagation**. [0.73 , 0.27] this is also called attention weights and these are **more balanced** as compared with unscaled attention weights.

Summary of Importance

Stablized training: Scaling prevets large dot products, which helps in stablizing the gradients during backpropagation, making the training process more stable and efficient.

Preventing Saturation: By scaling dot products, the softmax produces more balanced attention weights, prevents the model from focusing heavily on a single token and ignoring others.

Scaling ensures that the dot products are kept within a range that allows the softmax function to operate effectively, providing a more balanced distribution of attention weights.

Let's jump on our main sentence "The cat sat " , how do we scale here? We take first "The" ,Squared root of Dimension of Key (dk) = √4 = 2, see step 3 for how we got 2, 0 , 2 and then we use dk for scaling.
Scaled-Score(Qthe, Kthe) = 2 / 2 = 1
Scaled-Score(Qthe , Kcat) = 0 / 2 = 0
Scaled-Score(Qthe , Ksat) = 2 / 2 = 1

Likewise we scale "Cat" "sat" words,

Fifth step is to apply softmax. We use this function on scaled score to get attention weight of "The" word

attention weight of "The" = **Softmax([1 0 1])** , after this calculation we get these **vectors [0.4223 , 0.1554, 0.4223]**

For **Cat** and **sat** , it goes

attention weight of “cat” = **Softmax([0 1 1])** , after this calculation we get these **vectors [0.1554 , 0.4223, 0.4223]**

attention weight of “sat” = **Softmax([1 1 2])** , after this calculation we get these **vectors [0.2119 , 0.2119, 0.5762]**

Sixth step is to sum of all weight values. We multiply the attention wights by corresponding value vectors for the token “ The” , now we use Value vectors which are the intial fixed embedding vectors.

output(The) = 0.4223 . Vthe + 0.1554 . Vcat + 0.4223 . Vsat
= 0.4223 . [1 0 1 0] + 0.1554 . [0 1 0 1] + 0.4223 . [1 1 1 1]
= [0.4223, 0, 0.4223, 0] + [0, 0.1554, 0, 0.1554] + [0.4223, 0.4223, 0.4223, 0.4223]
= [1.2669, 0.9999, 1.2669, 0.9999]

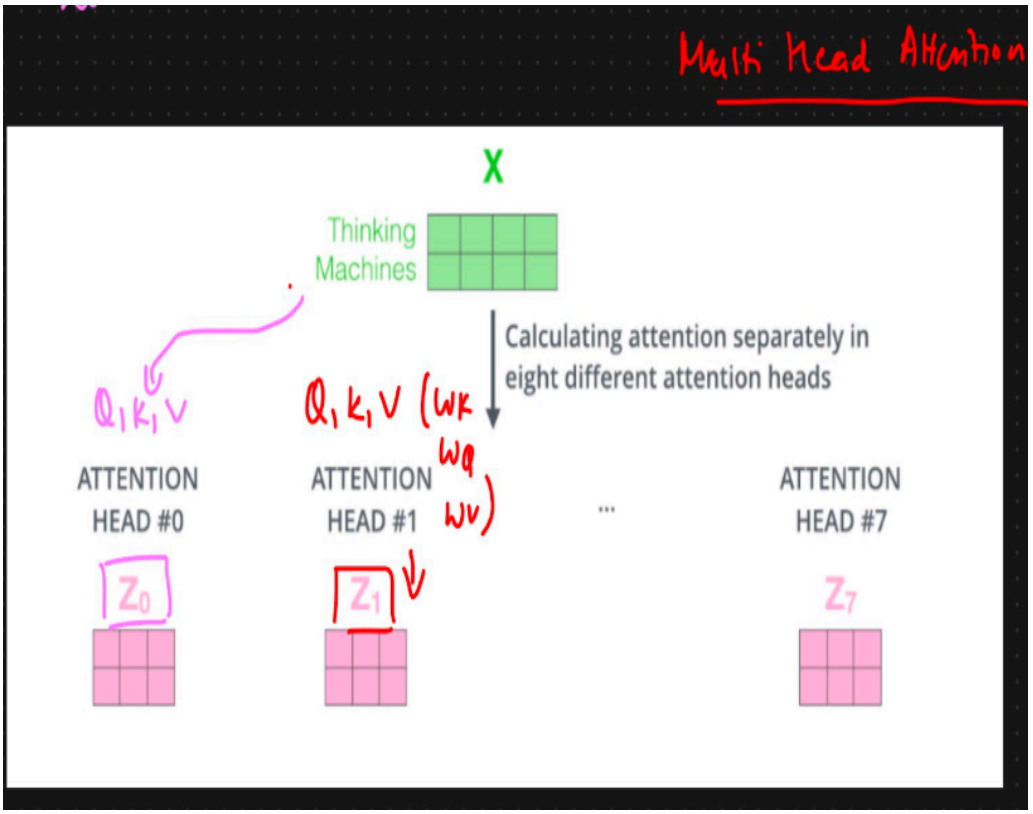
In other words, first we get fixed vectors [1 0 1 0] then use **self attention** then got **Contextual embedding** [1.2669, 0.9999, 1.2669, 0.9999]

To sump with the steps of Self attention, 1) calculate Q,K,V and initialize Weights of Q,K,V and weights will be trained **2) calculate attention score, 3)** Scaled values, **4)** passs it to softmax activation function, **5)** Calculat weighted sum of values.

Multi-head attention

Let’s consider the Contextual vector as an Attention head. As we know that Contextual vector also has other words dependency. For example, cat word may have a dependency of Sat word. Likewise, we create another attention head where it will capture some more words importance for that it will initialize weights and then it will create another Q,K,V then we get different contextual vector which is known as Z1 and it is compared to first head which known for Z0, similar we can create more Attention head so that it will expand model ability to focus on different position of

tokens. As per below diagram



Feed Forward Neural Networks

When we get all **attention head** then we **concatenate** and do **dot product** with W0 which are **new weights** that we will initialize to **calculate** Feed Forward N/N and after this we go ahead and calculate final output **Z.**

Positional Encoding

Positional Encoding is also known as **representing the order of sequence.**

The most important advantage of a transformer is that word tokens can be processed parallelly, while there is a **drawback** which is it lacks the sequential structure of the words, the transformer does not mean that x1 word comes first before x2.

Lets take an example, **1) Lion kills tiger, 2) Tiger kills lion,** now these both have different meanings, however, words are the same so when we give these words to the self-attention layer, it gives sthe ame vectors as outputs. So this is what missing in the self-attention layer and in order to prevent this, we use **Positional Encoding.**

We create positional encoding vector along with embedding layer which we pass into a self-attention layer and these positional encoding vector says about position of that words.

Also there is a question that why need to create positional encoding vector separately , if possible ,we can add positional in embedding layers. **It isn't possible as the reason is that** it may works for small sentences, while there will be so many words in a long sentence such as book, novel, and so on that time it will be difficult thats what we create positional encoding vector separately and add it with embedding layer before sending it into the self-attention layer.

So how do we create **Positional Encoding vector?** There are two types of Positional encoding 1) **sinusoidal positional encoding,** 2) **Learned Positional Encoding**

1) **Sinusoidal positional encoding -** It use sin and cosine functions of different frequencies to create positional encodings. All the values of vectors will be replaced by a formula
   P.E(pos, 2i) = sin(pos/10000^2i/d), **likewise there is one more formula,**
   P.E(pos, 2i+1) = cos(pos/10000^2i/d),
   Where pos = Position , i = is the dimension, d = is the dimensionality of Embeddings.

Let's understand it with an example, e.g The cat sat, now imagine all vectors for all words are below

The - [0.1 0.2 0.3 0.4]
cat - [0.5 0.6 0.7 0.8]
sat - [0.9 1.0 1.1 1.2]

P.E(pos, 2i) = sin(pos/10000^2i/d) P.E(pos, 2i+1) = cos(pos/10000^2i/d)

For our example, d = 4 
For position pos = 0

If we want to find out a positional encoding for 4 dimension vector , so calculate in combination of two like 0 and 1, then 2 and 3, So 0 will be calculated by sin formula and 1 will be calculated by cos formula, likewise with 2 and 3.

Why dont we use only Sin , instead of using both , so the reason is that all Values range from -1 to 1 and we may get same values and we may miss order of the elements. That's why we combine with another cosine formula

For the first word we represent into 4
Positon encoding.so we iterate through i

PE(0,0) = sin(0 / 10000 ^2(0)/4)
= sin(0) = 0

PE(0,1) = cos(0/ 10000 ^2(0)/4)
= cos(0) = 1

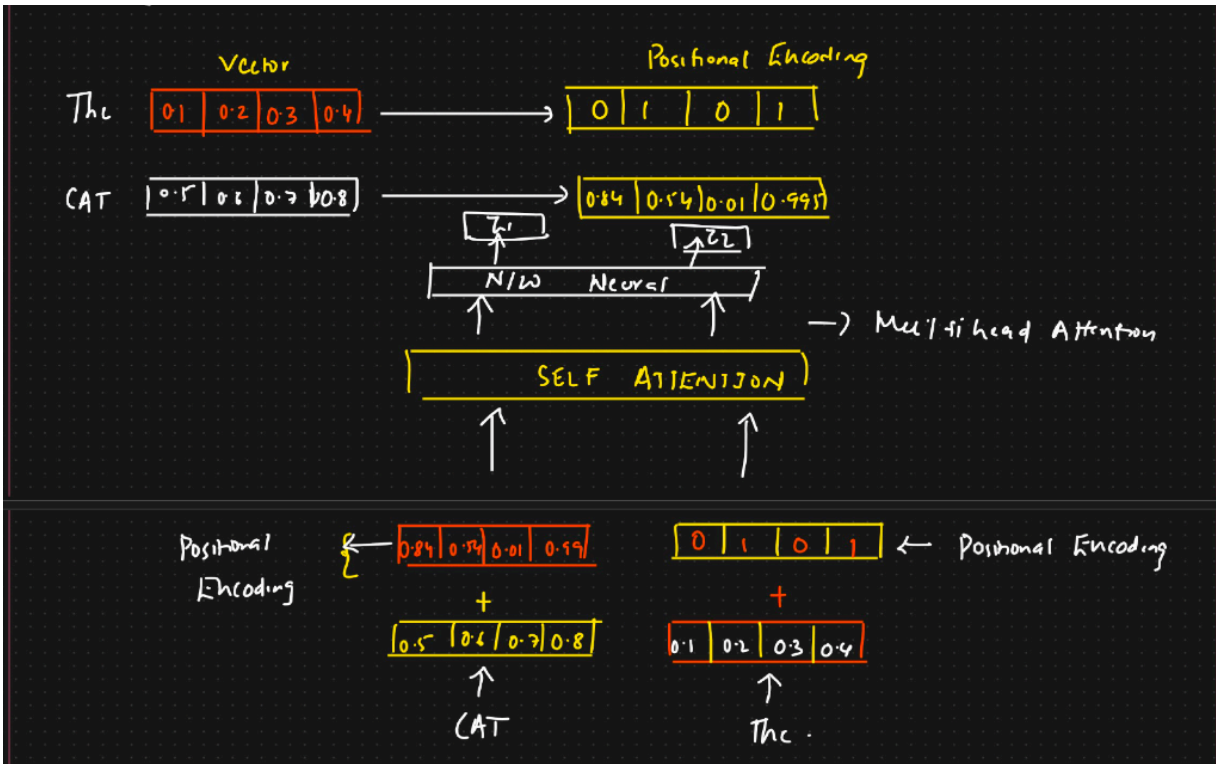
PE(0,2) = sin(0/10000^2(1)/4)
= sin(0) = 0

PE(0,3) = cos(0/ 10000^2(2)/4)
= cos(0) = 1

Now, what positional encoding we are getting P.E = [0 1 0 1] , This is for First word only

Now we find positional encoding for second word.

For Pos =1 ,
PE(1,0) = sin (1/10000^2(0) / 4) = sin(1) = 0.8415
PE(1,1) = cos(1/ 10000^ 2(1)/4) ~= 0.5403
PE(1,2) = sin (1/10000^2(2) / 4) ~= 0.01
PE(1,3) = cos(1/ 10000^ 2(3)/4) ~= 0.999



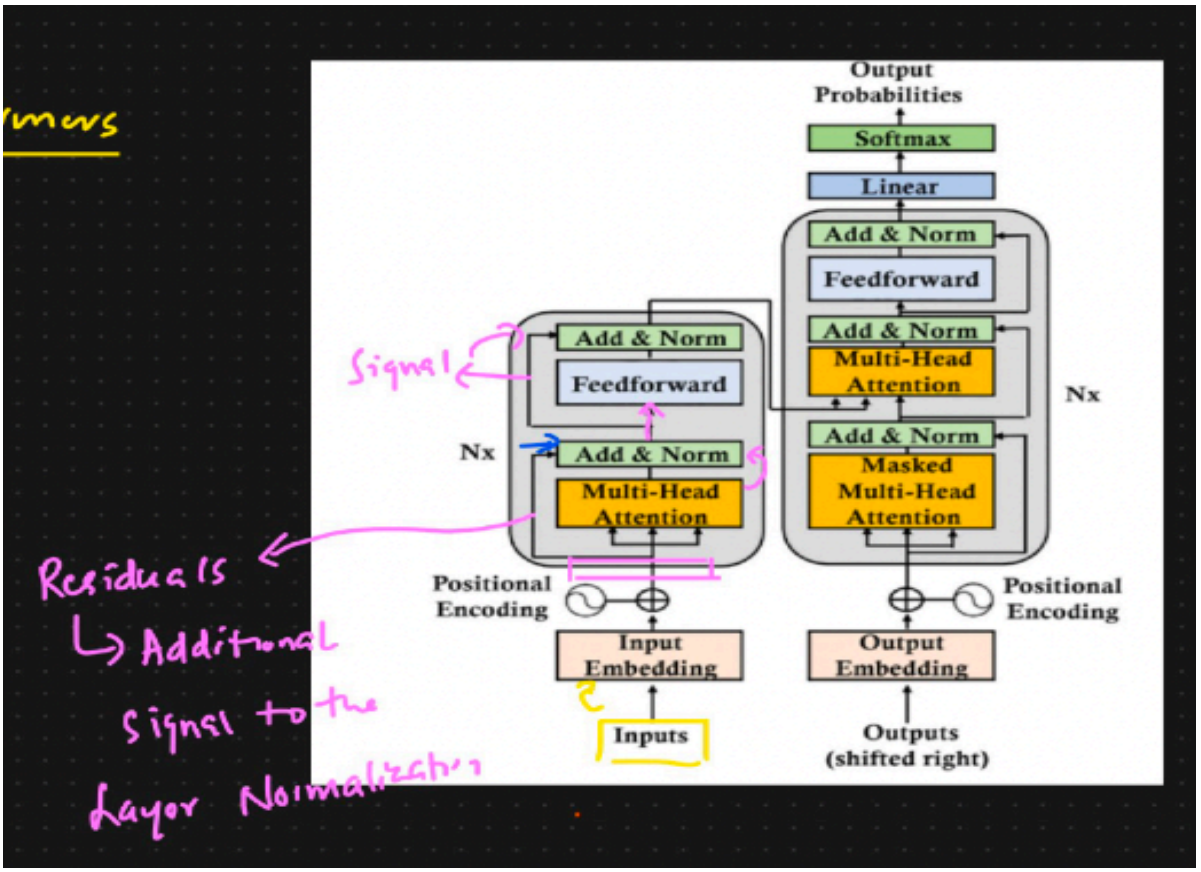
In conclusion, as per the above image , we sum up the values of vector and positional encoding and send it to self attention layer then multihead attention apply after that output of multi-head will pass to neural network and we get contextual vectors.

2) Learned Positional Encoding

Positional encoding are learned during training, it means we create Positional encoding matrix and needs to get updated during back-propagation.

Layer Normalization

Where do we add normalization? So when we calculate multi-head attention after the self-attention layer, then we add and normalize it



Now there is one more term which is **Residuals,** iwhen we add positional encoding in Input embedding vectors then it pass to ADD & normlize this is known as residuals. It provides **additional signals to the layer normalization.**

There are two type of normalization with the respect to specific deep learning, **1) Batch Normalization, 2) layer Normalization**

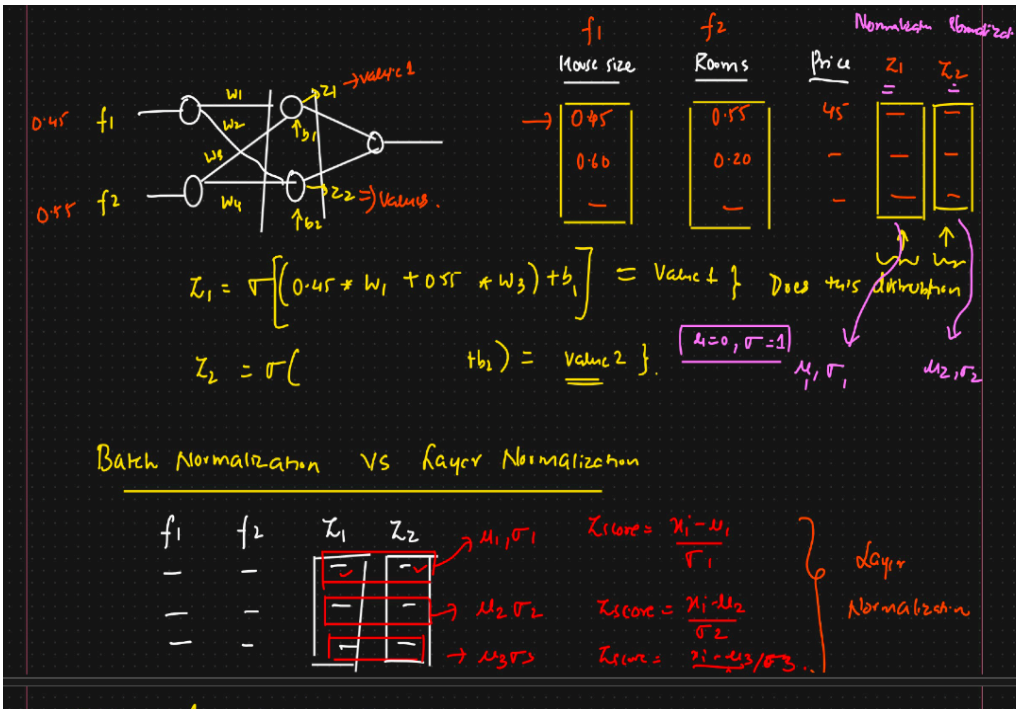
Let's take an example of ANN ,now we have two features home and no.of rooms, and output is price.
Now, before we provide it to model we perform normalization, which is **standard scaling**

$$Z_{score} = x_i - \mu \,/\, \text{std. Deviation}$$

We get all values in such a way that our μ will be 0, and std.deviation will be 1, likewise we apply min max scaler in deep learning when our dataset have images.

Advantages:

- **Improved Training stability** , what does it mean? It means our data might follow different distribution, however, when we use **Normalization** ,all information will lie near to zero where μ = 0 and std. Deviation is 1. Because of this, we won't face problem like vanishing and exploding gradient problem.
- **Faster convergence**



-
-

As per above image we find Z1 and Z2 then we **normalize it by** finding μ1 and σ1 in Z1 column, likewise we find μ2 and σ2 in Z2 colum then for every value we apply Zscore. By normalizing for Z1 column the Z2 column, this called **Batch Normalization.**

Instead of taking entire Z1 and Z2 column, we take row wise like shown in image ,take layer by layer then we find Zscore which is known as **Layer Normalization.**

Now there will be many hidden layers in ANN and it would be a great idea to use batch normalization after every hidden layer.

There are two important parameters such as **γ(gama),β(beta)** before learning it , first of all we learn what is the use of **layer normalization.**
These gama and beta also known as **scale and shift parameters**

Let's take z1 and z2 column where we have some zeros there, so usually we are performing Batch normalization, normalize it by calculating μ and σ , then apply Zscore on each values so those 0s value will also change.

In ANN, we definitely apply **Batch normalization,** while we use **layer normalization** in Transformer as every words are matter in NLP cases.

What if ? our z1 and z2 follows good distribution and we don't wanna change it then we use two parameters gamma and beta and specific learn it when we training neural network.

So, $Z_1 = \sigma[W_1^T x + b1]$, after this we assign learnable parameter

$Y =$ gama $[Z_1 - \mu_1 / \sigma_1] +$ beta , μ and σ which we are going to calculate ,

If our distributing isn't important and don't want normalization then we use these learnable parameters to tell that dont normalize in this scenario

Now **get back to transformer image,** the step was add & Nor, so it means the output we combine after multi head attention consider it as **X** and the output of positional encoding + embedding vectors consider it as **X'** , so this X is going to add with X' then we apply Layer normalization.